TS3 Trading System

Lime C++ Trading Api Overview

3/8/2022
Version 1.78.0

Contents

1 Revision History

Revision	Date	Author(s)	Description
1.4.0	7/9/2010	VV	Added *setCallbackCpuAffinity()* function.
1.4.1	7/9/2010	VV	Move some types to limeTypes.hh.
1.4.2	7/9/2010	VV	The *expireTime* new order property is now of type uint64_t instead of std::time_t.
1.4.3	7/9/2010	VV	The *FillInfo* struct now contains a 64-bit *transactTime* field which is the execution time reported by the market in milliseconds.
1.5.0	8/18/2010	VV	A new *isoGroupId* field has been added to OrderProperties which represents an identifier for a group of ISO orders pertaining to the same sweep. Mandatory for all ISO orders.
1.5.1	8/18/2010	VV	A new *saleAffirm* flag has been added to *OrderProperties*. It must be set for each long or short sale order by clients who trade away from Lime to represent that the client owns the security being sold, or will own it by settlement time.
1.6.0	10/1/2010	VV	The format of the *liquidity* field in the *FillInfo* structure is changed from char to integer, and the name of the field is changed from *markedLiquidityValue* to *liquidity*. Liquidity codes currently reported may be incorrect if the liquidity code value is greater than 9.
1.6.1	10/1/2010	VV	A new *side* field is added to the *FillInfo* structure in the *onOrderFill()* callback.
1.6.2	10/1/2010	VV	A shared version of the library is offered in the package.
1.7.0	11/3/2010	VV	Added *attributes* argument to *onOrderAccept()* callback. The new argument is an instance of a new *OrderAckAttributes* class.
1.8.0	11/16/2010	VV	The *onRouteStatusChange()* callback is deprecated. Route status queries and callbacks are now available via the ACS API.
1.8.1	11/16/2010	VV	A new *timeInForce* value, *timeInForceIntraDayCross*, is now supported.
1.8.2	11/16/2010	VV	Added new order property *discretionOffset*. It is the offset from displayed price. Positive for buy, negative for sell.

1.8.3	11/16/2010	VV	Added the ability to specify BATS and INET-FIX routing instructions.
1.8.4	11/16/2010	VV	Added new order property *batsDarkScan*. Used to specify orders scanned for dark liquidity.
1.8.5	11/16/2010	VV	Added new order property *batsNoRescrapeAtLimit*. Available only to fully routable IOC orders. After walking the price down to the limit there will be no final scrape at BATS.
1.8.6	11/16/2010	VV	Added new order property *arcaTracking*. An ARCA tracking order executes against outbound orders with a leaves quantity less than or equal to the size of the tracking limit order.
1.8.7	11/16/2010	VV	Added new order property *arcaPassiveLiquidity*. A passive liquidity order is never displayed externally. It is ranked behind display and reserver orders and ahead of all other orders.
1.8.8	11/16/2010	VV	Added new order property *routeToNyse*. Valid for NYSE-listed securities. If this flag is set, the order will sweep Nasdaq or EDGA/X, possibly removing liquidity. If it does not fill, it will be routed to NYSE for execution.
1.8.9	11/16/2010	VV	Added new order property *nyseClosingOffset*. Order participates in the closing trade on the opposite side of any imbalance.
1.9.0	3/29/2011	VV	Added new order property *targetLocationId*. It is used to identify NYSE trading partner.
1.9.1	3/29/2011	VV	Added *attributes* to the *onOrderReplace()* callback. As a result, price adjustment information is now available for Replace Acks.
1.10.0	5/27/2011	VV	Added *marketClOrdId* to the *OrderAckAttributes* class.
1.11.0	7/5/2011	VV	Added the ability to specify callback thread CPU affinity at thread creation through the API constructor.
1.11.1	7/5/2011	VV	Additional routing instructions are now supported for BATS.
1.11.2	7/5/2011	VV	Added support for DirectEdge routing instructions.
1.11.3	7/5/2011	VV	Added new order property *lockedCrossedAction*.

1.11.4	7/5/2011	VV	Added new order property *nyseParityReservation*. It is a flag used to specify that an order should be sent to NYSE floor broker as an eQuote reservation order.
1.11.5	7/5/2011	VV	Added new order property *nyseParityEquote*. Order is sent to NYSE as an order generated by a third party algorithm on behalf of a NYSE Floor Broker.
1.11.6	7/5/2011	VV	Added new order property *nyseParityStrategy*. To properly specify a NYSE Parity Reservation or eQuote order, a NYSE Parity Strategy must be specified.
1.12.0	7/25/2011	VV	Added *marketOrderId* to the *OrderAckAttributes* class.
1.12.1	7/25/2011	VV	Fixed the issue with the *marketClOrdId* Order Ack attribute resulting in extra characters appended to the end of the string.
1.13.0	7/30/2011	VV	Added *displayPriceAdjusted* and *displayPrice* to *OrderAckAttributes* class.
1.13.1	7/30/2011	VV	Additional routing instructions are now supported for BATS.
1.14.0	8/31/2011	VV	Added *execType*, *execId* and *contraBroker* to the *FillInfo* structure.
1.15.0	9/7/2011	VV	Buffering for messages received over TCP is increased to accommodate larger message sizes resulting from additional fields in the *FillInfo* structure introduced in version 1.14.
1.16.0	9/7/2011	VV	Eliminated ScaledPriceDiff type and made ScaledPrice signed.
1.16.1	9/7/2011	VV	Added support for PNP Blind orders on ArcaDirect.
1.16.2	9/7/2011	VV	Added support for *pegTypeAlternateMidpoint*.
1.16.3	9/7/2011	VV	US Options Trading has been enabled.
1.16.4	9/7/2011	VV	A constant *samePrice* is introduced to be used in cancelReplace calls. It must be used for US Options Replace if price is unchanged.

1.16.5	9/7/2011	VV	Three new optional callbacks are provided informing the API user about new orders and cancel replace requests added manually, e.g. via Lime Portal. The new callbacks are *onManualOrder()*, *onManualUSOptionsOrder()* and *onManualOrderReplace()*.
1.17.0	11/9/2011	VV	Added new order property *washTradePrevention*. It is only available for US Equities trading.
1.18.0	11/29/2011	VV	Added new order property *regularSessionOnly* for US Equities trading. When set, it indicates that the day TIF order is NOT eligible for premarket trading.
1.18.0	11/29/2011	VV	Added a variant of the *onLoginAccepted()* callback providing *eventId*.
1.19.0	12/19/2011	VV	Added support for new pegType values *pegTypePriceImprovedPrimary*, *pegTypePriceImprovedMarket* and *pegTypePriceImprovedMidpoint*.
1.20.0	12/19/2011	VV	Increased receive buffering and exposed limited control over transport properties.
1.21.0	1/27/2012	VV	Added new order property *shortSaleAffirmLongQuantity* for US Equities and Futures only. It is a further refinement of sale affirmation for DVP accounts. For short sales, sale affirmation guarantees that the full quantity exceeds customer's long position in the security being sold short and any such long position now has to be explicitly declared using the new property.
1.21.1	1/27/2012	VV	The above requirement exists for cancel replaces for sale-affirmed short sales as well. Thus a new *CancelReplaceProperties* class has been created with a *shortSaleAffirmLongQuantity* field for cancel replace requests.
1.22.0	2/28/2012	VV	Added new order property *marketDisplayPrice*. It is used to submit discretionary orders on INET-FIX market.

1.23.0	3/28/2012	VV	Added support for a new timeInForce value, *timeInForceAtOpenThenDay*. This value allows on-open orders to remain live if they are not filled at the open.
1.24.0	5/9/2012	VV	Added new order properties *nearPegOffset* and *farPegOffset* to the *OrderProperties* class. Both values must be specified in a valid order. Valid values are between 0 and 100 with increment of 10.
1.24.1	5/9/2012	VV	Added support for new Locked/Crossed actions for BATS/BYX.
1.25.0	7/28/2012	VV	Added new order property *retailPriceOffset* to the *OrderProperties* class. It is used to specify the offset of minimum price improvement value from the current Bid or Offer.
1.26.0	8/20/2012	VV	Added support for new pegType value *pegTypeMidpointDiscretionary*.
1.27.0	9/13/2012	VV	Added new order property *customerType* to the *USOptionsOrderProperties* class.
1.28.0	10/23/2012	VV	Added new order option *maxFloor*. Specifies the highest (for a buy) or lowest (for a sell) price to which an e-Quote or a d-Quote may peg. Zero is not a valid value. When *maxFloor* is used, price should be set to a non-marketable value.
1.28.1	10/23/2012	VV	Added new order option NYSE routing instructions.
1.28.2	10/23/2012	VV	Removed *nyseParityReservation* and *nyseParityEquote* order properties.
1.29.0	11/28/2012	VV	Added the ability to specify ARCA routing instructions.
1.29.1	11/28/2012	VV	Additional routing instructions are now supported for INET-FIX.
1.30.0	12/26/2012	VV	Ability to specify NYSE Closing Offset via a dedicated flag is deprecated. The option should be set using NYSE Routing Instructions.
1.31.0	2/23/2013	VV	Added *lastMkt* to the *FillInfo* structure. It is a string denoting the venue where execution took place. May be left unpopulated.

1.31.1	2/23/2013	VV	Added the ability to specify KMATCH routing instructions.
1.31.2	2/23/2013	VV	Pre and Post trading orders for Kmatch 2.0 are supported. To denote an order pre- or post- trading specify one of the new TIF values *timeInForcePreOpenSession* or *timeInForcePostCloseSession*.
1.31.3	2/23/2013	VV	New TIF values are supported *timeInForceLateLimitOnClose* and *timeInForceLateLimitOnOpen*.
1.32.0	3/14/2013	SRT	Added support for new pegType value *pegTypeMidpointPennySpread*.
1.33.0	4/30/2013	KGR	Added support for route delivery method, called *routingStrategy*.
1.34.0	9/1/2013	RL	Factor out platform dependencies in the Lime Trading API.
1.35.0	12/18/2013	KGR	Improved debugging capabilities. Move to new documentation format.
1.36.0	2/1/2014	KGR	Update handling of setting allowRouting false for market order.
1.36.1	2/1/2014	KGR	Improved handling of high traffic.
1.37.0	4/1/2014	KGR	Further improvements for infiniband high traffic handling.
1.38.0	7/1/2014	KGR	Add three client data order options.
1.38.1	7/1/2014	KGR	Clean up order option documentation.
1.40.0	7/11/2014	KGR	Quiet a false positive valgrind error.
1.42.0	9/24/2014	KGR	Add the ability to submit Algo orders.
1.44.0	12/10/2014	KGR	Add positionEffect to OrderAck.
1.44.1	12/11/2014	KGR	Add Side to OrderAck.
1.46.0	2/10/2015	KGR	Add "Generic" market routing selector.
1.48.0	3/25/2015	RL	Support prices above 214k.
1.50.0	5/21/2015	KGR	Support MaxFloor, MinQty, DiscretionOffset, and new TimeInForces for algo orders.
1.52.0	5/21/2015	KGR	Fix handling of corner case in placeAlgoOrder.
1.54.0	7/2/2015	SKP	Support MinimumTriggerVol for Equity new order and cancel/replace.
1.56.0	8/14/2015	SKP	Add Sale Affirmation flags for Algo Orders.
1.58.1	10/14/2015	SKP	Echo liquidity information in fill.
1.58.2	11/09/2015	JWK	Support partial cancel.

1.60.0	06/08/2016	SKP	Add invisible, pegType, pegDifference, minTriggerVol, and minTriggerPerc to LSR orders.
1.60.1	06/14/2016	SKP	Add quickstart option to LSR new orders and LSR CR.
1.60.0	06/24/2016	SKP	Add maxMarketOrderSlippageAmount option to LSR new orders and LSR CR.
1.62.0	07/08/2016	SKP	Add excludedVenues option to LSR new orders.
1.64.0	09/28/2016	JWK	Support Algo Sweep Type option.
1.64.0	09/29/2016	SKP	Add Tifs and regular session flag for trading session aware LSR Equities.
1.66.0	12/31/2017	EB	Add nanosecond timestamp field to FillInfo.
1.70.0	2/15/2019	EB	Add MELO order option to PegType enum.
1.76.0	7/7/2021	NZ	Add support for multiple routing instructions.
1.78.0	3/8/2022	NZ	Add New listener overload cancel with std::string reason, Add new TIF=G (NASDAQ ETC orders).

2 Introduction

The Lime Trading API provides an ultra low latency interface to the Lime trading platform. The API is written in C++ and is packaged as a binary and a set of header files. It resides in the *LimeBrokerage* namespace. The library is 64-bit. Both static (.a) and shared (.so) versions of the library are provided. We recommend using the static version where possible for more deterministic performance results.

An instance of the Lime Trading API enables trading on a single account. The API currently supports US Equities, Futures and Options[1] trading. All API trading calls are asynchronous, the API does not wait for a Lime trading platform response before returning control to the client application. Callbacks are returned in a dedicated callback thread, this is the only thread spawned by the API. Placing new calls into the API from API callbacks is prohibited. This includes the API destructor, which must be destructed in the same thread where it was constructed. Likewise, the API will place no calls from the context of a client application call.

The API is stateless. All state is maintained in the Lime trading platform running in a separate process. The state pertaining to reported events (eventID) is passed to the client application and is not maintained by the API.

3 Initialization and Callbacks

The Lime Trading Api is initialized by calling its constructor:

```
TradingApi(Listener& listener,
    const std::string& account,
    const std::string& user,
    const std::string& password,
    uint64_t eventId,
    bool cancelAllOnDisconnect,
    const std::string& hostname = nullHostname,
    TransportType transportType = transportTypeTcp,
    Const CpuSet& callbackCpuset = CpuSet::null);
```

The constructor requires the client application to specify a set of callback functions via a reference to the *Listener* class.

```
TradingApi(Listener &listener);

class Listner {
public:

    //
```

[1] Only simple US Options orders are currently supported.

```
    // Required trading callbacks
    //
    ...

    //
    // Optional trading callbacks
    //
    ...

    //
    // Session callbacks
    //
    virtual void onLoginAccepted();
    virtual void onLoginAccepted(uint64_t eventId);

    virtual void onLoginFailed(const std::string& reason) = 0;

    virtual void onConnectionFailed(const std::string& reason) = 0;

    virtual void onConnectionBusy();
    virtual void onConnectionAvailable();
}
```

Some of the callbacks are pure virtual functions (e.g. *onLoginFailed()*) while others are just virtual (e.g. callbacks placed upon recieving a session layer message). The former require the client application to provide its implementation of the callback; the latter are optional. Also note that some callbacks (e.g. on cancel reject) are venue dependant and may never be called when trading on venues that do not support them.

The API constructor initiates a connection to the Lime trading platform identified by *hostname* which can be either a hostname resolvable by DNS or local hosts file or an IP address. If the Lime trading platform is running on the same box as the API, hostname does not need to be specified and the default argument is used:

```
static const std::string nullHostname;
```

The next argument in the constructor identifies the transport type used for connectivity with the Lime trading platform. By default, the connection will be initiated over TCP/IP (hence the default argument *transportTypeTcp*). To use raw Infiniband transport, the last argument must be set to *transportTypeInfiniband*:

```
enum TransportType {
    transportTypeTcp,
    transportTypeInfiniband
};
```

The last constructor argument, *callbackCpuset*, allows setting CPU affinity of the callback thread at the time of its creation. It can be changed or reset later using the *setCallbackCpuAffinity()* method.

Upon initiating the connection, the API logs in to the specified trading account on the Lime trading platform using *account*, *username* and *password*. Trading on the account may commence after the *onLoginAccepted* callback is received.

If the connection cannot be established or a trading account login failure occurs, the API will return an *onConnectionFailed* or an *onLoginRejected* callback, respectively. The callbacks are returned asynchronously. Intraday connection status failures are communicated to the client application via the same *onConnectionFailed* callback.

A *cancelAllOnDisconnect* flag indicates whether all open orders on the account should be canceled when the connection goes down.

An API instance is associated with a dedicated messaging session. A single session is maintained for each account during a trading day. Every callback contains an eventID that uniquely identifies it throughout the day.[2] EventID values start from 1 and increase monotonically throughout the day. Note that gaps in eventID sequence are expected since some session-level messages (e.g. heatbeats) that consume eventIDs are not propagated to the application.

In case of a connection failure, the API instance is deactivated. A new instance of the API must be initiated to continue trading. The client application may use eventIDs to continue the trading session where it left off before the interruption by specifying the *eventId* argument in the API constructor to be the next value after the highest eventID received on the session so far. An eventID value of 1 will reult in replay of the session from the start. An eventID value of 0 is a special value that continues the session where it left off without replay.

If the order submission rate exceeds the Lime trading platform throughput for too long, buffers will fill up. At that point, the API starts rejecting calls instead of queuing them indefinitely and introducing unbounded delay. If a trading call cannot be delivered to the Lime trading platform, it is rejected synchronously with an appropriate status (see next section); optionally, an *onConnectionBusy()* callback is also placed in the callback thread. It is expected that the client application will reduce the rate of order submission once it detects congestion. The API will periodically poll the connection in congested state. When the condition clears and the connection can accept messages again, an *onConnectionAvailable()* callback is placed.

A callback thread may be pinned to a CPU core or excluded from a set of CPU cores using

[2]Exceptions are *onLoginAccepted* and trading reject callbacks for rejects generated in the API itself. These callbacks do not return eventIDs.

a special API function, *setCallbackCpuAffinity()*. This function call is synchronous.

4 Trading Calls

A trading call is accepted by the API if and only if it can be transmitted to the Lime trading platform. If the connection with the Lime trading platform is down or congested, calls are synchronously rejected. Below is the enumeration class *CallStatus* representing the possible values returned by a trading call:

```
enum CallStatus {
    statusSuccess,
    statusConnectionBusy,
    statusConnectionError
};
```

Note that initially accepted orders can still be rejected by the Lime trading platform or an exchange during further processing. In such cases, the rejects are submitted via a specified callback, asynchronously, to the client application via the callback thread. If the call returns any value other than statusSuccess, an order callback is not generated.

Route selection can be made by the client application or outsourced to the Lime trading platform. The latter functionality is achieved by specifying a special "demux" route uniquely defined for each venue in the *placeOrder()* or *placeUSOptionsOrder()* call.

4.1 placeOrder

```
CallStatus placeOrder(
    OrderId orderId,
    const std::string& route,
    const std::string& symbol,
    Side side,
    int quantity,
    ScaledPrice price = marketPrice,
    const OrderProperties& properties = nullProperties);
```

The client application will use this function to place a new US Equities or Futures order on a given route or venue.[3] The primary characteristics of an order - symbol, quantity, price, and side - are explicitly specified. The *orderId* in the trading calls must also be supplied by the trading application. Note that the *orderId* must be unique for a given account throughout the day.

The API assumes a limit order when a positive price value is submitted. If a price is not specified or is explicitly set to *marketPrice*, the order is interpreted as a market order. Setting price to zero has the same effect.

[3] Please refer to Appendix C of the Lime FIX Manual for valid values of the *route* argument.

Prices submitted via the API are expected to have a *ScaledPrice* format. This means that the prices are multiplied by a fixed factor to accommodate precision of up to N digits after the decimal point. The default precision is N=4 meaning that the dollar values are obtained by dividing the API values by 10,000.

```
static const int priceScalingFactor = 10000;
```

The following are the definitions of the types used in the call:

```
typedef uint64_t OrderId;
typedef int64_t ScaledPrice;

enum Side {
    sideBuy,
    sideSell,
    sideSellShort,
    sideSellShortExempt,
    sideBuyToCover
};
```

Optional order properties like *postOnly*, *timeInForce*, *ISO*, etc are summarized in the *properties* object. To specify a market routing instruction, specify the *marketRoutingSelector* field with the appropriate value (e.g. *marketRoutingBats* for a BATS routing instruction) and set the *routingInstructions* field to the desired value in string format. Below is the definition of the *OrderProperties* class:

```
class OrderProperties {
public:
    enum TimeInForce {
        timeInForceDay,
        timeInForceOpg,
        timeInForceIoc,
        timeInForceExtendedDay,
        timeInForceGoodTillDate,
        timeInForceAtTheClose,
        timeInForceTimeInMarket,
        timeInForceIntradayCross,
        timeInForceAtOpenThenDay = 9,
        timeInForceLateLimitOnOpen,
        timeInForceLateLimitOnClose,
        timeInForcePreOpenSession,
        timeInForcePostCloseSession,
        timeInForceExtendedTradingClose
    };

    enum PegType {
        pegTypeNone,
        pegTypePrimary,
        pegTypeMarket,
```

```
        pegTypeMidpoint,
        pegTypeAlternateMidpoint,
        pegTypePriceImprovedPrimary,
        pegTypePriceImprovedMarket,
        pegTypePriceImprovedMidpoint,
        pegTypePriceMidpointDiscretionary,
        pegTypeMidpointPennySpread
    };

    enum MarketRoutingSelector {
        marketRoutingNone,
        marketRoutingBats,
        marketRoutingInetFix,
        marketRoutingDirectEdge,
        marketRoutingNyse,
        marketRoutingArca,
        marketRoutingKmatch
    };

    enum WashTradePrevention {
        washTradePreventionNone,
        washTradeCancelNewest,
        washTradeCancelOldest,
        washTradeCancelBoth
    };

    TimeInForce getTimeInForce() const;
    uint32_t getTimeInMarket() const;
    uint64_t getExpireTime() const;
    PegType getPegType() const;
    ScaledPrice getPegDifference() const;
    ScaledPrice getDiscretionOffset() const;
    int getMaxFloor() const;
    int getMinQty() const;
    int getMinimumTriggerVol() const;
    uint32_t getISOGroupId() const;
    MarketRoutingSelector getMarketRoutingSelector() const;
    const std::string& getRoutingInstructions() const;
    const std::pair<MarketRoutingSelector, std::string> getRoutingInfo(
        unsigned int index) const;
    bool isRoutingAllowed() const;
    AllowRouting getRoutingAllowed() const;
    bool isInvisible() const;
    bool isPostOnly() const;
    bool isIso() const;
    bool isNasdaqPostOnly() const;
    bool isImbalanceOnly() const;
    bool isSaleAffirm() const;
    bool isBatsDarkScan() const;
    bool isBatsNoRescrapeAtLimit() const;
```

```
    bool isArcaTracking() const;
    bool isArcaPassiveLiquidity() const;
    bool isRouteToNyseSet() const;
    const std::string& getNyseParityStrategy() const;
    const std::string& getTargetLocationId() const;
    char getLockedCrossedAction() const;
    WashTradePrevention getWashTradePrevention() const;
    bool isRegularSessionOnly() const;
    int getShortSaleAffirmLongQuantity() const;
    ScaledPrice getMarketDisplayPrice() const;
    int getNearPegOffset() const;
    int getFarPegOffset() const;
    ScaledPrice getRetailPriceOffset() const;
    bool getRetailPriceOffsetSet() const;
    ScaledPrice getCeilingFloorPrice() const;
    const std::string& getRoutingStrategy() const;
    const std::string& getClientData1() const;
    const std::string& getClientData2() const;
    const std::string& getClientData3() const;

    void setTimeInForce(TimeInForce value);
    void setTimeInMarket(uint32_t value);
    void setExpireTime(uint64_t value);
    void setPegType(PegType value);
    void setPegDifference(ScaledPrice value);
    void setDiscretionOffset(ScaledPrice value);
    void setMaxFloor(int value);
    void setMinQty(int value);
    void setMinimumTriggerVol(int value);
    void setMarketRoutingSelector(MarketRoutingSelector value);
    void setRoutingInstructions(const std::string& value);
    bool addRoutingInfo(const std::pair<MarketRoutingSelector, std::string>
        &routingInfo);
    bool setRoutinginfo(const std::pair<MarketRoutingSelector, std::string>
        &routingInfo, int index);
    void clearRoutingInfo();
    void setRoutingAllowed(bool value);
    void unsetRoutingAllowed();
    void setInvisible(bool value);
    void setPostOnly(bool value);
    void setIso(bool value);
    void setNasdaqPostOnly(bool value);
    void setImbalanceOnly(bool value);
    void setISOGroupId(uint32_t value);
    void setSaleAffirm(bool value);
    void setBatsDarkScan(bool value);
    void setBatsNoRescrapeAtLimit(bool value);
    void setArcaTracking(bool value);
    void setArcaPassiveLiquidity(bool value);
    void setRouteToNyse(bool value);
```

```
    void setNyseParityStrategy(const std::string& value);
    void setTargetLocationId(const std::string& value);
    void setLockedCrossedAction(char value);
    void setWashTradePrevention(WashTradePrevention value);
    void setRegularSessionOnly(bool value);
    void setShortSaleAffirmLongQuantity(int value);
    void setMarketDisplayPrice(ScaledPrice value);
    void setNearPegOffset(int value);
    void setFarPegOffset(int value);
    void setRetailPriceOffset(ScaledPrice value);
    void resetRetailPriceOffset();
    void setCeilingFloorPrice(ScaledPrice value);
    void setRoutingStrategy(const std::string& value);
    void setClientData1(const std::string& value);
    void setClientData2(const std::string& value);
    void setClientData3(const std::string& value);

private:
    TimeInForce timeInForce;
    uint32_t timeInMarket;
    uint64_t expireTime;
    uint32_t isoGroupId;
    PegType pegType;
    ScaledPrice pegDifference;
    ScaledPrice discretionOffset;
    int maxFloor;
    int minQty;
    int minimumTriggerVol;
    std::pair<MarketRoutingSelector, std::string> routingInstructionsPair[
        maxNumRoutingInstructions];    std::string routingInstructions;
    AllowRouting allowRouting;
    bool invisible;
    bool postOnly;
    bool iso;
    bool nasdaqPostOnly;
    bool imbalanceOnly;
    bool saleAffirm;
    bool batsDarkScan;
    bool batsNoRescrapeAtLimit;
    bool arcaTracking;
    bool arcaPassiveLiquidity;
    bool routeToNyse;
    std::string targetLocationId;
    char lockedCrossedAction;
    std::string nyseParityStrategy;
    WashTradePrevention washTradePrevention;
    bool regularSessionOnly;
    int shortSaleAffirmLongQuantity;
    ScaledPrice marketDisplayPrice;
    int nearPegOffset;
```

```
    int farPegOffset;
    ScaledPrice retailPriceOffset;
    bool retailPriceOffsetSet;
    ScaledPrice ceilingFloorPrice;
    std::string routingStrategy;
    std::string clientData1;
    std::string clientData2;
    std::string clientData3;
};
```

To facilitate the handling of the optional order properties object, a *nullProperties* static object is defined and used as a default argument:

```
extern const OrderProperties nullProperties;
```

4.2 placeUSOptionsOrder

```
CallStatus placeUSOptionsOrder(
    OrderId orderId,
    const std::string& route,
    const USOptionsSymbol& symbol,
    Side side,
    PositionEffect positionEffect,
    int quantity,
    ScaledPrice price = marketPrice,
    const USOptionsOrderProperties& properties = nullUSOptionsProperties);
```

The client application will use this function to place new US Options orders on a given route or venue. The primary characteristics of an order - symbol, quantity, price, side and position effect - are explicitly specified. The symbol is specified via a *USOptionsSymbol* structure containing US Options symbol attributes:

```
struct USOptionSymbol {
    enum PutOrCall {
        put = 0,
        call
    };

    std::string baseSymbol;
    PutOrCall putOrCall;
    uint8_t expirationYear;  // last 2 digits (YY)
    uint8_t expirationMonth; // MM
    uint8_t expirationDay;   // DD
    USOptionScaledStrikePrice strikePrice;
};
```

Here US Option strike price is specified using a scaled price type similar to the one used for order limit prices but using a non-default factor of 1,000 to accommodate precision of up to 3 digits after the decimal point:

```
static const int usOptionsStrikePriceScalingFactor = 1000;
```

Like with US Equities and Futures orders, the API assumes a limit order unless price is set to *marketPrice* or is not specified at all. Note that zero is a valid price value for US Options orders.

Optional order properties like *timeInForce*, *postOnly*, *ISO*, etc are summarized in the *properties* object belonging to the *USOptionsOrderProperties* class defined below:

```
class USOptionsOrderProperties {
public:
    enum TimeInForce {
        timeInForceDay = 0,
        timeInForceOpg = 1,
        timeInForceIoc = 2,
        timeInForceGoodTillDate = 4,
        timeInForceAtTheClose = 5,
        timeInForceFok = 8
    };

    enum MarketRoutingSelector {
        marketRoutingNone,
        marketRoutingBats,
        marketRoutingSusquehanna
    };

    enum CustomerType {
        customerTypeNone = -1,
        customerTypeCustomer = 0,
        customerTypeFirm = 1,
        customerTypeFirmBrokerDealer = 2,
        customerTypeAwayMarketMaker = 5,
        customerTypeProfessional = 8
    };

    TimeInForce getTimeInForce() const;
    uint64_t getExpireTime() const;
    uint32_t getISOGroupId() const;
    ScaledPrice getDiscretionOffset() const;
    int getMaxFloor() const;
    int getMinQty() const;
    MarketRoutingSelector getMarketRoutingSelector() const;
    const std::string& getRoutingInstructions() const;
    bool isRoutingAllowed() const;
    AllowRouting getRoutingAllowed() const;
```

```
    bool isPostOnly() const;
    bool isIso() const;
    bool isAllOrNone() const;
    bool isArcaTracking() const;
    const std::string& getIseExposureFlag() const;
    char getIseDisplayWhen() const;
    int getIseDisplayRange() const;
    CustomerType getCustomerType() const;
    const std::string& getClientData1() const;
    const std::string& getClientData2() const;
    const std::string& getClientData3() const;

    void setTimeInForce(TimeInForce value);
    void setExpireTime(uint64_t value);
    void setISOGroupId(uint32_t value);
    void setDiscretionOffset(ScaledPrice value);
    void setMaxFloor(int value);
    void setMinQty(int value);
    void setMarketRoutingSelector(MarketRoutingSelector value);
    void setRoutingInstructions(const std::string& value);
    void setRoutingAllowed(bool value);
    void unsetRoutingAllowed();
    void setPostOnly(bool value);
    void setIso(bool value);
    void setAllOrNone(bool value);
    void setArcaTracking(bool value);
    void setIseExposureFlag(const std::string& value);
    void setIseDisplayWhen(char value);
    void setIseDisplayRange(int value);
    void setCustomerType(CustomerType type);
    void setClientData1(const std::string& value);
    void setClientData2(const std::string& value);
    void setClientData3(const std::string& value);

private:
    TimeInForce timeInForce;
    uint64_t expireTime;
    uint32_t isoGroupId;
    ScaledPrice discretionOffset;
    int maxFloor;
    int minQty;
    MarketRoutingSelector marketRoutingSelector;
    std::string routingInstructions;
    AllowRouting allowRouting;
    bool postOnly;
    bool iso;
    bool allOrNone;
    bool arcaTracking;
    std::string iseExposureFlag;
    char iseDisplayWhen;
```

```
    int iseDisplayRange;
    CustomerType customerType;
    std::string clientData1;
    std::string clientData2;
    std::string clientData3;
}
```

To facilitate the handling of the optional orders properties object, a *nullUSOptionsProperties* static object is defined and used as a default argument:

```
extern const USOptionOrderProperties nullUSOptionsProperties;
```

4.3 placeAlgoOrder

```
CallStatus placeAlgoOrder(
    OrderId orderId,
    const std::string& route,
    const std::string& symbol,
    Side side,
    int quantity,
    const std::string& strategy,
    ScaledPrice price = marketPrice,
    const AlgoOrderProperties& properties = nullAlgoProperties);
```

This call has the same semantics as a *placeOrder()* call except that it requires *strategy* to be specified[4] and accepts an *AlgoOrderProperties* object, defined below.

```
class AlgoOrderProperties {
public:
    enum TimeInForce {
        timeInForceDay,
        timeInForceOpg,
        timeInForceIoc,
        timeInForceExtendedDay,
        timeInForceGoodTillDate,
        timeinForceAtTheClose,
        timeInForceAtOpenThenDay = 9
    };

    enum PegType {
        pegTypeNone,
        pegTypePrimary,
        pegTypeMarket,
        pegTypeMidpoint,
        pegTypeAlternateMidpoint,
        pegTypePriceImprovedPrimary,
```

[4]Possible values are set by the venue and can be found in the venue specific documentation.

```
        pegTypePriceImprovedMarket,
        pegTypePriceImprovedMidpoint,
        pegTypeMidpointDiscretionary,
        pegTypeMidpointPennySpread
    };

    enum QuickstartSetting {
        quickstartSettingNone,
        quickstartSettingDisable,
        quickstartSettingEnable
    };

    enum SweepType {
        sweepTypeDefault,
        sweepTypeStandard,
        sweepTypeIso
    };

    TimeInForce getTimeInForce() const;
    uint64_t getExpireTime() const;
    ScaledPrice getDiscretionOffset() const;
    int getMaxFloor() const;
    int getMinQty() const;
    bool isAggressionSet() const;
    int getAggression() const;
    uint64_t getStartTime() const;
    uint64_t getEndTime() const;
    ScaledPrice getIWouldPx() const;
    bool isSaleAffirm() const;
    int getShortSaleAffirmLongQuantity() const;
    bool isInvisible() const;
    PegType getPegType() const;
    ScaledPrice getPegDifference() const;
    int getMinimumTriggerVol() const;
    ScaledPercent getMinimumTriggerPercentage() const;
    QuickstartSetting getQuickstartSetting() const;
    SweepType getSweepType() const;
    ScaledPrice getMaxMarketOrderSlippageAmount() const;
    const std::string& getExcludedVenues() const;
    bool isRegularSessionOnly() const { return regularSessionOnly; }
    const std::string& getClientData1() const;
    const std::string& getClientData2() const;
    const std::string& getClientData3() const;

    void setTimeInForce(TimeInForce value);
    void setExpireTime(uint64_t value);
    void setDiscretionOffset(ScaledPrice value);
    void setMaxFloor(int value);
    void setMinQty(int value);
    void setAggression(int value);
```

```
    void unsetAggression();
    void setStartTime(uint64_t value);
    void setEndTime(uint64_t value);
    void setIWouldPx(ScaledPrice value);
    void setSaleAffirm(bool value);
    void setShortSaleAffirmLongQuantity(int value);
    void setInvisible(bool value);
    void setPegType(PegType value);
    void setPegDifference(ScaledPrice value);
    void setMinimumTriggerVol(int value) const;
    void setMinimumTriggerPercentage(ScaledPercent value);
    void setQuickstartSetting(QuickstartSetting value)
    void setSweepType(SweepType value)
    void setMaxMarketOrderSlippageAmount(ScaledPrice value);
    void setExcludedVenues(const std::string& value);
    void setRegularSessionOnly(bool value) { regularSessionOnly = value; }
    void setClientData1(const std::string& value);
    void setClientData2(const std::string& value);
    void setClientData3(const std::string& value);

private:
    TimeInForce timeInForce;
    uint64_t expireTime;
    ScaledPrice discretionOffset;
    int maxFloor;
    int minQty;
    bool aggressionSet;
    int aggression;
    uint64_t startTime;
    uint64_t endTime;
    ScaledPrice iWouldPx;
    bool saleAffirm;
    int shortSaleAffirmLongQuantity;
    bool invisible;
    PegType pegType;
    ScaledPrice pegDifference;
    int minimumTriggerVol;
    ScaledPercent minimumTriggerPercentage;
    QuickstartSetting quickstart;
    SweepType sweepType;
    ScaledPrice maxMarketOrderSlippageAmount;
    std::string excludedVenues;
    bool regularSessionOnly;
    std::string clientData1;
    std::string clientData2;
    std::string clientData3;
};
```

To facilitate the handling of the optional order properties object, a *nullAlgoProperties*

static object is defined and used as a default argument:

```
extern const AlgoOrderProperties nullAlgoProperties;
```

4.4 placeUSOptionsAlgoOrder

```
CallStatus placeUSOptionsAlgoOrder(
    OrderId orderId,
    const std::string& route,
    const USOptionSymbol& symbol,
    Side side,
    PositionEffect positionEffect,
    int quantity,
    const std::string& strategy,
    ScaledPrice price = marketPrice,
    const USOptionsAlgoOrderProperties& properties =
        nullUSOptionsAlgoProperties);
```

This call has the same semantics as a *placeUSOptionsOrder()* call except that it requires *strategy* to be specified and accepts an *USOptionsAlgoOrderProperties* object, defined below.

```
class USOptionsAlgoOrderProperties {
public:
    enum TimeInForce {
        timeInForceDay,
        timeInForceOpg,
        timeInForceIoc,
        timeInForceAtTheClose = 5,
        timeInForceFok = 8
    };

    TimeInForce getTimeInForce() const;
    bool isAggressionSet() const;
    int getAggression() const;
    uint64_t getStartTime() const;
    uint64_t getEndTime() const;
    int getMaxFloor() const;
    bool isAllOrNone() const;
    const std::string& getClientData1();
    const std::string& getClientData2();
    const std::string& getClientData3();

    void setTimeInForce(TimeInForce value);
    void setAggression(int value);
    void unsetAggression();
    void setStartTime(uint64_t value);
    void setEndTime(uint64_t value);
    void setMaxFloor(int value);
```

```
    void setAllOrNone(bool value);
    void setClientData1(const std::string& value);
    void setClientData2(const std::string& value);
    void setClientData3(const std::string& value);

private:
    TimeInForce timeInForce;
    bool aggressionSet;
    int aggression;
    uint64_t startTime;
    uint64_t endTime;
    uint32_t maxFloor;
    bool allOrNone;
    std::string clientData1;
    std::string clientData2;
    std::string clientData3;
};
```

To facilitate the handling of the optional order properties object, a *nullUSOptionsAlgoProperties* static object is defined and used as a default argument:

```
extern const USOptionsAlgoOrderProperties nullUSOptionsAlgoProperties;
```

4.5 cancelOrder

```
CallStatus cancelOrder(OrderId orderId);
```

A cancel call requires only the orderID of the original order to be canceled. The ID for the cancel request itself is generated by the Lime trading platform. This call applies to all supported order types and asset classes.

4.6 partialCancelOrder

```
CallStatus partialCancelOrder(OrderId orderId, int leftQty);
```

A partial cancel call requires the orderID of the original order and the quantity to be left open (not including filled quantity). The ID for the cancel request itself is generated by the Lime trading platform. This call applies to US Equities only.

4.7 cancelReplaceOrder

```
CallStatus cancelReplaceOrder(
    OrderId orderId,
    OrderId replaceOrderId,
    int quantity,
    ScaledPrice price = samePrice,
```

```
    const CancelReplaceProperties = nullCrProperties);
```

A cancel replace request requires the *orderId* of the replacement order in addition to the ID of the order being canceled. In addition, *quantity* and *price* must be specified as either may be changed. Setting *quantity* or *price* to zero will be interpreted as meaning that the respective property should remain unchanged. Setting price to *samePrice* or not specifying it at all leaves the price unchanged as well. This call applies to US Equities and Futures only.

Optional cancel replace properties are summarized in the *properties* object. Below is the definition of the *CancelReplaceProperties* class:

```
class CancelReplaceProperties {
public:

    int getShortSaleAffirmLongQuantity() const;
    int getMinQty() const;

    void setShortSaleAffirmLongQuantity(int value);
    void setMinQty(int value);

    int getMinimumTriggerVol() const;
    void setMinimumTriggerVol(int value);

private:
    int shortSaleAffirmLongQuantity;
    int minQty;
    int minimumTriggerVol;
};
```

4.8 cancelReplaceUSOptionsOrder

```
CallStatus cancelReplaceUSOptionsOrder(
    OrderId orderId,
    OrderId replaceOrderId,
    int quantity,
    ScaledPrice price = samePrice);
```

This call has the same semantics as a *cancelReplaceOrder()* call except that it applies to US Options only. A key difference from the US Equities and Futures call is that a zero price is interpreted as a legitimate new price rather than an indication that a price should remain unchanged. For the latter purpose, the price should either be set to *samePrice* or not specified at all.

4.9 cancelReplaceAlgoOrder

```
CallStatus cancelReplaceAlgoOrder(
    OrderId orderId,
    OrderId replaceOrderId,
    int quantity,
    ScaledPrice price = samePrice,
    const AlgoCancelReplaceProperties& properties = nullAlgoCrProperties);
```

This call has the same semantics as a *cancelReplaceOrder()* call except that it accepts an *AlgoCancelReplaceProperties* object, defined below.

```
class AlgoCancelReplaceProperties {
public:
    enum QuickstartSetting {
        quickstartSettingNone,
        quickstartSettingDisable,
        quickstartSettingEnable
    };

    uint64_t getStartTime() const;
    uint64_t getEndTime() const;
    int getShortSaleAffirmLongQuantity() const;
    int getMinimumTriggerVol() const;
    ScaledPercent getMinimumTriggerPercentage() const;
    QuickstartSetting getQuickstartSetting() const;
    ScaledPrice getMaxMarketOrderSlippageAmount() const;

    void setStartTime(uint64_t value);
    void setEndTime(uint64_t value);
    void setShortSaleAffirmLongQuantity(int value);
    void setMinimumTriggerVol(int value);
    void setMinimumTriggerPercentage(ScaledPercent value);
    void setQuickstartSetting(QuickstartSetting value);
    void setMaxMarketOrderSlippageAmount(ScaledPrice value);

private:
    uint64_t startTime;
    uint64_t endTime;
    int shortSaleAffirmLongQuantity;
    int minimumTriggerVol;
    ScaledPercent minimumTriggerPercentage;
    QuickstartSetting quickstart;
    ScaledPrice maxMarketOrderSlippageAmount;
};
```

4.10 cancelReplaceAlgoOrder

```
CallStatus cancelReplaceUSOptionsAlgoOrder(
    OrderId orderId,
```

```
    OrderId replaceOrderId,
    int quantity,
    ScaledPrice price = samePrice,
    const USOptionsAlgoCancelReplaceProperties& properties =
        nullUSOptionsAlgoCrProperties);
```

This call has the same semantics as a *cancelReplaceUSOptionsOrder()* call except that it accepts an *USOptionsAlgoCancelReplaceProperties* object, defined below.

```
class USOptionsAlgoCancelReplaceProperties {
public:
    uint64_t getStartTime() const;
    uint64_t getEndTime() const;
    int getShortSaleAffirmLongQuantity() const;

    void setStartTime(uint64_t value);
    void setEndTime(uint64_t value);
    void setShortSaleAffirmLongQuantity(int value);

private:
    uint64_t startTime;
    uint64_t endTime;
    int shortSaleAffirmLongQuantity;
};
```

4.11 cancelAllOpenOrders

```
CallStatus cancelAllOpenOrders();
```

This call cancels all open orders for the account. If an order has been partially filled, the cancel request will be submitted for the unfilled portion of the order.

5 Trading Callbacks

The following callbacks are defined for the trading calls:

```
//
// Required Trading Callbacks
//
virtual void onOrderAccept(
    OrderId orderId,
    OrderId limeOrderId,
    const OrderAckAttributes &attributes,
    uint64_t eventId) = 0;

virtual void onOrderFill(
    OrderId orderId,
```

```
    const FillInfo& fillInfo,
    uint64_t eventId) = 0;

virtual void onUSOptionsOrderFill(
    OrderId orderId,
    const USOptionsFillInfo& fillInfo,
    uint64_t eventId) = 0;

virtual void onOrderCancel(
    OrderId orderId,
    uint64_t eventId) = 0;

virtual void onOrderCancel(
    OrderId orderId,
    const std::string& reason,
    uint64_t eventId) = 0;


virtual void onOrderPartialCancel(
    OrderId orderId,
    int leftQty,
    uint64_t eventId) = 0;

virtual void onOrderReplace(
    OrderId orderId,
    OrderId replaceOrderId,
    OrderId limeReplaceOrderId,
    const OrderAckAttributes &attributes,
    uint64_t eventId) = 0;

virtual void onOrderReject(
    OrderId orderId,
    const std::string& reason,
    uint64_t eventId) = 0;

//
// Optional Trading Callbacks
//

virtual void onOrderCancelReject(
    OrderId orderId,
    const std::sting& reason,
    uint64_t eventId);

virtual void onOrderCancelReplaceReject(
    OrderId orderId,
    OrderId replaceOrderId,
    const std::string& reason,
    uint64_t eventId);
```

```
virtual void onManualOrder(
    OrderId orderId,
    const ManualOrderInfo& info,
    uint64_t eventId);

virtual void onManualUSOptionsOrder(
    OrderId orderId,
    const ManualUSOptionsOrderInfo& info,
    uint64_t eventId);

virtual void onManualOrderReplace(
    OrderId orderId,
    OrderId replaceOrderId,
    int quantity,
    ScaledPrice price,
    uint64_t eventId);
```

Note that every callback contains an eventID that can be used to retrieve older messages when logging in after a connection interruption.

The *onOrderAccept* and *onOrderReplace* callbacks contain the *limeOrderId*'s of the new and replaced orders, respectively, in addition to the client supplied *orderId*'s. These are Lime generated orderIDs that the Lime trading platform uses for identifying orders to venues.

The *OrderAckAttributes* class used in the *onOrderAccept* and *onOrderReplace* callbacks is defined as follows:

```
struct OrderAckAttributes {
    OrderAckAttributes();
    ScaledPrice adjustedPrice;
    ScaledPrice displayPrice;
    bool displayPriceAdjusted;
    std::string marketClOrdId;
    std::string marketOrderId;
    PositionEffect positionEffect;
    Side side;
}
```

Here *adjustedPrice* is the price at which an order or replace was accepted by the market if it is different from the entered price. Some markets (e.g. NASDAQ) may re-price orders on entry. If the accepted price is the same as the entered price of the order or the cancel replace, as is usually the case, *adjustedPrice* is set to zero. The *displayPrice* is the price at which the order is displayed by the exchange. It is returned if it is different from the *adjustedPrice* and if the customer account is configured to receive displayed prices. Some exchanges do not supply the display price itself but inform whether display price is different from the price at which the order was accepted. In such event *displayPriceAdjusted* flag

will be set, if the account is configured for display price notifications. The flag is always set when non-zero *displayPrice* is provided. The *positionEffect* is the value of the positionEffect as sent to the market ,the trading server which may remark this value. The *side* is the value of the side as sent to the market, the trading server may remark this value.

The value of *marketClOrdId* is set to a clOrdId used by the Lime trading platform to identify the order to the market if it is different from the limeOrderId. This attribute will only be supplied if the customer account is explicitly configured for it. Similarly, *marketOrderId* is set to the orderID used by the exchange to identify the order. This attribute will only be supplied if the customer account is explicitly configured for it.

The *onOrderFill()* callback is specific to US Equities and Futures executions. It returns a *FillInfo* structure (below) with the execution parameters *lastShares*, *lastPrice*, *leftQty*, *liquidity*, *transactTime*, *execType* and *execId*. The *execType* is an enum that identifies execution as Fill, Bust, or Correction, and *execId* is a unique, Lime generated transaction identifier. For busts and corrections, the returned *execId* is that of the original execution that is being busted or corrected. The callback identifies the order using *orderId* and also returns original order value of *symbol* and reversed *side* for completeness. The *contraBroker* field is provided optionally, if passed back by an exchange, and so are *lastMarket* and *marketLiquidity*.

```
struct FillInfo {
    std::string symbol;
    Side side;
    int lastShares;
    ScaledPrice lastPrice;
    int leftQty;
    int liquidity;
    uint64_t transactTime;
    ExecType execType;
    std::string execId;
    std::string contraBroker;
    std::string lastMarket;
    std::string clientData1;
    std::string clientData2;
    std::string clientData3;
    std::string marketLiquidity;
    uint64_t transactTimeNanos;
}
```

Liquidity codes are itemized in Appendix B of the Lime FIX Manual. MarketLiquidity codes are detailed in section 7.6.1 of the Lime Fix Manual. The value of *transactTime* is based on the market timestamp provided with the fill and is the number of milliseconds since Unix epoch (Jan. 1, 1970). *transactTimeNanos* provides the same value with nanosecond precision.

Similarly, the *onUSOptionsOrderFill()* callback returns an *USOptionsFillInfo* structure, defined below. It is similar to its US Equities counterpart with the difference that the symbol is specified as a *USOptionSymbol* struct as opposed to a string in US Equities *FillInfo*:

```
struct USOptionsFillInfo {
    USOptionSymbol symbol;
    Side side;
    int lastShares;
    ScaledPrice lastPrice;
    int leftQty;
    int liquidity;
    uint64_t transactTime;
    ExecType execType;
    std::string execId;
    std::string contraBroker;
    std::string clientData1;
    std::string clientData2;
    std::string clientData3;
    std::string marketLiquidity;
    uint64_t transactTimeNanos;
}
```

Sometimes the orders on Lime accounts are placed out of band with Lime Trading API connections (e.g. using LimePortal). To facilitate the tracking of positions and exposure in the customer application, the API provides optional callbacks echoing the information about such manual orders: *onManualOrder()*, *onManualUSOptionsOrder()* and *onManualOrderReplace()*.[5] The order info returned by the first two callbacks referring to US Equities and Futures orders and US Options orders, respectively, is summarized in the following structures:

```
struct ManualOrderInfo {
    std::string symbol;
    std::string route;
    int quantity;
    ScaledPrice price;
    Side side;
}
```

and

```
struct ManualUSOptionsOrderInfo {
    USOptionSymbol symbol;
    std::string route;
    int quantity;
    ScaledPrice price;
```

[5] Accounts must be explicitly configured to enable this behavior.

```
    Side side;
    PositionEffect positionEffect;
}
```

6 Exceptions

Under some circumstances, the Lime Trading API may throw an exception. This can happen in the API constructor if an API version incompatibility is detected, if there is a connectivity problem or if invalid options are supplied. Some internal errors may also result in an exception on a trading call.

It is recommended that all Lime Trading API exceptions are caught. When an exception is caught, the affected API instance should be deleted.

Note that Lime Trading API exceptions are only thrown in the thread(s) where the API was constructed or where trading calls were placed. Exceptions will not be thrown from a callback thread.

The Lime Trading API Exception class is listed below:

```
class TradingApiException {
public:
    TradingApiException(const std::string& message);
    ~TradingApiException();
    const char what() const;
private:
    const std::string message;
};
```

A Example Client

The code in exampleClient.cc and exampleClient.hh provides a reference implementation for a process linking in the Lime Trading API.

To build the example trading application type 'make' in the directory where the Lime Trading API package was unpacked. The executable is called example-client. When you start example client, type 'help' at the exampleClient prompt to get the summary of the commands. You can login into Lime trading platform with your credentials using 'login' command, and trade at the stub LIME-ECN venue.

Example client presents a simple interface to test trading via the Lime Trading API. An instance of 'API context' instance is created upon execution of 'login' command. API context stores a pointer to the Lime Trading API instance and stores some useful information

like the current eventId. API contexts are identified by a numeric ID. All active instances of the API and associated eventId's can be displayed via a 'dump' command.

ExampleClient prompt indicates the ID of the API context currently in its focus. Multiple API instances can be maintained at the same time, and trades can be placed on them concurrently. To switch between API instances use 'switchAPI' command indicating the ID of the destination API. To remove an API context, use 'logout' command.

Trading callbacks from the API are received via the ExampleListener class that is derived from LimeBrokerage::Listener. Each API context has its own instance of the ExampleListener class.

Note that API must not be destructed from the context of a callback. When API instance has to be torn down as a result of a callback, a special flag in the API context is set. It is then checked from the main thread where actual cleanup is activated via removeApiCtx call.

Use 'quit' command to exit from the exampleClient.

B Fix Tag Mapping

Many API arguments can be mapped to Lime fix tags. For more information on the fix tags, please refer to the Lime FIX Manual.

B.1 Equities

The following table applies to *placeOrder()* calls and the *OrderProperties* class.

API Argument	Affected Fix Tags
orderId	11
symbol	55
quantity	38
price	40, 44
side	54
route	100
allowRouting	9011
arcaPassiveLiquidity	9054
arcaTracking	9040
batsDarkScan	9019
batsNoRescrapeAtLimit	9037
ceilingFloorPrice	9062

Continued on next page

API Argument	Affected Fix Tags
clientData1	9050
clientData2	9052
clientData3	9053
discretionOffset	389
expireTime	126
farPegOffset	9070
imbalanceOnly	9022
invisible	9003
iso	9017
isoGroupId	9060
lockedCrossedAction	9064
marketDisplayPrice	9509
maxFloor	111
minQty	110
minimumTriggerVol	9568
nasdaqPostOnly	9036
nearPegOffset	9069
nyseParityStrategy	9063
pegDifference	211
pegType	9034
postOnly	9004
regularSessionOnly	9066
retailPriceOffset	389, 9061
routeToNyse	9014
routingInstructions (with marketRoutingArca)	9068
routingInstructions (with marketRoutingBats)	9016
routingInstructions (with marketRoutingDirectEdge)	9065
routingInstructions (with marketRoutingGeneric)	9068
routingInstructions (with marketRoutingInetFix)	9032
routingInstructions (with marketRoutingKmatch)	9068
routingInstructions (with marketRoutingNyse)	9061
routingStrategy	9072
saleAffirm	9009, 9010
shortSaleAffirmLongQuantity	9067
targetLocationId	143
timeInForce	59
timeInMarket	9001
washTradePrevention	7928

B.2 Options

The following table applies to *placeUSOptionsOrder()* call and the *USOptionsOrderProperties* class.

API Argument	Affected Fix Tags
orderId	11
route	100
symbol	55, 201, 202, 9045, 9046
side	54
positionEffect	77
quantity	38
price	40, 44
allOrNone	9041
allowRouting	9011
arcaTracking	9040
clientData1	9050
clientData2	9052
clientData3	9053
customerType	204
discretionOffset	389
expireTime	126
iseDisplayRange	9058
iseDisplayWhen	9057
iseExposureFlag	9056
iso	9017
isoGroupId	9060
maxFloor	111
minQty	110
postOnly	9004
routingInstructions (with marketRoutingBats)	9016
routingInstructions (with marketRoutingGeneric)	9068
routingInstructions (with marketRoutingSusquehanna)	9051
timeInForce	59

B.3 Algo

The following table applies to *placeAlgoOrder()* calls and the *AlgoOrderProperties* class.

API Argument	Affected Fix Tags
orderId	11
route	100
symbol	55
side	54
quantity	38
strategy	9100
price	40, 44
aggression	9111
clientData1	9050
clientData2	9052
clientData3	9053
endTime	9101
excludedVenues	9323
invisible	9003
iWouldPx	9106
maxMarketOrderSlippageAmount	9322
minimumTriggerVol	9568
minimumTriggerPerc	9569
pegDifference	211
pegType	9034
quickstart	9321
sweepType	9325
saleAffirm	9009, 9010
shortSaleAffirmLongQuantity	9067
startTime	9102
timeInForce	59

B.4 Options Algo

The following table applies to *placeUSOptionsAlgoOrder()* calls and the *USOptionsAlgoOrderProperties* class.

API Argument	Affected Fix Tags
orderId	11
route	100
symbol	55, 201, 202, 9045, 9046

Continued on next page

API Argument	Affected Fix Tags
side	54
positionEffect	77
quantity	38
strategy	9100
price	40, 44
aggression	9111
allOrNone	9041
clientData1	9050
clientData2	9052
clientData3	9053
endTime	9101
maxFloor	111
startTime	9102
timeInForce	59