Scalability: Exchange Matching

Engineering Robust Server Software
Homework 4

For this assignment you will be writing an exchange matching engine – that is, a piece of software which will match buy and sell orders for a stock/commodities market. For this homework, you may write in any language (or combination of languages) you wish.

Here are some terms that we will use in the rest of the assignment description:

Symbol - A symbol identifies a particular stock or commodity. A symbol will be a sequence of 1 or more alphanumeric characters. For example, `SPY` is a symbol (for the S&P 500 index), as is `BTC` (for bitcoin). For our exchange, `T5asdf` would also be a valid symbol.

Position - A position is a symbol and the amount of that symbol owned. The amount may be fractional. If the amount is positive, the position is called "long" (meaning you own the stock). While real trading also allows the amount to be negative—which is called "short"—we disallow this behavior. A short position arises when you sell stock that you do not own (called a short sale). Short sales are not allowed in our market.

Account - An account is identified by a globally unique account number (sequence of one or more base-10 digits). An account contains, a current balance (in US dollars) and a (possibly empty) set of positions. The account balance must never become negative: a transaction which would cause a negative account balance cannot be processed. Likewise, any short sale is also rejected.

Order - An order is a request to buy or sell. An order contains the symbol to be traded, the amount to purchase (a negative amount means to sell), and a limit price. This price is the maximum price at which the symbol may be purchased (for buy order), or the minimum price at which the symbol may be sold (for sell orders). When a Buy order is placed, the total cost (shares * price) is deducted from the buyer's account (if insufficient funds are available, the order is rejected). When a sell order executes (see below), the seller's account is credited for the sale price. When a sell order is placed, the shares are deducted from the seller's account (if insufficient shares are available, the order is rejected). When a buy order executes, the buyer's account is credited the shares of the symbol being purchased.

Open Order - When an order is first placed, it is "open." This state basically means that you need to find the matching order to finish this transaction.

Match Orders - A buy order can be matched with a sell order, when (a) both orders are open (b) they are for the same symbol (c) their limit prices are compatible. "Compatible prices" means that there exists a price which satisfies the limit restrictions (as described above) of both orders. That is, the sell order's limit price is lower than the buy order's limit price. When two orders match, they may both be executed.

Executing an Order - Executing an order means adding money to the seller's account, creating a new position in the buyer's account (if they did not already have one for this symbol), and adjusting the number of shares in both the buyer's account to reflect the purchase their positions. As part of execution, the order's status changes from "open" to "executed". Note that the execution of both matching orders MUST be done atomically (including all parts: changing the seller's account balance, changing buyer's number of shares, and altering order status). When an order executes, a price must be determined (such that seller's limit price ≤ execution price ≤ buyer's limit price). We will define the execution price to be the price of the order that was open first – that is the one that has been open and waiting for a match.

Canceling an Order - A user may cancel an open order. Once an order is canceled, it MUST NOT match with any other orders and MUST NOT execute. Canceling a Buy order refunds the purchase price to the buyer's account. Canceling a Sell order returns the shares to the seller's account.

You will write a server to match orders and track accounts. Here the requirements for your server:

1. When your server is first started, it has
   (a) No symbols
   (b) No accounts
   (c) No orders

2. Your server will listen for incoming connections on port 12345.

3. When your server receives a connection, it will receive one line containing a base-10 unsigned integer, then immediately after the newline XML describing commands to perform. The integer on the first line is the length of the XML data which follows it (in bytes). The top-level node in the XML will either be `<create>` or `<transactions>` depending on what the client wants to do. **Your server must handle this exact format (including the line with the integer message size, as this will be used in the graded testing infrastructure.**

4. If the top-level node in the XML is `<create>` then the it contains 0 or more of the following children:

   a) `<account id="ACCOUNT_ID" balance="BALANCE"/>` This creates a new account with the given unique ID and balance (in USD). The account has no positions. Attempting to create an account that already exists is an error.

   b) `<symbol sym="SYM">` This creates the specified symbol. The symbol tag can have one or more children which are `<account id="ACCOUNT_ID">NUM</account>` These indicate that NUM shares of the symbol being created should be placed into the account with the given ID. Note that this creation is legal even if `sym` already exists: in such a case, it is used to create more shares of that symbol and add them to existing accounts.

Note that the children of a create MUST be processed in the order they appear in the input. This means that if an account is created, and then later used, it must exist. For example:

```
173
<?xml version="1.0" encoding="UTF-8"?>
<create>
  <account id="123456" balance="1000"/>
  <symbol sym="SPY">
    <account id="123456">100000</account>
  </symbol>
</create>
```

When the server processes a `<create>` input, it will reply with XML whose root node is `<results>` and whose children are either `<created>` or `<error>`, indicating the success or failure of each operation (in the same order they appeared in the input). Both the `created` and `error` MUST have the account id (for account creation) or sym and account id (for symbol creation) attributes matching the ones in the input. Additionally, the error tag should have a textual body describing what was wrong – e.g.,

```
<error id="456234">Account already exists</error>
```

5. If the top-level tag is `<transactions>` then

   a) The root tag MUST have an attribute identifying the account on which to perform the transactions (and the account must be valid).

   b) The root tag MUST have one or more children, each of which is an <order> tag, a <cancel> tag, or a <query> tag.

   c) The server MAY process the children nodes in any order.

   d) The <order> tag has three attributes: sym, amount, and limit. For example

      <order sym="SPY" amount="100" limit="145.67"/>

      Recall that a negative amount means sell.

The server will respond with XML whose root node is `<results>`. Its children will either be `<opened>` or `<error>`. Both of these will contain the same sym, amount, and limit as the requested transaction, and appear in the same order as the transactions in the input. Additionally, the `opened` will have an `id` attribute, which will provide a unique ID (made by the server) which can be used to query or cancel the transaction in the future.

The `error` tag SHOULD have a textual description of the problem. Note that if the account ID is invalid, then an `<error>` will be reported for each transaction. For orders

which purchase a symbol, if the server accepts the order, it MUST atomically open the order and reduce the balance in the buyer's account. If insufficient funds are available, the server MUST reject the request.

6. The `<query>` tag has one attribute: `id` which is the unique transaction ID of a previously created transaction. The server will reply with `<status>` which will have the `id` attribute identifying the transaction. The `<status>` tag will have one or more children: `<open shares=.../>`, `<canceled shares=... time=...>`, and/or `<executed shares=... price=... time=.../>`, indicating the status of the transaction. Note that open and canceled are mutually exclusive. As we will discuss later, a transaction might be partially executed (e.g., the user wants to buy 100 shares, but only 80 shares could be purchase so far), in which case an `executed` would show for the 80 shares bought, and `open` for the 20 remaining shares. The server MUST NOT reply with more than one `open` to a particular query. Likewise, an order might also have been partially executed, then canceled, in which case it would reply with an appropriate number of `executed` tags and one `canceled` tag. In the case of `time`, the value will be in seconds since the epoch.

7. The `<cancel>` tag has one attribute: `id` which is the unique transaction ID of a previously created transaction. This requests cancellation of any open portion of the order. Any part of the order which has already executed is unaffected. If cancellation is successful, the server replies with `<canceled>` whose children are the same as those of `<status>` above (except that `open` is not valid), indicating the new status of this transaction id. Canceling a buy order refunds the buyer's account for the cost of the canceled purchase price. This refund MUST be done atomically with cancellation of the order.

When your server matches orders, it MUST (a) give the best price match, and (b) break ties by giving priority to orders that arrived earlier. Matching an order may require splitting one order into parts, in which case the time/priority of the unfulfilled part remains the same as when the original order was created.

For example, suppose your server receives the following orders for commodity `X`

```
1: Buy 300 X @ $125
2: Sell 100 X @ $130
3: Buy 200 X @ $127
4: Sell 500 X @ $128
5: Sell 200 X @ $140
6: Buy 400 X @ $125
```

At this point, no orders have matched (and thus all orders are "open") as no price satisfies both a buy and a sell order. This means that your server has the following state:

Buy	Sell
	5: 200 @ $140
	2: 100 @ $130
	4: 500 @ $128
3: 200 @ $127	
1: 300 @ $125	
6: 400 @ $125	

Note that the orders which will be matched first are the ones in the middle. If a new order (ID=7) to sell 400 shares at $124 were to come into the system, it would first match with order 3, which would purchase 200 shares. The remaining 200 shares of this new order would then match with order 1. Since order 1 is to buy 300 shares, 100 shares would remain in order 1. When the orders match, they execute at the price of the order that was already open on the books. After matching order 7, you end up with the following

Buy	Sell
	5: 200 @ $140
	2: 100 @ $130
	4: 500 @ $128
1: 100 @ $125	
6: 400 @ $125	

Executed	
3: 200 @ $127	7: 200 @ $127
1: 200 @ $125	7: 200 @ $125

Note that order 1 has been split. If you `<query>` it, you will see 100 shares open, and 200 shares executed at $125. Such as:

```
<status id="1">
  <open shares="100"/>
  <executed shares="200" price="125" time="1519348326"/>
</status>
```

Likewise, if you `query 7`, you will see that it is fully executed, but was split into two transactions:

```
<status id="7">
  <executed shares="200" price="127" time="1519348325"/>
  <executed shares="200" price="125" time="1519348326"/>
</status>
```

Your deliverables for this assignment are as follows:

- Your server code, setup such that the TAs can run it in docker-compose
- A writeup, including graphs in which you analyze the **scalability** of your server. For this assignment, scalability refers to CPU core count. You will be load testing and analyzing how the throughput of your server changes as it runs across different numbers of cores. Please use good experimental methodology in creating these graphs (run the experiment multiple times, draw error bars, etc). The writeup must be in PDF format and named `report.pdf` inside a sub-directory called `writeup`.
- Your testing infrastructure, which you used to test your server (both for functionality and scalability). Your TAs should be able to reproduce your results (within the variance of normal experimental noise) from the test infrastructure you provide. This testing infrastructure should be in a sub-directory called `testing`.

Appendix A:

In order to help clarify and make it unambiguous about which XML response belongs to each request, the following request / response summary is provided. Comments are shown starting with "#".

```
==========================================
CREATE
```

Requests:

```
<create>
  <account id="ACCOUNT_ID" balance="BALANCE"/> #0 or more
  <symbol sym="SYM"> #0 or more
    <account id="ACCOUNT_ID">NUM</account> #1 or more
  </symbol>
</create>
```

Responses:

```
<results>
  <created id="ACCOUNT_ID"/> #For account create
  <created sym="SYM" id="ACCOUNT_ID"/> #For symbol create
  <error id="ACCOUNT_ID">Msg</error> #For account create error
    <error sym="SYM" id="ACCOUNT_ID">Msg</error> #For symbol create
error
</results>
```

* Note all "created" and "error" tags appear in same order as corresponding "account" or "symbol+account" tags in the request

==========

TRANSACTIONS

Requests:

```
<transactions id="ACCOUNT_ID"> #contains 1 or more of the below children
  <order sym="SYM" amount="AMT" limit="LMT"/>
  <query id="TRANS_ID">
  <cancel id="TRANS_ID">
</transactions>
```

Responses:

```
<results>
  <opened sym="SYM" amount="AMT" limit="LMT" id="TRANS_ID"/>
  <error sym="SYM" amount="AMT" limit="LMT">Message</error>
     #Note there is 1 error for every order/query/cancel of the transactions tag if ID is invalid

  <status id="TRANS_ID">
    <open shares=.../>
    <canceled shares=... time=.../>
    <executed shares=... price=... time=.../>
  </status>

  <canceled id="TRANS_ID">
    <canceled shares=... time=.../>
    <executed shares=... price=... time=.../>
  </canceled>
</results>
```

* Note all `opened` or `status` or `canceled` or `error` tags appear in same order as corresponding `order` or `query` or `cancel` tags in the request.