Project: Order Matching Engine

Tutorial

Team Members

- Swetha Shanmugam (UNI: ss6357)
- Zhejian Jin (UNI:zj2324)
- Fenglei Gu (UNI: fg2546)

Introduction

This project has two components - a Parser that parses ITCH data and an Order Book.

Parser

Download Nasdaq ITCH data

Nasdaq ITCH is chosen for the prototype product of our project. It has sample data on the official website emi.nasdaq.com - /ITCH/Nasdaq ITCH. To run the parser for the Nasdaq ITCH data, we need to first download the sample data from the website.

How to use the parser?

To use the parser, you only need to specify the input file path and output log file path in the main function, and then call the constructor and then call `BookBuilder::start()` function.

```
BookBuilder builder(file_path, outputMessageCSV);
builder.start();
```

Besides, we need to comment on the updateBook() function in the `BookBuilder::next().`

```
void BookBuilder::next(){
   message = message_reader.createMessage();
   if(!message.isEmpty()){
       bool validMessage = updateMessage();
       if(validMessage){
           // updateBook();
       }
   }
}
```

How to integrate with the order book?

To integrate with the orderbook, the only thing we need to modify is to comment on the updateBook() function in the `BookBuilder::next().`

```
void BookBuilder::next(){
   message = message_reader.createMessage();
   if(!message.isEmpty()){
       bool validMessage = updateMessage();
       if(validMessage){
           updateBook();
       }
   }
}
```

Sample output:

```
Welcome to the Nasdaq ITCH order matching engine
---------------------start-----------------------
Opened ../ExchangeDataViewer/data/03272019.PSX_ITCH50 to read ITCH 5.0.
messages.
Opened ./data/test.log for writing.
Begin building book and matching orders
Processed 10Million messages. 2 Mio messages per sec.
Processed 20Million messages. 2 Mio messages per sec.
Processed 30Million messages. 2 Mio messages per sec.
Processed 40Million messages. 1.73 Mio messages per sec.
Processed 50Million messages. 1.78 Mio messages per sec.
Processed 60Million messages. 1.71 Mio messages per sec.
---------------------end-----------------------
Finish building book and matching orders in 28seconds.
Total Add Order is 557615 and Total Delete Order is 523969
File ./data/test.log has been closed.
File ../ExchangeDataViewer/data/03272019.PSX_ITCH50 has been closed
Finished, processed 61784531 messages in 28seconds.
```

What is an order?

Before diving into the details of an order book, it is necessary to understand the different types of orders. An order is a command to buy/sell stocks of a particular company. There are multiple types of orders like market orders, limit orders, etc (Ref: wiki). However, we are going to focus on 4 types of orders that we support:

- Market Orders - These are buy/sell orders that are executed immediately at the current market price.
- Limit Orders - These are buy/sell orders that are executed at a particular price or better.
- Stop Orders - Orders that are converted to a Market Order when a stop price is met.
- Stop-Limit Orders - Orders that are converted to a Limit Order when a stop price is met.

For more information, please refer to
https://www.investor.gov/introduction-investing/investing-basics/how-stock-markets-work/types-orders

Create an Order

An order object consists of the following fields:

`unsigned order_id` - The order ID of the order
`unsigned owner_id` - Owner of the order
`unsigned quantity` - The quantity of the stock
`unsigned quote` - The price/quote of the order
`unsigned stop_price` - The stop_price in case of a STOP/STOP_LIMIT order
`OrderSide order_side` - The order side (BUY/SELL)
`OrderType order_type` - The order type (MARKET/LIMIT/STOP/STOP_LIMIT)
`bool all_or_none` - Indicates if the order has to be executed entirely or not. By default it is false.
`std::chrono::time_point<std::chrono::system_clock> timestamp` - timestamp of added order

Given below are few examples for creating different types of orders:

```
// A Buy Limit order with price = 999 and qty = 10
Order buy(3,2,999,10,OrderSide::BUY,OrderType::LIMIT,0);
// A Sell Market order with qty = 15
Order sell(4,2,0,15,OrderSide::SELL,OrderType::MARKET,0);
// A Stop Limit Sell order with qty = 15, stop_price = 700 and price=700
Order stop_sell1(9,2,700,700,15,OrderSide::SELL,OrderType::STOP_LIMIT,0);
// A Stop Sell order with qty = 15, stop_price = 700
Order stop_sell2(19,2,0,700,15,OrderSide::SELL,OrderType::STOP,0);
```

Creating an order book

We can create an order book for a stock (e.g: APPLE) in the following way:

```
#include "OrderMatcher/central_order_book.hh"

CentralOrderBook book;
std::string s = "APPLE";
book.add_symbol(s);
```

This way, we can create order books for multiple stocks.

Once the order book for the stocks are created, we can perform operations like adding an order, deleting an order, and fetching the best bid/ask at a particular time.

Let us look at examples of performing each of the above operations:

Adding an order

`StatusCode add_order(string symbol, Order& order)`

The method takes the symbol and order to add as input.
Let us add two orders - a Buy Limit order and a Sell Market order to the order book we just created.

```
Order buy(3,2,999,10,OrderSide::BUY,OrderType::LIMIT,0);
Order sell(4,2,0,15,OrderSide::SELL,OrderType::MARKET,0);
StatusCode s1 = book.add_order(s,buy);
StatusCode s2 = book.add_order(s,sell);
```

The method returns the status of the operation. Please refer to the manual for possible status codes.
When orders are added to the order book, they are automatically matched internally. The matched orders are written to a <symbol> file in the runtime directory. The format of matched result written in the file is “matched_order1;matched_order2;price;qty”.

Fetch an order

`std::optional<Order> get_order(unsigned int order_id)`

The method fetches an order of a particular order ID from the order book.
Let us fetch an order with order ID = 3 from the order book the following way:

```
auto sell_order_obj = book.get_order(3);
Order order = *sell_order_obj;
```

Find the best bid/buy price

`std::pair<StatusCode, unsigned> best_bid(string symbol) const`

This function returns the best bid/buy price for a particular stock symbol. This is the maximum buy price at that particular time.

Let us add two buy orders and check what is the best bid.

```
Order buy(3,2,999,10,OrderSide::BUY,OrderType::LIMIT,0);
Order buy1(4,2,990,10,OrderSide::BUY,OrderType::LIMIT,0);
book.add_order(s,buy);
book.add_order(s,buy1);
std::pair<StatusCode, unsigned> best_bid = book.best_bid(s);
Std::cout << best_bid.second; // 999
```

The best bid is 999 since that is the highest bid order.

Find the best ask/sell price

`std::pair<StatusCode, unsigned> best_ask(string symbol) const`

This function returns the best ask/sell price for a particular stock symbol. This is the minimum sell price at that particular time.

Let us add two sell orders and check what is the best ask.

```
Order sell(3,2,999,10,OrderSide::SELL,OrderType::LIMIT,0);
Order sell1(4,2,990,10,OrderSide::SELL,OrderType::LIMIT,0);
book.add_order(s,sell);
book.add_order(s,sell1);
std::pair<StatusCode, unsigned> best_ask = book.best_ask(s);
```

```
Std::cout << best_ask.second; // 990
```

The best ask is 990 since that is the lowest ask order.

Delete an order

`std::optional<Order> delete_order(unsigned int order_id)`

The method deletes an order of a particular order ID from the order book.
Let us delete an order with order ID = 3 from the order book the following way:

```
book.delete_order(3);
```

Build

To build the project run ./build.sh in the main directory.