


# Habari STOMP Client for ActiveMQ

***Getting started***

Version 10.3-RC2

# Table of contents


Legal notices . . . 1

**Part I: Getting started** . . . **2**

1. Installation . . . 3
    1.1. Requirements . . . 3
    1.2. Installation steps . . . 3
2. Directory structure . . . 5
3. Breaking changes and deprecation warnings in version 10.3 . . . 6
    3.1. Breaking changes . . . 6
    3.2. Deprecation warnings . . . 6
4. Tutorials . . . 9
    4.1. Quick start tutorial . . . 9

**Part II: Programming guide** . . . **13**

5. Communication adapters . . . 14
    5.1. Introduction . . . 14
6. The programming model . . . 17
    6.1. Classic and modern (simplified) API . . . 17
7. Connection factory . . . 18
    7.1. Overview . . . 18
    7.2. Creation and configuration . . . 18
    7.3. Connection URL parameters . . . 20
    7.4. Receipt support . . . 20
8. Connections and sessions . . . 24
    8.1. Connections use STOMP 1.2 by default . . . 24
    8.2. Step-by-step example . . . 24
    8.3. Session types overview . . . 28
    8.4. Transacted sessions . . . 29
9. Destinations . . . 32
    9.1. Introduction . . . 32
    9.2. Create a new destination . . . 32
10. Producer and consumer . . . 34
    10.1. Message producer . . . 34
    10.2. Message consumer . . . 35
    10.3. Synchronous receive . . . 35
11. Asynchronous receive (IMessageListener) . . . 37
    11.1. Overview . . . 37
    11.2. Using a listener . . . 37
    11.3. How delivery works . . . 38

11.4. Rules . . . 39
11.5. Sending messages . . . 40
11.6. Acknowledgement and transactions . . . 41
11.7. Connection loss . . . 42
11.8. Known issue: `Close` can hang . . . 43
11.9. When not to use it . . . 43
11.10. Changes in version 10.3 . . . 44
11.11. Future of the define . . . 45
12. Durable subscriptions . . . 46
12.1. Description . . . 46
13. Temporary queues . . . 47
13.1. Introduction . . . 47
14. Message options . . . 48
14.1. Standard properties . . . 48
14.2. Reserved property names . . . 49
14.3. Selectors . . . 50
15. Map messages . . . 51
15.1. Introduction . . . 51
16. Object messages . . . 56
16.1. Introduction . . . 56
17. Simplified API . . . 59
17.1. Interface types . . . 59
17.2. IMQContext interface . . . 59
17.3. IMQProducer interface . . . 59
17.4. IMQConsumer interface . . . 60
17.5. Source code example . . . 60
**Part III: Advanced topics . . . 61**
18. STOMP 1.2 . . . 62
18.1. Connection configuration . . . 62
18.2. Sending heartbeat signals . . . 63
18.3. Checking for incoming heartbeats . . . 63
18.4. Reading server-side heartbeats . . . 64
19. Optional STOMP 1.2 features not implemented . . . 65
19.1. Text encodings other than UTF-8 . . . 65
20. SSL/TLS support . . . 67
20.1. SSL communication adapter classes . . . 67
20.2. Indy SSL demo . . . 67
20.3. Support . . . 69
21. Logging with SLF4P . . . 70

21.1. Introduction . . . 70
21.2. IDE and project configuration . . . 70
21.3. LoggingHelper unit . . . 70
22. Conditional symbols . . . 72
22.1. Experimental or optional features . . . 72
23. Useful units . . . 74
23.1. BTStreamHelper unit . . . 74
23.2. BTJavaPlatform unit . . . 74
24. Example applications . . . 75
24.1. Overview . . . 75
24.2. ConsumerTool . . . 76
24.3. ProducerTool . . . 78
24.4. Delphi Chat . . . 79
24.5. Performance test . . . 80
24.6. PingBroker . . . 81
24.7. Throughput test . . . 82
25. Unit tests . . . 85
25.1. Introduction . . . 85
25.2. Test project configuration . . . 85
25.3. Test units . . . 85
25.4. Test execution . . . 86
25.5. STOMP 1.2 . . . 86
**Part IV: Reference . . . 87**
26. Broker differences . . . 88
26.1. Defaults . . . 88
26.2. Features and limitations . . . 88
26.3. Protocol details . . . 91
27. Library limitations . . . 93
27.1. MessageConsumer . . . 93
27.2. Message properties . . . 93
27.3. Multithreading . . . 94
27.4. Free Pascal specific restrictions . . . 94
27.5. Broker-specific limitations . . . 94
28. Frequently asked questions . . . 95
28.1. Technical questions . . . 95
29. Online resources . . . 99
29.1. Third-party libraries . . . 99
29.2. Specifications . . . 100
29.3. Online articles . . . 100

29.4. Online videos . . . 100
30. Support . . . 102
30.1. Bug reports and support inquiries . . . 102
**Part V: ActiveMQ** . . . **103**
31. Broker-specific notes . . . 104
31.1. Enable the STOMP transport . . . 104
31.2. Subscription options . . . 104
31.3. Selectors . . . 104
31.4. Object messages . . . 106
31.5. Broker-specific demos . . . 108
32. Connection troubleshooting . . . 112
32.1. Performance demo . . . 112
Index . . . 114

# Legal notices

## 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. ActiveMQ, Apache ActiveMQ, the Apache logo, and the Apache ActiveMQ project logo are trademarks of The Apache Software Foundation. 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.*

**NOTE** For broker-specific notes, see Chapter 31, *Broker-specific notes*.

# Part I: Getting started

# Chapter 1. Installation

## 1.1. Requirements

### 1.1.1. Development environment

The library may be used with

- **Embarcadero Delphi** 2009 Update 4 or higher
- **Free Pascal** 3.2.2 or higher, on Windows and Linux (x86_64). On Linux, a minimal client was compiled and run with all three TCP/IP libraries (Ubuntu 24.04, Lazarus 3.0); see Section 27.4, “Free Pascal specific restrictions”.

### 1.1.2. 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** (tested with the `2.4-stable` tag, version 2.4.13372, and with 2.4.17677)

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

### 1.1.3. 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.

## 1.2. Installation steps

The installer application will guide you through the installation process.

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

C:\Users\Public\Documents\Habarisoft\habari-activemq-10.3-RC2

[1] This is the primary adapter class being used for development and testing of the library.

[2] See chapter "Online resources" for download addresses.

# Chapter 2. 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
   \- HabariActiveMQGettingStarted.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
```

# Chapter 3. Breaking changes and deprecation warnings in version 10.3

## 3.1. Breaking changes

There are no breaking changes for applications that do not define `HABARI_BROKEN_MESSAGELISTENER`.

### 3.1.1. IMQConsumer interface (only with HABARI_BROKEN_MESSAGELISTENER)

With `HABARI_BROKEN_MESSAGELISTENER` defined, the `IMQConsumer` interface of the simplified API has two new methods, `GetMessageListener` and `SetMessageListener`, and the `MessageListener` property. Consumers created with `IMQContext.CreateConsumer` can now receive messages asynchronously, as `IMessageConsumer` can.

This affects only classes that implement `IMQConsumer` themselves, for example test doubles: add the two methods. Code that only uses `IMQConsumer` is not affected.

See: Chapter 11, *Asynchronous receive (IMessageListener)*

## 3.2. Deprecation warnings

### 3.2.1. HABARI_X_MAXLINELENGTH and HABARI_ENABLE_FAILOVER_PROTOCOL

Both conditional defines have been deprecated since version 10.2. The compiler reports it when they are set. **They will be removed in the next release.** A project that still defines them will then fail to compile, with a message naming the replacement. They are not simply dropped, because your setting would then be lost without notice, and the application would behave differently at run time.

Remove both defines from your project options now. The replacements have been available since version 10.2 and need no define. Set them on the connection before you call `Start`:

| Define | What it does | Replacement |
|---|---|---|
| `HABARI_X_MAXLINELENGTH` | Removes the line length limit of Internet Direct (Indy) for very large header lines (the define sets it to `MaxInt`) | `(Connection as IMaxLineLength).MaxLineLength := 1024 * 1024;` (unit `BTCommInterfaces`): choose a limit that fits your longest header line instead of removing the limit. Indy adapter only; the other adapters have no such limit. |
| `HABARI_ENABLE_FAILOVER_PROTOCOL` | Allows `failover:` connection URLs | `(Connection as IFailoverProtocol).EnableFailoverProtocol := True;` (unit `BTMgmtInterfaces`) |

Code example

```
Connection := Factory.CreateConnection;
// was HABARI_X_MAXLINELENGTH; the Indy default is 16,384 bytes (16 KB)
(Connection as IMaxLineLength).MaxLineLength := 1024 * 1024; // 1 MB
// was HABARI_ENABLE_FAILOVER_PROTOCOL
(Connection as IFailoverProtocol).EnableFailoverProtocol := True;
Connection.Start;
```

Do not simply use `MaxInt`, which is what the define set. A bounded limit still accepts every realistic header line, but stops a faulty or hostile peer from making the client read an endless line into memory.

With the simplified API, set them on the context, right after `CreateContext` and before the first `CreateConsumer`, `CreateProducer`, `CreateBrowser` or `Start`:

Code example

```
Context := Factory.CreateContext;
(Context as IFailoverProtocol).EnableFailoverProtocol := True;
Consumer := Context.CreateConsumer(Context.CreateQueue('orders'));
```

See: Section 5.1.4, “Maximum line length for Internet Direct (Indy)” and Section 8.2.4, “Failover connections”

## 3.2.2. HABARI_BROKEN_MESSAGELISTENER

The conditional define `HABARI_BROKEN_MESSAGELISTENER` will be removed in a future release. The code for asynchronous receive will then always be compiled in. The define will not be renamed.

The word “BROKEN” in the name is historical. `IMessageListener` is not deprecated: it is supported within the limits described in its chapter (one session per connection, no receipts, Indy adapter only).

When the define is removed, `IMessageConsumer` and `IMQConsumer` will always include `GetMessageListener` and `SetMessageListener`. This matters only if you have classes that implement one of these interfaces yourself.

See: Chapter 11, *Asynchronous receive (IMessageListener)*

### 3.2.3. HABARI_DEPRECATED_MESSAGELISTENER

The define `HABARI_DEPRECATED_MESSAGELISTENER` was renamed to `HABARI_BROKEN_MESSAGELISTENER` in version 10.2. A project that still defines the old name does not compile (“HABARI_DEPRECATED_MESSAGELISTENER is no longer usable”). Define `HABARI_BROKEN_MESSAGELISTENER` instead.

# Chapter 4. Tutorials

## 4.1. Quick start tutorial

This tutorial is a quick introduction to Habari STOMP Client for ActiveMQ: it walks you through the creation of a "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.

### 4.1.1. 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 ActiveMQ 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.

### 4.1.2. Adding code to the project

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

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

Code example

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

3. Compile and save the project.
4. Switch to Design view (F12), go to the Tool palette (Ctrl+Alt+P) and select TButton, add a button to the form.
5. Double-click the new button to jump to the Button1Click handler.
6. Add the following code to send the message:

Code example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Context: IMQContext;
begin
  Factory := TBTConnectionFactory.Create;
  Context := Factory.CreateContext;
  try
    Context.CreateProducer.Send(Context.CreateQueue('HelloMQ'), 'Hello world!');
  finally
    Context.Close;
  end;
end;
```

7. Add a second button and double-click it to jump to the Button2Click handler.
8. Add the following code to receive and display the message:

Code example

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Context: IMQContext;
  Consumer: IMQConsumer;
  Msg: IMessage;
begin
  Factory := TBTConnectionFactory.Create;
  Context := Factory.CreateContext;
  try
    Consumer := Context.CreateConsumer(Context.CreateQueue('HelloMQ'));
    Msg := Consumer.Receive(1000);

    if Assigned(Msg) then
      ShowMessage((Msg as ITextMessage).Text)
    else
      ShowMessage('Error: no message received');
  finally
    Context.Close;
  end;
end;
```

9. Compile and save the project.

The example uses the simplified API (see Chapter 17, *Simplified API*). `TBTConnectionFactory.Create` without parameters connects to the broker on the local computer, with the default port and user credentials.

### 4.1.3. Run the demo

- Launch the message broker.
- Start the application.
- Click Button1 to send the message to the queue.
- Click Button2 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 factory is created with the IP address or the host name of the message broker, for example `TBTConnectionFactory.Create('stomp://brokerhost')`.

### 4.1.4. 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.
```

### 4.1.5. Tutorial source code

The tutorial source code is included in the folder `demo\tutorial1`.

# Part II: Programming guide

# Chapter 5. Communication adapters

## 5.1. Introduction

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

### 5.1.1. 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.

### 5.1.2. 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 you 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.

## 5.1.3. Available communication adapters

The library includes adapter classes for TCP/IP libraries:

Table 1. Communication adapters

| Adapter class | Unit to add |
|---|---|
| **TBTCommAdapterIndy** (recommended)[1] | BTCommAdapterIndy |
| TBTCommAdapterSynapse | BTCommAdapterSynapse |
| TBTCommAdaptermORMot2 | BTCommAdaptermORMot2 |

## 5.1.4. 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` and every `IMQContext`.

Code example

```
uses
  BTCommAdapterIndy, BTConnectionFactory, BTMQInterfaces,
  BTCommInterfaces, SysUtils;
...
Connection := Factory.CreateConnection;

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

Connection.Start;
```

With the simplified API, set it on the context, right after `CreateContext` and before the first `CreateConsumer`, `CreateProducer`, `CreateBrowser` or `Start`: `(Context as IMaxLineLength).MaxLineLength := 1024 * 1024;`

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 mORMot 2 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 and will be removed in the next release. It still works, but the compiler now emits a warning; new code should use the `MaxLineLength` property instead.

[1] This is the primary adapter class being used for development and testing of the library.

# Chapter 6. The programming model

## 6.1. Classic and modern (simplified) API

Applications may choose between the classic API, which is based on message producers / message consumers, sessions, connections, and connection factories, and the simplified API, which moves methods of ISession and IConnection into a single IMQContext interface.

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

# Chapter 7. Connection factory

## 7.1. Overview

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

## 7.2. 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

```
procedure 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;
```

## 7.3. Connection URL parameters

### 7.3.1. Heartbeating support

STOMP 1.1 introduced heartbeating. Its configuration is covered in Chapter 18, *STOMP 1.2*.

## 7.4. 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."[1][2][3]

With Habari STOMP Client for ActiveMQ, 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.

A persistent message (`dmPersistent`) always requests a receipt for its SEND frame, with or without `send.receipt=true`. Everything in this section applies to it as well.

**NOTE** Do not use receipts on a connection with a listener, and do not send persistent messages on it. See rule 2 in Section 11.4, “Rules”.

**WARNING** **Known issue:** after sending, the client reads exactly one frame and expects it to be the RECEIPT. If the broker first sends a MESSAGE for a consumer on the same connection, the operation raises "No RECEIPT received", and that message is lost. Earlier versions behave the same, and a fix is planned. Request receipts, and send persistent messages, only on a connection or context that does not also receive messages, for example a separate one used only for sending.

### 7.4.1. 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?disconnect.receipt=true |

### 7.4.2. 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.

### 7.4.3. 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.

### 7.4.4. 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").

### 7.4.5. 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").

[1] https://stomp.github.io/stomp-specification-1.0.html

[2] https://stomp.github.io/stomp-specification-1.1.html#Header_receipt

[3] https://stomp.github.io/stomp-specification-1.2.html#Header_receipt

# Chapter 8. Connections and sessions

## 8.1. 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.

## 8.2. Step-by-step example

### 8.2.1. Overview

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

### 8.2.2. 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;
...
```

### 8.2.3. 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 section "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.

## 8.2.4. 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 every `IMQContext` 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;
```

With the simplified API, set it on the context, right after `CreateContext` and before the first `CreateConsumer`, `CreateProducer`, `CreateBrowser` or `Start`: `(Context as IFailoverProtocol).EnableFailoverProtocol := True;`

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 and will be removed in the next release. If it is still defined, every connection starts with `EnableFailoverProtocol` set to `True` and the compiler emits a warning. New code should set the property and remove the define from the project options.

### 8.2.5. Connection URL parameters

Connection URL parameters are documented in chapter "Connection factory" and in Chapter 18, *STOMP 1.2*.

### 8.2.6. 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 acknowledgement 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);
```

### 8.2.7. 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'));
```

### 8.2.8. 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.

## 8.3. Session types overview

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

Table 2. Session creation parameters

| Parameters | Client MUST acknowledge message receipt[1] | 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 |

① – The acknowledgement is part of the transaction (see Section 8.4.5, “Transacted message

acknowledgement").

## 8.4. 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.

Starting the connection again does not end the current transaction: neither a second `Start`, nor `Stop` followed by `Start`, nor `CreateProducer` and `CreateConsumer` on a transacted `IMQContext`, which start its connection. Up to version 10.3-RC1, each of these began a new transaction, and the work done before was never committed.

### 8.4.1. 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.

### 8.4.2. 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'));
```

## 8.4.3. 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

```
try
  // send messages ...

  // commit all messages
  Session.Commit;
except
  Session.Rollback;
  raise;
end;
```

**NOTE** Committing a transaction automatically starts a new transaction.

## 8.4.4. 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

```
try
  // send messages ...
  Session.Commit;
except
  // error: none of the messages become visible
  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.

### 8.4.5. Transacted message acknowledgement

In a transacted session, acknowledging a received message is part of the transaction. `Rollback`, or closing the connection without `Commit`, makes the message available again.

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.

[1] https://stomp.github.io/stomp-specification-1.2.html#SUBSCRIBE_ack_Header

# Chapter 9. Destinations

## 9.1. Introduction

The API supports two models:[1]

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 the 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 knows about the 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 topic 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.

## 9.2. Create a new destination

### 9.2.1. Queues

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

Code example

```
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.)

### 9.2.2. 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.)

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

# Chapter 10. Producer and consumer

## 10.1. 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.

### 10.1.1. Persistent messages

The delivery mode for outgoing messages may be set to persistent in one of two ways.

The client waits for the broker's receipt of every persistent message, so a persistent send takes a round trip to the broker. Send persistent messages on a connection or context that does not also receive messages; see [receipt-known-issue] and, for message listeners, rule 2 in Section 11.4, "Rules".

#### 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);
```

## 10.2. 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);
```

### 10.2.1. Message selector

A message consumer can be created with a **message selector**[1].

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

## 10.3. 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;
```

### 10.3.1. 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).

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

**ReceiveNoWait** The ReceiveNoWait method will return immediately. If no message is available, it will return nil.

[1] The RabbitMQ message broker does not support message selectors.

# Chapter 11. Asynchronous receive (IMessageListener)

## 11.1. Overview

Habari offers two ways to receive messages:

- **Synchronous:** your code calls `Consumer.Receive(Timeout)` and gets the next message or `nil`. This is the recommended way for most applications.
- **Asynchronous:** you attach an `IMessageListener` to a consumer, and a background thread calls its `OnMessage` method for every message that arrives.

The asynchronous option is compiled in only when the conditional symbol `HABARI_BROKEN_MESSAGELISTENER` is defined in your project options, and it works only with the Indy communication adapter. Like all conditional symbols, it is an optional feature, not covered by the free basic support plan.

The word “BROKEN” in the define is historical. The listener works well for a simple setup: one connection, one session, one or more consumers, and handlers that finish quickly. Its limits are listed in Section 11.4, “Rules”. Stay within them, because outside them it does not just slow down: it can corrupt the conversation with the broker. Later releases are planned to lift some of these limits (see Section 11.11, “Future of the define”).

## 11.2. Using a listener

1. Add `HABARI_BROKEN_MESSAGELISTENER` to the conditional defines of your project, and rebuild.
2. Use the Indy communication adapter (add `BTCommAdapterIndy` to a `uses` clause).
3. Implement `IMessageListener` and assign it to the `MessageListener` property of a consumer. This works on `IMQConsumer` (simplified API) and on `IMessageConsumer` (classic API).

Code example

```
type
  TOrderListener = class(TInterfacedObject, IMessageListener)
  public
    procedure OnMessage(const Message: IMessage);
  end;

procedure TOrderListener.OnMessage(const Message: IMessage);
begin
  try
    // keep this short: put the message on your own work queue and return
  except
    on E: Exception do
      LogError(E.Message); // your own logging: nothing else reports it
  end;
end;

...
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Context := Factory.CreateContext;
  Consumer := Context.CreateConsumer(Context.CreateQueue('orders'));
  Consumer.MessageListener := TOrderListener.Create;
```

With the classic API, set the property in the same way on an `IMessageConsumer` created by `Session.CreateConsumer`.

Delivery starts as soon as the listener is set, because the connection of a context is already started. Messages that arrived before are not lost: the listener receives them. Setting the listener to `nil` does not stop delivery; the consumer's messages are then dropped. To stop receiving, close the consumer.

The consumer keeps the listener alive through reference counting. Keep a reference to the context (or connection) and the consumer for as long as messages should be delivered.

The demo `demo/delphichat-listener` (HabariChatRooms) shows several chat rooms on one `IMQContext`, each with its own consumer and listener.

## 11.3. How delivery works

- **One thread per connection.** All subscriptions on a connection share one socket and one receive thread. The thread reads one message, calls your `OnMessage`, waits for it to return, then reads the next one.
- **Handlers run one at a time.** A slow handler for one destination holds up every other destination on the same connection. There is no fairness between subscriptions: messages are handled in the

order the broker sends them.

- **Which thread runs `OnMessage` depends on the application type:**

| | GUI application | Console application |
|---|---|---|
| `OnMessage` runs on | the **main thread**, handed over with `Synchronize` | the **receive thread** of the connection |
| Updating forms and controls in the handler | allowed directly | not allowed: use `TThread.Queue` or `Synchronize` |
| When the main thread is busy | messages wait until it is free | delivery is not affected |
| Sending and acknowledging from the handler and from your other code | all on the main thread, so they cannot interfere (see Section 11.5, “Sending messages”) | on different threads: protect them with a lock |

In a GUI application this has consequences you should be aware of:

- **A slow handler freezes the user interface,** because it blocks the main thread. And the other way round: while the main thread is busy (a long click handler, a modal dialog doing work), no message is delivered.
- `Synchronize` **waits for the main thread.** If the main thread itself waits for something the receive thread must deliver first, the application hangs. Do not block the main thread waiting for a message.
- **Everything that writes to the connection is safe only as long as it stays on the main thread.** If you move sending to a worker thread, the rules for other threads apply (see Section 11.5, “Sending messages”).
- The choice is made automatically from the application type and cannot be changed through the public API in version 10.3.

## 11.4. Rules

Breaking one of these rules is a correctness problem, not a performance problem.

1. **One session per connection.** A context has exactly one session, so it keeps this rule by itself. With several sessions on a connection that has listeners, messages can be lost or raise “Found no

matching consumer" errors.

2. **No receipts.** Do not use the `subscribe.receipt`, `unsubscribe.receipt` or `send.receipt` connection URL parameters on a connection with a listener, and do not send persistent messages (`dmPersistent`) on it: a persistent message always waits for a receipt, even without these parameters. Waiting for a receipt reads the socket on your thread while the receive thread reads it too. Inside `OnMessage` it reads the next message instead and calls `OnMessage` again before the first call has returned. Send persistent messages on a separate connection or context.
3. **Indy adapter only.** With the Synapse or mORMot 2 adapter, setting a listener raises `EIllegalStateException`.
4. **One writer at a time.** If your handler acknowledges messages, or sends messages, while another thread sends on the same connection, serialize those calls yourself. Otherwise frames can be interleaved on the wire.
5. **Do not stop consumers while other listeners are live.** You cannot stop one listener alone. Stopping a consumer, its session or the connection stops the receive thread for every listener on the connection. To end one listener, close its consumer.
6. **Catch exceptions in `OnMessage`.** Exceptions raised by a handler are logged only if the library is built with `HABARI_LOGGING`; otherwise they disappear without a trace. With automatic acknowledgement the message is already acknowledged, so it is lost.

## 11.5. Sending messages

The library has no lock on the connection's socket. Two threads writing to the same connection at the same time can mix up their frames on the wire. Writing covers more than `Send`: `Acknowledge` (in client acknowledgement modes), `Commit`/`Rollback`, `SendHeartbeat`, `CreateConsumer`, closing a consumer, and closing the connection all write to the socket. And because of rule 1, sending on the listener's connection means using a producer of the same context (or session).

| Where you send | Safe? |
|---|---|
| On a **separate connection or context** used only for sending | Always. It has its own socket. This is the recommended way. |
| Inside `OnMessage` | Yes, as long as no other thread writes to the same connection at the same time, and the message is not persistent. |
| On the main thread of a GUI application | Yes, for non-persistent messages. Handlers also run on the main thread there, so all writes happen on one thread. |

| Where you send | Safe? |
|---|---|
| On any other thread | Only if you protect **every** write to the connection with one lock of your own. |
| With `send.receipt=true`, or any persistent message (`dmPersistent`) | Never, not even inside `OnMessage` (see rule 2). Use a separate connection or context. |

A separate context for sending needs no coordination with the listener:

Code example

```
SendContext := Factory.CreateContext; // used only for sending
Producer := SendContext.CreateProducer;
Producer.Send(SendContext.CreateQueue('replies'), 'Order accepted');
```

Use the sending context from one thread, or protect it with a lock if several threads send.

## 11.6. Acknowledgement and transactions

A listener consumer uses the acknowledgement mode of its session. Call `Acknowledge`, `Nack`, `Commit` and `Rollback` from inside `OnMessage`: while the listener runs, the session belongs to the thread that calls it. All of these write to the connection, so rule 4 applies.

What happens to a message when `OnMessage` raises an exception depends on the mode:

| Session mode | Message after an exception in `OnMessage` |
|---|---|
| `amAutoAcknowledge` | **Lost.** The broker considered it delivered as soon as it was sent to the client. |
| `amClientIndividual` | Not acknowledged. The broker delivers it again later, typically after the consumer or connection is closed. |
| `amClientAcknowledge` | As above, **but** acknowledging a later message also acknowledges all earlier messages of the same consumer, including the failed one. |
| Transacted | The transaction is **not** rolled back automatically. Call `Rollback` yourself. |

For per-message control, use `amClientIndividual` and acknowledge each message at the end of the handler:

Code example

```
procedure TOrderListener.OnMessage(const Message: IMessage);
begin
  try
    ProcessOrder(Message);
    Message.Acknowledge; // only this message (amClientIndividual)
  except
    on E: Exception do
      LogError(E.Message); // not acknowledged: the broker delivers it again
  end;
end;
```

`Nack` tells the broker that the message was not processed. What the broker does with it is broker-specific: ActiveMQ moves it to a dead-letter queue, and RabbitMQ puts it back in the queue. It is tested with ActiveMQ and RabbitMQ.

**Acknowledging inside a transaction:**

In a transacted session, acknowledging a received message is part of the transaction. `Rollback`, or closing the connection without `Commit`, makes the message available again.

Transactions in general are described in chapter "Connections and sessions", section "Transacted sessions".

## 11.7. Connection loss

If the connection breaks (network failure, broker shutdown, or an ERROR frame from the broker), the receive thread ends **silently**. `OnMessage` is no longer called. No exception reaches your code, no event is raised, and the library does not reconnect. Failover applies only when a connection is started.

Ways to notice:

- **Server heartbeats.** Negotiate heartbeats in both directions, for example `stomp://localhost?connect.heart-beat=10000,5000` (the server sends a heartbeat every 5 seconds, the client promises one every 10 seconds). Then call `CheckHeartbeat` and `SendHeartbeat` on a timer that fires more often than the client interval, for example every 5 seconds; `IHeartbeat` is declared in the unit `BTSessionIntf`. `CheckHeartbeat` raises `EBTStompServerHeartbeatMissing` when nothing has arrived for more than twice the server's heartbeat interval, which also happens when the receive thread has ended. Checking only reads a timestamp, so it is safe from any thread; call `SendHeartbeat` from the thread that sends your messages (rule 4). Heartbeats are not available with OpenMQ.

  The client heartbeats matter for an application that only listens: Artemis closes a connection that

sends nothing within its connection TTL (60 seconds by default), even when the client asked for no heartbeats (`connect.heart-beat=0,5000`). The receive thread then ends, and `CheckHeartbeat` reports the loss.

Code example

```
// on a timer, on the thread that sends
(Connection as IHeartbeat).CheckHeartbeat; // raises if the server went silent
(Connection as IHeartbeat).SendHeartbeat;  // keeps the connection alive
```

- **The next write fails.** Send, `Acknowledge` or `SendHeartbeat` on the broken connection raises an exception. An application that only listens may never make such a call.
- **Your own watchdog**, for example “no message for five minutes” on a destination that is normally busy.

To recover, close the broken connection, ignoring exceptions from `Close`, and create a new connection, session, consumer and listener. With automatic acknowledgement, a message that was being handled when the connection broke may be lost. A future release is planned to report connection loss and handler errors through an exception listener.

## 11.8. Known issue: `Close` can hang

In rare cases, closing a connection with a receive thread does not return. To stop, the receive thread suspends itself. If `Close` asks the thread to end at the same moment, the request can get lost, and `Close` then waits for the thread forever. It was seen once in a full test run of several hundred connections. Earlier versions behave the same, and a fix is planned. Without `HABARI_BROKEN_MESSAGELISTENER` there is no receive thread, so this cannot happen.

## 11.9. When not to use it

If you have several busy destinations, need them isolated from each other, or need more than one session, use synchronous receiving instead:

- **One context (connection) per destination.** Separate contexts have separate sockets, so a backlog on one cannot hold up another.
- **A receive loop on a thread you own**, one per context:

Code example

```
procedure TOrderReceiver.Execute; // a TThread descendant
var
  Context: IMQContext;
  Consumer: IMQConsumer;
  Msg: IMessage;
begin
  Context := Factory.CreateContext; // one context for this thread only
  try
    Consumer := Context.CreateConsumer(Context.CreateQueue('orders'));
    while not Terminated do
    begin
      Msg := Consumer.Receive(1000);
      if Assigned(Msg) then
        HandleMessage(Msg); // keep this short, or queue the work
    end;
  finally
    Context.Close;
  end;
end;
```

- **Limit how much the broker sends at once** with a prefetch option in the destination name, for example `orders?activemq.prefetchSize=10` (ActiveMQ) or `orders?prefetch-count=10` (RabbitMQ).

## 11.10. Changes in version 10.3

- **Artemis:** listeners now receive messages. Up to version 10.2 they were never called.
- **Synapse and mORMot 2 adapters:** setting a listener raises a clear `EIllegalStateException` (“IMessageListener requires the Indy communication adapter; … has no receive thread”). Up to version 10.2 this was an access violation. The consumer can still be used with `Receive`.
- **Simplified API:** `IMQConsumer` has a `MessageListener` property, so consumers created with `IMQContext.CreateConsumer` can receive asynchronously too.
- **Synchronous receive:** with the define set, a session without a listener now receives exactly as without the define. Up to version 10.2, two sessions on one connection raised “Found no matching consumer” in `Receive`, and a received message lost its `subscription` property.
- **Subscription check:** `Receive` checks that a message belongs to the consumer’s own subscription, and it now does so with the define set as well. `message.subscription_check=false` still turns it off.
- **Creating and closing consumers while a listener runs** is now safe, also from inside `OnMessage`. Up to version 10.2 it could end delivery silently with an `EListError` on the receive thread.

- **Receive thread:** it is named “Habari receive thread” in the debugger, and a lock is no longer freed while the thread may still run.
- **Units:** `BTThread` and `BTStompReadThreadIndy` moved from `source/optional` to `source`, so the listener needs no extra search path.
- **Compiler warning:** defining the symbol now shows “IMessageListener works within documented limits (one session, no receipts, Indy only)”.

## 11.11. Future of the define

`HABARI_BROKEN_MESSAGELISTENER` will be **removed** in a future release. The listener code will then always be compiled, after the remaining limitations have been addressed. The define will not be renamed. When it is removed, the `IMessageConsumer` and `IMQConsumer` interfaces will always include `GetMessageListener` and `SetMessageListener`. This matters only if you have classes that implement one of these interfaces yourself. Applications that use the synchronous API are not otherwise affected.

# Chapter 12. Durable subscriptions

## 12.1. 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.[1]

The combination of the client ID and the 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.

### 12.1.1. 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 client ID and durable subscriber name.

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

[1] https://jakarta.ee/specifications/messaging/2.0/apidocs/javax/jms/topicsession

# Chapter 13. Temporary queues

## 13.1. Introduction

Temporary queues and temporary topics are destinations which exist only as long as the connection that created them. The application creates them with the session methods CreateTemporaryQueue and CreateTemporaryTopic, and each one gets a unique name. When the connection is closed, the broker deletes the temporary destination, together with any messages still on it.

Only consumers of the connection that created a temporary destination can receive from it. Any producer which knows the destination can send to it. This makes temporary queues the usual choice for replies in request/reply scenarios: the requester creates a temporary queue, sets it as the JMSReplyTo destination of the request message, and waits for the answer on it. The class TBTQueueRequestor in the unit BTQueueRequestor implements this pattern.[1]

### 13.1.1. Library support

STOMP temporary queues are supported by ActiveMQ, OpenMQ, and RabbitMQ. ActiveMQ and OpenMQ also support temporary topics.

| Broker | STOMP temporary queue | STOMP temporary topic |
|---|---|---|
| **ActiveMQ** | ✓ | ✓ |
| **Artemis** | ✗ | ✗ |
| **OpenMQ** | ✓ | ✓ |
| **RabbitMQ** | ✓ | ✗ |

The Habari STOMP Client for Artemis does not support temporary destinations; CreateTemporaryQueue and CreateTemporaryTopic raise an exception. Use a regular queue for replies instead. The Habari STOMP Client for RabbitMQ supports temporary queues only; CreateTemporaryTopic raises an exception.

### 13.1.2. Resource management

Close the connection as soon as the replies have been received, so that the broker deletes its temporary destinations.

[1] `TBTQueueRequestor` is in `source/optional` of all four products.

# Chapter 14. Message options

## 14.1. 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.[1]

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.

### 14.1.1. 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.[2] |
| Priority | The message priority level.[3] |
| ReplyTo | The Destination object to which a reply to this message should be sent. |

### 14.1.2. 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. |
| MessageID | The message ID which is set by the provider. |
| ReplyTo | The Destination object to which a reply to this message should be sent. |

## 14.2. Reserved property names

Some header names are defined by the STOMP specifications, and by broker-specific extensions of the STOMP protocol. These reserved STOMP header names cannot 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. |

### 14.2.1. 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

### 14.2.2. 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**).

## 14.3. Selectors

Selectors are a way of attaching a filter to a subscription to perform content-based routing. For details on selectors, see the reference on javax.jms.Message[4].

### 14.3.1. 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, 'vehicle=''car'' and color=''blue''');
```

All supported brokers support string type properties and operations in selectors. ActiveMQ also allows integer properties and operations in selectors (see the ActiveMQ notes on selectors[5]).

[1] https://jakarta.ee/specifications/messaging/2.0/apidocs/javax/jms/message

[2] For sending persistent messages please see the documentation for IMessageProducer.

[3] Clients set the Priority not directly on a message, but either on the producer or as a parameter in the Send method.

[4] https://jakarta.ee/specifications/messaging/2.0/apidocs/javax/jms/message

[5] https://activemq.apache.org/components/classic/documentation/selectors

# Chapter 15. Map messages

## 15.1. 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).

### 15.1.1. Usage example

Create a map message and add map entries:

Code example

```
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:

Code example

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

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

Enumerate map entries:

Code example

```
MapKeys := MapMessage.GetMapNames;

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

### 15.1.2. 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;
```

### 15.1.3. 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 an ITextMessage or IBytesMessage.

By default, the library will set this header property to the transformation identifier passed to SetMapMessageTransformer, so setting it yourself is optional.

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);
```

### 15.1.4. 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;
```

## 15.1.5. Ready-made map transformers

The folder `source/optional` contains two map message transformers. They are outside the standard support package (see the readme.txt there), and both need Delphi:

- **TBTMessageTransformerJSONMapDataObjects** encodes the map as JSON; its transformation identifier is `jms-map-json`, the name of ActiveMQ's JSON transformation for map messages. It needs the JsonDataObjects library, which is not included.
- **TBTMessageTransformerXMLMapDocument** (experimental) encodes the map as XML with Delphi's XMLDoc unit.

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

# Chapter 16. 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.”[1]

## 16.1. 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.

### 16.1.1. 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 object messages:

1. 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

2. create an instance of this class and register it on the IConnection instance with the helper procedure **SetObjectMessageTransformer** in unit BTConnection, together with a transformation identifier (see Section 15.1.3, “Transformation identifier”; object messages use it the same way)

| **NOTE** | Only one object message transformer may be active for one connection. |
|---|---|

Code example

```
Connection := Factory.CreateConnection;
try
  SetObjectMessageTransformer(Connection, TMyObjectMessageTransformer.Create,
    'my-object-message');

  Connection.Start;
  Session := Connection.CreateSession(amAutoAcknowledge);
  Queue := Session.CreateQueue('example.objects');
  Producer := Session.CreateProducer(Queue);
  Consumer := Session.CreateConsumer(Queue);

  // send
  ObjectMessage := Session.CreateObjectMessage;
  ObjectMessage.SetObject(MyObject);
  Producer.Send(ObjectMessage);

  // receive
  ObjectMessage := Consumer.Receive(1000) as IObjectMessage;
  if Assigned(ObjectMessage) then
  begin
    ReceivedObject := ObjectMessage.GetObject;
    try
      ... // use the object
    finally
      ReceivedObject.Free;
    end;
  end;
finally
  Connection.Close;
end;
```

The library does not free objects: the application frees the object it has sent, and the object that `GetObject` returns for a received message.

## 16.1.2. Transformer base class

Instead of implementing IMessageTransformer directly, a transformer can descend from **TBTMessageTransformer** (unit BTMessageTransformer in `source/optional`) and override only two

methods: **Encode** turns the object into the message body, **Decode** creates the object from it. The base class calls them for object messages and leaves all other messages unchanged. The units in `source/optional` are outside the standard support package (see the readme.txt there).

See unit **ObjectMessageTransformerTests** for integration / unit tests. Its transformer streams a TComponent descendant as text with ObjectBinaryToText and ObjectTextToBinary.

[1] https://www.oracle.com/technical-resources/articles/java/serializationapi.html

# Chapter 17. Simplified API

## 17.1. Interface types

The simplified API[1] is based on three interfaces which reduce the amount of client code:

- IMQContext
- IMQProducer
- IMQConsumer

## 17.2. IMQContext interface

An IMQContext object encapsulates both the IConnection and the ISession object of the classic API. The connection factory interface contains 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).

## 17.3. IMQProducer interface

An 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 an ITextMessage or IBytesMessage object by providing the text or bytes as a parameter.

Code example

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

## 17.4. IMQConsumer interface

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

With the conditional symbol `HABARI_BROKEN_MESSAGELISTENER` defined, `IMQConsumer` also has a `MessageListener` property, for receiving messages asynchronously. See Chapter 11, *Asynchronous receive (IMessageListener)*.

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

## 17.5. 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;
```

[1] Since version 6.0

# Part III: Advanced topics

# Chapter 18. STOMP 1.2

## 18.1. 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.

| Parameter | Description | Default |
|---|---|---|
| connect.accept-version[1] | Supported STOMP versions in ascending order | Broker-specific, see below |
| connect.host[2] | **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[3] | Heartbeat (outgoing, incoming) | 0,0 |

**Default STOMP version (broker-specific)**[4]

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 version (BTBrokerConsts.SH_DEFAULT_STOMP_VERSION) | | | |
|---|---|---|---|
| ActiveMQ | Artemis | OpenMQ | RabbitMQ |
| 1.2 | 1.2 | 1.2 | 1.2 |

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 and virtual host: localhost
- port: the default port (broker-specific, defined in unit BTBrokerConsts as DEFAULT_STOMP_PORT)
- accept-version: 1.2, the client requests version 1.2 of the protocol
- heart-beat: 1000,0, the client heartbeat interval is 1000 milliseconds, with no server heartbeat signals

### 18.1.1. Specification

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

## 18.2. Sending heartbeat signals

A client can use the **SendHeartbeat** method of the connection object to send a heartbeat 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 heartbeats automatically
- client messages which are sent after the heartbeat interval expires may be lost

## 18.3. Checking for incoming heartbeats

The client stores a timestamp of the last incoming data. If the time which elapsed since this timestamp is greater than two times the heartbeat 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 heartbeats
- the method only checks the time elapsed since the last heartbeat, it does not try to read any data from the connection

## 18.4. 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 heartbeat 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 heartbeat signals from the connection.

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

| **NOTE** | ReceiveHeartbeat raises an exception if there is anything other than a heartbeat on the socket. |
|---|---|

[1] https://stomp.github.io/stomp-specification-1.2.html#protocol_negotiation

[2] https://stomp.github.io/stomp-specification-1.2.html#CONNECT_or_STOMP_Frame

[3] https://stomp.github.io/stomp-specification-1.2.html#Heart-beating

[4] Since version 5.1 (2017.06)

# Chapter 19. Optional STOMP 1.2 features not implemented

The STOMP 1.2 specification marks some features as recommended (SHOULD) or optional (MAY, OPTIONAL). A client that leaves them out still conforms to the specification. This chapter lists the optional features that the Habari Client library does not implement, quotes the specification, and describes what the library does instead.

## 19.1. Text encodings other than UTF-8

The specification lets a sender name the text encoding of the body with a `charset` parameter in the `content-type` header:[1]

> The implied text encoding for MIME types starting with `text/` is UTF-8. If you are using a text based MIME type with a different encoding then you SHOULD append `;charset=<encoding>` to the MIME type. For example, `text/html;charset=utf-16` SHOULD be used if you're sending an HTML body in UTF-16 encoding. The `;charset=<encoding>` SHOULD also get appended to any non `text/` MIME types which can be interpreted as text. A good example of this would be a UTF-8 encoded XML. Its `content-type` SHOULD get set to `application/xml;charset=utf-8`
>
> All STOMP clients and servers MUST support UTF-8 encoding and decoding. Therefore, for maximum interoperability in a heterogeneous computing environment, it is RECOMMENDED that text based content be encoded with UTF-8.
>
> — STOMP 1.2 specification, Header content-type

The library supports UTF-8 only, which the specification requires. It does not support other text encodings.

### 19.1.1. Sending

- The library always encodes the body of text, map and object messages as UTF-8.
- With STOMP 1.1 and 1.2, it sets the `content-type` header to `text/plain`

(`application/octet-stream` for bytes messages). It does not add a `charset` parameter, because UTF-8 is the implied encoding for `text/` types.

- There is no option to choose another encoding.
- An application can set its own `content-type` as a string property (see the code example). The library sends this value unchanged but still encodes the body as UTF-8. A `charset` parameter other than `utf-8` in this value is therefore wrong.

Code example

```
Msg := Session.CreateTextMessage('<html>Hello World!</html>');
Msg.SetStringProperty(SH_CONTENT_TYPE, 'text/html');
Producer.Send(Msg);
```

### 19.1.2. Receiving

When the library creates a text message from an incoming frame, it checks the `content-type` header:

| content-type | Result |
|---|---|
| no parameter, for example `text/plain` or `text/html` | accepted, the body is decoded as UTF-8 |
| a parameter containing `charset=utf-8` (not case-sensitive), for example `text/plain;charset=UTF-8` | accepted, the body is decoded as UTF-8 |
| any other parameter, for example `text/plain;charset=utf-16`, `text/plain;charset=iso-8859-1` or `text/plain;format=flowed` | `Receive` raises EMQException with the message "Unsupported content type *value*" |

**NOTE** The check looks for the exact text `charset=utf-8`. A quoted value (`charset="utf-8"`) or the alias `utf8` raises the exception too.

To exchange text with clients in other languages, have them send UTF-8, as the specification recommends.

[1] https://stomp.github.io/stomp-specification-1.2.html#Header_content-type

# Chapter 20. SSL/TLS support

## 20.1. SSL communication adapter classes

Habari STOMP Client for ActiveMQ 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.

Table 3. Communication adapters with SSL/TLS support

| Adapter class | Unit |
|---|---|
| TBTCommAdapterIndySSL | BTCommAdapterIndySSL |
| TBTCommAdapterSynapseSSL | BTCommAdapterSynapseSSL |

### 20.1.1. 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

### 20.1.2. 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.

## 20.2. 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.
```

## 20.2.1. 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

## 20.2.2. 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 versions is to explicitly set the IOHandler PassThrough := False.

## 20.3. Support

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

# Chapter 21. Logging with SLF4P

## 21.1. 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

## 21.2. IDE and project configuration

In order to compile with SLF4P support,

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

### 21.2.1. Delphi

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

### 21.2.2. Lazarus

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

## 21.3. 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.
```

# Chapter 22. Conditional symbols

## 22.1. 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) is always welcome.

### 22.1.1. 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.

### 22.1.2. HABARI_BROKEN_MESSAGELISTENER

**Enables asynchronous receive with IMessageListener.**

Compiles in the `MessageListener` property of `IMessageConsumer` and `IMQConsumer`, and the receive thread that calls `OnMessage`. Indy communication adapter only. The listener is supported within the limits described in its chapter; like the other symbols in this chapter, it is not covered by the free basic support plan. The word “BROKEN” is historical, and the define will be removed in a future release, when the listener code will always be compiled in.

**See also**: Chapter 11, *Asynchronous receive (IMessageListener)*

### 22.1.3. HABARI_LOGGING

**Enables logging support.**

Requires the open source SLF4P logging facade.

**See also**: Logging with SLF4P

### 22.1.4. 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

# Chapter 23. Useful units

## 23.1. 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;
```

## 23.2. 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;
```

# Chapter 24. Example applications

## 24.1. Overview

Table 4. Example applications (in alphabetic order)

| 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 |
| delphichat-listener | Several chat rooms on one IMQContext with message listeners (HabariChatRooms; its project defines HABARI_BROKEN_MESSAGELISTENER) |
| heartbeat-server | Uses server-side heartbeating 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 |

### 24.1.1. 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

## 24.2. ConsumerTool

### 24.2.1. 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.

Table 5. ConsumerTool command line options

| Parameter | Default value | Description |
|---|---|---|
| AckMode | CLIENT_ACKNOWLEDGE | Acknowledgement 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 |

| Parameter | Default value | Description |
|---|---|---|
| Topic | false | true: topic false: queue |
| Transacted | false | true: transacted session |
| URL | localhost | server URL |
| User | | user name |
| Verbose | true | verbose output |

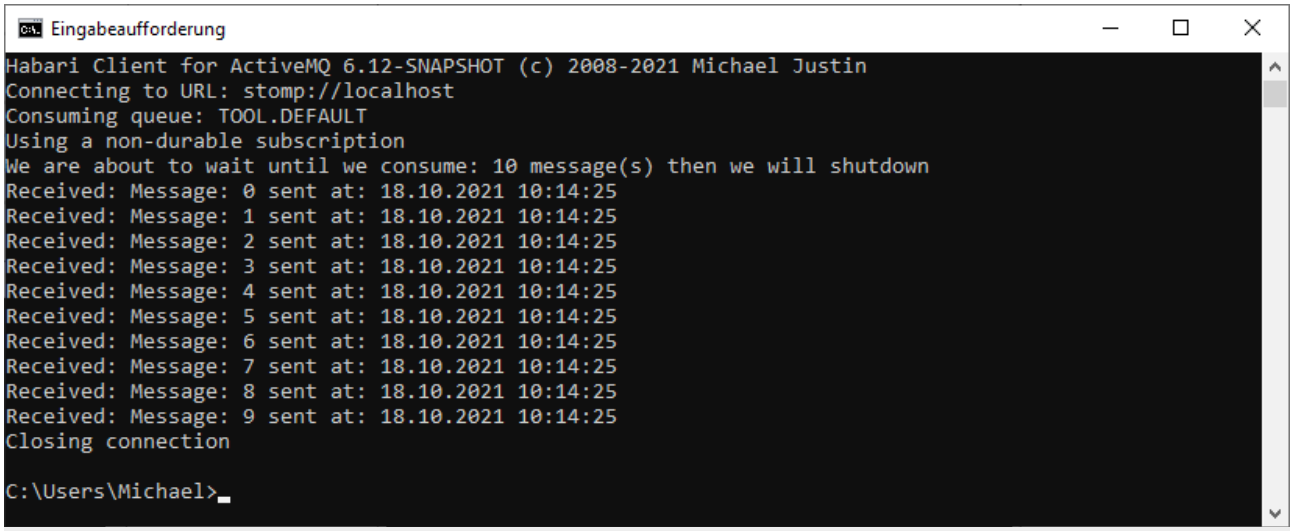



Figure 1. ConsumerTool demo application

## 24.2.2. 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
```

# 24.3. ProducerTool

## 24.3.1. 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.

Table 6. ProducerTool command line options

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

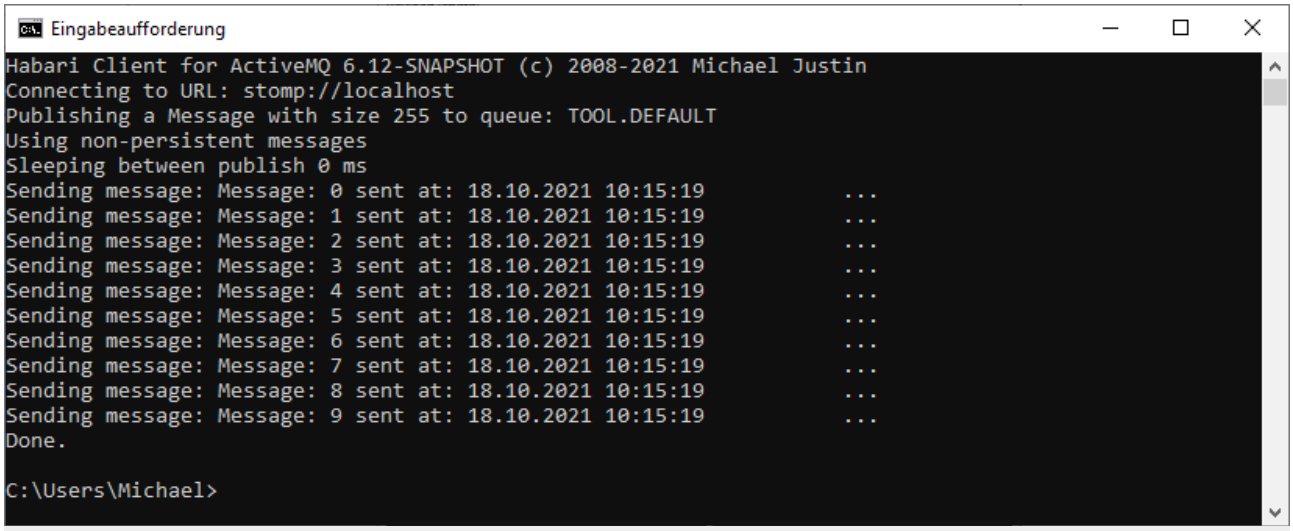



Figure 2. ProducerTool demo application

### 24.3.2. Examples

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

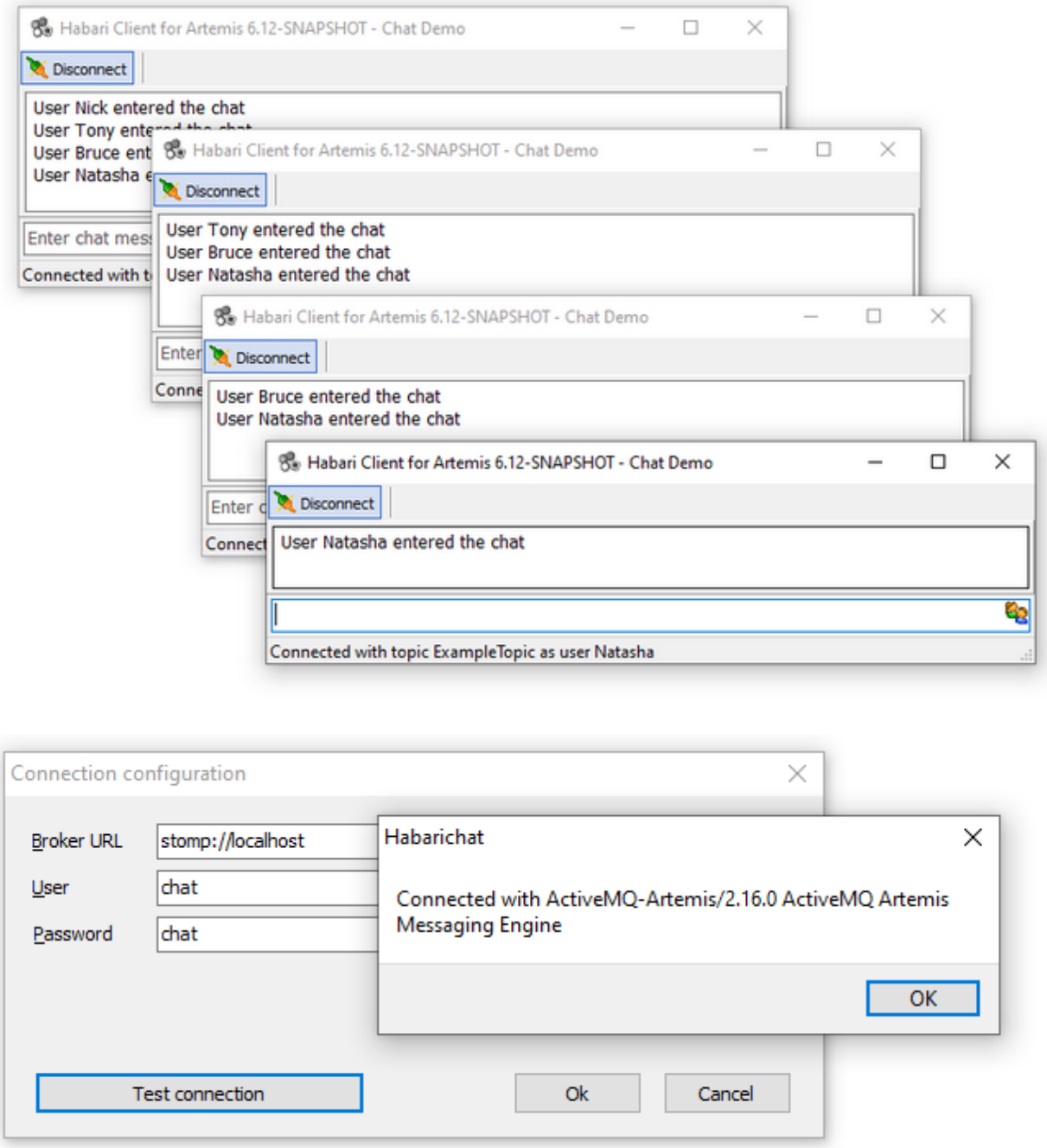
```
ProducerTool --MessageCount=10000
```

Send 10 messages to the topic ExampleTopic on the local broker

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

## 24.4. 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.[1]







## 24.5. Performance test

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

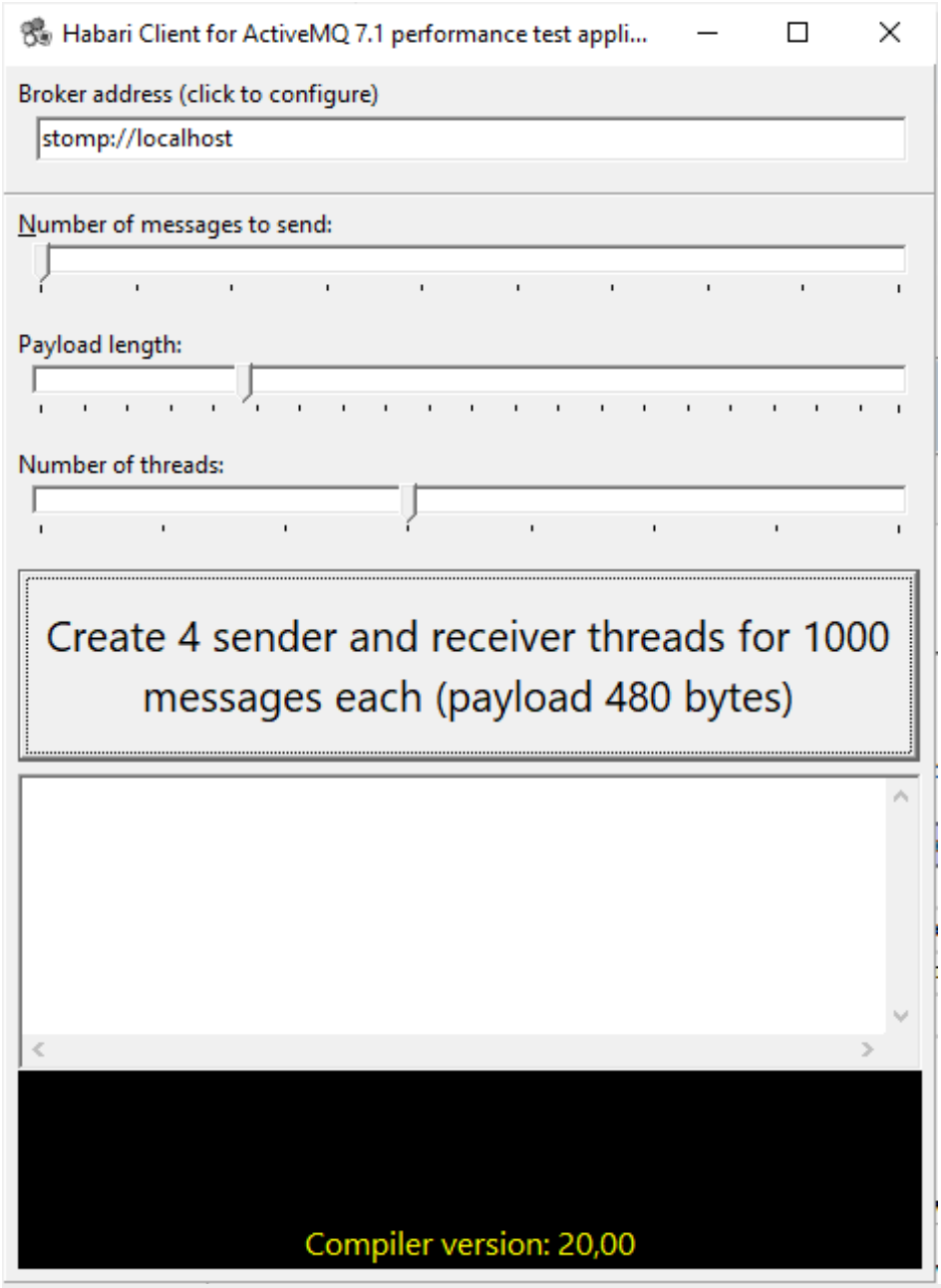



Figure 3. Performance test example application

- 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.

## 24.6. PingBroker

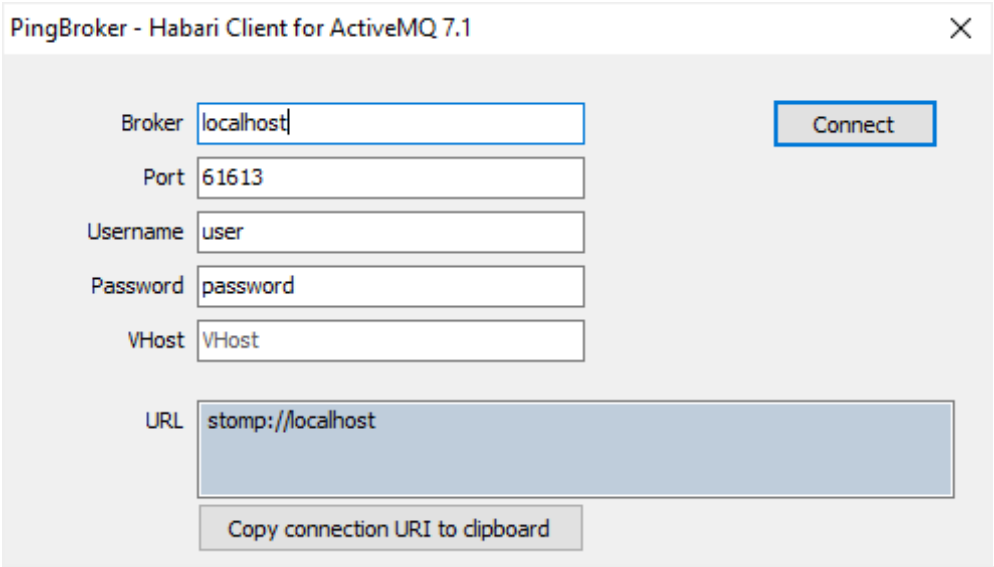



Figure 4. PingBroker example application

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

## 24.7. Throughput test

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

Table 7. Throughput test tool command line options

| Parameter | Default value | Description |
|---|---|---|
| Adapter | indy | Communication adapter: `indy` or `synapse` |
| Chart | false | Write the statistics as `elapsed;sent;received` lines (milliseconds since the start, running totals of sent and received messages) instead of the formatted output |
| Duration | 0 | Stop the test after this number of seconds; 0 runs until the user presses Enter |
| MaxBacklog | 100000 | The producer waits while more messages than this are unread; 0 = no limit |
| Mode | both | `both` runs producer and consumer; `produce` or `consume` runs one side only, to measure it alone (start the consumer and the producer in two windows) |
| Password | (broker-specific) | Password |
| Persistent | false | Send persistent messages |

| Parameter | Default value | Description |
|---|---|---|
| Verbose | false | Report when the producer waits because of the backlog limit |
| Subject | ExampleTopic | Topic name |
| URL | (broker-specific) | Connection URL |
| User | (broker-specific) | User name |

When the test stops (Enter or `--duration`), the producer stops first and the consumer receives the remaining messages. The test then reports the number of lost messages, for example `Sent: 209425, received: 209425, lost: 0`. The producer starts only after the broker has confirmed the subscription, so messages sent before it existed are not counted as lost. The report also shows the drain rate (the rate of the consumer alone while it receives the remaining messages) and how long the producer waited because of the backlog limit. Ctrl+C stops the test at once, without this report.

### 24.7.1. Examples

Use remote broker 'mybroker' and specify user and password

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

Run for 60 seconds and write the statistics to a file. The output has a line every two seconds; the difference between two lines gives the message rate of that interval.

```
tptest --chart --duration=60 > tptest.txt
```

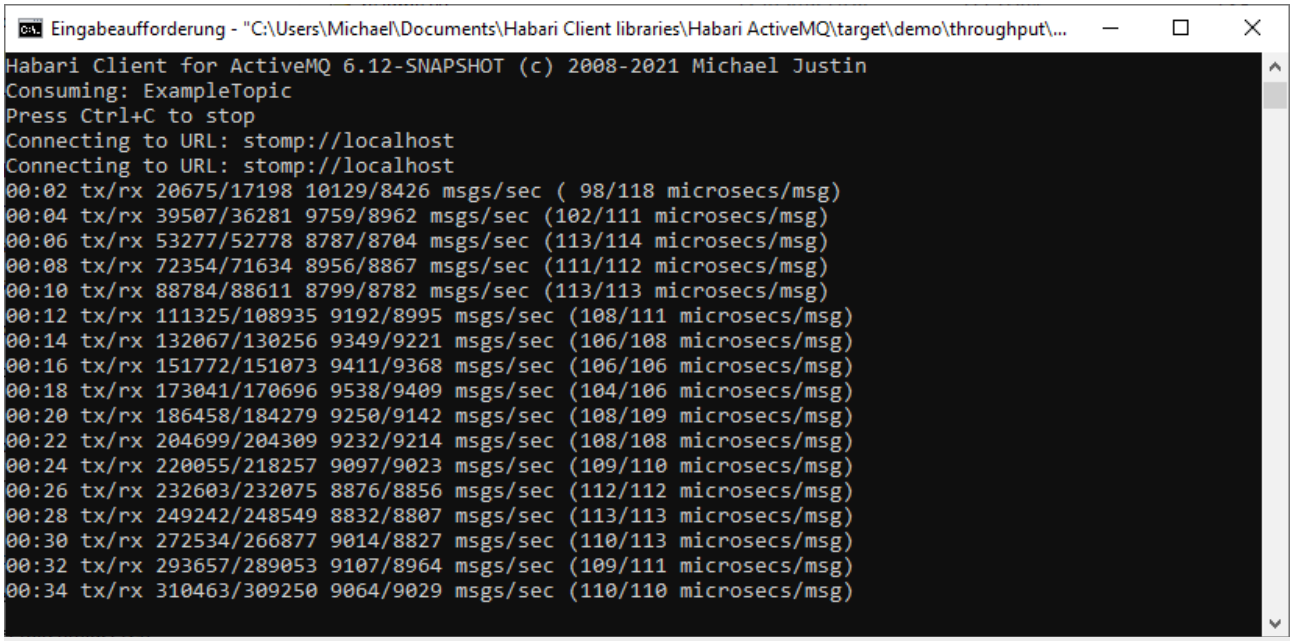



Figure 5. Throughput test tool output

[1] The screenshots were taken with an earlier version of Habari Client for Artemis. The window title differs, but the application works the same way with the current version and with all four brokers.

# Chapter 25. Unit tests

## 25.1. 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.

## 25.2. Test project configuration

### 25.2.1. Logging

To switch on SLF4P logging, add the conditional symbol HABARI_LOGGING (see Chapter 21, *Logging with SLF4P*) and rebuild the project. Set the DEFAULT_LOG_LEVEL constant in unit TestHelper to a valid SLF4P level.

### 25.2.2. Optional units

To switch on tests for optional units (connection builder and failover support), add the conditional symbol HABARI_TEST_OPTIONAL_UNITS and rebuild the project.

## 25.3. 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 |

| Unit tests | |
|---|---|
| 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 | Tests map and object message exchange with a message transformer |
| MessageListenerTests | Tests asynchronous receive with IMessageListener (new in 10.3; compiled only if HABARI_BROKEN_MESSAGELISTENER is defined) |
| 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.

## 25.4. Test execution

### 25.4.1. 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.

### 25.4.2. Test destinations

Most tests create a test-specific destination (a 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.

## 25.5. STOMP 1.2

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

# Part IV: Reference

# Chapter 26. Broker differences

The four Habari STOMP Client libraries share one code base, and most features work the same way with every broker. This chapter lists only what differs. A feature that is not listed here works the same with ActiveMQ, Artemis, OpenMQ and RabbitMQ.

Tested broker versions: Apache ActiveMQ 6.3.2, Apache ActiveMQ Artemis 2.57.0, Eclipse OpenMQ 6.9.0 and RabbitMQ 4.3.6.

Each limitation names its origin:

- **Broker:** the broker, or its STOMP connector, does not support the feature or does not handle it correctly. The client library cannot work around it.
- **Client:** the client library does not implement the feature for this broker, although the broker might support it.

Where the library detects an unsupported feature, it raises an exception when the feature is used instead of sending a request the broker would ignore.

## 26.1. Defaults

| | ActiveMQ | Artemis | OpenMQ | RabbitMQ |
|---|---|---|---|---|
| STOMP port | 61613 | 61613 | 7672 | 61613 |
| Default user / password | user / password | *(none)* | admin / admin | guest / guest |
| Message priority: default / maximum | 4 / 9 | 4 / 9 | 4 / 9 | 0 / 255 [1] |

1. RabbitMQ applies priorities only to queues declared with `x-max-priority` (see the RabbitMQ broker-specific notes). The library rejects values above 255.

## 26.2. Features and limitations

| | ActiveMQ | Artemis | OpenMQ | RabbitMQ |
|---|---|---|---|---|
| Heartbeating | Yes | Yes | No (broker) | Yes |
| Negative acknowledgement (`Nack`) | Yes | Partly (broker) | No (broker) | Yes |

| | ActiveMQ | Artemis | OpenMQ | RabbitMQ |
|---|---|---|---|---|
| Acknowledgement as part of a transaction | Yes | Partly (broker) | No (broker) | Yes |
| Message selectors | Yes | Yes | Yes | No (broker) |
| XPath selectors | Yes | Yes | No (broker) | No (broker) |
| NoLocal (ignore own messages) | Yes | Yes | Yes | No (broker) |
| Queue browser | Yes | Yes | No (broker) | No (broker) |
| Temporary queues and topics | Yes | No (broker) | Yes | Partly (broker) |
| Custom `content-type` header | Yes | Yes | No (broker) | Yes |
| Wildcards in queue names | Yes | No (broker) | No (broker) | No (broker) |
| Wildcards in topic names | Yes | Yes, own syntax | Yes | Yes |
| Composite destinations | Yes | No (broker) | No (broker) | No (broker) |
| Exclusive consumer | Yes | Partly (broker) | No (broker) | No (broker) |
| Message groups (`JMSXGroupID`) | Yes | Yes | No (broker) | No (broker) |
| Streams (`x-queue-type: stream`) | No (broker) | No (broker) | No (broker) | Yes |

## 26.2.1. Notes

**Heartbeating**

The OpenMQ STOMP connector does not support heartbeating. Detect connection loss by other means, for example with a periodic request.

**Negative acknowledgement**

**Artemis** accepts `Nack` but discards the message instead of delivering it again. The STOMP specification allows this, but the message is lost. **OpenMQ** does not know the NACK frame: it answers with an error (“Unknown STOMP command: NACK”) and closes the connection. Do not call `Nack` with OpenMQ.

**Acknowledgement as part of a transaction**

**Artemis** applies the acknowledgement at once and ignores its transaction: after a rollback, the message is not delivered again. **OpenMQ** never acknowledges a message that was received in a

transacted session, neither with `Acknowledge` nor with `Commit`, so the message is delivered again later. With OpenMQ, do not receive messages in a transacted session.

**Message selectors, NoLocal**

The RabbitMQ STOMP plugin has neither. The library raises `EIllegalArgumentException` when a consumer is created with a selector or with NoLocal. ActiveMQ and Artemis additionally support XPath selectors, and ActiveMQ supports numeric expressions in selectors (see the broker-specific notes of these two guides).

**Queue browser**

Neither the OpenMQ nor the RabbitMQ STOMP connector can browse a queue: RabbitMQ ignores a browse request and consumes the messages. `CreateBrowser` therefore raises an exception with these two brokers.

**Temporary queues and topics**

**Artemis** has no temporary destinations for STOMP clients, so `CreateTemporaryQueue` and `CreateTemporaryTopic` raise “Feature not supported”. Use a normal queue with a unique name for replies instead. **RabbitMQ** has temporary queues but no temporary topics, so `CreateTemporaryTopic` raises “Feature not supported”. A RabbitMQ temporary queue can only be used as a reply-to destination, and the broker subscribes to it when the first message with this reply-to is sent, so its consumer receives nothing before that. The broker acknowledges its messages automatically: create temporary queues in a session with `amAutoAcknowledge`. In a session with `amClientAcknowledge` or `amClientIndividual`, `CreateTemporaryQueue` raises `EIllegalStateException`.

**Custom `content-type` header**

The OpenMQ STOMP connector does not accept header names that contain hyphens. The library therefore marks text and bytes messages with its own `messagetype` header (see Section 26.3, “Protocol details”).

**Wildcards, composite destinations, exclusive consumer, message groups, streams**

These are broker extensions. The library passes the destination name and the headers through unchanged, so they work where the broker supports them. **Artemis** uses its own wildcard syntax (`#` and `*`); the delimiter is configured in `broker.xml`, and the configuration in the Artemis broker-specific notes sets it to `/`. With queues they do not work: a queue consumer with a wildcard receives nothing. Artemis has exclusive queues, but they are configured on the broker (`default-exclusive-queue` or the queue’s `exclusive` property). A STOMP client cannot request an exclusive consumer, as it can with ActiveMQ’s `activemq.exclusive` header.

**Streams**

A RabbitMQ stream keeps its messages after they have been read, and every consumer chooses

where it starts reading. A stream subscription needs a `prefetch-count` and client acknowledgement, and a message sent on another connection must repeat the stream's arguments as message properties, or it is lost. See the Getting Started guide of Habari STOMP Client for RabbitMQ.

## 26.3. Protocol details

These differences matter when your application builds destination names or headers itself, or exchanges messages with clients in other languages.

| | **ActiveMQ** | **Artemis** | **OpenMQ** | **RabbitMQ** |
|---|---|---|---|---|
| Destination names | `/queue/`, `/topic/`, `/temp-queue/`, `/temp-topic/` | `queue/`, `topic/` (no leading slash; routing configured in `broker.xml`) | `/queue/`, `/topic/`, `/temp-queue/`, `/temp-topic/` | `/queue/`, `/topic/`, and also `/exchange/`, `/amq/queue/`, `/temp-queue/`, `/reply-queue/` |
| Durable subscription | `activemq.subscriptionName` header, client ID required | `durable-subscription-name` header, client ID required | `durable-subscriber-name` header, client ID required | `persistent:true` and `auto-delete:false` on the subscription; no client ID needed |
| NoLocal header | `activemq.noLocal` | `no-local` | `no-local` | — |
| Expiration header | `expires`: absolute time | `expires`: absolute time | `expires`: time to live | `expiration`: time to live |
| How a received message is recognised as text | no `content-length` header | STOMP 1.1 and later: `content-type` starts with `text/`; STOMP 1.0: no `content-length` header | `messagetype:text` header | `content-type` starts with `text/` |

| | ActiveMQ | Artemis | OpenMQ | RabbitMQ |
|---|---|---|---|---|
| Flow control: how many unacknowledged messages the broker sends to a consumer | `activemq.prefetchSize`, in **messages**. Default 1000 for queues, 100 for persistent and 32766 for non-persistent topic subscriptions. 0 is not allowed. | `consumer-window-size`, in **bytes**. Default 10 KB with client acknowledgement; unlimited with automatic acknowledgement. `activemq.prefetchSize` is accepted, but also counts bytes. | No client setting. The broker property `imq.bridge.stomp.consumerFlowLimit` (default 1000 messages) applies to transacted subscriptions only. | `prefetch-count`, in **messages**. Default unlimited. |

Set these options as a parameter of the destination name, for example `CreateQueue('orders?activemq.prefetchSize=10')` or `CreateQueue('orders?consumer-window-size=0')`. The library adds them as headers to the SUBSCRIBE frame.

**Moving from ActiveMQ to Artemis:** `activemq.prefetchSize=10` means ten messages with ActiveMQ, but ten **bytes** with Artemis, which is less than one message. Use `consumer-window-size` with Artemis, and give it in bytes. A window of `0` delivers one message at a time: the next one arrives only after the previous one is acknowledged.

For the RabbitMQ destination types and the Artemis routing configuration, see the broker-specific notes of these products.

# Chapter 27. Library limitations

## 27.1. MessageConsumer

### 27.1.1. 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. Alternatively, one connection can deliver messages from several destinations to message listeners (see Chapter 11, *Asynchronous receive (IMessageListener)*).

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.

## 27.2. Message properties

### 27.2.1. Only string data type supported

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 timestamp value, which is the internal representation in Java. But still, these values cannot be used with date type expressions.

#### Broker-specific exceptions

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

## 27.3. Multithreading

*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.*

## 27.4. Free Pascal specific restrictions

- *the test suites run on Windows only; on Linux, a client compiled with Free Pascal 3.2.2 has been smoke-tested (connect, send, receive) with Indy, Synapse and mORMot2*
- *the included unit test project uses FPCUnit for Free Pascal / Lazarus instead of DUnit*
- *the ready-made map and object message transformers in source/optional do not support Free Pascal; message transformers written for the application work with both compilers*
- *the library source code uses the Delphi mode switch {$MODE DELPHI}*
- *other limitations or restrictions may apply*

## 27.5. Broker-specific limitations

### 27.5.1. Other broker-specific limitations

*For broker-specific notes, please read Chapter 31, Broker-specific notes.*

# Chapter 28. Frequently asked questions

## 28.1. Technical questions

### 28.1.1. 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.

### 28.1.2. 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.

## 28.1.3. Why am I getting “unexpected subscription id” or “Found no matching consumer” errors?

### Short answer

A message arrived for a subscription that the receiving consumer does not own. Usually the client closed a consumer while messages for it were still on the wire, and then received them 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 matches 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. Which error you see depends on how you receive:

- With `Receive`, the consumer raises an `EIllegalStateException`: “Received a MESSAGE frame with unexpected subscription id. Received "..." but expected "...".”
- With a message listener (`HABARI_BROKEN_MESSAGELISTENER`), the session finds no consumer for the message and raises “Found no matching consumer for message with subscription ...”. This also happens when a connection with listeners has several sessions (see Section 11.4, “Rules”, rule 1).

### 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.

The `Receive` check can be turned off with the connection URL parameter `message.subscription_check=false`. Do this only if you understand why the ids differ: the consumer then accepts messages of other subscriptions.

#### Example

Here is a small code example which causes this error[1]:

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;
```

The consumer is closed and a new consumer is created for the same destination (`Consumer.Close` and the second `CreateConsumer`).

The final `Receive` detects that the incoming message does not have a matching subscription id and raises an `EIllegalStateException`.

## 28.1.4. 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 3.2.2. Support

for non-Unicode Delphi ended in April 2017.

## 28.1.5. How can the client application detect network connection loss?

### Short answer

Use STOMP heartbeating.

### Long answer

By enabling heartbeating, the client can request server-side sending of heartbeat bytes.

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

To detect if the server has sent a heartbeat, the client calls the method ReceiveHeartbeat.

For more details, please check the section "Reading server-side heartbeats".

With an IMessageListener, call CheckHeartbeat instead (for example from a timer). A listener's receive thread already reads the connection, so ReceiveHeartbeat, which also reads it, would compete with it for incoming data and can take part of a message. CheckHeartbeat does not read the connection: it raises EBTStompServerHeartbeatMissing when nothing has arrived for more than twice the server's heartbeat interval, which also happens when the receive thread has ended. See Section 11.7, "Connection loss".

[1] This code example is included in the library unit test project.

# Chapter 29. Online resources

## 29.1. Third-party libraries

### 29.1.1. 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 Free Pascal.

| | |
|---|---|
| GitHub | https://github.com/IndySockets |

### 29.1.2. Ararat Synapse

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

### 29.1.3. Synopse mORMot 2

| | |
|---|---|
| GitHub | https://github.com/synopse/mORMot2 |
| Tested version | https://github.com/synopse/mORMot2/tree/2.4-stable |

### 29.1.4. DUnit

| | |
|---|---|
| SourceForge | https://sourceforge.net/projects/dunit/files/dunit/9.3.0/ |

### 29.1.5. SLF4P

Simple logging facade for Object Pascal

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

## 29.2. Specifications

| **STOMP – Simple (or Streaming) Text Oriented Messaging Protocol**[1] | |
|---|---|
| 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 |

## 29.3. Online articles

| Title | Broker |
|---|---|
| Firebird Database Events and Message-oriented Middleware[2] | All |
| Discover ActiveMQ brokers with Delphi XE4 and Indy 10.6[3] | ActiveMQ |
| How to use the RabbitMQ Web-STOMP Plugin[4] | RabbitMQ |
| RPC with Delphi client and Java server using RabbitMQ[5] | RabbitMQ |

## 29.4. Online videos

| Title | Broker |
|---|---|
| Introduction to Messaging With Apache ActiveMQ[6] | ActiveMQ |
| GlassFish Message Queue – High Availability Clusters[7] | OpenMQ |

[1] https://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol

[2] https://mikejustin.wordpress.com/2012/11/06/firebird-database-events-and-message-oriented-middleware/

[3] https://mikejustin.wordpress.com/2013/07/07/discover-activemq-brokers-with-delphi-xe4-and-indy-10-6/

[4] https://mikejustin.wordpress.com/2013/11/27/how-to-use-the-rabbitmq-web-stomp-plugin-with-delphi-and-free-pascal/

[5] https://mikejustin.wordpress.com/2013/05/21/rpc-with-delphi-client-and-java-server-using-rabbitmq/

[6] https://vimeo.com/12654513

[7] https://www.youtube.com/watch?v=RHUJBsy3udU

# Chapter 30. Support

## 30.1. 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.

### 30.1.1. Advanced support

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

# Part V: ActiveMQ

# Chapter 31. Broker-specific notes

## 31.1. Enable the STOMP transport

In versions before 6.3.0, ActiveMQ ships with stomp/amqp/mqtt/ws connectors enabled by default.

For 6.3.0 or newer, open the conf/activemq.xml file, and check if the transportConnector element is commented out. If yes, uncomment it.

```
<!--
                Additional transports are disabled by default to reduce the exposed
attack surface.
                Uncomment only the protocols you actually need, and prefer the secured
variants
                (openwire+ssl, amqp+ssl, stomp+ssl, mqtt+nio+ssl, wss) in production
deployments.
            -->
            <!-- <transportConnector name="amqp"
uri="amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFrameSize=10485760"
/> -->
            <transportConnector name="stomp"
uri="stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireFormat.maxFrameSize=1048576
0"/>
...
```

## 31.2. Subscription options

ActiveMQ supports broker-specific arguments which can be passed with the STOMP SUBSCRIBE command.[1]

These arguments can be passed in the CreateQueue method.

Code example

```
Session.CreateQueue('myqueue?activemq.prefetchSize=1');
```

This will add the header activemq.prefetchSize=1 to the SUBSCRIBE frame.

## 31.3. Selectors

### 31.3.1. Using numeric selectors to filter messages

Apache ActiveMQ 5.6 introduced support for numeric expressions in selectors[2]. STOMP sends every message property as a string, but for STOMP subscriptions ActiveMQ converts string properties to numbers when a selector compares them with a number. No extra setting is needed:

Code example

```
Consumer := Session.CreateConsumer(Destination, 'age > 18');
...
Msg.SetIntProperty('age', 21); // matches
```

**NOTE**

The ActiveMQ documentation on selectors[3] tells STOMP clients to start the selector with `convert_string_expressions:`. Do not add this prefix: ActiveMQ adds it to every STOMP selector itself, and a second prefix makes the selector invalid.

### 31.3.2. XPath selectors

ActiveMQ also supports XPath selectors, which filter on the XML message body instead of message properties. Write the XPath expression in single quotes after the keyword `XPATH`:

Code example

```
Consumer := Session.CreateConsumer(Destination, 'XPATH ''//title[@lang="eng"]''');
```

The consumer receives only messages whose body is XML and matches the expression. The broker has to parse the XML body of every message, so XPath selectors may be much slower than property selectors.

Check the expression carefully: ActiveMQ 6.3.2 accepts an invalid XPath expression without an error, neither in `CreateConsumer` nor later, and the consumer simply receives no messages.

The example project `xpath-selector` in the demo-brokerspecific folder uses three XPath selectors on one topic.

**NOTE**

The ActiveMQ documentation on selectors (linked in the note above) says that XPath support requires the Xalan library, which is not part of the distribution. ActiveMQ 6.3.2 evaluates XPath selectors with the XPath support built into Java, so Xalan is not needed there.

## 31.4. Object messages

### 31.4.1. Object serialization

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.[4] 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.

ActiveMQ supports object exchange between Java and non-Java clients using a Message Transformation between native Java objects and XML or JSON serialized objects.[5]

### 31.4.2. Broker-side transformation vs. client-side serialization

Habari STOMP Client for ActiveMQ offers two object exchange methods. In both, the Habari client sends the object as JSON or XML text. They differ in where that text is turned into an object again, and therefore in what Java clients receive:

**Broker-side transformation: Java clients receive a JMS ObjectMessage.** The broker transforms the JSON or XML text into a binary serialized Java object before JMS clients consume it, and back again for Habari clients. This requires that the Java class files are in the broker's class path.

**Client-side serialization: the broker stores the JSON or XML text unchanged.** Habari clients turn the text back into an object with their message transformer. All other clients, Java ones included, receive the same text as a text message. No Java class files are required.

#### Broker-side transformation

On the Java side, a Java client application does not need any special preparation to send and receive objects over ActiveMQ. The JMS API support for ObjectMessage provides all necessary methods, a session uses `Session#createObjectMessage(Serializable object)`[6] to create the message (passing a Java object as argument) which then can be sent and received just like a TextMessage or BytesMessage.

However, for the message transformation to and from JSON or XML, this object exchange method requires that a JAR containing a matching Java class file has to be deployed in the message broker, which will be used by the broker's message transformer. If this Java class is not compatible with the JSON or XML structure, the message transformation fails!

The example projects `jms-objectmessage` and `jms-mapmessage` in the demo-brokerspecific folder

show this exchange: a Java client (Maven project in the same folder) sends a JMS ObjectMessage or MapMessage, and the Delphi application receives it as XML through the ActiveMQ transformation `jms-object-xml` or `jms-map-xml`. The Delphi side registers the transformer with this identifier and adds `?transformation=jms-object-xml` (or `jms-map-xml`) to the queue name.

#### Pros

- Java clients do not need any special modifications to exchange objects with non-Java clients, Habari clients can be connected ('plugged in') / integrated easily with an existing JMS infrastructure
- Serialization from / to objects is performed on the server
- Serialization only occurs 'on demand' when the non-Java client reads or writes messages

#### Cons

- Requires installation of a JAR file in the message broker which contains the Java class (unless the class is already in the broker's class path)
- The transformation fails if the Java class and Pascal class declaration do not match
- The transformation fails if the Pascal and Java transformer libraries (JSON / XML) are not compatible

### Client-side serialization

There are almost no differences in the application code between the two methods.

Switching to client-side serialization requires only an additional property assignment on the object message: set the property `SH_TRANSFORMATION_CUSTOM` (header `transformation-custom`) to the transformation identifier. The library then sends no `transformation` header, so ActiveMQ does not transform the message, and the receiving Habari client still finds the transformer by this identifier:

Code example

```
ObjectMessage := Session.CreateObjectMessage;
ObjectMessage.SetObject(MyObject);
ObjectMessage.SetStringProperty(SH_TRANSFORMATION_CUSTOM, 'my-object-message');
Producer.Send(ObjectMessage);
```

Map messages work the same way.

The serialized objects will be stored in the message broker as TextMessage instances. The XML or

JSON text can be retrieved by a JMS Java client application just like any other JMS TextMessage. Java clients can use a JSON or XML parser to read the message content.

#### Pros

- Simple usage, no JAR installation required
- Java JMS client applications are still able to receive the serialized objects – they will appear as TextMessage instances, containing the JSON or XML text

#### Cons

- Deserialization of JSON or XML serialized Pascal objects to Java objects requires a decoder library (XStream or Jettison) on the Java client side

### 31.4.3. Memory management

#### Outgoing objects

The message transformer will not free objects which have been sent. To release the memory, the application has to explicitly free them when they are no longer used.

#### Incoming objects

The message transformer will create an object instance when an object message has been received. To avoid memory leaks, the application must free this instance when it is no longer in use.

## 31.5. Broker-specific demos

### 31.5.1. Broker statistics example

The ActiveMQ statistics plugin looks for messages sent to particular destinations. To query the running statistics of the message broker, send an empty message to a Destination (Queue or Topic) named `ActiveMQ.Statistics.Broker`, and set the JMSReplyTo field with the Destination you want to receive the result on. The statistics plugin will send an IMapMessage filled with the statistics for the running ActiveMQ broker.

Similarly, if you want to query the statistics on a Destination, send a message to the Destination name,

prepended with ActiveMQ.Statistics.Destination. For example, to retrieve the statistics on a Queue named test.foo send an empty message to the Queue `ActiveMQ.Statistics.Destination.test.foo`.

You can also use wildcards, and receive a separate message for every destination matched.

### Configuration

To configure ActiveMQ to use the statistics plugin, add the following to the ActiveMQ XML configuration:

```
...
  <plugins>
    <statisticsBrokerPlugin/>
  </plugins>
...
```

### Example output

When launched with parameter example.A, the demo application activemq-statistics will retrieve the information for queue example.A, and the output would look similar to this:

```
Request statistics for ActiveMQ.Statistics.Destination.example.A ...
memoryUsage=0
dequeueCount=0
inflightCount=0
messagesCached=0
averageEnqueueTime=0.0
destinationName=queue://example.A
size=0
memoryPercentUsage=0
producerCount=0
consumerCount=1
minEnqueueTime=0.0
maxEnqueueTime=0.0
dispatchCount=0
expiredCount=0
enqueueCount=0
memoryLimit=67108864
Press any key
```

Without a parameter, broker statistics will be returned:

```
Request statistics for ActiveMQ.Statistics.Broker ...
vm=vm://localhost
memoryUsage=0
storeUsage=66434225
tempPercentUsage=0
openwire=tcp://mj-PC:61616
brokerId=ID:mj-PC-52958-1272975061672-0:0
consumerCount=3
brokerName=localhost
expiredCount=0
dispatchCount=2
maxEnqueueTime=3.0
storePercentUsage=0
dequeueCount=2
inflightCount=0
messagesCached=0
tempLimit=107374182400
averageEnqueueTime=1.5
memoryPercentUsage=0
size=0
tempUsage=0
producerCount=0
minEnqueueTime=0.0
dataDirectory=C:\Java\apache-activemq-5.3.1\data
enqueueCount=64
stomp=stomp://mj-PC:61613?transport.closeAsync=false
storeLimit=107374182400
memoryLimit=67108864
Press any key
```

## 31.5.2. Delay and schedule message delivery

Apache ActiveMQ from version **5.4** has a persistent scheduler built into the ActiveMQ message broker. An ActiveMQ client can take advantage of a delayed delivery by using message properties.[7]

By setting properties of the message, a client can

- set the time in milliseconds that a message will wait before being scheduled to be delivered by the broker
- set the time in milliseconds to wait after the start time before scheduling the message again
- set the number of times to repeat scheduling a message for delivery
- or use a **cron** entry (for example "*0 * * * **" to set the schedule)

The example application shows how a message can be scheduled for delivery after 5 seconds.

To enable the scheduler, the broker element in the configuration file needs to include the **schedulerSupport** attribute set to true.

[1] See https://activemq.apache.org/components/classic/documentation/stomp

[2] https://issues.apache.org/jira/browse/AMQ-1609

[3] https://activemq.apache.org/components/classic/documentation/selectors

[4] https://www.oracle.com/technical-resources/articles/java/serializationapi.html

[5] https://activemq.apache.org/message-transformation.html

[6] https://docs.oracle.com/javaee/1.4/api/javax/jms/Session.html#createObjectMessage(java.io.Serializable)

[7] https://activemq.apache.org/delay-and-schedule-message-delivery.html

# Chapter 32. Connection troubleshooting

## 32.1. Performance demo

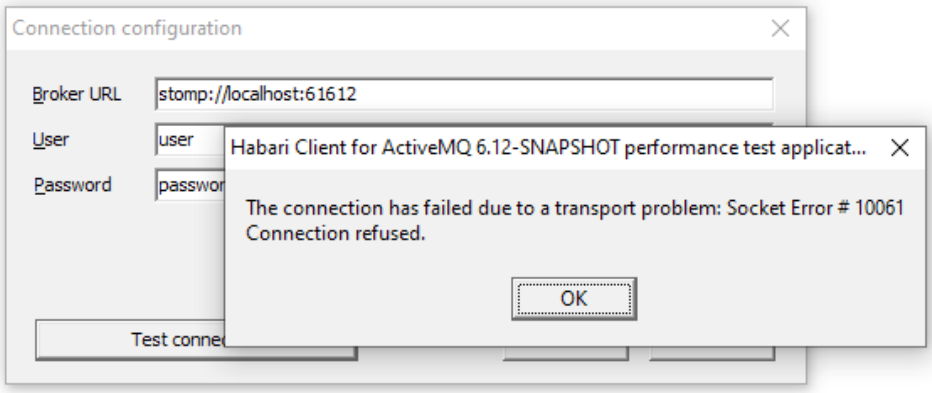
### 32.1.1. Socket error 10060 (connection timed out)

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

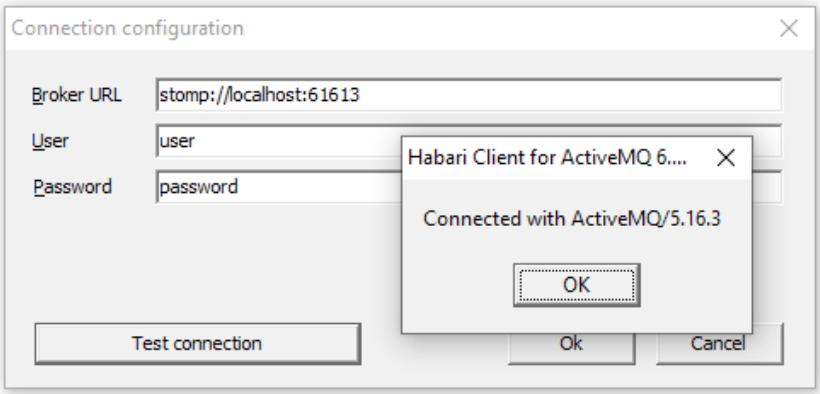
### 32.1.2. Socket error 10061 (connection refused)

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 ActiveMQ is 61613.

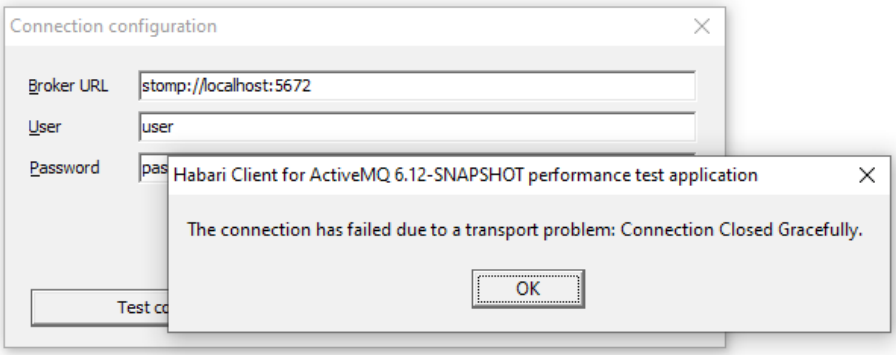



The port can be specified in the Broker URI:



### 32.1.3. 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 Closed Gracefully” error will occur.




For example, the ActiveMQ server will allow only AMQP connections on port 5672, not STOMP.

# Index


**B**

Broker differences, 88
Broker statistics, 108
BTBrokerConsts, 86
BTCommAdapterIndy), 24
Bug reports, 102

**C**

charset, 65
CheckHeartbeat, 63
conditional symbols
    HABARI_BROKEN_MESSAGELISTENER, 37
connect.accept-version, 62
connect.heart-beat, 62
connect.host, 62
Connection, 17, 24
connection loss, 42
Connection URL, 27
ConnectionFactory, 17
ConnectionFactory), 24
content-type, 65
CorrelationID, 48
CreateDurableSubscriber, 46
credentials, 86

**D**

DeliveryMode, 48
demos
    delphichat-listener, 38
Destination, 32, 48
DUnit, 3, 85

**E**

EBTStompServerHeartbeatMissing, 63
EIllegalStateException, 96
EMQException, 66
experimental features, 102
Expiration, 48

**F**

FPCUnit, 3, 85
Free Pascal, 3

**H**

HABARI_BROKEN_MESSAGELISTENER, 37
HABARI_DEPRECATED_MESSAGELISTENER, 8
HABARI_ENABLE_FAILOVER_PROTOCOL, 6
HABARI_SSL_SUPPORT, 68
HABARI_X_MAXLINELENGTH, 6
heartbeat
    connection loss, 42

**I**

IHeartbeat, 63
IMapMessage, 108
IMessageListener, 37
IMQConsumer, 60
IMQContext, 59
IMQProducer, 59
Internet Direct (Indy), 3

**J**

JMSReplyTo, 108

**L**

Limitations, 93
Linux, 3, 102
Logging, 70
LoggingHelper, 70

**M**

Map messages, 51
MapMessageTransformerTests, 55
message properties, 93
MessageConsumer, 17, 35
MessageID, 48
MessageProducer, 17, 34

**O**

Object messages, 56, 106
ObjectMessageTransformerTests, 58
OpenSSL, 68

**P**

Point-to-point, 32
Prefetch, 92
Priority, 48
Publish-and-subscribe, 32

**Q**

Queue, 32

**R**

Receive, 36
ReceiveHeartbeat, 64
ReceiveNoWait, 36
ReplyTo, 48, 48

**S**

Selector, 93
Selectors, 50
    XPath, 105
SendHeartbeat, 63
Session, 17, 27
SetDefaultAdapter, 68
SimpleLogger, 71
Socket error 10054, 113
Socket error 10060, 112
Socket error 10061, 112
SSL/TLS, 102
STOMP 1.2, 62
    optional features, 65
Support, 102
Synapse, 3

**T**

TCP, 93
Test destinations, 86
Timestamp, 48
Topic, 33
TopicSubscriber, 46
Transacted sessions, 29
transactions, 94

**V**

virtual host, 62