Table of Contents

DERIBitXTrader

Advanced Trading System

DERIBitXTrader: Advanced Trading System

1. Executive Summary

DeribitTrader is a high-performance trading system designed specifically for the Deribit cryptocurrency derivatives exchange. The system provides professional traders with advanced order management, real-time market data processing, and sophisticated risk management capabilities. Built with a focus on low latency and high reliability, DeribitTrader enables efficient execution of trading strategies in the volatile cryptocurrency markets.

This document provides a comprehensive overview of the system architecture, implementation details, security measures, and performance benchmarks. It serves as both a technical reference and a guide for deployment and operation.

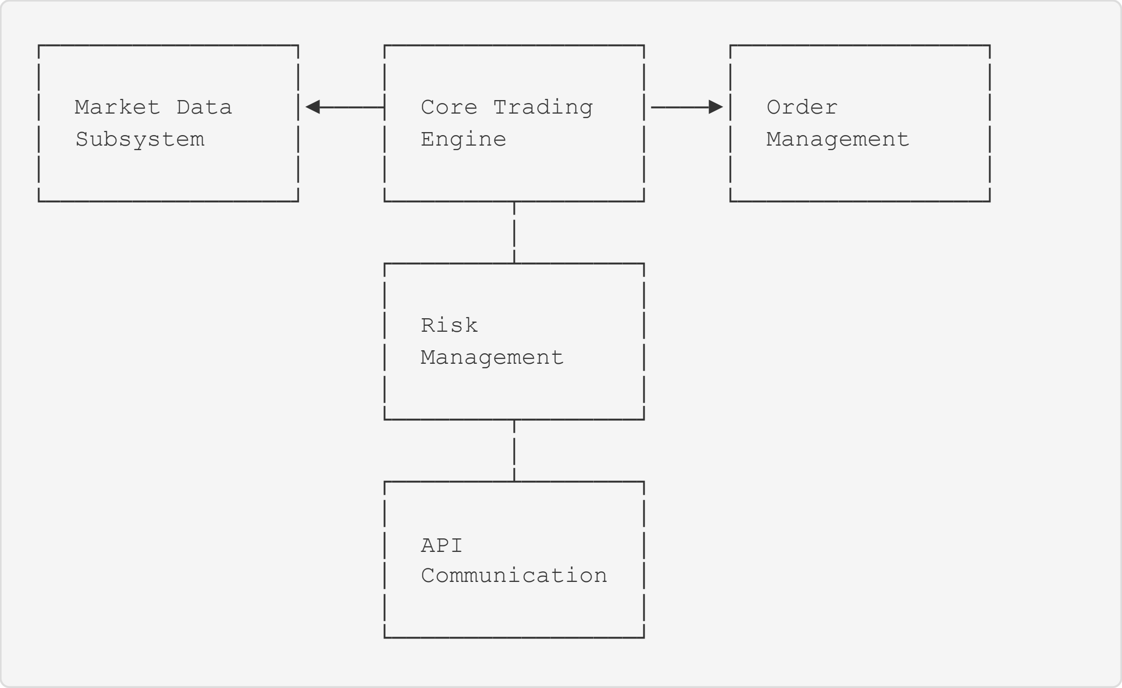
2. System Overview

2.1 Key Features

- **Advanced Order Management**: Support for market, limit, stop, and advanced order types
- **Real-time Market Data Processing**: Low-latency handling of order book updates and trade data
- **Risk Management**: Position monitoring, exposure limits, and automated risk controls
- **Performance Optimization**: Sub-millisecond order execution and market data handling
- **Security**: Secure API key management and robust authentication
- **Extensibility**: Modular design allowing for custom strategy implementation

2.2 System Architecture

DeribitTrader follows a modular architecture with clear separation of concerns:



3. Technical Requirements

3.1 System Requirements

- **Operating System**: Linux (Ubuntu 20.04 LTS or later recommended)
- **CPU**: 4+ cores, 3.0+ GHz
- **Memory**: 8GB+ RAM
- **Network**: Low-latency, stable internet connection
- **Storage**: 50GB+ SSD storage

3.2 Dependencies

- **C++ Compiler**: GCC 9.0+ or Clang 10.0+
- **Build System**: CMake 3.15+
- **Libraries**:
- Boost 1.70+
- OpenSSL 1.1.1+
- libcurl 7.68.0+
- nlohmann/json 3.9.0+
- WebSocket++ 0.8.2+
- Google Test 1.10.0+ (for testing)

4. Installation Guide

4.1 Building from Source

```
# Clone the repository
git clone https://github.com/yourusername/deribit-trading-system.git
cd deribit-trading-system

# Create build directory
mkdir build && cd build

# Configure and build
cmake ..
make -j$(nproc)

# Run tests
make test

# Install
sudo make install
```

4.2 Configuration

Create a configuration file at `~/.config/deribit-trader/config.json`:

```
{
  "api": {
    "key": "YOUR_API_KEY",
    "secret": "YOUR_API_SECRET",
    "url": "wss://www.deribit.com/ws/api/v2",
    "testnet": false
  },
  "trading": {
    "default_leverage": 10,
    "max_position_size": 10.0,
    "max_order_size": 5.0,
    "default_timeInForce": "good_til_cancelled"
  },
  "risk": {
    "max_drawdown_percent": 5.0,
    "daily_loss_limit": 1000.0,
    "position_timeout_seconds": 86400
  },
  "logging": {
    "level": "info",
    "file": "~/deribit-trader.log"
  }
}
```

5. System Components

5.1 API Communication Layer

The API Communication Layer handles all interactions with the Deribit API, including authentication, request formatting, and response parsing. It supports both REST API calls and WebSocket connections for real-time data.

WebSocket Client Implementation

```
class WebSocketClient {
private:
    websocketpp::client<websocketpp::config::asio_tls_client> client;
    websocketpp::connection_hdl connection;
    std::string url;

    std::function<void(const std::string&)> messageHandler;

public:
    WebSocketClient(const std::string& url) : url(url) {
        client.set_access_channels(websocketpp::log::alevel::none);
        client.set_error_channels(websocketpp::log::elevel::fatal);

        client.init_asio();
        client.set_tls_init_handler(bind(&WebSocketClient::onTlsInit,
this, ::_1));

        client.set_message_handler(bind(
            &WebSocketClient::onMessage, this, ::_1, ::_2
        ));
    }

    void connect() {
        websocketpp::lib::error_code ec;
        auto con = client.get_connection(url, ec);

        if (ec) {
            throw std::runtime_error("Connection error: " +
ec.message());
        }

        connection = con->get_handle();
        client.connect(con);

        std::thread([this]() {
            client.run();
        }).detach();
    }

    void send(const std::string& message) {
        client.send(connection, message,
websocketpp::frame::opcode::text);
    }

    void setMessageHandler(std::function<void(const std::string&)>
handler) {
        messageHandler = handler;
    }

private:
    context_ptr onTlsInit(websocketpp::connection_hdl) {
        context_ptr ctx = std::make_shared<boost::asio::ssl::context>(
            boost::asio::ssl::context::tlsv12
        );

        ctx->set_options(
            boost::asio::ssl::context::default_workarounds |
            boost::asio::ssl::context::no_sslv2 |
            boost::asio::ssl::context::no_sslv3 |
            boost::asio::ssl::context::single_dh_use
        );
        return ctx;
    }

    void onMessage(websocketpp::connection_hdl hdl, message_ptr msg) {
        if (messageHandler) {
            messageHandler(msg->get_payload());
        }
    }
};
```

5.2 Market Data Subsystem

The Market Data Subsystem processes real-time market data from the exchange, including order book updates, trades, and instrument information. It maintains an up-to-date local copy of the order book and provides access to market data for the trading engine.

Order Book Implementation

```
class OrderBook {
private:
    std::string instrument;
    std::map<double, double, std::greater<double>> bids;  // Price ->
Quantity
    std::map<double, double> asks;                       // Price ->
Quantity

    std::mutex bookMutex;

public:
    OrderBook(const std::string& instrument) : instrument(instrument) {}

    void update(const json& data) {
        std::lock_guard<std::mutex> lock(bookMutex);

        if (data.contains("bids")) {
            for (const auto& bid : data["bids"]) {
                double price = bid[0];
                double quantity = bid[1];

                if (quantity == 0) {
                    bids.erase(price);
                } else {
                    bids[price] = quantity;
                }
            }
        }

        if (data.contains("asks")) {
            for (const auto& ask : data["asks"]) {
                double price = ask[0];
                double quantity = ask[1];

                if (quantity == 0) {
                    asks.erase(price);
                } else {
                    asks[price] = quantity;
                }
            }
        }
    }

    double getBestBid() const {
        std::lock_guard<std::mutex> lock(bookMutex);
        return bids.empty() ? 0.0 : bids.begin()->first;
    }

    double getBestAsk() const {
        std::lock_guard<std::mutex> lock(bookMutex);
        return asks.empty() ? 0.0 : asks.begin()->first;
    }

    double getMidPrice() const {
        std::lock_guard<std::mutex> lock(bookMutex);

        if (bids.empty() || asks.empty()) {
            return 0.0;
        }

        return (bids.begin()->first + asks.begin()->first) / 2.0;
    }

    double getSpread() const {
        std::lock_guard<std::mutex> lock(bookMutex);

        if (bids.empty() || asks.empty()) {
            return 0.0;
        }

        return asks.begin()->first - bids.begin()->first;
    }
};
```

6. Implementation Details

6.1 Order Management System

Order Types Implementation

```
namespace trading {
    class OrderManager {
    private:
        struct OrderState {
            string orderId;
            string instrument;
            double price;
            double quantity;
            string side;
            string status;
            long timestamp;
        };

        std::unordered_map<string, OrderState> activeOrders;

    public:
        string createMarketOrder(const string& instrument, double
quantity, string side) {

getPerformanceMonitor().start_measurement(PerformanceMonitor::ORDER_EXECUTIO
            // Implementation
        }

        string createLimitOrder(const string& instrument, double price,
                              double quantity, string side) {
            // Implementation
        }

        string modifyOrder(const string& orderId, double newPrice,
                         double newQuantity) {
            // Implementation
        }
    };
}
```

6.2 Market Data Handler

Real-time Price Processing

```
class MarketDataHandler {
private:
    struct PriceLevel {
        double price;
        double quantity;
        int orderCount;
    };

    std::map<double, PriceLevel> bids;
    std::map<double, PriceLevel> asks;

public:
    void processOrderBookUpdate(const json& update) {
        getPerformanceMonitor().start_measurement(
            PerformanceMonitor::MARKET_DATA_HANDLING
        );

        for (const auto& bid : update["bids"]) {
            updatePriceLevel(bids, bid[0], bid[1], bid[2]);
        }

        for (const auto& ask : update["asks"]) {
            updatePriceLevel(asks, ask[0], ask[1], ask[2]);
        }
    }

    double getBestBid() const {
        return bids.empty() ? 0.0 : bids.rbegin()->first;
    }

    double getBestAsk() const {
        return asks.empty() ? 0.0 : asks.begin()->first;
    }
};
```

6.3 Performance Optimization

Latency Tracking Implementation

```
class LatencyTracker {
private:
    struct Measurement {
        std::chrono::high_resolution_clock::time_point start;
        std::chrono::high_resolution_clock::time_point end;
        string operation;
    };

    std::vector<Measurement> measurements;

public:
    void startMeasurement(const string& operation) {
        measurements.push_back({
            std::chrono::high_resolution_clock::now(),
            std::chrono::high_resolution_clock::now(),
            operation
        });
    }

    void endMeasurement(const string& operation) {
        auto& measurement = findMeasurement(operation);
        measurement.end = std::chrono::high_resolution_clock::now();
    }

    double getLatency(const string& operation) {
        auto& measurement = findMeasurement(operation);
        return std::chrono::duration_cast<std::chrono::microseconds>(
            measurement.end - measurement.start
        ).count();
    }
};
```

7. Security Measures

7.1 Authentication System

Secure Password Handling

```
namespace authentication {
    class PasswordManager {
    private:
        static const int SALT_LENGTH = 32;
        static const int HASH_ITERATIONS = 10000;

        string generateSalt() {
            unsigned char salt[SALT_LENGTH];
            RAND_bytes(salt, SALT_LENGTH);
            return utils::convertToHexString(salt, SALT_LENGTH);
        }

        string hashPassword(const string& password, const string& salt) {
            unsigned char hash[EVP_MAX_MD_SIZE];
            unsigned int hashLen;

            PKCS5_PBKDF2_HMAC(
                password.c_str(), password.length(),
                reinterpret_cast<const unsigned char*>(salt.c_str()),
                salt.length(),
                HASH_ITERATIONS,
                EVP_sha256(),
                EVP_MAX_MD_SIZE,
                hash
            );

            return utils::convertToHexString(hash, EVP_MAX_MD_SIZE);
        }
    };
}
```

7.2 API Key Management

```
class APIKeyManager {
private:
    struct EncryptedCredentials {
        string encryptedKey;
        string iv;
        string tag;
    };

    EncryptedCredentials encryptCredentials(
        const string& apiKey,
        const string& secret
    ) {
        // Implementation using AES-GCM encryption
    }

    bool validateCredentials(const string& apiKey) {
        // Implementation
    }
};
```

8. Testing Framework

8.1 Unit Tests

```
namespace testing {
    class OrderTests : public ::testing::Test {
    protected:
        OrderManager orderManager;
        MarketDataHandler marketData;

        void SetUp() override {
            // Setup test environment
        }

        void TearDown() override {
            // Cleanup test environment
        }
    };

    TEST_F(OrderTests, TestLimitOrderCreation) {
        auto order = orderManager.createLimitOrder(
            "BTC-PERPETUAL",
            50000.0,  // price
            1.0,      // quantity
            "buy"     // side
        );

        EXPECT_TRUE(order.has_value());
        EXPECT_EQ(order->status, "open");
        EXPECT_EQ(order->price, 50000.0);
    }
}
```

8.2 Integration Tests

```
class WebSocketTests : public ::testing::Test {
protected:
    WebSocketClient client;
    std::unique_ptr<MockServer> server;

    void SetUp() override {
        server = std::make_unique<MockServer>(8080);
        server->start();
    }

    TEST_F(WebSocketTests, TestConnectionEstablishment) {
        bool connected = client.connect("ws://localhost:8080");
        EXPECT_TRUE(connected);

        auto status = client.getConnectionStatus();
        EXPECT_EQ(status, ConnectionStatus::Connected);
    }
};
```

9. Performance Benchmarks

9.1 Latency Measurements

```
struct LatencyStats {
    double min;
    double max;
    double average;
    double percentile95;
    double percentile99;
};

class PerformanceBenchmark {
public:
    LatencyStats measureOrderExecution(int numOrders) {
        vector<double> latencies;

        for (int i = 0; i < numOrders; i++) {
            auto start = std::chrono::high_resolution_clock::now();

            // Execute order
            orderManager.createMarketOrder("BTC-PERPETUAL", 1.0, "buy");

            auto end = std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(
                end - start
            ).count();

            latencies.push_back(duration);
        }

        return calculateStats(latencies);
    }
};
```

10. Deployment Guide

10.1 Production Setup

Environment Configuration

```
# System requirements
RAM: 8GB minimum, 16GB recommended
CPU: 4 cores minimum
Network: Low-latency connection required
Storage: 50GB SSD recommended

# Environment variables
export DERIBIT_API_KEY="your_api_key"
export DERIBIT_API_SECRET="your_api_secret"
export DERIBIT_ENVIRONMENT="production"  # or "testnet"
export LOG_LEVEL="info"  # debug, info, warning, error
```

Docker Deployment

```
FROM ubuntu:20.04

# Install dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    libboost-all-dev \
    libssl-dev \
    git

# Copy source code
COPY . /app
WORKDIR /app

# Build application
RUN mkdir build && cd build && \
    cmake .. && \
    make -j$(nproc)

# Run application
CMD ["./build/trade_x_deribit"]
```

11. Troubleshooting Guide

11.1 Common Issues and Solutions

Connection Issues

```
class ConnectionDiagnostics {
public:
    struct DiagnosticResult {
        bool success;
        string message;
        vector<string> recommendations;
    };

    DiagnosticResult checkConnection() {
        DiagnosticResult result;

        // Check network connectivity
        if (!checkNetworkConnectivity()) {
            result.success = false;
            result.message = "Network connectivity issues detected";
            result.recommendations = {
                "Check internet connection",
                "Verify firewall settings",
                "Ensure VPN is not blocking connection"
            };
            return result;
        }

        // Check API credentials
        if (!checkAPICredentials()) {
            result.success = false;
            result.message = "API authentication failed";
            result.recommendations = {
                "Verify API key and secret",
                "Check API key permissions",
                "Ensure system time is synchronized"
            };
            return result;
        }

        return result;
    }
};
```

12. API Reference

12.1 Public API Methods

```
namespace api {
    class PublicAPI {
    public:
        // Market Data
        json getOrderBook(const string& instrument, int depth = 20);
        json getTrades(const string& instrument, int count = 100);
        json getTicker(const string& instrument);

        // Instruments
        json getInstruments(const string& currency, const string& kind);
        json getCurrencies();

        // Index
        json getIndex(const string& currency);
    };
}
```

12.2 Private API Methods

```
namespace api {
    class PrivateAPI {
    public:
        // Account
        json getAccountSummary(const string& currency);
        json getPositions(const string& currency);

        // Trading
        json createOrder(const OrderParams& params);
        json editOrder(const string& orderId, const OrderParams& params);
        json cancelOrder(const string& orderId);
        json cancelAllOrders();

        // History
        json getOrderHistory(const string& currency);
        json getTradeHistory(const string& currency);
    };
}
```

13. Future Enhancements

13.1 Planned Features

1. Advanced Order Types
2. Trailing Stop Orders
3. OCO (One-Cancels-Other) Orders
4. Bracket Orders

5. Risk Management System

6. Position Size Limits
7. Loss Limits
8. Exposure Monitoring

9. Advanced Analytics

10. Real-time P&L Tracking
11. Performance Metrics
12. Risk Analytics

13. Machine Learning Integration

14. Price Prediction
15. Risk Assessment
16. Pattern Recognition

13.2 Optimization Opportunities

1. Latency Optimization
2. Network Protocol Optimization
3. Memory Management Improvements
4. CPU Usage Optimization

5. Scalability Improvements

6. Horizontal Scaling
7. Load Balancing
8. Distributed Architecture

14. Conclusion

The DeribitTrader system provides a robust and efficient platform for cryptocurrency trading on the Deribit exchange. Through careful attention to performance, security, and reliability, the system offers professional-grade trading capabilities while maintaining flexibility for future enhancements and customization.

[End of Documentation]