Easy Commodity Trader API Guide

Version 4.6.3

TABLE OF CONTENTS

1 VERSION INFORMATION AND CHANGE LOG

Version	Description	Date	Author
0.1	Initial Draft	2016-03-07	James Watson
1.1	Market Depth	2016-09-16	Qiming Liu
1.2	Pricing Update	2016-10-28	Qiming Liu
1.3	Execution Feedback	2017-03-13	Qiming Liu
1.4	Market Segments Static Data	2017-06-22	Fabian Ottjes
3.0	Feature alignment with ECT	2018-01-22	Albert Rodriguez
3.1	Adding Good Till Session Feature	2018-02-15	Albert Rodriguez
3.4	Changes on Orders	2018-04-26	Albert Rodriguez
3.5	API tidy up for release	2018-05-25	Albert Rodriguez
3.6	Documentation clean-up	2018-05-31	Albert Rodriguez
3.7	Datetime alignment	2018-06-11	Albert Rodriguez
3.8	Datetime rollback	2018-06-15	Albert_Rodriguez
4.0	New API version	2018-07-24	Joe Field
4.1	RFQs	2018-08-03	Fabian Ottjes
4.2	RFQs: Currency support	2018-09-05	Fabian Ottjes
4.3	Reference Data: Added market area ids for order submission	2018-09-19	Joe Field
4.4	Trade Message Quantity Flow	2018-09-24	Fabian Ottjes
4.5	Removed Market Segments	2018-09-28	Fabian Ottjes
4.6	Introduction of ProductDefinition as replacement for Instrument	2018-11-13	Jan Truckenbrodt

Version 0.1

- Initial document

Version 1.1

- Price subscription returns an array for both bid prices and ask prices to reflect the entire market depth. The previous bid and ask fields are now obsolete.
- Price subscription now pushes delta updates, i.e. a snapshot is not sent on every update. Clients will need to process these updates to maintain a consistent state of prices. Each price (PriceQuantity message) now has an associated Id field to aid updates.
- Added Trade Execution section to describe how multi-depth trades are handled.

Version 1.2

- PriceQuantity now includes a UpdateType to indicate whether the update for the price in question is added, updated or removed. It is expected that the API consumer would use PriceQuantity.id to correlate the price updates. When the UpdateType is removed, the price and quantity fields are not populated.

Version 1.3

- If a trade execution errors, more information is now returned describing the reason for the error. ExecutionResponse now has a Message field.

Version 1.4

- The market segment attributes minimum trade quantity and quantity decimal places now belong to the market segment messages which are sent as part of the reference data messages. The same two attributes belonging to the PricesUpdate message are now marked as obsolete and should not be used anymore.

Version 3.0

- This version aligns all the existing changes occurred in the creation of ECT.
- It also introduces functionality to place resting limit orders.
- *Note that as a major version change it breaks compatibility with older versions*

Version 3.1

- Describing the new Good-Till-Session Behavior

Version 3.4

Some changes around orders

- Including the counterparty code and order source to the price stream
- *UpsertOrderRequest Version parameter is optional for updates*
- *UpsertOrder Contract Id and MarketArea Id are now int64 instead of int32*
- *SuspendOrderRequest Version parameter is now optional and changed its type to string*
- *ActivateOrderRequest Version parameter is now optional and changed its type to string*
- *CancelOrderRequest Version is now gone and required the corresponding Product Instance Id*
- *Minor API comment changes for better understanding*

Version 3.5

- Instrument now has the TimeZone
- *Product Instance now has the Instrument Id*
- *Product Instance name is now always the full display name on all objects*
- *UpsertOrderRequest contract id has now been changed to the generic name bookId*
- *OrderResponse now also sends the OrderId back, useful on Order Upserts*
- *Improvements on code comments for some properties*
- *Including in this document some basic steps gathering the most important data required for Upserting Orders*
- Changes in behaviour for OrderCommandResponse
    - We always now return an OrderId when possible
    - We always return an error inside the message property
- Documentation tidy up from client's feedback

Version 3.6

- Documentation cleanup

Version 3.7

- Alignment of dates, timestamp, delivery start and delivery end for trades, order and product instance all now follow the pattern yyyy-MM-ddTHH:mm:ss. Code comments in the proto file also reflect the change

Version 3.8

- Rolled back delivery start/end dates back previous format

Version 4.0

- Extensive rework of API for new version. See the comments in proto file for full details.
- Version 3.8 of the API continues to be supported with unchanged semantics
- Product Instance subscription is now separated from Price (renamed to Public Order) subscription
- Type usage has been consolidated: for example a lot of fields that were strings are now numeric types

Version 4.1

- Introduction of Request for Quotes (RFQs)

Version 4.2

- RFQs: A quote can now be requested in one of the currencies supported by the instrument; renaming of RFQ messages.

Version 4.3

- Reference Data: Added a new collection of Market Area ids to Instrument named OrderMarketAreaIds. This should now be used to select a market area id for order submission, instead of the existing MarketAreaIds collection. The MarketAreaIds collection should be used to select a market area id for RFQ submission.

Version 4.4

- Trade Message: A trade can now have multiple pairs of product instances and quantities (quantity flow) required for profile RFQs.

Version 4.5

- Market segments are no longer part of the reference data; orders no longer have to be upserted with market segment id.

Version 4.6

- To support new products for RFQ and continuous trading, a new entity called ProductDefinition has been introduced. It is meant to replace Instrument, so comprises all its properties plus new ones to have a richer product definition available.

2 INTRODUCTION

This document describes the AMPQ API which the Easy Commodity Trader server (herein referred to as *ECT*) exposes to external applications (herein referred to as *client*).

The API allows clients to subscribe to live pricing streams, trades to be executed and historical trade information to be requested.

The communication between client and server relies on the Advanced Message Queuing Protocol (AMQP) as a platform independent transport layer. ECT uses the RabbitMQ message broker which implement version 0-9-1 of the AMQP protocol. https://www.rabbitmq.com/tutorials/amqp-concepts.html. There are clients available for all major languages.

The messaged are encoded using google protocol buffer version 3 (referred to as ProtoBuf), see here https://developers.google.com/protocol-buffers/ for details. There is a .proto available that defines each message type. There are official implementations available in many languages including Java and C#.

The examples provided in the document will be in C# and are included for illustrative purposes only.

There are RabbitMQ and ProtoBuf libraries for many languages including but not limited to Java, Python, C++ and Go.

.

3 QUICK START

The following section provides a quick step by step description of how to establish a connection between an external application and ECT via AMQP.

1. Connect to the RabbitMQ broker using the broker credentials and/or certificate provided. Note here: the credentials are not the ECT user name and password, but the broker ones.
2. After a successful connection, the client should create and subscribe to a non-durable, exclusive, autodelete queue with a generated name.
3. Publish a *LoginRequest* with the ECT credentials provided during onboarding to the *ECT.Request* exchange, with an empty routing key.
4. If the login is successful, the client will receive a message with the matching correlation ID and *LoginResponse* as the Type. The deserialized payload will contain parameters for setting up heartbeats and the routing key that should be used for subsequent request messages.
5. Publish *KeepAlive* messages to the *ECT.Request* exchange, with the routing key specified in the *LoginResponse* message. It should do this every `keepAliveIntervalSeconds` seconds and should expect to receive at least one message (of any type) from ECT in that time. If no messages are received as part of normal operation, the server will send a KeepAlive message.

Note: when the client publishes a message to the *ECT.Request* exchange it should always include the following basic properties:

- ReplyTo: set to the queue name
- CorrelationId: set to a unique string
- Type: set to the type of the message as a string

If using version 4 or later of the API then the message should also include a header "Version" containing the major version as an integer. If this is not set the service will assume that the client is using version 3.

4 USING THE API

During the onboarding process, you will be provided with a hostname, a port number, a virtual host, a set of credentials and a client certificate for connecting to RabbitMQ and another set of credentials for connecting to ECT.

4.1 GENERATING PROTOBUF CONTRACTS

There are tools to generate classes from .proto files.

Have a look at the ProtoBuf tutorials for your language of choice https://developers.google.com/protocol-buffers/docs/tutorials to find out more.

4.2 CONNECTING TO ECT MESSAGE BROKER

The communication between Trading API clients and ECT is a brokered one, utilizing AMQP via RabbitMQ broker. To successfully connect to the message broker, the following requirements must be met:

- Connection must be encrypted using TLS 1.2, trusting ECT wild card certificate derived from:
  - DigiCert Global Root G2 root certificate, which can be obtained from the website: https://www.digicert.com/digicert-root-certificates.htm
- Certificate revocation list must be reachable
- Using AMQP version 0-9-1
- Outbound traffic to broker nodes on TCP port 5671 must be allowed
- ECT uses IP whitelisting for inbound traffic - the client must provide their IP addresses to ECT IT

The following endpoints are provided by ECT:

- Production:
  - node1-api.ect.energy (52.29.229.60)
  - node2-api.ect.energy (52.29.231.157)
- Demo/Integration:
  - node1-api-demo.ect.energy (35.157.46.17)
  - node2-api-demo.ect.energy (35.157.59.236)

The virtual host provided through the Ect message broker is:

- /ect2

The credentials for logging successfully into the message broker are as following:

- Username: <Provided by ECT Support when requesting Trading API access>
- Password: <Provided by ECT Support when requesting Trading API access>

Note: The credentials used to log into the message broker's virtual host are completely different from the ones used to establish an ECT Trading API session (see Chapter 4.3). They are broker specific and ECT core system does neither evaluate nor know them.

Example (C#):

Start by configuring the connection factory.

```
var connectionFactory = new ConnectionFactory
{
    HostName = config.HostName,
    Port = config.Port,
    UserName = config.Username,
    Password = config.Password,
    VirtualHost = config.VirtualHost,
    Ssl = new SslOption
    {
        Enabled = true,
        ServerName = config.HostName,
        Version = SslProtocols.Tls12,
    },
};
```

At this point you should be able to open a connection and create a channel.

```
var connection = connectionFactory.CreateConnection();
var model = connection.CreateModel();
```

You should now be able to connect to RabbitMQ. Next, set up the queue which will receive responses from ECT. The following code will create a queue that can only be accessed by the current connection.

```
var queueName = _model.QueueDeclare().QueueName;
```

You should create a consumer for this queue to pick up incoming messages.

```
var consumer = new EventingBasicConsumer(model);
model.BasicConsume(queueName, true, consumer);
consumer.Received += QueueOnReceived;
```

4.3 Logging Into ECT

After having successfully connected to the ECT message broker, the next step is to establish an ECT session itself.

This is done by sending a *LoginRequest* message. The credentials to use are the ones, send out via email and are different from the broker ones (see previous chapter).

> **Note:** The ECT trading API user's password has an expiration period of one year after creation. Also, the password expires 'silently' which means, that there is no notification mechanism in place informing about a looming password or session expiration!

Logging into ECT involves sending a *LoginRequest* message to the `ECT.Request` exchange with an empty routing key. The ReplyTo field should be set to the name of the client queue created earlier. ReplyTo, Type and CorrelationId should always be provided when sending requests to ECT to uniquely identify and correlate requests their respective responses on client side.

If using version 4 or later of the API then the message should also include a "Version" header, illustrated below:

```
var loginRequest = new LoginRequest
{
    UserName = «<ect user name>”,
    Password = “<ect password>”
};

var basicProperties = model.CreateBasicProperties();
basicProperties.ReplyTo = queueName;
basicProperties.Type = nameof(LoginRequest);
basicProperties.CorrelationId = Guid.NewGuid().ToString();
basicProperties.Headers = new Dictionary<string, object> {{"Version", 4}};

model.BasicPublish(
    exchange: "ECT.Request",
    routingKey: "",
    basicProperties: basicProperties,
    body: loginRequest.ToByteArray());
```

If the credentials are correct, ECT will send a message back to the client queue. The response will contain a routing key that should be set for all subsequent requests, as it is identifying the session – without it, messages cannot correctly corelated to the client’s session and therefore will be dropped silently. It will also contain some parameters for sending and receiving heartbeats.

4.4 Session time-out and sending KeepAlives

By successfully logging into ECT via trading API, a dedicated session has been created on ECT side for the connected client. The session has an expiration timer attached which automatically expires, if no message of any type has been received from the client for a period of:

LoginResponse.keepAliveIntervalSeconds x LoginResponse.disconnectAfterMissedKeepAlives

seconds. Any message received from the client will reset this timer. To avoid session expiration, the trading API client should regularly send messages to ECT, e.g. by *KeepAlive*.

The ECT service also sends messages to the client on a regular basis: if no message of any type has been sent to the client within a period of *LoginResponse.keepAliveIntervalSeconds* seconds, a dedicated *KeepAlive* message will be send.

Thus, a client should consider an active session to be timed-out, if he did not receive any type of message since a period of:

LoginResponse.keepAliveIntervalSeconds x LoginResponse.disconnectAfterMissedKeepAlives

seconds. If the client detects a timed-out session, it does not need to send a dedicated *DisconnectionRequest*, but can directly start over with a new *LoginRequest* as pointed out in the previous chapter.

Example code (C#):

```
var timer = new Timer(_ =>
{
    var keepAlive = new KeepAlive();

    var basicProperties = model.CreateBasicProperties();
    basicProperties.ReplyTo = _queueName;
    basicProperties.Type = nameof(KeepAlive);
    basicProperties.CorrelationId = Guid.NewGuid().ToString();
    basicProperties.Headers = new Dictionary<string, object> { { "Version", 4 } };

    model.BasicPublish(
        exchange: "ECT.Request",
        routingKey: RoutingKey,
        basicProperties: basicProperties,
        body: keepAlive.ToByteArray());
});

timer.Change(0, response.KeepAliveIntervalSeconds * 1000);
```

The picture below illustrates the session expiration of ECT:

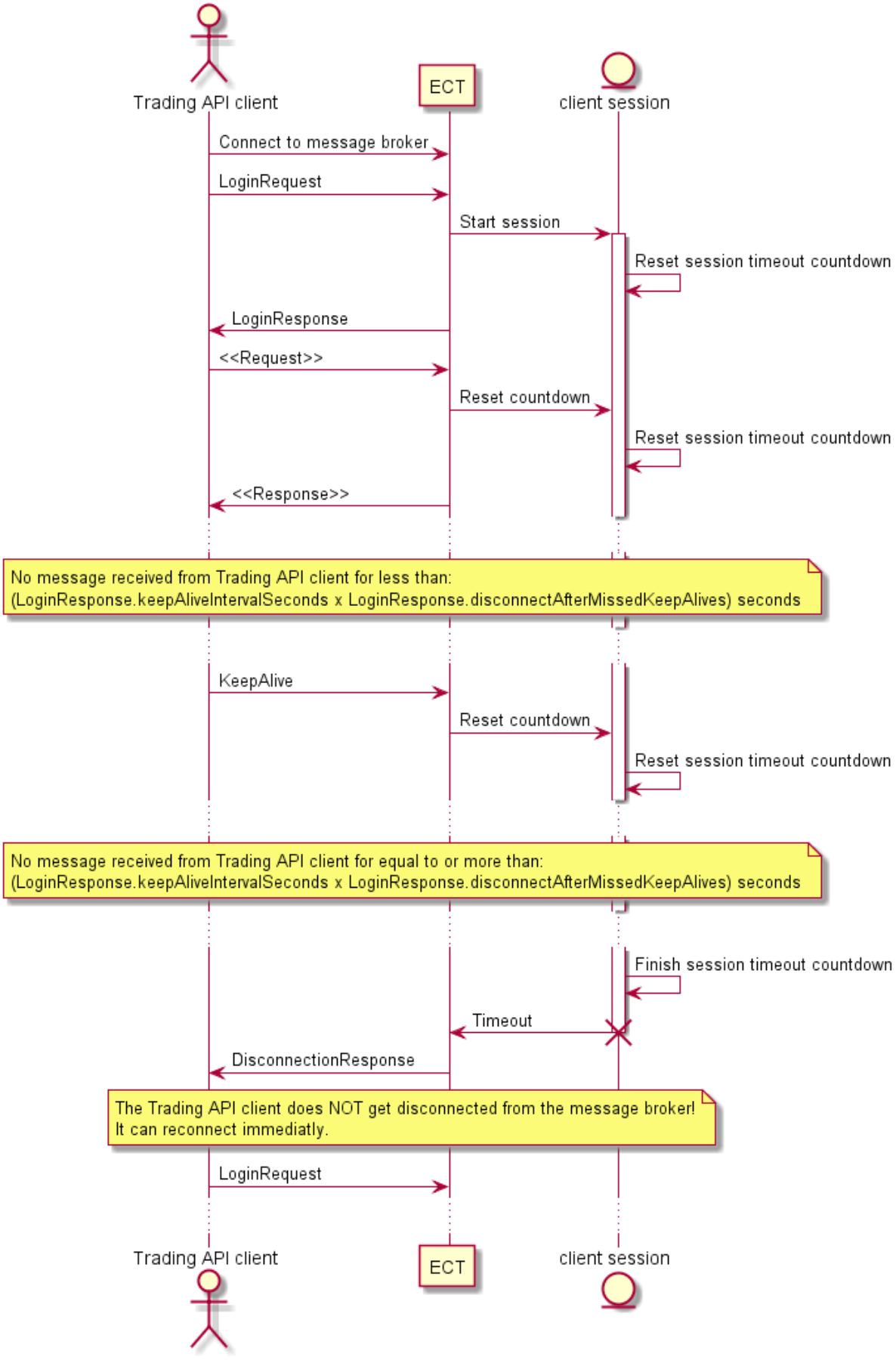

Figure 1: ECT session handling

4.5 SESSION DUPLICATION

Currently, there is no support for duplicated sessions in ECT. A duplicated session is a session sharing the same ECT credentials (username and password).

If any client established a valid session and a second client logs into ECT using the very same credentials, the first session will be terminated with *DisconnectionResponse* message having a *DisconnectionReason. DC_DUPLICATE_SESSION* value set.

4.6 GOOD TILL SESSION ORDER LIFECYLE

Trading API users' order lifecycle differs from that of Trading Ui users. All pending orders will be cancelled as soon as the session is closed either intentionally as a normal logout or forced (users permissions change or deactivation).

5 USING THE TRADING API

5.1 OPERATIONAL MODES

ECT has two operational modes:

1. Request/Response
2. Subsbriptions

5.2 REQUEST/RESPONSE

Some data and functionality can only be used through Request/Response messages. Each request message send to ECT must have a valid routing key set (see also section 4.3). All request messages not having a valid routing key, will be silently dismissed.

If any Request message contains a client set Correlation ID in the message header, each Response message generate by ECT will also have the very same Correlation ID set in its message header.

The following Request and correlated Response messages are available in the Trading API (see also the attached '.proto' file):

Request Type	Response Type
LoginRequest	LoginResponse
DisconnectionRequest	DisconnectionResponse
ReferenceDataRequest	ReferenceDataResponse
ProductInstancesSubscriptionRequest	ProductInstanceSubscriptionResponse
ProductInstancesUnsubscriptionRequest	ProductInstanceUnsubscriptionResponse
PublicOrdersSubscriptionRequest	PublicOrdersSubscriptionResponse
PublicOrdersUnsubscriptionRequest	PublicOrdersUnsubscriptionResponse
TradesSubscriptionRequest	TradesSubscriptionResponse
TradesUnsubscriptionRequest	TradesUnsubscriptionResponse
OrdersSubscriptionRequest	OrdersSubscriptionResponse
OrdersUnsubscriptionRequest	OrdersUnsubscriptionResponse
RfqsSubscriptionRequest	RfqsSubscriptionResponse
RfqsUnsubscriptionRequest	RfqsUnsubscriptionResponse
GetTradesRequest	GetTradesResponse
UpsertOrderRequest	OrderCommandResponse

CancelOrderRequest	OrderCommandResponse
WithdrawOrderRequest	OrderCommandResponse
ActivateOrderRequest	OrderCommandResponse
InsertRfqRequest	RfqCommandResponse
TradeRfqRequest	RfqCommandResponse
CancelRfqRequest	RfqCommandResponse
SubmitRfqQuoteRequest	RfqCommandResponse

5.3 SUBSCRIPTIONS

Subscriptions provide data by pushing them from the ECT server to the Trading API client, without having the client to send a dedicated *Request* message.

ECT supports the following subscriptions:

Request Type	Response Type	Update Type
ProductInstancesSubscriptionRequest	ProductInstancesSubscriptionResponse	ProductInstancesUpdate
PublicOrdersSubscriptionRequest	PublicOrdersSubscriptionResponse	PublicOrdersUpdate
OrdersSubscriptionRequest	OrdersSubscriptionResponse	OrdersUpdate
RfqsSubscriptionRequest	RfqsSubscriptionResponse	RfqsUpdate
TradesSubscriptionRequest	TradesSubscriptionResponse	TradesUpdate

For each type only ONE subscription exists per client session. If multiple subscription requests have been submitted to ECT they all will be combined into one single subscription of the respective type. This means especially, that any *UnsubscribeRequest* message send to ECT will stop ALL subscriptions for the specified subscription type:

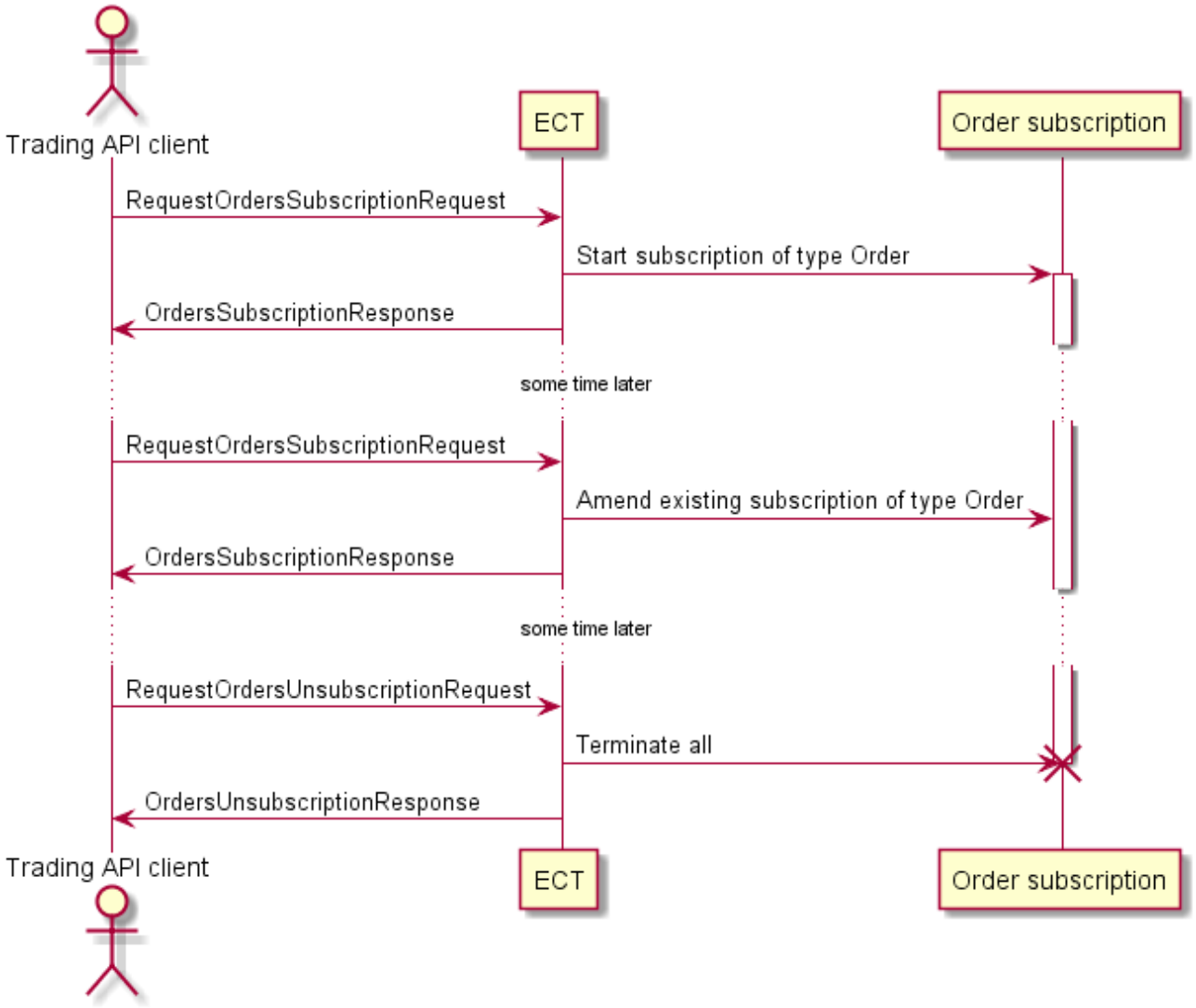

Figure 2: ECT subscription aggregation

5.3.1 Subscription Workflow

Each subscription is started with a *SubscriptionRequest* message of the respective type. The subscription has been started only, if the corresponding *SubscriptionResponse* message contains no error (*isError* set to *false*).

After the *SubscriptionResponse* message, ECT sends out a first *Update* message, having set the *isStateOfTheWorld* property to *true*. This Update message is said to contain the State-Of-The-World snapshot (hereafter called SOTW). The SOTW contains all entities at the current point in time, when the subscription has been started. Also, only entities for which the party has privileges are available through the subscription.

After the initial SOTW snapshot has been sent to the client, only delta Update messages, containing all changed entities since the last Update, will be send to the client. Delta Update messages have the *isStateOfTheWorld* property set to *false*.

The client must treat any SOTW Update message for an active subscription as reset: all data received before the SOTW Update must be considered invalid and only the new SOTW represents the current subscription. Normally, only one SOTW message will be send per subscription. But in case of subscription aggregation (see chapter 5.3), additional SOTW messages will be received for an active subscription:

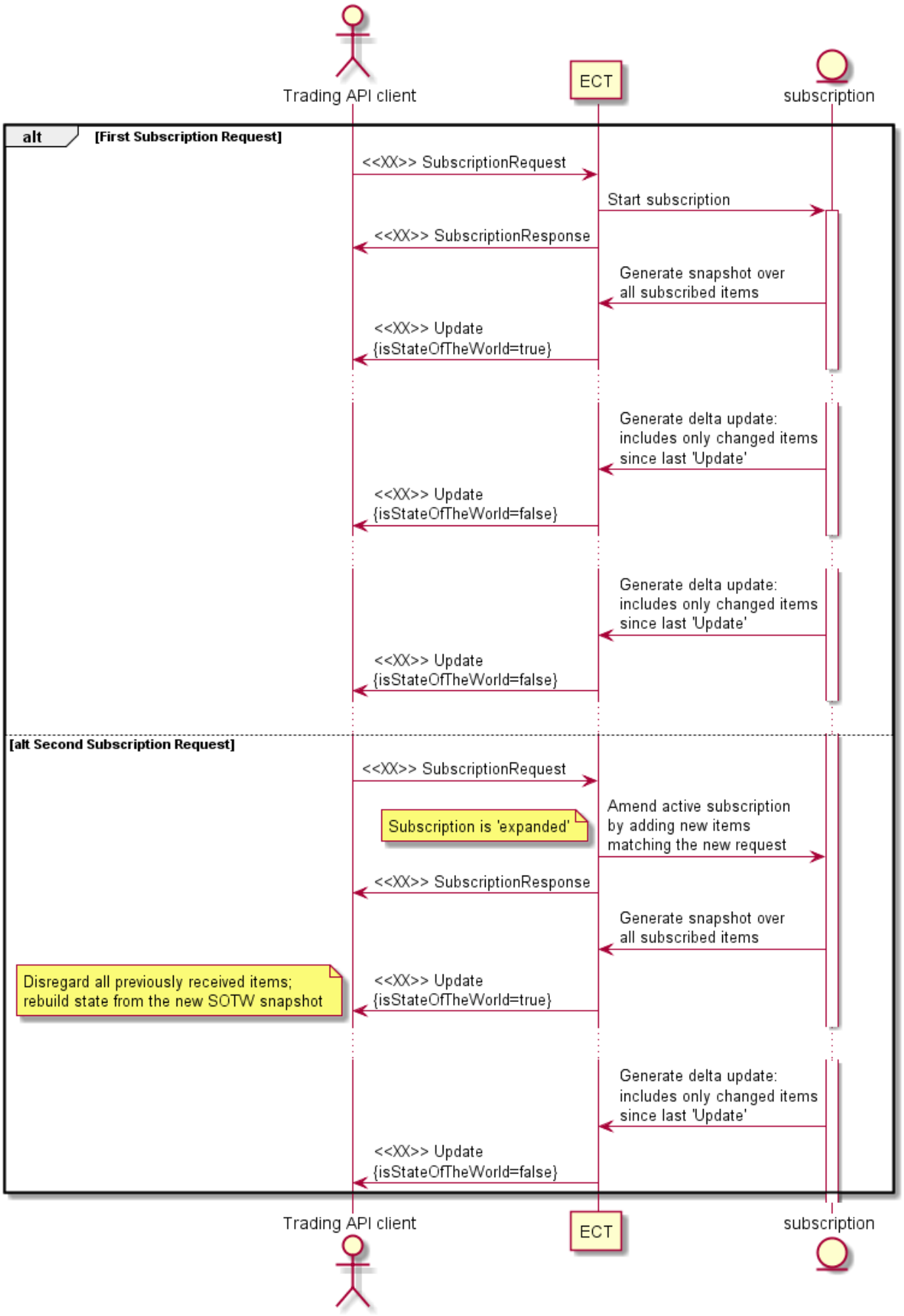

Figure 3: ECT subscription workflow

6 WORKING WITH DATA

ECT Trading API provides different kinds of data:

1. Static Data
2. Reference data
3. Subscription data

6.1 STATIC DATA

Static data is provided in as enumerations. Changes to an enumeration will result in an update of the Trading API version. With that, changes will be communicated to clients explicitly.

6.2 REFERENCE DATA

Reference data in ECT is used to describe the basic properties for tradable products. The difference between Static and Reference data is, that reference data can be enhanced without changing the Trading API version (e.g. by adding a new currency, which is just a new entry in the Currencies collection).

Further, all reference data is scoped:

- Global: this data is available to **all ECT parties**/users
- Own Party: this data is only available to users **of the same party** the user account used by the Trading API client.

Reference data can only be obtained through *Request*/*Response* messages, pushing of updates to reference data items is not available.

The following table describes the available properties of *RequestDataResponce* message in more detail:

Property	Type	Scope	Description
periodTypes	PeriodType collection	Global	Contains all period types used in ECT. A single period type could be: *Hour*, *Month*, …
commodities	Commodity collectuon	Global	Contains all commodities available in ECT. A single commodity could be: *Power*, *Gas*
currencies	Currency collection	Global	Contains all currencies available in ECT. A single currency could be: *EUR*, *CZK*
productTypest	ProductType collection	Global	Contains all product types in ECT. A single value could be: *Fix*, *Index*, *Swap*
quantityUnits	QuantityUnit collection	Global	Contains all quantity units available in ECT. A single value could be: *MW*
books	Book collection	Own Party	Contains all contract agreements or portfolios (unified under term *Book*) available for the ECT party.
marketAreas	MarketArea collection	Own Party	Contains all market areas used by at least one product definition setup for the ECT party. A single value could be: *Amprion*, *TTF*, *OTE*, …
indices	Index collection	Own Party	Contains all indices used by at least one product definition setup for the ECT party. A single value could be: *None*, *LEBA NCG DA CZK*

			Note: if a product does not use an index (e.g. fix price product) the index is always set to '*None*'
strategies	Strategy collection	Own Party	Contains all strategy names available for the ECT party.
productDefinitions	ProductDefinition collection	Own Party	Contains all product definitions available for the ECT party. A product definition (sometimes called a contract) defines the basic properties of a tradable product instance (the What, Where, How)

Hint: Although all entities provided through Trading API have numeric IDs, it is not recommended to rely on the ID (e.g. for mapping purposes). Reason is, that IDs are not equal over the different ECT environments (Production/Demo/Integration/…). Also, there is no common scheme, how entities obtain their ID. For mapping purposes, clients should rather use names.

Most of the reference data types comprise only two properties:

Property	Type	Description
id	Int64	A unique ID within the type. Used for reference purpose from other entities/items.
name	String	The concrete name of the object

This applies to types *PeriodType*, *Commodity*, *Currency*, *ProductType*, *QuantityUnit*, *Book*, *MarketArea*, *Index*.

ProductDefinition is more complex:

Property	Type	Description
id	Int64	A unique ID within the type. Used for reference purpose from other entities/items.
name	String	The concrete name of the object. e.g. *Power Dutch Intraday-Hour*
commodityId	Int64	References the underlying commodity of this object.
defaultCurrencyId	Int64	References the default currency in which prices are quoted.
additionalCurrencyIds	Int64 collection	Contains all additional currencies in which prices could be submitted or provided – if no additional currencies are supported, this collection is empty. The currency referenced in *defaultCurrencyId* is never part of this collection.
quantityUnitId	Int64	References the quantity unit in which quantities of the underlying commodity are provided or submitted.
periodTypeId	Int64	References the period type in which delivery periods of product instances referencing this object, are provided. e.g. *Hour*, *Month*
deliveryType	DeliveryType enum	An enum value defining how trades, done on product instances referencing this object, are settled. e.g. DT_PHYSICAL, DT_FINANCIAL
productTypeId	Int64	References the product type which defines the nature of this object. The following values are possible: • Fix: trades done are settled with a fix price derived directly from the trade

		• Index: trades done are settled with floating price derived from the settlement price of the product definition's index • Swap (Fix vs. Index): trades done are settled with floating price derived from the settlement price of the product definition's index and the fix price of the trade
indexId	Int64	References the index used for price calculation – if no index is used for price calculation (e.g. Fix) the well-known index '*None*' is referenced.
marketAreas	Int64 collection	References the market area used for (virtual) delivery or off-take of the underlying commodity.
timeZone	String	time zone name as provided by TZDB (aka IANA time zone data - https://www.iana.org/time-zones) in which delivery start and end of product instances referencing this object will be expressed.
minPriceStep	Double	minimum step in which prices can be submitted, e.g. 0.01 - also used to define the maximum number of allowed price decimals: 0.1 => 1 decimal 0.01 => 2 decimals
minQuantityStep	Double	minimum step in which quantity values can be submitted, e.g. 0.01 - also used to define the maximum number of allowed quantity decimals 0.1 => 1 decimal 0.01 => 2 decimals

6.3 SUBSCRIPTION DATA

As described in section 5.3, some data is available through subscriptions:

1. Product Instances
2. Public Orders
3. Private Orders (Party Owned)
4. Trades (Party Owned)
5. RFQs (Party Owned)

6.3.1 Product instances

Product instances represent concrete products. Depending on the setup, some product instances have an orderbook associated. If so and assuming the client has privileges to do so, it can submit orders into the orderbook of the product instance.

If a product instance does not have an orderbook associated, it still might be used in RFQ submission.

As all commands for public and private orders as well as RFQs need a product instance ID, a subscription to this type is vital for a Trading API client.

To start or expand a product instance subscription, the Trading API client should submit a *ProductInstancesSubscriptionRequest* message containing *ProductInstanceSubscriptionParams* which define the concrete product definitions for which all available product instances are to be retrieved (see also section 5.3).

The following table describes the *ProductInstance* type in detail:

Property	Type	Description
updateType	UpdateType enum	Defines the kind of change for this product instance: • UT_ADDED: the instance has been added (mostly due to new creation) • UT_UPDATED: an existing instance has been updated • UT_REMOVED: an existing instance has been removed (mostly due to expiry) If the message is a SOTW one, then this value is always UT_ADDED.
id	Int64	A unique ID within the type. Used for reference purpose from other entities/items.
name	String	The concrete name of the object. e.g. *GAS GPL HIGH - DA*
displayName	String	Short name of the product instance, mostly used for displaying purpose. e.g. *DA, H14*
deliveryStart	DateTimeOffset	A timestamp when the (virtual) delivery of the underlying's commodity will start. This is an inclusive boundary. The timestamp is expressed as local time with offset in the time zone of the referenced product definition.
deliveryEnd	DateTimeOffset	A timestamp when the (virtual) delivery of the underlying's commodity will end. This is an exclusive boundary. The timestamp is expressed as local time with offset in the time zone of the referenced product definition.

tradingEnd	DateTimeOffset	Time at which the product instance's orderbook will be closed for continuous trading and at which all orders will be evicted. Also, after the orderbook has been closed, submission of new orders will not be possible any more.
durationInHours	Double	The number of total hours in which a (virtual) delivery happens, e.g.: 0.25 => quarter hour 23 => 23 hours on daylight savings days
permissionsByBook	PermissionsByBook	Defines the permissions a Trading API client has on the product instance (e.g. buy or sell through specific contract agreement, instant vs. limit order, …)
productDefinitionId	Int64	References the ProductDefinition object defining the basic product properties.

Each product instance has two periods assigned: Trading period and Lifetime period.

The lifetime period defines, from when to when a product instance exists in general. Once the lifetime is exceeded the product instance is said to have expired. After it has expired, it will be removed from ECT.

The trading period defines, from when to when a product instance has an orderbook attached and thus orders could be setup. As soon as the trading period expires, the attached orderbook will be expired as well. All standing orders at the time of orderbook expiration will be expired as well. The order expiration will be pushed to the client through an existing private order subscription. Normally, the trading period ends BEFORE the lifetime, resulting in a gap, in which the product instance exists but cannot be traded anymore.

There are some product instances, which do not have a trading period at all. These ones are mainly meant for setting up RFQs.

At this point, there is no way to determine explicitly from *ProductInstance* fields, whether an orderbook is attached or not. Only the *OrderCommandResponse* to a sent *OrderRequest* command will provide this detail in form of the *CommandResult* code CR_FAILED_DUE_TO_MISSING_ORDER_BOOK.

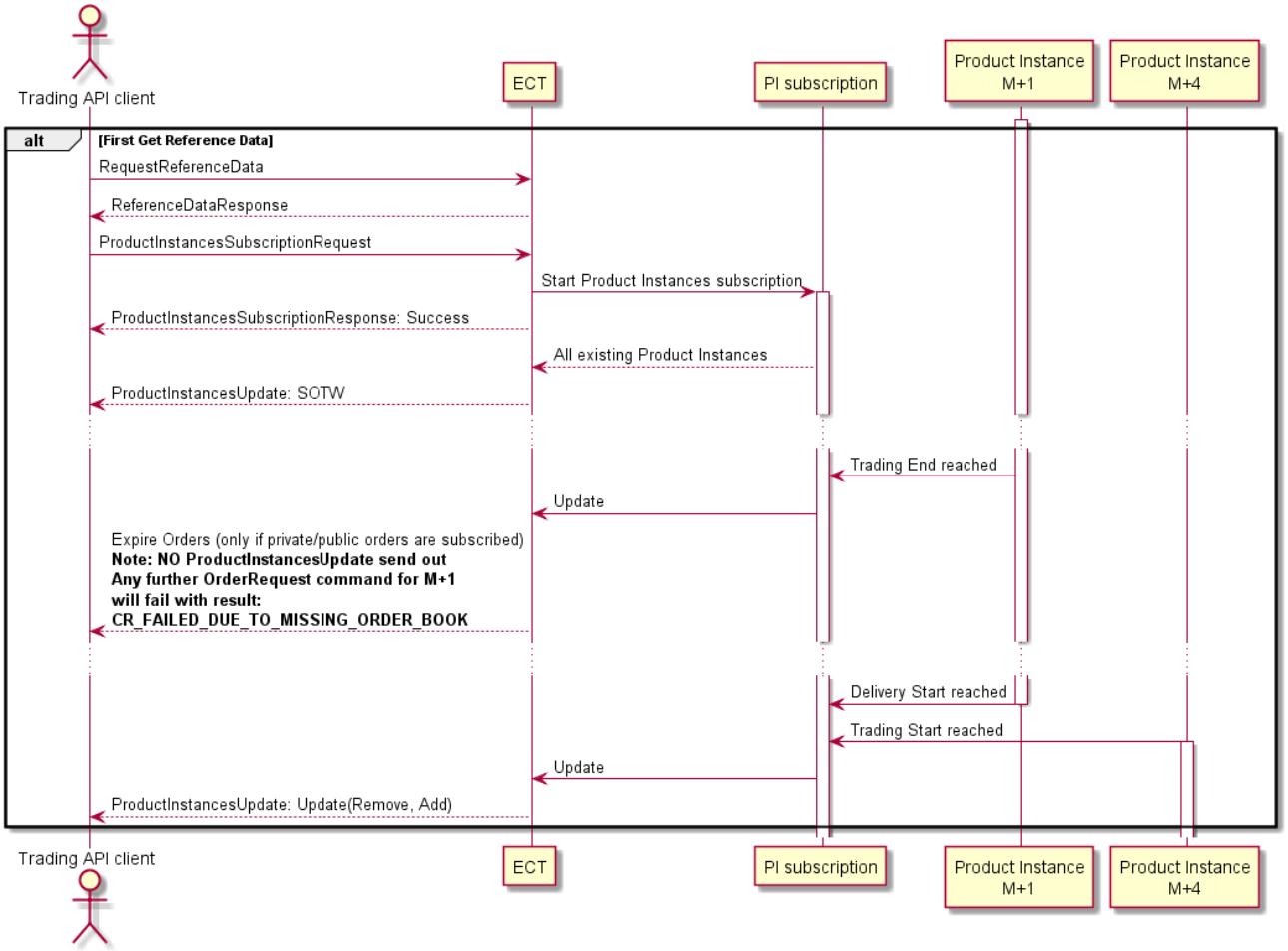

Figure 4: ECT Product Instance update and expiry

Some product instances have also a varying delivery start. E.g. all product instance for Gas Balance-Of-Week have a delivery start depending on the current point in time: as time passes, the delivery start gets adjusted until it is not possible any more and eventually the product instance will expire (removed). Each change of the delivery start will also be announced by a *ProductInstanceUpdate* message.

6.3.2 Public Orders

Public orders are orders available to all ECT clients, independently of the party. Through the public order subscription, it is possible for a client to retrieve a complete orderbook for a specific product instance.

To start or expand a public order subscription, the Trading API client should submit a *ProductInstancesSubscriptionRequest* message containing *ProductInstanceSubscriptionParams* which define the concrete product instances for which public orders are to be retrieved (see also section 5.3).

As by nature all clients could see all public orders, they only contain a basic set of data, to not disclose sensitive client information to other parties.

The following table describes the *PublicOrderUpdate* type in detail:

Property	Type	Description
updateType	UpdateType enum	Defines the kind of change for the whole orderbook of the product instance:

		• UT_ADDED: the product instance and orderbook have been added (e.g. subscription start or new creation of product instance) • UT_UPDATED: existing product instance's orderbook has been updated (e.g. single order has been added, removed or changed) • UT_REMOVED: existing product instance and associated orderbook have been removed (e.g. after expiration or change of permissions) If the message is a SOTW one, then this value is always UT_ADDED.
productInstanceId	Int64	The product instance associated to the affected orderbook.
bidUpdates	PublicOrder collection	Contains all bid orders affected by this update.
askUpdates	PublicOrder collection	Contains all ask orders affected by this update.

The following table describes the *PublicOrder* type in detail:

Property	Type	Description
updateType	UpdateType enum	Defines the kind of change for the order: • UT_ADDED: the order has been added (e.g. subscription start or new creation) • UT_UPDATED: existing order has been updated (e.g. if the owner changed the price or the order has been partially executed) • UT_REMOVED: existing order has been removed (e.g. after expiration or cancellation) If the message is a SOTW one, then this value is always UT_ADDED.
orderId	Int64	A unique ID of the order within ECT.
price	Double	The price of the order expressed in the default currency of the product instance's product definition.
quantity	Double	The quantity of the order expressed in the quantity unit as set in the product instance's product definition.
counterpartyCode	String	The short name of the party owning this order.
orderSource	OrderSource enum	Provides details about the order's owner: • OS_MINE: the order has been created by the Trading API user. • OS_MY_PARTY: the order has been created by a different user of the same party as the Trading API user belongs to. • OS_OTHER: any other order (not from the Trading API client's party)

6.3.3 Private Orders

Private orders are all orders owned by the party, the current Trading API user belongs to. So, these are especially the same orders as received through the public order subscription with *orderSource* set to *OS_MINE* or *OS_MY_PARTY*. In contrast to a public order, a private one contains more properties. Also, the private orders are not provided as part of orderbooks but as a single continuous stream.

Further there is no UpdateType field available to describe whether an order has been added, changed or removed. Instead this must be derived from a different field: *orderStatus*.

To start or expand a private order subscription, the Trading API client should submit a *ProductInstancesSubscriptionRequest* message containing *ProductInstanceSubscriptionParams* which define the concrete product instances for which public orders are to be retrieved (see also section 5.3).

The following table describes the private *Order* type in detail:

Property	Type	Description
orderId	Int64	A unique ID of the order within ECT.
version	Int32	The current version of the order - each change to any of the order's fields will increase the version value. Any operation on orders is only possible if the correct version is provided in *Request* messages send to ECT. This is to ensure multiple requests do not change orders in an unintended way (concurrency protection). Also note: if an order has been published on the subscription with version 0 (zero), it means the order has been created; any version value greater than 0 is an update or removal.
status	OrderStatus enum	Current status of the order: • OS_ACTIVE: the order is waiting in the orderbook to be matched • OS_DONE: the order has been removed from the orderbook and has been partially or fully executed • OS_REJECTED: The order has not been added to the orderbook as pre-checks failed • OS_SUSPENDED: the order is in the orderbook, but inactive and thus will not be used for matching • OS_CANCELLED_BY_USER: the order has been removed from the orderbook as it was cancelled by a user • OS_CANCELLED_BY_SYSTEM: order has been removed from the orderbook by system (e.g. permission change) • OS_EXPIRED: the order has been removed as the orderbook ceased to exist due to product instance expiration • OS_FAILED: the order has been removed from the orderbook and has not been (partially) executed at all
orderType	OrderType enum	Type of the order: • OT_FOK: Fill-Or-Kill - the order should be executed immediately upon adding it to the orderbook AND only with full quantity (no partial execution); if not possible, remove it immediately from the orderbook. • OT_GTC: Good-Till-Cancel - the order is set up as limit order, resting in the orderbook until it gets fully matched, cancelled, or if created by a Trading API client, the session terminates. Partial execution is allowed and will reduce the remaining quantity only.

		• OT_IOC: Immediate-Or-Cancel - the order should be executed immediately upon adding it to the orderbook. In contrast to Fill-Or-Kill, partial execution is allowed; The remaining (partial) order should be cancelled immediately after the matching process. • OT_AON: All-Or-Nothing - the order is set up as limit order, resting in the orderbook until it gets fully matched, cancelled, or if created by a Trading API client, the session terminates. In contrast to Good-Till-Cancel, partial execution is NOT allowed.
clientDirection	Direction enum	The order direction from the client's POV: • DIR_BUY: The client wants to buy the underlying • DIR_SELL: the client wants to sell the underlying
productInstanceId	Int64	References the product instance, which's orderbook contains the order.
price	Double	The price of the order expressed in the default currency of the product instance's product definition.
quantity	Double	The quantity of the order expressed in the quantity unit as set in the product instance's product definition.
filledQuantity	Double	Contains the already executed quantity in case of partial execution.
remainingQuantity	Double	Contains the remaining quantity in case of partial execution. If not executed yet, this value is equal to *quantity.*
totalVolume	Double	Contains the total volume, if the order would be fully executed. It takes schedules (Base, Peak, Atomic) into account.
marketAreaId	Int64	References the market area of the (virtual) delivery or take-off.
bookId	Int64	References the contract agreement or portfolio into which trades, resulting from this order, will be booked.
timeStamp	DateTimeOffset	The time stamp of when the order was created in ECT.
trader	String	Full name of the user, who owns(created) the order.
clientCorrelationId	String	An optional ID set by the client. This value can be used to track the order on client systems with an own ID. It is only visible to users of the same party.
comment	String	An optional comment set by the client. It is only visible to users of the same party.
strategy	String	An optional value set by the client. It is only visible to users of the same party.

6.3.1 Trades

The trades subscription is very much like the private orders one: only trades are published, which were concluded on orders of the party the Trading API client belongs to.

The following table describes the *Trade* type in detail:

Property	Type	Description
id	Int64	A unique ID of the trade within ECT.
party	String	The full name of the counterparty from the client's party POV.
timeStamp	DateTimeOffset	The time stamp of when the trade was created in ECT.

privateOrderId	Int64	If the trade has been concluded over an order, this value is the unique ID of the order (see also *Order.orderID*). If the trade has been concluded over an RFQ, this value is 0 (zero).
rfqId	Int64	If the trade has been concluded over an order, this value is 0 (zero). If the trade has been concluded over an RFQ, this value is the unique ID of the RFQ (see also *Rfq.rfqID*).
clientDirection	Direction enum	The order direction from the client's POV: • DIR_BUY: The client buys the underlying • DIR_SELL: the client sells the underlying
price	Double	The price of the trade expressed in the provided *currency*.
currency	String	The ISO-4217 code of the currency in which *price* is provided
quantityFlow	TradeQuantityFlowItem collection	A collection of *TradeQuantityFlowItem* items representing the different leg flows.
quantityUnit	String	The unit of measurement in which the *quantity* of each is *TradeQuantityFlowItem* expressed. (e.g. MW)
marketArea	String	Name of the market area of the (virtual) delivery or take-off.
book	String	The name of the contract agreement or portfolio into which the trade has been booked.
trader	String	The full name of the trader who concluded the trade. This field is always filled from POV of the Trading API client – so only the trader of the own party is shown here.
confirmationID	String	Contains the confirmation ID of the trade store used by ECT. This value is only provided in case of a RFQ trade. Also note: the trade life cycle is implemented in a way, that the trade is published BEFORE the confirmation ID of the underlying trade store has been received. As soon as the confirmation ID has been received, another *Trade* message will be published, having this field set to the received value.
clientCorrelationId	String	An optional ID set by the client. This value can be used to track the order or RFQ on client systems with an own ID. It is only visible to users of the same party.
comment	String	The optional comment set by the client on the order or RFQ. It is only visible to users of the same party.
strategy	String	The optional value set by the client on the order or RFQ. It is only visible to users of the same party.

Hint: Unlike *Order*, *Trade* does not have a *version* or an *updateType* field. This makes it harder to track changes. At best, the Trading API client tracks on its side the trades of interest and checks each Trade message for an already pushed trade ID – if a message with the same ID.had been pushed before, it is an update.

The following table describes the *TradeQuantityFlowItem* type in detail:

Property	Type	Description
productInstanceName	String	The name of the product instance, for which this (virtual) quantity flow is designated.
deliveryStart	DateTimeOffset	A timestamp when the (virtual) delivery of the quantity flow will start. This is an inclusive boundary. The timestamp is expressed as local time with offset in the time zone of the referenced product definition.
deliveryEnd	DateTimeOffset	A timestamp when the (virtual) delivery of the quantity flow will end. This is an exclusive boundary. The timestamp is expressed as local time with offset in the time zone of the referenced product definition.
quantity	Double	The quantity expressed in unit of measurement defined in *Trade.quantityUnit*.

6.3.1 RFQs

The *Rfq* subscription is very much like the private orders one: only *Rfq*s, which are submitted by the party of the Trading API client, are pushed through. As the *Order* type, *Rfq* has a *version* field, which is used to indicate changes on a *Rfq.* Further, *version* is used as protection against unintended concurrent change.

The following table describes the private *Rfq* type in detail:

Property	Type	Description
rfqId	Int64	A unique ID of the RFQ within ECT.
version	Int32	The current version of the RFQ - each change to any of the order's fields will increase the *version* value. Any operation on RFQs is only possible if the correct version is provided in *Request* messages send to ECT. This is to ensure multiple requests do not change RFQs in an unintended way (concurrency protection). Also note: if a RFQ has been published on the subscription with *version* 0 (zero), it means the RFQ has been created; any *version* greater than 0 is an update or removal.
status	RfqStatus enum	Current status of the RtFQ: • RS_SUBMITTED: the RFQ has been submitted and awaits quotation. • RS_QUOTED: the RFQ has been quoted and awaits acceptance. • RS_EXPIRED: the RFQ was quoted but the client did not accept nor cancel and thus after specified time the RFQ has been cancelled by the system. • RS_TRADED: the RFQ was quoted and the client accepted the quote. • RS_CANCELLED: the RFQ has been cancelled by either system or user.
direction	Direction enum	The RFQ direction from the client's POV: • DIR_BUY: The client wants to buy the underlying • DIR_SELL: the client wants to sell the underlying

requester	RfqTradingParty	Describes the submitting party (of the user who submitted the RFQ and waits for quotation)
quoter	RfqTradingParty	Describes the quoting party (of the user who generates the quote for the submitted RFQ)
marketAreaId	Int64	References the market area of the (virtual) delivery or take-off.
quantityFlow	QuantityFlowItem collection	A collection of *QuantityFlowItem* items representing the different leg flows.
isScheduled	Bool	This field is set to *true*, if the RFQ should be quoted at the specified *scheduledTime*; otherwise, the field is *false*.
scheduledTime	DateTimeOffset	Only used if *isScheduled* is set to *true*: The timestamp at which the quote should be submitted.
quoteExpiryTimestamp	DateTimeOffset	Only available, if a quote has been provided (see *status* RS_QUOTED): the timestamp at which the RFQ will automatically been cancelled, if the quote has not been accepted by the submitting client.
quote	Double	Only available, if a quote has been provided (see *status* RS_QUOTED): the quoted price referring to a single volume unit, provided by the RFQ quoter (e.g. in EUR per 1MWh, if currency is EUR and quantity unit MW). Depending on the underlying product type, this value could either be a fix price, or an add-on added to a price being available only at a later point in time (e.g. after index settlement)
currencyId	Int64	References the *Currency*, in which the quoted price is provided.
quoteComment	String	A public comment field, which is visible to both parties. Its content can only be set by *quoter*, not *requester*.
terminatedReason	TerminatedReason	Specifies the reason of the RFQ termination (see enum *TerminatedReason*): • TR_NONE: the default state, as long as the RFQ has not been terminated. • TR_PRODUCT_INSTANCE_EXPIRED: the RFQ has been expired, before any quote has been provided. • TR_QUOTE_EXPIRED: the RFQ has been expired after a quote was provided but before it was accepted. • TR_REQUESTER_CANCELLED: the RFQ has been actively been cancelled by the submitting party. • TR_QUOTER_CANCELLED: the RFQ has been cancelled by the quoting party. • TR_SYSTEM_CANCELLED: the RFQ has been cancelled by ECT because of technical errors.

The following table describes the *RfqTradingParty* type in detail:

Property	Type	Description
party	String	The full name of the party which submitted the RFQ or provided the quote.
trader	String	The full name of the trader who submitted the RFQ or the quote and thus belongs to *party*.
book	String	The contract agreement or portfolio under which the RFQ has been submitted or the quote has been provided.
clientCorrelationId	String	An optional ID set by the client. This value can be used to track the RFQ on client systems with an own ID. It is only visible to users of the same party.
comment	String	An optional comment set by the client. It is only visible to users of the same party.
strategy	String	An optional value set by the client. It is only visible to users of the same party.
timestamp	DateTimeOffset	A timestamp at which either RFQ has been submitted or the quote was provided (depending on the party's role)

The following table describes the *QuantityFlowItem* type in detail:

Property	Type	Description
productInstanceId	Int64	The name of the product instance, for which this (virtual) quantity flow is designated.
quantity	Double	The quantity expressed in unit of measurement defined in *quantityUnit* of the product definition referenced by the product instance.

6.4 ORDER MANAGEMENT

This section describes how to setup an Order and manage its life cycle:

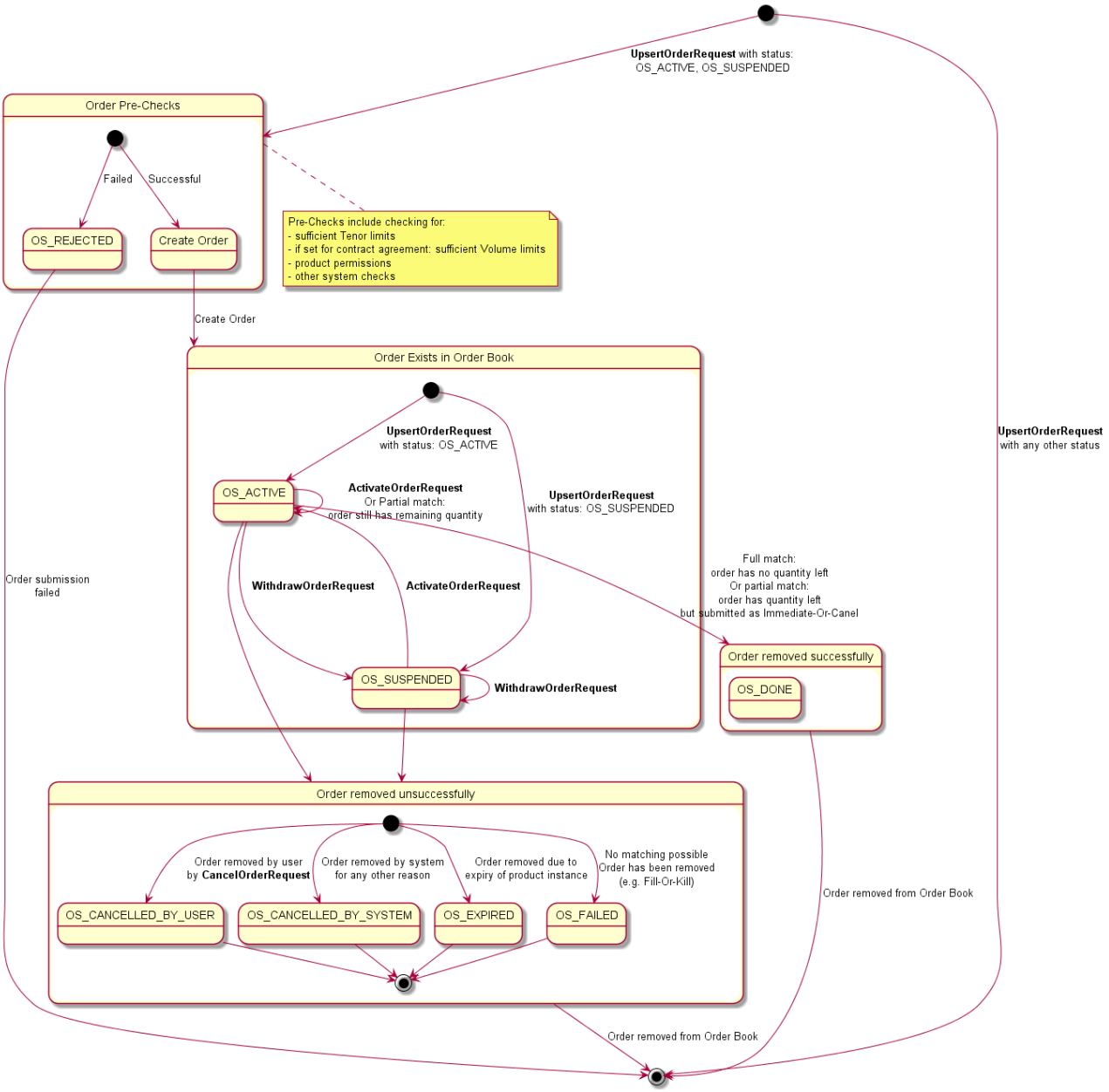

Figure 5: ECT order life cycle management

ECT Trading API has four different *OrderRequest* commands for managing the order life cycle:

- *UpsertOrderRequest*: this is used to either create a new order or to update an existing one.
- *WithdrawOrderRequest*: this is used, to suspend (aka withdraw, withheld) existing orders. If an order is suspended, it is still in the orderbook, but not available for matching any longer.
- *ActivateOrderRequest*: this is used, to activate a suspended (aka withdrawn, withheld) order. As soon as the order is active, it will be used in the matching process again.
- *CancelOrderRequest*: this is used to cancel an existing order. Once an order has been cancelled, it is removed completely from the orderbook.

The following table describes the *UpsertOrderRequest* fields:

Property	Type	Description
orderId	Int64	If a new order should be created: set it to 0 (zero) If an existing order should be updated: set it to the order's ID.
version	Int32	If a new order should be created: set it to 0 (zero) If an existing order should be updated: set it to the order's current version. You can also set the version to zero in order to update an order without the API validating the version field.
status	OrderStatus enum	If a new order should be created: set it to either *OS_ACTIVE* or *OS_SUSPENDED* to create it as active or suspended order. If an existing order should be updated: set it to the order's current status, as it must not change. To suspend or reactivate, use the *WithdrawOrderRequest* and *ActivateOrderRequest*.
bookId	Int64	The ID of the portfolio or contract agreement the order should be put into. Read-only, once the order has been created.
marketAreaId	Int64	The ID of the market area at which the order's (virtual) delivery should take place. Read-only, once the order has been created.
productInstanceId	Int64	The ID of the product instance into whose orderbook the order should be entered for matching. Read-only, once the order has been created.
direction	Direction enum	The direction of the order from client POV: • DIR_BUY: The client wants to buy the underlying • DIR_SELL: The client wants to sell the underlying Read-only, once the order has been created.
orderType	OrderType enum	Type of the order: • OT_FOK: Fill-Or-Kill - the order should be executed immediately upon adding it to the orderbook AND only with full quantity (no partial execution); if not possible, remove it immediately from the orderbook. • OT_GTC: Good-Till-Cancel - the order is set up as limit order, resting in the orderbook until it gets fully matched, cancelled, or if created by a Trading API client, the session terminates. Partial execution is allowed and will reduce the remaining quantity only. • OT_IOC: Immediate-Or-Cancel - the order should be executed immediately upon adding it to the orderbook. In contrast to Fill-Or-Kill, partial execution is allowed; The remaining (partial) order should be cancelled immediately after the matching process. • OT_AON: All-Or-Nothing - the order is set up as limit order, resting in the orderbook until it gets

		fully matched, cancelled, or if created by a Trading API client, the session terminates. In contrast to Good-Till-Cancel, partial execution is NOT allowed. Read-only, once the order has been created.
price	Double	The maximum price of the order. Depending on the orderType, the meaning is: • Limit order (resting): ○ DIR_BUY: The client wants to buy lower than or up to this price. ○ DIR_SELL: The client wants to sell higher than or equal to this price • Immediate execution (non-resting): ○ DIR_BUY: The client wants to buy lower than or up to this price. ○ DIR_SELL: The client wants to sell higher than or equal to this price
quantity	Double	The quantity expressed in unit of measurement defined in *quantityUnit* of the product definition referenced by the product instance.
clientCorrelationId	String	An optional ID set by the client. This value can be used to track the order on client systems with an own ID. It is only visible to users of the same party.
comment	String	An optional comment set by the client. It is only visible to users of the same party.
strategy	String	An optional value set by the client. It is only visible to users of the same party.

The following table describes the *WithdrawOrderRequest* fields:

Property	Type	Description
orderId	Int64	ID of an existing order to be suspended. If the order is already suspended, nothing will happen.
version	Int32	The current version of the order. If the version does not match the version on ECT side, the order was modified by someone else and the request will fail.

The following table describes the *ActivateOrderRequest* fields:

Property	Type	Description
orderId	Int64	ID of an existing order to be activated. If the order is already active, nothing will happen.
version	Int32	The current version of the order. If the version does not match the version on ECT side, the order was modified by someone else and the request will fail.

The following table describes the *CancelOrderRequest* fields:

Property	Type	Description
orderId	Int64	ID of an existing order to be cancelled and removed.
version	Int32	The current version of the order. If the version does not match the version on ECT side, the order was modified by someone else and the request will fail.

Each *OrderRequest* command, submitted from the client to ECT, will be acknowledged with an *OrderCommandResponse* from ECT.

To correlate a submitted *OrderRequest* with its *OrderCommandResponse,* it is recommended to set the *CorrelationID* header field on the AMQP message. If the *CorrelationID* was set, ECT automatically sets the value on the *OrderCommandResponse's* message header.

The following table describes the *OrderCommandResponse* type in detail:

Property	Type	Description
result	CommandResult enum	Contains a return code expressing the success or failure of the submitted command: • CR_SUCCEEDED • CR_FAILED_DUE_TO_INTERNAL_SERVER_ERROR • CR_FAILED_DUE_TO_MISSING_ORDER • CR_FAILED_DUE_TO_MISSING_ORDER_BOOK • CR_FAILED_DUE_TO_ORDER_VERSION_MISMATCH • CR_FAILED_DUE_TO_MISSING_USER • CR_FAILED_DUE_TO_MISSING_BOOK • CR_FAILED_DUE_TO_MISSING_PRODUCT_INSTANCE • CR_FAILED_DUE_TO_MISSING_MARKET_AREA • CR_FAILED_DUE_TO_EXCEEDING_QUANTITY_LIMITS • CR_FAILED_DUE_TO_INVALID_QUANTITY • CR_FAILED_DUE_TO_EXCEEDING_TENOR_LIMITS • CR_FAILED_DUE_TO_MISSING_TENOR_LIMIT • CR_FAILED_DUE_TO_INVALID_PRICE • CR_FAILED_DUE_TO_MISSING_PERMISSION_TO_TRADE_ON_BOOK • CR_FAILED_DUE_TO_MISSING_PERMISSION_TO_TRADE_PRODUCT_ORDER_TYPE_COMBINATION • CR_FAILED_DUE_TO_EXCEEDING_MAX_COMMENT_LENGTH • CR_FAILED_DUE_TO_ACCOUNT_TYPE • CR_FAILED_DUE_TO_DEACTIVATED_USER • CR_FAILED_DUE_TO_DEACTIVATED_PARTY • CR_FAILED_DUE_TO_DEACTIVATED_BOOK

As submitting orders need data from both Reference Data as well as Product Instances, the normal sequence for creating and managing orders should look like following:

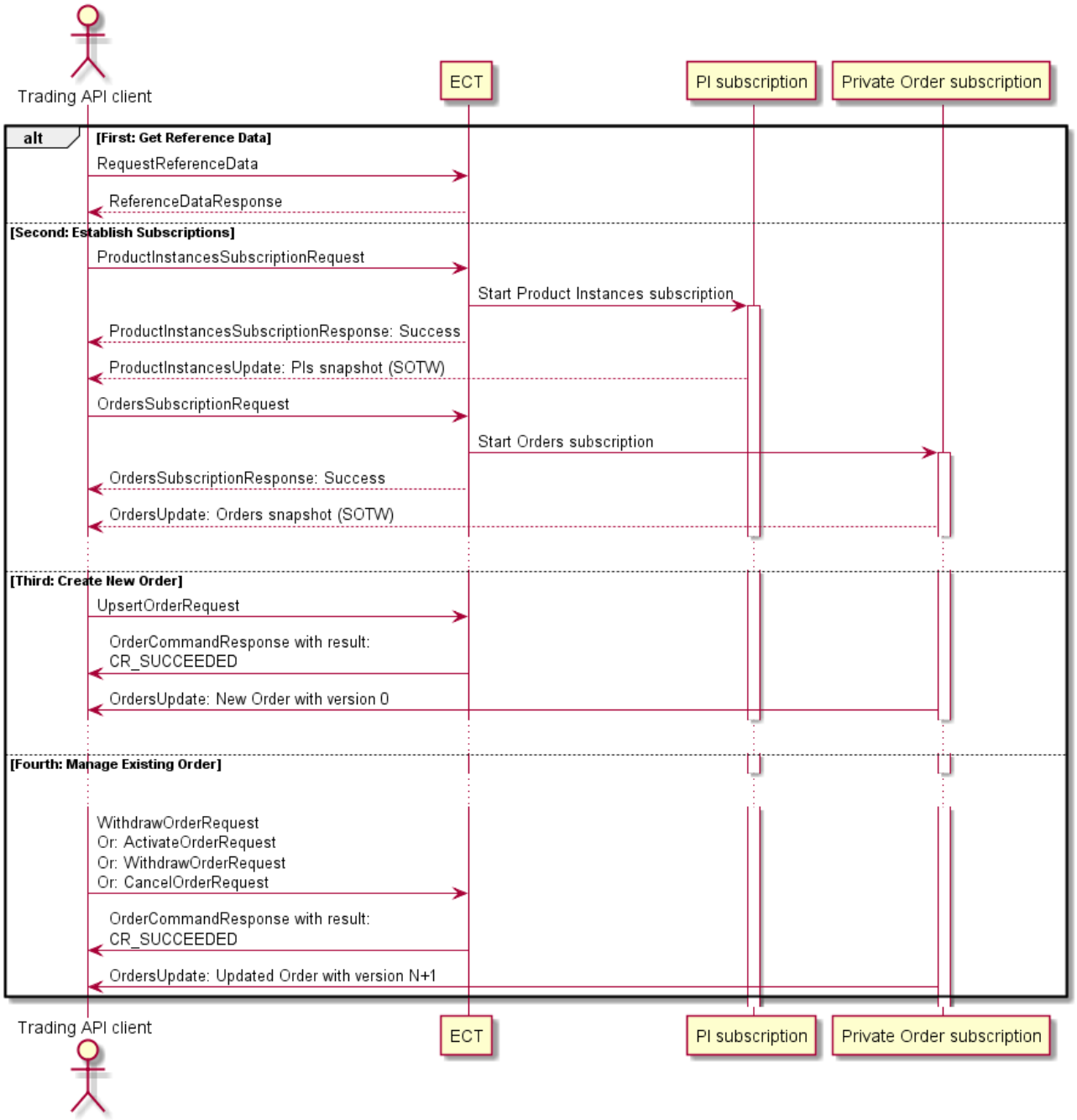

Figure 6: ECT order management

Hint: The *OrdersUpdate* event is **NOT send** out for orders setup by a Market Maker via Trading API! With that, there is a risk, that MarketMaker orders, changed through partial execution or via UI have a version which is not known to the MarketMaker Trading API client. In that particular case, the Trading API client cannot update (change) the order anymore, but has to cancel it and recreate.

6.5 TRADES

This section describes trade queries available in ECT. Each trades query is executed by sending a *GetTradesRequest*:

Property	Type	Description
fromInclusive	DateTimeOffset	Only trades having a *Trade.timestamp* equal to or greater than this value…
toExclusive	DateTimeOffset	AND having a *Trade.timestamp* less than this value will be returned.

As result, ECT will send back a *GetTradesResponse* containing all trades matching the query criteria or an error code:

Property	Type	Description
trades	Trade collection	Contains all *Trade* (see section 6.3.1 Trades) items matching the criteria of the related *GetTradesRequest* message.
isError	Bool	Boolean value, indicating whether an error occurred (*true*) or not (*false*).
errorMessage	String	If field *isError* is set to *true*, this field will contain a detailed error description.

6.6 RFQ MANAGEMENT

This section describes how to setup an RFQ and manage its life cycle:

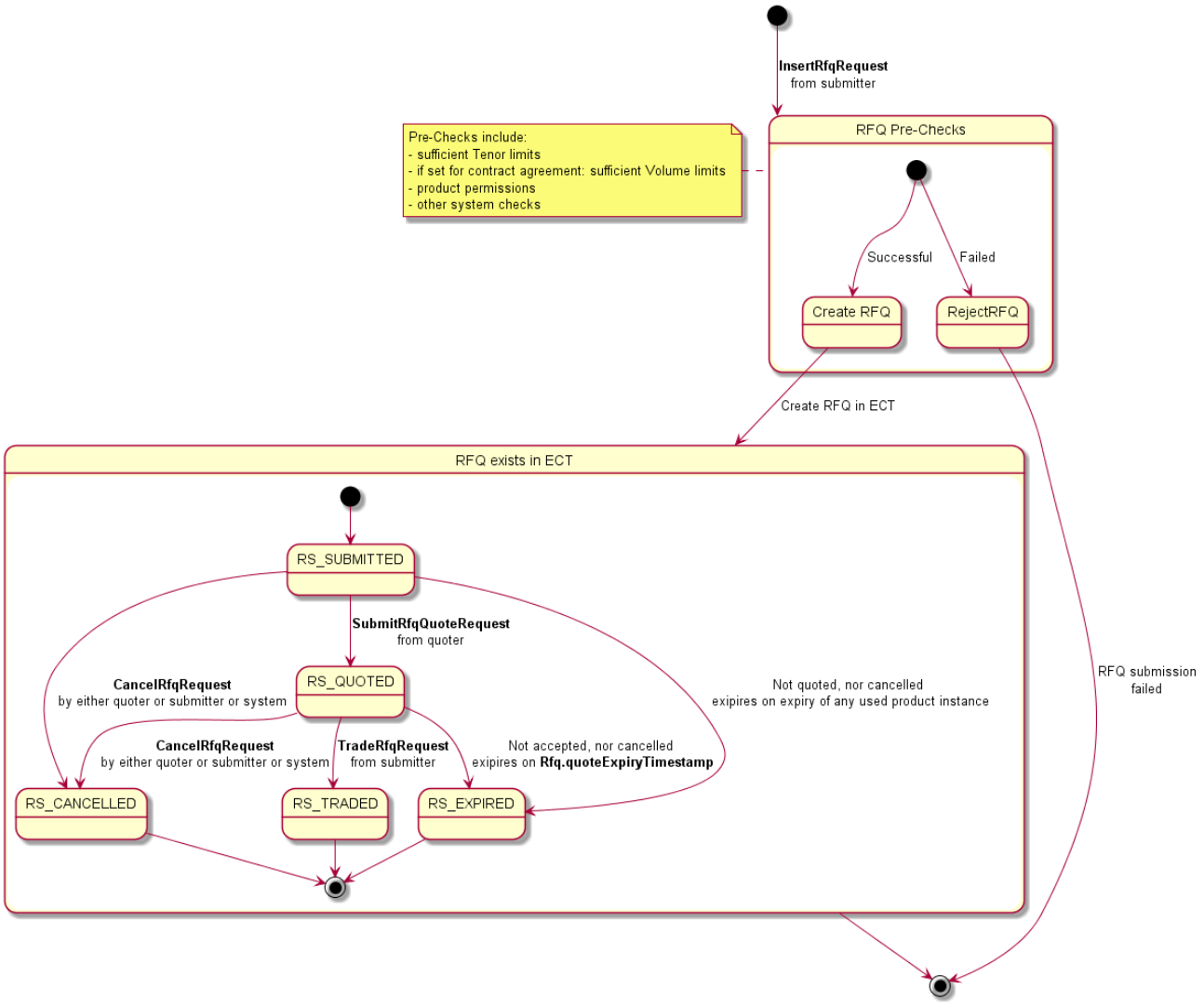

Figure 7: ECT RFQ life cycle management

ECT has two different roles for RFQ management:

- Submitter: the party which wants to receive the quote (price).
- Quoter: the party providing the quote (price).

ECT Trading API has four different *RfqRequest* commands for managing the RFQ life cycle:

- *InsertRfqRequest*: this is used by the RFQ submitter to create a new RFQ. In contrast to orders, an update of a once successfully submitted RFQ is not possible. If you need to change any property, the RFQ must be cancelled and re-submitted.
- *SubmitRfqQuoteRequest*: this is used by the RFQ quoter, to quote an existing RFQ. Once a quote is being provided, the RFQ can be accepted.
- *TradeRfqRequest*: this is used, to accept an already quoted RFQ. An RFQ, not being quoted, cannot be accepted. Once an RFQ has been accepted, a trade will be concluded over the RFQ between submitter and quoter.

- *CancelRfqRequest*: this is used to cancel an existing RFQ by either submitter or quoter (if applicable). Once an RFQ has been cancelled, it is removed completely from the system.

The following table describes the *InsertRfqRequest* fields:

Property	Type	Description
direction	Direction enum	The direction of the RFQ from submitter's POV: • DIR_BUY: The client wants to buy the underlying • DIR_SELL: The client wants to sell the underlying
bookId	Int64	The ID of the portfolio or contract agreement the RFQ should be put into. This is also the portfolio or contract agreement, into which the trade, resulting from the RFQ will be booked.
marketAreaId	Int64	The ID of the market area at which the RFQ's (virtual) main delivery should take place.
currencyId	Int64	The ID of the Currency in which the price of the RFQ (quote) should be delivered. This is only used, if the *ProductDefinition*, referenced by the product instances, has additional currencies set. (see also *ProductDefinition*.*additionalCurrencyIds*). If this value is set to 0 (zero), the default currency of the *ProductDefinition* will be taken.
quantityFlow	QuantityFlowItem collection	A collection of *QuantityFlowItem*s describing the (virtual) main delivery. A Standard RFQ only contains exactly one item, a profile RFQ can contain multiple items. (see also section **RFQs**)
isScheduled	Bool	If this value is set to true, the quote will be provided at the timestamp defined in field *scheduleTime*. If set to false, the quote will be delivered as soon as possible – note that there could be a delay under certain circumstances.
scheduledTime	DateTimeOffset	Only used if field *isScheduled* is set to true: the timestamp when the quote should be provided earliest.
clientCorrelationId	String	An optional ID set by the client. This value can be used to track the RFQ or trade concluded from it, on client systems with an own ID. It is only visible to users of the same party.
comment	String	An optional comment set by the client. It is only visible to users of the same party.
strategy	String	An optional value set by the client. It is only visible to users of the same party.

Hint: An RFQ submission is only allowed, if all used *ProductInstance*s in the *quantityFlow* items reference the very same *ProductDefintion*!

The following table describes the *SubmitRfqQuoteRequest* fields:

Property	Type	Description
rfqId	Int64	The ID of the existing RFQ to be quoted.

version	Int32	The current version of the RFQ. If the version does not match the version on ECT side, the RFQ was modified by someone else and the request will fail.
bookId	Int64	The ID of the portfolio or contract agreement the RFQ quote should be put into. This is also the portfolio or contract agreement, into which the trade, resulting from the RFQ will be booked into (on quoter side).
quote	Double	The quoted price for the RFQ. The price is quoted always in the default currency of the referenced *ProductDefinition*.
fxRate	Double	If the submitter submitted the RFQ with a currency other than the *ProductDefinition*'s default currency, then this value is the FX rate with which the quote should be multiplied in order to get the quoted price. E.g.: The referenced *ProductDefinition*'s default currency is EUR and the RFQ was requested in CZK. The RFQ quoter calculates a price of 12 CZK and submits an FX rate of 0.039. Therefore, the backend can multiply the quote by the FX rate in order to derive a final quote price of 0.468 EUR.
quoteComment	String	A public comment visible to all users of both RFQ parties: submitter and quoter. The value can only be set by the quoter, though.
quoteExpiryTimeStamp	DateTimeOffset	The time stamp at which latest the RFQ will expire if not accepted.
clientCorrelationId	String	An optional ID set by the client. This value can be used to track the RFQ or trade concluded from it, on client systems with an own ID. It is only visible to users of the same party.
comment	String	An optional comment set by the client. It is only visible to users of the same party.
strategy	String	An optional value set by the client. It is only visible to users of the same party.

The following table describes the *TradeRfqRequest* fields:

Property	Type	Description
rfqId	Int64	The ID of the existing RFQ to be traded/accepted.
version	Int32	The current version of the RFQ. If the version does not match the version on ECT side, the RFQ was modified by someone else and the request will fail.

The following table describes the *CancelRfqRequest* fields:

Property	Type	Description
rfqId	Int64	The ID of the existing RFQ to be cancelled.
version	Int32	The current version of the RFQ.

		If the version does not match the version on ECT side, the RFQ was modified by someone else and the request will fail.

Each *RfqRequest* command, submitted from the client to ECT, will be acknowledged with an *RfqCommandResponse* from ECT.

To correlate a submitted *RfqRequest* with its *RfqCommandResponse,* it is recommended to set the *CorrelationID* header field on the AMQP message. If the *CorrelationID* was set, ECT automatically sets the value on the *RfqCommandResponse's* message header.

The following table describes the *RfqResponseCommand* type in detail:

Property	Type	Description
result	RfqCommandResult enum	Contains a return code expressing the success or failure of the submitted RFQ command: • RCR_SUCCEEDED • RCR_FAILED_DUE_TO_INVALID_STATE • RCR_FAILED_DUE_TO_INTERNAL_SERVER_ERROR • RCR_FAILED_DUE_TO_NO_QUANTITY_FLOWS_SUBMITTED • RCR_FAILED_DUE_TO_INVALID_BOOK • RCR_FAILED_DUE_TO_UNKNOWN_MARKET_AREA • RCR_FAILED_DUE_TO_MARKET_AREA_MISMATCH • RCR_FAILED_DUE_TO_MISSING_PRODUCT_INSTANCE • RCR_FAILED_DUE_TO_PRODUCT_TEMPLATE_MISMATCH • RCR_FAILED_DUE_TO_DUPLICATE_PRODUCT_INSTANCE • RCR_FAILED_DUE_TO_INVALID_SCHEDULED_TIME • RCR_FAILED_DUE_TO_INVALID_QUANTITY • RCR_FAILED_DUE_TO_NON_TRADING_ACCOUNT_TYPE • RCR_FAILED_DUE_TO_DEACTIVATED_USER • RCR_FAILED_DUE_TO_DEACTIVATED_COMPANY • RCR_FAILED_DUE_TO_DEACTIVATED_DESK • RCR_FAILED_DUE_TO_DEACTIVATED_CONTRACT_AGREEMENTS • RCR_FAILED_DUE_TO_DEACTIVATED_PORTFOLIOS • RCR_FAILED_DUE_TO_MISSING_TENOR_LIMIT • RCR_FAILED_DUE_TO_EXCEEDING_TENOR_LIMITS • RCR_FAILED_DUE_TO_EXCEEDING_QUANTITY_LIMITS • RCR_FAILED_DUE_TO_INSUFFICIENT_PRODUCT_PERMISSIONS • RCR_FAILED_DUE_TO_INVALID_QUOTE_EXPIRY_DATE • RCR_FAILED_DUE_TO_MISSING_RFQ • RCR_FAILED_DUE_TO_NOT_RFQ_OWNER • RCR_FAILED_DUE_TO_INVALID_QUOTE • RCR_FAILED_DUE_TO_VERSION_MISMATCH • RCR_FAILED_DUE_TO_INVALID_CURRENCYCR_FAILED_DUE_TO_DEACTIVATED_BOOK
isError	Bool	Boolean value, indicating whether an error occurred (*true*) or not (*false*).
errorMessage	String	If field *isError* is set to *true*, this field will contain a detailed error description.

As submitting RFQs needs data from both Reference Data as well as Product Instances, the normal sequence for creating and managing RFQs from submitter perspective should look like following:

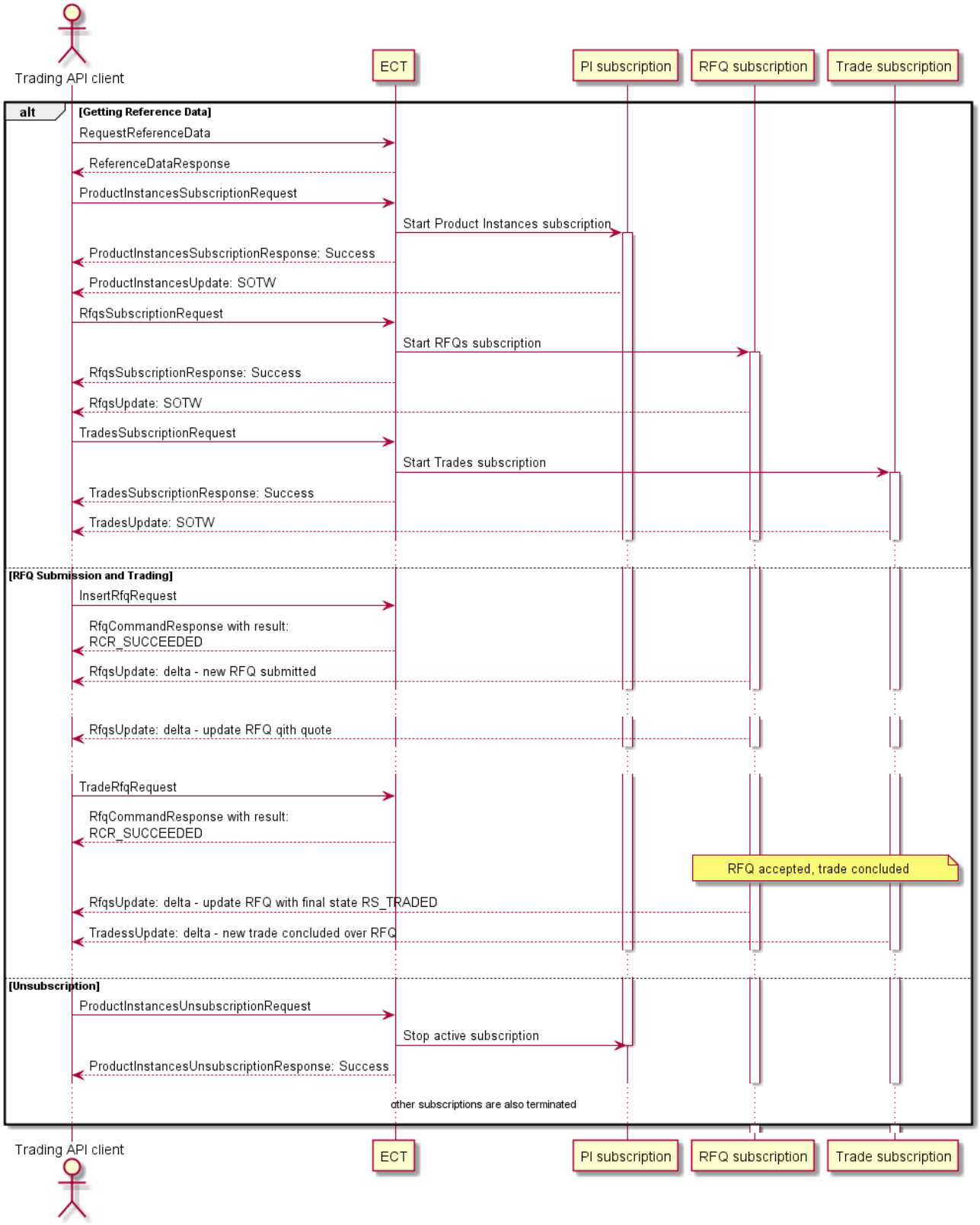

Figure 8: ECT RFQ submitter workflow

The normal sequence for managing RFQs from quoter perspective should look like following:

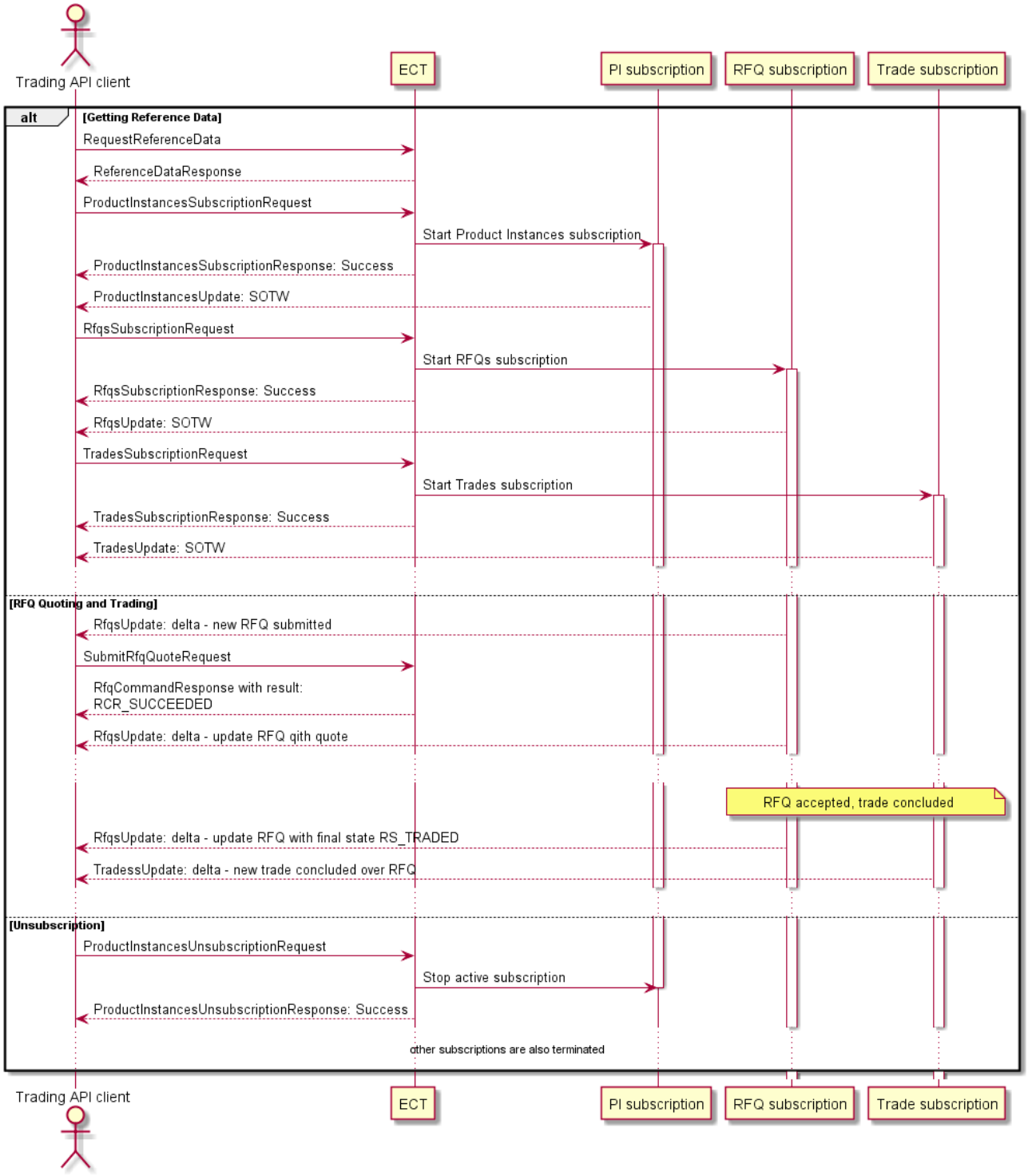

Figure 9: ECT RFQ quoter workflow