

Eurex Clearing

Eurex Clearing Messaging Interfaces Connectivity B: AMQP Programming Guide

Date	Vers.	Change
12 February 2021	7.1	Removed Qpid C++ from list of broker implementations (chapter 2) Removed information about temporary queues and AMQP 0-10 protocol (chapter 3.3.3.1)
28 May 2021	7.1	Added description of JMS Client ID uniqueness (chapter 3.3.2)

© Eurex Frankfurt AG 2021

Deutsche Börse AG (“DBAG”), Clearstream Banking AG (“Clearstream”), Eurex Frankfurt AG (“Eurex”), Eurex Clearing AG (“Eurex Clearing”), Eurex Securities Transactions Services GmbH (“Eurex STS”) and Eurex Repo GmbH (“Eurex Repo”) are corporate entities and are registered under German law. Eurex Global Derivatives AG is a corporate entity and is registered under Swiss law. Clearstream Banking S.A. is a corporate entity and is registered under Luxembourg law. Eurex Frankfurt AG is the administrating and operating institution of Eurex Deutschland. Eurex Deutschland is in the following referred to as the “Eurex Exchange”.

All intellectual property, proprietary and other rights and interests in this publication and the subject matter hereof (other than certain trademarks and service marks listed below) are owned by DBAG or its affiliates and subsidiaries or used under authorization by their respective owners, including, without limitation, all patent, registered design, copyright, trademark and service mark rights. While reasonable care has been taken in the preparation of this publication to provide details that are accurate and not misleading at the time of publication DBAG, Clearstream, Eurex, Eurex Clearing, Eurex Repo as well as the Eurex Exchange and their respective subsidiaries, servants and agents (a) do not make any representations or warranties regarding the information contained herein, whether express or implied, including without limitation any implied warranty of merchantability or fitness for a particular purpose or any warranty with respect to the accuracy, correctness, quality, completeness or timeliness of such information, and (b) shall not be responsible or liable for any third party’s use of any information contained herein under any circumstances, including, without limitation, in connection with actual trading or otherwise or for any errors or omissions contained in this publication.

This publication is published for information purposes only and shall not constitute investment advice respectively does not constitute an offer, solicitation or recommendation to acquire or dispose of any investment or to engage in any other transaction. This publication is not intended for solicitation purposes but only for use as general information. All descriptions, examples and calculations contained in this publication are for illustrative purposes only.

Eurex and Eurex Clearing offer services directly to members of the Eurex Exchange respectively to clearing members of Eurex Clearing. Those who desire to trade any products available on the Eurex market or who desire to offer and sell any such products to others or who desire to possess a clearing license of Eurex Clearing in order to participate in the clearing process provided by Eurex Clearing, should consider legal and regulatory requirements of those jurisdictions relevant to them, as well as the risks associated with such products, before doing so.

Only Eurex derivatives that are CFTC-approved may be traded via direct access in the United States or by United States persons. A complete, up-to-date list of Eurex derivatives that are CFTC-approved is available at: <https://www.eurex.com/ex-en/rules-regs/eurex-derivatives-us>.

In addition, Eurex representatives and participants may familiarize U.S. Qualified Institutional Buyers (QIBs) and broker-dealers with certain eligible Eurex equity options and equity index options pursuant to the terms of the SEC’s July 1, 2013 Class No-Action Relief. A complete, up-to-date list of Eurex options that are eligible under the SEC Class No-Action Relief is available at: <https://www.eurex.com/ex-en/rules-regs/eurex-derivatives-us/eurex-options-in-the-us-for-eligible-customers>. Lastly, U.S. QIBs and broker-dealers trading on behalf of QIBs may trade certain single-security futures and narrow-based security index futures subject to terms and conditions of the SEC’s Exchange Act Release No. 60,194 (June 30, 2009), 74 Fed. Reg. 32,200 (July 7, 2009) and the CFTC’s Division of Clearing and Intermediary Oversight Advisory Concerning the Offer and Sale of Foreign Security Futures Products to Customers Located in the United States (June 8, 2010).

Trademarks and Service Marks

Buxl[®], DAX[®], DivDAX[®], eb.rexx[®], Eurex[®], Eurex Repo[®], Eurex Strategy WizardSM, Euro GC Pooling[®], FDAX[®], FWB[®], GC Pooling[®], GCPI[®], MDAX[®], ODAX[®], SDAX[®], TecDAX[®], USD GC Pooling[®], VDAX[®], VDAX-NEW[®] and Xetra[®] are registered trademarks of DBAG or its affiliates and subsidiaries. All MSCI indexes are service marks and the exclusive property of MSCI Barra. ATX[®], ATX[®] five, CECE[®] and RDX[®] are registered trademarks of Vienna Stock Exchange AG. IPD[®] UK Quarterly Indexes are registered trademarks of Investment Property Databank Ltd. IPD and have been licensed for the use by Eurex for derivatives. SLI[®], SMI[®] and SMIM[®] are registered trademarks of SIX Swiss Exchange AG. The STOXX[®] indexes, the data included therein and the trademarks used in the index names are the intellectual property of STOXX Limited and/or its licensors. Eurex derivatives based on the STOXX[®] indexes are in no way sponsored, endorsed, sold or promoted by STOXX and its licensors and neither STOXX nor its licensors shall have any liability with respect thereto. PCS[®] and Property Claim Services[®] are registered trademarks of ISO Services, Inc. Korea Exchange, KRX, KOSPI and KOSPI 200 are registered trademarks of Korea Exchange Inc. The names of other companies and third party products may be trademarks or service marks of their respective owners.

Eurex Deutschland qualifies as manufacturer of packaged retail and insurance-based investment products (PRIIPs) under Regulation (EU) No 1286/2014 on key information documents for packaged retail and insurance-based investment products (PRIIPs Regulation), and provides key information documents (KIDs) covering PRIIPs traded on Eurex Deutschland on its website under the following link: <https://www.eurex.com/ex-en/rules-regs/priips-kids>.

In addition, according to Art. 14(1) PRIIPs Regulation the person advising on, or selling, a PRIIP shall provide the KID to retail investors free of charge.

Abstract

This document provides information about the connectivity for the AMQP based Eurex Clearing FIXML Interface, Eurex Clearing FpML Interface and Eurex Clearing Margin Calculator Interface. This document is intended to be a guide to Members developing applications, which will utilize this interface.

Keywords

Eurex Clearing FIXML Interface, Eurex Clearing FpML Interface, Eurex Clearing Margin Calculator Interface, Advanced Message Queuing Protocol, AMQP, Clearing, FIXML, FpML, XML, Development guide, Java, C++, .NET

Table of Contents

1	Introduction	8
1.1	Overview	8
1.1.1	Eurex Clearing FIXML Interface	8
1.1.2	Eurex Clearing FpML Interface	8
1.1.3	Eurex Clearing Margin Calculator Interface	8
1.1.4	Eurex Clearing Trade Entry Interface	8
1.1.5	AMQP	8
1.1.6	FIXML	9
1.1.7	FpML	9
1.2	Intended audience	9
1.3	Eurex Clearing Messaging Interface Connectivity documentation	9
1.4	Eurex Clearing FIXML Interface documentation	10
1.5	Eurex Clearing FpML Interface, Margin Calculator Interface and Trade Entry Interface documentation	10
1.6	Conventions used in this document	11
1.7	Examples used in this document	11
1.8	Organization of this document	11
1.9	Code examples	11
2	Obtaining the AMQP client API	13
2.1	Apache Qpid	13
2.2	SwiftMQ AMQP 1.0 Java client	14
2.3	AMQP.NET Lite client	14
2.4	Vert.x 3	14
2.5	Upgrading client libraries	14
3	Java	15
3.1	SSL / TLS Certificates	15
3.1.1	Client certificate	15
3.1.2	Broker public keys	16
3.1.3	SSL / TLS debugging	17
3.2	JMS	17
3.2.1	Java Naming and Directory Interface (JNDI)	18
3.2.2	Preparing connection and session	18
3.2.3	Creating a receiver/sender	19
3.2.4	Starting the connection	20
3.2.5	Thread safety	20
3.2.6	Receiving/sending messages	20
3.2.6.1	Preparing a request message	20
3.2.6.2	Sending a request message	21
3.2.6.3	Receiving a message using Message listener	21
3.2.6.4	Receiving a message using the receive() method	22

3.2.6.5	Message processing	23
3.2.7	Closing the connection	23
3.2.8	Error handling	24
3.3	Apache Qpid JMS client for AMQP 1.0	25
3.3.1	Connection Factory	25
3.3.2	Connection URI	25
3.3.3	Addresses	27
3.3.3.1	Receiving responses to requests	28
3.3.3.2	Sending requests	28
3.3.3.3	“ReplyTo” address in requests	28
3.3.3.4	Receiving broadcasts	29
3.3.4	Performance	29
3.3.4.1	Receive pre-fetching	29
3.3.4.2	Message acknowledgement	29
3.3.4.3	Sender synchronization	30
3.3.5	Logging	31
3.4	SwiftMQ AMQP 1.0 client	31
3.4.1	SSL Client Authentication	32
3.4.2	Preparing connection and session	34
3.4.3	Creating consumer and producer	34
3.4.4	Sending messages	35
3.4.5	Receiving messages	36
3.4.6	Closing the connection	37
3.4.7	Logging	37
4	C++	38
4.1	Environment setup under Linux	38
4.2	Environment setup under Windows	40
4.3	Specifying the destination (addresses)	41
4.3.1.1	Receiving responses to requests	41
4.4	Preparing connection and session	42
4.4.1	Auto reference handling	43
4.5	Creating a receiver/sender	43
4.6	Thread safety	45
4.7	Receiving/sending messages	45
4.7.1	Preparing a request message	45
4.7.2	Sending a request message	45
4.7.3	Receiving a message	46
4.7.4	Message processing	46
4.8	Closing the connection	46
4.9	Compilation and linking on the Linux operating system	47
4.10	Compilation and linkage under the Windows operating system	47
4.11	Logging	47
4.12	Error handling	48
4.13	Performance	49

4.13.1	Receive pre-fetching	49
4.13.2	Message acknowledgement	49
4.13.3	Sender synchronization	50
5	.NET	51
5.1	Apache Qpid Messaging API	51
5.2	AMQP.NET Lite	51
5.2.1	Environment setup	51
5.2.2	Preparing Connection and Session	52
5.2.3	Receiving / Sending messages	53
5.2.3.1	Creating receiver / sender	53
5.2.3.2	Using filters	53
5.2.3.3	Preparing a request message	53
5.2.3.4	Sending a request message	54
5.2.3.5	Receiving a message	54
5.2.3.6	Message Processing	54
5.2.4	Closing the connection	54
5.2.5	Logging	54
6	Python	55
7	Troubleshooting	56
7.1	Errors	56
7.1.1	Connection failure	56
7.1.2	Too many connections	56
7.1.3	Unknown destination	56
7.1.4	Invalid destination	56
7.1.5	Non-existent queue	56
7.1.6	Invalid queue	57
7.1.7	Full queue	57
7.1.7.1	Message count limit	57
7.1.7.2	Byte size limit	57
7.2	Lost connection	57
8	Glossary of terms and abbreviations	58

1 Introduction

1.1 Overview

1.1.1 Eurex Clearing FIXML Interface

The Eurex Clearing FIXML Interface provides Eurex Clearing Members with a highly flexible, standards compliant and cost-effective way to use Eurex Clearing services. Based on this interface, Members are allowed to choose and deploy their own operating systems and access interfaces.

1.1.2 Eurex Clearing FpML Interface

The Eurex Clearing FpML Interface provides EurexOTC Members with a highly flexible, standards compliant and cost-effective way to use EurexOTC Clear services. Based on this interface, Members are allowed to choose and deploy their own operating systems and access interfaces.

1.1.3 Eurex Clearing Margin Calculator Interface

The Eurex Clearing Margin Calculator Interface provides EurexOTC Members with a highly flexible, standards compliant and cost-effective way to use the EurexOTC Clear Margin Calculator service. Based on this interface, Members are allowed to choose and deploy their own operating systems and access interfaces.

1.1.4 Eurex Clearing Trade Entry Interface

The Eurex Clearing Trade Entry Interface provides EurexOTC Service Providers with a highly flexible, standards compliant and cost-effective way to use EurexOTC Clear services. Based on this interface, Approved Trade sources are allowed to choose and deploy their own operating systems and access interfaces.

1.1.5 AMQP

The Advanced Message Queuing Protocol (AMQP) constitutes the preferred transport layer for delivering messages. AMQP is an open standard with a specific focus on the financial services industry which can be used royalty free. Members can choose the platform and programming language for their client applications. More information is available at the AMQP homepage:

- <http://www.amqp.org/>

1.1.6 FIXML

Application layer messages on the Eurex Clearing FIXML Interface are based upon and compliant to the widely used FIX standard. FIXML is the XML vocabulary for creating Financial Information eXchange (FIX) protocol messages based on XML.

The Futures Industry Association (FIA)/Futures and Options Association (FOA) initiative for standardized post-trade processing has chosen FIX as the standard communication protocol. More information can be found here:

- <http://www.futuresindustry.org/downloads/FIMag/2007/Outlook/Outlook-Standards.pdf>

The specification of FIX 5.0 SP2 is provided here:

- <http://www.fixtradingcommunity.org/FIXimate/FIXimate3.0/>

To learn more about supported FIX/FIXML messages, please refer to “Volume 1: Overview” and volumes 3-5 which are available for download in the public section of the Eurex Clearing website.

1.1.7 FpML

Application layer messages on the Eurex Clearing FpML Interface are based upon and compliant to the widely used FpML standard. FpML – Financial products Markup Language – is the industry standard for complex financial products which is based on XML.

The specification for FpML 5.6 is provided here:

- <http://www.fpml.org>

To learn more about supported XML/FpML messages, please refer to “Volume 1: Overview” and “Volume 3: Trade Notification & Take-up Confirmation”, and “Volume 3-A: Post Trade Events” which is available for download in the Member Section of the Eurex Clearing website.

1.2 Intended audience

This document is intended for system designers and programmers who wish to develop/adapt their client application to interact with the services offered by the Eurex Clearing FIXML Interface, the Eurex Clearing FpML Interface, the Eurex Clearing Margin Calculator Interface or Eurex Clearing Trade Entry Interface.

This Programming Guide expects the knowledge of the Eurex Clearing FIXML Interface Specification or of the Eurex Clearing FpML Interface.

1.3 Eurex Clearing Messaging Interface Connectivity documentation

The Eurex Clearing FIXML, FpML, Margin Calculator and Trade Entry Interfaces share common connectivity documents for AMQP and WebSphere MQ:

- A: Overview

- **B: AMQP Programming Guide** (this document)
- E: AMQP Setup and Internals

All “Eurex Clearing Interfaces – Connectivity” documents are available for download on the Eurex Clearing website under the following paths:

For Eurex Clearing’s C7:

<https://www.eurex.com/ec-en/>

Support > Initiatives & Releases > C7 Releases > C7 Release 7.1 > System documentation > Interfaces

Simplified (especially error & exception handling and logging) code examples to provide better overview of the functionality are available for download on GitHub.

- <https://github.com/Eurex-Clearing-Messaging-Interfaces>

1.4 Eurex Clearing FIXML Interface documentation

The Eurex Clearing FIXML Interface documentation is organized as follows:

- Volume 1: Overview
- Volume 3: Transaction & Position Confirmation
- Volume 4: Transaction & Position Maintenance
- Volume 5: Public Broadcasts
- Volume 6: Message Samples

All documents and the public keys of the AMQP broker are available for download in the public section of the Eurex Clearing website under the following paths:

For Eurex Clearing’s C7:

<https://www.eurex.com/ec-en/>

Support > Initiatives & Releases > C7 Releases > C7 Release 7.1 > System documentation > Interfaces

1.5 Eurex Clearing FpML Interface, Margin Calculator Interface and Trade Entry Interface documentation

The Eurex Clearing FpML Interface, Eurex Clearing Margin Calculator Interface and Eurex Clearing Trade Entry Interface documentation is organized as follows:

- Volume 1: Overview
- Volume 3: Trade Notification & Take-Up Confirmation
- Volume 3-A: Post Trade Events

- Volume 3-B: EurexOTC Eurex FpML API for Trade Entry
- Volume 3-C: EurexOTC Clear Margin Calculator Interface

All documents and the public keys of the AMQP brokers are available for download in the Member Section of the Eurex Clearing website under the following path:

<https://www.eurex.com/ec-en/> -> Tech -> EurexOTC Clear -> System documentation -> Interfaces

1.6 Conventions used in this document

- **Cross references** to other chapters within this document are always clickable, but not marked separately.
- **Hyperlinks to websites** are underlined.

1.7 Examples used in this document

The Member **ABCFR** and the Eurex Clearing FIXML/FpML/Margin Calculator Interface account **ABCFR_ABCFRALMMACC1** are used in the examples in all chapters of this document.

1.8 Organization of this document

- Chapter 2 – Obtaining the AMQP Client API
 - Describes how the Apache Qpid client software can be obtained
- Chapter 3 – Java
 - Describes how to use the Java interface
- Chapter 4 – C++
 - Describes how to use the C++ interface
- Chapter 5 – .NET
 - Describes how to use the .NET interface
- Chapter 6 – Python
 - Describes how to use the Python clients
- Chapter 7 – Troubleshooting
 - Describes typical problems
- Chapter 8 – Glossary of Terms and Abbreviations
 - Glossary of terms and abbreviations used through the document

1.9 Code examples

Simplified (especially error & exception handling and logging) code examples are available, to provide better overview of the functionality. The examples are available for download on GitHub:

<https://github.com/Eurex-Clearing-Messaging-Interfaces>

2 Obtaining the AMQP client API

The Eurex Clearing interfaces support only AMQP 1.0 protocol (ISO 19464).

There are multiple implementations of the AMQP protocol. AMQP brokers for Eurex Clearing interfaces are using the Apache Broker-J broker implementation. However, the Members are not obligated to use the client libraries provided by the same vendor as Eurex Clearing is using. Eurex clearing interfaces should be compatible with every AMQP client library which:

- Supports AMQP 1.0 protocol
- Supports TLS encryption
- Supports TLS client authentication and SASL EXTERNAL mechanism

Following libraries were tested for compatibility with Eurex Clearing interfaces:

- Apache Qpid client libraries
- SwiftMQ AMQP 1.0 client library
- AMQP.NET Lite client library
- Vert.x 3 Proton client library

2.1 Apache Qpid

Apache Qpid is open source AMQP implementation licensed under the Apache License 2.0. More information can be found on the Qpid website: <http://qpid.apache.org>

The client libraries supporting AMQP 1.0 are available for multiple programming languages, including:

- C++
- C# .NET
- Java
- Python

Following components are expected to be compatible with Eurex Clearing interfaces:

- Qpid Messaging API (C++)¹
- Qpid JMS for AMQP 1.0 (Java, <http://qpid.apache.org/components/jms/index.html>)
- Qpid Dispatch router (AMQP 1.0 only)
- Qpid Proton C
- C++ and Python bindings for Qpid Proton

Members are free to choose any of the above mentioned libraries according to their own requirements. The last versions tested for compatibility with Eurex Clearing interfaces are:

- Qpid Messaging C++ and Python client 1.39.0

¹ The Qpid Messaging API is written in C++. Bindings are available for .NET, Python, Ruby and Perl.

- Qpid JMS client 0.51.0
- Qpid Dispatch router 1.12.0
- Qpid Proton C and its C++ and Python bindings version 0.30.0

The Apache Qpid project provides the documentation as well as API references for all of its components on its website (<http://qpid.apache.org/documentation.html>)

2.2 SwiftMQ AMQP 1.0 Java client

SwiftMQ offers a Java based AMQP 1.0 client, which is compatible with Eurex Clearing interfaces. This client is not open source, but is available for free. More information about this client can be found on <http://www.swiftmq.com>.

The last version tested for compatibility with Eurex Clearing interfaces was 9.7.1.

2.3 AMQP.NET Lite client

AMQP.NET Lite is AMQP 1.0 client library from Microsoft, which is written in pure .NET / C#. It is available as open source under the Apache License 2.0 and can be downloaded from GitHub:

- <https://github.com/Azure/amqpnetlite>

The AMQP.NET Lite library is compatible with Eurex Clearing interfaces since release 1.1.2.

2.4 Vert.x 3

Vert.x is a tool-kit for building reactive applications on the Java Virtual Machine. AMQP support is provided using the Vert.X Proton library. More information about Vert.x can be found on its website:

- <http://vertx.io/>

The last version tested for compatibility with Eurex Clearing AMQP Interfaces is 3.3.0.

The detailed description of the Vert.x 3 libraries and their interfaces is not part of this documentation. However simple programs for receiving broadcasts, sending requests and receiving responses using Vert.x is part of the code examples – see chapter 1.9 for more details.

2.5 Upgrading client libraries

It is recommended to always use the last stable version available. New releases of the client libraries usually bring many updates and bug fixes. It is recommended to follow the development and regularly upgrade to the latest version.

3 Java

This chapter contains the guide through the development of Eurex Clearing interface client programs in Java.

3.1 SSL / TLS Certificates

Eurex Clearing interfaces are using TLS encryption and certificate based client authentication to ensure the security. Both the public keys of the AMQP broker as well as the client certificate have to be provided to the client. Without them, the clients will be unable to connect / authenticate.

3.1.1 Client certificate

The guide for generating the client certificates is part of the "Volume A: Connectivity" document. Java clients require the client certificate to be provided in the Java Keystore format (JKS), where it is encrypted and protected by password. When using the keytool utility to generate the certificate, it will be created already in the JKS format. In case other tools were used to generate the certificate, it has to be converted first.

Following command can be used to convert certificate from PKCS12 format to the JKS format²:

```
$ keytool -importkeystore -srckeystore <PKCS12 certificate> -srcstoretype PKCS12 -  
destkeystore <JKS certificate>
```

For example:

```
$ keytool -importkeystore -srckeystore ABCFR_ABCFRALMMAC1.p12 -srcstoretype PKCS12 -  
destkeystore ABCFR_ABCFRALMMAC1.keystore  
Enter destination keystore password:  
Re-enter new password:  
Enter source keystore password:  
Entry for alias carex_testcalmmac1 successfully imported.  
Import command completed: 1 entries successfully imported, 0 entries failed or  
cancelled
```

During the conversion, new password to the JKS certificate as well as the password for the existing PKCS12 file has to be entered.

A certificate which is ready to be used by a Java client might look very similar to this:

```
$ keytool -list -v -keystore ABCFR_ABCFRALMMAC1.keystore  
Enter keystore password:
```

```
Keystore type: JKS  
Keystore provider: SUN
```

```
Your keystore contains 1 entry
```

```
Alias name: abcfr_abcfralmmac1  
Creation date: Feb 10, 2015  
Entry type: PrivateKeyEntry  
Certificate chain length: 1  
Certificate[1]:  
Owner: CN= ABCFR_ABCFRALMMAC1  
Issuer: CN= ABCFR_ABCFRALMMAC1
```

² Some of the Java clients are able to use the PKCS12 format directly

```
Serial number: 54bce281
Valid from: Mon Jan 19 11:54:57 CET 2015 until: Fri Jan 19 11:54:57 CET 2018
Certificate fingerprints:
    MD5: AA:22:FD:83:43:26:8B:79:30:CA:74:6D:31:7B:BA:72
    SHA1: FE:46:CD:84:86:49:34:15:7F:1D:D8:A5:B0:BE:CA:F6:19:C2:A6:3B
    SHA256:
40:8B:A1:1F:68:58:90:0D:B2:1D:4C:F0:77:2D:CD:39:E3:73:62:B9:FA:CF:BF:EA:A5:08:08:F2:86:4
F:8A:70
    Signature algorithm name: SHA512withRSA
    Version: 3
```

```
*****
*****
```

The keystore file needs to be provided to the Java client together with the password.

3.1.2 Broker public keys

When connecting to the broker, the Member application should verify the identity of the AMQP broker in order to protect against man in the middle attacks. The Eurex Clearing AMQP brokers use certificates signed by a trusted certification authority (CA). The public keys of Eurex Clearing interfaces can be used to verify their identity. Each interface has its own unique keys for simulation and production environments.

The public key(s) should be stored in a file called "truststore". The truststore is stored in Java Keystore (JKS) format, where it is encrypted and protected by password. The truststore file needs to be provided to the Java client together with the password. The truststore can contain multiple public keys.

The public keys of the AMQP broker are available on the website of Eurex Clearing under the following path:

Eurex Clearing FIXML Interface: <https://www.eurex.com/ec-en/>

Support > Initiatives & Releases > C7 Releases > C7 Release 7.1 > System documentation > Interfaces

Eurex Clearing FpML Interface: <https://www.eurex.com/ec-en/> -> Tech -> EurexOTC Clear -> System documentation -> Interfaces

Eurex Clearing Margin Calculator Interface: <https://www.eurex.com/ec-en/> -> Tech -> EurexOTC Clear -> System documentation -> Interfaces

Eurex Clearing Trade Entry Interface: <https://www.eurex.com/ec-en/> -> Tech -> EurexOTC Clear -> System documentation -> Interfaces

They can be easily loaded into a new truststore using import functionality of the keytool utility.³

```
>keytool -importcert -file <Broker1 certificate> -alias simulation -
keystore <Truststore filename>
```

```
Enter keystore password: <Password>
Owner: CN=ecag-fixml-simul.deutsche-boerse.com
Issuer: CN=VeriSign Class 3 Secure Server CA - G3
Serial number: ad550000002b7f9b8f4f31234af
```

³ See "Volume A: Connectivity" for more details about the keytool utility.

Valid from: Tue Apr 15 18:35:26 CEST 2012 until: Sun Apr 14 18:35:26 CEST 2014

Certificate fingerprints:

MD5: 8F:AE:D7:14:CD:37:3F:3B:E8:E7:F2:42:F3:14:BE:4E

SHA1:

94:52:92:97:7C:0A:D7:23:11:E6:43:69:B0:1F:C5:1B:9F:C2:D3:9B

Signature algorithm name: SHA1withRSA

Version: 3

Trust this certificate? [no]: yes

Certificate was added to keystore

The <Broker1 certificate> and <Truststore filename> as well as the <Password> values have to be replaced according to Member's environment. The resulting truststore file should contain the public keys of all brokers as trusted certificate entry:

```
>keytool -list -keystore <Truststore filename>
```

```
Enter keystore password: <Password>
```

```
Keystore-Typ: jks
```

```
Keystore-Provider: SUN
```

```
Your keystore contains 2 entries
```

```
simulation, 28.04.2011, trustedCertEntry,
```

```
Certificate fingerprint (MD5):
```

```
86:58:B9:E1:83:80:E6:68:63:7E:92:EA:30:4A:D5:91
```

```
production, 28.04.2011, trustedCertEntry,
```

```
Certificate fingerprint (MD5):
```

```
86:63:B9:EA:83:80:E6:6F:6C:AE:92:EB:40:A2:31:53
```

3.1.3 SSL / TLS debugging

In case of problems with the SSL / TLS connection, the applications using Apache Qpid Java API have a SSL debugging mode. This mode can be activated using the system property

```
-Djavax.net.debug=ssl
```

3.2 JMS

Some of the Java clients are based on Java Message Service (JMS). JMS is a message oriented middleware API, which is a part of the Java Platform Enterprise Edition. More information about Java Message Service can be found at the Oracle website - <http://www.oracle.com/technetwork/java/index-jsp-142945.html>.

The goal of this chapter is not to provide a comprehensive guide to JMS API, but to provide code snippets illustrating the work with the Eurex Clearing FIXML/FpML/Margin Calculator Interface in Java.

The details which are specific to the different JMS implementations will be described in the subsequent chapters.

3.2.1 Java Naming and Directory Interface (JNDI)

The JMS applications typically use the Java Naming and Directory Interface (JNDI) to obtain a connection factory, connection URI and message source / target addresses. The JNDI configuration might be kept separate from the application – for example stored in a properties file. But it can be also dynamically created *Properties* or *HashMap* object.

The JNDI properties are used by the client application to connect to the broker and send or receive messages. The connection factory name as well as the syntax of the connection URI and addresses are different for different APIs.

The properties file has to be loaded and processed into the application. The classes `Properties` (`java.util`) and `InitialContext` (`javax.naming`) will be used. Following example shows how to load the properties from a file:

```
InitialContext ctx;  
Properties properties = new Properties();  
properties.load(new FileInputStream("<PropertiesFile>"));  
ctx = new InitialContext(properties);
```

The `<PropertiesFile>` has to be replaced according to the Member's environment. As a result, an `InitialContext` object is created in variable `ctx` containing all JNDI resources defined in the properties file. The context will be used later to retrieve the connection string and the destinations.

3.2.2 Preparing connection and session

The connection (class `Connection` from `javax.jms`) is created using the `ConnectionFactory` class (`javax.jms`). The connection factory has to be initialized using the connection string from our context:

```
ConnectionFactory fact = (ConnectionFactory)ctx.lookup("connection");
```

The connection factory is used to create a connection:

```
Connection conn;  
conn = fact.createConnection();
```

After these steps, the connection is created in the `conn` object and connects to the AMQP broker. However, it is in state `STOPPED`. The `STOPPED` state allows the applications to send messages, but not to receive them. In order to receive messages, the connection has to be started (see chapter 3.2.4). This gives the application enough time to prepare for receiving of messages (create receivers, queues, listeners, ...).

Using the prepared connection, a session can be created.⁴ The session is an instance of class `Session` from package `javax.jms`:

```
Session sess;  
sess = conn.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

⁴ For more details about the differences and relationship between connection and session, please visit JMS documentation or AMQP specification.

The `Session.CLIENT_ACKNOWLEDGE` parameter is instructing the session, that the acknowledgments of the messages will be done manually by the client application. In case the acknowledgement should be done automatically by the application, the `Session.AUTO_ACKNOWLEDGE` option should be used. Using auto-acknowledgements without transactions is not recommended (see Eurex Clearing FIXML/FpML/Margin Calculator Interface Specification, "Volume E: AMQP Setup & Internals" for more details about reliability).

A session should be used as a long lasting resource and shouldn't be created too often. For instance creating a new session for sending each message in a tight loop can result in following exception:

javax.jms.JMSException (Exception when sending message:timed out waiting for session to become open (state=DETACHED))"

Instead, the application should create a session before entering the loop and re-use the session.

When using the Spring framework, the `SingleConnectionFactory` should not be used because it recreates `Session` and `Producer` each time a message is to be sent. Instead, for example the `CachingConnectionFactory` should be used. The `CachingConnectionFactory` keeps both `Session` and `Producer` created and attached.

3.2.3 Creating a receiver/sender

After the connection and session have been prepared, a receiver or producer can be prepared next. The producer is an instance of class `MessageProducer` (`javax.jms`). The producer can be created by the session, using the method `createProducer (...)`. The producer is always bound to a specific destination, queue or topic which can be created from the context which has been prepared in chapter **Error! Reference source not found.:**

```
MessageProducer requestProducer;  
Destination requestDest = (Destination)ctx.lookup("requestAddress");  
requestProducer = sess.createProducer(requestDest);
```

The message receiver is an instance of class `MessageConsumer`⁵ (`javax.jms`). The receiver is created in the same way as the producer. Just instead of using the session's `createProducer (...)` method, the method `createConsumer (...)` is used. The receivers for receiving responses or broadcasts are created in the same way. Just the destination (and the address which has been used to create the destination) is different:

```
MessageConsumer responseConsumer;  
Destination responseDest = (Destination)ctx.lookup("responseAddress");  
responseConsumer = sess.createConsumer(responseDest);
```

When creating the consumer, you can also specify a selector to receive only selected messages. The selector can be either based on a message property or on a message application property. On the wire, the JMS selector is translated to AMQP filter and the filtering of messages is done directly on the AMQP broker. The JMS selector follows the JMS syntax. You can filter based on application properties:

⁵ The JMS and AMQP are using slightly different terminology. The JMS term `Consumer` corresponds to the AMQP term `receiver`.

```
responseConsumer = sess.createConsumer(responseDest,  
"BusinessDate='20160813'");
```

Or you can filter based on message properties – for example using JMS Correlation ID:

```
responseConsumer = sess.createConsumer(responseDest,  
"JMSCorrelationID='" + correlationID + "'");
```

There are multiple methods to get the messages from the receiver. One of them is a usage of a message listener. Message listener is a special object, which implements the `MessageListener` interface from package `javax.jms`. In order to use the listener, it has to be registered with the producer. The registration can be done using the `setMessageListener(...)` method of the receiver:

```
responseConsumer.setMessageListener(new Listener());
```

The listener will be described in detail in chapter 3.2.6.3.

3.2.4 Starting the connection

With connection, session and receiver ready, the connection can be started:

```
conn.start();
```

Only when the connection is started, the application can receive messages from the AMQP broker. If the application is intended to only send messages, the start of the connection is not necessary.

3.2.5 Thread safety

The JMS Session object is not thread safe. Since a `MessageProducer/MessageConsumer` is bound to a Session it cannot be used from more than one thread at the same time. For multi-threaded access it is necessary to use a separate session (and underlying objects) from each thread.

3.2.6 Receiving/sending messages

3.2.6.1 Preparing a request message

To prepare a new message, the `TextMessage` class (`javax.jms`) can be used. For request messages, only the message body and the reply to key have to be filled. The message body can be entered when a new message is constructed, using the session's method `createTextMessage(...)`. After preparing the message, the reply to destination, queue or topic can be assigned to it. As before, the destination for the reply to parameter is created from the context object. With the destination being ready, the method `setJMSReplyTo(...)` can be used to assign it to the message.

```
TextMessage message = sess.createTextMessage("<FIXML>...</FIXML>");  
Destination replyDest = (Destination)ctx.lookup("replyAddress");  
message.setJMSReplyTo(replyDest);
```

3.2.6.2 Sending a request message

The message prepared in the previous chapter can be sent using the message producer. Since the producer has been initialized with the destination already at the beginning, it is not necessary to use the request destination again:

```
requestProducer.send(message);
```

Depending on the specific client, the messages might be by default sent synchronously or asynchronously.

The request queues have only limited capacity and when the queue is almost full a flow control mechanism will be activated by the broker (the exact queue sizes as well as the flow control thresholds for different interfaces can be found in the Volume E of this documentation). When the flow control is activated for the given request queue, the broker will delay sending the confirmations of received messages. However, the flow control support in the Java JMS API is only limited and when the flow control is activated the client will only wait for certain time and afterwards the send call fails with an exception. When sending the messages asynchronously, the client will continue sending messages and can exceed the queue capacity despite the flow control.

The recommended way how to ensure the queue capacity will not be exceeded in Java JMS client is to track the number of outstanding requests (requests which were sent and not yet responded to) within the application and stop sending messages when the number of outstanding requests reaches the flow control threshold.

3.2.6.3 Receiving a message using Message listener

Message listener can be every object which implements the `MessageListener` interface (`javax.jms`). The assignment of the listener to the receiver is described in chapter 3.2.3. The `MessageListener` interface has only one method, called `onMessage (...)`. This method is called whenever the receiver receives a new message. The message is passed to the `onMessage ()` method as a parameter and can be either processed inside of the method or passed to another object. In case the session has been created with manual acknowledgements, the message should be acknowledged after its processing is finished (see chapter 3.2.2 for more details). The acknowledgement can be done using the call of the `acknowledge ()` method of the message. When using auto-acknowledgements, it is not necessary to acknowledge the message manually. One listener object can be used by multiple receivers.

```
public class Listener implements MessageListener
{
    public void onMessage(Message msg) {
        // Processing of the message
        try {
            // Acknowledging the message manually
            msg.acknowledge();
        } catch (JMSEException e) {
            // Handling the exception
        }
    }
}
```

Please note, that the JMS Session object is not thread-safe. Therefore, it should not be used concurrently from multiple threads. If it is planned to receive concurrently messages from multiple sources, then one should create different sessions and create for each session one MessageConsumer, since the MessageConsumer is created and assigned to one session only. Afterwards, each MessageListener will then be assigned to different MessageConsumers and therefore to different Sessions.

Using single session for multiple MessageConsumers has the effect that all calls to their onMessage () methods are serialized and the parallel message consuming is not used.

The message listener is used in the broadcast receiver example.

3.2.6.4 Receiving a message using the receive() method

Messages can be also received using the receive () method of the MessageConsumer instance:

```
Message msg = responseConsumer.receive();
// Processing of the message
msg.acknowledge();
```

Using parameters of the receive () method, the application can either wait until a message is received for a limited (pass the timeout in milliseconds as a parameter to the method) or unlimited time. Using the method receiveNoWait (), a message can be received without waiting (if there is no message waiting, the method will return null).

In case the session has been created with manual acknowledgements, the message should be acknowledged after its processing is finished (see chapter 3.2.2 for more details). The acknowledgement can be done using the call of the acknowledge () method of the message. When using auto-acknowledgements, it is not necessary to acknowledge the message manually.

The receive () method is used in the response receiver example.

3.2.6.5 Message processing

The received message is returned from the `receive()` method or passed to the `onMessage()` listener method as an instance of the more generic class `Message (javax.jms)`. In order to process the message, it has to be casted either to `TextMessage` or the `ByteMessage`. The JMS API decides based on the message payload and message properties whether the message will be handled as `TextMessage` or `ByteMessage`. The messages received on Eurex Clearing interfaces might be presented in both types, depending on the message sender and the exact content.

With an instance of `TextMessage` class, it is easy to retrieve the message body. The method `getText(...)` will return the body as a `String` object. `ByteMessage` has the methods `getBodyLength()` and `readByte()`, which can be used to read the message body byte by byte. Since the FIXML/FpML/Margin Calculator response or broadcast messages are text based, the byte content has to be transformed to a string using a `StringBuilder` class.

```
if (msg instanceof TextMessage)
{
    TextMessage textMessage = (TextMessage) msg;
    messageText = textMessage.getText();
    // process the message body
}
else if (msg instanceof ByteMessage)
{
    ByteMessage bytesMessage = (ByteMessage) msg;
    StringBuilder builder = new StringBuilder();

    for (int i = 0; i < bytesMessage.getBodyLength(); i++) {
        builder.append((char)bytesMessage.readByte());
    }

    // process the message body
}
else
{
    // Unexpected message delivered
}
```

All other message attributes can be retrieved using the usual getter methods.

3.2.7 Closing the connection

When the application is exiting, it should properly close all AMQP related objects. The receivers, producers, session and connection all have a method `close()`, which will properly close them:

```
responseConsumer.close();  
requestProducer.close();  
sess.close();  
conn.close();
```

3.2.8 Error handling

In JMS the errors are handled by catching the proper exceptions. The errors are either synchronous (e.g. creating the session fails) or asynchronous (e.g. the connection to the broker is lost as the client is waiting for messages). This chapter explains how to properly handle and recover from such situations.

A client application should be designed in a way that it is resilient to the above errors, it doesn't get stuck when error occurs and at the same time it doesn't start consuming more and more resources. The main building blocks for each client application are (starting from the top) a connection, session and receiver/sender. Closing a session automatically closes all receivers/senders beneath it and closing a connection automatically closes all underlying sessions.

A typical way of handling the chain of creating the producer/consumer may look like:

```
try {  
    connection = fact.createConnection();  
    session = connection.createSession(false,  
                                       Session.CLIENT_ACKNOWLEDGE);  
    Destination requestDestination = (Destination)  
                                     ctx.lookup("requestAddress");  
    MessageProducer requestProducer;  
    requestProducer = session.createProducer(requestDestination);  
} catch (JMSEException e) {  
    e.printStackTrace();  
} finally {  
    connection.close();  
}
```

In the above example we omitted the creation of the context and connection factory for better readability. Creation of the connection, session or message producer can result in a failure and in that case an exception is thrown. In the catch block, we print the stack trace and continue with a finally block by closing the connection, which is executed also in the case when try block finished without any failure.

The `JMSEException` is the root class for exceptions thrown by JMS API methods. Catching `JMSEException` provides a generic way of handling all exceptions related to the JMS API. One can catch the subclasses of this exception (e.g. `IllegalStateException`, `InvalidDestinationException`) which are described in the JMS API documentation. In some cases, depending on the type of error, it is not necessary to close the whole connection, but only the session and/or restart the producer/consumer. However, the JMS doesn't define what happens in terms of the connection preservation in each case; neither there is a straightforward way of checking whether the connection or session is still valid. Closing and restarting the connection can be therefore considered as the safest option.

The above example illustrates how to catch exceptions synchronously, i.e. when some particular JMS API method fails.

For applications with only asynchronous message consumers, there exists an `ExceptionListener` interface behavior as follows. If a JMS provider detects a serious problem with a `Connection` object, it informs the `Connection` object's `ExceptionListener`, if one has been registered. It does this by calling the listener's `onException` method, passing it a `JMSEException` argument describing the problem. In practice, when an exception listener is called, the connection is broken and the JMS service is no longer available for the connection.

The example source codes present the way how to register the `ExceptionListener` and how to notify the main thread in case an asynchronous exception occurs. Such a class has to implement the `ExceptionListener` interface and override the `onException` method. Afterwards, the instantiated class can be registered using `Connection`'s `setExceptionListener` method.

Generally, the client application should properly check all JMS API methods for exceptions and in case the exception occurred, an application can, at a minimum, log the problem and clean up its resources. An application can also notify any interested parties that need to be notified of such a problem. An application should be designed with a clean initialization setup, so it would be feasible to reinitialize the JMS objects when the exception occurs (either synchronous or asynchronous).

3.3 Apache Qpid JMS client for AMQP 1.0

3.3.1 Connection Factory

The Apache Qpid JMS client for AMQP 1.0 has its own connection factory, which is used to resolve the JNDI properties. The properties file has to contain the identification of the context factory:

```
java.naming.factory.initial=org.apache.qpid.jms.jndi.JmsInitialContextFactory
```

3.3.2 Connection URI

The connection URI specifies where the AMQP client should connect and what connection parameters should be used. In the JNDI properties, the connection URI should be placed like this:

```
connectionfactory.[jndiname]=<ConnectionURL>
```

for example:

```
connectionfactory.connection=<ConnectionURL>
```

For connecting to Eurex Clearing interfaces the connection string needs to specify:

- The correct IP address/hostname of the broker
- The correct port of the broker
- The path to the keystore with the client certificate
- Alias of the member certificate in the keystore
- The path to the truststore with the broker public keys
- Passwords for the keystore and truststore
- Idle timeout

The connection URI has a following format:

```
amqp://hostname:port[?option=value[&option2=value...]]
```

or for SSL connections:

```
amqps://hostname:port[?option=value[&option2=value...]]
```

The options needed to connect to the Eurex AMQP Interfaces are:

- `jms.clientID`
- `transport.trustStoreLocation`
- `transport.trustStorePassword`
- `transport.keyStoreLocation`
- `transport.keyStorePassword`
- `transport.keyAlias`
- `amqp.idleTimeout`

Example connection string:

```
amqps://<Hostname>:<Port>?jms.clientID=<ClientID>&transport.trustStoreLocation=<PathToTruststore>&transport.trustStorePassword=<TruststorePassword>&transport.keyStoreLocation=<PathToKeystore>&transport.keyStorePassword=<KeystorePassword>&transport.keyAlias=<KeystoreAlias>&amqp.idleTimeout=<HeartbeatInterval>
```

The client ID is the unique identifier of a Member application and can be defined according to the Member's needs. Multiple connections with the same client ID will be refused. [The client ID, if used, must be unique across all the connections of the broker \(FIXML/FpML/Margin Calculator/Trade Entry\). This limit applies per broker.](#)

Idle timeout needs to be specified in number of milliseconds. If not used than the Qpid JMS client is using default idle timeout / heartbeat of 60000 milliseconds. The recommended idleTimeout interval is between 30000 and 120000 milliseconds.

An example connection string for the Eurex Clearing FIXML Interface may then look like this:

```
amqps://ecag-fixml-simul.deutsche-boerse.com:10170?jms.clientID=my-test-  
client1&transport.trustStoreLocation=truststore.jks&transport.trustStorePassword=123456&  
transport.keyStoreLocation=ABCFR_ABCFRALMMACC1.keystore&transport.keyStorePassword=12345  
6&transport.keyAlias=abcf_r_abcf_ralmmacc1&amqp.idleTimeout=60000
```

Additional connection options can be found in the documentation on <http://qpid.apache.org/documentation.html>

The client supports automatic failover / reconnect. To enable the failover, the connection URI has to be wrapped into a *failover* prefix. The options starting with “jms.” Should be used outside of the failover enclosure while the other options (e.g. starting with “amqp.” or “transport.”) should stay inside. Alternatively, the other options can be used outside of the failover enclosure with the prefix “failover.nested.” – such options would apply to all brokers.

```
failover:(amqps://hostname:port[?option=value[&option2=value...]])[?failoverOption=value  
[&failoverOption2=value...][&jmsOption=value...][&nestedOption=value...]
```

The failover supports among other following options:

- failover.reconnectDelay
- failover.maxReconnectAttempts
- failover.useReconnectBackOff
- failover.reconnectBackOffMultiplier

These options can be used to control how many times and in which time intervals the client should try to reconnect to the broker. For example:

```
failover:(amqps://ecag-fixml-simul.deutsche-  
boerse.com:10170?...)?failover.reconnectDelay=30000&failover.maxReconnectAttempts=10&fail  
over.useReconnectBackOff=false&jms.clientID=myClient&failover.nested.amqp.idleTimeout=6  
0000
```

Additional failover options can be found in the documentation on <http://qpid.apache.org/documentation.html>

3.3.3 Addresses

Addresses are used to describe the message target or message source.⁶ The address is a string, which is passed as a parameter to a receiver or a sender, where it is handled. An address always resolves to a node – either queue or topic. This chapter will focus on the specific address strings, which can be used to interact with the Eurex Clearing interfaces.

Every application needs 4 different address string types in order to fully utilize the Eurex Clearing interfaces:

1. Receiving broadcasts
2. Receiving responses

⁶ Client APIs from other providers will use a different approach for defining message targets and sources. Please refer to the documentation of the used API for more details.

3. Sending requests

4. "ReplyTo" address in requests

In the JNDI properties, the type *queue* should be used for receiving messages (broadcast address and response address) and the type *topic* for sending messages (reply address and request address):

```
queue.[jndiname]=<Address>  
topic.[jndiname]=<Address>
```

for example:

```
topic.requestAddress=<Address>  
queue.responseAddress=<Address>
```

3.3.3.1 Receiving responses to requests

As described in the Eurex Clearing FIXML/FpML/Margin Calculator Interface Specification, "Volume E: AMQP Setup & Internals", receiving responses to requests can be done either using an auto-delete response queue which has to be created by the client application and bound to the response exchange or using the predefined response queue.

The following address string can be used as a template:

```
<ResponseQueueName>
```

The placeholders in this template have to be replaced with the appropriate values:

```
queue.responseAddress=response.ABCFR_ABCFRALMMACC1
```

3.3.3.2 Sending requests

The request messages should be sent to the request exchange, which is specific for each Member. Since the request exchange is already predefined, the address string is simpler than the address string for receiving responses:

```
<RequestExchange>
```

The placeholders in this template have to be replaced with the appropriate values:

```
topic.requestAddress=request.ABCFR_ABCFRALMMACC1
```

3.3.3.3 "ReplyTo" address in requests

The "ReplyTo" address is assigned as a property to the request message. It encodes both the reply to exchange as well as the reply to routing key:

```
<ResponseExchange>/<ResponseRoutingKey>
```

The placeholders in this template have to be replaced with the appropriate values:

```
topic.replyAddress=response/response.ABCFR_ABCFRALMMACC1
```

The response to a request message sent with the reply to address above can be received by a receiver created using the example address from chapter **Error! Reference source not found.**

3.3.3.4 Receiving broadcasts

To receive broadcast messages, it is necessary to create a receiver on the broadcast queues predefined during the technical maintenance. The following address string can be used as a template:

<*PredefinedBroadcastQueue*>

The <*PredefinedBroadcastQueue*> placeholder has to be replaced by the real name of the Members' broadcast queue. The address string corresponding to the queue for the trade confirmation broadcast stream of Member ABCFR, account ABCFR_ABCFRALMMACC1 will be as follows:

```
queue.responseAddress=broadcast.ABCFR_ABCFRALMMACC1.TradeConfirmation
```

3.3.4 Performance

3.3.4.1 Receive pre-fetching

AMQP brokers typically push messages to client consumers without explicit client requests (asynchronously, in the background) up to a certain number of unsettled messages. The next time a message would be passed on to the application code, it is usually taken from this buffer (avoiding synchronous I/O). This buffering capacity of a client is configurable and it is typically set to hundreds of messages by default. Setting it too low can have a negative impact on message throughput (less overlap of message processing and background I/O). Setting it too high can have a negative impact on client memory consumption (pre-fetch buffers need to hold many messages). Also, all messages pre-fetched by one consumer are "locked" to that consumer (and will not be delivered to any other consumer reading the same queue) until the consumer releases/rejects them. This can lead to a less than ideal load balancing in case of parallel consumption and processing of messages from a single broker queue.

In Apache Qpid JMS client for AMQP 1.0 client, the pre-fetch capacity can be specified in the connection address string using several options. For receiving from queues, following two options are relevant:

- `jms.prefetchPolicy.queuePrefetch`
- `jms.prefetchPolicy.all`

The default prefetch limit is set to 1000. Additional prefetch options can be found in the documentation on <http://qpid.apache.org/documentation.html>

3.3.4.2 Message acknowledgement

Message acknowledgement is synchronous by default. In case a client application requires asynchronous message acknowledgement (e.g. doesn't require guarantee that the acknowledged message was removed from a broker queue before proceeding further), it can be enabled using the "jms.sendAcksAsync" connection option.

When using explicit acknowledgement of received messages, doing one-by-one synchronous acknowledgement of messages can severely degrade performance. Message consumption rate is then limited by the network round-trip latency between the client and the broker. For example, a round-trip time of 100ms would limit the synchronous throughput to no more than 10 messages per second.

Message receiving performance in the synchronous acknowledge mode can be improved via block processing of messages. The following Java code example is an extension of the code example from section 3.2.6.4 and illustrates block processing. It will receive and process up to 100 messages, then finally acknowledge all received messages in a single call (thus mitigating the impact of network latency):

```
int maxCount = 100;
Message message, lastMessage = null;
// receive and process up to 100 messages
while ((message = responseConsumer.receiveNoWait()) != null) {
    // process the message
    System.out.println(message.toString());
    // remember the last processed message
    lastMessage = message;
    // check total block size
    if (--maxCount <= 0) {
        break;
    }
}
// acknowledge ALL previously received messages
if (lastMessage != null) {
    lastMessage.acknowledge();
}
```

3.3.4.3 Sender synchronization

When sending messages (as shown in section 3.2.6.2), the send call is by default synchronous for messages which are persistent and are not part of a transaction. All other messages are sent asynchronously. Asynchronous sending means that a message being sent is not guaranteed to arrive at a broker by the time the send call returns in a client application. When a message is sent synchronously, the send call will wait for confirmation from the broker. However, sending messages synchronously can have a significant negative impact on performance in case of significant network latency between the client and the server.

Unlike in C++, it is not possible to invoke explicit session synchronization in JMS (to achieve block or batch sending and mitigate the negative impact of network latency).

Following options in the connection URI can be used to change this behavior:

- `jms.forceAsyncSend`
- `jms.alwaysSyncSend`

The option `.jms.alwaysSyncSend` can force the client to send all messages synchronously. The option `.jms.forceAsyncSend` does the exact opposite – it forces the client to send all messages asynchronously.

3.3.5 Logging

The Apache Qpid JMS client for AMQP 1.0 uses Simple Logging Facade for Java (SLF4J), which serves as a simple facade or abstraction for various logging frameworks. SLF4J allows the end-user to plug in the desired logging framework at deployment time. There is only a single mandatory dependency, namely **slf4j-api** library for using the client API.

According to the desired logging framework, one of the following SLF4J bindings can be included:

- **slf4j-log4j12** - Binding for log4j, a widely used logging framework. Needs to place `log4j.jar` on the class path.
- **slf4j-jdk14** - Binding for `java.util.logging`, also referred to as JDK 1.4 logging
- **slf4j-nop** - Binding for NOP, silently discarding all logging.
- **slf4j-simple** - Binding for Simple implementation, which outputs all events to `System.err`. Only messages of level INFO and higher are printed. This binding may be useful in the context of small applications.
- **slf4j-jcl** - Binding for Jakarta Commons Logging. This binding will delegate all SLF4J logging to JCL.

The simplest way to see the Java Qpid API log messages is to include the **slf4j-simple** binding library into the project path. All Qpid messages from level INFO and higher will be printed into the standard error output.

For more advanced configuration and also if the project already uses log4j logging framework, one can use **slf4j-log4j12** binding. It is required to place also the **log4j.jar** library into the project's class path. Configuration for the log4j is extensive and can be found on the project's website (<http://logging.apache.org/log4j/>). By default when no configuration file could be located the `DefaultConfiguration` will be used. This will cause logging output to go to the console.

To enable the logging of AMQP frames sent / received by the client, the Frame logger can be enabled by using following option in the connection URI:

```
amqp.traceFrames=true
```

Another option can be used to display the raw AMQP bytes which the client sends / receives:

```
amqp.traceBytes=true
```

3.4 SwiftMQ AMQP 1.0 client

Unlike the Apache Qpid client, the SwiftMQ client is not JMS based client. By default, this client doesn't support the SSL / TLS client authentication. But the support can be easily added using a custom SSL socket factory.⁷

⁷ <http://blog.effective-messaging.com/2013/10/connecting-with-swiftmq-java-client-to.html>

3.4.1 SSL Client Authentication

While the SwiftMQ client doesn't support SSL / TLS client authentication, it allows the use of custom SSL socket factories. That can be utilized to add the support for the SSL client authentication. The new socket factory will be based on the original socket factory shipped with the SwiftMQ client, which supports only SSL / TLS server authentication.

The new socket factory will use a custom key manager when initializing the SSL context. Everything else will be left for the original `JSSocketFactory` from the package `com.swiftmq.net`:

```
public class MySSLSocketFactory extends JSSocketFactory {
    private String alias;

    public MySSLSocketFactory(String alias)
    {
        this.alias = alias;
    }

    public Socket createSocket(InetAddress addr, int port) throws UnknownHostException,
    IOException
    {
        return initializeSSLContext().getSocketFactory().createSocket(addr, port);
    }

    public Socket createSocket(String host, int port) throws UnknownHostException,
    IOException
    {
        return initializeSSLContext().getSocketFactory().createSocket(host, port);
    }

    private SSLContext initializeSSLContext()
    {
        SSLContext sslContext = null;
        KeyManager[] keyManagers = null;

        try {
            sslContext = SSLContext.getInstance("TLS");
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }

        try {
            keyManagers = new KeyManager[] { new MyKeyManager(alias) };
        } catch (GeneralSecurityException e1) {
            e1.printStackTrace();
        } catch (IOException e1) {
            e1.printStackTrace();
        }

        try {
            sslContext.init(keyManagers, null, null);
        } catch (KeyManagementException e) {
            e.printStackTrace();
        }

        return sslContext;
    }
}
```

The key manager will pass the alias of the certificate which should be used for the client authentication:


```
public class MyKeyManager extends X509ExtendedKeyManager {
    private String alias;
    private X509ExtendedKeyManager originalKeyManager;

    public MyKeyManager(String alias) throws GeneralSecurityException, IOException
    {
        this.alias = alias;

        KeyStore ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream(new
File(System.getProperty("javax.net.ssl.keyStore"))),
System.getProperty("javax.net.ssl.keyStorePassword").toCharArray());
        KeyManagerFactory kmf =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
        kmf.init(ks,
System.getProperty("javax.net.ssl.keyStorePassword").toCharArray());
        originalKeyManager = (X509ExtendedKeyManager)kmf.getKeyManagers()[0];
    }

    @Override
    public String chooseClientAlias(String[] keyType, Principal[] issuers, Socket
socket) {
        return alias;
    }

    @Override
    public String chooseServerAlias(String keyType, Principal[] issuers, Socket socket)
{
        return originalKeyManager.chooseServerAlias(keyType, issuers, socket);
    }

    public X509Certificate[] getCertificateChain(String alias)
    {
        return originalKeyManager.getCertificateChain(alias);
    }

    public String[] getClientAliases(String keyType, Principal[] issuers)
    {
        return new String[]{alias};
    }

    public PrivateKey getPrivateKey(String alias)
    {
        return originalKeyManager.getPrivateKey(alias);
    }

    public String[] getServerAliases(String keyType, Principal[] issuers)
    {
        return originalKeyManager.getServerAliases(keyType, issuers);
    }

    public String chooseEngineClientAlias(String[] keyType, Principal[] issuers,
SSLSEngine engine)
    {
        return alias;
    }

    public String chooseEngineServerAlias(String keyType, Principal[] issuers, SSLSEngine
engine)
    {
        return originalKeyManager.chooseEngineServerAlias(keyType, issuers, engine);
    }
}
```

The key manager will make sure that the proper certificate is used for the authentication.

3.4.2 Preparing connection and session

The keystore and truststore files which will be used for establishing the connection with the AMQP broker are specified as system properties:

```
System.setProperty("javax.net.ssl.trustStore", <PathToTruststore>);  
System.setProperty("javax.net.ssl.trustStorePassword", <TruststorePassword>);  
System.setProperty("javax.net.ssl.keyStore", <PathToKeystore>);  
System.setProperty("javax.net.ssl.keyStorePassword", <KeystorePassword>);
```

e.g.:

```
System.setProperty("javax.net.ssl.trustStore", "truststore");  
System.setProperty("javax.net.ssl.trustStorePassword", "123456");  
System.setProperty("javax.net.ssl.keyStore", "ABCFR_ABCFRALMMACC1.keystore");  
System.setProperty("javax.net.ssl.keyStorePassword", "123456");
```

With the new socket factory ready, the connection can be established. First, the `AMQPContext` has to be initialized to specify that the SwiftMQ library should behave as AMQP client. With the context, the `Connection` can be created. The username and password should be left as an empty string. To be able to connect to Eurex Clearing interfaces, the SASL mechanism needs to be set to `EXTERNAL`. Additionally, the client has to be instructed to use the the socket factory prepared in previous chapter. Once everything is ready, the client can connect to the broker:

```
AMQPContext ctx = new AMQPContext(AMQPContext.CLIENT);  
Connection connection = new Connection(ctx, <Hostname>, <Port>, "", "");  
connection.setMechanism("EXTERNAL");  
connection.setSocketFactory(new MySSLSocketFactory(<CertificateAlias>));  
connection.setIdleTimeout(<idleTimeout_in_milliseconds>);  
connection.connect();
```

e.g.:

```
AMQPContext ctx = new AMQPContext(AMQPContext.CLIENT);  
Connection connection = new Connection(ctx, ecag-fixml-simul.deutsche-boerse.de, 10170,  
"", "");  
connection.setMechanism("EXTERNAL");  
connection.setSocketFactory(new MySSLSocketFactory("abcfr_abcfralmmacc1"));  
connection.setIdleTimeout(60000);  
connection.connect();
```

Once the connection is established, the session can be created on top of it:

```
Session session = connection.createSession(1000, 1000);
```

The `createSession(...)` method accepts two parameters. These define the size of the incoming and outgoing windows – how many unsettled (unacknowledged) messages can be received from the broker or sent to the broker.

3.4.3 Creating consumer and producer

Once the session is ready, it can be used to create message consumer or producer. To create a consumer, the `createConsumer(...)` method should be used. It requires several parameters:

- The address of the message source (queue name)
- The quality of service

- Link credit
- No local filter
- Selector filter

The quality of service should be in most case AT_LEAST_ONCE, because AT_MOST_ONCE reliability can result in a message loss and EXACTLY_ONCE reliability is not supported by the AMQP broker used on Eurex Clearing interfaces. The no local filter is irrelevant for Eurex Clearing interfaces, because the client application is not allowed to send messages to the same queues where it receives the messages from. The selector filter can be used to receive only selected messages from the queue.

```
Consumer c = session.createConsumer(<QueueName>, 1000, QoS.AT_LEAST_ONCE, true, null);
```

For example to receive responses:

```
Consumer c = session.createConsumer("response.ABCFR_ABCFRALMMACC1 ", 1000, QoS.AT_LEAST_ONCE, true, null);
```

Or to receive broadcasts:

```
Consumer c = session.createConsumer("response. ABCFR_ABCFRALMMACC1.TradeConfirmation", 1000, QoS.AT_LEAST_ONCE, true, null);
```

You can also use the selector filter to receive only some messages. Filter can be based on application property:

```
Consumer c = session.createConsumer("response. ABCFR_ABCFRALMMACC1.TradeConfirmation", 1000, QoS.AT_LEAST_ONCE, true, "BusinessDate='20160813'");
```

Or based on message property - for example correlation ID:

```
Consumer c = session.createConsumer("response.ABCFR_ABCFRALMMACC1 ", 1000, QoS.AT_LEAST_ONCE, true, "amqp.correlation_id='" + correlationID + "'");
```

Creating a message producer is very similar to the consumer. The createProducer(...) method requires only two parameters:

- The address of the message target (AMQP exchange / topic name)
- The quality of service

Again, the quality of service should be in most case AT_LEAST_ONCE, because AT_MOST_ONCE reliability can result in a message loss and EXACTLY_ONCE reliability is not supported by the AMQP broker used on Eurex Clearing interfaces.

```
Producer p = session.createProducer(<ExchangeName>, QoS.AT_LEAST_ONCE);
```

For example to send requests:

```
Producer p = session.createProducer("request.CBKFR_TESTCALMMACC1", QoS.AT_LEAST_ONCE);
```

3.4.4 Sending messages

Messages sent to Eurex Clearing interfaces should contain the XML based payload as String type value:

```
AMQPMessage msg = new AMQPMessage();  
msg.setAmqpValue(new AmqpValue(new AMQPString("<FIXML>...</FIXML>")));
```

Every request message should contain the reply to information, which tells Eurex system where should the response message be sent.

```
Properties msgProp = new Properties();
```

```
msgProp.setReplyTo(<ReplyToAddress>);  
msg.setProperties(msgProp);
```

The reply to should be set to response exchange / response routing key:

```
Properties msgProp = new Properties();  
msgProp.setReplyTo(new AddressString("response/response.ABCFR_ABCFRALMMACC1"));  
msg.setProperties(msgProp);
```

Once the message is prepared, it can be sent using the producer:

```
p.send(msg);
```

In SwiftMQ, the messages are sent asynchronously.

3.4.5 Receiving messages

Messages can be received from the message consumer using several different `receive(...)` methods. The SwiftMQ client offers a choice between several blocking and non-blocking variants. It also supports possibility to setup asynchronous callback when a message is available. More details about the different methods can be found in the SwiftMQ documentation.

Following example shows how to receive the message using a blocking receiver with timeout. It also shows how to extract the XML from the message and accept it after it is processed.

```
while (true) {  
    respMsg = c.receive(60000);  
  
    if(respMsg == null) {  
        System.out.println("-I- No message in queue for 60 seconds. Finishing ...");  
        break;  
    }  
  
    System.out.println("-I- Response message received");  
  
    Data data = respMsg.getData().get(0);  
    StringBuilder builder = new StringBuilder();  
    byte[] bytes = data.getValue();  
  
    for (int i = 0; i < bytes.length; i++) {  
        builder.append((char) bytes[i]);  
    }  
  
    System.out.println("XML payload: " + builder.toString());  
  
    // Process message  
  
    respMsg.accept();  
}
```

}

3.4.6 Closing the connection

When the client application is exiting, it should close all consumers, producers, session and the connection itself. All these objects can be closed using their `close()` method.

```
p.close();  
c.close();  
session.close();  
connection.close();
```

3.4.7 Logging

A debugging mode can be enabled by setting following system properties to true:

- `swiftmq.amqp.debug`
- `swiftmq.amqp.frame.debug`

The first one would print additional debug messages, while the second one would print even the sent / received AMQP frames.

4 C++

This chapter contains the guide through the development of Eurex Clearing FIXML/FpML/Margin Calculator Interface client programs in C++ language under the Linux and Windows operating systems using the Apache Qpid C++ API, version 0.32. The library is available for download at <http://qpid.apache.org>. Developing C++ clients using different client libraries is not covered by this chapter. The C++ based client application can be divided into 6 different parts:

1. Environment setup
2. Specifying the destination (message source or target)
3. Preparing connection and session
4. Creating a receiver / sender
5. Receiving / sending of messages
6. Closing the connection

The code examples are simplified (especially error & exception handling and logging) to provide better overview of the functionality. This chapter will contain only code snippets - complete examples are available for download on the Eurex website (see chapter 1.9 for more details).

The Qpid C++ API is also expected to work on other platforms as well – for example on Solaris or AIX. The Qpid C++ API works also on the ARM hardware platform.

4.1 Environment setup under Linux

In order to successfully connect to the Eurex Clearing FIXML/FpML/Margin Calculator Interface, the account certificate (public and private key) as well as a certificate for verification of the AMQP broker has to be passed to the C++ Qpid library. Within Linux operating systems the locations of certificates are passed via exporting the proper environment variables.

How to generate private and public keys is described in “Volume A: Connectivity”. The public keys of Eurex Clearing brokers can be obtained from the public part of the Eurex Clearing website for the Eurex Clearing FIXML Interface and from the Member part of the Eurex Clearing website for the Eurex Clearing FpML Interface and Eurex Clearing Margin Calculator Interface. The following text will assume that the following files are either prepared according to “Volume A: Connectivity” or are downloaded from the Eurex Clearing website:

1. Member’s public key in a printable encoding format according to RFC 1421 standard
`<ABCFR_ABCFRALMMACC1>.crt`
2. Member’s private key in PKCS12 format
`<ABCFR_ABCFRALMMACC1>.p12`
3. Broker’s public key in a printable encoding format according to RFC 1421 standard
`<Broker certificate>.crt`

The C++ Qpid API requires all certificates to be stored in a certificate database created by NSS `certutil` utility. Creating the database and importing the certificates into it can be achieved with few steps.⁸ The database will be stored in the subdirectory `<cert_dir>` of the current directory. Following command creates the empty database. It will ask for a password to protect it.

```
$ certutil -N -d cert_dir
```

Next, the public key of the broker needs to be imported. It will be imported under an alias `<BrokerCertAlias>` into the database.

```
$ certutil -A -d cert_dir -n "<BrokerCertAlias>" -t "P,," -i  
<Broker certificate>.cert
```

The next step is importing Member's public key and creating an alias `<CertAlias>` for it in the database.

```
$ certutil -A -d cert_dir -n "<CertAlias>" -t ",," -i  
<ABCFR_ABCFRALMMACC1>.cert
```

Finally, the last step is to import the Member's private key. The key will be automatically matched with the just imported public part of the pair.

```
$ pk12util -d cert_dir -i <ABCFR_ABCFRALMMACC1>.p12
```

To ensure that the certificate database is prepared correctly, its content can be listed:

```
$ certutil -L -d cert_dir
```

An output like this should be produced:

Certificate Nickname	Trust Attributes
	SSL,S/MIME,JAR/XPI
ABCFR_ABCFRALMMACC1	u,u,u
ecag-fixml-simul	P,,

In the previous example the alias `<CertAlias>` for the Member's certificates is `ABCFR_ABCFRALMMACC1` and the alias `<BrokerCertAlias>` for the broker public key is `ecag-fixml-simul`.

Once the database with certificates is prepared, several Qpid environment variables need to be properly exported. C++ Qpid API uses these variables for establishing the SSL / TLS connection to the broker.

Exporting the following three variables in the Bash Shell will do the job:

```
$ export QPID_SSL_CERT_DB=cert_dir  
$ export QPID_SSL_CERT_NAME=<CertAlias>  
$ export QPID_SSL_CERT_PASSWORD_FILE=<PWD_FILE>
```

⁸ In case the self-signed certificate has been created using the NSS `certutil` utility as described in "Volume A: Connectivity", it is already stored in an existing database. This database can be reused and in such case the only necessary task is to import the broker public keys.

QPID_SSL_CERT variable should point to the directory where the database was created. QPID_SSL_CERT_NAME variable holds the name – an alias for the Member’s certificate which is contained in the database. Finally, QPID_SSL_CERT_PASSWORD_FILE should point to the text file containing the database password which was entered during the initial creation of the database.

4.2 Environment setup under Windows

The SSL / TLS authentication in the C++ client library for Windows is supported since the Apache Qpid release 0.26.

The Qpid client application under the Windows can either use certificates for authentication against the broker from the system’s certificate store or the certificates may be provided to the application from files.

To use certificates from the system’s store, one has to first properly import them. The public key for verifying the identity of Eurex’s AMQP broker has to be always stored in the system’s certificate store. A tool called **certmgr.msc** is a Microsoft Management Console (MMC) snap-in that ships with Windows and can be used to manage the certificate stores for users, computers, and services. First the broker’s public key needs to be imported. It is assumed that the public key is on the file-system in a printable encoding format according to RFC 1421 standard:

<Broker certificate>.crt

Inside the certmgr.msc one expands the *Trusted Root Certification Authorities* store and clicks with right button on the *Certificates* folder. Choosing *Import* then guides us to select the CRT file from the file-system and finally accept trusting the certificate.

Member’s private certificate can be used either from the system’s certificate store or directly from a PKCS12 file.

The Member’s certificate can be imported into the system’s certificate store by double clicking on the file holding the private key in PKCS12 format, e.g.:

<ABCFR_ABCFRALMMACC1>.p12

The popped-up dialog again guides us through the import of the key into the personal registry.

Environment variables have to be used to tell the application which certificate should be used. The variable QPID_SSL_CERT_STORE can be used to configure the store where the certificate was imported. If not specified, it defaults to the “MY” or “Personal” store. The environment variable QPID_SSL_CERT_NAME specifies the certificate which should be used. The certificate is specified using its “friendly name”.

```
set QPID_SSL_CERT_STORE=<CertificateStore>  
set QPID_SSL_CERT_NAME=<friendlyName>
```

For example:

```
set QPID_SSL_CERT_STORE=Personal  
set QPID_SSL_CERT_NAME=CN=ABCFR_ABCFRALMMACC1
```


To use the certificate directly from the PKCS12 file, the environment variable QPID_SSL_CERT_FILENAME has to specify the PKCS12 file, the variable QPID_SSL_CERT_PASSWORD_FILE the password file and the variable QPID_SSL_CERT_NAME the friendly name of the certificate. The password file is a plain text file containing the password to the PKCS12 file.

```
set QPID_SSL_CERT_FILENAME=<certificateFile>
set QPID_SSL_CERT_PASSWORD_FILE=<passwordFile>
set QPID_SSL_CERT_NAME=<friendlyName>
```

for example:

```
set QPID_SSL_CERT_FILENAME=ABCFR_ABCFRALMMACC1.p12
set QPID_SSL_CERT_PASSWORD_FILE=ABCFR_ABCFRALMMACC1.pwd
set QPID_SSL_CERT_NAME=abcf_r_abcf_ralmmacc1
```

4.3 Specifying the destination (addresses)

To describe the message target or message source, C++ based API from the Apache Qpid project uses “Addresses” – a strings passed as a parameters to a receiver or a sender, where they are processed. This chapter will focus on the specific address strings, which can be used to interact with the Eurex Clearing interfaces.

Every application will need 4 different address string types in order to fully utilize the Eurex Clearing interfaces:

- Receiving broadcasts
- Receiving responses
- Sending requests
- “ReplyTo” address in requests

NOTE: The formatting of the sample addresses below is for display purposes only. Actual address strings are formatted as a single line and do not contain line breaks.

4.3.1.1 Receiving responses to requests

Unlike the 0-10 client which is creating the temporary queue and binding it to the response exchange, the 1.0 client simply connects to the predefined queue. As a result the address is not as complicated:

```
<ResponseQueueName>;
{
  create: receiver,
  assert: never,
  node:
  {
    type: queue
  }
}
```

```
}
```

The placeholders in this template have to be replaced with the appropriate values, e.g.:

```
response.ABCFR_ABCFRALMMACC1;
{
    create: receiver,
    assert: never,
    node:
    {
        type: queue
    }
}
```

4.4 Preparing connection and session

A connection is created by instantiating an object of type `Connection` and needs to be initialized using the connection string of the form `"amqp:ssl:<IP/Hostname>:<Port>"`.

```
Connection connection("amqp:ssl:<IP/Hostname>:<Port>");
connection.setOption("reconnect", true);
connection.setOption("transport", "ssl");
connection.setOption("sasl_mechanisms", "EXTERNAL");
```

In the above code snippet the `<IP/Hostname>` and `<Port>` placeholders need to be replaced by the actual hostname and port of the broker.

The C++ client doesn't automatically select the newest supported AMQP protocol. It always connects using AMQP 0-10 by default. In order to connect using AMQP 1.0, the "protocol" option has to be used when creating the connection object:

```
Connection connection("amqp:ssl:<IP/Hostname>:<Port>", "{ protocol: amqp1.0 }");
connection.setOption("reconnect", true);
connection.setOption("transport", "ssl");
connection.setOption("sasl_mechanisms", "EXTERNAL");
```

The protocol has to be specified in the constructor – it cannot be specified using the `setOption(...)` method later.

If a connection is opened using the `reconnect` option, it will transparently reconnect if the connection is lost. The failover behavior can be modified using connection options. More details about the available options can be found in the documentation to the Apache Qpid C++ API. The most important options are:

- **reconnect:** true/false (enables/disables reconnect entirely)
- **reconnect_urls:** list of urls to try when connecting
- **reconnect_timeout:** seconds (give up and report failure after specified time)
- **reconnect_limit:** n (give up and report failure after specified number of attempts)

The application which is interested in automated failover handling between the list of nodes should have option **reconnect** set to *true* and the list of nodes should be passed to the **reconnect_urls** option:

```
connection.setOption("reconnect", true);

connection.setOption("reconnect_urls", "amqp:ssl:ecag-fixml-simul.deutsche-
boerse.com:10170");
```

Eventually, using timeout and limit parameters the application can control how much time is it allowing Qpid library to try another node in the list. The example source code illustrates how to use failover handling using the list of broker nodes.

The failover based on a node list distributed by the broker (using `amq.failover` exchange) is not supported on Eurex Clearing FIXML/FpML/Margin Calculator Interface brokers.

Additionally, the `heartbeat` option can be used to specify the heartbeat interval. Heartbeats are disabled by default. You can enable them by specifying a heartbeat interval (in seconds) for the connection via the `heartbeat` option, e.g.:

```
connection.setOption("heartbeat", 120);
```

With the above option the application requests that heartbeats should be sent every 120 seconds. If two successive heartbeats are missed the connection is considered to be lost. The use of heartbeat is recommended. The recommended heartbeat interval is between 30 and 120 seconds.

After these steps, the connection needs to be opened and new session created:

```
connection.open();
Session session = connection.createSession();
```

4.4.1 Auto reference handling

In the Qpid C++ library, all messaging objects (Connection, Session, Sender, and Receiver) use internal handlers to keep track of references to underlying data. Therefore, it is safe to e.g. create a connection inside some method and return it by value. The copy constructor will automatically increase the internal count. Therefore, the connection will not be closed, if the destructor is called on the method's object (after the return call).

4.5 Creating a receiver/sender

After the connection and session have been prepared, a receiver or a producer can be instantiated. A `Receiver` object can be instantiated using a `createReceiver()` method from the `Session`. The receiver is always bound to a specific destination (`Address`) which was initialized with the proper destination string. In the following code snippets the full destination strings are omitted and represented by the placeholder `<Dest_Address>`. Full addresses can be found in the example source codes.

```
const std::string responseAddress = "<Dest_Address>";
Receiver receiver = session.createReceiver(responseAddress);
```

Receivers can use filters to receive only selected messages. The filter has to be incorporated into the address used to create the receiver. It is added into the address

```
<QueueName>;  
{  
    create: receiver,  
    assert: never,  
    node:  
    {  
        type: queue  
    },  
    link:  
    {  
        selector: \"property = value\"  
    }  
}
```

For example to filter messages based on business date:

```
broadcast.ABCFR_ABCFRALMMACC1.TradeConfirmation;  
{  
    create: receiver,  
    assert: never,  
    node:  
    {  
        type: queue  
    },  
    link:  
    {  
        selector: \"BusinessDate = 20160813\"  
    }  
}
```

Or to filter messages based on correlation ID:

```
response.ABCFR_ABCFRALMMACC1;  
{  
    create: receiver,  
    assert: never,  
    node:  
    {  
        type: queue  
    },  
    link:  
    {  
        selector: \"amqp.correlation_id = '123456'\"  
    }  
}
```

The producer (sender) can be created in a very similar way by instantiating the `Sender` object and initializing it with the proper destination.

```
const std::string requestAddress = "<Dest_Address>";  
Sender sender = session.createSender(requestAddress);
```

4.6 Thread safety

The C++ Qpid client objects are thread-safe (Session, Receiver, Consumer) and therefore it is possible to have two threads sending on the same session. However, it is still recommended to use separate sessions for separate threads.

4.7 Receiving/sending messages

4.7.1 Preparing a request message

To prepare a new message, the `Message` class can be used. For request messages, only the message body and the reply-to key have to be filled. The message body can be entered using the message's method `setContent()` which accepts string representing body as a parameter. The reply to parameter is created by calling the message's `setReplyTo()` method with reply-to address passed as a parameter:

```
Message requestMsg;  
const std::string replyAddress = "<Dest_Address>";  
requestMsg.setReplyTo(replyAddress);  
requestMsg.setContent("<FIXML> ... </FIXML>");
```

4.7.2 Sending a request message

The message prepared in the previous chapter can be sent using the message producer. Since the producer has been initialized with the destination already at the beginning, it is not necessary to use the request destination again. The messages are sent asynchronously by default:

```
sender.send(requestMsg);
```

In order to send the message synchronously, the `sync` parameter of the method `send` should be set to `true`:

```
sender.send(requestMsg, true);
```

When sending the messages asynchronously, the session should be synchronized after every few messages in order to make sure that the requests which were sent asynchronously were delivered to the broker. The session can be synchronized using the `sync` method of the session object:

```
session.sync();
```

The `sync` method will block until the broker confirms that it received and stored all of the messages.

The request queues have only limited capacity and when the queue is almost full a flow control mechanism will be activated by the broker (the exact queue sizes as well as the flow control thresholds for different interfaces can be found in the Volume E of this documentation, chapter 4.1.1.6). When the flow control is activated for the given request queue, the broker will delay sending the confirmations of received messages. That will cause the synchronous `send` calls or `sync` calls to block your application until it can send the next request message. When the messages are sent asynchronously, the client will ignore the flow control measures applied by the broker and can easily exceed the size of the request queue. Therefore in order to avoid exceeding the request queue capacity, the requests should be either sent synchronously or the session should be synchronized often enough to avoid exceeding the queue capacity.

4.7.3 Receiving a message

Messages can be received using the `fetch()` method of the `Receiver` instance:

```
Message msg = receiver.fetch();  
// Processing of the message  
session.acknowledge(msg);
```

Using parameters of the `fetch()` method, the application can either wait until a message is received for a limited (pass the `Duration::<TIME_UNIT>*<TIME>` timeout as a parameter to the method) or unlimited time. The way how to implement an asynchronous message listener using POSIX threads is shown in the example source codes.

The message should be acknowledged after its processing is finished. The acknowledgement can be done using the call of the `acknowledge()` method of the session. Unlike Java Qpid API, the C++ library doesn't support automatic acknowledgement, therefore the client application is always responsible for proper acknowledgement handling.

4.7.4 Message processing

The received message is returned from the `fetch()` method as an instance of the class `Message`. The content of the message can be received in the form of string calling the `getContent()` method.

```
std::cout << msg.getContent() << std::endl;
```

4.8 Closing the connection

When the application is exiting, it should properly close all AMQP related objects. The receivers, producers, session and connection all have a method `close()`, which will properly close them. Closing a connection automatically destroys all underlying sessions and producers/consumers connected to it. However, before closing the connection, the session must be synchronized with the broker:

```
session.sync();  
connection.close();
```

4.9 Compilation and linking on the Linux operating system

The following text assumes the tools used for compilation and linkage of code on the Linux operating system are from GNU Compiler Collection (gcc, g++).

For compilation of the source codes the compiler has to be informed where the Qpid header files are. The linker has to be informed which Qpid libraries the executable needs to be linked with and where to find these libraries. Let us assume the Qpid was installed in the `$QPID_HOME` directory.

The Qpid header files should be then located in `$QPID_HOME/include` while Qpid libraries in `$QPID_HOME/lib`. The location of the header files is passed to g++ via `-I` option, the location of the libraries using the `-L` option.

The following command then compiles `broadcast_receiver.cpp` source code:

```
g++ -I$(QPID_HOME)/include -c broadcast_receiver.cpp
```

And linking the final executable `broadcast_receiver` with proper Qpid libraries is achieved with:

```
g++ -o broadcast_receiver broadcast_receiver.o -L$(QPID_HOME)/lib -  
lqpidmessaging -lqpidtypes
```

In the above example the executable was linked with the Qpid libraries called `qpidmessaging`, `qpidtypes`.

The compiler and linker flags and options might be different on different Linux distributions.

For SSL / TLS authentication to work, Qpid has to be compiled with SASL2 library support (the location of the library has to be recognized during the installation process). On some platforms/environments it may be also required to explicitly invoke loading of the `sslconnector.so` module. To accomplish this, the environment variable `QPID_LOAD_MODULE` should point to the `$(QPID_HOME)/lib/qpid/client/sslconnector.so` library.

4.10 Compilation and linkage under the Windows operating system

Compiling and linking Qpid C++ client programs under the Windows is performed within the Visual Studio.

4.11 Logging

The Qpid C++ clients can both use environment variables to enable logging. Linux and Windows systems use the same named environment variables and values.

Enabling the logging under the Linux and configuring its verbosity can be achieved by setting up environment variable:

```
$ export QPID_LOG_ENABLE=<Level>[+]
```

Where `<Level>` can be one of *trace*, *debug*, *info*, *notice*, *warning*, *error*, or *critical*. Specifying the ending '+' mark will capture all events starting from the `<Level>` and above, while without using the mark one will receive only the events belonging to the selected level. Higher logging verbosity may be especially helpful during the connection troubleshooting.

From a Windows command prompt, use the following command format to set the environment variable:

```
$ set QPID_LOG_ENABLE=<Level>[+]
```

Clients also use `QPID_LOG_OUTPUT` to determine where logging output should be sent. This is either a file name or the special values `stderr`, `stdout`, or `syslog`:

```
$ export QPID_LOG_TO_FILE="/tmp/myclient.out"
```

To control the logging from within the application, the classes `Logger` and `LoggerOutput` from the `qpId::messaging` namespace can be used. More details can be found in the Apache Qpid documentation.

4.12 Error handling

A client application should be designed in such a way that it is resilient to the errors, it does not get stuck when error occurs and at the same time it doesn't start consuming more and more resources. All exceptions the Qpid messaging API can throw are derived from the `MessagingException`. There are a couple of exceptions related to the common type of errors having following base exception:

- **AddressError** - related to processing addresses used to create senders and/or receivers
 - **MalformedAddress** - syntax error in the address
 - **ResolutionError** - error in interpreting address
 - **AssertionFailed** - asserted node properties are not correct
 - **NotFound** - node is not found
- **TransportFailure** - loss of the underlying connection
- **TargetCapacityExceeded** - lack of capacity on queue
- **NoMessageAvailable** – no message on queue

Certain exceptions may render the session invalid; once these occur, subsequent calls on the session will throw the same class of exception. One can test whether the session is valid at any time using the `hasError()` and/or `checkError()` methods on `Session`. Some exceptions may even destroy the connection; to test whether the connection object is still valid, one can call `isOpen()` method.

Generally, the client application should properly check all Qpid C++ API methods for exceptions and in case the exception occurred, an application can, at a minimum, log the problem and clean up its resources. An application can also notify any interest parties that need to be notified of such a problem. An application should be designed with a clean initialization setup, so it would be feasible to reinitialize the objects when the exception occurs.

4.13 Performance

4.13.1 Receive pre-fetching

AMQP brokers typically push messages to client consumers without explicit client requests (asynchronously, in the background), up to a certain number of unacknowledged messages. The next time a message would be passed on to the client application code, it is usually taken from this buffer (avoiding synchronous I/O). This buffering capacity of a client is configurable and it is typically set to hundreds of messages by default. Setting it too low can have a negative impact on message throughput (less overlap of message processing and background I/O). Setting it too high can have a negative impact on client memory consumption (pre-fetch buffers need to hold many messages). Also, all messages pre-fetched by one consumer are “locked” to that consumer (and will not be delivered to any other consumer reading the same queue) until the consumer releases/rejects them. This can lead to a less than ideal load balancing in case of parallel consumption and processing of messages from a single broker queue.

A client normally cannot have more outstanding (unacknowledged) messages than the configured pre-fetch since a broker will stop pushing messages in that situation.

The pre-fetch capacity can be configured for each receiver e.g. (extension of code from section 4.5):

```
receiver.setCapacity(100);
```

4.13.2 Message acknowledgement

Message acknowledgement is asynchronous by default. In case a client application requires synchronous message acknowledgement (e.g. to be absolutely sure that a message was removed from a broker queue before proceeding further), it can be achieved by explicitly setting the second parameter of the call to ‘true’ (‘false’ when omitted):

```
session.acknowledge(msg, true);
```

When using explicit acknowledgement of received messages (as described in section 4.7.3), doing one-by-one synchronous acknowledgement of messages can severely degrade performance. In that case, message consumption rate cannot exceed the inverse of the network round-trip latency between the client and the broker. For example, a round-trip time of 100ms would limit the synchronous throughput to no more than 10 messages per second.

Message receiver performance in the synchronous acknowledge mode can be improved via block processing of messages. The following code is an extension of the code from section 4.7.3 and illustrates block processing. It will receive and process up to 100 messages, then finally acknowledge all received messages in a single call (thus mitigating the impact of network latency):

```
int maxCount = 100;
msg::Message msg;
// receive up to 100 messages
while (receiver.fetch(msg, msg::Duration::IMMEDIATE))
{
```

```
// process the message
std::cout << "Message: " << msg.getContent() << std::endl;
// check total block size
if (--maxCount <= 0)
{
    break;
}
}
// acknowledge all previously fetched messages
session.acknowledge(true);
```

4.13.3 Sender synchronization

When sending messages (as shown in section 4.7.2), the send call is asynchronous by default. This means that a message being sent is not guaranteed to arrive at a broker by the time the send call returns in a client application. Any send call can be made synchronously (i.e. wait for message delivery confirmation) via the second parameter of the call:

```
sender.send(requestMsg, true);
```

The above will implicitly synchronize any messages previously sent (asynchronously) via the same session (in addition to the message being sent). However, this can have a significant negative impact on performance due to network latency between the client and the server. It is similar to the synchronous acknowledge after every message discussed in section 4.13.2.

It is also possible to explicitly synchronize the session:

```
session.sync();
```

This will synchronously wait until the client receives delivery confirmations for all messages previously sent via the session. This way, clients can employ (reliable) block/burst message sending.

5 .NET

5.1 Apache Qpid Messaging API

The .NET Binding for the C++ Qpid Messaging Client is a library that gives any .NET program access to Qpid C++ Messaging objects and methods. These bindings provide mappings between classes in the .NET Binding and the underlying C++ Messaging API. Since the .NET binding provides only a wrapper around the C++ library, the .NET library supports all features of the C++ library including the AMQP 1.0 support.

The binding consists of following two components:

- **.NET Messaging Binding Library**
 - Provides access to unmanaged, native Qpid Messaging C++ core run time system
- **.NET Messaging Managed Callback Library**
 - An extension of the .NET Messaging Binding Library that provides message callbacks in a managed .NET environment.

The mechanism of creating the Connection, Session, Receiver/Producer objects is the same as in the underlying C++ API with the respect of .NET naming conventions and garbage collection. Regarding receiving messages, the .NET callback library provides additional mechanism for asynchronous message reception by the registered callback object (implementing the `ISessionReceiver` interface). The asynchronous receiver is used in the "BroadcastReceiver" source code example (see chapter 1.9 for more details).

.NET bindings propagate all the underlying Qpid exceptions from the C++ layer in one common `QpidException`. Therefore, one cannot distinguish in the common catch block in which situation was the exception thrown and what exactly was caused by this exception (e.g. when the thrown exception invalidates the session or even the connection). It is recommended to check the session (`Session.HasError`) and connection (`Connection.IsOpen`) and eventually to re-initialize them.

5.2 AMQP.NET Lite

AMQP.NET Lite is an AMQP 1.0 client library written in pure .NET. It can be downloaded from <https://github.com/Azure/amqpnetlite> - version 1.1.2 or newer is required to connect to Eurex Clearing interfaces.

5.2.1 Environment setup

AMQP.NET Lite client is using Windows certificate store as the main source of certificates. The Eurex Clearing public key has to be present in the "Trusted Root Certification Authorities" store and the member's private key has to be stored in the "Personal" store. To import the public / private keys, the Windows Certificate Manager utility can be used (`certmgr.msc`).

Additionally to the certificates stored in the Windows Certificate Store, the public key belonging to the member certificate has to be stored in a file. This file will be loaded by the client application and used to tell the client library which private key should be used.

5.2.2 Preparing Connection and Session

The connection is created using the `ConnectionFactory`. Before opening the connection, the factory has to be configured to support SSL.

```
ConnectionFactory factory = new ConnectionFactory();
```

The public key of the member certificate has to be added to the `ClientCertificates`. It will use the file which was prepared in previous chapter:

```
factory.SSL.ClientCertificates.Add(X509Certificate.CreateFromCertFile("c:\\path\\to\\certificate\\ABCFR_ABCFRALMMACC1.crt"));
```

By default, the library would select the SSL certificate for the client authentication based on the list of supported certification authorities as published by the AMQP broker. This would not work on the Eurex Clearing interfaces because only self-signed certificates are used. The

`LocalCertificateSelectionCallback` has to be used to select the proper certificate.

```
factory.SSL.LocalCertificateSelectionCallback = (a, b, c, d, e) =>  
X509Certificate.CreateFromCertFile("c:\\path\\to\\certificate\\ABCFR_ABCFRALMM  
ACC1.crt");
```

To validate the broker certificate, the `RemoteCertificateValidationCallback` should be configured. A new method for the server certificate validation has to be created:

```
public static bool ValidateServerCertificate(object sender, X509Certificate  
certificate, X509Chain chain, SslPolicyErrors sslPolicyErrors)  
{  
    if (sslPolicyErrors == SslPolicyErrors.None  
        {  
            return true;  
        }  
  
    Console.WriteLine("Certificate error: {0}", sslPolicyErrors);  
    return false;  
}
```

This method would let the server certificate validation pass only in case there were no errors. The method has to be used in the callback:

```
factory.SSL.RemoteCertificateValidationCallback = ValidateServerCertificate;
```

Additionally to the SSL settings, the maximal frame size should be set to 64kB and the SASL mechanism has to be set to EXTERNAL.

```
factory.AMQP.MaxFrameSize = 64 * 1024;  
factory.SASL.Profile = SaslProfile.External;
```

The factory can be also used to configure `idleTimeout`. Idle timeout needs to be specified in number of milliseconds. The recommended `idleTimeout` interval is between 30000 and 120000 milliseconds.

```
factory.AMQP.IdleTimeout = 60000;
```

When the factory is properly configured, it can be used to create the connection. The broker address is in the format "amqps://<hostname>:<port>", for example "amqps://ecag-fixml-simul.deutsche-boerse.com:10170".

```
Address brokerAddress = new Address("amqps://ecag-fixml-simul.deutsche-boerse.com:10170");
Connection connection = await factory.CreateAsync(brokerAddress);
```

With the connection ready, the session can be opened:

```
Session session = new Session(connection);
```

5.2.3 Receiving / Sending messages

5.2.3.1 Creating receiver / sender

`ReceiverLink` and `SenderLink` classes represent the message receiver and sender. They both accept three different parameters:

1. The AMQP session
2. The name of the receiver / sender link
3. The address where the messages should be received from / sent to

The address is always the queue name, for example:

```
SenderLink sender = new SenderLink(session, "request-sender",
"request.ABCFR_ABCFRALMMACC1");
ReceiverLink receiver = new ReceiverLink(session, "response-receiver",
"response.ABCFR_ABCFRALMMACC1");
```

5.2.3.2 Using filters

The `ReceiverLink` can be configured to use AMQP filters. The `FilterSet` has to be prepared containing all filters which should be used when creating the receiving link. To use filters, different `ReceiverLink` constructor has to be used, which allows passing the whole message source and not only the address:

```
Map filters = new Map();
filters.Add(new Symbol("apache.org:selector-filter:string"), new
DescribedValue(new Symbol("apache.org:selector-filter:string"),
"amqp.correlation_id='123456'"));
ReceiverLink receiver = new ReceiverLink(session, "response-receiver", new
Source() { Address = "response.ABCFR_ABCFRALMMACC1", FilterSet = filters},
null);
```

5.2.3.3 Preparing a request message

The `Message` class is used to create a new message. Additionally to the message payload, the `ReplyTo` property has to be set to `"response/response.ABCFR_ABCFRALMMACC1"` so that the response message can be routed back to the client.

```
Message request = new Message("Hello world!");  
request.Properties = new Properties();  
request.Properties.ReplyTo = "response/response.ABCFR_ABCFRALMMACC1";
```

5.2.3.4 Sending a request message

The message can be sent using the `Send` method:

```
sender.Send(request);
```

The `Send` method will send the message synchronously. There is additional method `SendAsync` to send messages asynchronously.

5.2.3.5 Receiving a message

Messages can be received using the `Receive` method of the `ReceiverLink` object. The parameter defines the timeout for which the receiver will wait for new message to arrive.

```
Message response = receiver.Receive(60000);
```

In case no message is received during the timeout interval, `null` value will be returned.

5.2.3.6 Message Processing

The messages received from Eurex have the payload encoded as single binary data section. To decode it, the `BodySection` has to be first converted to the `Amqp.Framing.Data` and afterwards to UTF string:

```
Amqp.Framing.Data payload = (Amqp.Framing.Data) response.BodySection;  
String payloadText = Encoding.UTF8.GetString(payload.Binary);
```

5.2.4 Closing the connection

To close the connection, call the `Close` method:

```
connection.Close();
```

5.2.5 Logging

The client can trace the AMQP protocol frames. To switch it on, the trace level and the listener have to be configured:

```
Trace.TraceLevel = TraceLevel.Frame;  
Trace.TraceListener = (f, a) => Console.WriteLine(String.Format(f, a));
```

6 Python

The API that supports AMQP 1.0 is a Python wrapper around the Apache Qpid C++ library. Just like the C++ library it supports 1.0 protocol. This library is available in the `qpid_messaging` package (or `qpid` package in older Qpid versions). While the library interface is slightly different from the C++ library, it is using the same SSL implementation as the C++ API. Therefore, the certificate formats as well as the environment setup are identical.

APIs is using the same addresses to identify the message sources and targets as the C++ clients.

The detailed description of the Python libraries and their interfaces is not part of this documentation. However, simple programs for receiving broadcasts, sending requests and receiving responses using both these libraries are part of the code examples – see chapter 1.9 for more details.

7 Troubleshooting

7.1 Errors

During a message exchange between a client and the broker several error situations may occur due to a misconfiguration or malfunctioning software.

7.1.1 Connection failure

The following reasons can lead to failure to establish a connection with the broker.

- Host unreachable
- Invalid host certificate
- Invalid client key

7.1.2 Too many connections

When the limit of maximum number of connections is reached.

```
Permission PERFORM_ACTION(connect) is denied for : VirtualHost 'default' on  
VirtualHostNode 'default' [condition = amqp:not-allowed]
```

7.1.3 Unknown destination

When a request is sent to an invalid address.

```
Unknown destination 'request.ABCFR_TESTCALMMACC1X' [condition = amqp:not-  
found]
```

7.1.4 Invalid destination

When a request is sent to an address to which the client does not have the right to publish.

```
Permission PERFORM_ACTION(publish) is denied for : Exchange  
'request.ABCFR_TESTCALMMACC2' on VirtualHost 'default' [condition = amqp:not-  
allowed]
```

7.1.5 Non-existent queue

Attempt to consume a message from a non-existent queue

```
Could not find destination for source  
'Source{address=broadcast.ABCFR_TESTCALMMACC1.PublicX,durable=none,expiryPolic  
y=link-  
detach,dynamic=false,defaultOutcome=Modified{deliveryFailed=true},outcomes=[am  
qp:accepted:list, amqp:released:list,  
amqp:rejected:list],capabilities=[queue]}' [condition = amqp:not-found]
```


7.1.6 Invalid queue

Attempt to consume a message from a queue which the client does not have the right to consume.

```
Permission CREATE is denied for : Consumer '17|1|qpId-  
jms:receiver:ID:bc025dfc-ac00-42aa-95d1-  
62f07dafa0ac:1:1:1:broadcast.ABCFR_TESTCALMMACC2.Public' on Queue  
'broadcast.ABCFR_TESTCALMMACC2.Public' [condition = amqp:unauthorized-access]
```

7.1.7 Full queue

7.1.7.1 Message count limit

When a request queue message count limit is reached.

```
Maximum depth exceeded on 'request_be.ABCFR_TESTCALMMACC1.C7' :  
current=[count: 6001, size: 4212630], max=[count: 6000, size: 6144000]  
[condition = amqp:resource-limit-exceeded]
```

7.1.7.2 Byte size limit

When a request queue byte size limit is reached.

```
Maximum depth exceeded on 'request_be.ABCFR_TESTCALMMACC1.C7' :  
current=[count: 4536, size: 6144580], max=[count: 6000, size: 6144000]  
[condition = amqp:resource-limit-exceeded]
```

7.2 Lost connection

It can happen that a connection between the broker and the client can be lost. To detect such failure client applications are advised to specify *idle-timeout* which will enable a *heart-beat* mechanism on the established connection. This way if a connection is lost both client and the broker can detect it and act accordingly.

In case of connection loss client can choose to reconnect automatically. If a connection loss is a result of a broker technical maintenance, it can be expected that the broker may not be available for several minutes. To cover such situation, it is suggested to automatically retry to connect every minute for at least 30 minutes.

8 Glossary of terms and abbreviations

<i>Term / Abbr.</i>	<i>Definition</i>
AMQP	Advanced Message Queuing Protocol - standard for Messaging Middleware.
Apache Qpid	Open source implementation of AMQP protocol
Binding	A binding is a relationship between a message queue and an exchange. The binding specifies routing arguments that tell the exchange which messages the queue should get.
Broker	AMQP middleware messaging server
Eurex System	Eurex hosts
Exchange	An exchange accepts messages from a producer application and routes them to message queues according to prearranged criteria.
EXTERNAL authentication	AMQP authentication mechanism based on SSL / TLS certificates
FIX	The Financial Information Exchange Protocol
FIXML	FIX business messages in XML syntax
FpML	Financial products Markup Language is the industry-standard protocol for complex financial products. It is based on XML.
Message	A message is the atomic unit of routing and queuing. Messages have a header consisting of a defined set of properties, and a body that is an opaque block of binary data.
Queue	A message queue stores messages in memory or on disk, and delivers these in sequence to one or more consumer applications. Message queues are message storage and distribution entities. Each message queue is entirely independent.
Routing key	A message property used in bindings to specify the exchange – queue relationship.
SASL	Simple Authentication and Security Layer
SSL	Secure Sockets Layer – cryptographic protocol designed to provide communication security over the Internet
TLS	Transport Layer Security – cryptographic protocol designed to provide communication security over the Internet and successor to SSL protocol.
XML	Extensible Markup Language