DeribitXTrader: Advanced Trading System

Developed By : Gaurav Singh

Contents

1 DeribitXTrader: Advanced Trading System

1.1 1. Executive Summary

DeribitTrader is a high-performance C++ trading client designed specifically for the Deribit cryptocurrency derivatives exchange. It provides a command-line interface for interacting with the Deribit API (v2) via WebSocket, enabling users to manage connections, authenticate, place orders, fetch market data (including order books and positions), subscribe to real-time data streams, and monitor performance latency. Built using C++17 and CMake, it leverages libraries like IXWebSocket, nlohmann/json, and fmt for robust and efficient operation.

This document provides a comprehensive overview of the system features, architecture, technical requirements, build process, usage instructions, testing procedures, and potential future enhancements.

1.2 2. System Overview

1.2.1 2.1 Key Features

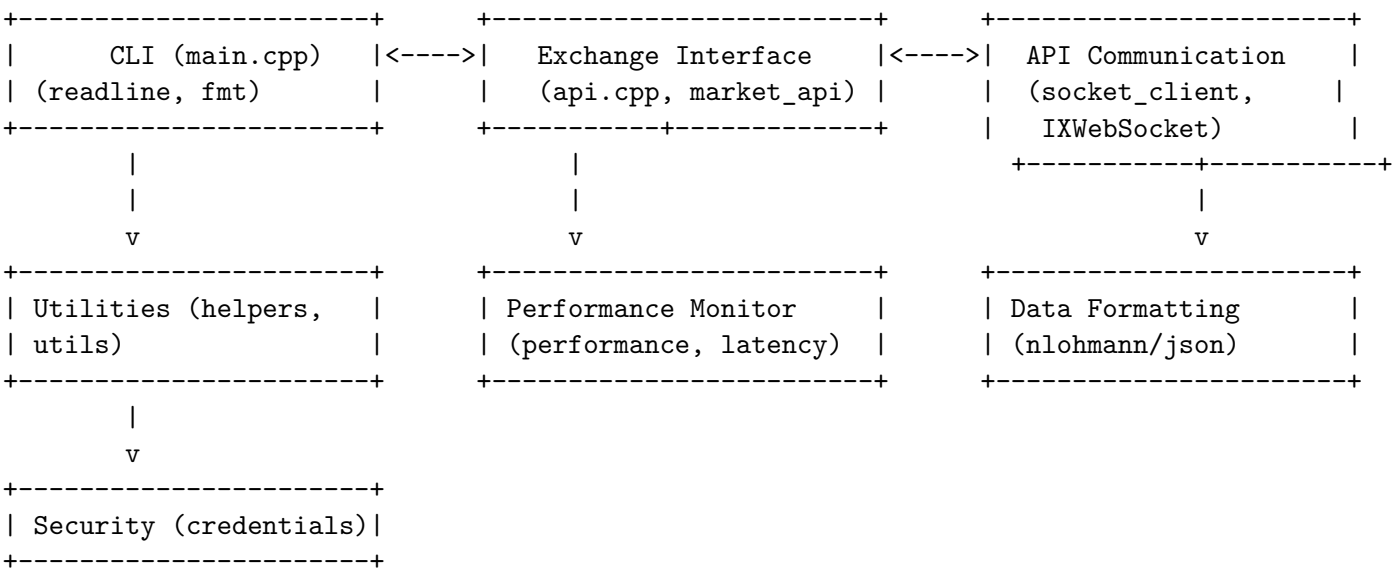
- **WebSocket Connectivity**: Establishes and manages connections to the Deribit WebSocket API (v2).
- **Authentication**: Securely authenticates sessions using client credentials (`client_id`, `client_secret`).
- **Order Management**: Places basic buy and sell orders via API calls.
- **Market Data**: Fetches order book snapshots and subscribes/unsubscribes to real-time market data channels (e.g., price index).
- **Position Management**: Retrieves open orders and current account positions.
- **Performance Monitoring**: Tracks and reports latency for key operations like API request/response cycles.
- **Command-Line Interface (CLI)**: Interactive shell (`readline`) for executing commands and viewing data streams.
- **Testing Suite**: Includes unit, integration, and performance tests using Google Test.

1.2.2 2.2 System Architecture

DeribitTrader employs a modular C++ architecture:

- **Core Application (`src/main.cpp`)**: The main entry point, providing the interactive CLI using `readline`. It parses user commands and delegates actions to the appropriate modules.
- **API Communication (`network/socket_client.cpp`, `websocket/websocket_client.h`)**: Manages WebSocket connections using the `IXWebSocket` library. Handles connection lifecycle, sending/receiving messages, and basic error handling.
- **Exchange Interface & API Logic (`exchange_interface/market_api.cpp`, `api/api.cpp`)**: Translates high-level user commands (e.g., "buy", "subscribe") into formatted JSON-RPC 2.0 requests specific to the Deribit API. Manages subscription state.
- **Data Formatting (`data_format/json_parser.hpp`, `json/json.hpp`)**: Utilizes `nlohmann/json` for parsing incoming JSON responses from the WebSocket and for constructing outgoing JSON requests.
- **Authentication & Security (`authentication/`, `security/credentials.cpp`)**: Handles the `public/auth` flow and stores credentials temporarily in memory during a session.

- **Performance Monitoring (`performance/monitor.cpp`, `latency/tracker.cpp`)**: Uses `std::chrono` to measure the duration of specific operations and generates simple latency reports.
- **Utilities (`helpers/utility.cpp`, `utils/utils.cpp`)**: Provides common helper functions, including console output formatting (`fmt`) and command parsing.
- **Testing (`tests/`)**: Contains separate executables for unit, integration, and performance tests built with Google Test.



1.3 3. Technical Requirements

1.3.1 3.1 Prerequisites

- **Operating System**: Linux or macOS (developed/tested on macOS).
- **C++ Compiler**: C++17 compatible (e.g., GCC 9+, Clang 10+).
- **Build System**: CMake 3.10 or higher.
- **System Libraries**:
  - OpenSSL (for TLS/WSS support in IXWebSocket).
  - `readline` library (for the interactive CLI).
- **Project Dependencies** (Managed via CMake `FetchContent` or system):
  - **IXWebSocket**: Core WebSocket communication library.
  - **nlohmann/json**: JSON parsing and serialization.
  - **fmt**: String formatting and colored console output.
  - **Google Test**: Framework for unit, integration, and performance tests.

1.4 4. Installation and Building

1.4.1 4.1 Build Instructions

1. **Clone the Repository**:

```
# Replace with the actual repository URL if applicable
git clone https://github.com/yourusername/DERIBitXTrader.git
cd DERIBitXTrader
```

2. **Create and Navigate to Build Directory**:

   ```
   mkdir -p build && cd build
   ```

3. **Configure with CMake**:

   ```
   # This command finds system libraries (OpenSSL, readline)
   # and fetches IXWebSocket, fmt, and Google Test sources.
   cmake ..
   ```

   Troubleshooting: Ensure OpenSSL development headers (`libssl-dev` on Debian/Ubuntu, `openssl` via Homebrew on macOS) and `readline` headers (`libreadline-dev` or `readline`) are installed if CMake reports errors finding them. Ensure `pkg-config` is installed to help locate `nlohmann-json` if installed via package manager.

4. **Build the Project**:

   ```
   make
   ```

   This compiles the source code, links libraries, and creates the main executable `trade_x_deribit` and test executables (e.g., `unit_tests`) within the `build` directory (or subdirectories like `build/tests`).

1.4.2 4.2 Configuration

Currently, the application does not use a dedicated configuration file. API credentials (`client_id`, `client_secret`) must be provided interactively via the `deribit authorize` command after establishing a connection. The Deribit API endpoint (Testnet vs Mainnet) is selected via the `connect` command (`deribit connect` defaults to Testnet).

1.5 5. System Components (Implementation Details)

1.5.1 5.1 API Communication Layer (`network/socket_client.cpp`)

- Wraps `ix::WebSocket` to manage the connection state (`ConnectionDetails`).
- Handles incoming messages via callbacks (`onMessageCallback`).
- Provides methods like `connect`, `close`, `send`.
- Uses `std::mutex` and `std::condition_variable` (`connection_mutex`, `connection_cv`) likely for synchronizing send/receive operations or waiting for responses in the main thread.

```
// Conceptual structure based on socket_client.h/cpp
class SocketEndpoint {
private:
    std::map<int, ConnectionDetails::ptr> connection_metadata;
    std::map<int, std::shared_ptr<ix::WebSocket>> ws_clients;
    // ... mutexes, counters ...

    void onMessageCallback(const ix::WebSocketMessagePtr& msg, int connection_id);

public:
    int connect(const std::string& uri);
    void close(int id, int code, const std::string& reason);
    int send(int id, const std::string& message); // Sends JSON string
```

```
    ConnectionDetails::ptr get_metadata(int id);
    void streamSubscriptions(const std::vector<std::string>& connections); // Toggles streamin
};
```

1.5.2 5.2 Exchange Interface (`api/api.cpp`, `exchange_interface/market_api.cpp`)

- `api::processRequest` acts as a dispatcher, parsing the user's command string.
- Based on the command (e.g., `buy`, `sell`, `subscribe`, `get_positions`), it constructs a `nlohmann::json` object representing the JSON-RPC request.
- Includes parameters like `instrument_name`, `amount`, `price`, `channels`, etc., as required by the specific Deribit API method.
- Returns the JSON request as a string, ready to be sent via `SocketEndpoint::send`.
- Manages a list of active subscription channel names.

```
// Conceptual structure based on api.cpp
namespace api {
    nlohmann::json createBaseRequest(const std::string& method);
    std::string processRequest(const std::string& command) {
        // 1. Parse command string (e.g., using stringstream)
        // 2. Identify target API method (e.g., "private/buy", "public/subscribe")
        // 3. Create base JSON request object
        // 4. Add specific parameters based on command arguments
        // 5. Return json_request.dump();
    }
    void addSubscription(const std::string& channel);
    void removeSubscription(const std::string& channel);
    std::vector<std::string> getActiveSubscription();
}
```

1.5.3 5.3 Performance Monitoring (`performance/monitor.cpp`)

- Implemented as a singleton (`getPerformanceMonitor`).
- Uses a `std::map` to store start times (`std::chrono::high_resolution_clock::time_point`) keyed by operation type.
- `end_measurement` calculates the duration and accumulates statistics (total time, count).
- `generate_report` formats the collected statistics into a human-readable string.

```
// Conceptual structure based on performance/monitor.h
enum class OperationType { /* e.g., API_CALL, JSON_PARSE, ... */ };

class PerformanceMonitor {
private:
    struct Stats { /* total_duration, count */ };
    std::map<OperationType, Stats> accumulated_stats;
    std::map<OperationType, std::chrono::high_resolution_clock::time_point> start_times;
    // ... mutex ...
public:
    void start_measurement(OperationType type);
    void end_measurement(OperationType type);
```

```
    std::string generate_report();
    void reset();
};
```

1.6 6. Security Measures

1.6.1 6.1 Authentication (`deribit authorize` command)

- The primary authentication mechanism involves sending the `client_id` and `client_secret` over the secure WebSocket (WSS) connection using the `public/auth` API method.
- The resulting access and refresh tokens are likely handled implicitly by the Deribit API session on the server-side for subsequent private calls within that connection. The client itself doesn't appear to explicitly store or manage these tokens based on the provided code.

1.6.2 6.2 API Key Handling (`security/credentials.cpp`)

- The `Credentials` class provides a simple in-memory storage for the `client_id` and `client_secret` provided by the user during the session.
- **Important**: Storing credentials only in memory means they are lost when the application closes and must be re-entered. For production use, secure storage (e.g., encrypted configuration file, OS keychain, environment variables) is strongly recommended. The current implementation is suitable for testing but not secure for production keys.

1.6.3 6.3 Secure Connection (WSS)

- The use of `wss://` URIs and linking against OpenSSL ensures that the communication channel between the client and Deribit is encrypted via TLS.

1.7 7. Testing Framework (`tests/`)

The project utilizes the Google Test framework for automated testing, crucial for ensuring the correctness and reliability of trading software components.

1.7.1 7.1 Test Structure

- **Unit Tests (`tests/unit/`)**: Focus on isolating and testing individual classes or functions. Examples:
  - `test_credentials.cpp`: Tests the `Credentials` class.
  - `test_json_parser.cpp`: Verifies JSON parsing logic.
  - `test_utility.cpp`: Tests helper functions.
  - `test_performance_monitor.cpp`: Tests the latency tracking mechanism.
- **Integration Tests (`tests/integration/`)**: Verify the interaction between different modules. Examples:
  - `test_deribit_api.cpp`: Tests the generation of API request strings.
  - `test_websocket_connection.cpp`: Tests establishing and interacting with a WebSocket connection (potentially against a mock server or Deribit Testnet).
- **Performance Tests (`tests/performance/`)**: Measure the execution speed of critical operations. Examples:
  - `test_json_performance.cpp`: Benchmarks JSON parsing/serialization speed.

- `test_websocket_performance.cpp`: Measures WebSocket message send/receive latency.
- `test_market_api_performance.cpp`: Benchmarks the time taken to process API requests/responses.

1.7.2 7.2 Running Tests

1. Ensure the project is built (Section 4.1). CMake automatically configures test executables.
2. Navigate to the build directory: `cd build`
3. Run tests using CTest:

   ```
   ctest --verbose
   ```

4. Alternatively, run individual test suites directly from `build/tests/`:

   ```
   cd tests
   ./unit_tests
   ./integration_tests
   ./performance_tests
   ```

5. A script `scripts/run_tests.sh` might exist to automate the build and test execution process.

1.8 8. Usage (CLI Commands)

Start the interactive client by running the executable from the `build` directory:

`./trade_x_deribit`

You will be presented with the `tradexderibit>` prompt.

Workflow Example:

1. **Connect**: `deribit connect` (Connects to Testnet, assigns an ID, e.g., 0)
2. **Authorize**: `deribit 0 authorize YOUR_CLIENT_ID YOUR_CLIENT_SECRET`
3. **Subscribe**: `deribit 0 subscribe deribit_price_index.btc_usd`
4. **View Stream**: `view_stream` (Press Ctrl+C to stop streaming)
5. **Check Positions**: `deribit 0 positions`
6. **Place Order**: `deribit 0 buy BTC-PERPETUAL 1 50000 type=limit` (Example, exact params might vary)
7. **Check Latency**: `show_latency_report`
8. **Disconnect**: `close 0`
9. **Exit**: `quit`

Full Command List:

- `help`: Display available commands.
- `connect <URI>`: Connect to a specific WebSocket URI.
- `deribit connect`: Connect to Deribit TESTNET (`wss://test.deribit.com/ws/api/v2`).
- `show <id>`: Show connection details (ID, Status, URI).
- `show_messages <id>`: Display raw JSON messages received on this connection.
- `close <id>`: Close the specified connection.
- `send <id> <json_message>`: Send a raw JSON string message.

- `deribit <id> authorize <client_id> <client_secret>`: Authenticate the connection.
- `deribit <id> buy <instrument> <amount> <price> [options...]`: Place a buy order. (Options like `type=limit/market`, `label=...` likely parsed in `api::processRequest`).
- `deribit <id> sell <instrument> <amount> <price> [options...]`: Place a sell order.
- `deribit <id> get_open_orders [instrument=<name>]`: Fetch open orders.
- `deribit <id> positions`: Fetch current account positions.
- `deribit <id> orderbook <instrument> [depth=<number>]`: Fetch the order book.
- `deribit <id> subscribe <channel_name>` / `deribit <id> subscribe <channel_name_1> <channel_name_2> ...`: Subscribe to one or more channels (e.g., `deribit_price_index.btc_usd`, `book.BTC-PERPETUAL.100ms`).
- `deribit <id> unsubscribe <channel_name>` / `deribit <id> unsubscribe <channel_name_1> ...`: Unsubscribe from channels.
- `view_subscriptions`: List channels the client is currently subscribed to.
- `view_stream`: Toggle continuous display of incoming messages from subscribed channels. (Use Ctrl+C to exit stream view).
- `show_latency_report`: Display performance metrics collected by the monitor.
- `reset_report`: Clear collected performance metrics.
- `quit` or `exit`: Terminate the application.

1.9 9. Troubleshooting

- **Connection Failed**:
  - Verify the URI (`wss://test.deribit.com/ws/api/v2` or `wss://www.deribit.com/ws/api/v2`).
  - Check network connectivity and firewall settings. Ensure outbound connections to Deribit IPs/ports are allowed.
  - Check OpenSSL installation and compatibility.
- **Authentication Error**:
  - Double-check `client_id` and `client_secret`.
  - Ensure the API key is active and has the required permissions on the Deribit platform.
  - Check if your system clock is synchronized (required for TLS/API authentication).
- **Command Not Recognized**:
  - Use `help` to see the list of valid commands.
  - Check command syntax and required arguments.
- **No Data Stream**:
  - Ensure you are connected (`show <id>`).
  - Ensure you are subscribed (`view_subscriptions`).
  - Use `view_stream` to toggle output. Check if messages are arriving using `show_messages <id>`.
- **Build Errors**:
  - Verify all prerequisites (CMake, C++ compiler, OpenSSL, readline) are installed correctly, including development headers.
  - If `FetchContent` fails, check network connection or Git installation.
  - Clean the build directory (`rm -rf build/*`) and retry `cmake .. && make`.

1.10 10. API Reference (Deribit API v2 Methods Used)

The client interacts with the Deribit API v2 via JSON-RPC 2.0 over WebSocket. Key methods used include:

Public Methods:

- `public/auth`:
  - **Purpose**: Authenticates the WebSocket session using API credentials. Required before calling any private methods.
  - **Used by**: `deribit <id> authorize <client_id> <client_secret>`
  - **Parameters**: `grant_type="client_credentials"`, `client_id`, `client_secret`.
  - **Response**: Contains `access_token`, `refresh_token`, `expires_in`, `token_type`. The client implicitly relies on the session being authenticated server-side after this call.
- `public/subscribe`:
  - **Purpose**: Subscribes the WebSocket connection to receive real-time updates for specified channels.
  - **Used by**: `deribit <id> subscribe <channel_name> [...]`
  - **Parameters**: `channels` (array of strings, e.g., `["deribit_price_index.btc_usd", "book.BTC-PERPETUAL.100ms"]`).
  - **Response**: Confirmation of successful subscriptions. Subsequent messages on the WebSocket will be notifications for these channels.
- `public/unsubscribe`:
  - **Purpose**: Unsubscribes the WebSocket connection from previously subscribed channels.
  - **Used by**: `deribit <id> unsubscribe <channel_name> [...]`
  - **Parameters**: `channels` (array of strings).
  - **Response**: Confirmation of successful unsubscriptions.
- `public/get_order_book`:
  - **Purpose**: Retrieves a snapshot of the order book (bids and asks) for a specific instrument.
  - **Used by**: `deribit <id> orderbook <instrument> [depth=<number>]`
  - **Parameters**: `instrument_name`, `depth` (optional, number of price levels).
  - **Response**: Contains lists of `bids` and `asks` (price, amount pairs), `timestamp`, `instrument_name`, etc.
- `public/get_index_price`:
  - **Purpose**: Retrieves the current index price for a specified index (e.g., BTC, ETH). Often used via subscriptions.
  - **Used by**: Subscriptions like `deribit_price_index.btc_usd`. Direct call possible but less common in this client.
  - **Parameters**: `index_name`.
  - **Response**: Contains `index_name` and `price`.
- `public/ticker`:
  - **Purpose**: Retrieves summary information (ticker data) for a specific instrument, including best bid/ask, last price, volume, etc.
  - **Used by**: Potentially used by subscriptions like `ticker.<instrument_name>.<interval>` or could be called directly (though not explicitly shown as a command).
  - **Parameters**: `instrument_name`.
  - **Response**: Contains fields like `best_ask_price`, `best_bid_price`, `last_price`, `instrument_name`, `stats`, `state`, `timestamp`.

Private Methods (require authentication via `public/auth`):

- `private/buy`:
  - **Purpose**: Places a buy order on the exchange.

  - **Used by**: `deribit <id> buy <instrument> <amount> <price> [options...]`
  - **Parameters**: `instrument_name`, `amount`, `type` (e.g., `limit`, `market`), `price` (required for limit orders), `label` (optional), potentially others like `time_in_force`, `post_only`, etc.
  - **Response**: Details of the placed order, including `order_id`, `order_state`, etc.
- `private/sell`:
  - **Purpose**: Places a sell order on the exchange.
  - **Used by**: `deribit <id> sell <instrument> <amount> <price> [options...]`
  - **Parameters**: Same as `private/buy`.
  - **Response**: Details of the placed order.
- `private/get_open_orders_by_instrument`:
  - **Purpose**: Retrieves all open orders for the authenticated user on a specific instrument.
  - **Used by**: `deribit <id> get_open_orders [instrument=<name>]` (likely calls this or `private/get_open_orders_by_currency` if no instrument is specified).
  - **Parameters**: `instrument_name`, potentially `type` (e.g., `limit`, `stop`, `all`).
  - **Response**: An array of open order objects, each containing details like `order_id`, `instrument_name`, `direction`, `amount`, `price`, `order_state`, etc.
- `private/get_positions`:
  - **Purpose**: Retrieves the current positions held by the authenticated user for a specific currency or instrument.
  - **Used by**: `deribit <id> positions` (likely calls `private/get_positions` with the currency derived from the instrument or a default like BTC/ETH).
  - **Parameters**: `currency` (e.g., `BTC`, `ETH`), potentially `kind` (e.g., `future`, `option`).
  - **Response**: An array of position objects, detailing `instrument_name`, `size`, `average_price`, `floating_profit_loss`, `index_price`, etc.
- `private/edit`:
  - **Purpose**: Modifies an existing open order (e.g., change price or amount).
  - **Used by**: (Not explicitly listed as a command, but a common trading function).
  - **Parameters**: `order_id`, `amount`, `price`.
  - **Response**: Details of the modified order.
- `private/cancel`:
  - **Purpose**: Cancels a specific open order.
  - **Used by**: (Not explicitly listed as a command, but essential for order management).
  - **Parameters**: `order_id`.
  - **Response**: Details of the cancelled order.
- `private/cancel_all`:
  - **Purpose**: Cancels all open orders across all instruments for the account.
  - **Used by**: (Not explicitly listed as a command).
  - **Parameters**: None.
  - **Response**: Number of orders cancelled.

(Refer to the official Deribit API v2 documentation for the most accurate and complete details on method names, parameters, responses, and channel naming conventions.)

1.11 11. Future Enhancements

- **Configuration File**: Implement loading of settings (API keys, URLs, default parameters) from a secure configuration file (e.g., JSON, YAML) or environment variables.

- **Advanced Order Types**: Add support for more complex orders (stop-loss, take-profit, trailing stops).
- **Robust Error Handling**: Improve parsing and reporting of API errors returned in JSON responses. Add more resilient network error handling and reconnection logic.
- **State Management**: Persist subscription state or other settings across application restarts.
- **Risk Management Module**: Implement pre-trade risk checks (e.g., max order size, max position size).
- **Logging Framework**: Integrate a dedicated logging library (e.g., spdlog) for configurable logging to files and console.
- **Order/Trade History**: Add commands to fetch historical order and trade data.
- **GUI**: Develop a graphical user interface as an alternative to the CLI.

1.12 12. License