Microservices Architecture Performance Analysis with Eureka

Group 19: Jason Chen, Hugh Song, Karry Dong, River Chen, Allen Liu

Platform and Structure:

Group 19 chose Eureka for register services.It is a service registry that is part of the Netflix OSS (Open Source Software) suite. It allows services to register with it and provides a way for other services to discover and consume those registered services. Eureka operates on a client-server architecture, where the client registers with the Eureka server and periodically sends heartbeats to indicate that it's still available. The server maintains a registry of available services and makes it available to other services that need to consume them. Eureka is designed to be highly available and resilient to failures, making it a popular choice for microservices-based architectures.

We use schools and students information requests as an example for the application. The client service will send API call request to school service and school service will then register on Eureka and send API request to student service at the same time. Eventually, the student servie will register on spring Eureka.

Implementation Details:

We then use AWS EC2 to implement the distributed system. It is a web servie that provides scalable computing capacity in the cloud, which allows users to rent virtual servers, such as instances, on which we can run our own services.

Eureka interface provides a simple and intuitive way for us to monitor and manage the microservices structure, helping us to ensure the services are available, reliable, and performing as expected, the following image is the Eureka interface. We uploaded Eureka to Github repository (https://github.com/cl456852/eureka.git)

API call request is also widely used the project, which is a communication made by a client to the services to access and use a specific API (Application Programming Interface), to define a set of rules and protocols for building and integrating software applications, allowing different systems to interact with each other.

Results and Measurements:

The measurement method is to use JMeter to create 100 concurrent users to call the Application Gateway. We set the loop count to 10, which means every user made by JMeter will connect 10 times. This resulted in a total of 1,000 requests.

Since the VM that the JMeter tool runs on and the Application Gateway server VM is located in the same subnet, we can use Application Gateway VM’s private IP for access. We are making the API calls by HTTP GET request.

We conducted tests with four different cases. The cases differ in the location of the Virtual Machines (VMs), deployment methods (through AWS ECS or Docker build), and

communication patterns (API or queue). The main measurement methods are latency and throughput.

The test results are shown in the below diagram:

Case	Deployment method	VM Location	Communication Pattern	Min latency (ms)	Max latency (ms)	Average latency (ms)	Throughput (requests per minute)
1	AWS ECS	2	API	126	262	132	599.778
2	AWS ECS	1	API	4	42	6	605.914
3	Docker	1	API	4	137	12	607.33
4	AWS ECS	1	Queue	16	145	22	604.906

Analysis of results:

The results presented in the table appear to be consistent with the general understanding of the factors affecting latency and throughput. The deployment method, VM location, and communication pattern can all have an impact on performance.

The VM location plays a significant role in latency. Cases 2, 3, and 4, which all use VMs in the same location (US-EAST-1), have lower latencies compared to Case 1, which uses VMs in two locations (US-EAST-1 and US-WEST-2). This suggests that the average latency decreases as the instances are located closer to each other.

The use of AWS ECS as a deployment method in Case 2 may provide advantages such as scalability and better resource management. The Docker deployment in Case 3 has slightly higher latency while the throughput remains about the same.

Lastly, the communication pattern affects the latency and throughput as well. The use of a queue-based communication pattern in Case 4 results in higher latency compared to the API-based communication in Case 2, despite both cases using AWS ECS and VMs in the same location. The maximum latency for queue communication is much higher than for API communication. The throughput for both communication patterns is similar.

The argument of correctness can be made based on the 0% error rate in all cases, which indicates the reliability of the system. Throughput remains relatively consistent across all cases, ranging from 599.778 to 607.33 requests per minute.

In conclusion, the data demonstrate the influence of instance placement, deployment method, and communication patterns on latency and throughput. These insights can help inform future optimizations and design decisions for the system.