Getting started with BTActiveMQClient

Michael Justin

A short guide for the first steps with the JMS client library

Contents

Introduction

About BTActiveMQClient

BTActiveMQClient for Delphi™ is a library which provides easy access to the Apache ActiveMQ messaging system.

With BTActiveMQClient, applications can connect to Apache ActiveMQ servers, subscribe to queues and topics, send and receive messages and objects, and work with transactions.

How Can I Use It?

Here are some examples for software solutions built on top of a Message Broker like Apache ActiveMQ:

- **Intranet News Ticker Application**: using the publish and subscribe communication model, news can be delivered to all registered client applications. The message sender works like a broadcast station, and does not care if clients don't listen.
- **Load Balancing**: using the point-to-point or queuing model, many 'worker' applications can be installed on different computers. Every new message sent to the queue will be delivered only to one client. The server will keep messages until they are expired or delivered to a client.
- **Persistent Storage**: messages and objects can be stored in the Object Broker and retrieved even after a restart.
- **Interprocess Communication**: applications can use point-to-point messages to exchange information between each other even if the receiver currently is not running.

Limitations

- The ActiveMQ Stomp connector does not support password authentication in ActiveMQ versions before 5.1, development snapshots of Apache ActiveMQ 5.1 are already available. For more information please visit http://activemq.apache.org/stomp.html
- Internet Direct (Indy) for Free Pascal is still in development (see "Indy 10 Lazarus/FreePascal Port", http://www.indyproject.org/Sockets/fpc/)

- OverByte ICS and TClientSocket communication adapters do not support console applications, they do not compile under Free Pascal and currently do not allow message receiving

---

About Apache ActiveMQ

Apache ActiveMQ is the most popular and powerful open source Message Broker and Enterprise Integration Patterns provider.

Apache ActiveMQ is fast, supports many Cross Language Clients and Protocols, comes with easy to use Enterprise Integration Patterns and many advanced features while fully supporting JMS 1.1 and J2EE 1.4.

- Read more about Apache ActiveMQ here: **http://activemq.apache.org/**

License

The library includes royalty-free distribution rights and upgrades for one year.
Full license text

Communication Adapters

Feature Matrix

	Features	Indy 10	Synapse	ICS V6	TClient-Socket
D	Send/receive **text** messages in Delphi GUI applications	Yes/Yes	Yes/Yes	Yes/(-)	Yes/(-)
	Send/receive **binary** messages in Delphi GUI applications	Yes/Yes	Yes/Yes	Yes/(-)	Yes/(-)
	Send/receive **text** messages in Delphi CONSOLE mode applications	Yes/Yes	Yes/Yes	[1]	[1]
	Send/receive **binary** messages in Delphi CONSOLE mode applications	Yes/Yes	Yes/Yes	[1]	[1]
FP	Send/receive **text** messages in Free Pascal GUI applications	Yes/Yes	Yes/Yes	[2]	[2]
	Send/receive **binary** messages in Free Pascal GUI applications	Yes/Yes	Yes/Yes	[2]	[2]
	Send/receive **text** messages in Free Pascal CONSOLE mode applications	Yes/Yes	Yes/Yes	[2]	[2]
	Send/receive **binary** messages in Free Pascal CONSOLE mode applications	Yes/Yes	Yes/Yes	[2]	[2]

[1] Console mode is not supported by design - these components require Windows Handles

[2] Free Pascal is not supported by these components

Roadmap

1.0 Release

- Include documentation for basic message sending and receiving
- Receive messages with ICS and TClientSocket
- **done:** Rewrite Header data type
- **done:** Add binary message support (use **content-length** header) **(Indy and Synapse)**
- **done:** Include Free Pascal compiler projects in build script
- **done:** Add runtime selection of communication library
- **done:** Support object exchange over SOAP
- **done:** Include object exchange demo
- **done:** Include Free Pascal example applications with source
- **done:** Extend IPrimitiveMap implementation with strong typed Get/Set methods
- Do not use STOMP-specific destination name prefixes like /queue and /topic
- **done:** Add connection- and session based interface
- Check all in-source 'to do' entries
- Add Habari Express component and documentation

1.1 Release

- Remove need for Connect - loop

1.2 Release

- Provide a method to consume exactly one message
- Support ActiveMQ Administration messages

- Support ActiveMQ extensions to JMS

1.x Release

- Support REST protocol
- Add MIME encoding
- Extend logging framework, support for format strings and Exceptions
- Queue/Topic Subscription property (designtime editing, TCollection based)
- SOAP message exchange examples (asynchronous SOAP)

Installation

Requirements

- Borland/CodeGear Delphi 6 or higher
- Apache ActiveMQ version 4 or 5
- A TCP/IP communication library (see below)

Important Notes

- Sending and receiving of objects requires Delphi 7 or higher. The library is designed to use methods that were added to TRemotable: ObjectToSOAP and SOAPToObject. These methods are available since Delphi 7.
- Sending and receiving of objects in FreePascal requires the Web Service Toolkit (http://wiki.lazarus.freepascal.org/Web_Service_Toolkit) or binary serialization.

More information on object exchange will be available in the release version.

Supported Communication Libraries

BTActiveMQClient now supports the following communication libraries out of the box.

Indy 10.2

The Indy communication library is available at www.nevrona.com/indy

The Free Pascal version of Indy is available as a separate download: "Indy 10 Lazarus/FreePascal Port" http://www.indyproject.org/Sockets/fpc/

Indy supports both GUI-based and console mode applications.

Synapse

The Synapse communication library is available at synapse.ararat.cz

Synapse supports both GUI-based and console mode applications.

Overbyte ICS V6 for Delphi 7 and up

The ICS communication library is available at www.overbyte.be

TClientSocket and ICS will only work in GUI-based applications.

TClientSocket

This socket communication component is included in Delphi. TClientSocket has been declared deprecated by Borland / CodeGear.

TClientSocket and ICS will only work in GUI-based applications.

Download Apache ActiveMQ

Apache ActiveMQ http://activemq.apache.org/download.html

Upgrades

If you upgrade from older versions, make a backup of your existing version and make sure that you also have a backup of your own source codes.

If you upgrade from old versions, component properties may have changed and this could cause error messages when you open existing projects with the new version installed.

Component Installation in Delphi

Habari Express

Habari Express provides simplified methods to exchange messages using an Apache ActiveMQ Message Broker.

To install the Habari Express component in the Delphi component palette, follow these steps:

1. Create a new Delphi Package Project
2. Add the file btHabariExpress_reg.pas
3. Install the package

The component will appear in a new palette page with the title 'habari'.

Component installation is optional. Runtime creation and usage of the component is possible without disadvantage.

ActiveMQ Configuration

Enabling the ActiveMQ Broker for Stomp

In the Apache ActiveMQ default configuration, Stomp is already enabled.

ActiveMQ supports the Stomp protocol and the Stomp - JMS mapping. This makes it easy to write a client in pure Ruby, Perl, Python or PHP for working with ActiveMQ. Please see the Stomp site for more details.

For configuration details see http://activemq.apache.org/stomp.html

Send and Receive Messages

Send Text Messages

Source code for a simple application which sends a test message:

TODO use THabariExpress for this example.

```
program SendOneMessage;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  BTJMSClient in '..\..\source\BTJMSClient.pas',
  BTCommAdapterIndy in '..\..\source\BTCommAdapterIndy.pas';

var
  Conn: TBTJMSClient;

begin
  Conn := TBTJMSClient.Create(nil);

  try
    Conn.Connect;
    Conn.Logger.Info('Send a message');
    Conn.SendText('/queue/onemessage', 'This is a test message');
    Conn.Disconnect;
    WriteLn('Hit any key');
    ReadLn;
  finally
    Conn.Free;
  end;

end.
```

Used Units

BTJMSClient contains the TBTJMSClient class which provides the SendText method.

BTCommAdapterIndy contains the Indy communication adapter class. By including this unit, it will register the adapter class with an internal list of all

available communication adapters. If the program reaches the line where the TBTJMSClient instance is created and assigned to the variable **Conn**, the Indy adapter is already registered and an instance of this class will be created and used as the communication adapter for this connection.

Note The sequence of units which register a communication adapter class is important! The source code for the GUI demo shows how one of the registered communication adapters can be selected at runtime.

Connect to the Server

With the line

```
Conn.Connect;
```

the client connects to the server.

Send the Message

The line

```
Conn.SendText('/queue/onemessage', 'This is a test message');
```

sends the text message to the queue 'onemessage' on the ActiveMQ server.

Send Binary Messages

The GUI demo includes an option to send binary files as JMS messages. The following code uses TFileStream and TStringStream to load the selected file into the memory and the Send method of the TBTJMSClient instance Conn to transmit the file content.

Note that the BytesMessage object is declared with the type IBytesMessage, so the allocated memory for this object will be released automatically (without Free).

```
procedure TDemoMainForm.SendFile(Sender: TObject);
var
  FileStream: TFileStream;
  S: TStringStream;
  BytesMessage: IBytesMessage;
begin
  if not OpenDialog1.Execute then
  begin
```

```
    Exit;
  end;

  S := TStringStream.Create('');
  try
    FileStream := TFileStream.Create(OpenDialog1.FileName,
fmOpenRead or fmShareDenyWrite);

    try
      S.CopyFrom(FileStream, FileStream.Size);

      BytesMessage := TBTJMSBytesMessage.Create;
      BytesMessage.Content := S.DataString;

      Conn.Send(BytesMessage, Destination);

    finally
      FileStream.Free;
    end;
  finally
    S.Free;
  end;

end;
```

Receive Text Messages

To receive text messages, the client has to subscribe to a queue or topic on the server. The messages will be delivered asynchronous to an event handler which has to be provided by the client.

```
var
  Destination: IDestination;
  Consumer: IMessageConsumer;

begin
 ...
  // create a destination
  Destination := Conn.CreateQueue(Dest);

  // create a consumer
  Consumer := Conn.CreateConsumer(Destination, Listener);

  ...
end;
```

The second CreateConsumer parameter is a reference to an object which implements the IMessageListener interface. This interface only contains one procedure, OnMessage:

```
  IMessageListener = interface(IInterface)
    procedure OnMessage(Message: IMessage);
  end;
```

In the SubscriberDemo example, the listener is implemented in the connection class (TBTJMSSimpleSubscriber).

Send and Receive Objects

Send Objects using SOAP Serialization

Requirements

Delphi 7 or higher is required for TBTSoapAdapter. The TBTSoapAdapter component needs a TXMLDocument instance which is included in Delphi.

TBTSoapAdapter can serialize and deserialize instances of classes which inherit from TRemotable. All restrictions for Soap serializations for this class apply.

SOAP Adapter Component

The TBTSoapAdapter class can be used for object serialization. To use it, create an instance of this class – for example in the FormCreate method:

```
procedure TSendObjectsForm.FormCreate(Sender: TObject);
begin
  SoapAdapter := TBTSoapAdapter.Create(Self);
  SoapAdapter.XMLDocument := XMLDocument1;
end;
```

Object Serialization

In the Delphi GUI demo, an ExampleObject will be used for the SOAP serialization. Its properties can be entered in a form and will be serialized immediately to its XML string representation. All that is necessary is a call of SoapAdapter.ObjectToSOAP:

```
MemoXML.Text := SoapAdapter.ObjectToSOAP(ExampleObject, 'Name');
```

Send a Serialized Object

The resulting XML string can be sent using the standard JMS send methods.

Receive Objects using SOAP Deserialization

Demo Applications

Simple GUI Demo

The GUI demo provides a simple user interface to some of the core communication functions of the library.

PublisherDemo

The PublisherDemo application will send 2000 messages to the destination /queue/a on the local ActiveMQ server.

SubscriberDemo

The SubscriberDemo application will subscribe to the destination /queue/a on the local ActiveMQ server and wait for 2000 incoming messages.

Stomp Protocol Specification, Version 1.0

Initially the client must open a socket (I'm going to presume TCP, but really it is kind of irrelevant). The client then sends:

```
CONNECT
login: <username>
passcode:<passcode>

^@
```

The `^@` is a null (control-@ in ASCII) byte. The entire thing will be called a Frame in this doc. The frame starts with a command (in this case CONNECT), followed by a newline, followed by headers in a <key>:<value> with each header followed by a newline. A blank line indicates the end of the headers and beginning of the body (the body is empty in this case), and the null indicates the end of the frame.

After the client sends the CONNECT frame, the server will always acknowledge the connection, by sending a frame which looks like:

```
CONNECTED
session: <session-id>

^@
```

The `session-id` header is a unique identifier for this session (though it isn't actually used yet).

At this point there are a number of commands the client may send

- SEND
- SUBSCRIBE
- UNSUBSCRIBE
- BEGIN

- COMMIT
- ABORT
- ACK
- DISCONNECT

Client Commands

SEND

The SEND command sends a message to a destination in the messaging system. It has one required header, **destination**, which indicates where to send the message. The body of the SEND command is the message to be sent. For example:

```
SEND
destination:/queue/a

hello queue a
^@
```

This sends a message to the **/queue/a** destination. This name, by the way, is arbitrary, and despite seeming to indicate that the destination is a "queue" it does not, in fact, specify any such thing. Destination names are simply strings which are mapped to some form of destination on the server - how the server translates these is left to the server implementation. See this note on mapping destination strings to JMS Destinations for more detail.

SEND supports a **transaction** header which allows for transaction sends.

It is recommended that SEND frames include a content-length header which is a byte count for the length of the message body. If a content-length header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

SUBSCRIBE

The SUBSCRIBE command is used to register to listen to a given destination. Like the SEND command, the SUBSCRIBE command requires a **destination** header indicating which destination to subscribe to. Any messages received on the subscription will henceforth be delivered as MESSAGE frames from the server to the client. The **ack** header is optional, and defaults to **auto**.

```
SUBSCRIBE
```

```
destination: /queue/foo
ack: client

^@
```

In this case the **ack** header is set to **client** which means that messages will only be considered delivered after the client specifically acknowledges them with an ACK frame. The valid values for **ack** are **auto** (the default if the header is not included) and **client**.

The body of the SUBSCRIBE command is ignored.

Stomp brokers may support the **selector** header which allows you to specify an SQL 92 selector on the message headers which acts as a filter for content based routing.

You can also specify an **id** header which can then later on be used to UNSUBSCRIBE from the specific subscription as you may end up with overlapping subscriptions using selectors with the same destination. If an **id** header is supplied then Stomp brokers should append a **subscription** header to any MESSAGE commands which are sent to the client so that the client knows which subscription the message relates to. If using Wildcards and selectors this can help clients figure out what subscription caused the message to be created.

UNSUBSCRIBE

The UNSUBSCRIBE command is used to remove an existing subscription - to no longer receive messages from that destination. It requires either a **destination** header or an **id** header (if the previous SUBSCRIBE operation passed an id value). Example:

```
UNSUBSCRIBE
destination: /queue/a

^@
```

BEGIN

BEGIN is used to start a transaction. Transactions in this case apply to sending and acknowledging - any messages sent or acknowledged during a transaction will be handled atomically based on the transaction.

```
BEGIN
transaction: <transaction-identifier>
```

```
^@
```

The **transaction** header is required, and the transaction identifier will be used for SEND, COMMIT, ABORT, and ACK frames to bind them to the named transaction.

COMMIT

COMMIT is used to commit a transaction in progress.

```
COMMIT
transaction: <transaction-identifier>

^@
```

The **transaction** header is required, you must specify which transaction to commit!

ACK

ACK is used to acknowledge consumption of a message from a subscription using client acknowledgment. When a client has issued a SUBSCRIBE frame with the **ack** header set to **client** any messages received from that destination will not be considered to have been consumed (by the server) until the message has been acknowledged via an ACK.

ACK has one required header, **message-id**, which must contain a value matching the **message-id** for the MESSAGE being acknowledged. Additionally, a **transaction** header may be specified, indicating that the message acknowledgment should be part of the named transaction.

```
ACK
message-id: <message-identifier>
transaction: <transaction-identifier>

^@
```

The **transaction** header is optional.

ABORT

ABORT is used to roll back a transaction in progress.

```
ABORT
transaction: <transaction-identifier>

^@
```

The **transaction** header is required, you must specify which transaction to abort!

DISCONNECT

DISCONNECT does a graceful disconnect from the server. It is quite polite to use this before closing the socket.

```
DISCONNECT

^@
```

Standard Headers

Some headers may be used, and have special meaning, with most packets

Receipt

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.

```
SEND
destination:/queue/a
receipt:message-12345

Hello a!^@
```

Server Frames

The server will, on occasion, send frames to the client (in additional to the initial CONNECTED frame). These frames may be one of:

- MESSAGE
- RECEIPT
- ERROR

MESSAGE

MESSAGE frames are used to convey messages from subscriptions to the client. The MESSAGE frame will include a header, **destination**, indicating the destination the message was delivered to. It will also contain a **message-id** header with a unique identifier for that message. The frame body contains the contents of the message:

```
MESSAGE
destination:/queue/a
message-id: <message-identifier>

hello queue a^@
```

Would be a sample message.

It is recommended that MESSAGE frames include a **content-length** header which is a byte count for the length of the message body. If a **content-length** header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte, and if a **content-length** is not specified the first null byte encountered signals the end of the frame.

RECEIPT

Receipts are issued from the server when the client has requested a receipt for a given command. A RECEIPT frame will include the header **receipt-id**, where the value is the value of the **receipt** header in the frame which this is a receipt for.

```
RECEIPT
receipt-id:message-12345

^@
```

The receipt body will be empty.

ERROR

The server may send ERROR frames if something goes wrong. The error frame should contain a **message** header with a short description of the error, and the body may contain more detailed information (or may be empty).

```
ERROR
message: malformed packet received

The message:
-----
MESSAGE
destined:/queue/a

Hello queue a!
-----
Did not contain a destination header, which is required for message
propagation.
^@
```

It is recommended that ERROR frames include a **content-length** header which is a byte count for the length of the message body. If a **content-length** header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte, and if a **content-length** is not specified the first null byte encountered signals the end of the frame.

Demo Application Source Code

PublisherDemo

```
program PublisherDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  BTJMSClient in '..\..\source\BTJMSClient.pas',
  BTStompInterfaces in '..\..\source\BTStompInterfaces.pas',
  BTJMSInterfaces in '..\..\source\BTJMSInterfaces.pas',
  BTStompTypes in '..\..\source\BTStompTypes.pas',
  BTStompConnection in '..\..\source\BTStompConnection.pas',
  BTSupportInterfaces in '..\..\source\BTSupportInterfaces.pas',
  BTCommAdapterIndy in '..\..\source\BTCommAdapterIndy.pas',
  BTCommAdapter in '..\..\source\BTCommAdapter.pas';

const
  Dest = '/queue/a';

  SEND_COUNT = 2000;

var
  Conn: TBTJMSClient;
  I: Integer;
  L: ILogging;
  Compiler: string;

begin
  {$IFDEF FPC}
  Compiler := 'Free Pascal';
  {$ELSE}
  Compiler := 'Borland Delphi';
  {$ENDIF}

  Conn := TBTJMSClient.Create(nil);
```

```
  try
    if ParamCount = 1 then
      Conn.Host := ParamStr(1)
    else
      Conn.Host := 'localhost';

    L := Conn.Logger;
    L.Info(Conn.Version + ' Compiler: '  + Compiler);

    L.Info('Connect to server');
    Conn.Connect;
    Sleep(1500);

    for I := 0 to SEND_COUNT - 1 do
    begin
      L.Info(Format('Sending test message %d to %s on %s...',
[I+1, Dest, Conn.Host]));
      Conn.SendText(Dest, Format('This is test message %d',
[I+1]));
    end;

    Conn.Disconnect;

    L.Info('Hit any key');
    ReadLn;

  finally
    Conn.Free;
  end;

end.
```

SubscriberDemo

```
program SubscriberDemo;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Classes,
  SyncObjs,
  BTJMSClient in '..\..\source\BTJMSClient.pas',
  BTStompInterfaces in '..\..\source\BTStompInterfaces.pas',
  BTJMSInterfaces in '..\..\source\BTJMSInterfaces.pas',
  BTStompTypes in '..\..\source\BTStompTypes.pas',
  BTStompConnection in '..\..\source\BTStompConnection.pas',
  BTSupportInterfaces in '..\..\source\BTSupportInterfaces.pas',
  BTCommAdapterSynapse in
'..\..\source\BTCommAdapterSynapse.pas';
  // BTCommAdapterIndy in '..\..\source\BTCommAdapterIndy.pas';

const
  Dest = '/queue/a';

type

  TBTJMSSimpleSubscriber = class(TBTJMSClient, IMessageListener)
  private
    CS: TCriticalSection;
  public
    I: Integer;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure OnMessage(Message: IMessage);
  end;

const
  EXPECTED = 2000;

var
  Conn: TBTJMSSimpleSubscriber;
  L: ILogging;
  Compiler: string;

  D: IDestination;
  C: IMessageConsumer;
```

```
  { TBTJMSSimpleSubscriber }

constructor TBTJMSSimpleSubscriber.Create(AOwner: TComponent);
begin
  inherited;
  I := 0;
  CS := TCriticalSection.Create;
end;

destructor TBTJMSSimpleSubscriber.Destroy;
begin
  CS.Free;
  inherited;
end;

procedure TBTJMSSimpleSubscriber.OnMessage(Message: IMessage);
begin
  CS.Enter;

  I := I + 1;

  Logger.Info(Format('Message %d received: %s', [I,
ITextMessage(Message).Text]));

  (* if I <= EXPECTED then
    Conn.Ack(JMSMessage.MessageId); *)

  if I = EXPECTED then
  begin
    Conn.Disconnect;
  end;

  CS.Leave;
end;

begin
{$IFDEF FPC}
  Compiler := 'Free Pascal';
{$ELSE}
  Compiler := 'Borland Delphi';
{$ENDIF}

  Conn := TBTJMSSimpleSubscriber.Create(nil);

  try
    if ParamCount = 1 then
      Conn.Host := ParamStr(1)
    else
```

```
      Conn.Host := 'localhost';

    L := Conn.Logger;
    L.Info(Conn.Version + ' Compiler: ' + Compiler);

    Sleep(1500);

    L.Info('Connect to server');
    Conn.Connect;

    L.Info(Format('Wait for %d incoming messages', [EXPECTED]));
    D := Conn.CreateQueue(Dest);
    C := Conn.CreateConsumer(D, Conn);

    while Conn.StompConnected do
    begin
      Sleep(50);
    end;

    L.Info('Hit any key');
    ReadLn;

  finally
    Conn.Free;
  end;

end.
```

Index

Reference