Flow Traders
Graduate Software Development Program
C++ Case Study

Welcome to the Flow Traders C++ Graduate 2021 case study.

Case study guidelines:

- Submit your solution as a part of a local git repository, under a branch called **flow/graduates/2021**;
- Write simple, efficient and readable code;
- You have 36 hours to submit your solution.

How to succeed

- Produce code that is GCC or CLANG compatible (socket implementations based on Win API will not be accepted);
- Use C++ 17, its standard containers and the std algorithms as much as possible;
- Have fun.

Vocabulary

- Financial instrument: everything that can be bought or sold over a financial exchange;
- Order: intention of a trader to buy or sell a financial instrument;
- Trade: Deal between two parties, e.g. I buy a quantity X from you at price Y;
- Net Position/Position: Quantity of a specific financial instrument that I own or owe to someone. It is a positive number when I own a financial instrument, it is a negative number when I owe a financial instrument to someone.

Overview

As every good trading company we need to keep an eye on our risk. Your goal is to write the Risk server that will help us to remain in business.

The risk server will receive order and trades, after every order an answer is expected, the risk server can either accept or reject the order.

Requirements

1. The application has to be a TCP server;
2. The application will calculate the hypothetical worst net position for every trader, further details below;
3. The application compare the hypothetical worst net position for the incoming order and compare it with a threshold;
4. If the hypothetical worst net position is greater than the threshold the application has to reject the order;
5. When an order is rejected the state of the system does not change;
6. When an order is accepted the state of the system is updated, a new hypothetical worst net position is calculated;
7. When a trader get disconnected all the information relative to his/her position get discarded;
8. Every trade with a positive quantity is considered a Long trade;
9. Every trade with a negative quantity is considered a Short trade;
10. Every message contains 1 header and 1 message only, check the "Messages specification" section.

Hypothetical worst net position calculation

For every financial instrument involved you need to keep track of the current worst possible position. The worst possible position is evaluated as follows:

For every financial instrument:

- Do a net sum of all the received trades quantity (AKA NetPos)
- Do a net sum of all the buy orders quantity (AKA BuyQty)
- Do a net sum of all the sell orders quantity (AKA SellQty)
- Calculate the Buy side as max(BuyQty, NetPosition + BuyQty)
- Calculate the Sell side as max(SellQty, SellQty - NetPosition)

Input

Data	Source	Description
Orders	Socket	Orders operation: Add Modify Cancel
Trade confirmation	Socket	The exchange sent you a trade confirmation, this means that an order has been matched on the exchange side.

Data	Source	Description
Max Buy position	command line argument	Max buy position threshold to apply, the same threshold is applied for every financial instrument
Max Sell position	command line argument	Max sell position threshold to apply, the same threshold is applied for every financial instrument

Output

After each order the application should respond with an output message:

Data	Destination	Description
Response	Socket	The response can be: Yes this order can be sent to the exchange No this order cannot be sent to the exchange

Example

System configuration:

- Buy threshold: 20
- Sell threshold: 15

	Flow					Other Stock					
Message	**Net Pos**	**Buy Qty**	**Sell Qty**	**Buy Hypothetical worst**	**Sell Hypothetical worst**	**Net Pos**	**Buy Qty**	**Sell Qty**	**Buy Hypothetical worst**	**Sell Hypothetical worst**	**Transaction Accepted**
Message: New order ID: 1 Instrument: Flow Price: 10 Qty: 10 Side: Buy	0	10	0	10	0	0	0	0	0	0	Yes
Message: New order	0	10	0	10	0	0	0	15	0	15	Yes

ID: 2 Instrument: Other Stock Price: 1.5 Qty: 15 Side: Sell											
Message: New order ID: 3 Instrument: Other Stock Price: 1.5 Qty: 4 Side: Buy	0	10	0	10	0	0	4	15	4	15	Yes
Message: New order ID: 4 Instrument: Other Stock Price: 1.5 Qty: 20 Side: Buy	0	10	0	10	0	0	4	15	4	15	No
Message: Trade Instrument: Other Stock Price: 1.5 Qty: -4 Side: N/A	0	10	0	10	0	-4	4	15	4	19	N/A
Message: Order Delete ID: 3 Instrument: Other Stock	0	10	0	10	0	-4	0	15	0	19	N/A

Messages specification

Socket message binary format

Bytes																		
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**...**	**N**
Header																Message		

Header

The input protocol that you are going to handle has different properties such as:

- Protocol version, a different version might have different message formats;
- Sequence number, it is used to check the consistency of your data flow.

These extra information can be used to verify the consistency of your incoming data.

```
struct Header
{
    uint16_t version; // Protocol version
    uint16_t payloadSize; // Payload size in bytes
    uint32_t sequenceNumber; // Sequence number for this package
    uint64_t timestamp; // Timestamp, number of nanoseconds from Unix epoch.
} __attribute__ ((__packed__));
static_assert(sizeof(Header) == 16, "The Header size is not correct");
```

New order

```
struct NewOrder
{
    static constexpr uint16_t MESSAGE_TYPE = 1;

    uint16_t messageType; // Message type of this message
    uint64_t listingId; // Financial instrument id associated to this message
    uint64_t orderId; // Order id used for further order changes
    uint64_t orderQuantity; // Order quantity
    uint64_t orderPrice; // Order price, the price contains 4 implicit decimals
    char side; // The side of the order, 'B' for buy and 'S' for sell
} __attribute__ ((__packed__));
static_assert(sizeof(NewOrder) == 35, "The NewOrder size is not correct");
```

Delete order

```
struct DeleteOrder
{
    static constexpr uint16_t MESSAGE_TYPE = 2;

    uint16_t messageType; // Message type of this message
    uint64_t orderId; // Order id that refers to the original order id
} __attribute__ ((__packed__));
static_assert(sizeof(DeleteOrder) == 10, "The DeleteOrder size is not correct");
```

Modify Order Quantity

```
struct ModifyOrderQuantity
{
    static constexpr uint16_t MESSAGE_TYPE = 3;

    uint16_t messageType; // Message type of this message
    uint64_t orderId; // Order id that refers to the original order id
    uint64_t newQuantity; // The new quantity
} __attribute__ ((__packed__));
static_assert(sizeof(ModifyOrderQuantity) == 18, "The ModifyOrderQuantity size is not correc
```

Trade

```
struct Trade
{
    static constexpr uint16_t MESSAGE_TYPE = 4;

    uint16_t messageType; // Message type of this message
    uint64_t listingId; // Financial instrument id associated to this message
    uint64_t tradeId; // Order id that refers to the original order id
    uint64_t tradeQuantity; // Trade quantity
    uint64_t tradePrice; // Trade price, the price contains 4 implicit decimals
} __attribute__ ((__packed__));
static_assert(sizeof(Trade) == 34, "The Trade size is not correct");
```

Order response

```
struct OrderResponse
{
    static constexpr uint16_t MESSAGE_TYPE = 5;

    enum class Status : uint16_t
    {
        ACCEPTED = 0,
       REJECTED = 1,
    };

    uint16_t messageType; // Message type of this message
    uint64_t orderId; // Order id that refers to the original order id
    Status status; // Status of the order
} __attribute__ ((__packed__));
static_assert(sizeof(OrderResponse) == 12, "The OrderResponse size is not correct");
```