Scalability: Exchange Matching

- Author: Yuepeng Li (yl737), Meihong Ge (mg476)
- Date: 30th, March 2021
- Course: ECE 568, Engineering Robust Server Software
- Term: 2021 spring
- Professor: Brian Rogers

Duke Community Standard, Affirmation

I affirm that each submission complies with the Duke Community Standard and the guidelines set forth for this assignment.

Abstraction

In this assignment, we implement an exchange machine engine which can match buy and sell orders for a stock/commodities market. Users can create accounts, positions open and cancel orders. The opened order will be automatically stored in a database and been processed spontaneous if here exists any matched orders. This engine can handle multi-threads concurrently by using a thread pool. We also conduct functionality and scalability tests to demonstrate the performance of this machine, which will be described in details later. The implementation is written in C++ and use PostgreSQL as the database.

Functionality

The exchange machine engine can create accounts, positions and orders The opened order can be cancelled. The creation and operation polices are as followed:

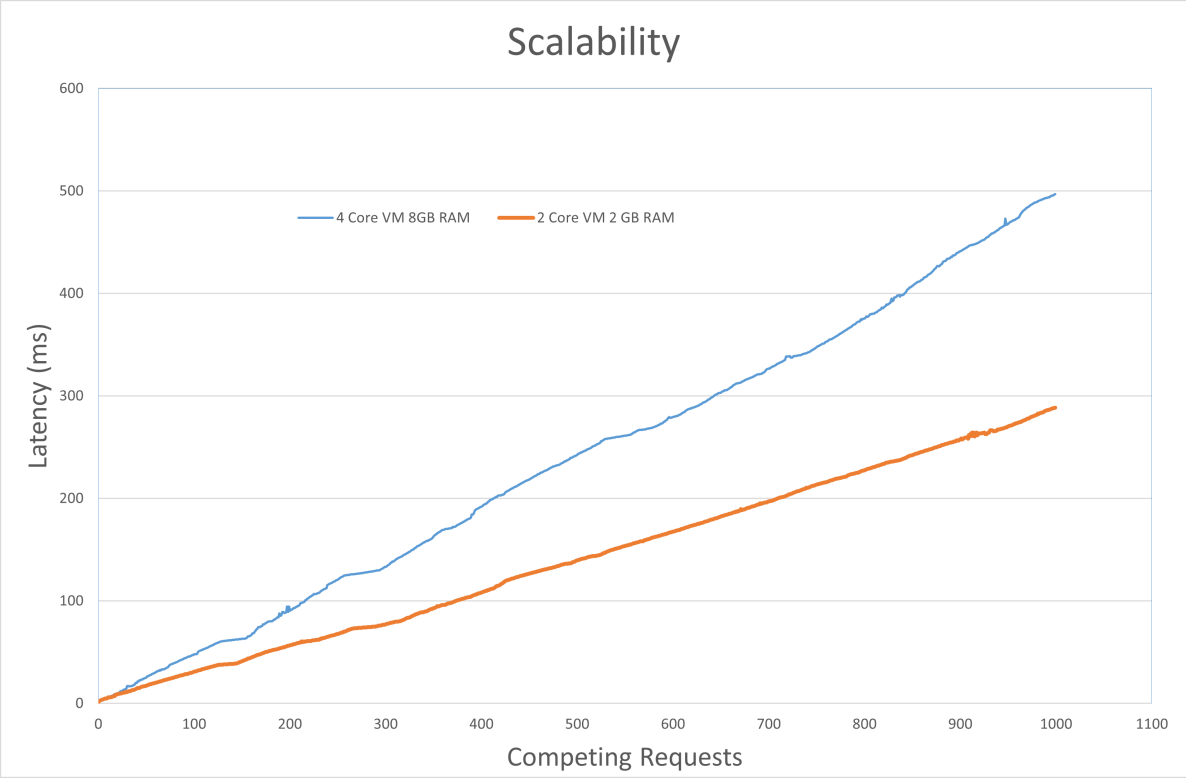
1. One account can only be created once. If an account is created for multi-times, the final account and its balance will be the amount for the first time creation.
2. Positions can be created for several times and the amount of the same position creation will accumulate. But you cannot create a position within a non-existent account.
3. An order will be executed automatically if there exists matched buy/sell order. If not, it will be stored in the database.
4. An order can be executed partially.
5. An order can be cancelled, and the user will get the information of that order including the open and executed part.

In order to test the functionality, we imitated the example given in the pdf and created several xml file that would carry out these steps, all of which we have test case covered:

1. Adding account and positions (creating a existing account is a error case, creating a existing position will add to the old one).
2. Adding orders (creating orders deduct corresponding amount of money or shares depending the type of order `Buy/Sell`).
3. Carry out the transaction (this step automatically execute after the previous step, but remain idle is no doable transactions; if the balance carried in the order is superfluous, it is returned to the order owner).
4. Do query or cancel (query will return all the transaction marked by the given account id and order id, even if the order is split when pairing; cancel will delete the transaction that is not done, and return the order detail if the transaction already happened)

Scalability

Since the engine is designed for multi-thread, we create 1000 requests per test, and for each request, it creates a child thread. The server will handle these threads concurrently. Since the maximum connection of a PostgreSQL database is 100, so we add an thread pool with the size of 50, which means only 50 threads can be processed at the same time. To measure the performance, we calculate the run time for each thread. The following figures show the time cost and throughput for 1000 requests per run on two different virtual machines: 4 core and 2 core. All the data are calculated from the average of ten time's tests.



VM System	Total Requests	Total Latency	Throughput
2 Core	1000	237.8s	4.2 requests per second
4 Core	1000	140.7	7.1 requests per second

From the Scalability figure we can find that as the number of requests increases, the latency for each request response is getting longer. This is because we only allow 50 threads compute concurrently, so the later requests have to wait until the previous requests finished. Another thing worth mentioning is that for a 4 Core VM, the performance will be better with shorter latency since it can conduct a request faster. This can also be found from the throughput data, which shows that the 4 core system has a larger throughput.