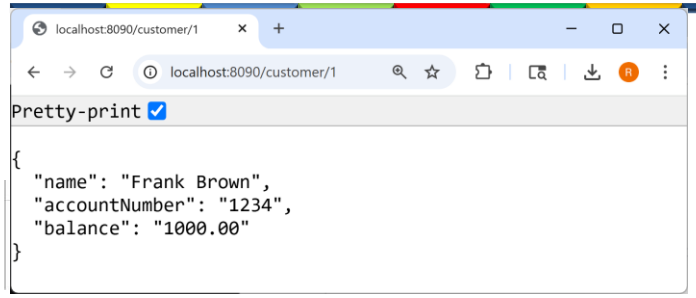
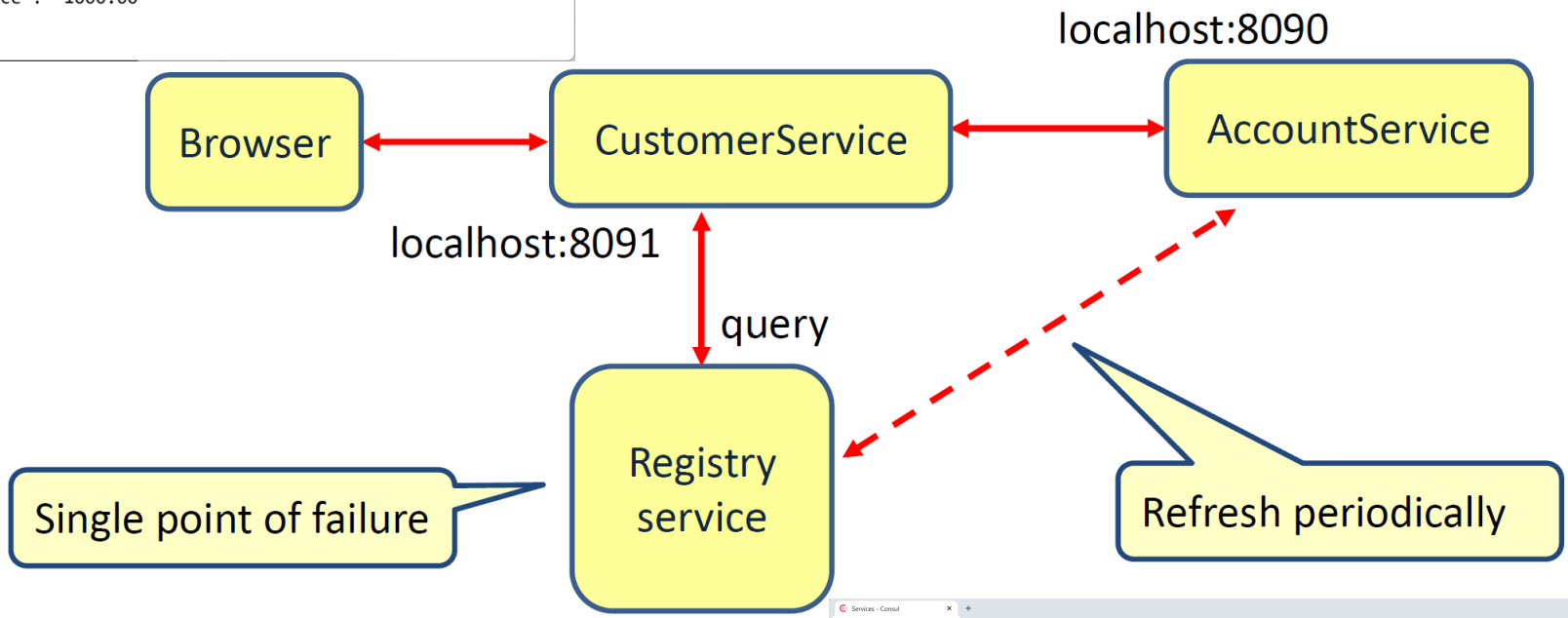
Lesson 7

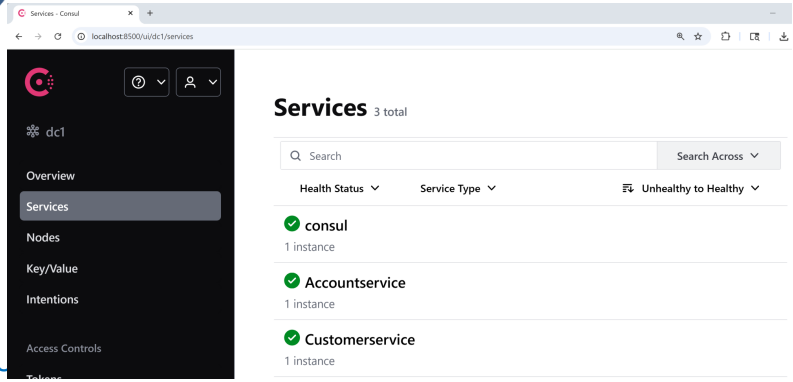
MICROSERVICES

SERVICE REGISTRY: CONSUL

Registry



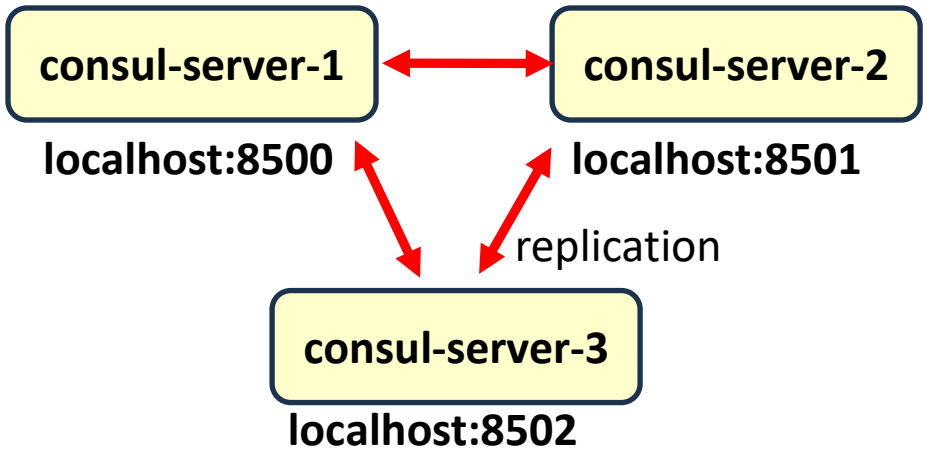




Consul cluster

```
version: '3.8'
services:
  consul-server-1:
    image: hashicorp/consul:latest
    container_name: consul-server-1
    command: "agent -server -bootstrap-expect=3 -client=0.0.0.0 -ui -retry-join=consul-server-2 -retry-join=consul-server-3"
    ports:
      - "8500:8500" # UI and HTTP API
      - "8600:8600/udp" # DNS
    networks:
      - consul-net

  consul-server-2:
    image: hashicorp/consul:latest
    container_name: consul-server-2
    command: "agent -server -client=0.0.0.0 -ui -retry-join=consul-server-1 -retry-join=consul-server-3"
    ports:
      - "8501:8500" # UI and HTTP API (different host port to avoid conflict)
    networks:
      - consul-net

  consul-server-3:
    image: hashicorp/consul:latest
    container_name: consul-server-3
    command: "agent -server -client=0.0.0.0 -ui -retry-join=consul-server-1 -retry-join=consul-server-2"
    ports:
      - "8502:8500" # UI and HTTP API (different host port to avoid conflict)
    networks:
      - consul-net

networks:
  consul-net:
    driver: bridge
```

Consul cluster



AccountService

application.yml

```
spring:
  application:
    name: Accountservice
  cloud:
    consul:
      host: localhost
      port: 8501

server:
  port: 8091
```

Register in Consul on port 8501

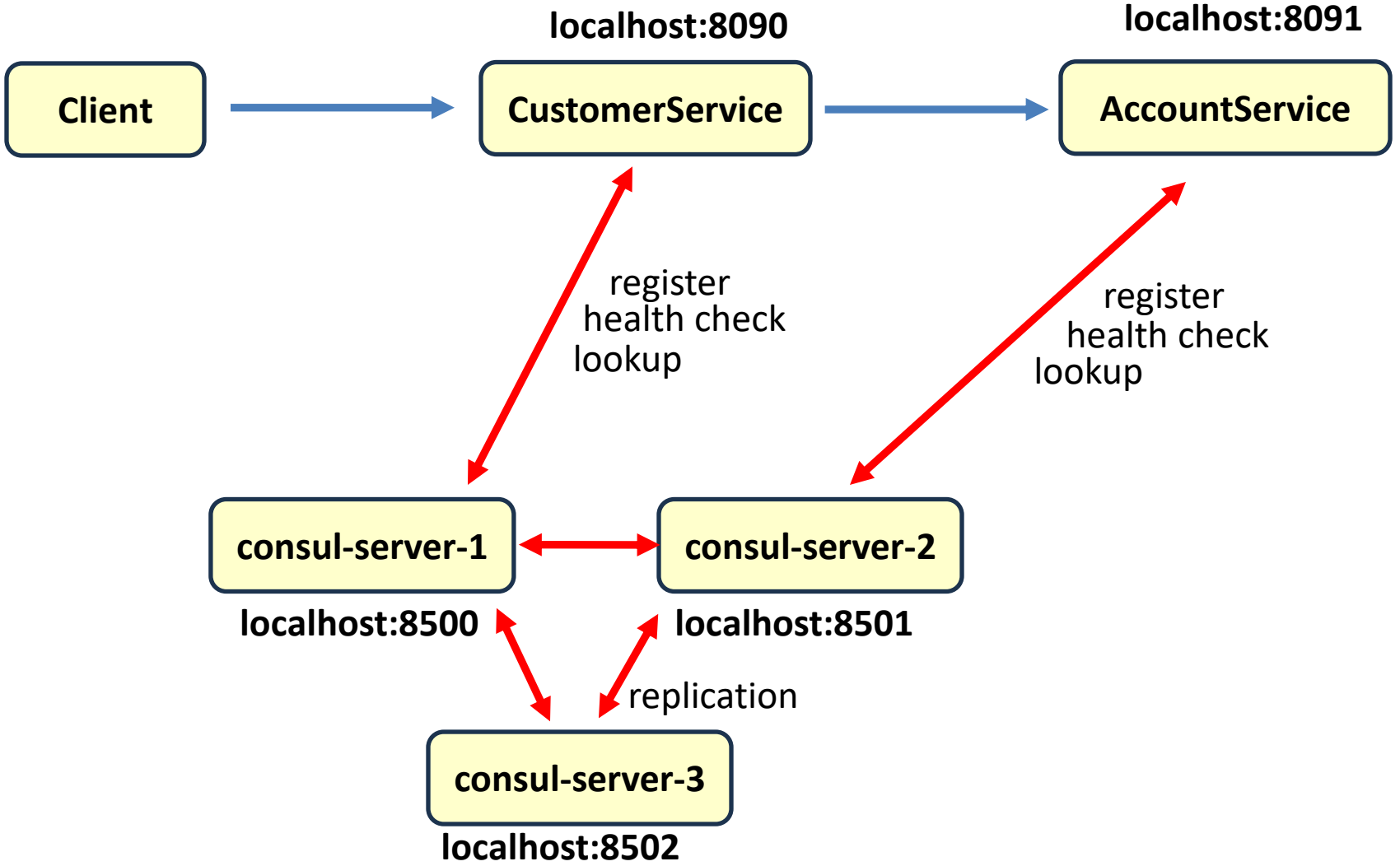
CustomerService configuration

application.yml

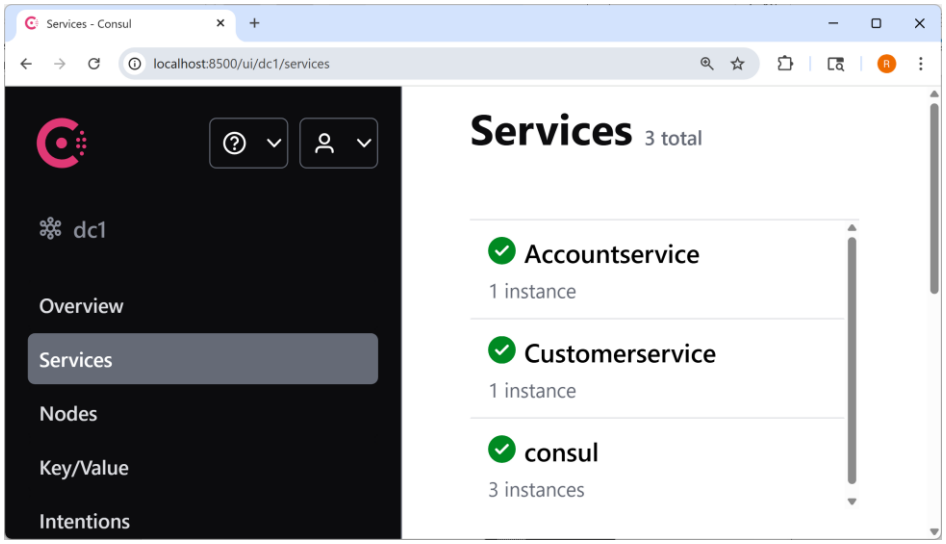
```
spring:
  application:
    name: Customerservice
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        enabled: true
        prefer-ip-address: true
        instance-id: ${spring.application.name}:${random.value}


server:
  port: 8090
```

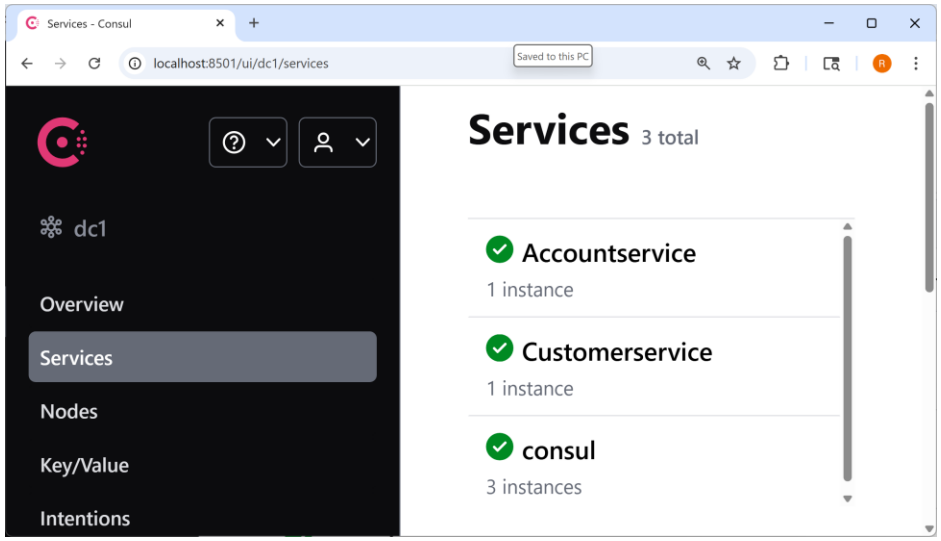
Register in Consul on port 8500

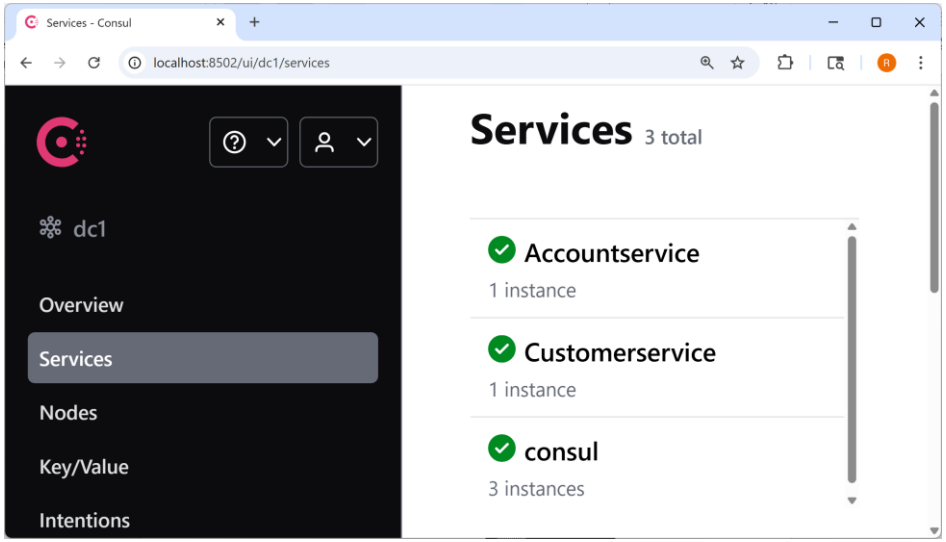
Consul cluster and replication



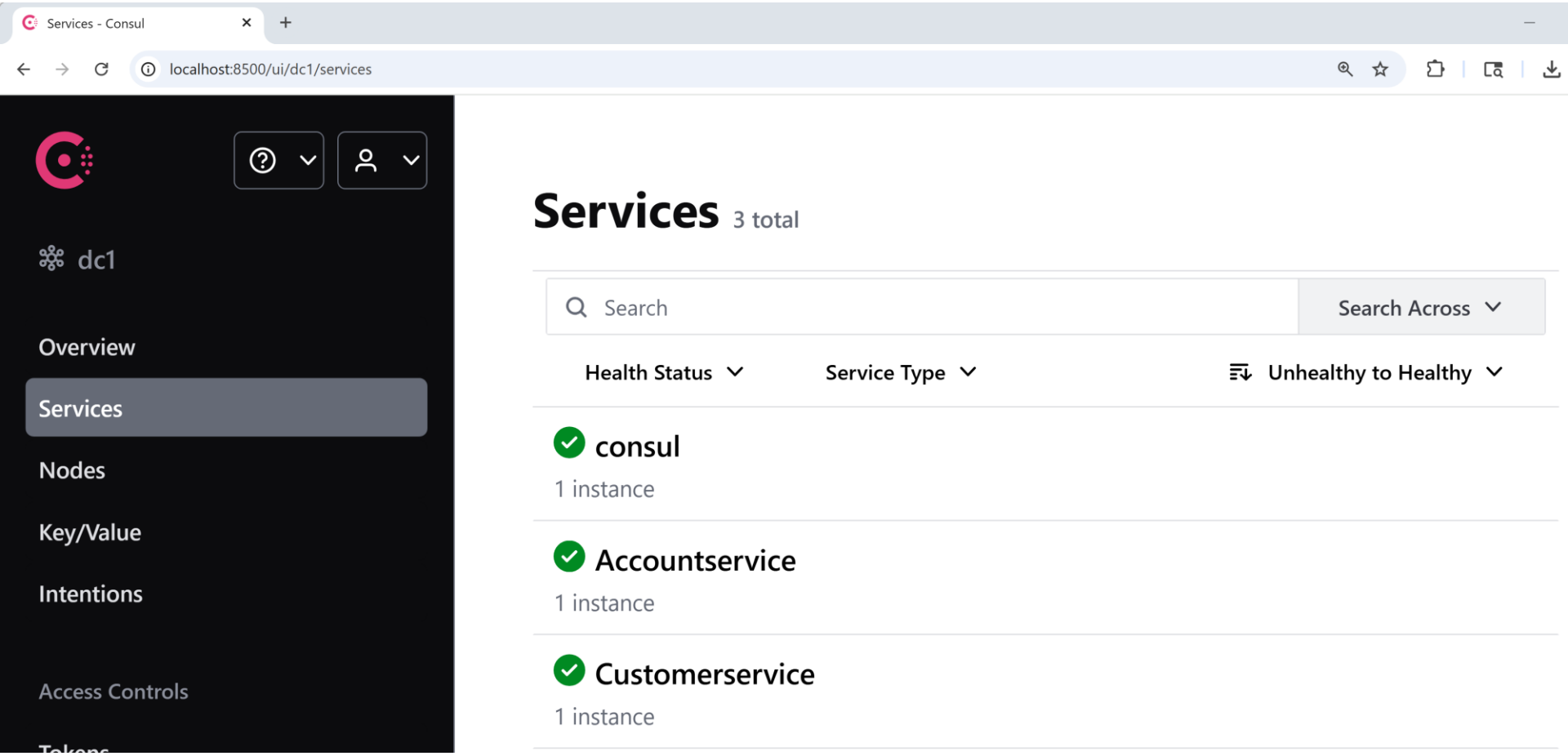
Replication



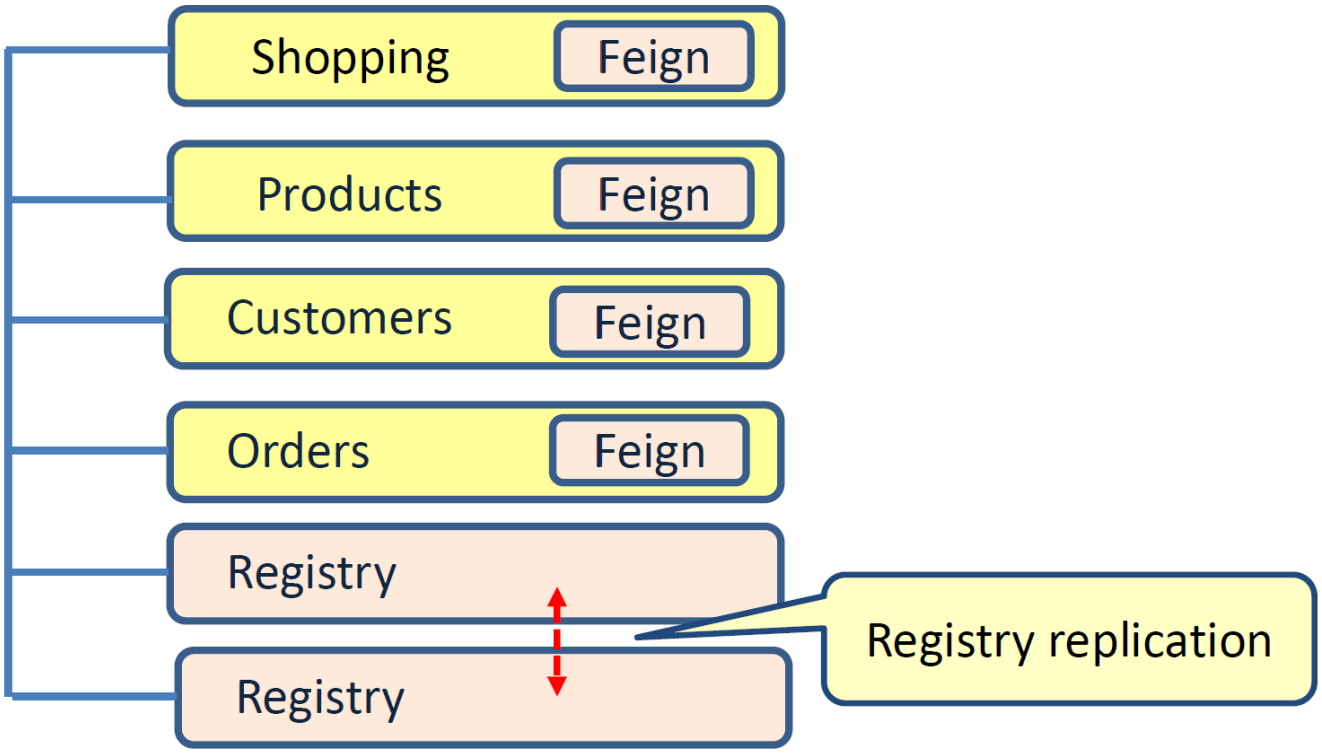




Running the CustomerService



Implementing microservices

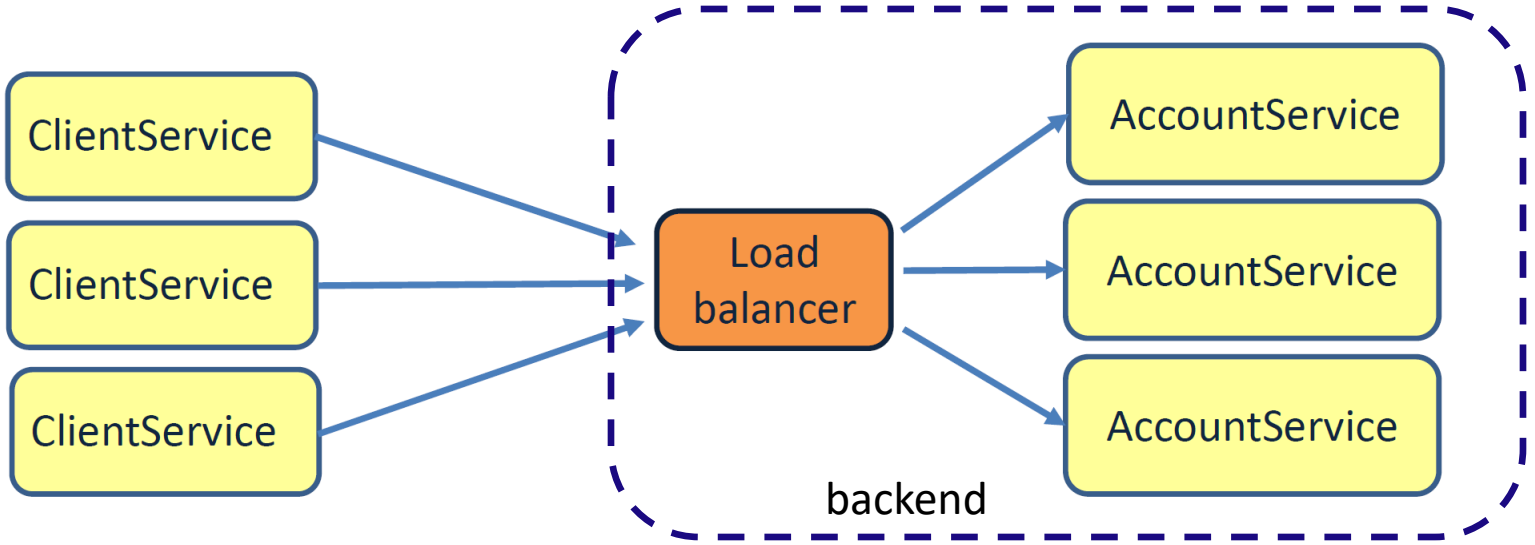


Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry
Performance	
Resilience	Registry and replicas
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

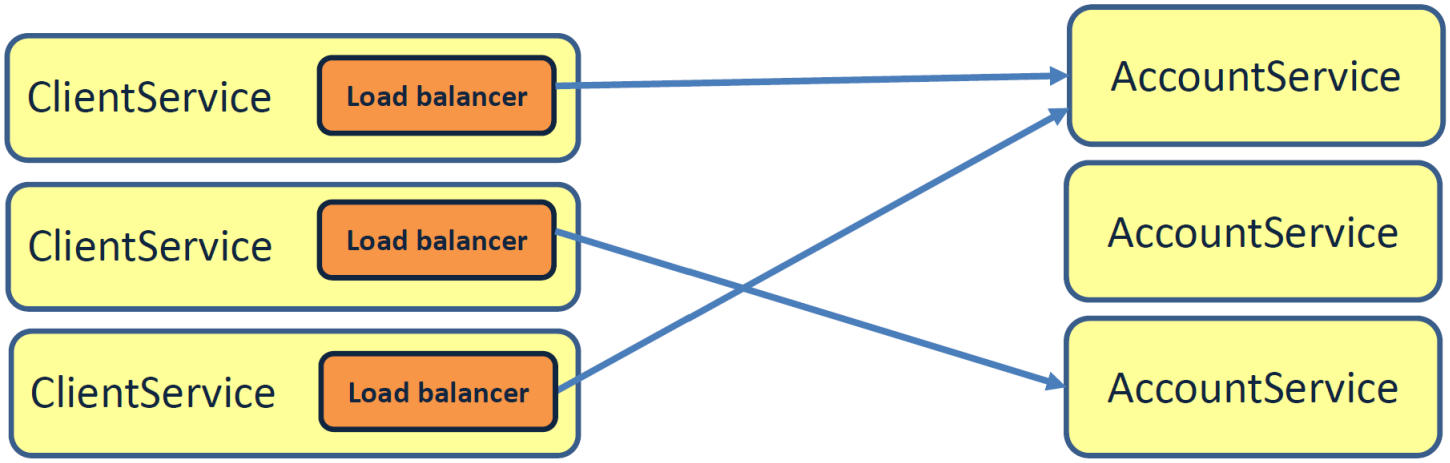
LOAD BALANCING

Server side load balancing



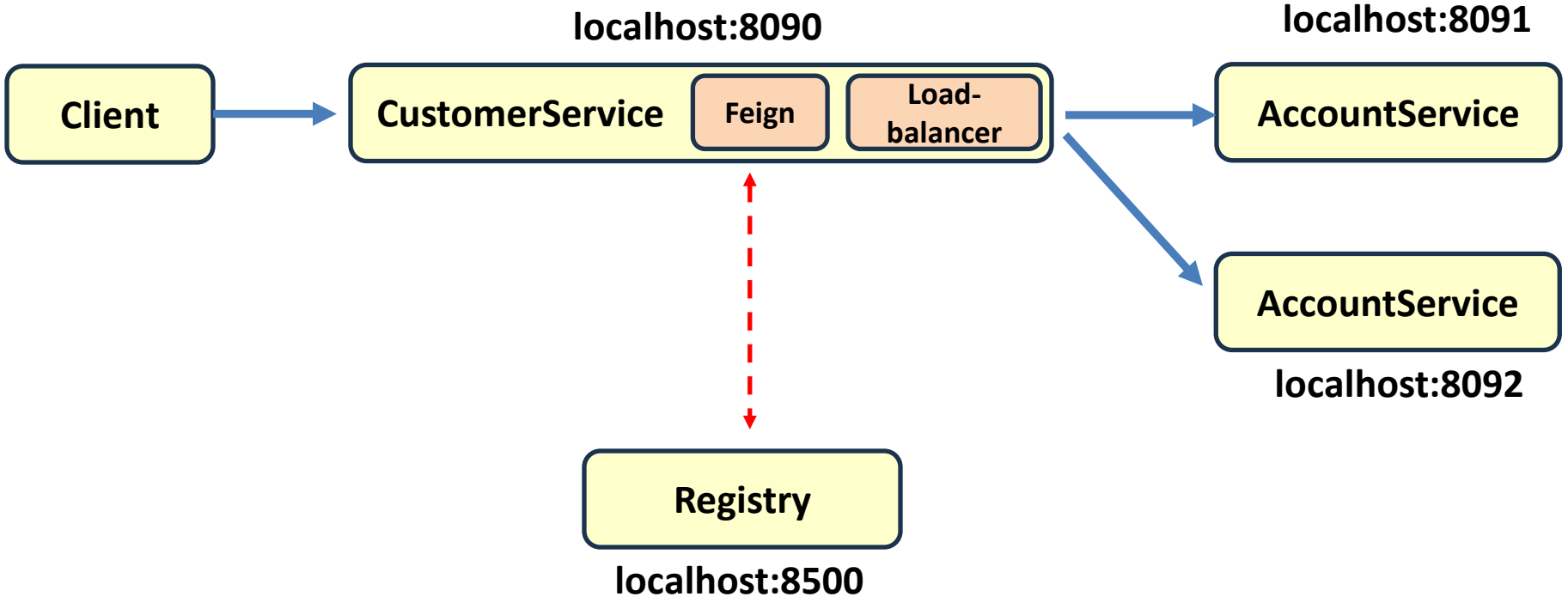
- Single point of failure
- If we add a new instance of AccountService, we need to reconfigure the load balancer
- Extra hop (performance)
- Every microservice needs its own load balancer
- Same load balance algorithm for every client
- Scaling limitation, load balance can handle only a certain number of requests

Client side load balancing



- No single point of failure
- Simplifies service management
- Only one hop (performance)
- Auto discovery with registry based lookup (flexibility)
- Every client can use its own load balancing algorithm
- Unlimited scalable

Spring cloud load balancer



Accountservice 1

```
@RestController
public class AccountController {
   @RequestMapping("/account/{customerid}")
   public Account getAccount(@PathVariable("customerid") String customerId) {
      return new Account("1234", "1000.00");
   }
}
```

1000.00

```
spring:
  application:
    name: Accountservice
  cloud:
    consul:
      host: localhost
      port: 8501

server:
  port: 8091
```

Same name, different port

Accountservice 2

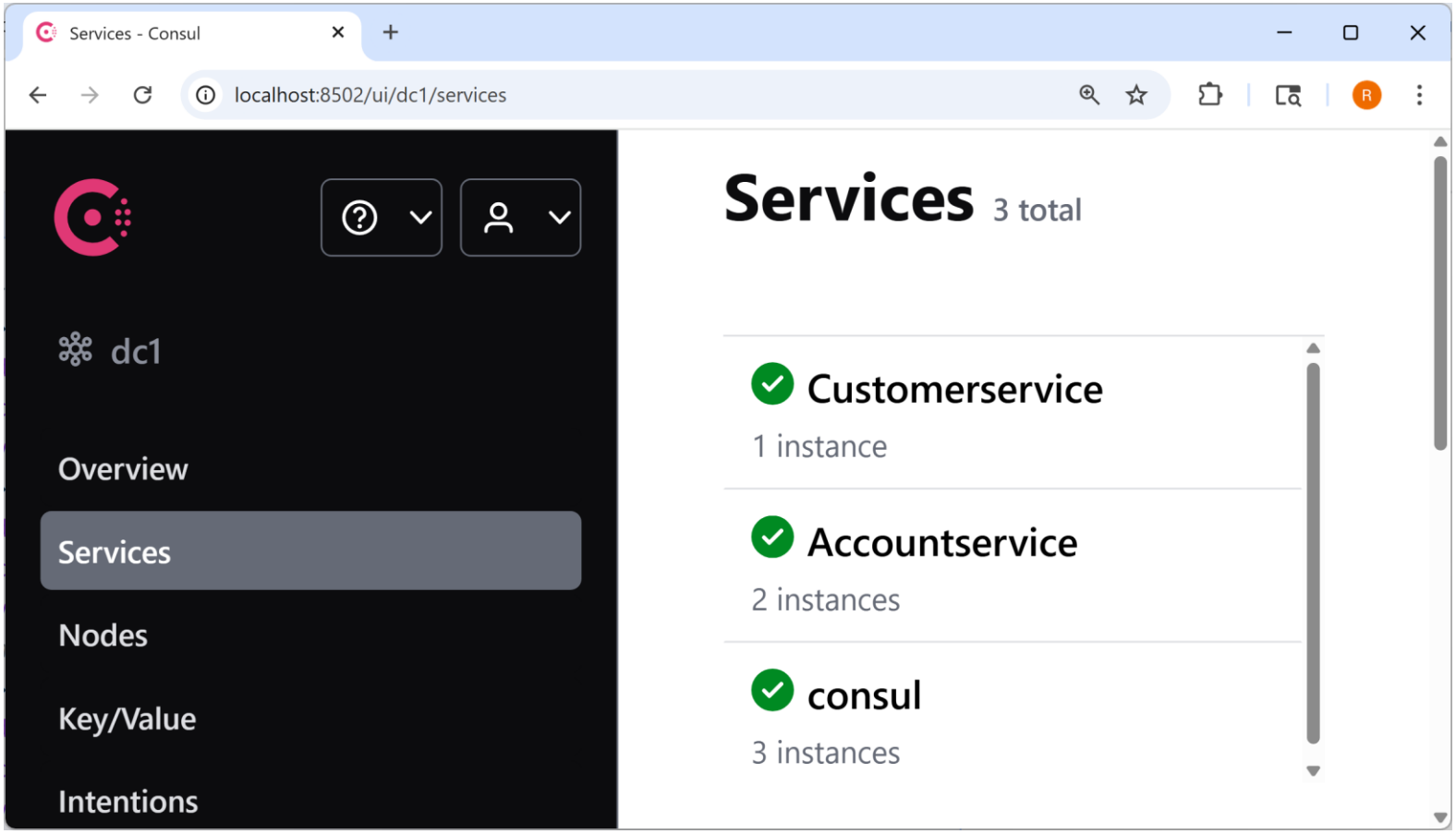
```
@RestController
public class AccountController {
   @RequestMapping("/account/{customerid}")
   public Account getAccount(@PathVariable("customerid") String customerId) {
      return new Account("1234", "1000.00");
   }
}
```

2000.00

```
spring:
  application:
    name: Accountservice
  cloud:
    consul:
      host: localhost
      port: 8501

server:
  port: 8092
```

Same name, different port

Two account services

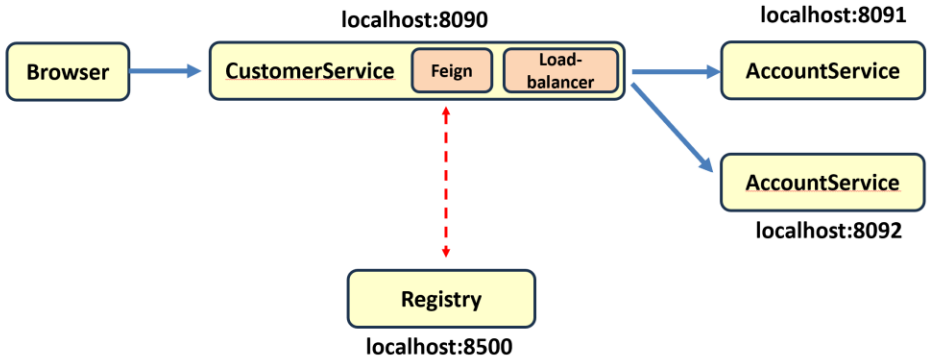


CustomerService: the controller

```
@RestController
public class CustomerController {
   @Autowired
   AccountFeignClient accountClient;

   @RequestMapping("/customer/{customerid}")
   public Customer getName(@PathVariable("customerid") String customerId) {
      Account account = accountClient.getName(customerId);
      return new Customer("Frank Brown", account.accountNumber(), account.balance());
   }

   @FeignClient("Accountservice")
   interface AccountFeignClient {
      @RequestMapping("/account/{customerid}")
      public Account getName(@PathVariable("customerid") String customerId);
   }
}
```

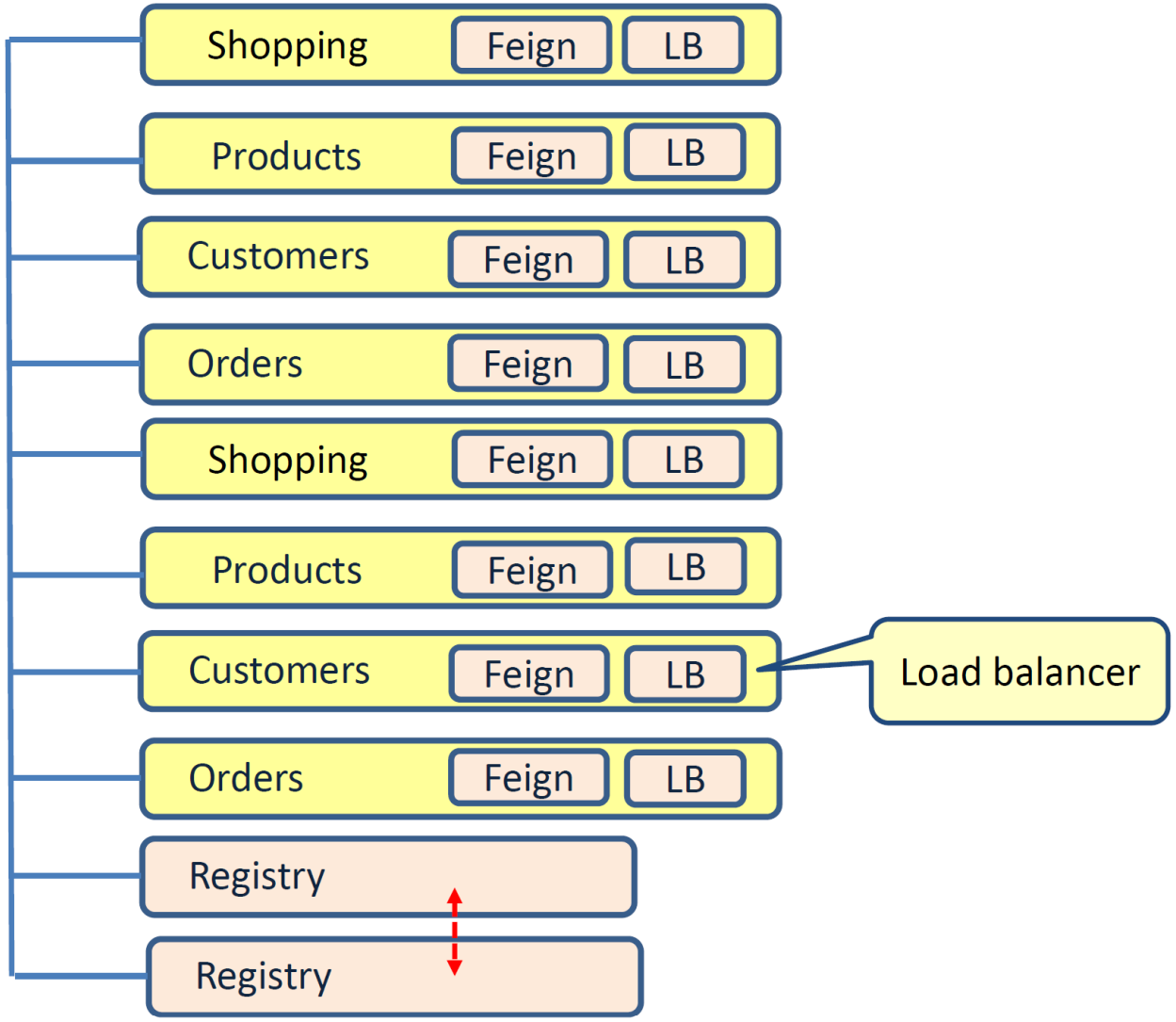
Feign automatically uses Spring cloud load balancer together with the Registry

Round robin load balancing



Feign does automatically load balance the calls using the registry

Implementing microservices

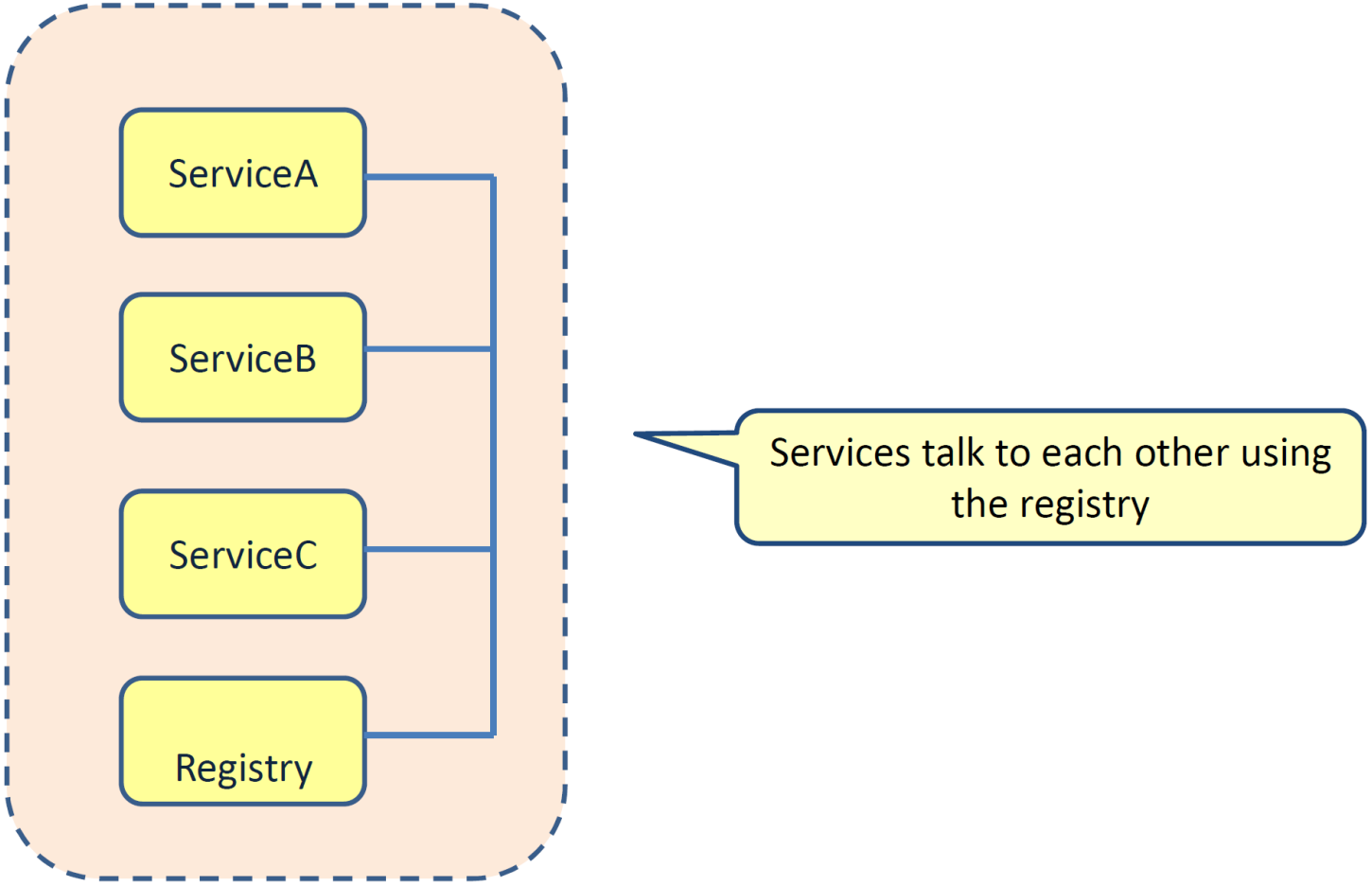


Challenges of a microservice architecture

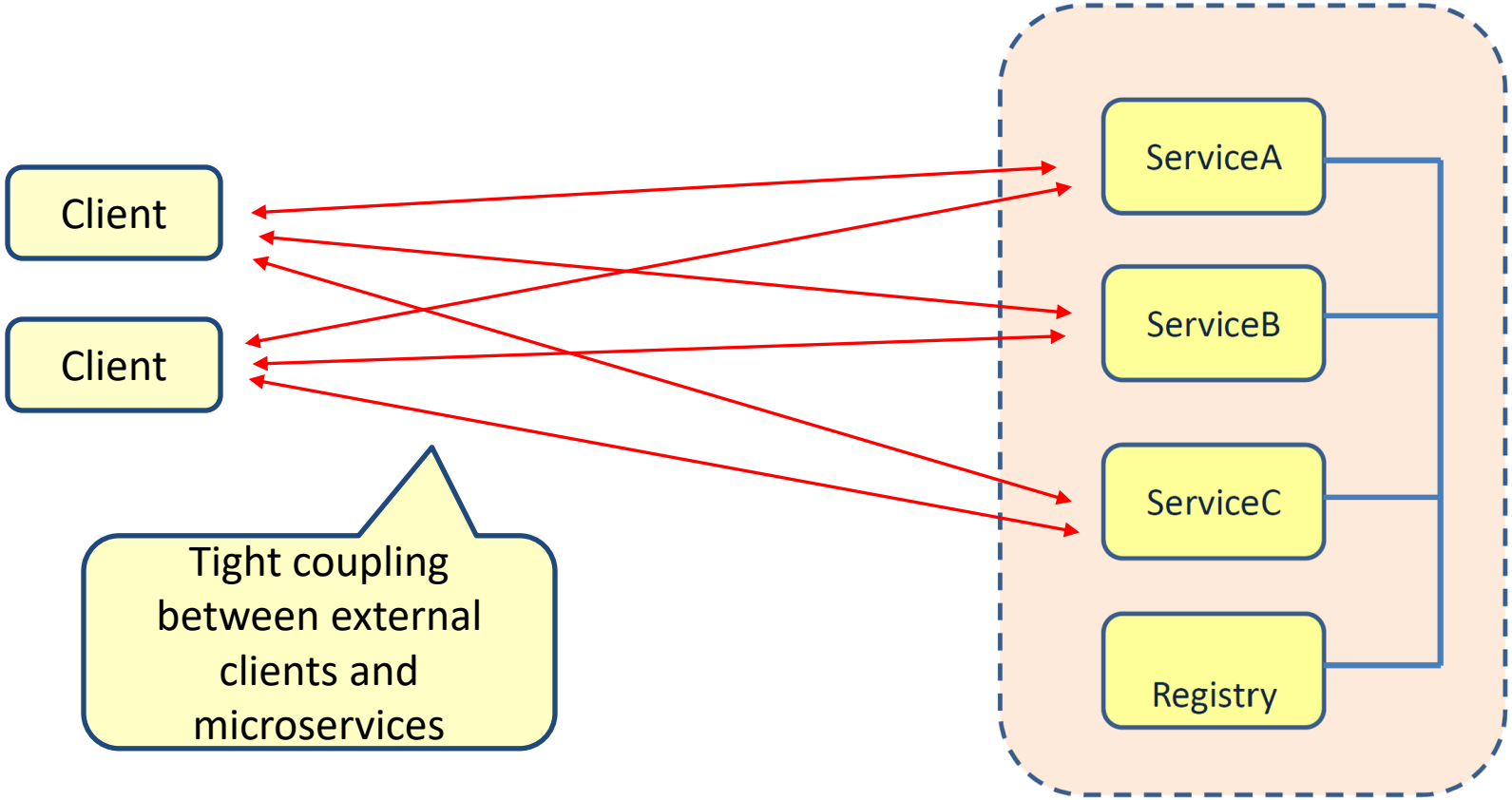
Challenge	Solution
Complex communication	Feign Registry
Performance	
Resilience	Registry and replicas Load balancing between multiple service instances
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

API GATEWAY

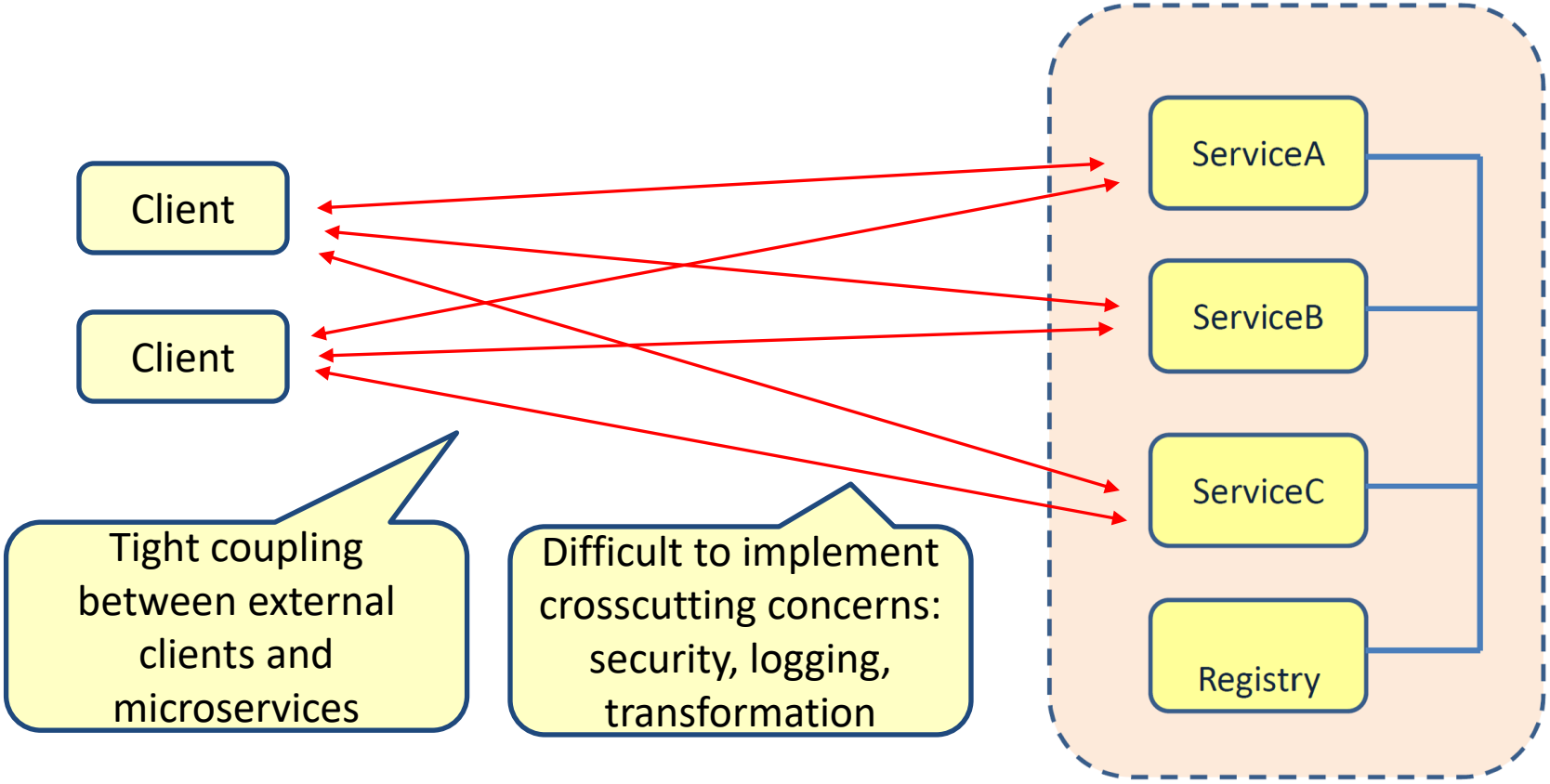
Microservice architecture



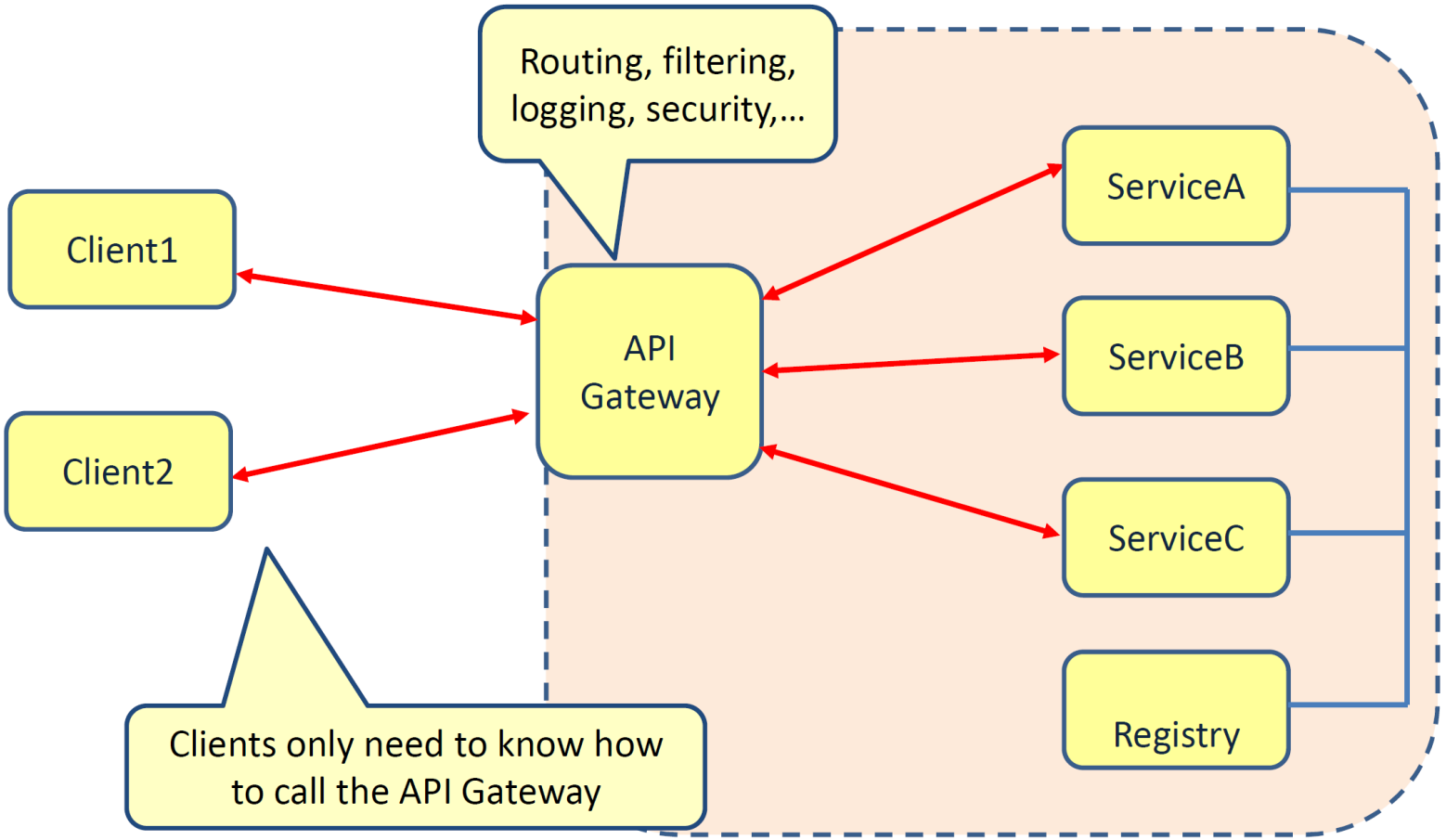
Adding external clients



Adding external clients



API Gateway


Routing, filtering, logging, security,…
Client1
Client2
API Gateway
ServiceA
ServiceB
ServiceC
Registry
Clients only need to know how to call the API Gateway

StudentService

```
@RestController
public class StudentController {
   @RequestMapping("/students/{studentid}")
   public String getStudent(@PathVariable("studentid") String studentId) {
      return "Frank Brown";
   }
}
```

```
spring:
  application:
    name: Studentservice
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        enabled: true
        prefer-ip-address: true
        instance-id: ${spring.application.name}:${random.value}


server:
  port: 8095
```

GradingService

```
@RestController
public class GradingController {
   @RequestMapping("/grades/{studentid}/{courseid}")
   public String getGrade(@PathVariable("studentid") String studentId,
                          @PathVariable("courseid") String courseId) {
      return "A+";
   }
}
```

```
spring:
  application:
    name: Gradingservice
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        enabled: true
        prefer-ip-address: true
        instance-id: ${spring.application.name}:${random.value}


server:
  port: 8096
```

API Gateway

```
@SpringBootApplication
@EnableDiscoveryClient
public class ApiGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway-server-
webflux</artifactId>
</dependency>
```

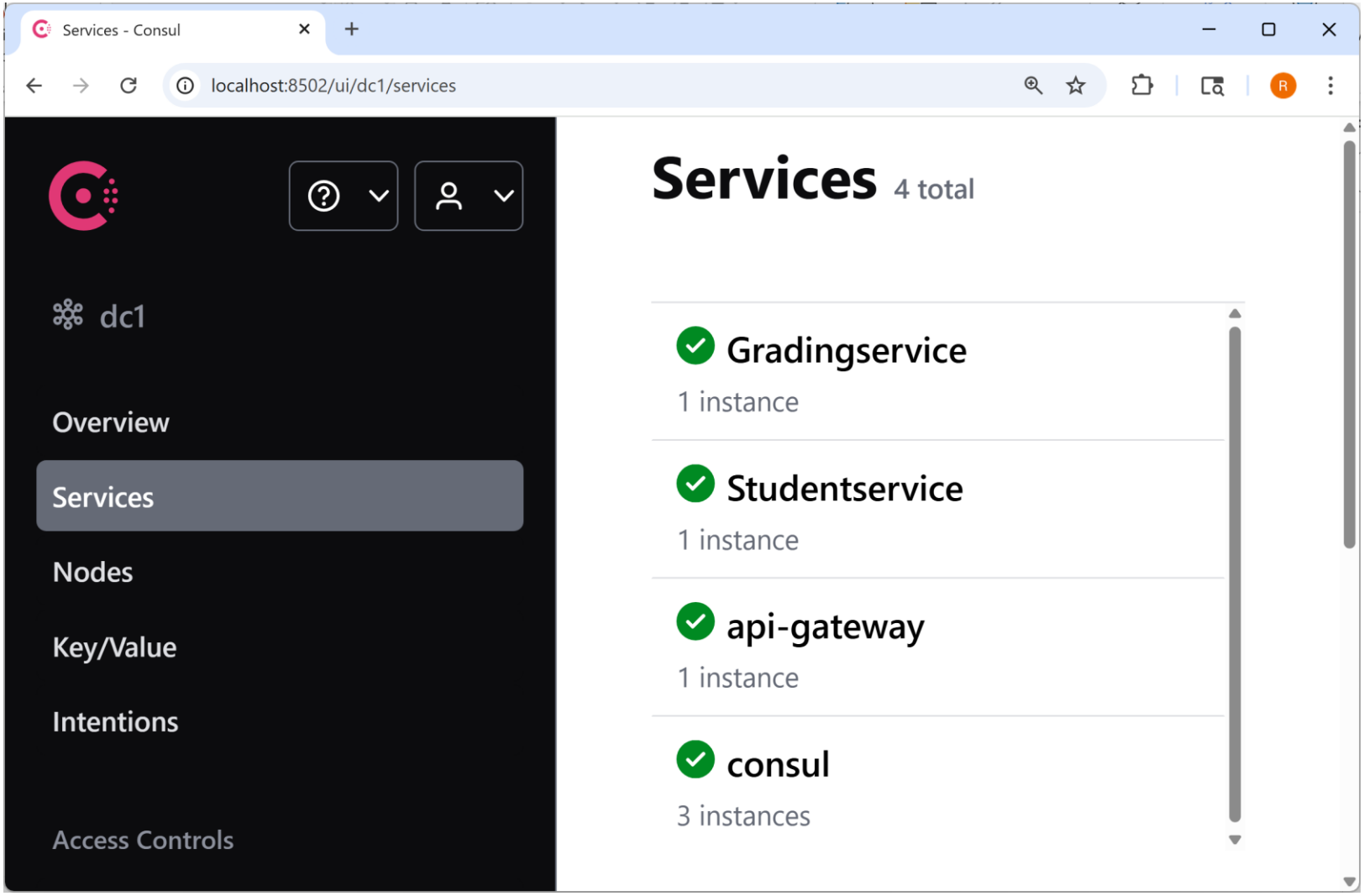
API Gateway

```
spring:
  application:
    name: api-gateway
  cloud:
    consul:
      host: localhost
      port: 8500
    gateway:
      server:
        webflux:
          routes:
            - id: studentModule
              uri: lb://Studentservice
              predicates:
                - Path=/students/**
            - id: gradingModule
              uri: lb://Gradingservice
              predicates:
                - Path=/grades/**
server:
  port: 8080
```

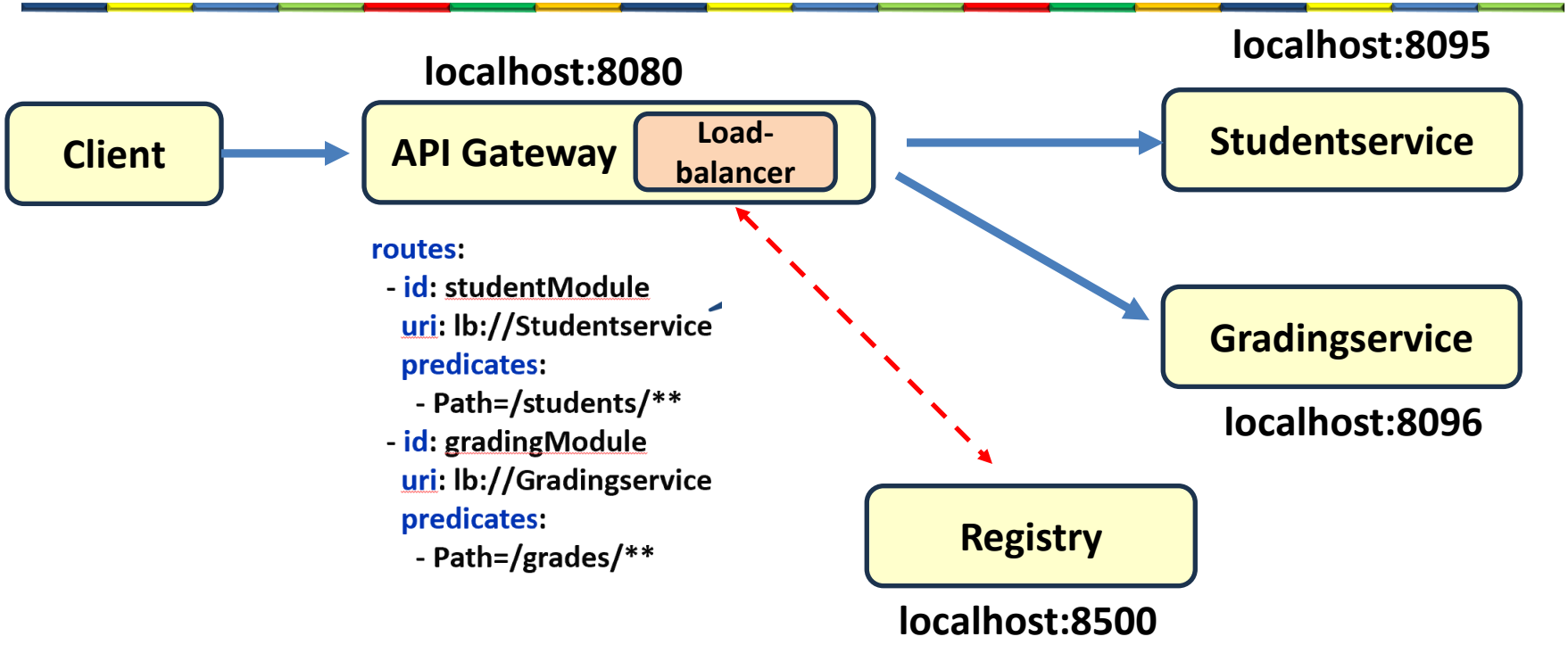
Configure name of the service, so registry is used

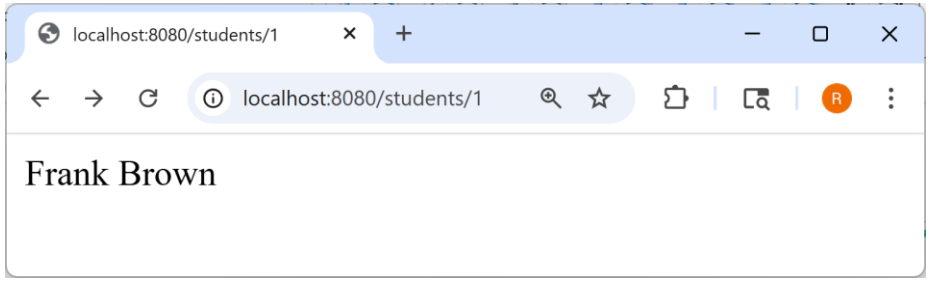
lb is load balancing

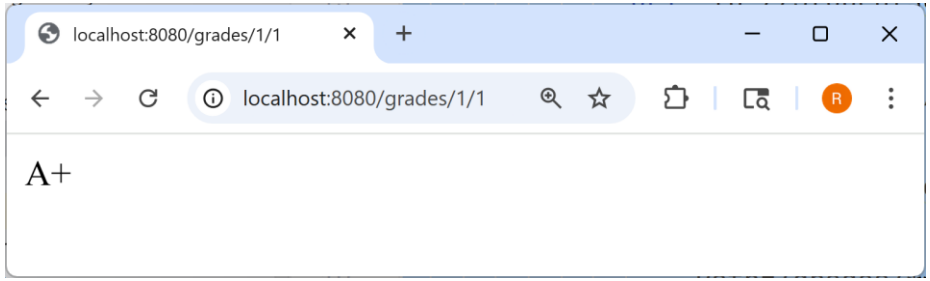
API Gateway


Services - Consul
localhost:8502/ui/dc1/services
dc1
Overview
Services
Nodes
Key/Value
Intentions
Access Controls
Services 4 total
Gradingservice
1 instance
Studentservice
1 instance
api-gateway
1 instance
consul
3 instances

API Gateway


localhost:8095
localhost:8080
Client
API Gateway
Load-balancer
Studentservice
routes:
- id: studentModule
uri: lb://Studentservice
predicates:
- Path=/students/**
- id: gradingModule
uri: lb://Gradingservice
predicates:
- Path=/grades/**
Gradingservice
localhost:8096
Registry
localhost:8500


localhost:8080/students/1
localhost:8080/students/1
Frank Brown


localhost:8080/grades/1/1
localhost:8080/grades/1/1
A+

Java based config

```
@Configuration
public class BeanConfig {
   @Bean
   public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
      return builder.routes()
            .route(r -> r.path("/students/**")
                  .uri("lb://StudentService"))
            .route(r -> r.path("/grades/**")
                  .uri("lb://GradingService"))
            .build();
   }
}
```

Build-in predicates

Name	Description	Example
After Route	It takes a date-time parameter and matches requests that happen after it	After=2017-11-20T...
Before Route	It takes a date-time parameter and matches requests that happen before it	Before=2017-11-20T...
Between Route	It takes two date-time parameters and matches requests that happen between those dates	Between=2017-11-20T..., 2017-11-21T...
Cookie Route	It takes a cookie name and regular expression parameters, finds the cookie in the HTTP request's header, and matches its value with the provided expression	Cookie=SessionID, abc.
Header Route	It takes the header name and regular expression parameters, finds a specific header in the HTTP request's header, and matches its value with the provided expression	Header=X-Request-Id, \d+
Host Route	It takes a hostname ANT style pattern with the . separator as a parameter and matches it with the Host header	Host=**.example.org
Method Route	It takes an HTTP method to match as a parameter	Method=GET
Path Route	It takes a pattern of request context path as a parameter	Path=/account/{id}
Query Route	It takes two parameters—a required param and an optional regexp and matches them with query parameters	Query=accountId, 1.
RemoteAddr Route	It takes a list of IP addresses in CIDR notation, like 192.168.0.1/16, and matches it with the remote address of a request	RemoteAddr=192.168.0.1/16

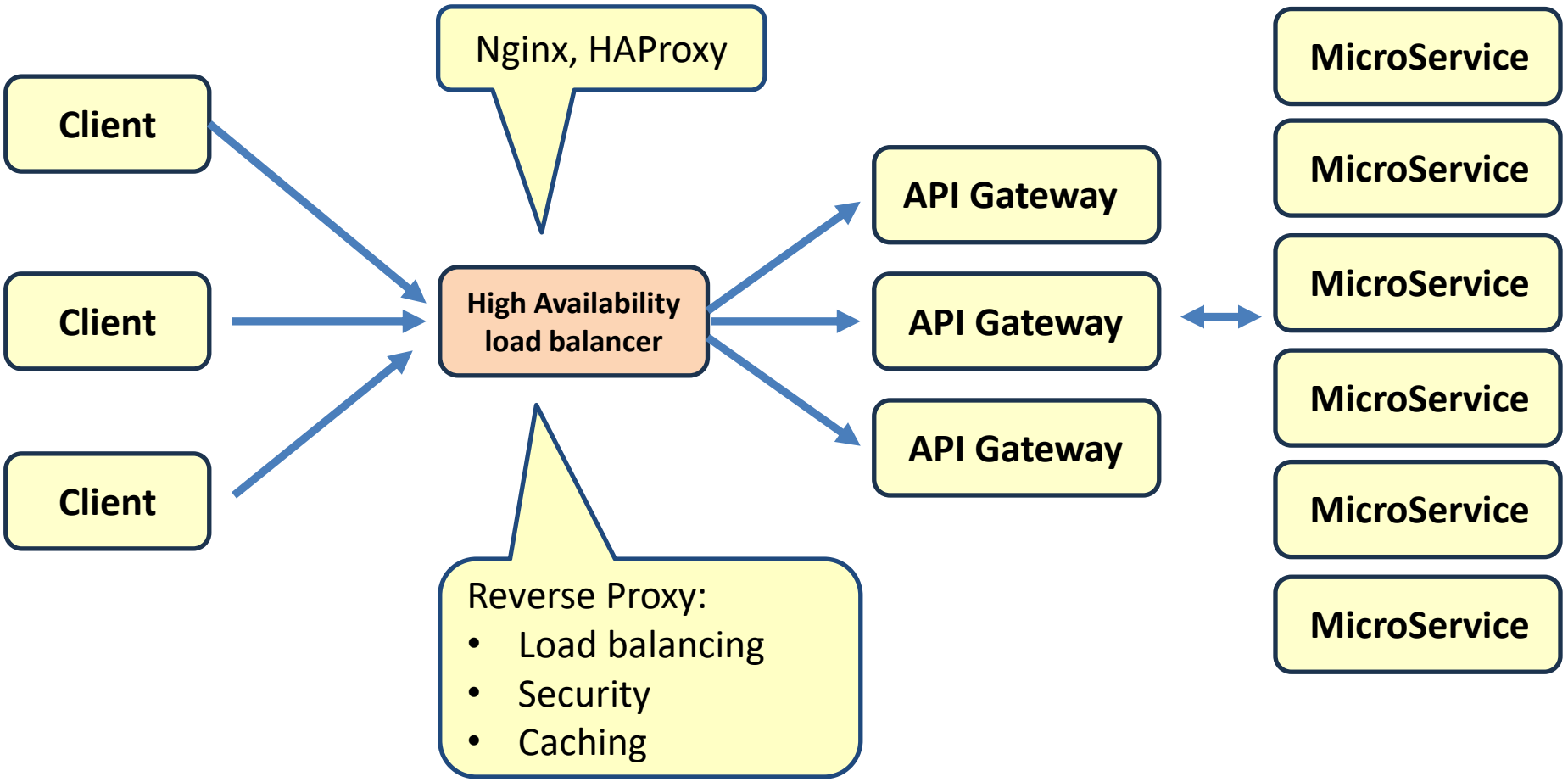
Custom Global filter

- Global filter is applied to all routes

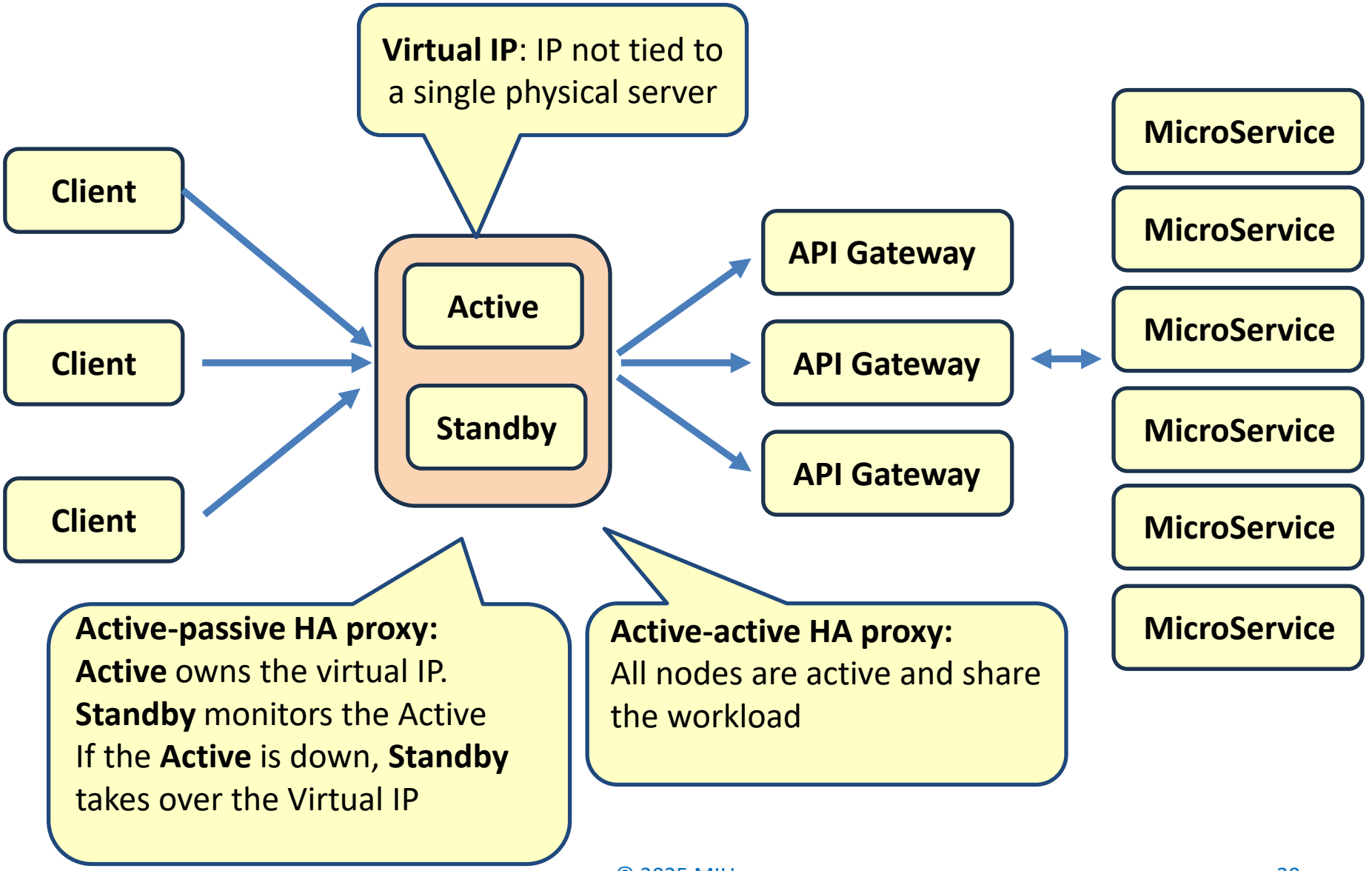
```
@Component
public class PreLastPostGlobalFilter
      implements GlobalFilter {

   @Override
   public Mono<Void> filter(ServerWebExchange exchange,
                            GatewayFilterChain chain) {
      System.out.println("Pre Global Filter "+exchange.getRequest().getURI());
      return chain.filter(exchange)
            .then(Mono.fromRunnable(() -> {
               System.out.println("Post Global Filter "+exchange.getResponse().getStatusCode());
            }));
   }
}
```

```
Pre Global Filter http://localhost:8080/students/1
Post Global Filter 200 OK
```

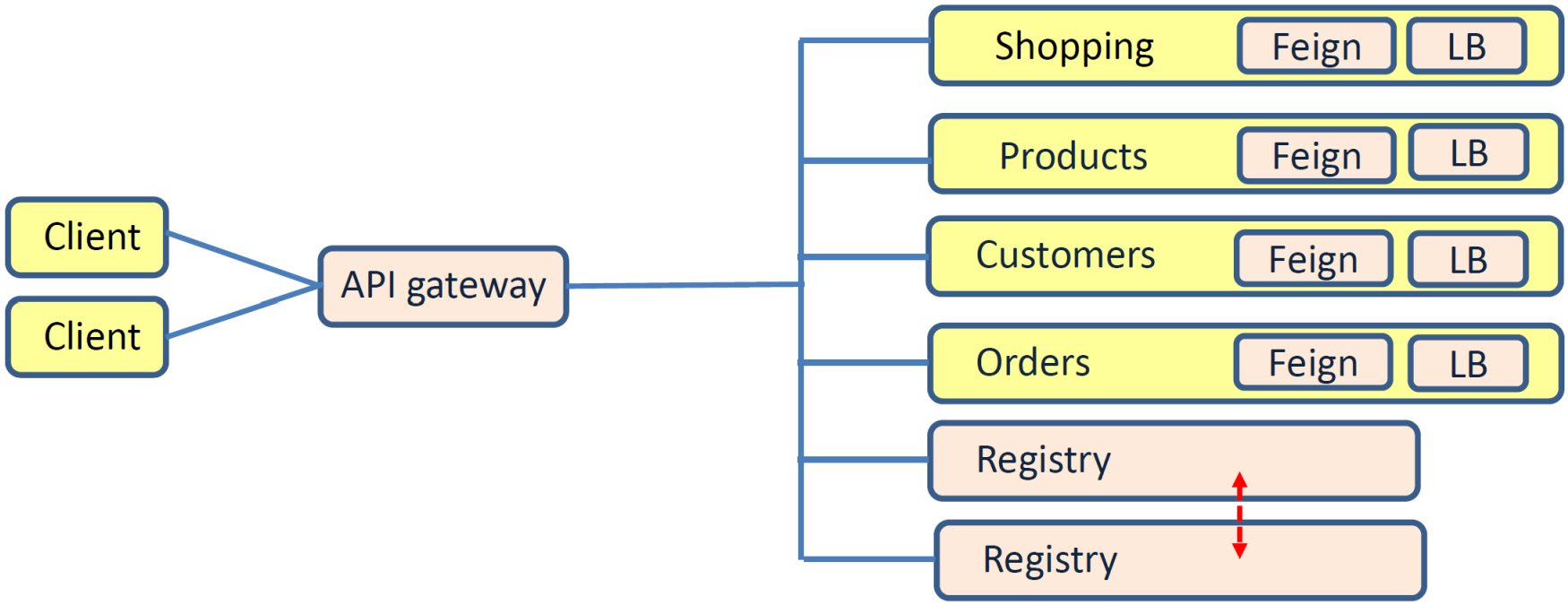
Avoid single point of failure



Avoid single point of failure



Implementing microservices



Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API Gateway
Performance	
Resilience	Registry and replicas Load balancing between multiple service instances
Security	
Transactions	
Following the process	
Keep data in sync	
Keep interfaces in sync	
Keep configuration in sync	
Monitor health of microservices	
Follow/monitor business processes	

Consul cluster and replication