Duke University

ECE568 Assignment 4 Report

Group Member:
Ruihan Xu(rx29)
Chunyan Jiang(cj162)

1. Project Overview and Basic Testing

In this project, we have built an exchange matching machine using python. The matching machine fulfills most of the basic requirements. The server is composed of 4 files, including server.py, creation.py, transaction.py and tables.py. The database we used is Postgresql and the ORM is SQLAlchemy. The server.py is mainly focused on socket connection and database connection. The tables.py defines several classes that is used to create, delete and edit table in sql. After receiving an incoming request, the xml.etree.ElementTree will be used to parse the XML. Transact and create defined in the other two files will be called.

Additionally, there is a file called drop_db.py. This file is used to drop all the tables within the database and create new empty tables. The file will save some time in testing.

Apart from those files, there is a testing folder contains all the testing files. Since this project is mainly concerned about the scalability, I will briefly talk about the basic functionality tests here.

To test the basic requirements, first run *python3 drop_db.py* to clear up the database. Then run *python3 server.py* to start the server. Using another VM or terminal to run *python3 testing_basic.py*.

The testing file will create two users with a balance of 1000000 and three positions. The received XML is

```
------------create users-------------------
157
<?xml version='1.0' encoding='utf8'?>
<result><created id="0" /><created id="0" sym="SPY" /><created id="0"
sym="XYZ" /><created id="0" sym="ABC" /></result>
------------create users-------------------
157
<?xml version='1.0' encoding='utf8'?>
<result><created id="1" /><created id="1" sym="SPY" /><created id="1"
sym="XYZ" /><created id="1" sym="ABC" /></result>
```

Then, the testing file will create orders. User1 will want to sell 100 shares of SPY and User2 will want to buy 100 shares.

```
------------create sell order-------------------
112
<?xml version='1.0' encoding='utf8'?>
<results><opened amount="100" id="1" limit="123.45" sym="SPY" /></results>
------------create buy order-------------------
111
<?xml version='1.0' encoding='utf8'?>
<results><opened amount="100" id="2" limit="150.0" sym="SPY" /></results>
```

Now, let us query the two orders. Since they are perfectly matched, the XML appears to be executed.

```
---------------query the two orders-----------------
252
<?xml version='1.0' encoding='utf8'?>
<results><status id="1"><executed price="123.45" shares="100" time="2019-04-
03 20:11:54.676936" /></status><status id="2"><executed price="150.0"
shares="100" time="2019-04-03 20:11:54.684803" /></status></results>
```

The next step is to create a third order. After that, we will try to cancel order one and order three. Ordered 1 will not be canceled since it is already executed. The XML should be:

```
----------------create a third order--------------------
113
<?xml version='1.0' encoding='utf8'?>
<results><opened amount="1000" id="3" limit="123.45" sym="SPY" /></results>
------cancal the first and third order------
291
<?xml version='1.0' encoding='utf8'?>
<results><status id="3"><open shares="1000" /></status><canceled
id="3"><canceled share="1000" time="2019-04-03 20:11:54.707270"
/></canceled><canceled id="1"><executed price="123.45" share="100"
time="2019-04-03 20:11:54.676936" /></canceled></results>
```

Lastly, query the third order again to see if the status has changed.

```
------query the third order------
141
<?xml version='1.0' encoding='utf8'?>
<results><status id="3"><canceled shares="1000" time="2019-04-03
20:11:54.707270" /></status></results>
```

Therefore, our server meets the basic requirements.

2. Scalability Analysis

Now we want to dig into the Scalability of our code. To test this, we need to massively load our matching machine with requests under different running mode, i.e. with different cores, single thread vs multithread ,etc.

First, let us take a look at servers running with different cores. The testing_creation.py was used to test the case. Client running testing_creation.py will generate a number of users according to the command line arguments. Each user will have 3 positions and 3 orders. Fig.1 shows the time consumed to finish creating a specific no. of users with different cores.

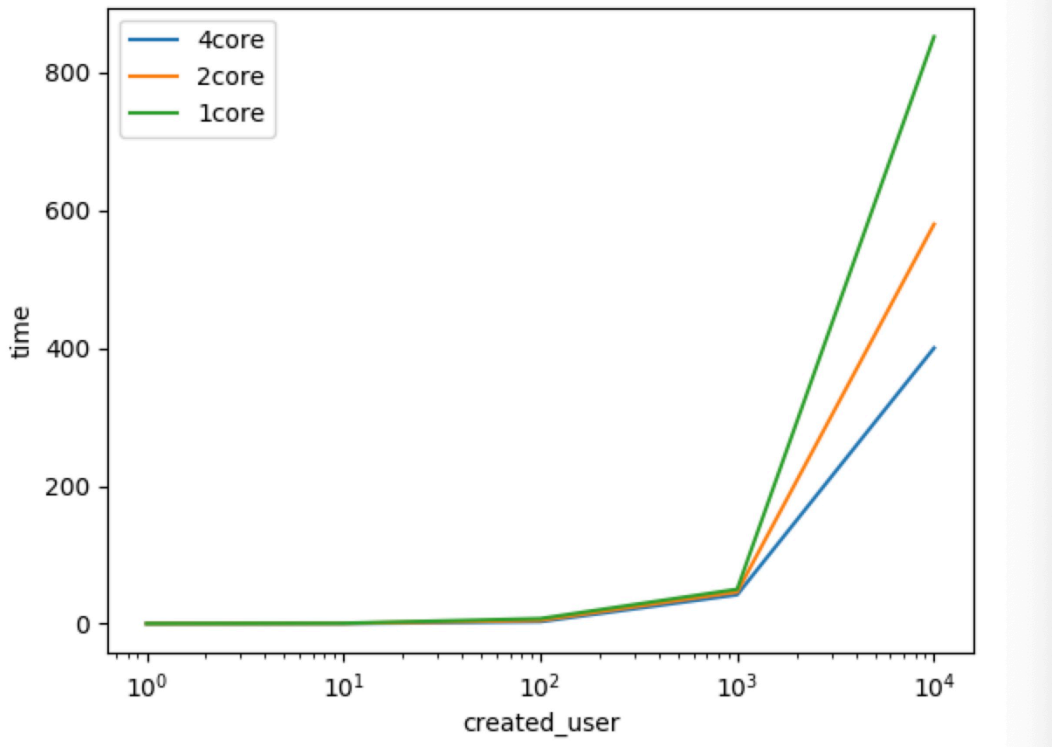

Figure 1

From the figure, it seems that the difference in using different no. of cores is not very significant. However, this is the x-axis used log-scale. Looking into the data, which you can find in plotTest.py, it is pretty clear that given a fixed number of created users, the time consumed using 1 core is approximately 1.5 times of the time consumed using 2 cores. Similarly, the time consumed using 2 cores is approximately 1.5 times of the time consumed using 4 cores. Therefore, we can give the conclusion that with different cores, the performance of our server is largely impacted.

Now let us move on to the multithread part. This part is tested using testing_creation_threading.py, which performs the same operation as testing_creation.py. The only difference is that instead of setting up one TCP connection and send multiple requests, each request will have its own TCP connection. This way, the server can choose to deal with the incoming connections in a multithreaded way. We will compare this with what

we have done previously. The result is shown in fig.2.

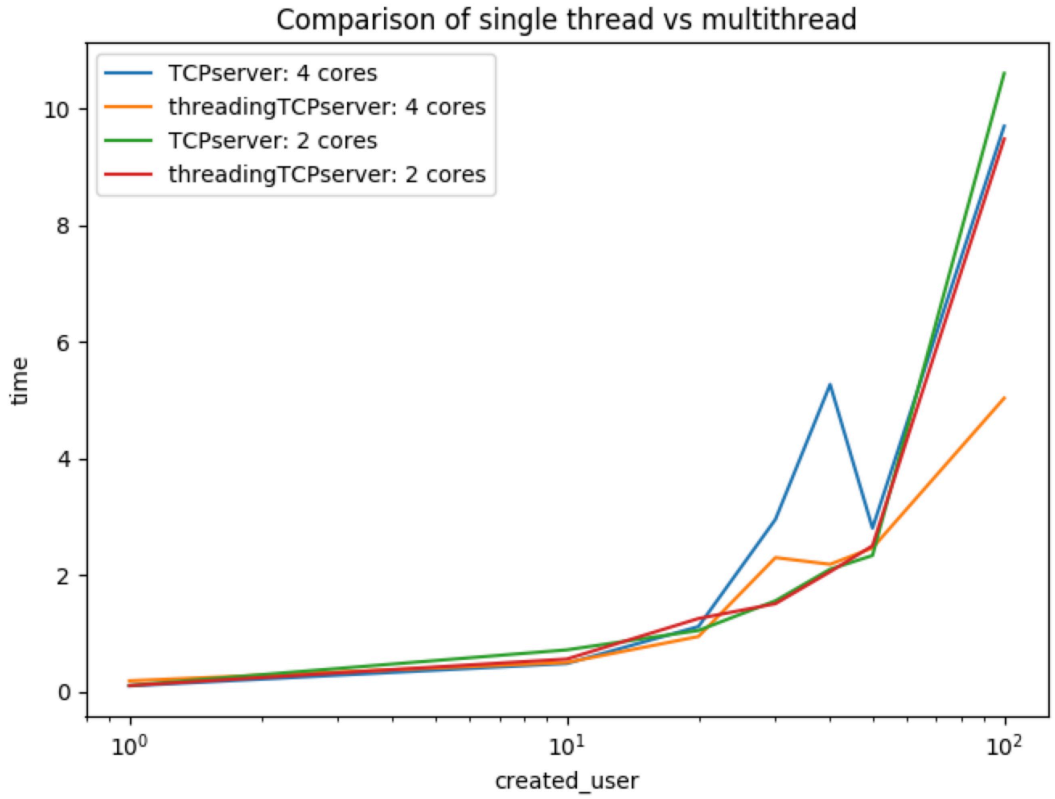

Figure 2

From the figure and the testing, we can see that the resulting time consumed has high variance and the difference between different method are not significant. There exist many reasons why multi-threaded version is not performing better than single-threaded version. The first reason is the overhead of multi-threading. As we can see, each thread only has several interactions with the database. Creating a thread is simply an overkill. Another reason is the implementation of ThreadingTCPServer. It will try to synchronize every request which may cause performance degradation.

Focusing on multithreading, I decided to create a large database and send lots of requests. Testing_massive_create.py creates 10,000 users and 30,000 orders. Testing_query_serial.py sends 4000 requests through one connection to the server. Within each request, 10 queries will be performed to increase the load of each request. Testing_query_multithread.py will be

used to send 4000 requests through 16 connections, that is 250 connections each. First, run testing_query_serial.py and we get the result:

```
rx29@vcm-8327:~/erss-hwk4-cj162-rx29$ python3 testing_query_serial.py
The execution time for 0 is 35.6512732505798 34
```

It took the server 35.65 seconds to complete all the requests. Now, run testing_query_multithread.py:

```
rx29@vcm-8327:~/erss-hwk4-cj162-rx29$ python3 testing_query_multithread.py
The execution time for 6 is 35.79532432556152
The execution time for 5 is 35.86113238334656
The execution time for 2 is 35.90615487098694
The execution time for 7 is 35.97006297111511
The execution time for 1 is 35.98412084579468
The execution time for 2 is 36.09527373313904
The execution time for 1 is 36.251718044281006
The execution time for 4 is 36.29578137397766
The execution time for 3 is 36.36610746383667
The execution time for 5 is 35.547199726104736
The execution time for 3 is 35.57682967185974
The execution time for 8 is 36.64049291610718
The execution time for 4 is 35.666696071624756
The execution time for 8 is 35.71616768836975
The execution time for 6 is 35.774094581604004
The execution time for 7 is 35.77673149108887
total time: 36.79736375808716
```

The expected result is to complete the whole process in around 2 seconds. However, the total time remains the same for each thread even though the no. of requests is much smaller.

After having this result, I believe that that something was wrong in ThreadingTCPServer. To prove my idea, I programmed the server_v2.py using a different approach. Instead of

ThreadingTCPServer, server_v2.py utilized basic threading, which guarantees that each incoming request will be handled in different threads. The result is shown here:

```
rx29@vcm-8327:~/erss-hwk4-cj162-rx29$ python3 testing_query_serial.py
The execution time for 0 is 35.425191164016724
rx29@vcm-8327:~/erss-hwk4-cj162-rx29$ python3 testing_query_multithread.py
The execution time for 3 is 35.92656850814819
The execution time for 6 is 36.074703216552734
The execution time for 1 is 36.16062927246094
The execution time for 2 is 36.16195487976074
The execution time for 8 is 36.54750156402588
The execution time for 3 is 36.63172006607056
The execution time for 2 is 36.70083570480347
The execution time for 4 is 36.74649906158447
The execution time for 6 is 36.80591559410095
The execution time for 1 is 36.82346725463867
The execution time for 8 is 36.85885524749756
The execution time for 4 is 36.880398750305176
The execution time for 5 is 36.90280747413635
The execution time for 7 is 36.91306757926941
The execution time for 5 is 36.94278836250305
The execution time for 7 is 36.94230318069458
total time: 36.951221227645874
```

To my surprise, the result remains the same. This means that the multi-threading part is correct but we still got a serialized result. I created test_query_multi_sleep.py and test_query_multi_nonsleep.py. In the sleep version, I add time.sleep() to separate the threads. The result turns out to be:

```
rx29@vcm-8327:~/erss-hwk4-cj162-rx29$ python3 testing_query_mulit_sleep.py
The execution time for 1 is 0.949333667755127
The execution time for 2 is 0.9093050956726074
The execution time for 3 is 0.9840290546417236
The execution time for 4 is 0.8969969749450684
^[[Ctotal time: 40.03016948699951
rx29@vcm-8327:~/erss-hwk4-cj162-rx29$ python3 testing_query_multi_nonsleep.py
The execution time for 2 is 3.9361133575439453
The execution time for 1 is 3.9483604431152344
The execution time for 3 is 3.986682415008545
The execution time for 4 is 4.099680662155151
total time: 4.103444576263428
```

It is very clear that even if threads are running together, the actual execution of the requests are still in a serial order. The database implementation has a high possibility to this result. But for the query part, the only related operation is filter(). Due to the time constrain, I do not have more time to dig deeper into this problem.