CS3301 Microservice Report

170011474

February 2020

1 Overview

The report analyses the Microservice architecture with an e-commerce application illustrated. The case is an Amazon-like e-commerce platform to offer online shopping. After a customer submits an order request, the application would verify the order based on financial information of customers. If a purchase is valid, then the fees would be deduced from buyer's bank accounts. Delivery would be allocated and dispatched.

Main functionalities of it are identified and different architecture patterns are evaluated. Furthermore, a design of it is developed. Particular aspects of the design will be demonstrated to offer a brief and intuitive opinion.

2 Is Microservice Architecture Suitable for It?

Before Microservice came out, monolithic architecture dominated our world with a collection of functionalities integrated to one container (Richardson, 2019). Those functionalities cannot execute independently since they cannot share resources in a wider context.

Although Monolithic style provides developers with benefits, such as the ease of applying radical changes, of developing, and of testing, potential drawbacks appeared as companies grew in size (Richardson, 2019). Evolving and maintaining a large-size monolith software are complicated. Any modification would affect other parts and hence a re-loading is required. Running a monolithic application only results in sub-optimal circumstances since running environments are not always identical and consistent. Same programming language and framework needs to be retained. Those monolithic applications could not be that scalable due to its high-intense central integration.

Microservice architecture appeared as a remedy to this situation. A microservice is a cohesive and independent process interacting via messages (Dragoni et al., 2017). It is loosely coupled without heavy dependencies, independently deployable and highly maintainable. It enables a distributed system consisting of many microservice and each service has its own database. Based on single responsibility principle, which states that each component only takes responsibility of a single function, it limits the range where a bug can occur. Possible

adaptation to a new version of application would not take a longer time to reboot, only a rebooting in the modified module. Running environments have a high degree of freedom to be configured and a higher scalability can be achieved. Therefore, microservices architecture can address issues caused by the monolithic architecture.

However, microservice architecture also possesses some shortcomings. Communication between services over a network can have poor performance, for the possible data losing and overhead exists in transforming. Apart from that, transaction security is concerned during distributed communication. Additionally, if each service is maintained by one team, then coordination and collaboration are supposed to carefully defined.

From evaluation above, firstly, we can know microservice architecture is a great match to our e-commerce application, since functionalities of our application can be safely split into relatively independent module services. Each service can have its own database so it is easier and faster to manipulate data. For example, customer service can use its database to store customer information, the database of order service can store order information. In this way, large data sets do not have to be stored in a single central database, which reduces searching time and possible sparse values. Secondly, considering that a successful e-commerce application tends to be fast-changing to keep pace with preferences of customers and technology, frequent changes and update on a monolithic system can spread cascading effect and increase potential risks. Therefore, since features of microservice architecture can settle down those hazardous cases, it is more suitable than traditional monolithic one.

Overall, after researching on characteristics of microservice and our e-commerce application, we can draw a conclusion that microservice architecture is indeed an optimal choice.

3 Design of the Architecture with Netflix

There is a wide range of libraries and tools offered by Netflix platform, and some of them are closely related to the microservice architecture. So only those relevant tools are discussed in our design.

3.1 Main Functionalities and Architecture of E-Commerce Application

Main functionalities of the e-commerce application were delegated into several parts: order service, customer service, delivery service and payment service. Resources needed to implement this model are databases for each service which can be manipulated in a consistent, available and fault tolerant way.

By using `Single Responsibility Principle`, the separation of following services is verified to be independent enough so there is no overlapped area or duplicate service. It is illustrated as below:

3.1.1 Order Service

Order Service creates an order request in which order ID (key), date and time, delivery fees, and any other information related to that order (such as price) should be accurately stored in its own database. After retrieving financial information from payment service's database based on customer ID, it can decide to approve, reject (if the buyer does not have enough credits to buy it) the order. If the order is approved, then payment service starts to work to reserve money and invokes delivery service. The order state (approved or rejected) will be sent back to customer service.

Notice that other classes related to orders can be created, not limited to `OrderState` which represents states as "approved, pending, canceled" and so on. `Order` will be an instance storing all relevant information, including timestamp, product price, shipping fees, order state etc.

The basic functions include **(pseudo code)**:

1. `createOrder(int timestamp, float deliveryFees, ...)` creates an order and an order ID and calls `storeOrder()`
2. `storeOrder(int orderID, Order order)` uses `orderID` as a key and `Order` as a value;
3. `updateOrder(int orderID, Order order)` updates an existing order;
4. `fetchOrder(int orderID)` returns an order instance from database;
5. `removeOrder(int orderID)` returns a current in-progress order instance in database;
6. `verifyOrder(int orderID)` invokes Payment Service to check the buyer's financial condition;
7. `approveOrder()` updates order state and invokes Delivery Service;
8. `rejectOrder()` calls `removeOrder()` from database;
9. `updateState(int orderID, OrderState state)` obtains the order from database first, and then updates the state. When order state becomes "completed" then calls `completeOrder(int orderID)`;
10. `completeOrder(int orderID)` notifies customers this update and calls `removeOrder()`.

3.1.2 Customer Service

Customer Service can create a new customer instance and store information of customers, including their bank accounts, address, customer id, name and other personal data into its own database.

The basic functions include **(pseudo code)**:

1. `createCustomer(...)` creates an customer;
2. `storeCustomer(int customerID, Customer customer);`
3. `fetchCustomer(int customerID);`
4. `updateCustomer(int customerID, Customer customer);`

3.1.3 Delivery Service

Delivery Service deals with the delivery for an approved order. It can create a carrier instance and delivery request. After extracting address information from customer service's database, a delivery request would be automatically allocated to a suitable carrier who probably works near the shipping destination based on an algorithm. It also communicates with order service when a delivery is successfully made so the order state is updated to "complete".

A class `DeliveryState` is implemented to record states "prepared", "dispatched", "delivered", "completed". "completed" means the buyer confirms he or she has received that package.

The basic functions include**(pseudo code)**:

1. `createDelivery(...)` creates a delivery instance and calls `storeDelivery()`;
2. `storeDelivery(int deliveryID, Delivery delivery);`
3. `updateDelivery(int deliveryID, Delivery delivery);`
4. `fetchDelivery(int deliveryID);`
5. `removeDelivery(int deliveryID);`
6. `updateState(int deliveryID, DeliveryState state)` updates the current delivery state. If the state becomes "completed", then it will notify Order Service, and Order Service will retrieve the corresponding order and update its state as well.

3.1.4 Payment Service

Payment Service manages bills and fees a customer should pay, and connects to his or her bank to charge. After an order is approved, the payment service would reserve or freeze the corresponding amount of money. After an delivery is made, the payment service would then take the money away from the customer's bank accounts.

A class `PaymentState` is implemented to represent states "reserved", "completed", stored into Payment instance.

The basic functions include**(pseudo code)**:

1. `createPayment(...);`

2. `storePayment(int paymentID);`
3. `fetchPayment(int paymentID);`
4. `removePayment(int paymentID);`
5. `updateState(int paymentID, PaymentState state)`
6. `verifyOrder(int orderID)` checks if the buyer has enough money to purchase. The result will be returned to Order Service;
7. `reserveMoney(int paymentID, float fees)` reserves fees and shipping fees, updates the PaymentState to "reserved".
8. `deduceMoney(int paymentID, float fees)` uses `fetchPayment()` to fetch a specific payment instance from database and then deduces the "frozen" fees from bank accounts. After that, the Payment state will be updated as "completed". `removePayment()` will be called to remove the completed payment instance.

3.2 Netflix Ribbon with Netflix Eureka - IPC Mechanism Powered by Service Discovery

Ribbon is a client-side inter-process communication library offering loading balance and other functions. It can cooperate with Eureka and Hystrix to provide developers with a more reliable, fault-tolerant model. Eureka is a library to offer service discovery. Since service instances can dynamically assign network locations rather than an everlasting static address, it is very important in modern networking that Eureka enables client to determine dynamic locations.

For our e-commerce application, we can build one Eureka server and many Eureka clients for service instances and the application client (see figure 1). Each service instance is a Eureka client. **In our design, each service instance needs to:**

1. do **service registry** with their ip address and port number sent to the Eureka server. Thus, Eureka server forms a list of available service instances.
2. send **heartbeats** periodically (typically every 30 seconds) to the Eureka server to renew its membership in the list. Otherwise, it will be removed from the list.

When the application needs to call one of registered service, it will query Eureka server by service name to see whether there is an available service in the list. After retrieving service information, it uses loading balance algorithm in Ribbon to choose one of the target service instances and hence the application sends a running request (Richardson, 2015).
To illustrate this mechanism in context of our case, we can use an entire workflow of a successful purchase by a customer James whose customer ID is '1' and he just registered as a new customer.

1. At the very beginning, once all services started up, they registered themselves in the Eureka server.

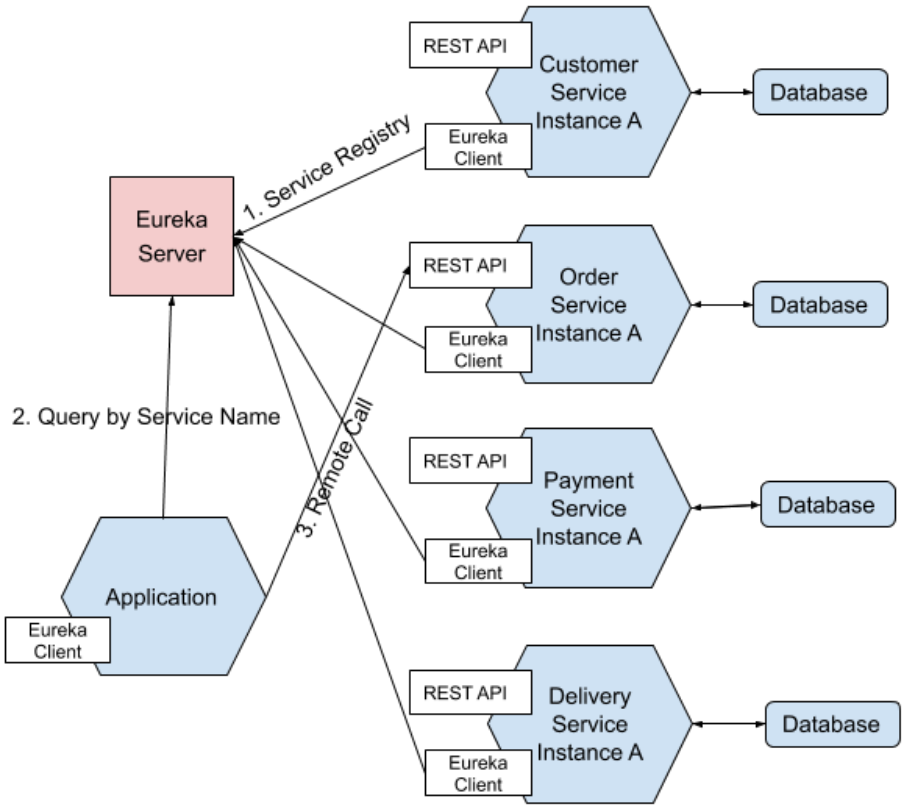

Figure 1: Design with Eureka and Ribbon

2. Application would ask the Eureka server for customer service. The server looks up its database and then picks one instance for it and response.

3. It then uses the genuine ip address to call local functions of that instance to create a new customer with all information related to him stored.

4. Application asks the Eureka server for order service and then the instance of order service creates an order instance, recording order ID '1', customer ID '1' and order state as 'pending' and other information in its database.

5. The instance starts to verify order details after the it queried the customer service for financial data. It confirms that James has enough money so it approves the order and updates it.

6. The instance asks the Eureka server for payment service to reserve money.

7. Afterwards, it asks the Eureka server for delivery service, makes a remote call, and then an instance starts to create a new delivery instance and assign it to a carrier.

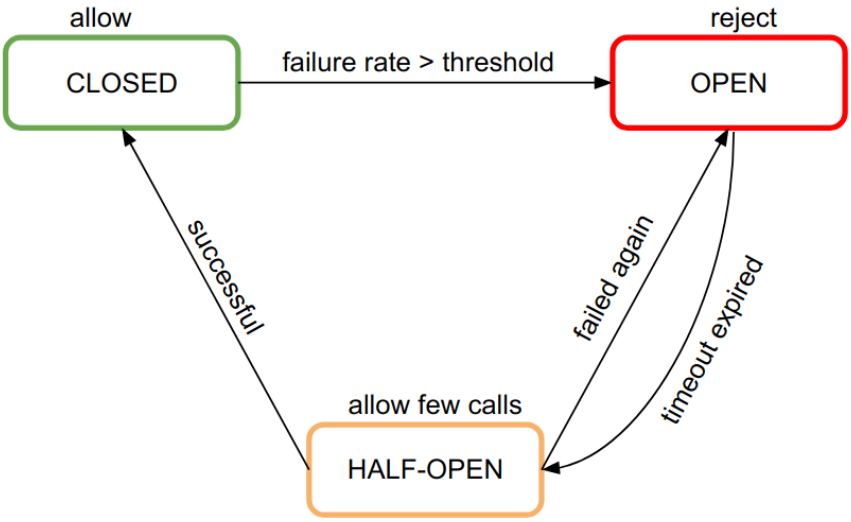

Figure 2: circuit breaker

8. After the carrier updates the delivery state to "complete", an instance of order service updates the status of the order and invokes payment service, and then an instance deduces the money from his bank accounts.

3.3 Netflix Hystrix - Fault Tolerance

In traditional model,if the total number of thread resources is 100, when one service instance gets latent, all threads requesting to that dependency are delayed and even blocked in a short time. Then other unused threads are used to request until all 100 threads are used up. Or in another case, when the order service calls the payment service and the payment service is latent, it would affect the order service in turn and more services are affected. Thus, the failure spreads from one service to a bunch of services.

Hystrix is a Netflix library designed particularly for this latency and fault tolerance. It can control cascading failure and enable resilience in complex distributed systems where failure is inevitable (Netflix, 2016). **The following steps show how Hystrix can be embedded into our e-commerce application:**

1. Each request to an external dependency (or service) is wrapped in a command, and each client performs in a thread.

2. Check whether a response is cached. If so, the result is directly returned.

3. The circuit breaker of it controls passing and blocking between order service and payment service in the previous case. To use an analogy, the circuit breaker acts like a custom and the maximum capacity of waiting room is the threshold of a thread pool. In this case, if the circuit is open, this command would not be executed and a "downgraded" response is returned (see figure 2: circuit breaker).

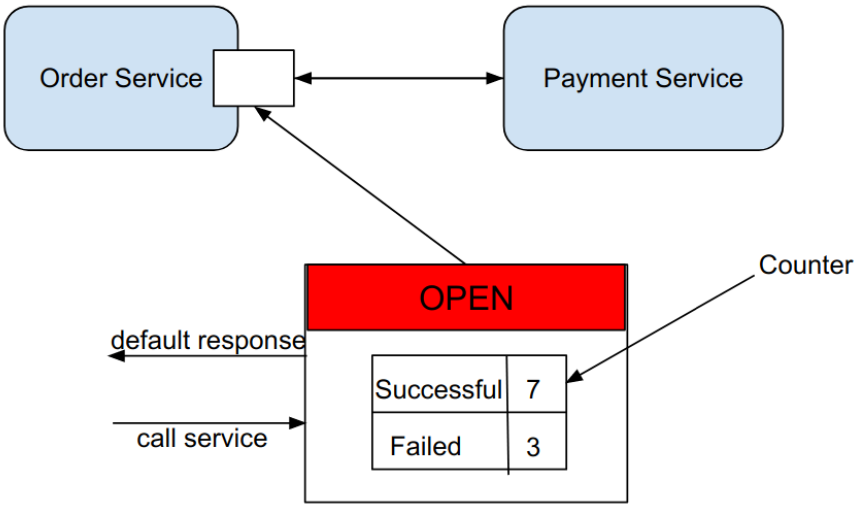

Figure 3: circuit breaker in design

4. It also isolates dependencies from each other and limits concurrent access via thread pools. Once the thread pool saturates or failure percentage hits a certain threshold, the circuit breaker would open the circuit to break request flows.

5. Execute the command.

For example, if payment service is down, and requests to it pile up and the number of them reaches 500 (threshold 500 is assumed) in the thread pool, then circuit breaker prevents any upcoming request for a period of time to wait for recovering. After recovering, the circuit breaker would turn the circuit to a half-open state, passing some requests for testing. If they are fine, then circuit closes. Otherwise, circuit opens to continue blocking.

All components mentioned above, such as a command, circuit breaker, response cache, all requests to a same service would be isolated from others into one thread pool, and so on, those need to be part of our design and strengthen Ribbon and Eureka (see figure 3: circuit breaker in design).

4 Evaluation of the Design

4.1 Evaluation of Microservice Architecture

The microservice architecture has profound influence on the design of many service platforms now, and its advantages and potential risks are discussed above in Section 2.

4.2 Evaluation of Ribbon, Eureka, and Hystrix

Ribbon with Eureka is based on a client-side service discovery. The client-side pattern means the client will determine the locations of usable service instances and use load balancing idea

to send requests. This pattern is more straightforward to develop. Since the client has knowledge of the available instances, it can make clever decisions for load-balancing mechanism.

However, along with this promising advantage, the drawback is obvious as well: each client is coupled with the service registry which is Eureka server in this case. When developers are building this architecture, they need to make more efforts to implement code for every possible programming language used by clients (Richardson, 2019).

In our design, we also adopted thread pools rather than semaphores to control the concurrent access to one dependency service in Hystrix. Both of them manages to fully protect our application. Compared to thread pools, semaphores cannot guarantee an effective timeout method, so the premise that clients are trustable should be met. However, clients' behaviour cannot always be determined. In this way, thread pools are more reliable and take care of edge cases.

What has to be mentioned is that thread pools are not always the optimal strategy, since computational overhead is brought and calling a thread is more expensive than operating a semaphore. This needs to be considered when population of users grows.

5 Conclusion

In conclusion, the microservice architecture does fit our e-commerce business logic. The tools and frameworks in microservice, such as Ribbon, Eureka, and Hystrix, provides it with more powerful fault-handling and dynamic-search capacity. Potential risks and issues also raise along with those benefits.

References

[1] Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017). *Microservices: yesterday, today, and tomorrow.* [online] Hal.inria.fr. Available at: https://hal.inria.fr/hal-01631455/document [Accessed 12 Feb. 2020].

[2] Netflix. (2016). *Netflix/Hystrix.* [online] Available at: https://github.com/Netflix/Hystrix [Accessed 16 Feb. 2020].

[3] Richardson, C. (2015). *Introduction to Microservices — NGINX.* [online] NGINX. Available at: https://www.nginx.com/blog/introduction-to-microservices/ [Accessed 16 Feb. 2020].

[4] Richardson, C. (2018). *Microservices patterns.*

[5] Richardson, C. (2019). *What are microservices?.* [online] microservices.io. Available at: https://microservices.io [Accessed 12 Feb. 2020].