Architecture and Profiling

Introduction of the System Architecture

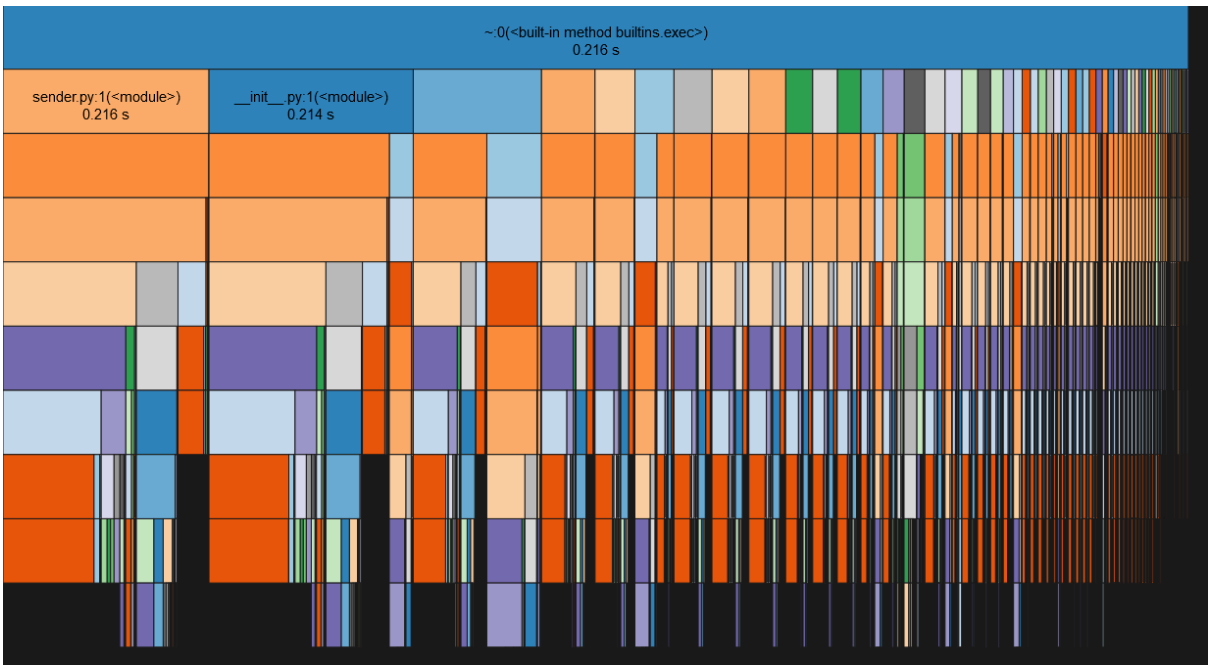
Our microservices architecture is designed to effectively handle large volumes of requests by utilizing Docker containers and database migrations. The architecture uses a VM with 75 GB of RAM to log into and then run a version of docker. The architecture has three services: user, product, and order services. These do not communicate with each other; all communication goes through the ISCS. All requests go to the order service, which, if the request is for the user or product, will forward the request to the ISCS, which will handle it. The ISCS handles the request by forwarding it to the proper server. For the database, we used a Postgresql database to handle more requests. In addition, we also made a simple hashmap cache to optimize the performance. The cache mainly improves the performance of GET requests as the data of the response would be fetched directly from the cache instead of the database, so we don't need to connect to the database when each time handling GET requests. Each time the data is created, updated or deleted in the database, the same operation is also applied to the cache.

Why We Made These Choices

We chose Postgres for our database needs due to its scalability, reliability, and feature set, which can handle concurrent transactions with good data integrity. Docker was selected for containerization because of its lightweight on CPU resources, ease of deployment and compatibility with the lab machines. Which is crucial to achieving good results on requests. Caching is enabled for GET Requests to speed up the overall processing power. ISCS is used for internal communication, together with the load balancer, to attain maximum request handling results.

Profiling Of Architecture

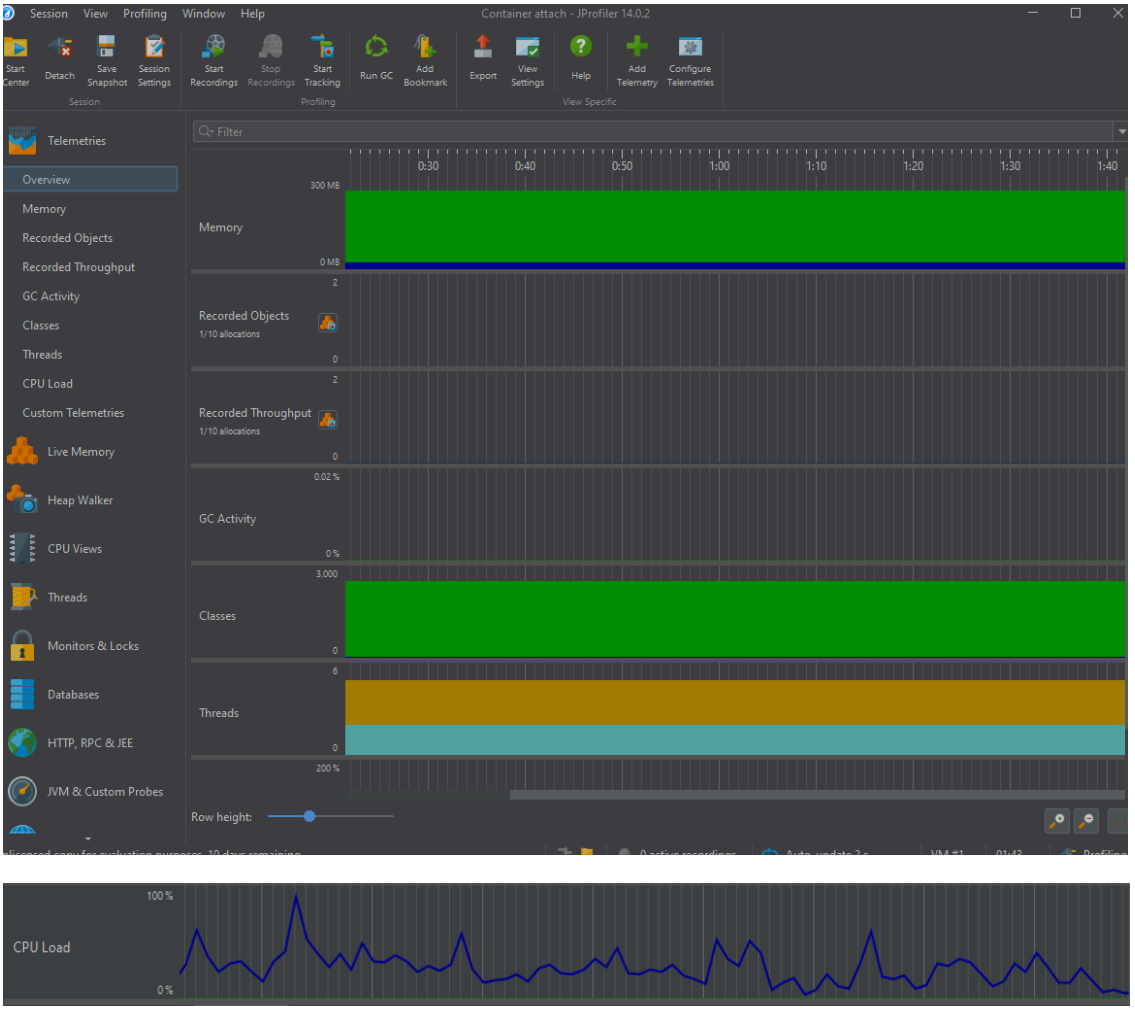
Snakeviz with Cprofile on the sender.py



The above picture did not tell me much because most of these are Python's own calls. From looking at snakeviz of sender.py, I knew I had to test the Java files themselves. I could also try the ISCS.py, but it seems that the Python files were waiting for the Java files to finish processing the request, indicating a bottleneck primarily in the processing of requests by Java services. So it would be better to look at the Java files processing requests and try to speed up the architecture that way.

For that, I used JProfiler in intelJ. The following JProfiler is for UserService.java, and the service currently takes at least 100 requests per second. These requests are GET requests to the database for user_id 2011.

Figure 2: No Hikari, 100 GET request via sender.py



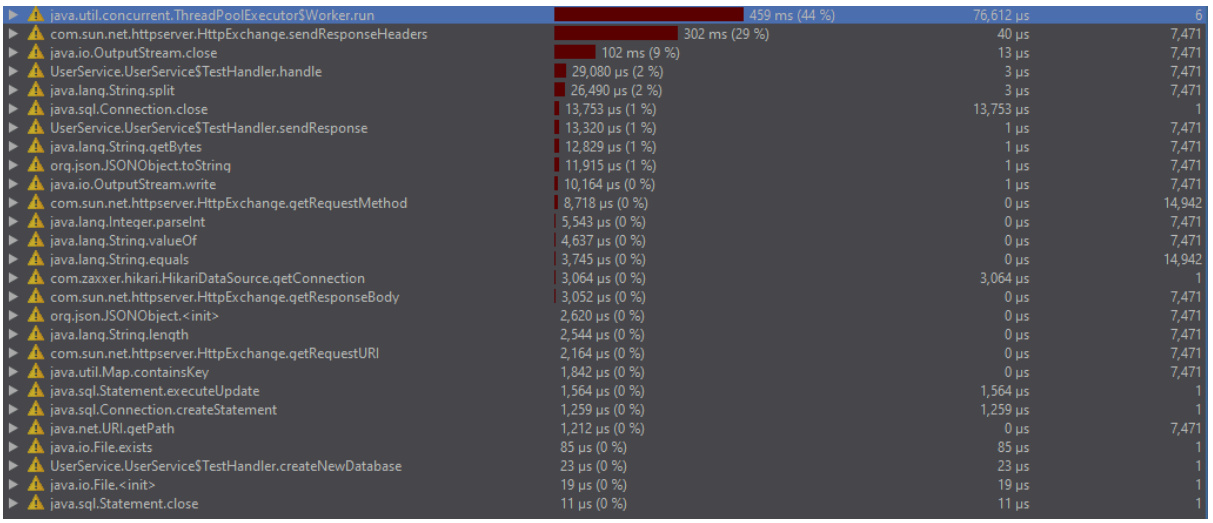
The profiler above used only one single thread. Also above is the CPU load of UserService. After some googling research, I decided that increasing the parallel processes could help handle spikes and mitigate the identified bottlenecks. Hikari is known for its lightweight and high-performance connection pooling capabilities, which allow us to manage database connections more efficiently and increase throughput during high volume. So I used Hikari, whose README.md file describes itself as a package for connection pools that is responsive to spikes in requests.

After implementing Hikari for UserService and OrderService, the CPU looked more like the one below. Now it still seems like there are more spikes than a regular Java program, but there is a noticeable improvement. However, as shown below, the thread count still did not increase, and UserService only used one thread on my computer. There are more threads, but they are, for some reason, all still waiting, even with Hikari. I have also increased the Java executors thread pool to 50.

However, after sender.py recorded a drop in the request process, I reversed it back to 10.

Figure 3: Hikari 100 GET-request per second via sender.py

I then looked at the hotspots of my UserService.java; the following methods take the most time.



java.util.concurrent.ThreadPoolExecutor$Worker.run	459 ms (44 %)	76,612 µs	6
com.sun.net.httpserver.HttpExchange.sendResponseHeaders	302 ms (29 %)	40 µs	7,471
java.io.OutputStream.close	102 ms (9 %)	13 µs	7,471
UserService.UserService$TestHandler.handle	29,080 µs (2 %)	3 µs	7,471
java.lang.String.split	26,490 µs (2 %)	3 µs	7,471
java.sql.Connection.close	13,753 µs (1 %)	13,753 µs	1
UserService.UserService$TestHandler.sendResponse	13,320 µs (1 %)	1 µs	7,471
java.lang.String.getBytes	12,829 µs (1 %)	1 µs	7,471
org.json.JSONObject.toString	11,915 µs (1 %)	1 µs	7,471
java.io.OutputStream.write	10,164 µs (0 %)	1 µs	7,471
com.sun.net.httpserver.HttpExchange.getRequestMethod	8,718 µs (0 %)	0 µs	14,942
java.lang.Integer.parseInt	5,543 µs (0 %)	0 µs	7,471
java.lang.String.valueOf	4,637 µs (0 %)	0 µs	7,471
java.lang.String.equals	3,745 µs (0 %)	0 µs	14,942
com.zaxxer.hikari.HikariDataSource.getConnection	3,064 µs (0 %)	3,064 µs	1
com.sun.net.httpserver.HttpExchange.getResponseBody	3,052 µs (0 %)	0 µs	7,471
org.json.JSONObject.<init>	2,620 µs (0 %)	0 µs	7,471
java.lang.String.length	2,544 µs (0 %)	0 µs	7,471
com.sun.net.httpserver.HttpExchange.getRequestURI	2,164 µs (0 %)	0 µs	7,471
java.util.Map.containsKey	1,842 µs (0 %)	0 µs	7,471
java.sql.Statement.executeUpdate	1,564 µs (0 %)	1,564 µs	1
java.sql.Connection.createStatement	1,259 µs (0 %)	1,259 µs	1
java.net.URI.getPath	1,212 µs (0 %)	0 µs	7,471
java.io.File.exists	85 µs (0 %)	85 µs	1
UserService.UserService$TestHandler.createNewDatabase	23 µs (0 %)	23 µs	1
java.io.File.<init>	19 µs (0 %)	19 µs	1
java.sql.Statement.close	11 µs (0 %)	11 µs	1

Looking at the hot spots, it seems that we can cache some of our IO in UserService and ProductService to avoid rereading or rewriting data that has already been seen. For that, we implemented a simple hashmap to improve the architecture's speed. This way, repetitive database queries are reduced significantly. The hash map should cache, read, and write to the database. There is no need to write if the data is the same in the cache. This strategy is thus very effective for User and Product services, where specific requests are expected. Data consistency and reliability are also preserved in our caching logic.

Challenges of Current

One of the main challenges we faced was balancing the thread pool size for optimal performance without overloading the system resources. Also, it seems the current requests handled per second are around the 150-200 range (at least, only tested on personal machines as labs are bugged of now), which we will implement further techniques to increase the threshold further.

Conclusion

While our current optimizations have significantly improved the services' scalability and responsiveness during high requests, we can even push the boundaries further to achieve even higher throughput and lower latency.

We have completed profiling, optimizing and scaling our microservices architecture and managed to enhance our system’s performance substantially, preparing it for the increased load anticipated for the demo. We aim to continuously improve and look forward to implementing further optimizations to ensure the services remain robust and scalable for the demo.