DEUTSCHE BÖRSE

DFS500 AMQP Public Message Interface - CMM

XBID

Version	V 9.00
Status	Final
Filename	DFS500 AMQP Public Message Interface – CMM V9.00
Date	18/01/2018
Author	XBID Project Team
Reviewer	XBID Project Manager

Deutsche Börse AG

Mailing Address
Mergenthalerallee 61
65760 Eschborn

Web
www.deutsche-boerse.com

Table of Contents

1 Introduction

1.1 Purpose

This document describes the Public Message Interface (PMI), called CATRINA (Capacity Trading Interface for Applications), that the Capacity Management Module (CMM) provides to its clients.

The CMM PMI allows clients to communicate with the Capacity Management Module via a programmable interface.

The communication with CMM is based on *Advanced Message Queuing Protocol* (AMQP) as the transport layer. AMQP is a platform- and language-neutral open standard for the wire protocol. CMM is currently using AMQP implementation from RabbitMQ version 3.2.3, which supports AMQP versions 0-9-1, 0-9 and 0-8. See www.amqp.org and www.rabbitmq.com for more information.

Payload of the messages sent over AMQP is formatted in XML. See www.w3.org/XML for information about XML.

1.2 Overview

Clients using the PMI communicate with an AMQP server, which in turn communicates with CMM. CMM itself is behind a firewall and is not directly accessible for the clients.

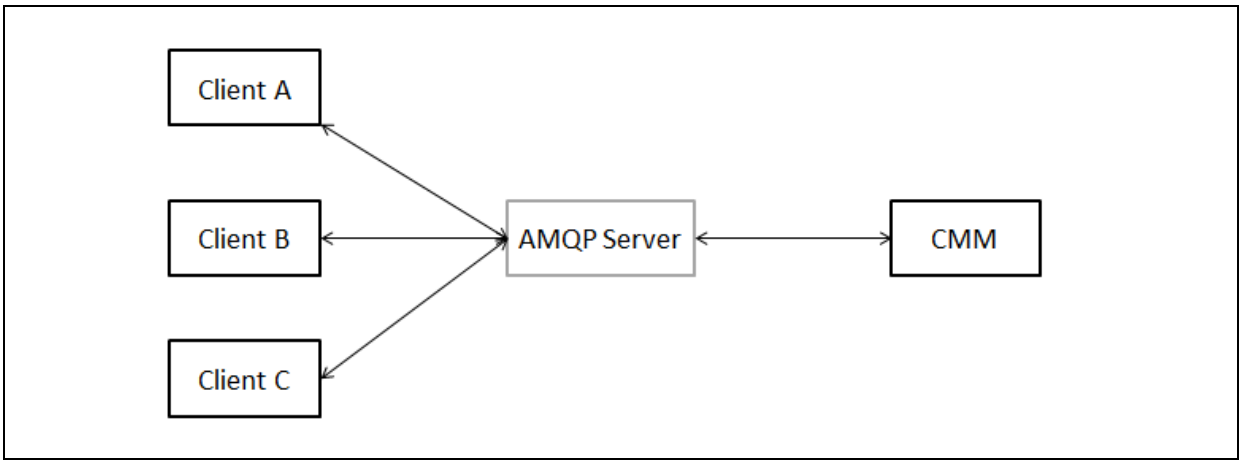

Figure 1: Overview of communication between clients and the Capacity Management Module

The next section drafts the first steps required to get started with the CMM PMI.

Section 3 Message Exchange Patterns describes how clients communicate with CMM.

Section 4 Security describes encryption of the communication between clients and the AMQP server as well as client authentication and authorization.

The types of messages sent between the client and CMM, their content and formatting are specified in section 5 Message Payload Format.

1.3 Referenced Documents

This document refers to:

- PPI160 Key Project Terms and Glossary
- HLS100A Functional Description Capacity Management Module

2 Getting Started

To get started with the CMM PMI, a user needs to:

- Request an account on the AMQP server at the granting authority. The customer will obtain the following package:
  - A SSL client certificate and password to connect to the AMQP Server (see section Security)
  - The source code of the reference client implementation
  - The message format defined by the XML Schema files (the files are part of the source code for the reference client implementation). The customer will need these schema files to be able to correctly format messages sent to the CMM System and read its responses. See section for details. See section Message Payload Format for details.

- Download the latest AMQP client library for RabbitMQ from www.rabbitmq.com (currently RabbitMQ officially provides client libraries for Java, .NET and Erlang). Libraries from other AMQP vendors are not supported due to interoperability issues (vendor interoperability expected since AMQP version 1-0).
- Implement a client. How a participant's program should communicate with CMM is described in section 3 Message Exchange Patterns; the message format is specified in detail in section 5 Message Payload Format .

2.1 AMQP

AMQP (Advanced Message Queuing Protocol) is wire-level protocol that enables message-based communication through the use of an AMQP compliant middleware server (the message broker). In the case of CMM this message broker is RabbitMQ.

AMQP defines a modular set of components for the broker as well as standard rules for connecting them. There are 3 main types of components, which are connected into processing chains to create the desired functionality:

- The **"exchange"** receives messages from publisher applications and routes these to "message queues" based on arbitrary criteria, usually message properties or content
- The **"message queue"** stores messages until they can be safely processed by a "consuming" application (or multiple applications)
- The **"binding"** defines the relationship between a message queue and an exchange and provides the message routing criteria

The exchanges, message queues and bindings that are set up by the CMM system are described in more detail in the following chapter.

Please refer to the AMQP and RabbitMQ documentation for more details on the use and configuration capabilities of both AMQP and the RabbitMQ client library.

2.2 Connecting to Server

The first step every client program needs to take is to establish a connection to the AMQP server. This connection can then be used to create channels that are used for communication with the server. All threads of a client program typically share the same connection.

When creating a connection, it is necessary to specify the following:

- server host, port and virtual host to connect to
- SSL context: client certificate, client private key and a trusted root CA certificate

While creating a connection, it is quite important that an External authentication mechanism is chosen. It is done with

ConnectionFactory.setSaslConfig(DefaultSaslConfig.EXTERNAL)

It is recommended but not required to set the user name while creating the connection

ConnectionFactory.setUsername(userName)

It is necessary that all connections to the AMQP server are created with the AMQP heartbeats enabled. The recommended heartbeat period is 30-60 seconds. Connections that do not have heartbeats enabled will time out on the firewall in case there is no traffic on the connection for a longer period of time.

In the RabbitMQ Java client, the AMQP-level heartbeats are enabled using method *ConnectionFactory.setRequestedHeartbeat(int)*, specifying the heartbeat interval length in seconds.

In addition, client programs may want to register a shutdown listener on the connection or on the channels. This can help the application to detect connection loss faster.

2.3 CMM PMI User Roles

Depending on the purpose of the connection two PMI roles are available:

- Explicit Participant PMI
- Allocation Reporting PMI

The allocation reporting functionality allows the client to subscribe to a private steam via the PMI and to request and receive information from CMM about all allocations made by their balancing group.

The explicit participant PMI functionality allows the client to subscribe to a border they are interested in, provided they have access to it. Once a subscription to the border is done the client can:

- Receive capacity information based on predefined triggering events,
- Send allocation requests in XML format,
- Receive allocation confirmation in XML format,
- Request and receive allocation reports.

2.4 Time Zones

CMM offers capacity across several time zones. To insure consistency the PMI defines all time attributes, including start and end time of the allocation requests, in UTC.

3 Message Exchange Patterns

Two basic patterns of message exchange between the client and CMM are supported:

- *Request-response* communication, where the client issues a request and waits for a response from CMM. Each response message is addressed to a specific client (the one that issued the request). This type of communication is initiated by the client.
- *Broadcast* communication, where CMM publishes notifications that are not addressed to any specific client. Clients may subscribe to receive notification broadcasts they are interested in. This type of communication is initiated by CMM.

Details of the messages are described in the following sections.

3.1 Broadcasts

Broadcast messages sent by CMM are distributed (pushed) via the AMQP server to all clients that have subscribed to receive the information.

3.1.1 Broadcast Types

CMM broadcasts different information depending on the PMI user roles.

The *Hearbeat* broadcast, which is sent in regular intervals, is supported for both roles. It allows clients to monitor the availability of the application and is described in detail in the following section.

3.1.1.1 Heartbeat Broadcasts

A durable fanout exchange named *cmm.heartbeat* is used for broadcasting heartbeat messages. This exchange is accessible for all clients.

Each heartbeat message contains information about the heartbeat interval length (in the "X_Period" header) as well as a timestamp of when the message was sent (in the body). This allows the clients to monitor the availability of CMM and calculate message latency.

Please note the difference between application-level heartbeat broadcasts sent by CMM and AMQP-level heartbeats:

- AMQP heartbeats are defined by the AMQP protocol specification and allow the client to monitor the existence of a connection between the client and the AMQP server.
- Heartbeat broadcasts published by CMM are proprietary to the CMM PMI and allow the client to monitor the availability of the application.

3.1.1.2 Explicit Participant PMI Broadcasts

CMM broadcasts the following information to the Explicit Participant PMI users:

- *ATC Data* notifications inform clients about changes in the available transfer capacity (ATC).

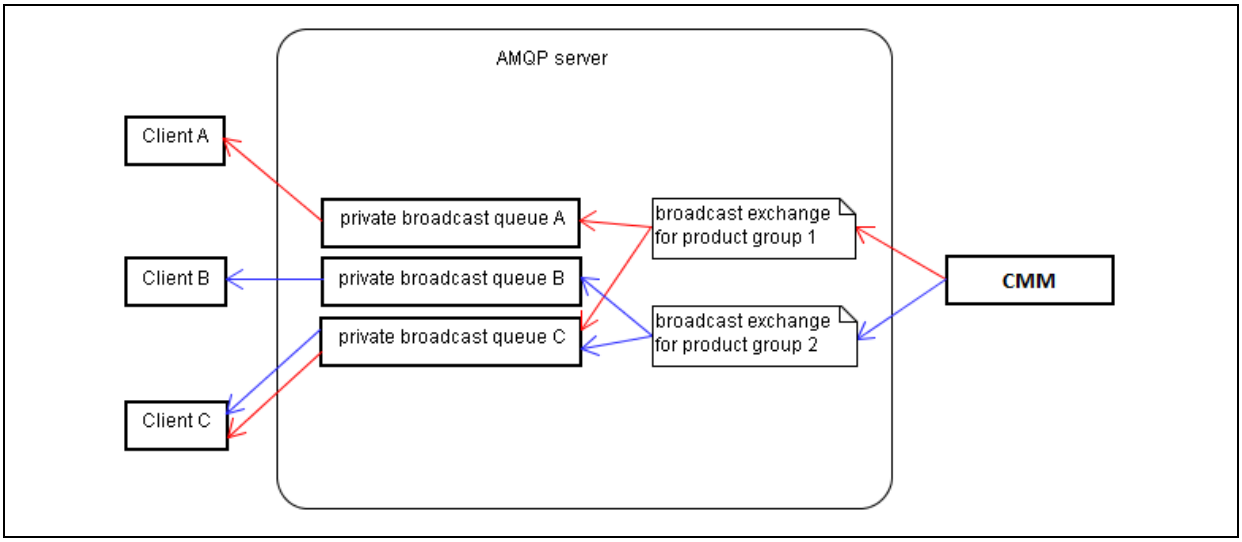

Figure 3: ATC broadcast communication schema from the CMM to the Explicit Participant PMI.

3.1.1.2.1 ATC Broadcast

A header exchange named *cmm.atc.<border>* is set up on the AMQP server for each border. CMM sends messages to these broadcast exchanges whenever it needs to publish new ATC information to clients.

The following header attributes are used for the ATC Data Notifications:

- "X_Border" specifying the short name of the border as defined on the Reference Data GUI, e.g. DE-FR;
- "X_OutArea" specifying the EIC code of an outbound area (TSO);
- "X_InArea" specifying the EIC code of an inbound area (TSO);
- "X_Day" specifying the date for which the ATC information is broadcasted;
- "X_Event" specifying the event, which triggered the broadcast. Possible values are: PUBLISH, ADJUST_CAPACITY, ON_BEHALF, ALLOCATION, ALLOCATION_PMI.

The ATC Data Notifications are sent immediately after a triggering event happens and is always in a compressed format.

3.1.1.3 Allocation Reporting PMI Broadcasts

For the purpose of the allocation reporting CMM broadcasts the *Allocation Broadcast* as described in the following subchapter.

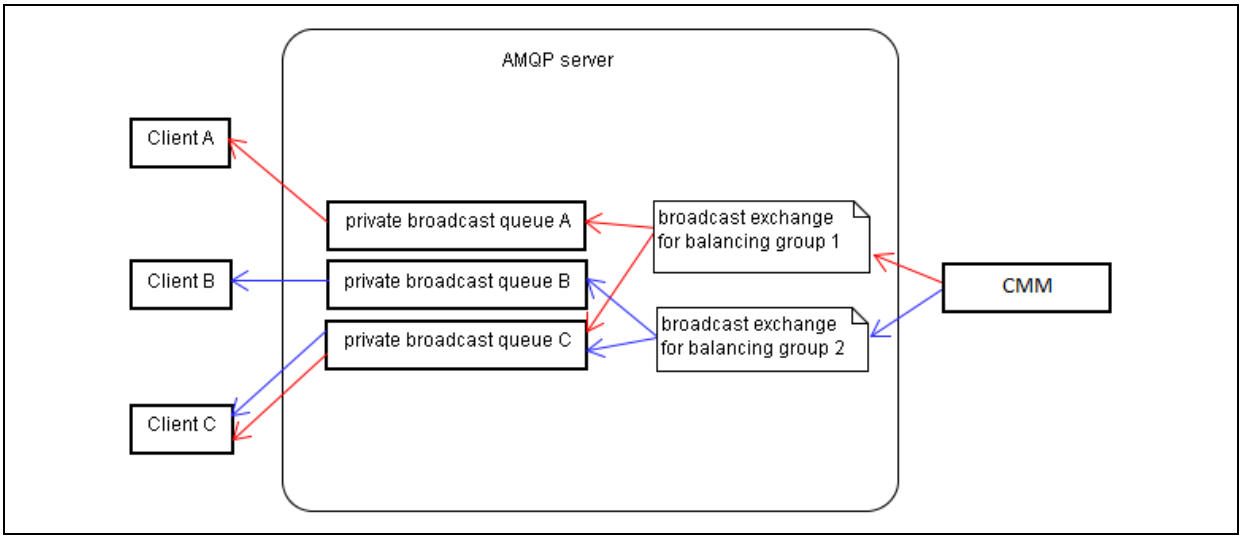

Figure 4: Allocation broadcast communication schema from the CMM to the Allocation Reporting PMI.

3.1.1.3.1 Allocation Broadcast

A header exchange named *cmm.allocation.<balancing_group>* is set up on the AMQP server for each balancing group. CMM sends messages to these broadcast exchanges whenever the respective balancing group performs an allocation in a compressed format.

The following header attributes are used for the Allocation Broadcasts:

- "X_BG" specifying the EIC code of the specific Balancing Group;
- "X_OutArea" specifying the EIC code of an outbound area (TSO);
- "X_InArea" specifying the EIC code of an inbound area (TSO);
- "X_Event" specifying the event, which triggered the broadcast. The Allocation Broadcast is sent immediately after a triggering event happens. Possible values are: PUBLISH, ADJUST_CAPACITY, ON_BEHALF, ALLOCATION, ALLOCATION_PMI;
- "X_Sequence_Id", which is used to identify the order of the broadcasts and to find out if some broadcasts have been lost. The sequence will always be increased by one for the next broadcast. It will be in-memory only (NOT persistent) which means that when CMM shuts down or terminates, the sequence will be reset to 0. This is sufficient and whenever the client gets a value which is not expected (i.e. value different than last_value+1) they should request the allocation data for the respective period.

3.1.2 How to Receive Broadcasts

By default, each client is subscribed to all broadcasts that are available for their role, as long as they are authorized for it. So after the Login request is received, the system creates the user's private broadcast queue. Whenever a message is broadcasted by CMM, a copy of the message is placed into the broadcast queue and the call back method is invoked on the registered consumer.

To stop receiving all broadcasts, the client should cancel its consumer. The AMQP server will automatically delete the client's private queue (after a configured period).

3.1.2.1 Filtering Broadcasts

Should a client wish to receive only a subset of broadcasts, they must provide binding arguments to the login request. Since CMM creates the broadcast queue and all bindings for the client, this is the only way the clients can filter what messages to receive.

The binding arguments use a comma separated list of pairs "header-name: expected-value" with the following operators:

- Separator for different arguments : ",",

- Separator for argument key and argument value ":" ,
- Separator for multiple arguments values "|".

The concrete headers are mentioned in the separate broadcast chapter.

For example the argument X_Border:DE-FR|AT-DE, X_Event:ALLOCATION" means that the client will receive only ATC broadcasts (no other broadcasts have 'X_Border' header), that the broadcasts will be delivered only for DE-FR and AT-DE borders and only in case of a new allocation.

The only exception to the rule above is the argument "x-match" which is not a message header but rather a standard AMQP binding argument. When the "x-match" argument is set to "any", just one matching header value is sufficient. Alternatively, setting "x-match" to "all" mandates that all values must match.

Leaving the "*arguments*" attribute empty means that there are *no binding restrictions* and the client will receive all broadcasts.

3.1.3 Failover Processing

In case of AMQP server (broker) shutdown (due to failure or restart), the client subscriptions are lost. If the client has registered a shutdown listener, it will receive a shutdown notification from the broker. After successful reconnect to the broker, the clients have to re-subscribe.

In case of the application server failure, the client subscriptions will stay active, but clients will not receive any broadcasts until the server is restarted. Once the application is restarted, delivery of broadcasts will resume with no further actions required on the client side.

Any broadcast messages published by CMM while a client was disconnected will be lost for that client.

3.1.4 Flow Control

When a client is consuming messages too slow, a backlog of messages waiting for processing may build up in the private queue(s) owned by this client.

The clients are therefore required to consume (and acknowledge, if applicable) each message as soon as possible. If the processing of a message is a non-trivial task, the receiver must accept responsibility separately from the actual processing of the message: the message must be consumed, acknowledged and stored in a memory buffer of the client, where it awaits processing.

It is important that clients consume messages as quickly as possible. Should a client fail to adhere to these rules and cause technical problems due to slow consuming, it may be denied access to the PMI by disabling its account on the AMQP server.

Here again there is a queue expiration timer if the connection between the AMQP server and CMM is down. Any disconnection for longer than a specified time period (for example 30 seconds) would lead to expiry of the queues and deletion of the information within the queues.

3.2 Request-Response

Clients may manage their capacity allocations by sending requests to CMM. Requests are sent to private client-specific request exchanges. After processing a request, CMM sends the response to a private response queue that belongs to the client that has sent the request.

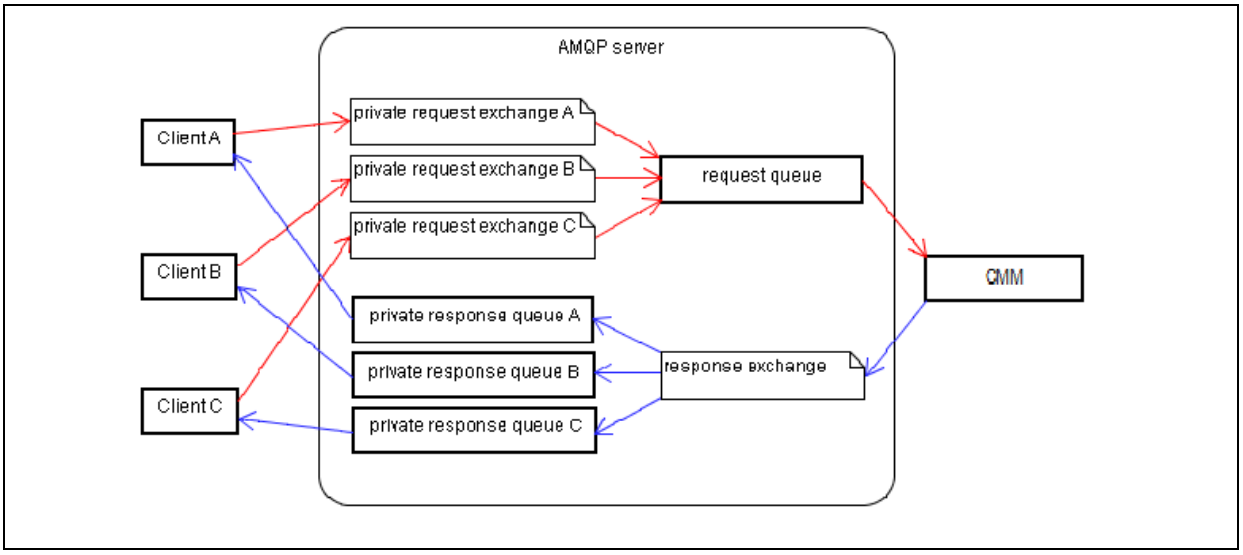

Figure 2: Request-response communication between clients and the CMM

A durable fanout exchange named *cmm.request.<user_name>* is set up on the AMQP server for each client. These exchanges are used for client requests of all request types.

Always, the *Login request* should be the first message to be sent to CMM. Only after this request, the application can process any other (type of) request. See the following chapter.

3.2.1 Request Types

CMM supports different request types based on the PMI user role. The *Login* request type is available for all users.

The *Login* request is used to connect to the message interface. Upon a successful processing of the Login request, two queues are created – cmm.broadcastQueue.<user_name> and cmm.responseQueue.<user_name>. The Login-response will then be sent to the response-queue. The response-queue name should be used in the reply-to header for all requests. For a description of how request details are represented in the message payload, see chapter 5 Message Payload Format.

3.2.1.1 Explicit Participant PMI Request Types

For the purpose of the Explicit Participant PMI user, CMM supports the following request type categories:

- **Management requests:** The management requests are modifying data (updating the CMM database) and are handled by the Core itself. It processes all management requests such as Allocation Request.

3.2.1.1.1 Inquiry requests:

The inquiry requests are read-only and are handled by the CMM module. It processes requests like ATC Data and Allocation Data. Further information about the request are displayed in the subchapter below. Allocation requests

Within the *Allocation Requests* it is possible to send multiple requests of the same type in a single message. CMM returns a collection of results, containing one result for each individual request.

CMM uses a configurable limit for the maximum allowed number of items that may appear in a single request. Should this limit be exceeded, the request will be rejected and the client will get an error response containing information about the current limit setting. Note that this is not a limitation imposed by the PMI, but a feature of CMM itself.

When allocating capacity the request must specify a client ID which is sent back in the response and stored together with the allocation in CMM's database (unlike correlation-id which is only for client's reference). Ideally, the clients should use unique client IDs, however CMM does not validate it, i.e. if two requests with the same client ID are received CMM will process them as separate requests and they might result in separate allocations.

3.2.1.1.2 Inquiry Requests

The following inquiry request types are supported:

- *ATC Data Request* is used to retrieve information about the current available transfer capacity.

For inquiry requests, it is possible to send a maximum of one request in each message. CMM returns a single response for each inquiry request.

3.2.1.2 Allocation Reporting PMI Request Types

For the purpose of the Allocation Reporting PMI user CMM supports the *Allocation Data Request*. It is used to retrieve a complete list of all allocations within given timestamps. The amount of delivery days for which the allocation data can be requested, is configurable via the Reference Data GUI (see chapter 8.1.1 Create Balancing Group in DFS700 Reference Data Module).

3.2.2 How to Send Requests

The queues for each client are created automatically after the login to the PMI (i.e. successful processing of the Login request); therefore a client that wants to send a request to CMM has to do the following:

- Use a *return listener* for the AMQP channel to be able to detect unroutable requests.
- Create a message and put the XML-encoded request into the message body.
- Set message attribute *content*-type. This allows the system to detect the correct message body structure definition. I.e. the version of XML schema. Currently the only supported one is “text/xml;v=1.0”.
- Set message attribute *correlation-id* to a unique request ID that can be used to match responses with requests.
- Optionally, set message attribute *expiration* which defines the TTL (Time To Live) period in milliseconds (see details below).
- Send the message to exchange *cmm.request.*<*user_name*> with routing key *capacity.request* with *mandatory flag*. Please avoid using the *immediate* mode since it is not supported by RabbitMQ server since version 3.0.

  Note: There are different routing keys depending on the request type category:
  - Management request: cmm.request.management
  - Inquiry request: cmm.request.inquiry

Once CMM processes the request, it will send a response to the client’s response queue. The response message is sent with attribute *correlation-id* set to the value contained in the request. This allows the client to match responses with requests.

CMM will send a response to each request. As long as the XML schema version used by the client is supported by CMM, the response will be encoded using the same schema version that was used for the request.

For details about the request and response formats, see section 5 Message Payload Format.

3.2.2.1 Message Expiration

By specifying message expiration, the client may ensure that the request is discarded if it is not consumed by CMM within the specified time period.

The value of the expiration property describes the TTL (Time-To-Live) period in milliseconds and if the message is not consumed within the given period it will not be delivered to CMM. This is fully supported by RabbitMQ since version 3.0. See http://www.rabbitmq.com/ttl.html.

3.2.2.2 Invalid and Unroutable Requests

If CMM cannot process a request because the request is incorrect, a duplicate or cannot be fulfilled, it will send a negative response. The response message contains details about why the request could not be processed.

If CMM cannot process the request because the XML schema version in the request message header is missing or invalid, it will send an error response encoded using the default schema version.

If CMM cannot process the request because its XML code is invalid, it will send a special error response.

If CMM cannot process the request because the platform is down, the request message will be discarded by the AMQP server. In this case the client will be notified about this via its return listener (but only if the message has been sent with the mandatory flag).

3.2.3 How to Receive Responses

Technically, the client can receive a response in two ways:

- Synchronously using *Channel.basicGet*. This method requires that the client keeps polling the response queue.
- Asynchronously using a *consumer*. The client registers a consumer for the response queue and AMQP invokes the consumer when a message arrives. This method allows the client to wait passively without consuming CPU.

Within the CMM Public Message Interface clients shall receiving messages asynchronously using a consumer because it reduces message latency and resource consumption. The maximum wait time should be limited, because it is possible that the response never arrives (see chapter 3.2.4 Failover Processing below).

Once a response is received, the client should extract the following message attributes:

- Attribute *content-type* containing the XML schema version used for encoding of the response. This allows the client to choose the correct XML schema for XML unmarshalling.
- Attribute *correlation-id* containing the correlation ID from the request. This allows the client to match responses with requests.

3.2.4 Failover Processing

If the server is not able to process requests (it is not listening on the request queue), client's attempts to send request messages will be rejected by the AMQP server (because the messages are sent with mandatory flag). This ensures that requests are either processed quickly or rejected.

It is recommended that clients register a shutdown listener for the AMQP connection.

In case of AMQP server shutdown (due to failure or restart), all requests and responses being transmitted or waiting in queues are lost. If the client uses a shutdown listener, it will receive a shutdown notification from AMQP.

Both requests and response messages are delivered on a best-effort basis. Under exceptional circumstances, for example in case of failure of the AMQP server or the Capacity Management Module, request and response messages may be lost. Thus, it may happen that a client will never receive a response to its request (even if the request has been processed successfully by CMM).

3.2.5 Flow Control

If clients send requests too often, a backlog of unprocessed requests could build up on the server side, resulting in delays in request processing, negatively affecting all clients.

Several measures are taken to prevent this scenario.

3.2.5.1 Request Rate Limit

CMM constantly monitors the request rate for each client. If the number of requests sent by a client within a configured time period (for example 1 minute) exceeds the configured limit, CMM will start to reject requests from that client, until the request rate goes back below the limit.

When the request rate is exceeded, instead of processing an incoming request, CMM sends back a special *back off error response* containing details about the length of the measurement period and the request rate limit.

3.2.5.2 Broker Message Expiration

Clients may specify an expiration time for each request message. This allows the clients to protect themselves against the scenario, where a request waiting in the request queue for a long time becomes obsolete, but then gets processed even if the client does not wish to execute the request anymore. See section 3.2.2.1 for more details.

Additionally, for security reasons there is a RabbitMQ broker timeout defined (x-message-ttl), i.e. if a request is submitted and could not be processed by the broker within the configured time period, the request will not be processed anymore and no response message will be distributed.

3.2.5.3 AMQP Server Flow Control

In the AMQP protocol, *flow control* is an emergency procedure used to halt the flow of messages from a peer. It works in the same way between client and server and is implemented by the AMQP *Channel.Flow* command. Flow control is the only mechanism that can stop an over-producing publisher.

This feature is, however, not used by the RabbitMQ server. Instead, in case the server hits a preconfigured memory watermark, it uses TCP backpressure to temporarily block all connections that publish messages.

The RabbitMQ server persists the queue content to the disk, if required. This allows the server to keep more messages than fit into the machine memory; the queue size is theoretically only limited by the disk space.

In case of very high traffic, the AMQP server can be forced to temporarily throttle publishers to gain time to move some queue contents to disk, releasing memory for new messages. Once this is done, the server will start receiving messages again. (Under normal circumstances, this should not happen because the measures described above should always keep the queues relatively small.)

In addition, if the connection between the AMQP server and CMM is broken for longer than a specified time frame (for example 30 seconds), all queues would expire and the information in the queues would be deleted.

3.2.5.4 Recommendations

To avoid flow control issues it is recommended that clients consume messages in no-acknowledgment mode. In Java, this is done by calling *method Channel.basicConsume* with parameter *noAck* set to true.

Optionally, clients may consider using separate connections for producing and consuming of messages. In this setup, even if the AMQP server blocks the producing connection, the client application will still be able to consume responses and broadcasts.

4 Security

All communication between the client and the AMQP server is encrypted using the *Secure Socket Layer* (SSL) v3 protocol. Client and server certificates are used to establish a trusted connection. The usage of asymmetric encryption ensures confidentiality, authentication, message integrity assurance and non-repudiation of origin.

4.1 Server Certificate

The CMM System uses a Server Certificate signed by a well-known and trusted *Certificate Authority* (CA) like VeriSign or Comodo.

Usually the CA root certificates from these CAs are known and trusted by the software development frameworks like Java or .NET. If not, clients need to add the CA root certificate to a list of trusted certificates. The needed CA root certificate files can be downloaded directly from the CA's Website[1].

The exact way how this is done depends on the platform and programming language in which the client application is developed. Please note that the CA root certificate has to be imported into a location used by the client application's runtime environment, not into a web browser.

4.1.1 Using CA Root Certificate in Java

For Java clients, the CA root certificate must be imported into a *keystore*, which will be used to initialize the *Trust Manager* used by the client application. The certificate can be imported using the `keytool` utility, which is part of the Java runtime environment:

```
keytool -import -alias cmm-ca -file <cacert> -keystore <keystore>
```

In the command above, `<cacert>` is path to the CA certificate file in PEM format (`cacert.pem`) and `<keystore>` is path to the keystore file. The user must confirm that they trust the certificate being imported.

The keystore file will be created if it does not exist yet. Java keystores are protected by password; choose a password when first creating the keystore and then use it whenever accessing the keystore.

Refer to Java documentation for more information about the `keytool` utility. Example code can be found in the reference client implementation.

[1] VeriSign: http://www.verisign.com/support/roots.html
Comodo: https://support.comodo.com/index.php?_m=downloads&_a=view&parentcategoryid=1

4.2 Client Certificate

An organization that wants to use the Public Message Interface of the CMM System has to apply for an account on the AMQP server. The following is provided when a new account is set up:

- AMQP server host names, port number and virtual host name.
- Trader's Login ID if it doesn't already exist.
- Client certificate and private key that will be used by the client when connecting to the AMQP server.
- CA root certificate that has to be imported as a trusted certificate on the client side.

The client certificate:

- complies to X.509v3 specification,
- uses key length of 2048 bits,
- subject contains an unique identifier as the *Common Name*,
- is signed by Deutsche Börse CA,
- is provided in format PKCS#12 (`.p12`).

The client's private key is used to verify the client's identity when communicating with the AMQP server. Should a third party get hold of the client's private key, it could forge client's identity and access the encrypted communication with the server.

It is therefore extremely important to keep the private key secure.

4.2.1 Using Client Certificate in Java

To be able to use the client certificate and private key in a Java client, they need to be loaded into a keystore, which is used to initialize a *Key Manager*. Java supports the PKCS#12 format (`.p12`), where the private key is protected by a passphrase. The passphrase is provided to the client alone and it must be used when loading the `.p12` file into the keystore.

Please refer to the reference client implementation for sample Java code. For further details regarding SSL security, please consult the SSL specification, AMQP specification and www.rabbitmq.com/ssl.html.

4.3 Authentication

When a client connects to the AMQP server using the SSL protocol, both peers mutually authenticate each other by exchanging their certificates during an SSL handshake. All subsequent communication between the client and the server is encrypted using a key and encryption algorithm agreed on during the handshake. For connecting to the AMQP Server the username and password is required. The AMQP Server is connected to an internal LDAP Server which will verify the given credentials. In case of an unsuccessful authentication the client won't be able to connect to the AMQP Server.

Whenever a client sends a request to the AMQP server, it uses a client-specific exchange. The exchange name contains the client user name, which can be extracted by the CMM System when processing the request to identify from which client the request came.

4.4 Authorization

Authorization of client actions occurs on two levels:

4.4.1 Authorization by AMQP server

The AMQP server verifies client's privileges when a client accesses resources stored on the AMQP server. It is only allowed to consume from private response queue and private broadcast queue, and to publish to private request exchange.

The bindings among user's queues and internal exchanges ensure that only certain data can be received by the client.

In case of insufficient privileges, the attempt to access the resource is immediately (synchronously) rejected by AMQP.

4.4.2 Authorization by CMM

CMM verifies privileges when processing requests from clients.

In case of insufficient privileges, the request is rejected by CMM. From the client point of view, this rejection occurs asynchronously (a negative response message is sent to the client).

5 Message Payload Format

All messages sent between CMM and the clients have XML-encoded payload. The only exception is the heartbeat broadcast whose payload is simply the timestamp.

This section describes the payload format.

5.1 Overview

The payload format specification comes in the form of an XML schema, which specifies the allowed structure and format of XML elements and attributes. Clients are provided with the respective archive file upon successful account setup in the system. The format is briefly explained here.

5.1.1 XML Validity and Data Binding

It is important that clients produce valid XML code when sending requests to CMM. CMM uses a validating parser, which rejects any requests that do not strictly conform to the XML schema.

To make parsing and creating of XML data easier, and to avoid problems with validity of the XML code, clients are encouraged to use XML data binding.

For Java clients, classes generated using *Java Architecture for XML Binding* (JAXB) versions 2.1 are provided along with the XML schemas.

5.1.2 Schema Documentation

Reference HTML documentation is provided along with the XML schema, providing a convenient way of browsing the schema structure. The documentation lists elements, attributes and types used in the schema interconnected by hyperlinks along with the corresponding portions of the XML schema source code.

To display the schema HTML documentation, please extract file *xml-schema-docs.jar* and open file *index.html* in a web browser.

5.1.3 Schema Version

Every message sent or received by CMM must specify in its metadata the XML schema version that is used to encode its payload. This information is stored in message properties in field *content-type* as a string. Currently the only supported content-type is - “text/xml;v=1.0”.

This information can be used to validate that both peers use the same version of the payload format. If the content-type is not known, CMM tries decode the payload with the default schema “text/xml;v=1.0”.

This flexibility will be useful if CMM needs to support multiple schema versions in the future. CMM is currently ready to operate using 2 PMI versions – the latest version and the previous one. DBAG provides bugfixing, analysis and documentation update, if required.

5.1.4 Message Encoding

In order to reduce network traffic CMM allows any message to be zipped.

CMM has a configurable set of broadcasts and responses that are always zipped. The compressed messages can be recognised by the *“content-encoding”* property which is set to “*gzip*”.

For requests, clients may decide themselves whether to compress the messages or not. The GZIP format should be used for compressing the payload and any compressed messages must also have the *“content-encoding”* property set to “*gzip*”.

5.2 Requests

Each request sent to the CMM must conform to schema *cmm-explicit.xsd*. It should always contain one of the following elements:

- Login Request
- Allocation Request
- ATC Data Request
- Allocation Data Request

Only requests with a correct combination of user and balancing group are processed further by the system.

5.2.1 Login Request

The Login Request must always be the first request after the client gets (re)connected.

Login Request messages contain the *Login* element which optionally can contain an attribute named "*arguments*", defining what kinds of broadcasts the client wishes to receive.

Example: <Login arguments="X_Border:AT-CZ|AT-SK,X_Event:ALLOCATION,x-match:all" />

For more information on binding arguments and broadcast filters please see chapter 3.1.2.1 Filtering Broadcasts.

5.2.2 Allocation Request

Allocation Request messages contain the *AllocationReq* element, which contains:

- Attribute *allocationType* defining what shall happen when an allocation request cannot be fulfilled due to insufficient capacity. Valid values are:
  - ACE (all contracts equal)
  - AON (all or nothing)
  - IOC (immediate or cancel)
- Element *Allocations*, containing an interconnector (with In and Out Delivery areas) and a sequence of *Allocation (Alc)* elements. Each *Alc* element represents an allocation for a single period.

All elements *Alc* in the request are processed together, within a single transaction, respecting the requested allocation type. However, IOC allocations for different borders in the same request are treated independently. It is also valid for all capacity allocation requests submitted by explicit participants.

Each allocation should have a *client ID*, assigned by the client, which will appear in the response, so that the individual allocations from the request can be matched with the response items. In addition, each allocation is also assigned a unique allocation *ID* by CMM, which also appears in the response to the allocation request.

Example:

```
<AllocationReq type="AON">
  <Allocations inArea="10YDK-1--------W" outArea="10YDE-EON------1">
    <Alc clientId="123456" bmCode=""
         periodStart="2014-07-09T15:00:00.000Z"
         periodEnd="2014-07-09T16:00:00.000Z"
         quantity="1000000"/>
  </Allocations>
</AllocationReq>
```

5.2.3 ATC Data Request

ATC Data Request messages contain the *AtcDataRequest* element, specifying a date and the interconnector for which current ATC is being retrieved.

Example:

```
<AtcDataReq date="2014-07-09T00:00:00.000Z">
   <InterConnector area1="10YDE-EON------1" area2="10YDK-1--------W"/>
</AtcDataReq>
```

5.2.4 Allocation Data Request

Allocation Data Request messages contain the *AllocationDataRequest* element, specifying a time interval and the interconnector for which the allocation data is to be retrieved.

Example:

```
<AllocationDataReq balancingGroup="11XEEX-INTRA---D"
          from="2014-06-09T00:00:00.000+02:00" to="2014-07-10T13:42:09.000+02:00">
   <interConnector area1="10YFR-RTE------C" area2="10YDE-RWENET---I"/>
</AllocationDataReq>
```

5.3 Responses

Responses sent by CMM also conform to the schema *cmm-explicit.xsd*.

Response elements in general contain attribute *success*, which tells if the request has been processed successfully or not. If the processing failed, attribute *message* contains a human-readable description of why the processing failed.

Example of attribute *success*:

```
<AllocationResp>
  <Alc clientId="12345" message="The service is in HALT." success="false"/>
</AllocationResp>
```

5.3.1 Login Response

Login response contains messages element which is empty. However, important feature of it is that prior to sending it, CMM system creates the client's private queues as described in 3.2.1 Request Types.

5.3.2 Allocation Response

Allocation Response messages contain *an AllocationResp* element, containing a sequence of *Alc* sub-elements, one for each allocation that appeared in the request.

Each *AllocationResp* repeats the client ID from the request, so that response items can be matched with request items. If the allocation is successful the response also contains the allocation ID (alcId) assigned by CMM and the amount of confirmed quantity.

The confirmed capacity might depend on the *allocation type* if there is insufficient capacity:

- If the allocation type IOC is used the confirmed capacity would be less than or equal to the requested quantity
- If the allocation type AON is used a response with success 'false' would be returned
- If the allocation type ACE is used only capacity available throughout all contracts would be confirmed

Example:

```
<AllocationResp>
  <Alc alcId="302572" clientId="12345" confQty="1000"
       message="Allocated" success="true"/>
</AllocationResp>
```

5.3.3 ATC Data Response

ATC Data Response message contains an *AtcDataResp* element, which includes a sequence of *AtcDataList* elements. Each *AtcDataList* specifies a date, a border and an In and Out Delivery area. It contains a sequence of *AtcData* elements. Each *AtcData* element specifies the start and end of the period and the available quantity (in kilowatts).

Example:

```
<AtcDataResp success="true">
  <AtcDataList border="DE-DK" date="2014-07-10T22:00:00.000Z"
               inArea="10YDK-1--------W" outArea="10YDE-EON------1">
    <AtcData periodEnd="2014-07-10T23:00:00.000Z" periodStart="2014-07-10T22:00:00.000Z" quantity="2000000"/>
    <AtcData periodEnd="2014-07-11T00:00:00.000Z" periodStart="2014-07-10T23:00:00.000Z" quantity="2000000"/>
```

```
...
 <AtcData periodEnd="2014-07-11T22:00:00.000Z" periodStart="2014-07-11T21:00:00.000Z" quantity="2000000"/>
 </AtcDataList>
</AtcDataResp>
```

5.3.4 Allocation Data Response

Allocation Data Response message contains an *AllocationDataResp* element, which contains a sequence of *AllocationDataList* elements, one for each balancing group and for the interconnector and direction. Each *AllocationDataList* element specifies:

- balancing group,
- importing and exporting Delivery areas
- and contains a sequence of *AllocationData* elements. Each AllocationData element specifies the allocation ID, activity type, the quantity and the time of the allocation.

Example:

```
<AllocationDataResp message="OK" success="true">
        <AllocationDataList balancingGroup="DBS-TESTTRADER-3" inArea="10YFR-RTE------C" outArea="10YDE-RWENET---I">
        <AllocationData activityType="ALLOCATION" allocationId="302573" periodEnd="2014-07-11T16:00:00.000Z" periodStart="2014-07-11T15:00:00.000Z" quantity="1000" time="2014-07-10T12:02:58.789Z"/>
        <AllocationData activityType="ALLOCATION" allocationId="302574" periodEnd="2014-07-11T16:00:00.000Z" periodStart="2014-07-11T15:00:00.000Z" quantity="1000" time="2014-07-10T12:03:08.100Z"/>
        </AllocationDataList>
</AllocationDataResp>
```

5.3.5 Error Response

Error Response message is sent if a request is invalid (if parsing or validating the request XML code against the schema *cmm-explicit.xsd* fails). It contains an *ErrorResponse* element with a human-readable error description.

The client can find out which request caused the error using the response message correlation ID.

Example:

```
<ErrorResp message="Unknown format."/>
```

5.4 Broadcasts

Broadcasts sent by CMM also conform to schema *cmm-explicit.xsd*.

5.4.1 Heartbeat

Heartbeat message is the only non-xml message sent by the platform.

It only contains the timestamp of the moment it was sent with the format “YYYY-MM-DD'T'HH:mm:ss.SSSZ”.

5.4.2 ATC Data Notification

ATC data notification messages contain *the AtcDataNtf* element, which includes a sequence of *AtcDataList* elements which were already described in section 5.3.3 ATC Data Response.

All ATC values for the given border or interconnector are broadcasted as soon as the allocation for the day starts, as well as every time the border or interconnector is set to status Halt and then back to status Allocation.

Every time a new allocation happens the difference between the values in the last broadcast and the current values is broadcasted for both direction of the border. This can be either one or several hours depending on the type of the allocation and the ramping constraints, if there are any configured.

Example:

```
<AtcDataNtf>
   <AtcDataList      date="2014-07-10T22:00:00.000Z"      inArea="10YFR-RTE------C"      outArea="10YDE-RWENET---I"
border="DE-FR">
      <AtcData periodStart="2014-07-11T15:00:00.000Z" periodEnd="2014-07-11T16:00:00.000Z" quantity="9998000"/>
   </AtcDataList>
</AtcDataNtf>
```

Please notice that the broadcast information is sent on interconnector level, i.e. the In- and OutArea fields, as shown above, are the EIC codes of the Delivery Area. For borders with common ATC, i.e. where the available capacity is common for all interconnectors of the border, this broadcast is sent for the leading interconnector, i.e. the interconnector of the Shipping agent. The participants, however, can still allocate for all interconnectors of the common ATC border. For more information on Delivery Areas, common ATC and Shipping agent check chapter 2 Functional Overview in *HLS100A Functional Description Capacity Management Module*.

5.4.3 Allocation Data Notification

Allocation data notification messages contain the *AllocationDataNtf* element, which includes the data as already described in section 5.3.4 Allocation Data Response.

This broadcast is triggered every time there is a new allocation for the respective balancing group. If border/interconnector status changes the system doesn't broadcast allocation data.

Example:

```
<AllocationDataNtf>
<AllocationDataList balancingGroup="DBS-TESTTRADER-3" inArea="10YFR-RTE------C" outArea="10YDE-RWENET---I">
   <AllocationData allocationId="302574" time="2014-07-10T12:03:08.100Z" activityType="ALLOCATION"
periodStart="2014-07-11T15:00:00.000Z"
            periodEnd="2014-07-11T16:00:00.000Z" quantity="1000"/>
</AllocationDataList>
</AllocationDataNtf>
```