ECE568 HW3 Scalability Analysis

Yifan Shao (ys319) & Qin Sun (qs33)

1 Testing Infrastructure

The testing infrastructure is available in the folder `Client`.

1.1 `RandomActionGenerator.java`

`RandomActionGenerator` is a test case provider.

- `createAccount` generates an XML-format request to create a new account, set its balance and add 3 different symbols to it.
- `getRandomAction` creates an XML-format request to create a new transaction, which is either a selling or a buying order.

1.2 `Client.java`

`Client` prompts for the testing IP, port and the testing scale (which is the number of test cases). Then it runs designated number of clients, sending requests and waiting for responses and records the time at the same time. When the process ends, the time in `ms` will be shown on the screen.

2 Test Results & Analysis

2.1 Test Procedures

2.1.1 Test for 50% CPU usage

To ensure that the test results will not be influenced by network connection, we run the server and client on the same VM.

For each test, we need to restart the server in order to clear the database. Otherwise, the testing will no longer be accurate.

1. Run `cgcreate -g cpu:/test` to create a group for testing.
2. Limit the CPU usage of our server to 50% by running

   ```
   cgset -r cpu.cfs_period_us=100000 test cgset -r cpu.cfs_quota_us=50000 test
   ```

3. Run `cgexec -g cpu:/test sudo docker-compose up &` to make sure that the server runs under the limit of 50% CPU usage.
4. Start the client independently by running

   ```
   javac client/*.java
   java client/Client
   ```

5. Record the time to run 1000, 2000, 3000, 4000, 5000, 6000 and 10000 clients and compare the results.

2.1.2 Test for 100% CPU usage

In this part, we just run

```
sudo docker-compose up
```

and start the client. Then record the time taken to run the same number of clients as above.

2.2 Results

We get the testing results as follows (the numbers indicate the time taken for the server to create N accounts, add 3 symbols to each of them and work with N orders based on the accounts and symbols (where N is the scale, or the number of requests).

2.2.1 Results for 100% CPU usage

When we allow the server to occupy all available CPU resources, we get the following results:

Scale	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)	Avg (ms)	Time/Request
2000	20086	20262	20766	20371.3	10.1857
4000	43960	44851	44165	44325.3	11.0813
6000	69969	68803	70470	69747.3	11.6246
8000	93498	93228	92724	93150	11.6438
10000	121395	119131	120113	120213	12.0213

2.2.2 Results for 50% CPU usage

When we set a limit to ensure that the server utilizes at most 50% CPU resources, we get the results like:

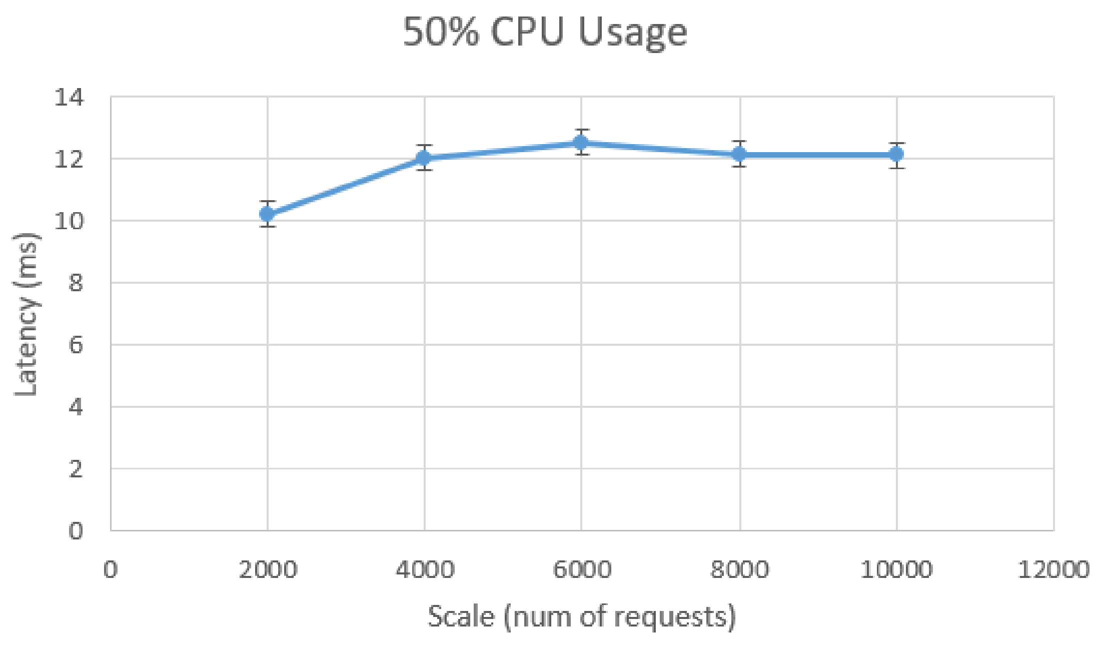
Scale	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)	Avg (ms)	Time/Request
2000	20796	19906	20685	20462.3	10.2312
4000	45414	48978	49971	48121	12.0303
6000	77421	77305	70682	75136	12.5227
8000	96474	98047	97436	97319	12.1649
10000	122444	121141	120711	121432	12.1432

2.3 Scalability Analysis

In this part, we analyze the performance by comparing how latency changes, *i.e.*, by comparing the `Time/Request` with each other.

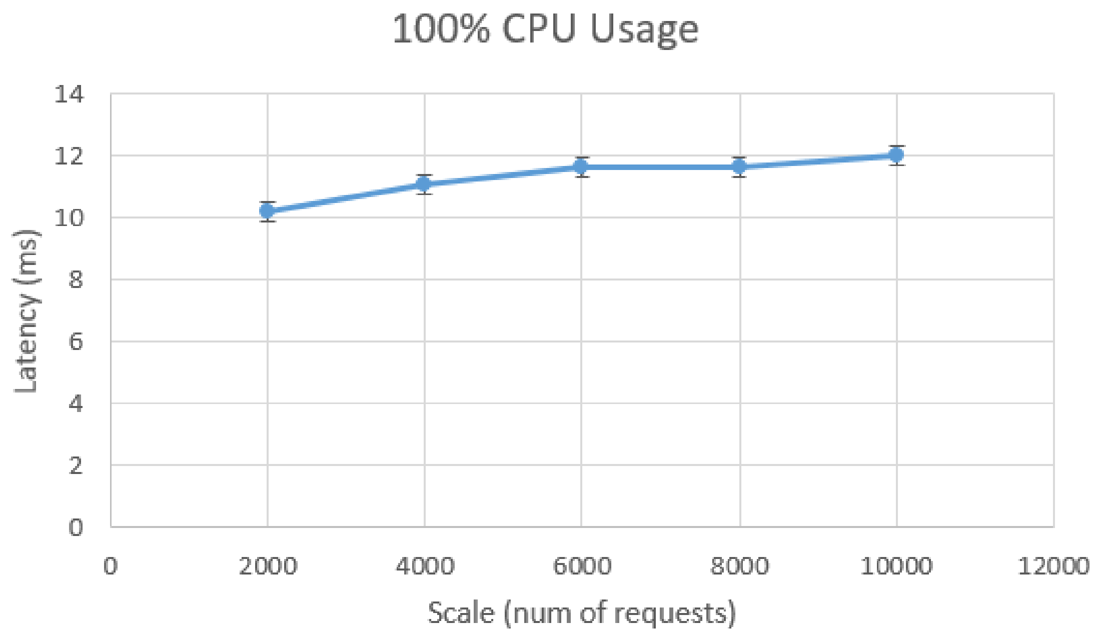
2.3.1 Performance change when the number of requests increases

We first analyze how our performance changes when the number of competing requests increases. For 50% CPU usage, the results are



The latency first increases, and then it roughly remains unchanged as the number of requests increases.

For the 100% CPU usage, the results are:

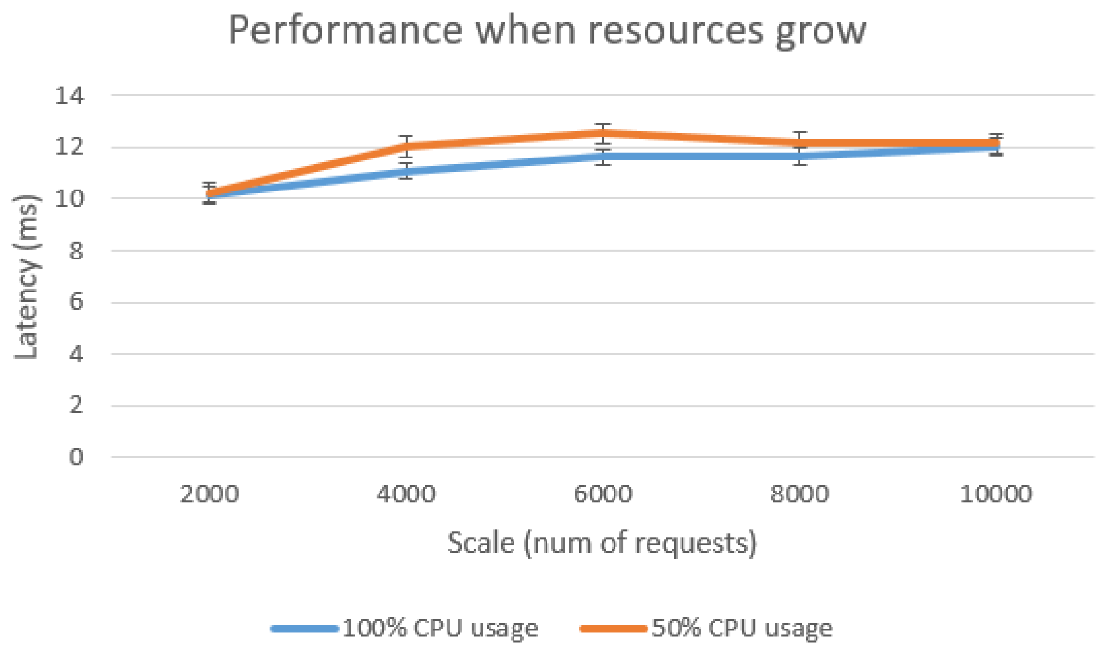


The latency increases as the number of competing requests grows.

Clearly, the latency tends to increase when there are more competing requests, because the requests competes for communication and computing resources. But the latency increase is not so significant.

2.3.2 Performance change when the resources grow

To compare how the performance improves when we give the server more computing resources, we compare the results of 50% usage and 100% usage by putting them into the same figure:



The orange line is the situation when only 50% CPU resources are available, while the blue line shows the results for 100% CPU usage. From the figure, we can verify that the performance of our server improves when more resources are granted, although the improvement is not significant. In fact, we can calculate the percentage of latency reduction using the formula

$$\% \text{ of Improvement} = \frac{T_{100\% \text{ usage}} - T_{50\% \text{ usage}}}{T_{50\% \text{ usage}}} \times 100\%$$

and the results are as follows:

Scale	Improvement
2000	0.44%
4000	7.89%
6000	7.17%
8000	4.28%
10000	1.00%

The improvement is most significant when the request numbers are between 4000 and 6000. As the requests grow continuously, it becomes less significant. The most possible reason for this change is that we let all threads share the same database connection. As more and more threads compete for that connection resource, more CPU resources are not helpful for performance enhancement.

2.4 Possible Improvements

Currently we use one database connection for all threads. Although we improve the CPU resources, the database connection becomes the bottleneck for us to improve the speed for server to work on requests. So we may need to provide multiple database connections for threads.