Project: Order Matching Engine

Design Document

Team Members

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

Source Code: https://github.com/FungluiKoo/Order-Matching-Engine

Parser

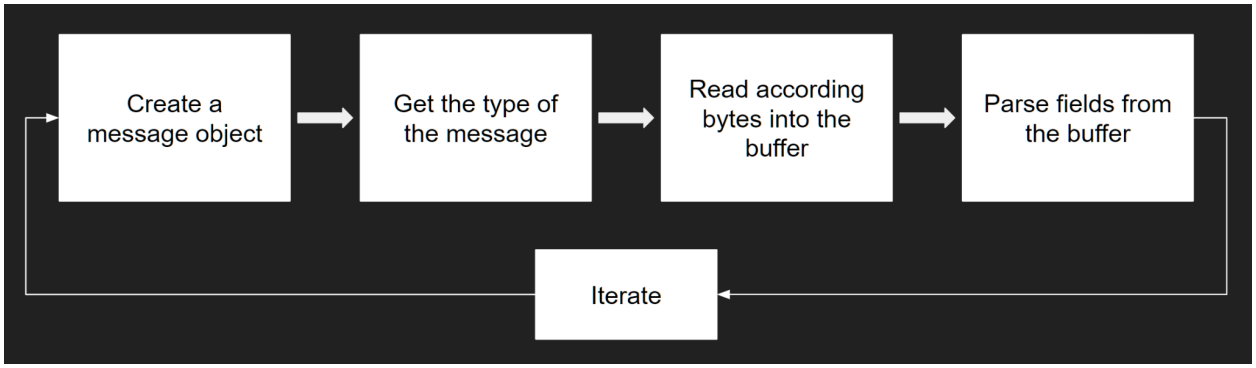
Data Source

Nasdaq ITCH is chosen for the prototype product of our project. It has sample data on the official website, also it is widely used for companies doing electronic trading in real life. The ITCH Protocol uses SBE (Simple Binary Encoding) protocol for low latency and deterministic performance. It is optimized for low latency of encoding and decoding while keeping bandwidth utilization reasonably small. All integer fields are big endian (network byte order) binary encoded numbers.

For the parser we have adapted the parser at https://github.com/martinobdl/ITCH and modified it to add/delete orders in our order book.

Reader

The Reader class reads bytes from the stream and parses messages.
The following flow diagram shows how the data is parsed in Reader class.



BookBuilder

The BookBuilder class is used for building central order books and matching orders. It provides direct APIs to the users. Here are the members and functions in the BookBuilder:

Members:

```
Message message; // object for parsed message
CentralOrderBook centralBook; // object for central OrderBook
Reader message_reader; // object for reader that is used for parsing
Writer parserWriter;  // object for writer that writes parsed results
time_t totalTime;    // used for record total time taken
std::vector<std::string> SymbolFilters =
    { "AAPL", "MSFT", "TSLA", "AMZN"}; // used for filtering the messages
```

The private members include many objects from other classes. The BookBuilder encapsulates each resource into the class by using the RAII technique.

Functions:

```
BookBuilder(const std::string &inputMessagePath,
            const std::string &outputMessageCSV); // constructor
~BookBuilder(); // destructor
void start();  // start of the building central order books and matching
orders process
void next();  // thet function that iterates through the data file to find
the next message and
void updateBook(); // the function that builds central order books and
matches orders
```

The above functions provide direct APIs to the users. A user will only call the `BookBuilder::start()` function in the main file to start book building and order matching.

Central Order Book

The CentralOrderBook maintains the order book for multiple stocks. The order books are stored in an unordered map. Given below is the data structure.

```
std::unordered_map<string, OrderBook> order_book_map;
```

Functions:

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

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

StatusCode delete_order(unsigned int order_id);

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

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

The above functions in turn call the corresponding function of the relevant order book. E.g: on adding a new order for symbol ‘APPLE’, the function add_order(order) will be invoked for Apple’s orderbook.

Details of the above functions can be found in the manual.

Order Book

The OrderBook class maintains the order book for a particular stock symbol. E.g: APPLE and GOOGLE would have separate order books.

Main Data Structures

```
// key=price level; value=a list of Order
std::unordered_map<unsigned, std::list<Order>> buypool, sellpool,
stop_buy_pool, stop_sell_pool;
```

```
// stores current levels of the hashmaps (sellpool and stop_buy_pool)
std::set<unsigned, std::less<unsigned>> sellprices, stop_buy_prices;
// stores current levels of the hashmaps (buypool and stop_sell_pool)
std::set<unsigned, std::greater<unsigned>> buyprices, stop_sell_prices;
```

Split std::map into std::set and std::unordered_map

We know that accessing std::map takes O(log n) time while accessing std::unordered_map takes O(1) only. However, we want to know the largest/smallest key (price level) in O(1). Hence, we split std::map into two structures:

- A std::set keeping the keys (price levels)
- A std::unordered_map keeping the key-value pairs.

Below we analyze their expected time complexity for add_order and delete_order operations. Without loss of generality, assume the order.type is buy, and that there are P price levels, M orders per price level.

For AddOrder(order):

If buypool.count(order.quote)==0: // O(1) on average, O(P) worst-case
buyprices.insert(order.quote) // O(log P) per call. Each order has 1/M possibility of being a new price. Thus, average **O(1/M * log P)**
Access buypool[order.quote] takes O(1) on average, O(P) worst-case
Insert a new order into buypool[order.quote] takes O(1) on average. (Linear search from the end, it's highly likely that the new order will be finally located at or close to the end)
Therefore, on average, addOrder takes **O(1/M * log P)** on average.

If we use a single std::map instead of the hashmap:
Access buypool[order.quote] takes **O(log P)**.
Insert the new order into the still takes O(1) as above.
The total complexity is **O(log P)**, which is worse unless M is O(1).

For DeleteOrder(order):

Get ordertype and price level from the hashmap findID takes O(1) on average.
// Assume the order type is buy
Access buypool[price] takes O(1) on average, O(P) worst-case
Finding the position of the order in the list buypool[price] takes **O(M)** - this is the same complexity as using order_map, since we still need to find the position within the list (the address of the order is not the same)
Erase the item based on the position takes O(1),
Check if the list is empty takes O(1)

If the list is empty:

buypool.erase(price) takes O(1) on avg, O(P) worst-case

buyprices.erase(price) takes O(log P) per call. Each order has 1/M possibility of being the last order of that price. Thus, average **O(1/M * log P)**

Therefore, on average, delete order takes **O(M + 1/M * log P)**.,O(P+M+1/M*log P) on worst-case

If we use std::map instead of hashmap:
Access buypool[price] takes **O(log P)**
Finding the position of the order in the list takes **O(M)** as above.
Erase the item takes O(1) as above.
Check if the list is empty takes O(1) as above.
If the list is empty:

buypool.erase(price) takes O(log P) per call. Each order has 1/M possibility of being the last order of that price. Thus, average O(1/M * log P), the same as above.

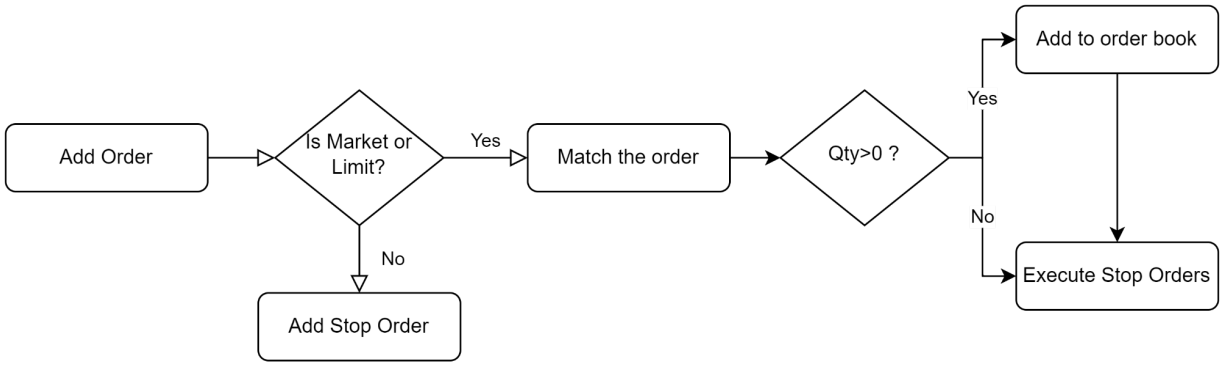
The total complexity is **O(M + log P)**, which is worse unless M is O(1).

In conclusion, splitting std::map into std::unordered_map and std::set does have better expected time complexity.

Functions

```
StatusCode add_order(Order&);
```

The function adds an order to the order book. Given below is the flow diagram of the add_order function.

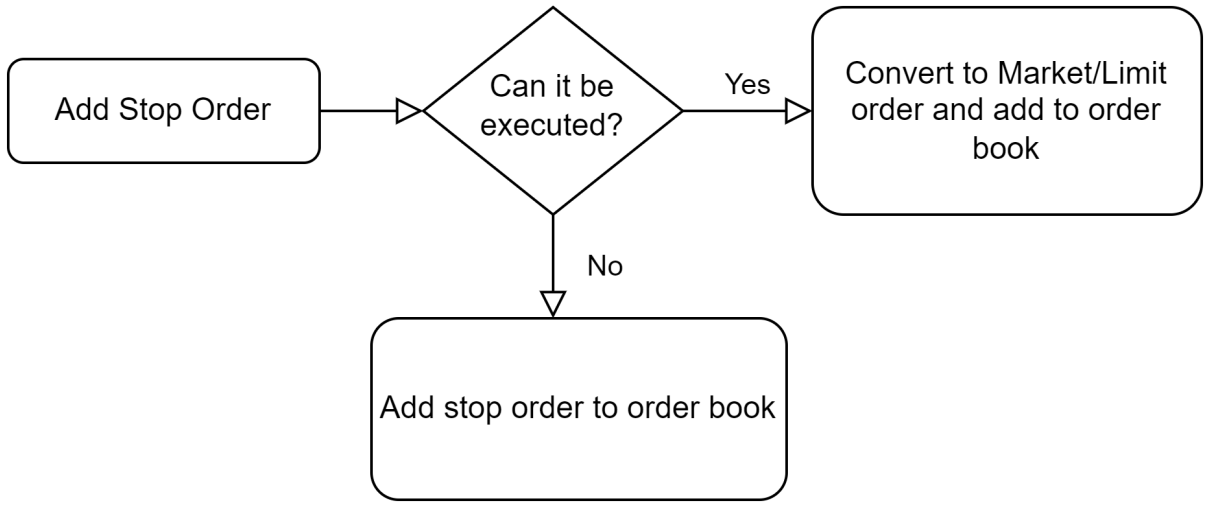


It internally uses the add_to_orderbook function to update the order pool. Given below is the signature of the add_to_orderbook function.

```
template<typename Comp>
StatusCode add_to_orderbook(Order& order, unsigned level,
std::set<unsigned, Comp>& prices, std::unordered_map<unsigned,
std::list<Order>>& pool)
```

The function is generalised to work for all types of orders. Market and Limit orders are stored in the buypool/sellpool and their price levels are stored in buyprices/sellprices. Stop and Stop-Limit orders are stored in the stop_buy_pool/stop_sell_pool and their price levels are stored in stop_buy_prices/stop_sell_prices.

Given below is the code flow of the add_stop_order function.



Every time an order is added, we check if we can execute stop orders. This is done by the following function:

```
template<typename Pred, typename Comp>
void execute_stop_orders(unsigned stop_price, std::set<unsigned,Comp>&
prices,std::unordered_map<unsigned, std::list<Order>>& order_pool, Pred p){
for (auto f = prices.begin(); f != prices.end();) {
        if(!p(*f, stop_price))
            break;
         // activate all stop orders at price level *f
}
}
```

The above function is invoked to activate stop-buy and stop-sell orders the following way:

```
 //if stop price at a level <= market price, stop order is activated
    auto buy_pred = [](unsigned stop_price, unsigned sell_market_price) {
     return stop_price <= sell_market_price;
    };
    //execute stop buy orders if possible
    execute_stop_orders(get_sell_market_price(), stop_buy_prices,
stop_buy_pool, buy_pred);

    //if stop price at a level >= market price, stop order is activated
    auto sell_pred = [](unsigned stop_price, unsigned buy_market_price) {
     return stop_price >= buy_market_price;
    };
    //execute stop sell orders if possible
    execute_stop_orders(get_buy_market_price(), stop_sell_prices,
stop_sell_pool, sell_pred);
```

All the stop orders satisfying the predicate are activated.

Other functions supported:

```
std::optional<Order> get_order(unsigned int);
StatusCode delete_order(unsigned int);
unsigned best_ask()const;
unsigned best_bid()const;
```

Matching

When a new order comes, we match it with the (previously unmatched) orders in our pool. Let's say that a new buy order comes. If its quote is higher than the lowest sell price in our pool, then we can match them. However, if the new buy order is Fill or Kill (FOK), meaning that the buyer would rather kill (cancel) the order if the order can not be fully filled immediately, we might need to drop the new order and not match anything. This happens when there are not enough sell orders satisfying (lower than) the quote price. The buyer might have set too low a bidding price, or have set too high a quantity. This suggests that when you want to take over a company you need to be prepared to pay a higher price - which is also why stock prices would rise before takeovers.

Testing

The order book has been tested by running different scenarios using Google Test. Apart from that, some amount of manual testing has been done.

References:

- https://github.com/martinobdl/ITCH
- enewhuis/liquibook: Modern C++ order matching engine (github.com)
- chronoxor/CppTrader: High performance components for building Trading Platform such as ultra fast matching engine, order book processor (github.com)
- jamesdbrock/hffix: Financial Information Exchange Protocol C++ Library (github.com)
- (63) How traders orders get matched (exchange matching algorithms) - YouTube
- CME Globex Matching Algorithm Steps - Electronic Platform Information Console - Confluence (cmegroup.com)