Simulation of Stock Exchange and operations of HFT

Table of Contents:

Introduction

Team Details

Team name: **DigitalDynamos**

Name	Email	Roll Number
Parsania Ramya Parulbhai	ParsaniaRamya.Parulbhai@iiitb.ac.in	IMT2023015
Chaitya Nirmal Shah	chaityanirmal.shah@iiitb.ac.in	IMT2023055
Velidanda Krishna Sai	velidanda.sai@iiitb.ac.in	IMT2023094
Hitanshu Seth	Hitanshu.Seth@iiitb.ac.in	IMT2023100
Aaryan Rajesh Antala	Aaryan.Antala@iiitb.ac.in	IMT2023548
Satyam Ambi	Satyam.Ambi@iiitb.ac.in	IMT2023623

Project Overview:

This project aims to simulate how an HFT (High-Frequency Trading) firm executes its strategies to profit through the stock exchanges. The project also simulates how a stock exchange manages its order book to match and execute orders efficiently using advanced data structures and algorithms to minimise latency.

Purpose:

- The project offers a realistic simulation of an HFT system and order book management, helping users understand how the entire system functions.
- The project uses strategies such as arbitrage, market making, etc., and advanced algorithms and data structures to reduce the latency of managing the order book.

***High-Level Functionality*:**

- Performance analysis: Provides insights into the profits and losses of the firm.
- Simulating strategies of HFT firms.
- Process incoming buy/sell orders and match them based on price and quantity.

***Scope*:**

Included:

- Live management of order books that keep track of buy/sell prices.
- Simulation of basic HFT strategies.
- Implementation of the classes involved in Java, and C++ implementation of the algorithms being used by the HFT firm.

Not included:

- Real-time data from actual financial markets.
- Execution of real trades or financial transactions.

Objectives

The primary objectives of this project are:

- Implement high-performance trading algorithms in C++:
    - Maintain the order book containing thousands of orders.
    - Develop and implement efficient data structures to match, add and delete orders.
    - Based on the order, utilise the appropriate data objects and execute algorithms to identify all profitable trading strategies.
- Build a backend and user interface in Java:
    - Users can see the orders placed by the HFT.
    - Users can see the profit/loss of the HFT.
    - Plotting graphs for various data like profit/loss.
- Use multithreading to let the HFT use multiple strategies when trading to maximise profit
    - Because multiple strategies are implemented for the HFT, it is more optimal for the HFT to apply the appropriate approach to maximise profit.
    - Based on the order book data, the HFT would use the strategy most fit for the scenario or even use multiple strategies.

4. Risk Mitigation:

- Monitor Risk: Track cumulative PnL and trade returns to assess financial performance.
- Control Risk: Enforce trading halts when thresholds for maximum loss or Value at Risk (VaR) are breached.
- Evaluate Performance: Compute key metrics like VaR and Sharpe Ratio to analyze risk-adjusted returns.

- Support Decisions: Provide real-time trading assessments through the isTradingAllowed method.
- Report Metrics: Output cumulative PnL, VaR, and Sharpe Ratio for transparency and analysis.T

- This class ensures disciplined trading and mitigates financial risks effectively.

System Overview

***Technical Specifications*:**

- Frontend: Java (UI)
- Backend: Java (Server-side logic): For implementing HFT strategies and to simulate different cases
- Core Logic: C++ (Trading algorithms and network monitoring): For order matching and efficient data structures

***Input/Output Requirements*:**

Input:

- Buy and sell orders placed by the HFT system.
- Order details: Order ID, Order type (buy/sell), price and quantity

Output:

- Data of Matched orders with executed price, quantity, and timestamp.
- Logs showing how the HFT strategies perform under different market conditions.
- Detailed report on the profits and number of matched orders on the performance of the HFT strategies

Functional Requirements

Detailed Features:

- **Order Book Management**: The system will maintain an order book in real-time containing current bid and ask prices while managing new orders, cancelling and modifying existing orders, and sorting current orders on priority (the highest bid will match the lowest ask).

- **Order Matching Engine**: Orders are matched according to prices (If a bidding price of an order matches or exceeds the ask price the trade is executed) and quantity and then removes the order from the order book and logs the trade.
- **Arbitrage Strategy**: The arbitrage strategy involves exploiting price differences of the same asset across different markets. HFT systems buy the asset at a lower price in one market and sell it at a higher price in another. Thus profiting from the discrepancy.
- **Market-Making Strategy**: The market-making strategy involves continuously placing buy and sell limit orders on a stock to profit from the bid-ask spread. HFT earns small, consistent profits by capturing the difference between the buying and selling prices.

Use Cases:

1. **Arbitrage Trading:** Suppose the HFT system detects a price discrepancy between two stock exchanges. For example:
    - Stock exchange A lists a stock at ₹10.50, and exchange B lists the stock at ₹10.55. The HFT places a buy order at ₹10.50 in exchange A and a sell order for the same stock in exchange B at ₹10.55. Thus, the order-matching engine executes both trades, profiting the HFT from the arbitrage opportunity.
2. **Market Making:**
    - The HFT places a buy order slightly below the current market price and places a sell order slightly above the current market price.
    - As the market prices change the system adjusts these orders to stay close to the market prices.

Development Setup

The development environment for the High-Frequency Trading (HFT) Simulation project is configured to ensure efficient implementation, testing, and debugging. Below are the specifications and tools used for the development setup:

Instructions:

Development Environment:

1. **Operating System**: macOS/Linux/Windows (based on developer preference)

2. Programming Languages:
Java: Used for simulation logic and core API functionalities.
C++: Used for performance-critical matching engine and data structures (e.g., AVL tree for the order book).

3. IDE (Integrated Development Environment):
IntelliJ IDEA: For Java development.
Visual Studio Code: For C++ development.

4. **Build Tools**:

Java:

JDK 17 or higher.
Maven/Gradle (Maven recommended for dependency management).
C++:
GCC (g++) for Linux/macOS.
CMake for managing build dependencies.

Git Commands:

Initial Setup:

1. **Clone the repository: https://github.com/melohub-xbit/HFT-Simulator-DigitalDynamos.git**

2. **Navigate to the project directory**: cd HFT-Simulator-DigitalDynamos

Workflow

Backend:

As soon as the simulation is started by running the HFTSimulation.java file, the random order generation function is started on a thread. It continuously gener-ates new orders based on certain restricted conditions. These orders are then added to the order book using the JNI interface in the respective AVL Trees, i.e buy tree or sell tree. These random generated orders get stored in the AVL Tree on the basis of price of the orders. These orders details are also stored in a file named “e1_orders.txt” and “e2_orders.txt”. Now concurrently with the addition of orders, there are two strategies being run using executor service. The market-making strategy places buy and sell orders around the mid-price of a stock to profit from the bid-ask spread. The spread is dynamically adjusted based on market volatility, ensuring adaptability to price fluctuations. Inventory levels are tracked to avoid over-selling, ensuring logical and realistic trade execution. The arbitrage strategy exploits price discrepancies for the same asset across two exchanges by simultaneously buying low on one and selling high on the other. The algorithm is event-driven, executing trades only when the price spread exceeds transaction costs. Order sizes are fixed, ensuring straightforward execution while maintaining efficiency. The details of the orders which the strategies generate also get stored in the files named “market_making.txt” and “arbritage.txt”.

Next, in one more thread matching of the orders take place, and the trade gets executed between the matched orders from buy tree and sell tree in the exchange’s orderbook.

Finally, as a additional feature we have also implemented a risk management class ensures controlled trading by dynamically tracking inventory levels, allowing sell orders only when sufficient stocks are available. It integrates a stop-loss mechanism to limit losses by exiting trades when the cumulative PnL falls below a predefined threshold. Additionally, a Sharpe Ratio calculation helps assess risk-adjusted performance, preventing the strategy from operating under unfavourable conditions. This risk

Frontend:

The frontend is built on Swing. There are two buttons on the bottom of the GUI named “Start Monitoring”, “Stop Monitoring”, and “Exit”. As soon as start monitoring is clicked, the simulation gets started. The buy order logs and the sell order logs of the trade executed are shown side by side on the frontend GUI. Furthermore, there is a graph which shows the profit trend in real time, the increase in profit by a trade executed is shown by a green line, while loss is indicated by a red line. The real time profit of the HFT is also shown in the GUI at the bottom. Lastly, when Stop Monitoring button is pressed, the simulation gets paused. To continue, the simulation once can again press “Start Simulation”. Finally, clicking on “Exit”terminates the trading and exits the program.

Important Files & Folders

The following is the structure of the folders:

1. **cpp** : This folder has two sub folders, JNI and Core. The Core folder consists of C++ source files which contain the implementation of the AVLTree, OrderBook, and class Order. The order book class in cpp contains the methods which will be called by java through JNI. The JNI folder contains the jni header file and the implementation of the JNI functions. The important files in the Core folder are :AVLTree.cpp, OrderBook.cpp and Order.cpp.

2. Java: This directory contains the Java source files for the project, organised into several sub-directories:

- **exchange**: Contains the Exchange and OrderBook classes, which handle the exchange-related functionality.
- **gui**: Contains the MatchedOrdersGUI class, which is responsible for the graphical user interface.
- **rml**: Contains the RiskManagement class, which handles the risk management aspects of the project.
- **strategy**: Contains the various trading strategy classes, such as ArbitrageStrategy, MarketMakingStrategy, TradingStrategy, and HFTSimulation.

3. RandomOrderGeneration.java: A class that generates random orders for the simulation.

4. **HFTSimulation.java:** This is the main file which runs the simulation, it creates instances of exchanges, and uses threading to run the following processes: AddingOrders, Running the two strategies and Matching the orders.

How to run the Program:

In Linux based systems:

1. Go to the root directory of the project and run the following command in the terminal **-**

“g++ -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/darwin" -shared -o libOrderBookNative.dylib cpp/jni/OrderBookNative.cpp cpp/core/OrderBook.cpp cpp/core/AVLTree.cpp cpp/core/Order.cpp”

This command is used to compile and create a shared library (.dylib file) from C++ source files using g++, the GNU C++ compiler, for use in a Java Native Interface (JNI) integration.

2. Go to the /java folder and run : **“javac *.java”** . This will compile all the java files.

3. Run the following command to start the program:

“java -Djava.library.path=. HFTSimulation”

In Windows:

1. g++ -c -fPIC -I"C:\Program Files\Java\jdk-23\include" -I"C:\Program Files\Java\jdk-23\include\win32" cpp/core/OrderBook.cpp cpp/core/AVLTree.cpp cpp/core/Order.cpp cpp/jni/OrderBookNative.cpp

2. g++ -shared OrderBook.o AVLTree.o Order.o OrderBookNative.o -o OrderBookNative.dll

3. javac -h . java/exchange/OrderBook.java

4. cd java

5. javac *.java

6. java -Djava.library.path="{your_path_to_root_directory_of_project}" -cp java HFTSimulation

Note: Replace the address and java version with address and version in your system.

Conclusion

The **High-Frequency Trading (HFT) Simulator** successfully serves as platform for understanding and analysing the complexities of modern financial markets. By using advanced data structures like AVL trees for order book management and leveraging the synergy of C++ for performance-critical components and Java for interface-level operations, the simulator provides both accuracy and efficiency.

This tool offers a practical environment for testing algorithms, studying market dynamics, and simulating real-time trading scenarios. It equips users with insights into the challenges of high-frequency trading, such as latency, order matching, and market impact, while fostering innovation in strategy development.

With its modular design and extensibility, the HFT Simulator is a significant step toward bridging the gap between theoretical knowledge and practical application in the field of algorithmic trading. It lays a solid foundation for future improvements, like integrating machine learning models for predictive analytics and including live market data for real-world testing.

Additional Features Included:

Risk Management Functionality:It integrates a stop-loss mechanism to limit losses by exiting trades when the cumulative PnL falls below a predefined threshold.

Additionally, a Sharpe Ratio calculation helps assess risk-adjusted performance, preventing the strategy from operating under unfavourable conditions.

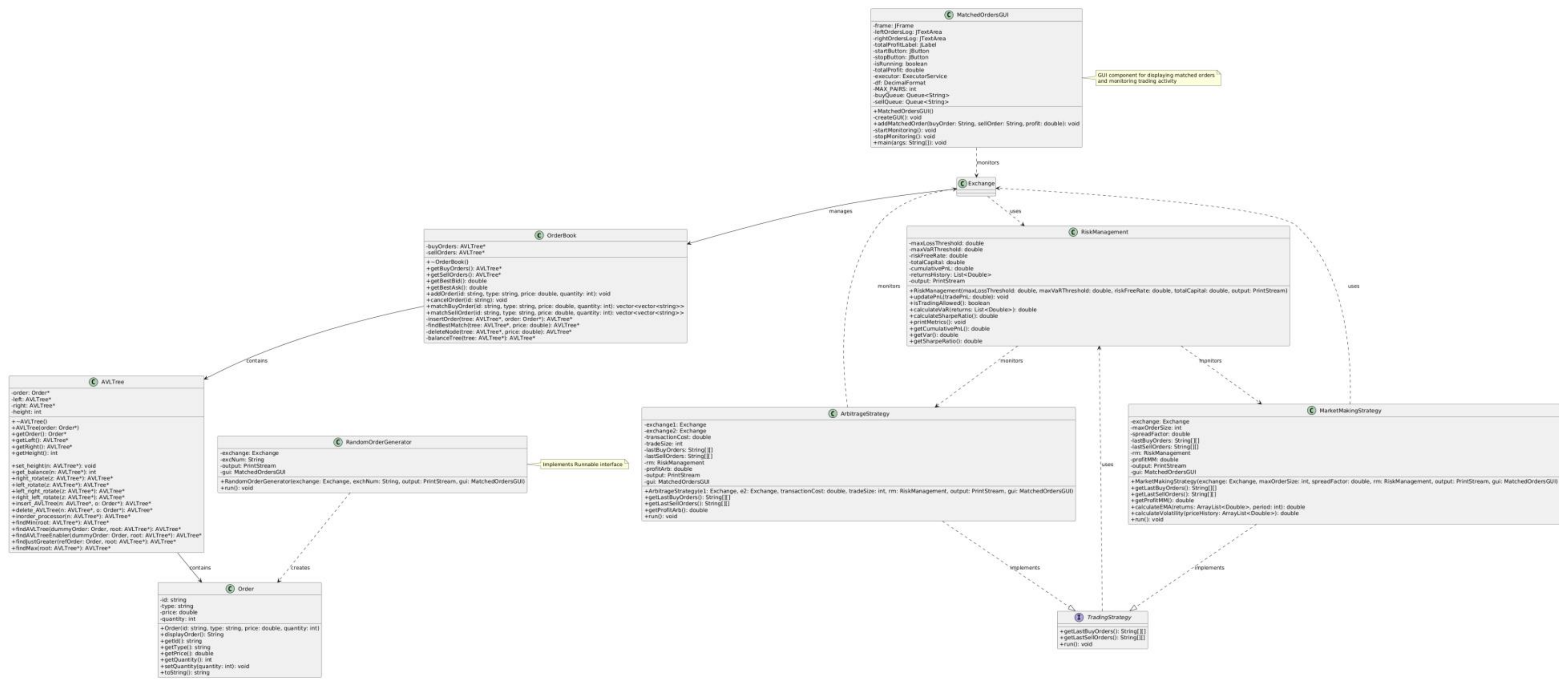
Optional Features

- Network Monitoring: Detects network anomalies and potential security threats.
    - Set up a network monitoring system that monitors the network traffic between the HFT and the Exchange, and analyzes the order execution time, and other performance metrics.
    - Based on a predefined set of threshold values for the metrics, it is determined whether or not the order is feasible.

- Order Entry:
    - The user enters the order details, and following this, the order is added to the order book.
    - If any sell orders match then the system executes the trade updating the order book and logs the trade.

UML Diagram

-> UML diagram on Next Page


MatchedOrdersGUI
-frame: JFrame
-leftOrdersLog: JTextArea
-rightOrdersLog: JTextArea
-totalProfitLabel: JLabel
-startButton: JButton
-stopButton: JButton
-isRunning: boolean
-totalProfit: double
-executor: ExecutorService
-df: DecimalFormat
-MAX_PAIRS: int
-buyQueue: Queue<String>
-sellQueue: Queue<String>
+MatchedOrdersGUI()
-createGUI(): void
+addMatchedOrder(buyOrder: String, sellOrder: String, profit: double): void
-startMonitoring(): void
-stopMonitoring(): void
+main(args: String[]): void
GUI component for displaying matched orders and monitoring trading activity
monitors
Exchange
manages
uses
monitors
uses
OrderBook
-buyOrders: AVLTree*
-sellOrders: AVLTree*
+~OrderBook()
+getBuyOrders(): AVLTree*
+getSellOrders(): AVLTree*
+getBestBid(): double
+getBestAsk(): double
+addOrder(id: string, type: string, price: double, quantity: int): void
+cancelOrder(id: string): void
+matchBuyOrder(id: string, type: string, price: double, quantity: int): vector<vector<string>>
+matchSellOrder(id: string, type: string, price: double, quantity: int): vector<vector<string>>
-insertOrder(tree: AVLTree*, order: Order*): AVLTree*
-findBestMatch(tree: AVLTree*, price: double): AVLTree*
-deleteNode(tree: AVLTree*, price: double): AVLTree*
-balanceTree(tree: AVLTree*): AVLTree*
RiskManagement
-maxLossThreshold: double
-maxVaRThreshold: double
-riskFreeRate: double
-totalCapital: double
-cumulativePnL: double
-returnsHistory: List<Double>
-output: PrintStream
+RiskManagement(maxLossThreshold: double, maxVaRThreshold: double, riskFreeRate: double, totalCapital: double, output: PrintStream)
+updatePnL(tradePnL: double): void
+isTradingAllowed(): boolean
+calculateVaR(returns: List<Double>): double
+calculateSharpeRatio(): double
+printMetrics(): void
+getCumulativePnL(): double
+getVar(): double
+getSharpeRatio(): double
contains
monitors
monitors
AVLTree
-order: Order*
-left: AVLTree*
-right: AVLTree*
-height: int
+~AVLTree()
+AVLTree(order: Order*)
+getOrder(): Order*
+getLeft(): AVLTree*
+getRight(): AVLTree*
+getHeight(): int
+set_height(n: AVLTree*): void
+get_balance(n: AVLTree*): int
+right_rotate(z: AVLTree*): AVLTree*
+left_rotate(z: AVLTree*): AVLTree*
+left_right_rotate(z: AVLTree*): AVLTree*
+right_left_rotate(z: AVLTree*): AVLTree*
+insert_AVLTree(n: AVLTree*, o: Order*): AVLTree*
+delete_AVLTree(n: AVLTree*, o: Order*): AVLTree*
+inorder_processor(n: AVLTree*): AVLTree*
+findMin(root: AVLTree*): AVLTree*
+findAVLTree(dummyOrder: Order, root: AVLTree*): AVLTree*
+findAVLTreeEnabler(dummyOrder: Order, root: AVLTree*): AVLTree*
+findJustGreater(refOrder: Order, root: AVLTree*): AVLTree*
+findMax(root: AVLTree*): AVLTree*
RandomOrderGenerator
-exchange: Exchange
-excNum: String
-output: PrintStream
-gui: MatchedOrdersGUI
+RandomOrderGenerator(exchange: Exchange, exchNum: String, output: PrintStream, gui: MatchedOrdersGUI)
+run(): void
Implements Runnable interface
ArbitrageStrategy
-exchange1: Exchange
-exchange2: Exchange
-transactionCost: double
-tradeSize: int
-lastBuyOrders: String[][]
-lastSellOrders: String[][]
-rm: RiskManagement
-profitArb: double
-output: PrintStream
-gui: MatchedOrdersGUI
+ArbitrageStrategy(e1: Exchange, e2: Exchange, transactionCost: double, tradeSize: int, rm: RiskManagement, output: PrintStream, gui: MatchedOrdersGUI)
+getLastBuyOrders(): String[][]
+getLastSellOrders(): String[][]
+getProfitArb(): double
+run(): void
uses
MarketMakingStrategy
-exchange: Exchange
-maxOrderSize: int
-spreadFactor: double
-lastBuyOrders: String[][]
-lastSellOrders: String[][]
-rm: RiskManagement
-profitMM: double
-output: PrintStream
-gui: MatchedOrdersGUI
+MarketMakingStrategy(exchange: Exchange, maxOrderSize: int, spreadFactor: double, rm: RiskManagement, output: PrintStream, gui: MatchedOrdersGUI)
+getLastBuyOrders(): String[][]
+getLastSellOrders(): String[][]
+getProfitMM(): double
+calculateEMA(returns: ArrayList<Double>, period: int): double
+calculateVolatility(priceHistory: ArrayList<Double>): double
+run(): void
contains
creates
implements
implements
Order
-id: string
-type: string
-price: double
-quantity: int
+Order(id: string, type: string, price: double, quantity: int)
+displayOrder(): String
+getId(): string
+getType(): string
+getPrice(): double
+getQuantity(): int
+setQuantity(quantity: int): void
+toString(): string
TradingStrategy
+getLastBuyOrders(): String[][]
+getLastSellOrders(): String[][]
+run(): void