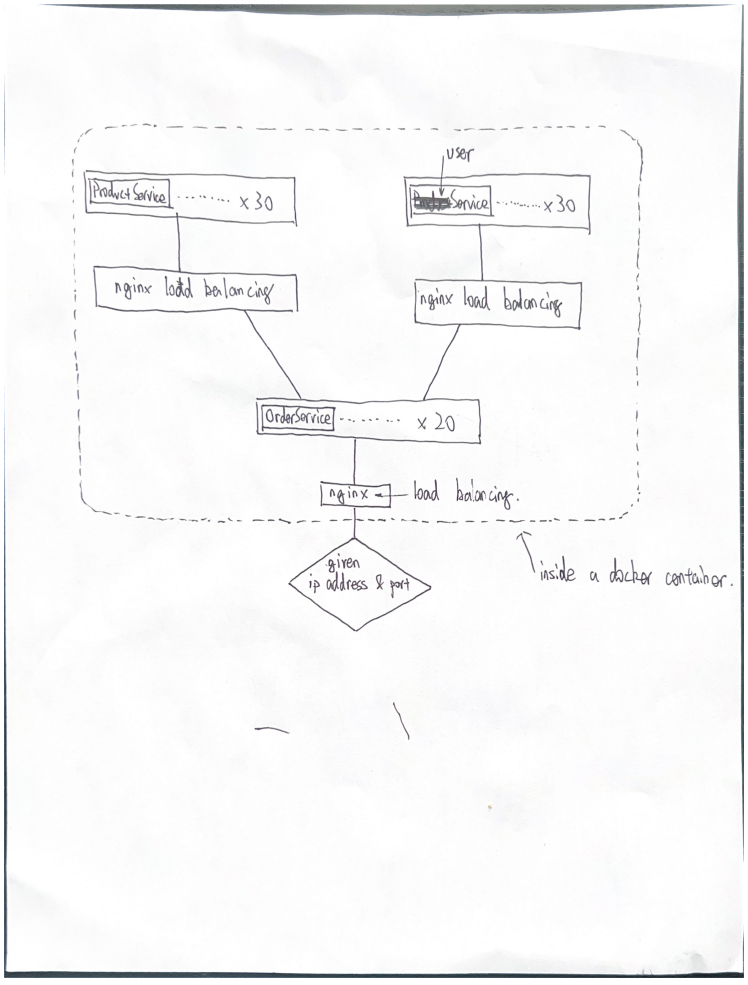
A2 Component 2 Report

March 2025

Architecture



Explanation: We use nginx inside Docker container as load balancing solution. We started 20 instances of OrderService, which acts as an order request handler

and product & user request proxy. Behind OrderService we started 30 instances of ProductService and 30 instances of UserService to implement horizontal scaling.

Profiling Process and Justification of our Current Approach

We generated a 4400 line testcase where every instruction inside should return 200 if it is sent out in the correct order. Before we started implementing horizontal scaling, we measured time cost of our original code:

```
calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Elapsed time: 7.538276 seconds
```

We can see the efficiency is 4400req/7.54s, which is way below the requirement. We then try to increase the number of threads in each HTTP handler from 20 to 100 (e.g. set httpServer.setExecutor(Executors.newFixedThreadPool(100))), and we got following profiling result:

```
calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Elapsed time: 7.011422 seconds
```

We noticed that the improvement is not big enough and when we repeat the test, sometimes the increase of thread may leader to longer processing time of 4400 requests. So we discarded this adjustment.
Next, we used a handwritten round-robin load balancer in front of OrderService, and we started 10 instances of OrderService. We got following profiling result:

```
calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Elapsed time: 4.946769 seconds
```

If we clear the database and then rerun the same test cases, we got the following:

```
calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Processed 4400 commands in 5.17 seconds.

calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Processed 4400 commands in 4.25 seconds.

calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Processed 4400 commands in 3.74 seconds.

calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Processed 4400 commands in 3.68 seconds.

calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Processed 4400 commands in 3.59 seconds.

calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
calico@OrangeBraincell a1 % Processed 4400 commands in 3.72 seconds.
```

We noticed that everytime on the first run, our handwritten load balancer took a longer time (about 5 seconds) but after running the same testcases again and again, the time cost have reduced to a minimum of approximately 3.6 seconds. However, the issue with handwritten load balancer is it failed to improve further: We've tried to increase the number of both OrderService and UserService, ProductService instances, but the profiling result is still in same pattern.
Then, we tried to profile memory usage using JProfiler. We attached out running OrderService process to JProfiler and got the following result:

Name	Instance Count	Size
byte[ ]	37,216 (20 %)	4,742 kB
java.lang.String	29,956 (16 %)	718 kB
java.lang.Object[ ]	6,743 (3 %)	437 kB
java.util.HashMap$Node	6,650 (3 %)	212 kB
java.util.concurrent.ConcurrentHashMap$Node	6,060 (3 %)	193 kB
java.lang.Class	4,227 (2 %)	512 kB
jdk.internal.net.http.common.MinimalFuture	2,614 (1 %)	83,648 bytes
java.util.ArrayList	2,141 (1 %)	51,384 bytes
int[ ]	2,031 (1 %)	146 kB
java.lang.Object	1,883 (1 %)	30,128 bytes
char[ ]	1,862 (1 %)	567 kB
java.util.TreeMap$Entry	1,859 (1 %)	74,360 bytes
java.lang.Class[ ]	1,536 (0 %)	43,200 bytes
java.lang.invoke.MemberName	1,498 (0 %)	71,904 bytes
java.util.ImmutableCollections$List12	1,433 (0 %)	34,392 bytes
java.util.ArrayList$Itr	1,326 (0 %)	42,432 bytes
java.util.LinkedHashMap$Entry	1,296 (0 %)	51,840 bytes
java.lang.String[ ]	1,251 (0 %)	46,296 bytes
sun.security.util.ObjectIdentifier	1,228 (0 %)	39,296 bytes
java.util.LinkedList$Node	1,132 (0 %)	27,168 bytes
java.lang.invoke.LambdaForm$Name	1,117 (0 %)	35,744 bytes
java.lang.invoke.MethodType	1,108 (0 %)	44,320 bytes
java.lang.invoke.MethodType$ConcurrentWeakInternSet...	1,099 (0 %)	35,168 bytes
java.lang.invoke.ResolvedMethodName	1,053 (0 %)	25,272 bytes
java.util.concurrent.locks.ReentrantLock$NonfairSync	1,040 (0 %)	33,280 bytes
java.util.HashMap	980 (0 %)	47,040 bytes
java.util.HashMap$Node[ ]	886 (0 %)	115 kB
java.util.TreeMap	880 (0 %)	42,240 bytes
java.util.concurrent.locks.ReentrantLock	876 (0 %)	14,016 bytes
java.util.LinkedList	842 (0 %)	26,944 bytes
java.util.LinkedList$ListItr	792 (0 %)	25,344 bytes
java.util.Optional	764 (0 %)	12,224 bytes
sun.security.util.DerInputStream	761 (0 %)	30,440 bytes
Total from 1,624 rows:	181,539 (100 %)	13,316 kB

One thing we noticed is that a big portion of memory is spend on String objects, so we adjusted our methods of constructing IP address string from using StringBuilder to just simple String literals(e.g. using the addition operator). We then ran the 4400 lines of testcases again, and got the following result:

```
[calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
 calico@OrangeBraincell a1 % Elapsed time: 6.331167 seconds

[calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
 calico@OrangeBraincell a1 % Elapsed time: 5.383700 seconds

[calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
 calico@OrangeBraincell a1 % Elapsed time: 4.306875 seconds

[calico@OrangeBraincell a1 % ./runme.sh -w /Users/calico/Documents/301/CSC301A1/longworkload2.txt
 calico@OrangeBraincell a1 % Elapsed time: 3.975579 seconds
```

We noticed there is no significant improvement by adjusting String construction method.

Finally, we started to use Nginx as the load balancer. Similar to previous sections, we deploy a load balancing rule in front of 10 instances of OrderService, and we also deploy a load balancing rule in front of 10 instances of ProductService and UserService. With Nginx load balancing we got a much better profiling result:

```
vscode ➜ /workspaces/CSC301A1/a1 (main) $ POST http://127.0.0.1:3000/user -> 200
GET http://127.0.0.1:3000/user/100 -> 404
GET http://127.0.0.1:3000/user/100 -> 404
POST http://127.0.0.1:3000/user -> 404
GET http://127.0.0.1:3000/product/200 -> 404
POST http://127.0.0.1:3000/product -> 200
POST http://127.0.0.1:3000/product -> 404
POST http://127.0.0.1:3000/user -> 200
POST http://127.0.0.1:3000/order -> 404
GET http://127.0.0.1:3000/user/101 -> 200
POST http://127.0.0.1:3000/order -> 404
POST http://127.0.0.1:3000/product -> 200
GET http://127.0.0.1:3000/product/201 -> 404
POST http://127.0.0.1:3000/order -> 404
POST http://127.0.0.1:3000/user -> 200
GET http://127.0.0.1:3000/user/101 -> 200
POST http://127.0.0.1:3000/user -> 404
POST http://127.0.0.1:3000/product -> 404
Processed 4400 commands in 1.30 seconds.
POST http://127.0.0.1:3000/product -> 200
GET http://127.0.0.1:3000/user/102 -> 404
POST http://127.0.0.1:3000/user -> 404
POST http://127.0.0.1:3000/product -> 200
GET http://127.0.0.1:3000/product/202 -> 404
POST http://127.0.0.1:3000/product -> 404
POST http://127.0.0.1:3000/user -> 200
POST http://127.0.0.1:3000/product -> 404
GET http://127.0.0.1:3000/user/102 -> 200
POST http://127.0.0.1:3000/user -> 200
POST http://127.0.0.1:3000/user -> 404
POST http://127.0.0.1:3000/order -> 404
```

We noticed that our load balancer achieved 4400 commands in 1.3 seconds, which is above 3000/s requirement.
We further fine-tune the number of instances started for each service. After some trial-and-error we found the optimal number of instances is 20 OrderServices, 30 UserService, and 30 ProductServices(as shown in part 1 architecture diagram). Using this configuration, we got the following profiling result:

```
vscode ➜ /workspaces/CSC301A1/a1 (main) $ POST http://127.0.0.1:3000/product -> 409
GET http://127.0.0.1:3000/product/200 -> 200
POST http://127.0.0.1:3000/product -> 200
POST http://127.0.0.1:3000/product -> 409
POST http://127.0.0.1:3000/product -> 404
POST http://127.0.0.1:3000/order -> 200
GET http://127.0.0.1:3000/product/201 -> 200
POST http://127.0.0.1:3000/order -> 200
POST http://127.0.0.1:3000/user -> 200
POST http://127.0.0.1:3000/product -> 200
POST http://127.0.0.1:3000/user -> 404
POST http://127.0.0.1:3000/product -> 404
Processed 4400 commands in 0.57 seconds.
POST http://127.0.0.1:3000/user -> 409
POST http://127.0.0.1:3000/product -> 409
GET http://127.0.0.1:3000/user/102 -> 200
GET http://127.0.0.1:3000/user/101 -> 200
GET http://127.0.0.1:3000/user/104 -> 200
POST http://127.0.0.1:3000/order -> 200
POST http://127.0.0.1:3000/order -> 200
POST http://127.0.0.1:3000/user -> 404
GET http://127.0.0.1:3000/user/100 -> 200
GET http://127.0.0.1:3000/user/101 -> 200
POST http://127.0.0.1:3000/user -> 200
GET http://127.0.0.1:3000/user/100 -> 200
POST http://127.0.0.1:3000/user -> 409
POST http://127.0.0.1:3000/order -> 200
POST http://127.0.0.1:3000/user -> 404
POST http://127.0.0.1:3000/user -> 409
```

We achieved a throughput way above the 3000/s requirement. However, we noticed that due to our client's configuration: async sending requests, some of the requests are received in the wrong order so some of these requests return non-200 response code. If we adjust our client workload parser to synchronous sending mode, the throughput of our API endpoints can still meet 2000/s requirement, due to hardware limitations of our group member's local machines.