Micro Futures Trading System

A headless algorithmic trading platform for Interactive Brokers that executes trading strategies on micro futures contracts with real-time market data, order execution, and portfolio management.

Table of Contents

Overview

This system provides a headless backend for algorithmic trading strategies focused on micro futures contracts through Interactive Brokers (IB) TWS (Trader Workstation) or IB Gateway. It implements a real-time trading engine with comprehensive logging, interactive terminal visualization, and statistical arbitrage strategies.

The system was designed as a replacement for a Streamlit-based frontend, focusing on reliability, performance, and detailed logging instead of GUI interactions.

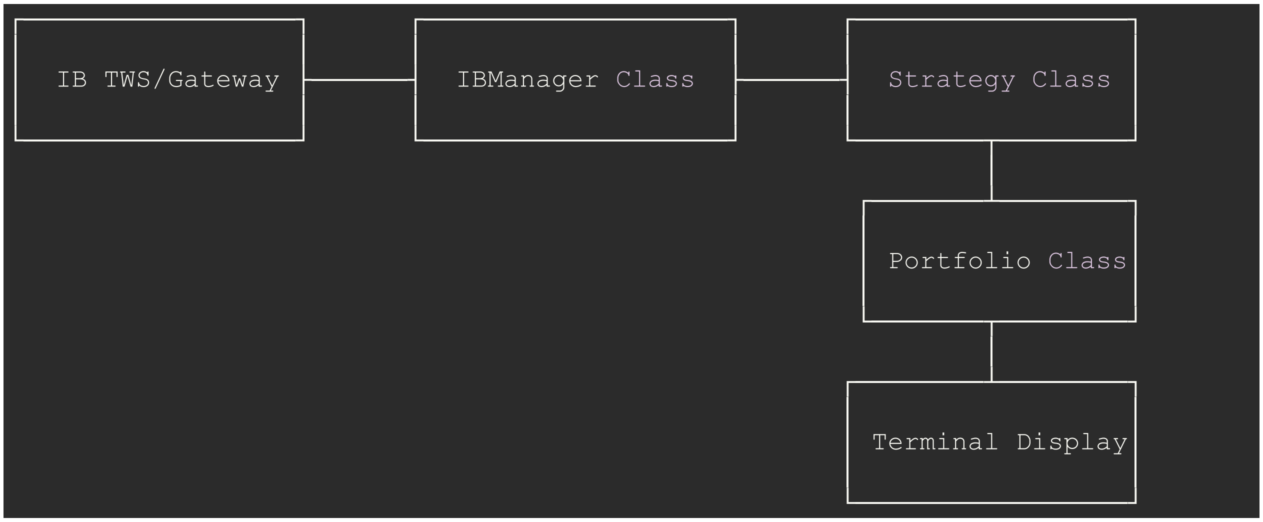
Features

- **Interactive Brokers Integration**: Full connection to IB TWS with automatic reconnection handling
- **Headless Operation**: Runs without GUI requirements, suitable for server deployment
- **Real-time Market Data Processing**: Subscribes to and processes live market data feeds
- **Statistical Arbitrage Strategy**: Implements a pairs trading strategy between Micro Crude Oil futures and related ETFs
- **Portfolio Management**: Tracks positions, calculates P&L, and manages capital allocation
- **Rich Terminal Interface**: Provides real-time updates through a color-coded terminal dashboard
- **Comprehensive Logging**: Detailed event logging with structured JSON exports

- **Fault Tolerance**: Handles connection issues, market data errors, and order execution failures
- **Configuration Options**: Flexible command-line parameters for customization

Architecture

The system follows a modular architecture with the following components:



- **IB TWS/Gateway**: External Interactive Brokers trading platform
- **IBManager**: Manages connection, market data, and order execution with IB
- **Strategy**: Implements trading logic for specific market opportunities
- **Portfolio**: Manages capital allocation and tracks performance
- **Terminal Display**: Visualizes system status and trading information

Code Breakdown

Terminal Display Utilities

The `TerminalDisplay` class provides formatted output to the terminal:

python

```python
class TerminalDisplay:
    """Utility class for formatted terminal output"""

    @staticmethod
    def clear_screen():
        """Clear the terminal screen"""
        os.system('cls' if os.name == 'nt' else 'clear')

    @staticmethod
    def print_status(status, details=""):
        """Print a status message with optional details"""
        timestamp = datetime.now().strftime("%H:%M:%S")
        if status.lower() == "success":
            print(f"{Fore.GREEN}[✓] {timestamp} - {details}")
        elif status.lower() == "error":
            print(f"{Fore.RED}[✗] {timestamp} - {details}")
        # ... other methods
```

This class handles:

- Clearing the screen for dashboard updates
- Printing color-coded status messages
- Displaying formatted tables for market data
- Showing connection status, portfolio information, and positions
- Visualizing recent trades and system events

IB API Integration

The `IBManager` class handles all interactions with Interactive Brokers:

python

```
class IBManager:
    def __init__(self, host='127.0.0.1', port=7497, client_id=1,
config_path=None):
        self.host = host
        self.port = port
        self.client_id = client_id
        self.config_path = config_path
        self.ib = IB()
        self.connected = False
        # ... other initialization
```

Key responsibilities:

- **Connection Management**: Establishes and maintains connection to TWS/Gateway
- **Market Data Handling**: Subscribes to and processes real-time market data
- **Order Execution**: Places and tracks orders for trading strategies
- **Position Tracking**: Monitors current positions and average costs
- **Error Handling**: Processes and logs IB API errors
- **Health Checks**: Periodically verifies connection status
- **Logging**: Records all activities for later analysis

The `connect()` method is particularly important:

python

```
def connect(self):
    """Connect to IB TWS with enhanced logging and retry logic"""
    if not self.connected:
        self.connection_attempts += 1
        try:
            logger.info(f"Connecting to IB TWS at {self.host}:{self.port}
(Attempt {self.connection_attempts})")
            # ... connection logic
            self.ib.connect(self.host, self.port, clientId=self.client_id)
            # ... event handler setup
            self.ib.reqAccountUpdates()  # Request account updates
            # ... start event loop
            return True
        except Exception as e:
            # ... error handling
            return False
    return True
```

This method handles connection attempts with retry logic and detailed logging.

Strategy Implementations

The system implements a base `Strategy` class that defines the interface for all trading strategies:

python

```
class Strategy:
    def __init__(self, name, capital, ib_manager):
        self.name = name
        self.capital = capital
        self.active = False
        self.ib = ib_manager
        # ... other initialization

    def check_signals(self, market_data):
        """Check for trading signals based on market data"""
        # To be implemented by subclasses
        pass

    def execute_trades(self, signals):
        """Execute trades based on signals"""
        # To be implemented by subclasses
        pass
```

The `MicroCrudeOilArbitrageStrategy` extends this base class to implement a statistical arbitrage strategy:

python

```
class MicroCrudeOilArbitrageStrategy(Strategy):
    def __init__(self, name, capital, ib_manager,
                 primary_symbol='MCL', secondary_symbol='USO',
                 z_entry=2.0, z_exit=0.5, lookback=20):
        super().__init__(name, capital, ib_manager)
        # ... strategy-specific initialization
```

Key strategy components:

- **Z-Score Calculation**: Measures the deviation of the spread from its historical mean
- **Signal Generation**: Generates entry and exit signals based on z-score thresholds
- **Position Sizing**: Calculates appropriate position sizes based on capital
- **Trade Execution**: Implements the mechanics of executing trades
- **P&L Tracking**: Calculates and records profit and loss

Portfolio Management

The `Portfolio` class manages a collection of strategies:

python

```
class Portfolio:
    def __init__(self, initial_capital=2000):
        self.initial_capital = initial_capital
        self.current_capital = initial_capital
        self.strategies = {}
        # ... other initialization
```

Its responsibilities include:

- **Strategy Management**: Adding and removing strategies
- **Capital Allocation**: Distributing capital among strategies
- **Performance Tracking**: Monitoring overall portfolio performance
- **Optimization**: Finding optimal capital allocation among strategies

Main Application

The main application logic is in the `run_headless_app()` function:

python

```
def run_headless_app(config_file=None, host='127.0.0.1', port=7497,
client_id=1, initial_capital=2000,
                    log_dir='logs', check_interval=5,
save_logs_interval=3600, display_interval=15):
    # ... initialization

    # Initialize IB Manager
    ib_manager = IBManager(host=host, port=port, client_id=client_id)

    # Initialize Portfolio
    portfolio = Portfolio(initial_capital=initial_capital)

    # Create strategies
    # ... strategy initialization
    # Connect to IB TWS
    # ... connection logic

    # Main loop
    try:
        while True:
            # Check connection
            # Get market data
            # Check for signals and execute trades
            # Update portfolio
            # Save logs
            # Update dashboard
            # Sleep
    except KeyboardInterrupt:
        # ... cleanup
    finally:
        # ... cleanup
```

This function:

1. Initializes the system components
2. Establishes connection to TWS
3. Runs the main trading loop
4. Handles interruptions and cleanup

Installation

Prerequisites

- Python 3.8 or higher
- Interactive Brokers Trader Workstation (TWS) or IB Gateway
- IB account with market data subscriptions for desired contracts

Required Python Packages

apache

```
ib_insync>=0.9.70
pandas>=1.3.0
numpy>=1.20.0
scipy>=1.7.0
colorama>=0.4.4
tabulate>=0.8.9
```

Installation Steps

1. Clone the repository:

   bash

```
git clone https://github.com/yourusername/micro-futures-trading-
system.git
cd micro-futures-trading-system
```

2. Create a virtual environment:

   bash

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. Install dependencies:

   bash

```
pip install -r requirements.txt
```

4. Configure Interactive Brokers TWS:
   - Enable API connections in TWS: File → Global Configuration → API → Settings
   - Set the port number (default: 7497 for paper trading, 7496 for live)
   - Check "Allow connections from localhost only" for security

Configuration

Command-line Arguments

The application accepts several command-line arguments:

Argument	Description	Default
`--host`	IB TWS host address	127.0.0.1
`--port`	IB TWS port number	7497
`--client-id`	IB API client ID	1
`--config`	Path to configuration file	None
`--capital`	Initial capital	2000
`--log-dir`	Directory for log files	logs
`--check-interval`	Signal check frequency (seconds)	5
`--log-interval`	Log saving frequency (seconds)	3600

Argument	Description	Default
`--display-interval`	Dashboard update frequency (seconds)	15

Example Configuration File

You can create a configuration file (JSON format) for persistent settings:

json

```
{
  "connection": {
    "host": "127.0.0.1",
    "port": 7497,
    "client_id": 1
  },
  "trading": {
    "initial_capital": 2000,
    "check_interval": 5,
    "risk_tolerance": 0.5
  },
  "logging": {
    "log_dir": "logs",
    "log_interval": 3600
  },
  "display": {
    "display_interval": 15
  },
  "strategies": {
    "MicroCrudeOilArbitrage": {
      "active": true,
      "allocation": 100,
      "parameters": {
        "primary_symbol": "MCL",
        "secondary_symbol": "USO",
        "z_entry": 2.0,
        "z_exit": 0.5,
        "lookback": 20
      }
    }
  }
}
```

Usage

Basic Usage

1. Start Interactive Brokers TWS or IB Gateway
2. Allow API connections in TWS settings
3. Run the application:

   bash

```
python main.py
```

Advanced Usage

Specify custom parameters:

bash

```
python main.py --port 7496 --capital 5000 --display-interval 30
```

Use a configuration file:

bash

```
python main.py --config config.json
```

Run in production mode:

bash

```
python main.py --host 192.168.1.100 --port 7496 --client-id 2 --log-dir
/var/log/trading
```

Dashboard Navigation

The terminal dashboard automatically updates based on the display interval. It shows:

- Connection status to IB TWS
- Portfolio value and P&L
- Current positions
- Live market data
- Recent trades
- Strategy status and metrics
- System events

Press Ctrl+C to gracefully exit the application, which will close all positions and save logs.

Deployment

Local Deployment

For running on your local machine:

1. Ensure TWS is running and configured to allow API connections

2. Start the application with appropriate parameters
3. Keep the terminal window open while trading

Server Deployment

For running on a dedicated server:

1. Set up IB Gateway on the server (recommended over TWS for server deployment)
2. Configure IB Gateway to start automatically
3. Create a systemd service (Linux) or Windows service to run the application

Example systemd service file (`/etc/systemd/system/trading-system.service`):

ini

```ini
[Unit]
Description=Micro Futures Trading System
After=network.target

[Service]
Type=simple
User=trading
WorkingDirectory=/home/trading/micro-futures-trading-system
ExecStart=/home/trading/micro-futures-trading-system/venv/bin/python main.py --config /home/trading/config.json
Restart=on-failure
RestartSec=5s

[Install]
WantedBy=multi-user.target
```

Enable and start the service:

bash

```bash
sudo systemctl enable trading-system.service
sudo systemctl start trading-system.service
```

Docker Deployment

You can also containerize the application:

1. Create a Dockerfile:

dockerfile

```dockerfile
FROM python:3.9-slim

WORKDIR /app

```

```
 requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

 . .

CMD ["python", "main.py"]
```

2. Build and run the Docker container:

bash

```
docker build -t micro-futures-trading-system .
docker run -it --network="host" micro-futures-trading-system python main.py -
-host host.docker.internal
```

Note: Using `--network="host"` or `host.docker.internal` helps connect to TWS running on the host machine.

Troubleshooting

Common Issues

1. **Connection Issues**
   - Ensure TWS is running and API connections are enabled
   - Verify the port number matches TWS settings
   - Check that the client ID is unique
2. **Market Data Problems**
   - Verify you have the necessary market data subscriptions
   - Check for errors in the logs related to market data
   - Ensure the symbols are correctly specified
3. **Order Execution Failures**
   - Verify you have sufficient funds
   - Check TWS settings for order confirmations
   - Look for specific error codes in the logs

Logs Analysis

The system creates several log files:

- `trading_system.log`: Main application log
- `logs/connection_log_*.json`: Connection events
- `logs/market_data_log_*.json`: Market data events
- `logs/order_execution_log_*.json`: Order execution events

These logs can be analyzed to diagnose issues and monitor system performance.

Contributing

Contributions are welcome! Here's how you can contribute:

1. Fork the repository
2. Create a feature branch: `git checkout -b feature/new-feature`
3. Commit your changes: `git commit -am 'Add new feature'`
4. Push to the branch: `git push origin feature/new-feature`
5. Submit a pull request

License

This project is licensed under the MIT License - see the LICENSE file for details.

Development Roadmap

Future enhancements planned for this system:

1. **Additional Strategies**
   - Implement micro-equity futures strategies
   - Add volatility trading strategies
   - Support for options on futures
2. **Machine Learning Integration**
   - Predictive models for market moves
   - Adaptive parameter optimization
   - Anomaly detection for risk management
3. **Enhanced Reporting**
   - Daily performance reports via email
   - Web dashboard for remote monitoring
   - Trade analysis and strategy evaluation tools
4. **Risk Management**
   - Position sizing optimization
   - Drawdown controls
   - Correlation-based portfolio construction

Contributions in these areas are particularly welcome!