MTS SDK .NET Integration Guide

December 2023

sportradar

SPORTS TECHNOLOGY. REIMAGINED.

1 Document history

Version	Author	Date	Comments
1.11	Aleš Mrak,	2023-12-01	Added comments on best practices how to use SDK,
	Dejan Pavšek		Added chapter Client – MTS information interchange, Updated document template
1.10	David Hrovat	2020-10-20	Added configuration property sslServerName
1.9	David Hrovat	2019-08-09	Added configuration property ticketResponseTimeoutPrematch
1.8	Srđan Tot	2019-05-28	Added description about custom bet
1.7	Srđan Tot	2019-04-29	Added new timeout configuration properties
1.6	Srđan Tot	2019-02-27	Added timeout configuration properties
1.5	Srđan Tot	2019-02-13	Added MTS Client API configuration properties
1.4	David Hrovat	2018-10-11	Added exclusiveConsumer configuration property
1.3	David Hrovat	2017-11-10	Added port number configuration property
1.2	David Hrovat	2017-09-12	Added provideAdditionalMarketSpecifiers property to config section
1.1	Uros Bregar	2017-09-09	Added description about Rest and Cache log Added accessToken configuration property Updated method SetIdUof
1.0	Uros Bregar	2015-03-13	Initial version

2 Table of contents

3 Getting started

Before starting to use the SDK, please read the MTS documentation found at docs.betradar.com, chapter Managed Trading Services (MTS) / MTS – Ticket Integration Manual.

SDK examples and code documentation is available at the https://sdk.sportradar.com.

4 Client – MTS information interchange

4.1 Virtual host, username and password

Please contact your Betradar MTS-integration manager for your username and password.

As a convention, this username is used in the virtual host's name as well as for the names of your designated Exchanges and Queues.

The virtual hostname is then simply /username (a slash followed by the username).

4.2 List of interchanged messages

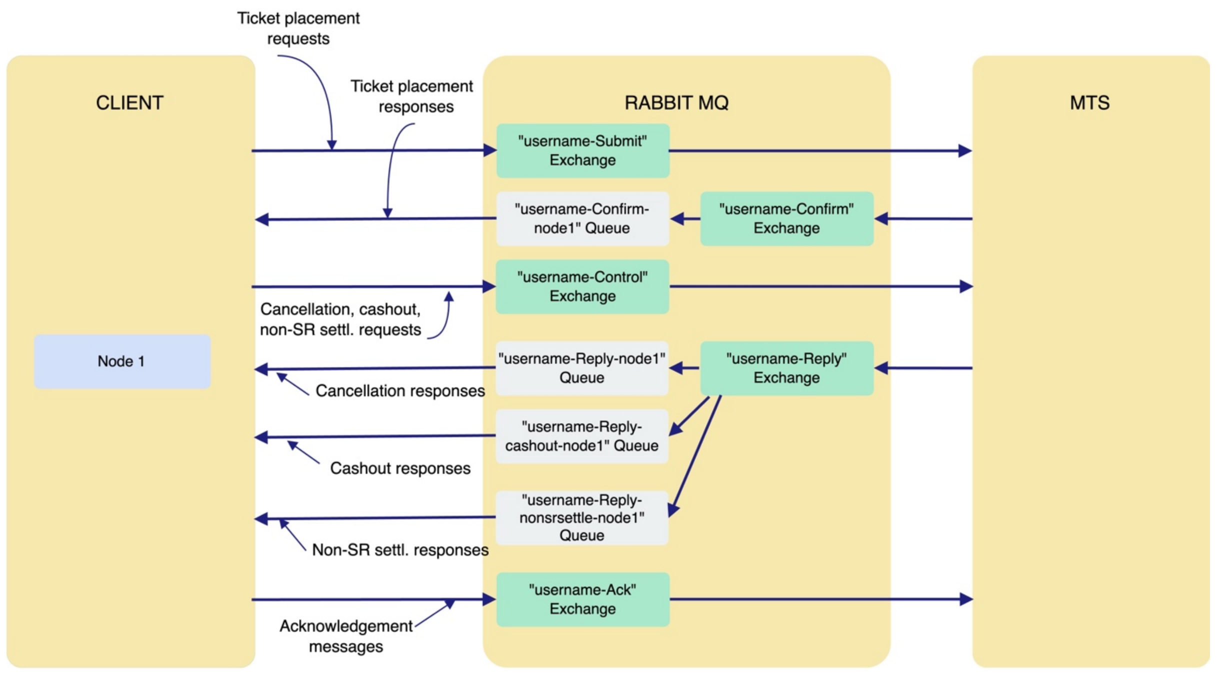
The basic functionality is the submission of tickets (Ticket placement request) and receiving the outcome (Ticket placement response) of the validation process performed by MTS. The outcome is suggestion whether a ticket should be accepted or rejected. A ticket represents one or more bets placed by end customers (punters).

Client (bookmaker or platform) submits the punter's ticket information, which represents the main message/payload for the validation process. Additionally, the client has an option to respond, informing MTS about their decision regarding the MTS' recommendation. It means they have the option to submit Info Ticket acceptance.

Other messages include:

- Submitting a Ticket cancellation, receiving a response and in some cases submitting an acknowledgement message
- Submitting a Ticket cashout request and receiving a response
- Submitting a Non-SR content settlement request and receiving a response (non-Sportradar content means betting events that are managed by the client rather than Sportradar, so also their settlement information must be submitted by the client)

The following diagram depicts types of messages and the RabbitMQ Exchanges and Queues that the messages should be submitted to or consumed from, respectively.


Ticket placement requests
Ticket placement responses
CLIENT
RABBIT MQ
MTS
"username-Submit" Exchange
"username-Confirm-node1" Queue
"username-Confirm" Exchange
"username-Control" Exchange
Cancellation, cashout, non-SR settl. requests
Node 1
"username-Reply-node1" Queue
"username-Reply" Exchange
Cancellation responses
"username-Reply-cashout-node1" Queue
Cashout responses
"username-Reply-nonsrsettle-node1" Queue
Non-SR settl. responses
"username-Ack" Exchange
Acknowledgement messages

4.3 RabbitMQ exchanges

The following exchanges are used:

Exchange name	Purpose	Type	Access/Options	Description
username-Submit	submit ticket information	fanout	write	ticket placement requests
username-Control	submit control messages	topic	write	sending: cancellation, cashout, non-SR settlement requests
username-Confirm	receive ticket acceptance recommendations (accepted/ rejected)	topic	read/ configure	ticket placement responses
username-Reply	receive control message responses (accepted /rejected)	topic	read/ configure	cancellation, cashout and non-SR settlement responses
username-Ack	submit : info ticket acceptance, info ticket cancellation acceptance	topic	write	customer feedback on MTS' ticket acceptance/rejection recommendation (in case of explicit acks needed for national state lotteries) customers' final decisions on ticket cancellations

Exchanges are durable and are declared and maintained by the MTS side. Clients need to know which one to publish a message to and which one to bind the Queue to before consuming messages.

4.4 RabbitMQ queues

Queue name	Type	Purpose	Description
username-Confirm	any queue	receiving ticket acceptance recommendations	ticket placement response
username-Reply	any queue	receiving responses to cancellation requests	cancellation responses (accepted /rejected)
username-Reply-cashout	any queue	receiving responses to cashout requests	cashout responses (accepted /rejected)
username-Reply-nonsrsettle	any queue	receiving responses to non-SR settlement requests	non-SR settlement responses (accepted /rejected)

The client is obliged to:

- declare,
- bind,
- consume from,
- maintain the queues of all custom needs

Details are listed in the Creating and Binding the RabbitMQ queues chapter below.

4.4.1 Queue naming convention

The client has the right to instantiate any AMQP-entity within their vhost that follows the following naming-pattern:

“username-(Submit|Confirm|Ack|Control|Reply|Reply-cashout|Reply-nonsrsettle)” your username followed by a dash followed by one of the “|” (or) – separated strings ‘ within the brackets followed by any valid string. According to the RabbitMQ documentation a valid string here can be empty or a sequence of these characters: letters, digits, hyphen, underscore, period or colon.

4.4.2 Queue naming recommendation

We recommend to name all Queues respective of the Exchange you want to bind them to:

- username-Confirm-nodeX, if you intend to bind this Queue to your Confirm-Exchange
- username-Reply-nodeX, if you intend to bind this Queue to your Reply-Exchange

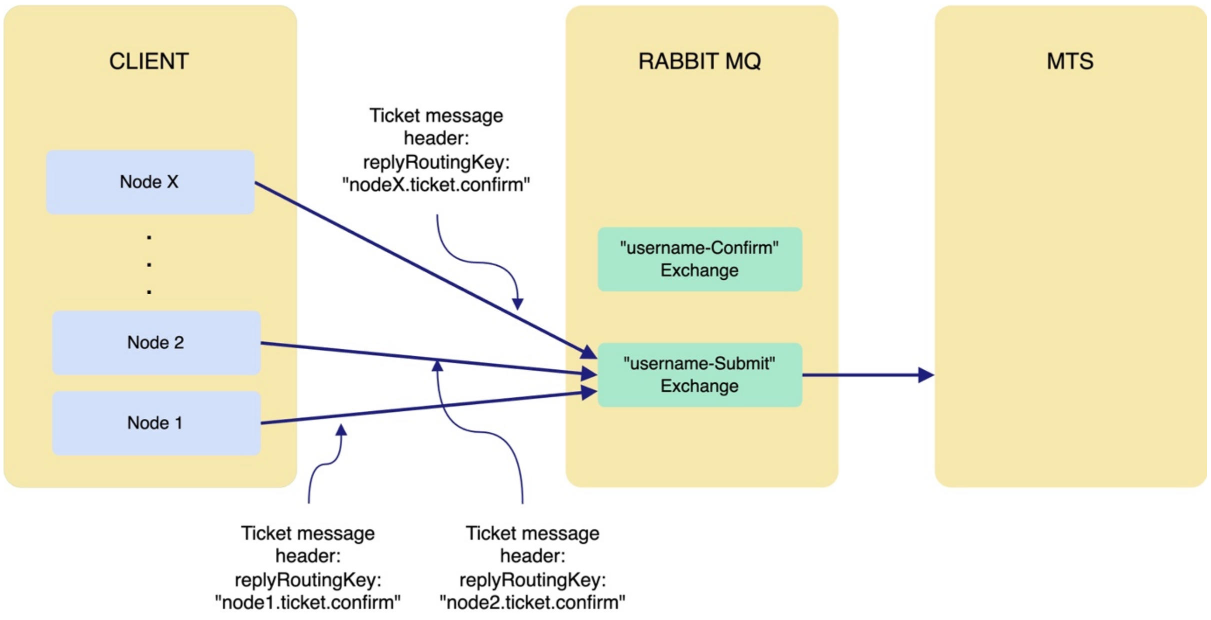
Here “username-Reply-” and ”username-Control-” are strings where “–nodeX” represents the id you dedicate internally on your side to the client-node within your cluster that you wish to receive the MTS response on (typically the one you send the related message to MTS in the first place with), of which the X stands for an integer (see chapter Queue naming convention).

4.5 Message control

4.5.1 Determining response queues, replyRoutingKey

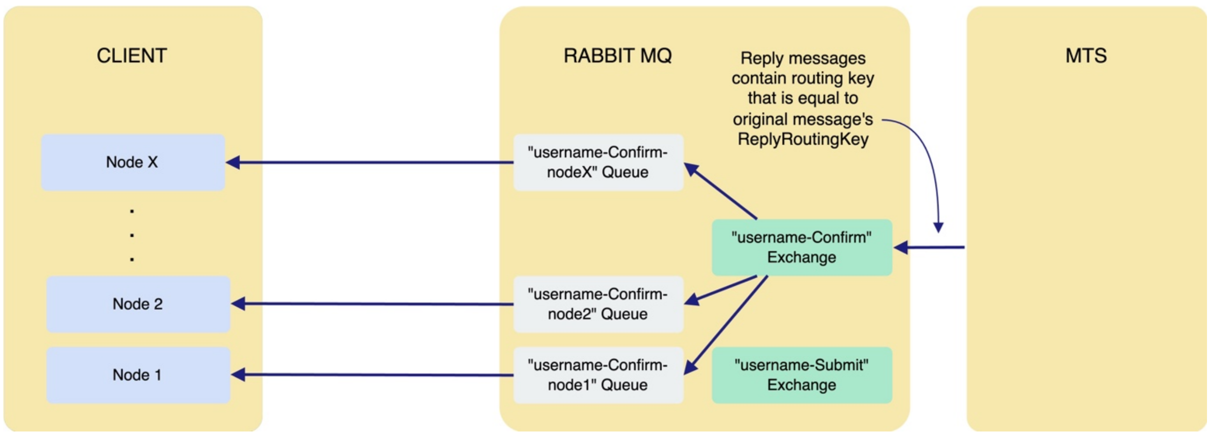
The following diagram shows a typical client setup. There are multiple server nodes (for performance or availability reasons) that independently submit tickets to the Submit exchange. For creating and binding queues, see the corresponding chapter below.

1A) Ticket placement

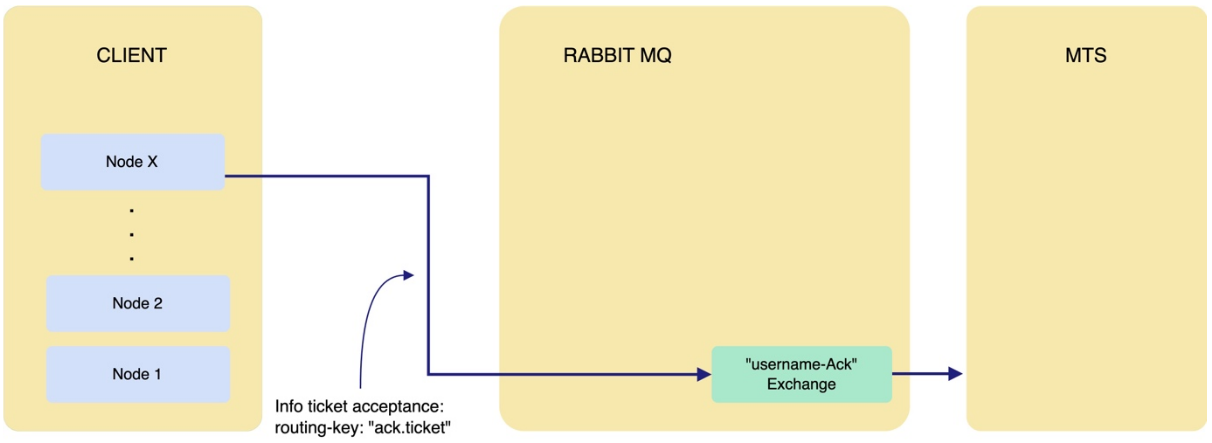

CLIENT
RABBIT MQ
MTS
Node X
Node 2
Node 1
"username-Confirm" Exchange
"username-Submit" Exchange
Ticket message header: replyRoutingKey: "nodeX.ticket.confirm"
Ticket message header: replyRoutingKey: "node1.ticket.confirm"
Ticket message header: replyRoutingKey: "node2.ticket.confirm"

However, the ticket response must be sent exactly to the node from which the ticket originated. This is achieved by multiple Confirm queues (each of them designated for a particular node) and the replyRoutingKey. Every ticket that is submitted to the Submit exchange must contain the replyRoutingKey in the message header. This key will be used by MTS as the Routing Key when submitting the response to the Confirm exchange.

1B) Ticket placement response

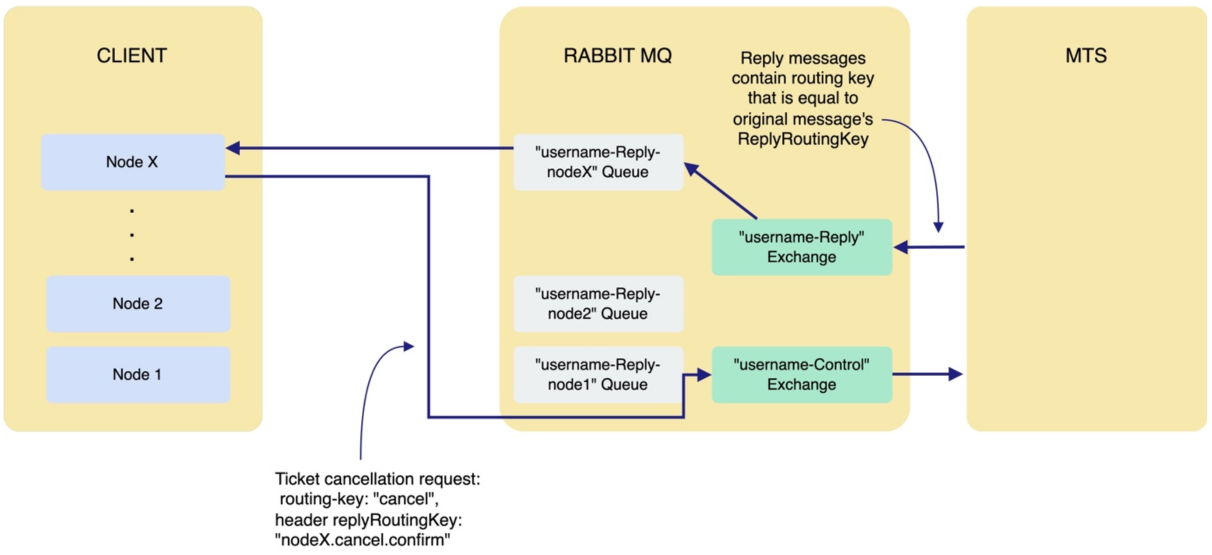

CLIENT
RABBIT MQ
MTS
Reply messages contain routing key that is equal to original message's ReplyRoutingKey
Node X
Node 2
Node 1
"username-Confirm-nodeX" Queue
"username-Confirm-node2" Queue
"username-Confirm-node1" Queue
"username-Confirm" Exchange
"username-Submit" Exchange

1C) Acknowledgement (in the case if explicit acks are needed for national state lotteries) - (distinguish from “consumer acks”)



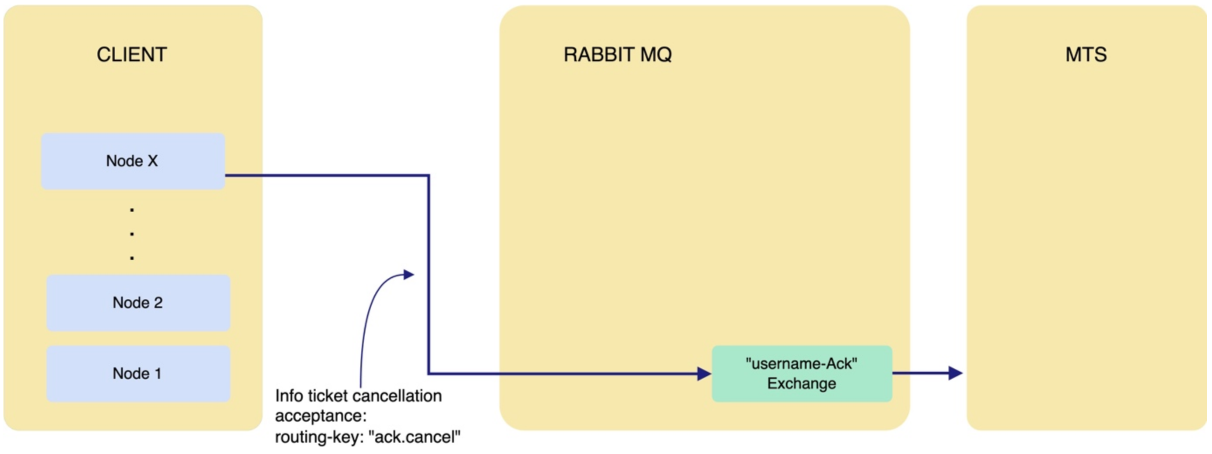
The replyRoutingKey is used in the same way also in the cases of Ticket cancellation, Ticket cashout and Ticket settlements with non-SR content.

2A) Ticket cancellation and response

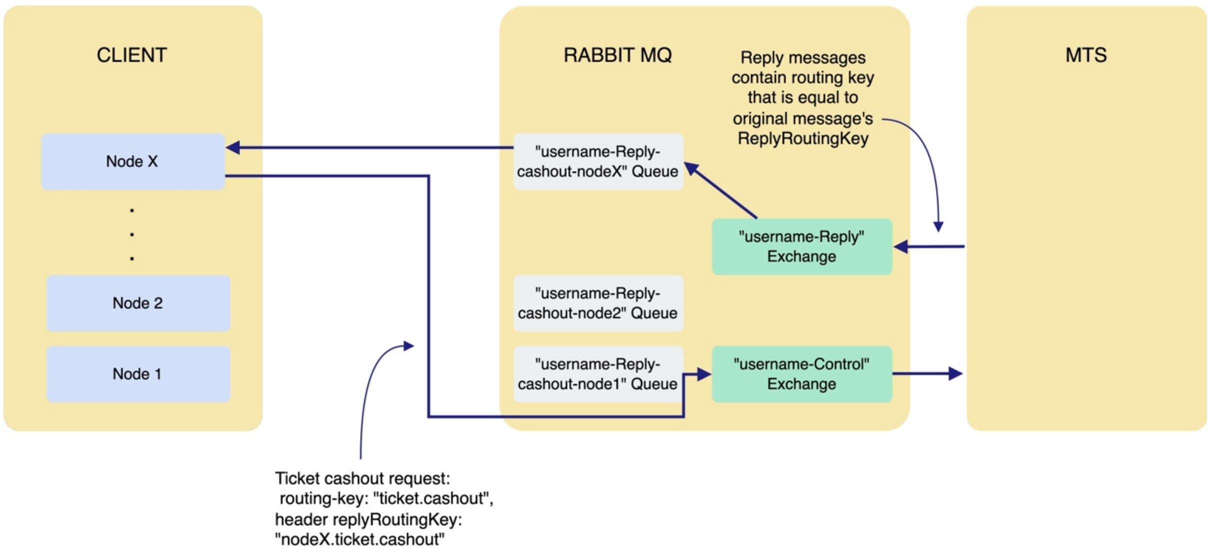


CLIENT
RABBIT MQ
MTS
Reply messages contain routing key that is equal to original message's ReplyRoutingKey
Node X
Node 2
Node 1
"username-Reply-nodeX" Queue
"username-Reply" Exchange
"username-Reply-node2" Queue
"username-Reply-node1" Queue
"username-Control" Exchange
Ticket cancellation request: routing-key: "cancel", header replyRoutingKey: "nodeX.cancel.confirm"

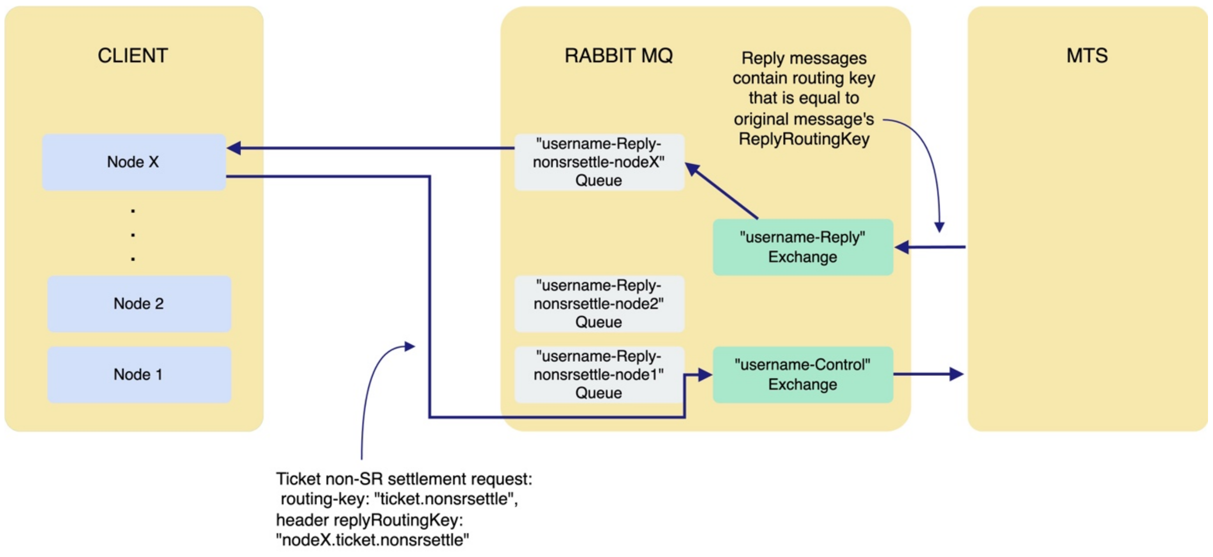
2B) Cancellation acknowledgement


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
"username-Ack" Exchange
MTS
Info ticket cancellation acceptance:
routing-key: "ack.cancel"

3) Ticket cashout and response


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
Reply messages contain routing key that is equal to original message's ReplyRoutingKey
"username-Reply-cashout-nodeX" Queue
"username-Reply" Exchange
"username-Reply-cashout-node2" Queue
"username-Reply-cashout-node1" Queue
"username-Control" Exchange
MTS
Ticket cashout request:
routing-key: "ticket.cashout",
header replyRoutingKey:
"nodeX.ticket.cashout"

4) Ticket settlements with non-SR content


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
Reply messages contain routing key that is equal to original message's ReplyRoutingKey
"username-Reply-nonsrsettle-nodeX" Queue
"username-Reply" Exchange
"username-Reply-nonsrsettle-node2" Queue
"username-Reply-nonsrsettle-node1" Queue
"username-Control" Exchange
MTS
Ticket non-SR settlement request: routing-key: "ticket.nonsrsettle", header replyRoutingKey: "nodeX.ticket.nonsrsettle"

4.5.2 replyRoutingKey convention

The value of this header field can contain any valid non-empty string as a prerequisite for a successful response relay. However, it must match at least one binding-key of at least one of the queues you bound to the respective exchange.

Note: If you don’t set a valid replyRoutingKey for tickets, it will be set by MTS automatically to the string “not_set”.

If you don’t set a valid replyRoutingKey for cancellation requests it will be set by MTS automatically to the string “cancel”.

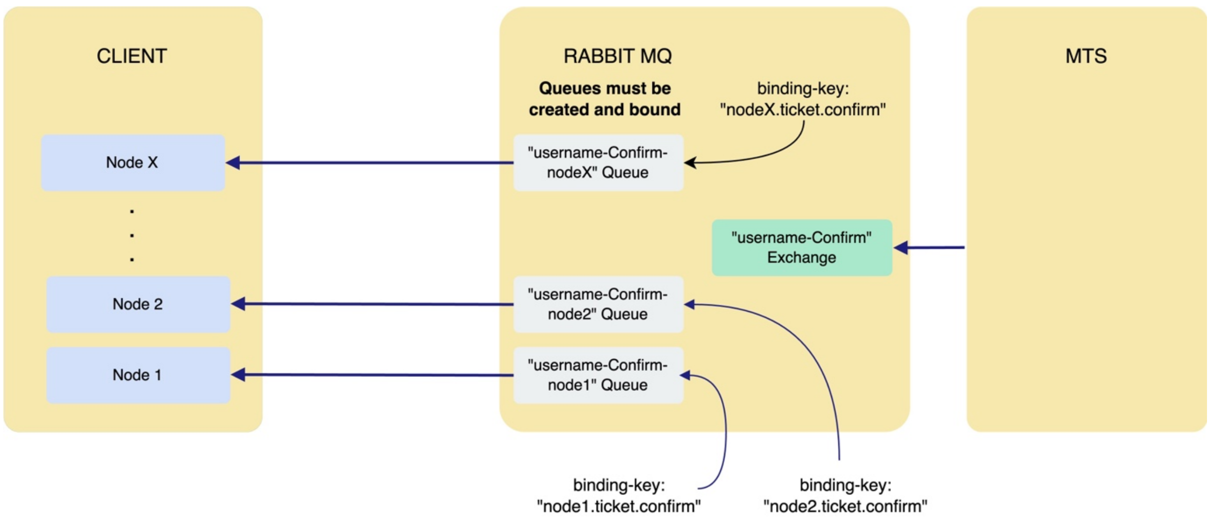
4.5.3 Recommendation

We recommend the following values if it comes to the replyRoutingKeys and the matching binding- keys for your Queues (see also Chapter Queue Naming Recommendations):

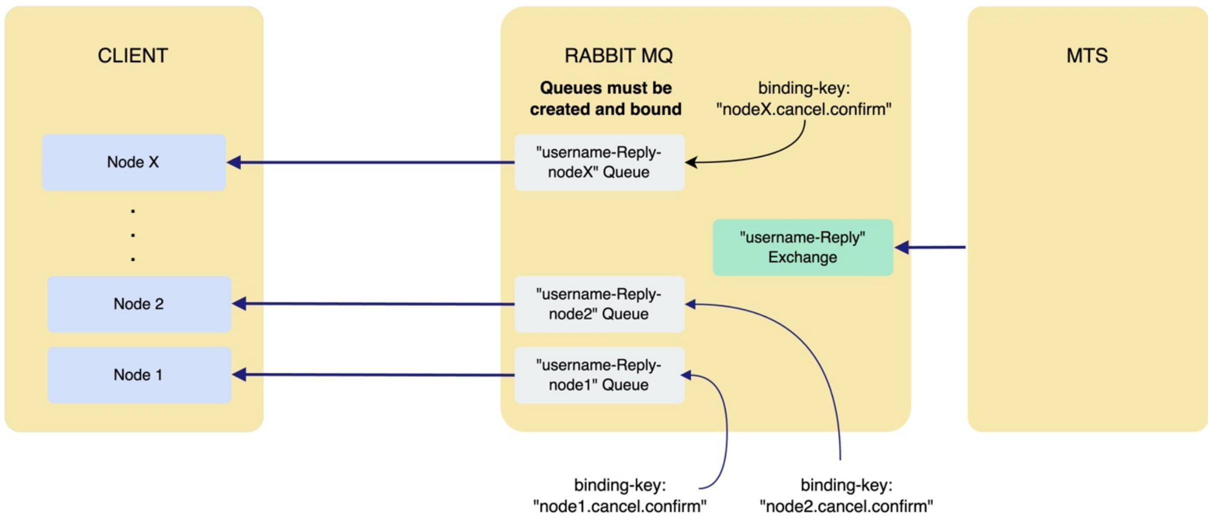
message-type	**replyRoutingKey & Queue binding-key**
ticket (info)	nodeX.ticket.confirm
cancellation request	nodeX.cancel.confirm
cashout request	nodeX.ticket.cashout
non-SR settlement request	nodeX.ticket.nonsrsettle

4.5.4 Creating and binding the RabbitMQ queues

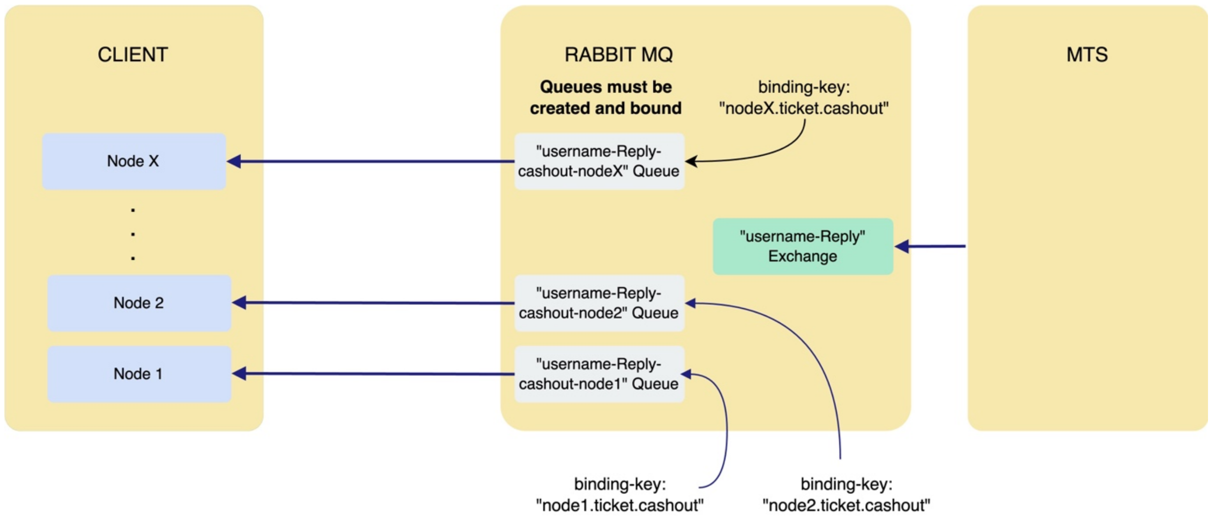
Prior to using the RabbitMQ, clients must create and bind the queues. Naming convention defines that the queue names follow the pattern: "username-Confirm-nodeX" (where X is the number of the corresponding queue). When binding the "username-Confirm-nodeX" queue, the binding key should be: binding-key: "nodeX.ticket.confirm" (where X is the number of the corresponding queue).


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
Queues must be created and bound
binding-key: "nodeX.ticket.confirm"
"username-Confirm-nodeX" Queue
"username-Confirm" Exchange
"username-Confirm-node2" Queue
"username-Confirm-node1" Queue
binding-key: "node1.ticket.confirm"
binding-key: "node2.ticket.confirm"
MTS

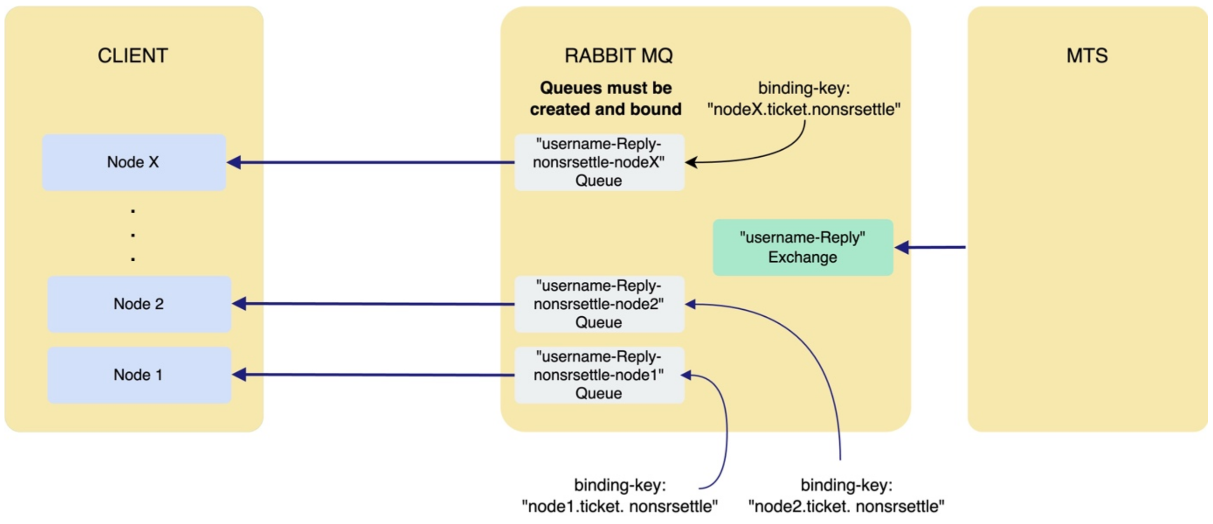
Queues related to cancellation responses are created and bound in this way:


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
Queues must be created and bound
binding-key: "nodeX.cancel.confirm"
"username-Reply-nodeX" Queue
"username-Reply" Exchange
"username-Reply-node2" Queue
"username-Reply-node1" Queue
binding-key: "node1.cancel.confirm"
binding-key: "node2.cancel.confirm"
MTS

And in the same way also queues used for cashout responses:


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
Queues must be created and bound
binding-key: "nodeX.ticket.cashout"
"username-Reply-cashout-nodeX" Queue
"username-Reply" Exchange
"username-Reply-cashout-node2" Queue
"username-Reply-cashout-node1" Queue
binding-key: "node1.ticket.cashout"
binding-key: "node2.ticket.cashout"
MTS

Non-SR settlement responses:


CLIENT
Node X
Node 2
Node 1
RABBIT MQ
Queues must be created and bound
binding-key: "nodeX.ticket.nonsrsettle"
"username-Reply-nonsrsettle-nodeX" Queue
"username-Reply" Exchange
"username-Reply-nonsrsettle-node2" Queue
"username-Reply-nonsrsettle-node1" Queue
binding-key: "node1.ticket. nonsrsettle"
binding-key: "node2.ticket. nonsrsettle"
MTS

4.5.5 correlationId

For identifying which MTS response corresponds to which message sent initially by you towards MTS we offer a more convenient option, apart from a tedious parsing of the message payload in order to then compare the ticket-id.

A more convenient and efficient way is having you set the message header field correlationId. Its value will be found then on the corresponding MTS responses in their header field of the same name.

4.6 Connecting a queue consumer without MTS SDK

In the exceptional case that you need to connect a queue consumer without the MTS SDK, follow these instructions:

Important

When using queue declare, please make sure that you use **exactly** the following queue arguments (**no more, no less**)

x-queue-master-locator: min-masters

Otherwise, the MTS SDK will not work correctly.

5 Logging

To make it easier to find the log entry associated with a particular action that occurred within the SDK, the logs are split into several files.

- Feed log: contains log entries for all messages send to or received from the MTS
- Rest log: contains log entries for all messages send or received from API calls[1]
- Cache log: contains log entries for all messages related to internal cache(s)
- Execution log: contain log entries for all important actions and all error / warning conditions which occur within the SDK
- Client interaction log: logs the interaction between the user code and the SDK
- Statistics log: contains periodically written statistic information

To enable the SDK logging, the logging framework used by the SDK must be properly configured. The configuration is done by calling the following method:

```
SdkLoggerFactory.Configure(new FileInfo("config_file_path"));
```

The default configuration file can be obtained from the SDK example project available on the SDK site.
Logs are used by the support team, so it is recommended that you send them along with any emails regarding the issue.

[1] REST API calls are made only when building ticket with UnifiedFeed selections

sportradar
SPORTS TECHNOLOGY. REIMAGINED.

6 Configuration

The configuration needed by the SDK must be provided via the app.config file, which must contain the following section:

```
<mtsSdkSection
      username="username",
      password="password",
      host=" mtsgate-ci.betradar.com ",
      vhost="/vhost",
      useSsl="false",
      node="3",
      bookmakerId="1",
      limitId="1",
      currency="EUR",
      channel="Internet",
      accessToken="your_uf_access_token",
      provideAdditionalMarketSpecifiers="true",
      port="5671",
      exclusiveConsumer="true",
      keycloakHost="https://mts-auth.sportradar.ag",
      keycloakUsername="username",
      keycloakPassword="password",
      keycloakSecret="secret",
      mtsClientApiHost="http://10.200.24.234:9211/edge/proxy",
      ticketResponseTimeout="15000",
      ticketResponseTimeoutPrematch="5000",
      ticketCancellationResponseTimeout="600000",
      ticketCashoutResponseTimeout="600000",
      ticketNonSrSettleResponseTimeout="600000" />
```

Required attributes:

- `username`: Username used to connect to the AMQP broker. Betradar provides this value.
- `password`: Password used to connect to the AMQP broker. Betradar provides this value.
- `host`: The hostname of the AMQP broker. Please use the following hostnames unless the integration team provides different ones.
  - Integration environment: mtsgate-ci.betradar.com
  - Production environment: mtsgate-t1.betradar.com

Optional attributes:

- `vhost`: The name of the virtual host configured on the AMQP broker. If the value is not specified, the value of '/username' attribute is used as virtual host.

- `useSsl`: The value specifying whether SSL will be used when connecting to the broker. Default value is `true`.
- `node`: This value is used to filter MTS responses which were produced as responses to requests send by different SDK instances. In most configurations each SDK should use different node value. Default value is `1`.
- `bookmakerId`: When provided, it is used as the default value for the `BookmakerId` on the ticket. The value can be overridden when building the ticket. Betradar provides this value.
- `limitId`: When provided, it is used as the default value for the `LimitId` property on the ticket. The value can be overridden when building the ticket. Betradar provides the set of available values.
- `currency`: When provided, it is used as the default value for the Currency property on the ticket. The value must comply with the ISO 4217 standard.
- `channel`: When provided, it is used as the default value for the `SenderChannel` property on the ticket. Value must be one of the `SenderChannel` enumeration members.
- `accessToken`: When selections are build using UnifiedOdds ids, the accessToken is used to access sports API. Also ensure that server running the sdk is whitelisted on api.betradar.com. Betradar provides this value.
- `provideAdditionalMarketSpecifiers`: This value is used to indicate if the sdk should add market specifiers for specific markets. Only used when building selection using UnifiedOdds ids. If this is set to true and the user uses UOF markets, when there are special cases (market 215, or $score in SOV/SBV template), sdk automatically tries to add appropriate specifier; if set to false, user will need to add this manually.
- `port`: Port should be chosen through the useSsl property. Manually setting port number should be used only when non-default port is required.
- exclusiveConsumer: The value specifying whether the rabbit consumer channel should be exclusive. Default value is `true`.
- keycloakHost: The auth server for accessing MTS Client API.
- keycloakUsername: The default username used to get access token from the auth server. It can be overridden when the MTS Client API methods are called.
- keycloakPassword: The default password used to get access token from the auth server. It can be overridden when the MTS Client API methods are called.
- keycloakSecret: The secret used to get access token from the auth server.
- mtsClientApiHost: The MTS Client API host.
- ticketResponseTimeout: The ticket response timeout in ms. Default value is 15000ms and it can't be less than 10000ms or greater than 30000ms. (Also default for tickets with selections using “live” ids)
- ticketResponseTimeoutPrematch: The ticket response timeout in ms. Default value is 5000ms and it can't be less than 3000ms or greater than 30000ms. Used for tickets containing selections with “lcoo” id.
- ticketCancellationResponseTimeout: The ticket cancellation response timeout in ms. Default value is 600000ms and it can't be less than 10000ms or greater than 3600000ms.
- ticketCashoutResponseTimeout: The ticket cashout response timeout in ms. Default value is 600000ms and it can't be less than 10000ms or greater than 3600000ms.
- ticketNonSrSettleResponseTimeout: The ticket non-Sportradar response timeout in ms. Default value is 600000ms and it can't be less than 10000ms or greater than 3600000ms.
- sslServerName: The server name that will be used to check against SSL certificate.

For more information on ticket properties, please refer to the MTS_Ticket_Integration document.

7 Obtaining the SDK

The SDK is provided as a code library (Sportradar.MTS.SDK.dll) available from the SDK site and via the NuGet package manager. The use of the NuGet package manager is recommended as it supports update notifications and makes it easier to obtain new releases of the SDK.

8 SDK setup and teardown

The SDK is setup by the following steps:

- Creating an instance of the `MtsSdk` class.
- Attaching to the following events exposed by the `MtsSdk` type.
    - SendTicketFailed – raised if the ticket could not be send to the AMQP broker within the set timeout (usually 15 seconds). This usually indicates an Internet connection or firewall issues.
    - TicketResponseReceived – occurs when a response to ticket placement or ticket cancellation request from the MTS is received.
    - UnparsableTicketResponseReceived – occurs when the response from the MTS cannot be deserialized. This usually indicates that a deprecated version of the SDK is being used.
    - TicketResponseTimedOut - event to notify user if the ticket response did not arrive in timely fashion (when sending in non-blocking mode). Timeouts are set using ticketResponseTimeout, ticketCancellationResponseTimeout and ticketCashoutResponseTimeout.
- Opening the created `MtsSdk` instance.

These steps can be performed by the following code:

```
var config = MtsSdk.GetConfiguration();
var mtsSdk = new MtsSdk(config);
mtsSdk.SendTicketFailed += OnSendTicketFailed;
mtsSdk.TicketResponseReceived += OnTicketResponseReceived;
mtsSdk.UnparsableTicketResponseReceived += OnUnparsableTicketResponseReceived;
mtsSdk. TicketResponseTimedOut += OnTicketResponseTimedOut;
mtsSdk.Open();
```

For more information on how to handle the events, refer to the SDK examples and/or the SDK code documentation.

Once the initialized `MtsSdk` instance is no longer needed, it must be teardown in order to release the resources it is holding. It is also recommended to detach from events before disposing of the instance. This can be accomplished using the following code:

```
mtsSdk.SendTicketFailed -= OnSendTicketFailed;
mtsSdk.TicketResponseReceived -= OnTicketResponseReceived;
mtsSdk.UnparsableTicketResponseReceived -= OnUnparsableTicketResponseReceived;
mtsSdk. TicketResponseTimedOut -= OnTicketResponseTimedOut;
mtsSdk.Close();
```

9 Building ticket instances

The SDK uses a “builder pattern” to simplify the process of creating new ticket instances. Below is a list of the most noticeable builders.

- `TicketBuilder`: A root builder used as a starting point when building tickets.
- `SenderBuilder`: Used to specify the information about a ticket sender (bookmaker).
- `EndCustomerBuilder`: Used to build `EndCustomer` instances, representing the punter associated with the ticket. This information is part of the send element.
- `BetBuilder`: Used to build bet instances, which is part of the ticket. Each ticket must contain at least one bet.
- `SelectionBuilder`: Used to build selection instances, which are parts of bet. Each bet must contain at least one selection.

Below is a code snippet that builds a ticket containing the mandatory information. Please note that some information from the configuration is automatically applied to the ticket, so changing the configuration may cause the snippet below to produce an incomplete ticket. For more information refer to the configuration section of this document and the MTS_Ticket_Integration document.
Builders can be obtained on mtsSdk instance through `BuilderFactory`.

```
var _builderFactory = _mtsSdk.BuilderFactory;
var ticket = _builderFactory.CreateTicketBuilder()
    .SetTicketId("ticketId")
    .SetSender(_builderFactory.CreateSenderBuilder()
        .SetCurrency("EUR")
        .SetEndCustomer(_builderFactory.CreateEndCustomerBuilder()
            .SetId("customerClientId")
            .SetConfidence(1)
            .SetIp(IPAddress.Loopback)
            .SetLanguageId("en")
            .Build())
        .Build())
    .AddBet(_builderFactory.CreateBetBuilder()
        .SetBetId("betId")
        .SetBetBonus(1)
        .SetStake(1, StakeType.Total)
        .AddSelectedSystem(1)
        .AddSelection(_builderFactory.CreateSelectionBuilder()
            .SetEventId(1)
            .SetId("selectionId")
            .SetOdds(11000)
            .Build())
        .Build())
    .BuildTicket();
```

10 Sending tickets to MTS

The SDK supports two ways of sending tickets to the MTS. The recommended way is to use non-blocking mode. Non-blocking means that the execution of the current thread is not blocked after the ticket is sent and the response from MTS (TicketResponseReceived event) is processed in another thread. To send a ticket in a non-blocking mode, the following line can be used:

```
mtsSdk.SendTicket(ticket);
```

For this mode, the TicketResponseTimedOut event is also available to notify the user if the ticket response has not arrived in timely fashion.

Important consideration

When using the TicketResponseReceived event, it is advisable to queue the response to the client's own threads. This is because if the client code is executed inside the event handler and if it takes a long time, the entire SDK instance may be disconnected from the RabbitMQ server due to response timeout. Response handling should be short when used directly on the TicketResponseReceived event, as it is called on the consumer thread. It is also advisable to use try and catch patterns in the handler to log and handle any problems while processing the response.

When the ticket is sent in blocking mode, the current thread is blocked until a response is received from MTS or a timeout occurs (usually 15 seconds). When using blocking mode, the TicketResponseReceived event for that ticket is never raised. The following line can be used to send a ticket in blocking mode:

```
var ticketResponse = mtsSdk.SendTicketBlocking(ticket);
```

11 Custom bet

The CustomBetManager provides an easy way to fetch available selections for a selected event and calculate the probability for a list of provided selections. To get a reference to the CustomBetManager, use the following property:

```
mtsSdk.CustomBetManager;
```

To get available selections for the provided event, use the following method:

```
manager.GetAvailableSelectionsAsync(eventId);
```

To calculate probability for a list of selections, use the following method:

```
manager.CalculateProbability(selections);
```

CustomBetManager uses builder pattern to simplify creation of selections. To create a selection, use the following methods:

```
manager.CustomBetSelectionBuilder
        .SetEventId(eventId)
        .SetMarketId(marketId)
        .SetOutcomeId(outcomeId)
        .SetSpecifiers(specifiers)
        .Build();
```

12 Tips and tricks

12.1 Using multiple SDK instances

If the client wants to use multiple SDK instances for more ticket throughput, the following needs to be done on the client side. Each use of the SDK instance should be in a separate host (process) because the library uses singletons to create interfaces. If exclusive connection is enabled (which is enabled by default settings, see the ***exclusiveConsumer*** property in *mtsSdkSection* section) the new instance on a different host must **not** use the same node id (see the ***node*** property in *mtsSdkSection section*) for connecting to MTS Border RabbitMQ, if so then the new connections will fail to connect to RabbitMQ server. The *exclusiveConsumer* property is there to protect the client from consuming responses from other SDK instances and that SDK would wait for a response that would never arrive.

12.2 Checking connection status

The SDK uses the ConnectionStatus interface (which can be obtained from the MtsSdk instance) to retrieve information about the status of the connection (if it is connected, time of connection, time of disconnection, etc.).

There are also ConnectionChange events that are triggered when the connection status changes.

12.3 Connection and reconnection handling using the SDK

If the connection goes down, the client should keep the initial SDK instance open and use its connection handling to reconnect, rather than opening new instances. Opening new instances can cause issues with hitting connection limits on the server side. The SDK retry window is not configurable as it also helps to protect our servers from heavy loads in case of problems. In any case, running a single instance of the SDK in such cases is the most reliable way, and even the intended way from the SDK perspective. Also, SDK reconnections to the server are retried with a delay window that usually starts at 100 milliseconds and increases with each reconnect error (to protect against heavy loads), but it is not greater than 5 seconds.

12.4 Additional notes for connection whitelisting

MTS would like to take this opportunity to reiterate that clients should be cautious when whitelisting MTS IPs as this can lead to an unplanned total loss of network connectivity and should therefore be avoided where possible. Any connectivity issues resulting from such a client setup are therefore the sole responsibility of the client and are considered a risk of which the client is fully aware.
Clients should rely on verified TLS connections with MTS, as MTS provides a valid server certificate during the TLS handshaking phase, which clients should verify based on the FQDN they are connecting to.

In the case the SDK throws this error without an additional close-reason, the suggested approach is to check that whitelisting is correctly applied for the client's IP.

Error with close-reason example

```
Error opening the consumer channel with channelNumber: 577866037 and queueName:
vhost_name-Confirm-node2000.
RabbitMQ.Client.Exceptions.BrokerUnreachableException: None of the specified
endpoints were reachable --->
RabbitMQ.Client.Exceptions.OperationInterruptedException: The AMQP operation was
interrupted: AMQP close-reason, initiated by Peer, code=530, text="NOT_ALLOWED -
access to vhost '/vhost_name refused for user 'vhost_name': connection limit (30)
is reached", classId=10, methodId=40, cause= (edited)
```

Error without close-reason example

```
11:50:03,682	ChannelFactory  INFO	Creating connection ...
11:50:04,198	RabbitMqConsumerChannel		INFO	Error opening the
consumer channel with channelNumber: 1199410282 and queueName: vhost_name-
Confirm-node1.
11:50:04,200	RabbitMqConsumerChannel		INFO	Opening the consumer
channel will be retried in next 1500 ms.
11:50:05,709	ChannelFactory  INFO	Creating connection ...
11:50:06,234	RabbitMqConsumerChannel		INFO	Error opening the
consumer channel with channelNumber: 1199410282 and queueName: vhost_name-
Confirm-node1.
```

12.5 Building selection instances

The SDK supports markets used by the three Betradar feeds – LO (Live Odds), LCoO (Live Cycle of Odds) and UF (Unified Feed) implemented by different methods on the `SelectionBuilder` type.

- `SetId(string id);`
  This method should be used when building string representations of the market identifiers directly (without the help from the SDK).
- `SetIdLo(int type, int subType, string sov, string selectionId);`
  This method should be used when building market identifiers from information provided by the LO feed.
- `SetIdLcoo(int type, int sportId, string sov, string selectionId);`
  This method should be used when building market identifiers from information provided by the LCoO feed.
- `SetIdUof(Product product, URN sportId, int marketId, string selectionId, IDictionary<string, string> specifiers, IReadOnlyDictionary<string, object> sportEventStatus);`
  This method should be used when building market identifiers from information provided by the UF feed. Note: this method will throw if accessToken is not provided. Method parameter sportEventStatus needs the following keys:
    - HomeScore (home_score in sport event status)
    - AwayScore (away_score in sport event status)
    - Server (current_server in sport event status)

  If you are using UnifiedFeed sdk the map with the correct keys may be obtained:
  var sportEventStatusProperties = sportEvent.Status.Properties;

END OF DOCUMENT