Getting started with

Habari Client for ActiveMQ

Version 6.11

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.

Contents

Broker-specific information

For broker-specific notes, please read chapter Broker-specific notes on page 79 ff.

Installation

Requirements

Development Environment

- **Embarcadero Delphi** 2009 Update 4 or higher

- or -

- **Free Pascal** 3.0.4 or higher[1]

Lazarus 2.0.8 or newer is required to run the **FPCUnit** test suite. The DUnit test suite and the GUI demo applications require Delphi 2009 for compilation.

TCP/IP Communication Library

- **Internet Direct (Indy) 10.6** (recommended)

- or -

- **Synapse** Release 40[2] (deprecated)

Installation steps

The installer application will guide you through the installation process.

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

C:\Users\Public\Documents\Habarisoft\habari-activemq-6.11

1 Support for Free Pascal versions below 3.2.0 will be removed in a future release
2 Only release 40 of Ararat Synapse is used for Habari Client library development and tests

Communication Adapters

Introduction

Habari 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.

Configuration of communication adapters

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

Registration of communication adapter class

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

Code example

```
program ClientUsingIndy;

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

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

Default adapter class

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

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

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

Code example

```
program ClientUsingIndyOrSynapse;

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

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

Available communication adapters

The library includes two adapter classes for TCP/IP libraries, one for Indy (Internet Direct) and one for Synapse.

Adapter Class	Unit
TBTCommAdapterIndy	BTCommAdapterIndy
TBTCommAdapterSynapse	BTCommAdapterSynapse

Table 1: Communication Adapters

Limitations of the Synapse communication adapter class

- The Synapse library does not support the ConnectTimeout property in synchronous socket operation mode, as connect timeouts are handled by the operating system. Indy uses a background thread to abort the connect operation.[3]
- Release 40 of Ararat Synapse is used for Habari Client library development and tests. This is the last announced release, dated April 24, 2012. This release is compatible for Delphi versions before XE4[4]. If you use a newer release of Ararat Synapse, please let me know if you encounter any API incompatibilities or other problems.

3 http://www.ararat.cz/synapse/doku.php/public:howto:connecttimeout
4 http://docwiki.embarcadero.com/RADStudio/XE4/en/Global_Variables

The Programming Model

Habari Client libraries use a programming model which is based on message producers and message consumers, sessions, connections and connection factories.

The basic API is the same for all library versions to allow easy migration between supported message brokers (with the exception of broker-specific features).

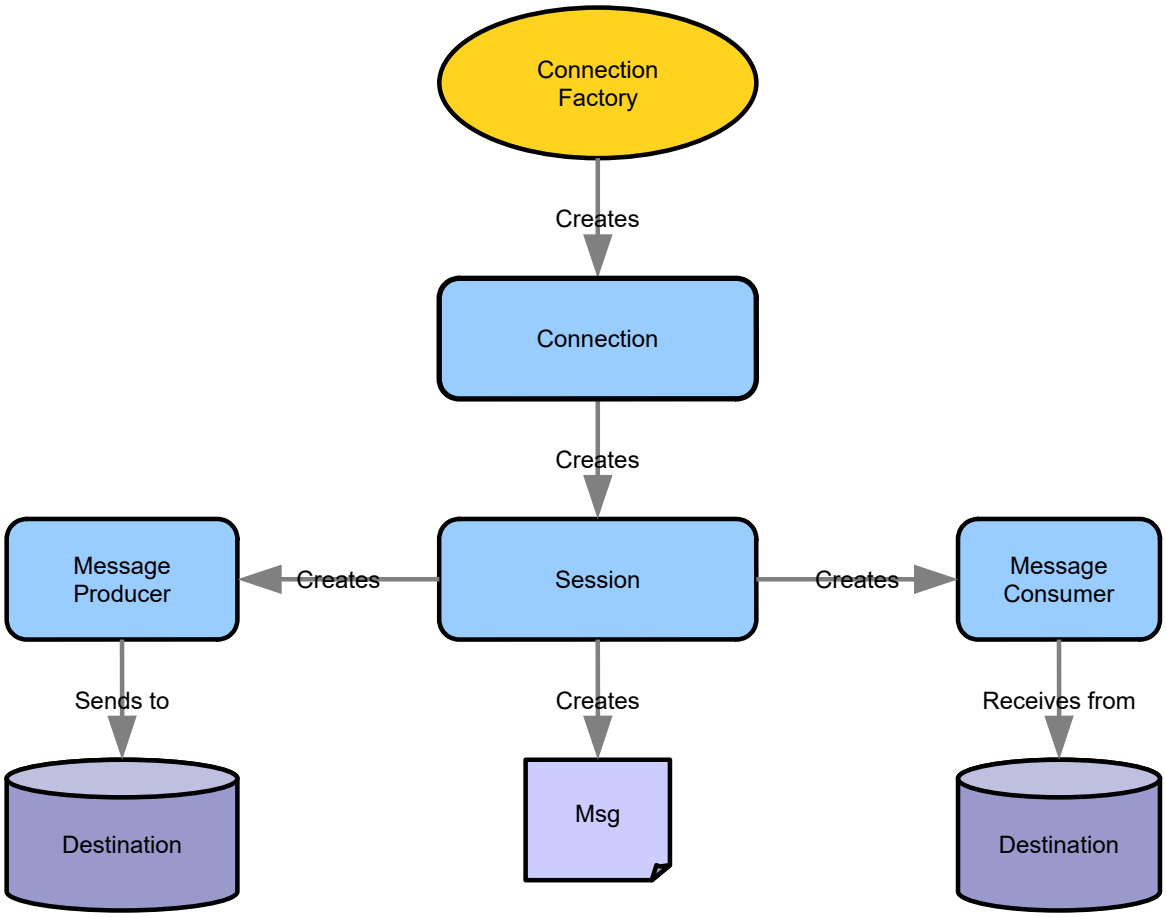

Illustration 1: Programming Model

New simplified API

See also: section Simplified API on page 44.

Tutorials

Quick Start Tutorial

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

This tutorial takes less than 10 minutes to complete.

Setting up the project

To create a new project:

1. Start the Delphi IDE.
2. In the IDE, choose File > New > VCL Forms Application – Delphi
3. Choose Project > Options … to open the Project Options dialog
4. In the options tree on the left, select 'Delphi Compiler'
5. Add the source directory of Habari 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.

Adding code to the project

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

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

Code example

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

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

Code example

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

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

  Connection.Close;
end;
```

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

Code example

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Consumer: IMessageConsumer;
  Msg: ITextMessage;
begin
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Connection := Factory.CreateConnection;
  Connection.Start;

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

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

  Connection.Close;
end;
```

16. Compile and save the project

Run the demo

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

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

Check for memory leaks

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

Code example

```
program HelloMQ;

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

{$R *.res}

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

Tutorial source code

The tutorial source code is included in the demo folder. It does not include a .proj file so you still need to add the Habari and Indy source paths to the project options.

Connection Factory

Overview

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

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

Creation and configuration

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

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

Code example

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

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

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

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

Code example

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

  // ------------------------------------------------------------
  // additional configuration
  // ------------------------------------------------------------

  Factory.BrokerURL := 'broker.example.com';
  Factory.UserName := 'guest';
  Factory.Password := 'guest';
  Factory.ClientID := 'myclientId';
  Factory.SendTimeOut := 10000;
  Factory.ConnectTimeOut := 10000; // Indy only

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

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

Code example

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

  C := F.CreateConnection;

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

Connection URL parameters

Heart-beating Support

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

Failover Support

The Failover transport layers reconnect logic on top of the Stomp transport.[5]

The Failover configuration syntax allows you to specify any number of composite URIs. The Failover transport randomly chooses one of the composite URI and attempts to establish a connection to it. If it does not succeed, a new connection is established to one of the other URIs in the list.

Example for a failover URI:

```
failover:(stomp://primary:61613,stomp://secondary:61613)
```

5 http://activemq.apache.org/failover-transport-reference.html

Failover Transport Options

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt (in ms)
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts (in ms)
backOffMultiplier	2.0	The exponent used in the exponential backoff attempts
maxReconnectAttempts	-1	-1 is default and means retry forever, 0 means don't retry (only try connection once but no retry) If set to > 0, then this is the maximum number of reconnect attempts before an error is sent back to the client
randomize	true	use a random algorithm to choose the the URI to use for reconnect from the list provided

Table 2: Failover Transport Options

Example URI:

```
failover:(stomp://localhost:61616,stomp://remotehost:61616)?
initialReconnectDelay=100&maxReconnectAttempts=10
```

Code example

```
Factory := TBTConnectionFactory.Create('failover:(stomp://primary:61616,stomp://
localhost:61613)?maxReconnectAttempts=3&randomize=false') do
try
  Conn := Factory.CreateConnection;
  Conn.Start;
  ...
  Conn.Stop;
finally
  Conn.Close;
end;
```

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."[678]

With Habari 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.

Receipt Support Parameters

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

SUBSCRIBE Receipt

To erquest server reseipts for SUBSCRIBE frames, use the optional connection URL parameter, subscribe.receipt.

Code example

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

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

UNSUBSCRIBE Receipt

To erquest server reseipts for UNSUBSCRIBE frames, use the optional connection URL parameter, unsubscribe.receipt.

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

Code example

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

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

SEND Receipt

To erquest server reseipts for SEND frames, use the optional connection URL parameter, send.receipt.

Code example

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

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

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

DISCONNECT Receipt

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

Code example

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

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

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

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

Connections and Sessions

Connections use Stomp 1.2 by default

Connections use Stomp 1.2 by default since

- Habari Client for Apache ActiveMQ 5.1
- Habari Client for Apache Artemis 5.1
- Habari Client for RabbitMQ 5.1

With OpenMQ, the library still uses Stomp 1.0.The default protocol version is defined in the BTBrokerConsts unit. The Stomp version may be specified by a connection URL parameter.

Step-by-Step Example

Overview

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

Add required units

Three units are required for this example

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

The SysUtils unit is necessary for the exception handling.

Code example

```
program SendOneMessage;

{$APPTYPE CONSOLE}

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

Creating a new Connection

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

Code example

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

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

Connection URL Parameters

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

Creating a Session

To create the communication session,

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

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

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

Code example

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

Using the Session

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

Code example

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

Closing a Connection

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

Code example

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

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

Session types overview

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

Parameters	**Client MUST acknowledge message receipt**[9]	**Transaction support for**		**STOMP Version**
		Send	Ack	
CreateSession(False, amAutoAcknowledge)	No	-	-	1.0
CreateSession(False, amClientAcknowledge)	Yes (cumulative effect)	-	-	1.0
CreateSession(False, amClientIndividual)	Yes	-	-	1.2
CreateSession(True, amAutoAcknowledge)	No	✓	-	1.0
CreateSession(True, amClientAcknowledge)	Yes (cumulative effect)	✓	✓ ①	1.0
CreateSession(True, amClientIndividual)	Yes	✓	✓ ①	1.2
CreateSession(True, amTransactional)	No	✓	-	1.0

Table 3: Session creation parameters

① – not supported by ActiveMQ Artemis

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.

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);
```

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

or (using the older API version)

Code example

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

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

Send messages

Now send messages using the transacted session.

Code example

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

Committing a transaction

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

Code example

```
// send messages ...

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

Note: committing a transaction automatically starts a new transaction

Rolling back a transaction

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

Code example

```
// send messages ...

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

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

Transacted message acknowledgement

Some library versions (see table "Communication Adapters" on page 10) support transactions also for the acknowledgement of received messages.

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

Code example

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

  // process the message
  ...

  // acknowledge the message
  Msg.Acknowledge;

  ...

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

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

Destinations

Introduction

The API supports two models:[10]

1. point-to-point or queuing model
2. publish and subscribe model

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

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

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

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

Create a new Destination

Queues

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

10 Java Message Service. (2007, November 21). In Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Java_Message_Service

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)

Topics

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

Code example

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

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

Producer and Consumer

Message Producer

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

Code example

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

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

Persistent messages

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

Setting the default delivery mode

Code example

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

Setting the delivery mode for an individual message

Code example

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

Message Consumer

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

Code example

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

Message Selector

A message consumer can be created with a **message selector**[11].

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

Synchronous Receive

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

11 The RabbitMQ message broker does not support message selectors

Code example

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

Receive and ReceiveNoWait

There are three different methods for synchronous receive:

Receive	The Receive method with no arguments will block (wait until a message is available).
Receive(TimeOut)	The Receive method with a timeout parameter will wait for the given time in milliseconds. If no message arrived, it will return nil.
ReceiveNoWait	The ReceiveNoWait method will return immediately. If no message arrived, it will return nil.

Durable Subscriptions

Description

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

The message broker retains a record of this durable subscription and insures that all messages from the topic's publishers are retained until they are acknowledged by this durable subscriber or they have expired.[12]

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

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

Creation

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

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

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

12 http://download.oracle.com/javaee/5/api/javax/jms/TopicSession.html

Temporary Queues

Introduction

> "Temporary destinations (temporary queues or temporary topics) are proposed as a lightweight alternative in a scalable system architecture that could be used as unique destinations for replies. Such destinations have a scope limited to the connection that created it, and are removed on the server side as soon as the connection is closed." ("Designing Messaging Applications with Temporary Queues", by Thakur Thribhuvan [13])

Library Support

Temporary destinations are supported by

- ActiveMQ
- OpenMQ
- RabbitMQ

Resource Management

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

13 http://onjava.com/pub/a/onjava/2007/04/10/designing-messaging-applications-with-temporary-queues.html

Message Options

Standard Properties

The Apache ActiveMQ message broker supports some JMS standard properties in the STOMP adapter. These properties are based on the JMS specification of the Message interface.[14]

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

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

Properties for outgoing messages

JMSCorrelationID	The correlation ID for the message.
JMSExpiration	The message's expiration value.
JMSDeliveryMode	Whether or not the message is persistent.[15]
JMSPriority[16]	The message priority level.
JMSReplyTo	The Destination object to which a reply to this message should be sent.

Properties for incoming messages

JMSCorrelationID	The correlation ID for the message.
JMSExpiration	The message's expiration value.
JMSDeliveryMode	Whether or not the message is persistent.
JMSPriority	The message priority level.
JMSTimestamp	The timestamp the broker added to the message.

14 http://download.oracle.com/javaee/5/api/javax/jms/Message.html
15 For sending persistent messages please see documentation for IMessageProducer
16 Clients set the JMSPriority not directly, but either on the producer or as a parameter in the Send method

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

Reserved property names

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

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

Examples

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

Prefix for custom headers

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

Selectors

Selectors are a way of attaching a filter to a subscription to perform content based routing. For more documentation on the detail of selectors see the reference on javax.jmx.Message[17].

Supported message brokers

Message selectors are supported by

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

Code example

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

All supported brokers allow supports string type properties and operations in selectors. ActiveMQ also allows integer properties and operations in selectors (see special note[18]).

17 http://docs.oracle.com/javaee/5/api/javax/jms/Message.html
18 http://activemq.apache.org/selectors.html

Map Messages

Introduction

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

Usage Example

Create a map message and add map entries:

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

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

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

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

Enumerate map entries:

```
MapKeys := MapMessage.GetMapNames;

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

Map Message Transformer

To send and receive map messagers, 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 and object message transformation. This interface defines two methods, **ConsumerTransform** and **ProducerTransform**.

Interface

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


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

Implementation guide for map messages:

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

Code example

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

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

  Connection.Start;

  // send / receive messages

finally
  Connection.Close;
end;
```

Transformation Identifier

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

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

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

Code example

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

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

Producer.Send(MapMessage);
```

Example ProducerTransform implementation with TStrings

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

Notes:

- the method uses a method of a helper interface, IContentProvider.SetContent, to write the body content
- the method returns nil if the passes message is no map message

Code example

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

  if Supports(AMessage, IMapMessage, TmpMapMsg) then
  begin
    MapStrings := TStringList.Create;
    try
      Keys := TmpMapMsg.GetMapNames;

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

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

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

Object Messages

> "Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time."[19]

Introduction

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

Object Message Transformer

To send and receive object messagers, 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 and object message transformation.

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

Code example

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


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

Implementation guide for map messages:

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

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

- Note: only one object message transformer may be active for one connection

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

Simplified API

New interface types

The new API[20] is based on three new interfaces which reduce the amount of client code:

- IMQContext
- IMQProducer
- IMQConsumer

IMQContext interface

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

Code example

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

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

IMQProducer interface

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

20 Since version 6.0

Code example

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

IMQConsumer interface

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

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

Source code example

Code example

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

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

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

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

Stomp 1.2

Connection configuration

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

All Parameters are case sensitive.

Parameters can be omitted to use the default value.

Switch	Description	Default
connect.accept-version[21]	Supported Stomp versions in ascending order	Broker specific, see below
connect.host[22]	**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[23]	Heart beat (outgoing, incoming)	0,0

Default Stomp version (broker-specific)[24]

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

Default Stomp version			
ActiveMQ	Artemis	OpenMQ	RabbitMQ
1.2	1.2	1.0	1.2

21 http://stomp.github.com//stomp-specification-1.2.html#protocol_negotiation
22 http://stomp.github.com//stomp-specification-1.2.html#CONNECT_or_STOMP_Frame
23 http://stomp.github.com//stomp-specification-1.2.html#Heart-beating
24 Since version 5.1 (2017.06)

Connection Factory Code Example:

Code example

```
Factory := TBTConnectionFactory.Create(
  'stomp://localhost:61613?connect.accept-version=1.2&connect.heart-beat=1000,0');
```

This example creates a connection factory with these connection settings

host: localhost

port: 61613

accept-version: 1.2

heart-beat: 1000,0

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

Specification

For details see the Stomp specification pages:

http://stomp.github.com//stomp-specification-1.1.html

http://stomp.github.com//stomp-specification-1.2.html

Sending heart-beat signals

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

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

Code example

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

Notes:

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

Checking for incoming heartbeats

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

Code example

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

Notes:

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

Reading server-side heartbeats

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

This method takes one argument, TimeOut.

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

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

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

Example Applications

Directory	Description
common	Shared units (see below)
common-consumertool	Receive messages from broker
common-consumertool-fpc	Free Pascal version of ConsumerTool
common-producertool	Send messages to broker
common-producertool-fpc	Free Pascal version of `producertool`
common-producertool-ssl	Send messages to broker with SSL connection
common-tests	DUnit tests
common-tests-fpc	FPCUnit tests
delphichat	Simple chat client (Delphi 2009)
heartbeat-server	Uses server-side heart-beating to check the connection / server health [25]
performance	Multi-threaded performance test application (Delphi 2009)
reconnect	Send messages and reconnect on connection failure
rpc	Use temporary queues to implement request/response style communication (not supported on all message brokers[26])
textmessage	Simple text message example
throughput	Produces and consumes messages continuously
throughput-fpc	Free Pascal version of `throughput`
transactions	Transaction example
tutorial1	Tutorial one
tutorial2	Tutorial two

Table 4: Example Applications (in alphabetic order)

25 Requires STOMP 1.2; not supported by OpenMQ
26 Not available with ActiveMQ Artemis message broker

Shared units for demo projects

The directory `demo/common` contains shared units:

- connection configuration form
- command line parameter support class
- LoggingHelper example unit (see "Logging with SLF4P" on page 61)

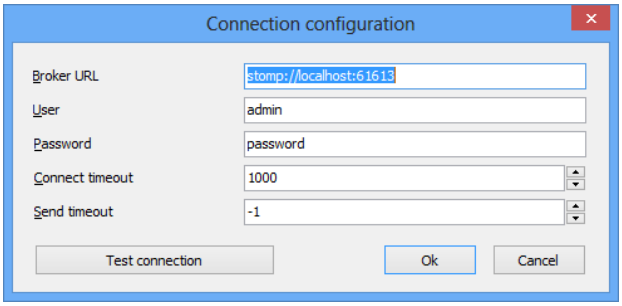

Illustration 2: Connection configuration dialog example

ConsumerTool

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

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

Table 5: ConsumerTool Command Line Options

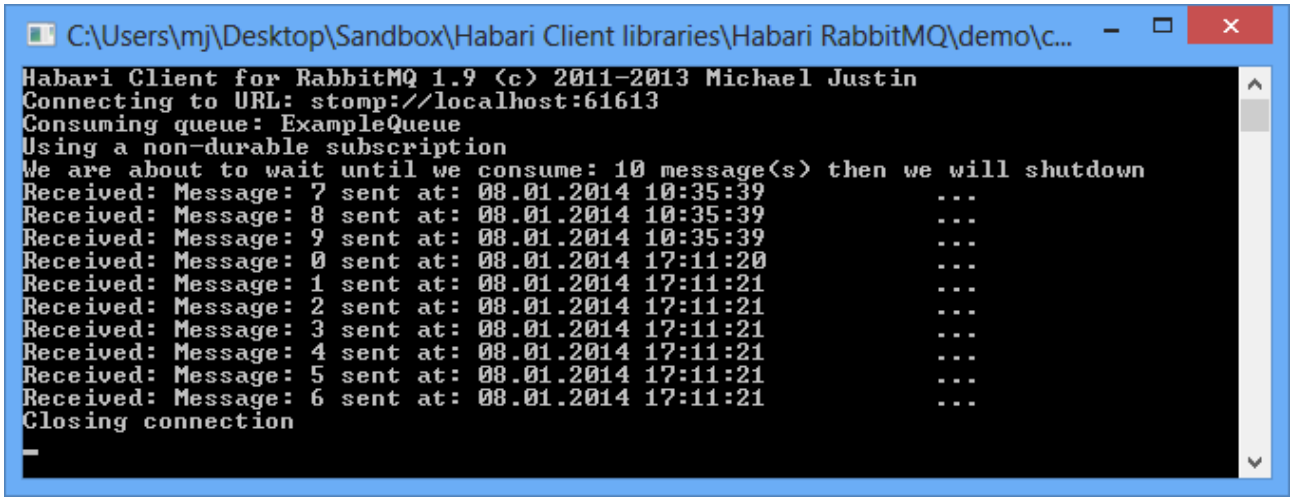

Illustration 3: ConsumerTool demo application

Examples

Receive 1000 messages from local broker

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

Receive 10 messages from local broker and wait for any key

```
ConsumerTool --PauseBeforeShutDown
```

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

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

ProducerTool

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

Parameter	Default	Description
MessageCount	10	Number of messages
MessageSize	255	Length of a message in bytes
Persistent	false	Delivery mode 'persistent'
SleepTime	0	Pause between messages in milliseconds
Subject	TOOL.DEFAULT	Destination name
TimeToLive	0	Message expiration time
Topic	false	Destination is a topic
Transacted	false	Use a transaction
URL	localhost	Message broker URL
Verbose	true	Verbose output
User		User name
Password		Password

Table 6: ProducerTool Command Line Options

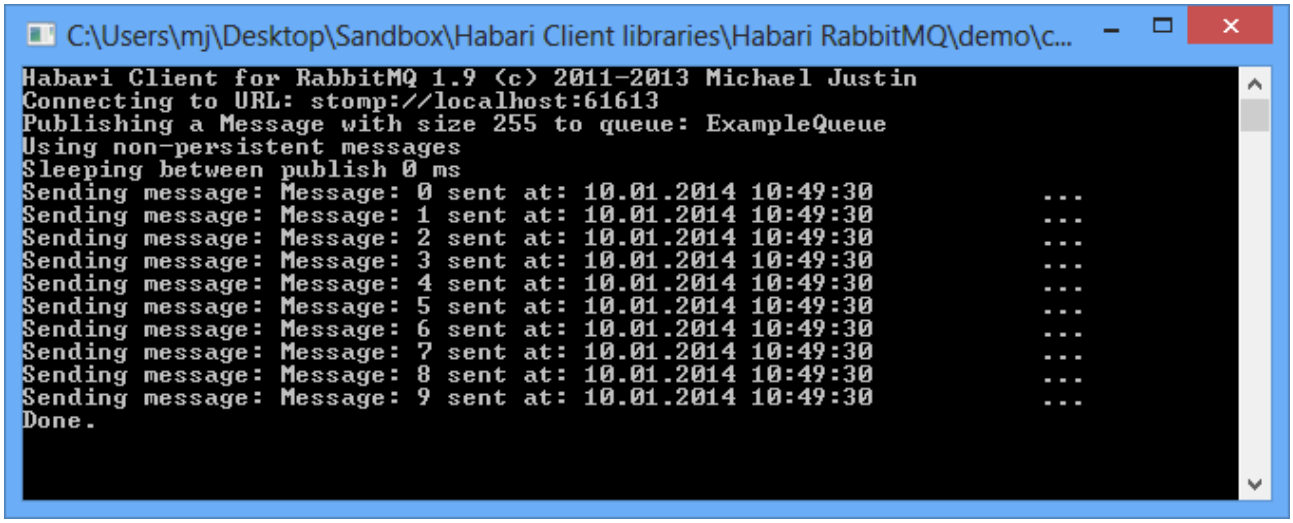

Illustration 4: ProducerTool demo application

Examples

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

```
ProducerTool –-MessageCount 10000
```

Send 10 messages to the topic ExampleTopic on the local broker

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

Performance test

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

- A broker configuration dialog can be invoked by clicking the URL field
- The communication library (Indy or Synapse) can be selected
- 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.

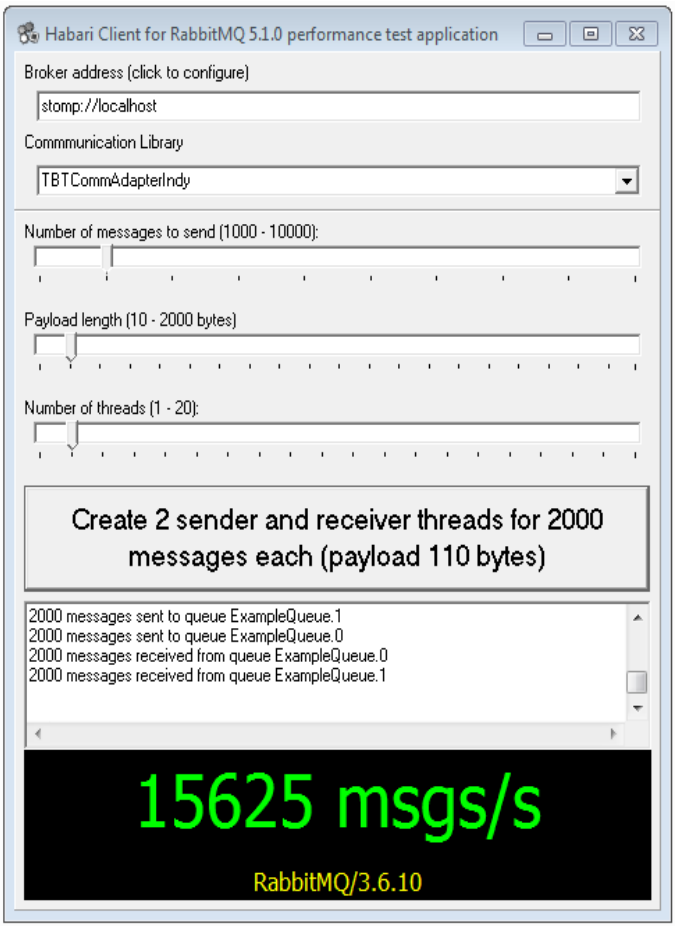

Illustration 5: Performance Test Application

Habari Client for ActiveMQ 5.1 includes an enhanced performance test application, which optionally collects message rates of multiple test runs and displays the sample median. Shown above is an example for a client configuration:

- 21 test runs (triggered by a shift-click on the test button)
- 2000 messages per thread
- 210 bytes payload
- two producer threads, two consumer threads

To start the long-running tests, shift-click on the run button. Taking all test samples takes around ten seconds.

Throughput test

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

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

Table 7: Throughput Test Tool Command Line Options

Examples

Use remote broker 'mybroker' and specify user and password

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

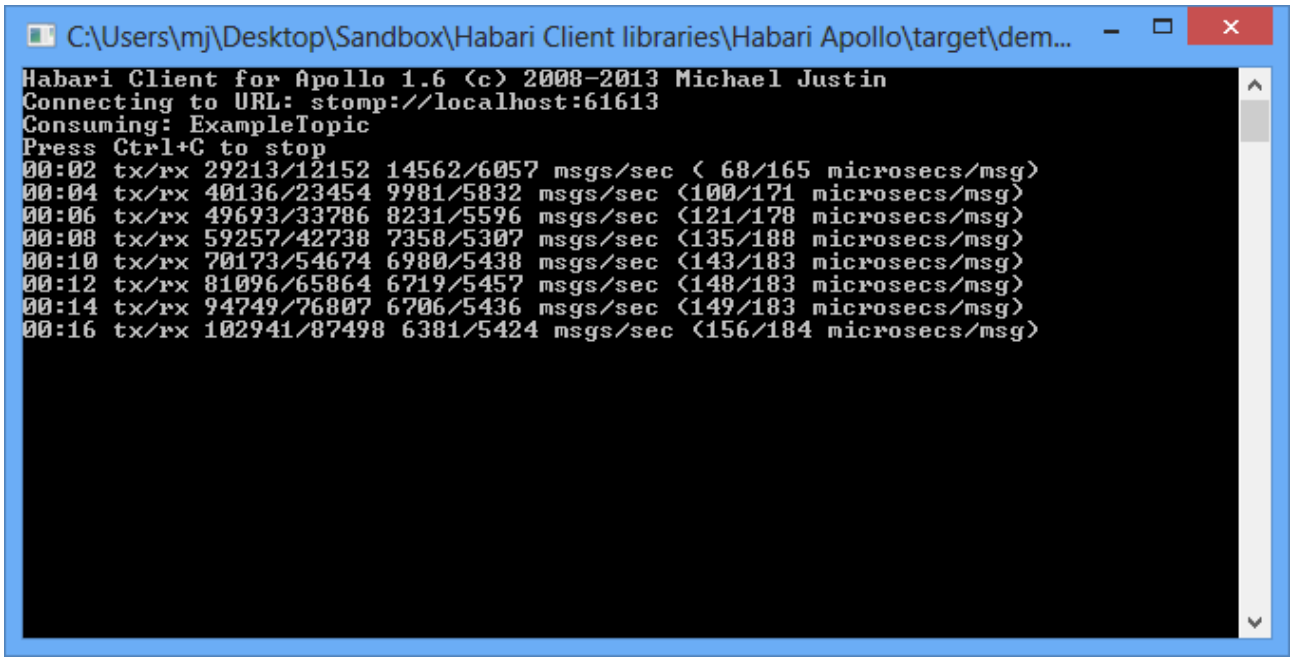

Illustration 6: Throughput test tool output

Unit Tests

Introduction

Habari Client libraries include DUnit and FPCUnit tests. They require the classic DUnit framework (included in Delphi 2009) or FPCUnit (included in Lazarus 2.6).

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

Test project configuration

Logging

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

Raw message logging

To switch on raw logging, add the conditional symbol HABARI_RAW_TRACE and rebuild the project. The project has the {$APPTYPE CONSOLE} flag, which will cause a console window to open.

Optional units

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

Synapse communication adapter

To switch from Indy to Synapse for the tests, add the conditional symbol HABARI_TEST_SYNAPSE and rebuild the project.

Test units

The common-tests folder contains these units

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

Unit tests	
ApiTests	Tests Habari Client core API methods – part 1
BasicTests	Tests Habari Client core API methods – part 2
BrokerExtensionsTests	Tests broker-specific features and extensions of the STOMP protocol
HabariExtensionsTests	Tests non-standard features provided by the Habari Client library
HabariTypesTests	Tests internal data types
ObjectExchangeTests[27]	Tests object message exchange (for Delphi DUnit only)
Stomp12Tests	Tests features introduced with version 1.2 of the STOMP standard
StubServerTests	Tests using a simple local Stomp server

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

Test execution

Requirements

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

Test destinations

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

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

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

27 only added to the test suite if TEST_OPTIONAL_UNITS is defined

STOMP 1.2

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

Logging with SLF4P

Introduction

Habari Client libraries include the free open source logging framework SL4FP as an optional dependency.

SLF4P is available at https://github.com/michaelJustin/slf4p

IDE and project configuration

In order to compile with SLF4P support,

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

Delphi

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

Lazarus

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

LoggingHelper unit

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

Code example

```
uses
  LoggingHelper,
  ...
begin
  // set up logging
  LoggingHelper.ConfigureLogging;
```

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

Code example

```
unit LoggingHelper;

interface

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

const
  DEFAULT_LOG_LEVEL = 'info';

procedure ConfigureLogging(const LogLevel: string = DEFAULT_LOG_LEVEL);

implementation

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

end.
```

Conditional Symbols

Caution

All conditional symbols enable experimental or optional features, which are not covered by the free basic support plan. Feedback (suggestions for improvements, feature requests, and bug reports) are always welcome.

Conditional symbols for release builds

HABARI_ALLOW_UNKNOWN_URL_PARAMS

Disables strict connection URL parameter checking.

If this symbol is defined, connection URLs may contain arbitrary parameters. By default, the library only accepts well-known connection parameters and raises an exception for unknown parameters.

Broker versions: all broker versions.

HABARI_LOGGING

Enables logging support. Requires the open source SLF4P logging facade.

Broker versions: all broker versions.

See also: Logging with SLF4P

HABARI_SSL_SUPPORT

Enables SSL support. Support for SSL connections is an advanced / optional feature, technical support is not included in the basic support plan.

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.

Broker versions: all broker versions.

See also: SSL/TLS Support

HABARI_USE_INTERCEPT

Enables detailed logging of Stomp message frames

This uses the Indy interceptor implementation in unit IdInterceptSimLog.

All communication data will be logged to a file. A new file will be created for every new STOMP connection. The file is located in a folder below the current working directory.

If this symbol is defined in a release build, a compiler warning will be emitted:

```
HABARI_USE_INTERCEPT should not be used for release builds
```

Broker versions: all broker versions.

Indy communication adapter only

Note: this feature requires permissions

- create a directory in the current directory if it does not exist
- create files

Conditional symbols for unit test projects

HABARI_TEST_OPTIONAL_UNITS

Enables tests for experimental / optional units.

HABARI_TEST_SYNAPSE

Enables Synapse communication adapter in DUnit/FPCUnit tests, default is Indy.

Supported for: all versions.

HABARI_TEST_USE_MGMT_API

Enables additional test steps

If this symbol is defined, a broker-specific management client will be used to perform one or more of these actions:

- create destinations on the message broker (test preparation)
- destroy destinations on the message broker (cleanup)
- check destinations for their pending message count

Actual actions depend on the message broker type, see HabariTestCase unit source code for details.

Only available with the DUnit test suite, not for FPCUnit.

Available since version 5.2.0 (2017.10)

Status: This is work in progress / experimental

Broker versions: Apache ActiveMQ, Apache ActiveMQ Artemis and RabbitMQ. For OpenMQ, a “no op” client will be used to keep the test source code compatible between all broker versions.

SSL/TLS Support

SSL communication adapter classes

Habari 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.

Adapter Class	Unit
TBTCommAdapterIndySSL	BTCommAdapterIndySSL
TBTCommAdapterSynapseSSL	BTCommAdapterSynapseSSL

Table 8: Communication Adapters with SSL Support

Mixed Use

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

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

SSL configuration

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

Indy SSL Demo

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

Code example

```
program ProducerToolIndySSL;

{$APPTYPE CONSOLE}

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

begin
  BTAdapterRegistry.SetDefaultAdapter(TBTCommAdapterIndySSL);

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

Notes

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

Example output

```
Habari Client for RabbitMQ 5.1.0 (c) 2008-2017 Michael Justin
Connecting to URL: stomp+ssl://localhost:61614
Publishing a Message with size 255 to queue: ExampleQueue
Using persistent messages
Sleeping between publish 0 ms
313 INFO habari.TBTCommAdapterIndySSL - Verifying SSL certificate
313 INFO habari.TBTCommAdapterIndySSL - Issuer: /C=GB/ST=Greater Manchester/L=Sa
lford/O=COMODO CA Limited/CN=COMODO RSA Domain Validation Secure Server CA
313 INFO habari.TBTCommAdapterIndySSL - Not After: 09.04.2018 01:59:59
313 INFO habari.TBTCommAdapterIndySSL - Verifying SSL certificate
313 INFO habari.TBTCommAdapterIndySSL - Issuer: /C=GB/ST=Greater Manchester/L=Sa
lford/O=COMODO CA Limited/CN=COMODO RSA Domain Validation Secure Server CA
313 INFO habari.TBTCommAdapterIndySSL - Not After: 09.04.2018 01:59:59
329 INFO habari.TBTStompClient - Connected with RabbitMQ/3.6.10 using STOMP 1.2
Sending message: Message: 0 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 1 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 2 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 3 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 4 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 5 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 6 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 7 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 8 sent at: 28.06.2017 10:26:43           ...
Sending message: Message: 9 sent at: 28.06.2017 10:26:43           ...
Done.
```

Support

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

Useful Units

BTStreamHelper unit

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

Code example

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

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

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

  Size := Length(Msg.Content);

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

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

BTJavaPlatform unit

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

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

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

Library Limitations

MessageConsumer

How do I implement synchronous receive from multiple destinations?

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

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

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

Message properties

Only string data type supported by Stomp

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

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

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

Broker-specific exceptions

Apache ActiveMQ 5.6 introduced support for numeric expressions in JMS selectors8.

Multi threading

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

Free Pascal specific restrictions

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

Broker-specific limitations

Transacted Sessions

Transactional acknowledging

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

Other broker specific limitations

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

Frequently Asked Questions

Technical questions

Why am I getting 'undeclared identifier IndyTextEncoding_UTF8'?

Short answer

Your Indy version is too old.

Long answer

The library requires a current Indy 10.6.2 version.

Solution

Please download a newer Indy version.

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

Short answer

Your Synapse version does not support your version of Delphi

Long answer

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

Solution

Please use Indy instead of Synapse or use a compatible version of Synapse.

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

Short answer

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

Long answer

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

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

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

Solution

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

Example

Here is a small code example which causes this error[28]:

Code example

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

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

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

Does the library support non-Unicode Delphi versions?

Short answer

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

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

Long answer

The library makes uses of language features which have been added in Delphi 2009 / Free Pascal 3.0.4. Support for non-Unicode Delphi ended in April 2017.

How can the client application detect network connection loss?

Short answer

Use Stomp heart-beating

Long answer

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

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

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

For more details, please check the paragraph “Reading server-side heartbeats” on page 48.

Online Resources

Third-party libraries

Indy

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

Project home	https://www.indyproject.org/
GitHub	https://github.com/IndySockets

SLF4P

SLF4P is a simple logging facade for Object Pascal, developed with Dephi 2009 and Lazarus 2.0. Tested with DUnit and FPCUnit.

Project home	https://github.com/michaelJustin/slf4p

JsonDataObjects

JsonDataObjects is a JSON parser for Delphi 2009 and newer

GitHub	https://github.com/ahausladen/JsonDataObjects

Synapse

Project home	http://synapse.ararat.cz
Subversion	http://svn.code.sf.net/p/synalist/code/trunk/

Specifications

Stomp – Simple (or Streaming) Text Oriented Messaging Protocol[29]

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

Online articles

Title	Broker
Firebird Database Events and Message-oriented Middleware[30]	All
Discover ActiveMQ brokers with Delphi XE4 and Indy 10.6[31]	ActiveMQ
Official RabbitMQ Management REST API Documentation[32]	RabbitMQ
How to use the RabbitMQ Web-Stomp Plugin[33]	RabbitMQ
RPC with Delphi client and Java server using RabbitMQ[34]	RabbitMQ

29 http://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol
30 https://mikejustin.wordpress.com/2012/11/06/firebird-database-events-and-message-oriented-middleware/
31 https://mikejustin.wordpress.com/2013/07/07/discover-activemq-brokers-with-delphi-xe4-and-indy-10-6/
32 https://mikejustin.wordpress.com/2012/10/26/official-rabbitmq-management-rest-api-documentation/
33 https://mikejustin.wordpress.com/2013/11/27/how-to-use-the-rabbitmq-web-stomp-plugin-with-delphi-and-free-pascal/
34 https://mikejustin.wordpress.com/2013/05/21/rpc-with-delphi-client-and-java-server-using-rabbitmq/

Online Videos

Title	Broker
Introduction to Messaging With Apache ActiveMQ[35]	ActiveMQ
GlassFish Message Queue – High Availability Clusters[36]	OpenMQ

35 http://vimeo.com/12654513
36 http://www.youtube.com/watch?v=RHUJBsy3udU

Support

Bug reports and support inquiries

Please send bug reports and support inquiries to Habarisoft and specify your message broker type and version.

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

Advanced support

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

Broker-specific notes

Authentication plugin

To enable a simple authentication plugin, add these lines to the <plugins> element in the broker configuration:

```
<simpleAuthenticationPlugin>
    <users>
        <authenticationUser username="system" password="manager"
            groups="users,admins"/>
        <authenticationUser username="user" password="password"
            groups="users"/>
        <authenticationUser username="guest" password="password" groups="guests"/>
    </users>
</simpleAuthenticationPlugin>
```

Subscription options

As documented on http://activemq.apache.org/stomp.html, ActiveMQ supports broker-specific arguments which can be passed with the STOMP SUBSCRIBE command.

These arguments can be passed in the CreateQueue command.

Code example

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

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

Selectors

Using numeric selectors to filter messages

Apache ActiveMQ 5.6 introduced support for numeric expressions in selectors[37]. See http://activemq.apache.org/selectors.html for STOMP-specific requirements to support numeric selectors.

37 https://issues.apache.org/jira/browse/AMQ-1609

Using XPath to filter messages

Apache ActiveMQ supports XPath based selectors when working with messages containing XML bodies.

Code example

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

This XPATH expression matches all documents with a “title” root element which has a lang attribute with the value “en”, for example:

```
<title lang=”en”>hello xpath</title>
```

Object Messages

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.[38] 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.[39]

“Delphi Only” vs. “Cross-Language” Object Exchange

Habari Client for ActiveMQ offers two object exchange methods.

Cross-Language	**Objects on the broker are encoded using Java binary serialization.** The Delphi application sends a JSON or XML serialized object to the broker, who transforms it into a binary serialized Java object first before it can be consumed by JMS clients. This broker-side transformation requires that Java class files for the class are in the broker's class path.
Delphi Only	**Objects on the broker are encoded using JSON or XML.** The message broker exchanges the objects between Delphi (and other) clients serialized as JSON or XML text, no special Java support files are required.

38 http://java.sun.com/developer/technicalArticles/Programming/serialization/
39 http://activemq.apache.org/message-transformation.html

“Cross-Language” Object Exchange

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)`[40] 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 methods requires that a JAR containing a matching Java class file has to be deployed in the message broker, which will be used by the brokers message transformer. If this Java class is not compatible with the JSON or XML structure, the message transformation fails!

Pros

- Java clients do not need any special modifications to exchange objects with non-Java clients, Delphi 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 brokers classpath)
- The transformation fails if the Java class and Delphi class declaration don't match
- The transformation fails if the Delphi and Java transformer libraries (JSON / XML) are not compatible

“Delphi Only” Object Exchange

There are almost no differences of the Delphi code for “Cross-Language” and “Delphi Only” object exchange methods.

Switching to “Delphi Only” object exchange requires only an additional property assignment on the object message.

The serialized objects will be stored in the messages 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

40 http://download.oracle.com/javaee/1.4/api/javax/jms/Session.html#createObjectMessage%28java.io.Serializable%29

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

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.

Broker Specific Demos

Directory	Description
activemq-advisory	Example for advisory messages.
activemq-schedule	Example code for "Delay and Schedule Message Delivery" (p. 86).
activemq-statistics	Example code for "Broker Statistics Example" (p. 84)
jms-mapmessage	
jms-objectmessage	
loadbalancing	The **LoadServer** application will connect with ActiveMQ on localhost and create a directory for outgoing files. Copy a file to the files directory. The application will now send it every five seconds to a ActiveMQ queue, including the file name, file size and a sequence number. (For safety reasons in this demo, the file will not be deleted.) The **LoadClient** application will connect with ActiveMQ and create a directory for incoming files. If the application finds a file, it will be downloaded with a filename including a time stamp. If you start the application multiple times, ActiveMQ will distribute the files to all running clients. Requires Jedi Code Library (JCL)

Table 9: Advanced Demo Applications

Broker Statistics Example

ActiveMQ supports Broker plugins, which allows the default functionality to be extended, and new with version 5.3 of Apache ActiveMQ is a Statistics plugin, which enables statistics about the running broker, or Queues and Topics to be queried.

The statistics plugin looks for messages sent to particular destinations. To query the running statistics of a 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 a 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.DestinationTest.Foo`.

You can also use wildcards too, 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.Destinationexample.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
```

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.[41]

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 to wait 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.

41 http://activemq.apache.org/delay-and-schedule-message-delivery.html

Known broker-specific bugs and failures

Free Pascal 3.0 bug

- A Free Pascal 3.0.0 bug in LocalTimeToUniversal causes wrong message expiration timestamp values[42]. The bug is fixed in Free Pascal 3.0.2.

42 http://mantis.freepascal.org/view.php?id=29176

Index

Reference

Table Index

Illustration Index