Dodgy Coder Group Project Report

Yu Long	Zeyu Bai	Wendi Song	Yating Zhang
21200600	21200198	20210851	21200539

Introduction

The system is an online shopping website that allows users to log in and search for goods, then submit orders and pay for them. It's a distributed system that consists of five micro-services including a gateway server, goods service, order service, payment service, and user service.

The business is extremely simplified as the focus of the project is distribution features. The functional features are listed below.

- Gateway server: distributes the requests to corresponding microservices.
- Goods service: provides an API that returns all goods available, and another API that returns a good by id.
- Order service: provides an API for submitting orders of multiple goods, meanwhile initiating payment using payment service.
- Payment service: provides a payment API for submitting payments for orders. It also verifies a user using user service.
- User service: provides a login API that verifies the username and password.

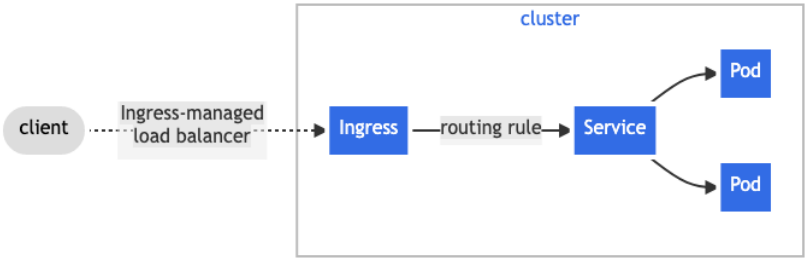
Technology Stack

The goal of this project is to design a distributed system that fulfills the aforementioned business requirement in a highly scalable and fault-tolerant fashion. The system is designed in microservices using REST protocol so that each team member can develop and maintain their own service, which also means that new services can be added easily. Spring Cloud and Kubernetes are the main frameworks that help us to implement distributed systems and microservices.

Gateway and Ingress

In this project, a gateway is used to receive and route all requests from clients to four backend services. Spring Cloud Gateway strives to provide an easy-to-use and effective approach to route to APIs.

The gateway server is deployed with two replicas in this project, port forwarding is used to set up an entry for clients. However, the Kubernetes Service port forwarding function will only select one pod among many pods of a Service. Ingress makes available HTTP and HTTPS routes to Kubernetes cluster services from outside the cluster. Rules set on the Ingress resource govern how traffic is routed. It provides load balancing at the gateway entry. Here is a simple example where an Ingress sends all its traffic to a service:



Service Discovery and Load Balance

The key problem of microservices is service discovery, which allows services to scale up without hardcoding the IP addresses of target services. Kubernetes Service[4] is an abstraction that defines a logical set of Pods of labels. In

other words, microservices inside Kubernetes can find other services by their service names, i.e, labels. Kubernetes Service also has out-of-box load balancing that distributes the traffic among multiple pods.

Spring Cloud also provides service discovery and load balance by Eureka and Ribbon. However, they need another component to integrate with Kubernetes Service. Since we are creating a new system, it's a good idea to directly use the cloud-native features of Kubernetes.

REST

REST services are data-oriented[5]. Most of our services match this pattern, e.g, goods, orders, payments, and users are all resources. With the standard HTTP methods and response codes, we can easily build APIs that interact with them. Compared with other protocols such as WebServices and messages, REST is more lightweight and the API is easier to understand. Distributed objects such as RMI and CORBA are other potential approaches, however, CORBA is very complex in terms of the Interface Definition Language, and RMI is highly coupled with Java and the framework so it's not flexible. However, specification is one problem with REST. So, we introduced Swagger to design the API specifications. OpenApi generator is employed to generate the template server stub and client code to avoid integration issues between services.

Feign REST client provides an easier way to interact with a REST service, which is reflected in the following features: The IP addresses of the microservices interface will be located automatically by Feign according to their names, and a client-side load balancer will be included when using Eureka together. However, in the later stage, we turned to using the service discovery functionality of Kubernetes as described earlier. Therefore, we only took advantage of the HTTP client service of Feign client.

Compared with RestTemplate, Feign is easier and handier to use. Thanks to the OpenApi generator, we get a template code for client APIs, and what we need to do is simply extend the template and change the annotation to reflect the target service name.

Resilience4j

At first two options were considered to solve the problem of system latency and fault tolerance, which are Hystrix and Resilience4j. However, Hystrix is no longer in active development so we chose the latter.

Resilience4j is a lightweight fault tolerance library that provides several core modules such as circuit breaker, rate limiter, bulkhead, retry, time limiter, fallback, and cache; they can easily be implemented with Spring Cloud. Circuit breaker, time limiter, and fallback from this library are selected in the project.

- Circuit breaker: It will temporarily block possible failures when the system is seriously struggling, failing fast is better than making clients wait.
- Time limiter: It limits the duration of execution. Beyond a certain wait interval, a successful result is unlikely.
- Fallback: It provides an alternative result for failures, in this project, a default message will be returned to the client that indicates the failure of the system.

Rate limiter

A rate limiter can be used to prevent excessive usage of resources, such as bandwidth or computational resources, or to limit the number of requests that users or clients can make in a given time period. Rate limiting enables fault tolerance in our applications, making our applications more stable and robust. It can also be used to protect systems from DoS attacks.

Bucket4j is a library used in the user service of our distributed system to implement rate limiting, which uses a counter to track the number of requests made within a given time period, using a token bucket algorithm to allow a certain number of requests per time period.

There are other libraries that offer rate limiters, such as Resilience4j.

MongoDB

For each microservice, we used an individual MongoDB database to store related data. For example, the database in Payment Service stores the corresponding information of a specific payment.

The main reason MongoDB was chosen over other databases is that as a NoSQL database, MongoDB is scalable as its data is not coupled relationally, it is also fault-tolerant because of the ability to replicate and bring down servers for maintenance. Also, it's easy to interact and connect with spring boot applications.

However, because we couldn't connect to MongoDB using the wifi in UCD, we commented on the database-related code when doing the integration and the testing. But the function of the database was tested to be working in other networks.

Docker

Docker is a tool designed to make it easier to create, deploy, and run applications using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. This could avoid inconsistencies in packages in different environments.

Docker can be used to deploy and manage individual components of a distributed system, in which each component can be packaged in its own container, which makes it easier to deploy and manage them with Kubernetes.

Kubernetes and kind

Kubernetes is a platform for running and managing containers from many container runtimes. We mainly used the functionality of Kubernetes like Resource Management for Pods and Containers, Horizontal Pod Autoscaling and Restart Policy to set the resources that each container can allocate, achieve the horizontal scalability of our services and enable fault tolerance respectively. Besides, we also rely on the Service and load balancing features as described above. Docker Swarm is another container orchestration platform that is easy to use and has a lot of features including scalability and load balancing. However, we need more granular control of the resource for scaling and monitoring, as well as more flexibility in cluster management and automation. That's why we chose Kubernetes over Docker Swarm.

To efficiently manage the cluster, we need a powerful tool. k9s is a terminal UI to interact with Kubernetes clusters. The dashboard is very useful by listing all the resources on a single page. We heavily used it in development and testing.

kind is a tool that can start local Kubernetes clusters using docker containers. It helps us set up the local Kubernetes infrastructure. Compared with Minikube, yet another popular tool, it's more lightweight thanks to docker containers. It also consumes fewer resources than other competitors.

System Overview

There are five services in the system: gateway server, goods service, order service, payment service, and user service, as shown in the system architecture diagram below.

Gateway server

The service is responsible for receiving and routing requests from clients to microservices in the system, it also protects the routed services by applying circuit breakers, time limiters, and fallback to them.

Goods service

This service provides information about goods. It consists of two simple APIs: one allows the user to get all product information through a single GET request, and the other allows the user to get the relevant information by item id.

Order service

Order service is responsible for order submission and payment initialisation. It has an API for submitting orders of multiple goods and initiates payment via payment service client. Orders are saved into a standalone MongoDB instance.

A circuit breaker is applied on payment service invocation so that in case of frequent errors or time out, the dependent service could be protected by cutting off the request. The time limiter is applied to assist the circuit breaker by canceling requests that last over one second.

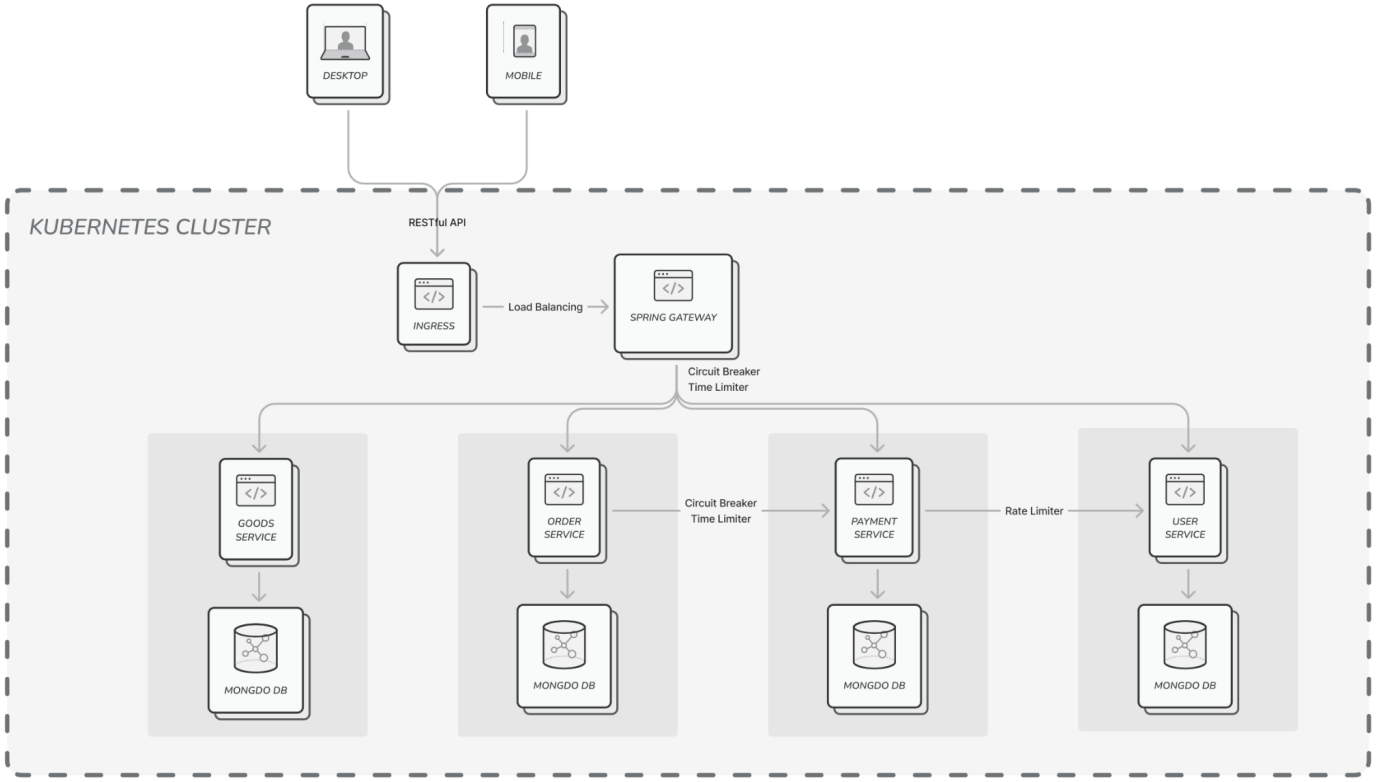
User Service

To simplify our applications, the User Server only provides a login API that verifies the username and password, which is implemented by a GET method (In reality, it is always implemented by the post method). User service is only used in the payment service when a user pays for an order.

A rate limiter is offered in the user service. If there are too many requests to the user service in one second, it will protect the user service through handling the specific number of requests and returning the 429 status code for the rest of requests directly.

Payment Service

The main function of the payment service is to make a payment for an order. This is implemented by firstly testing if the user login is successful, then sending an HTTP POST request to add a new payment record. In the meantime, it adds this record to the MongoDB instance of the payment service. The payment service is in the middle of the service chain between the order service and the user service, so it may propagate some errors from both sides.



System Architecture Diagram

User flow

The standard user flow of this system is to simulate making an online order.

- The user lists goods: make GET /api/goods request to the goods service
- The user submits an order: make PUT /api/order request to the order service
- The order service initiates an order payment: make POST /api/payment request to the payment service
- The payment service verifies a user account: make GET /user/login request to the user service

To test and demonstrate the process, we created a JMeter script that invokes the order service which will trigger another two RESTful service calls internally.

Scalability

1. Vertical Scalability Implemented by Kubernetes

   We set a memory request and a CPU request for each container to determine the minimum number of resources to be allocated to the containers. If a workload is observed to be using more resources than defined in its spec, the VPA will compute a new, more appropriate set of values. The CPU limit and memory limit are also set to determine the maximum number of resources and a container is not allowed to use more than its resource limit. If a process in the container tries to consume more than the allowed amount of resources, the system kernel terminates the process that attempted the allocation, with an out-of-memory error[1]. We decide on the number and limits of requests during the testing phase.

2. Horizontal Scalability Implemented by Kubernetes

   HorizontalPodAutoscaler is implemented as a Kubernetes API resource and controller. This resource determines the behavior of the controller. A horizontal pod autoscaling controller running within the Kubernetes control plane periodically adjusts the expected size of its target (e.g., deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric specified [2].

   We implemented Horizontal Scalability when deploying the order service, by setting the metric server of Kubernetes to monitor the CPU utilization of the first pod of the order service, when the average utilization of the CPU is beyond 60%, it will scale up to two replicas of pods. When the average utilization of CPU is below 60% for a certain period, it will scale down to one replica.

3. Horizontal Scalability of MongoDB cluster

   Compared to relational databases, MongoDB can obtain horizontal scalability more easily as the data is not relationally coupled. This horizontal scaling is realized by sharding and replica sets.

   Sharding is horizontal scaling by spreading data across multiple nodes. Each node contains a subset of the overall data. Replica sets seem similar to sharding, but they differ in that the dataset is duplicated. Replication allows for high availability, redundancy/failover handling, and decreased bottlenecks on read operations. In our project, we rely on these features of MongoDB to realize scaling and fault tolerance in terms of databases.

Fault tolerance

1. Time limiter

   The system has time limiters in two places, one is the order service and the other is the gateway. Because the order service needs to respond in a short time, so we set the time limit here to one second. Therefore, if there is no result within one second, 408 will be returned. Gateway just extensively adds a time limit of eight seconds to each service. If there is no response to a request through the gateway after eight seconds, then a fallback message will be returned to the user, telling the user that there is a problem with the service.

2. Circuit breaker

   Order service is critical since it relies on downstream services which rely on other services. It is important to stop the request to downstream services if they are largely failing or responding slowly. Therefore we apply a circuit breaker on the payment service calls, so that in case of a surge of errors in the payment service, the order service will stop making new requests to it and allow the payment service to recover. Then the circuit breaker will try a couple of requests in the half-open state and close itself once the service is back.

3. Fallback

   For the gateway, any internal server error or timeout of other services will result in a fallback message, which will be presented to the user. For order service, a fallback will be returned only if it fails to call payment service, and this fallback can only be seen by order service.

4. Rate limiter

A rate Limiter is implemented in user service. It enables fault tolerance in our applications, making our applications more stable and robust. In the UserRestController code, we set the specific number of requests allowed to be processed in a second. If the number of requests per second is more than the specified number, then the excess will be returned directly to Response code 429, which indicates the user has sent too many requests in a given amount of time.

5. Restart Policy

   As we mentioned above, the java process of service in the container may be terminated when the process in the container tries to consume more than the allowed resources. Then the livenessProbe determines that applications running within containers are not operational, and the probe will fail and kubelet kills the container. In this case, the container is subjected to its restart policy [3]. The restart policy hasn't been mentioned in our YAML file, cause the restart policy is always by default which is in line with our purpose. The Restart Policy of Kubernetes also enables the fault tolerance of our distributed applications.

Contributions

Yu Long

- Research SpringCloud, Circuit Breaker, Service Discovery, Kubernetes, kind, OpenApi Generator
- Develop order service and integrate with payment service with circuit breaker
- Create JMeter script for testing
- Create Kubernetes deployment scripts
- Adjust deployment configurations and Record demonstration video
- Record audio for circuit breaker, horizontal scaling and rate limiter

Zeyu Bai

- Research on MongoDB's feasibility for the project
- Develop Goods Service
- Develop Gateway Server integrated with Resilience4j
- Implement Kubernetes Ingress load balancer between client and gateway
- Part of audio recording of the video

Wendi Song

- Research how to use Kubernetes to achieve scalability and fault tolerance for micro services
- Develop User Service and implement Rate Limiter with Bucket4j
- Deploy metrics server of Kubernetes and implement horizontal scalability and vertical scalability of Kubernetes
- Part of audio recording of the video
- Edit video

Yating Zhang

- Research on Feign and Netflix Eureka.
- Develop payment service and integrate with user service
- Create JMeter script for testing the circuit breaker in payment service(but didn't implement circuit breaker in the end)
- Part of the report writing

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

1. Integrate Eureka with Kubernetes for service discovery

When we did the proof-of-concept (POC), we didn't consider Kubernetes. So when we started to deploy applications into the Kubernetes cluster, we found Eureka didn't work because the IP addresses of services are not reachable. Thus, we decided to find a solution to integrate Eureka with Kubernetes. There is a project in Spring Cloud that does this, however, it adds more complexity to the design. What's more, we were not able to set it up in the development environment.

After researching the Kubernetes service, we found it is also capable of service discovery and service side load balancing. Therefore, we gave up Eureka and directly used the Kubernetes Service as the service discovery approach.

2. Implement load balancing between gateway and client when using Kubernetes port forwarding

   If we directly forward the gateway Service to localhost, all requests will go to one pod although there are two pods for this Service due to the Kubernetes port forwarding default setting. Later we found that we can solve this problem by adding an additional Ingress-Nginx layer before the gateway.

3. Integration with user service

   When doing the integration with the user service, Feign Client couldn't locate the IP address of the user service successfully, after examining the code and debugging with Yu together, we found the name of user.name registered in Feign is an environment configure name which resolves to the username of the host operating system. To fix it, we changed the configuration key of the user service to a unique name of "userservice.name".

What would you have done differently if you could start again?

1. In the research and POC stage, we should consider Kubernetes so that we can identify the issue with IP addresses of service discovery described above, instead of causing design change in the development phase.
2. Customize the metrics server of Kubernetes and let it monitor the number of packets received per second for each pod to achieve scalability. Because when we start the container, the Java thread may consume a lot of CPU resources, there is a probability that it reaches the target to scale up replicas, but it is not intended. The number of packets received per second for each pod would be the better option for target reference for horizontal scalability.

What have you learnt about the technologies you have used? Limitations? Benefits?

1. One thing I have learned about MongoDB is the benefits of the non-relational database with regard to the relational database. As a non-relational database, MongoDB can easily scale up as it doesn't need to join tables together. Also, it is suitable for web application development because of the flexible schema and the ability of scalability and fault tolerance. However, the limitation of MongoDB is obvious: it doesn't support join. Looking up a collection can be very time-consuming.
2. Kubernetes is powerful and flexible. We implement all of the features we needed for this project. However, the learning curve is very high. We spent a lot of time understanding the concepts and configurations. That being said, it is a good tool to implement a distributed system. With the assistance of tools such as kind and k9s, we can easily set up a local environment in a few minutes.

Reference

[1] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[2] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[3] https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy

[4] https://kubernetes.io/docs/concepts/services-networking/service/

[5] Lecture Notes: T7.1-REST-Intro.pptx