NETFLIX

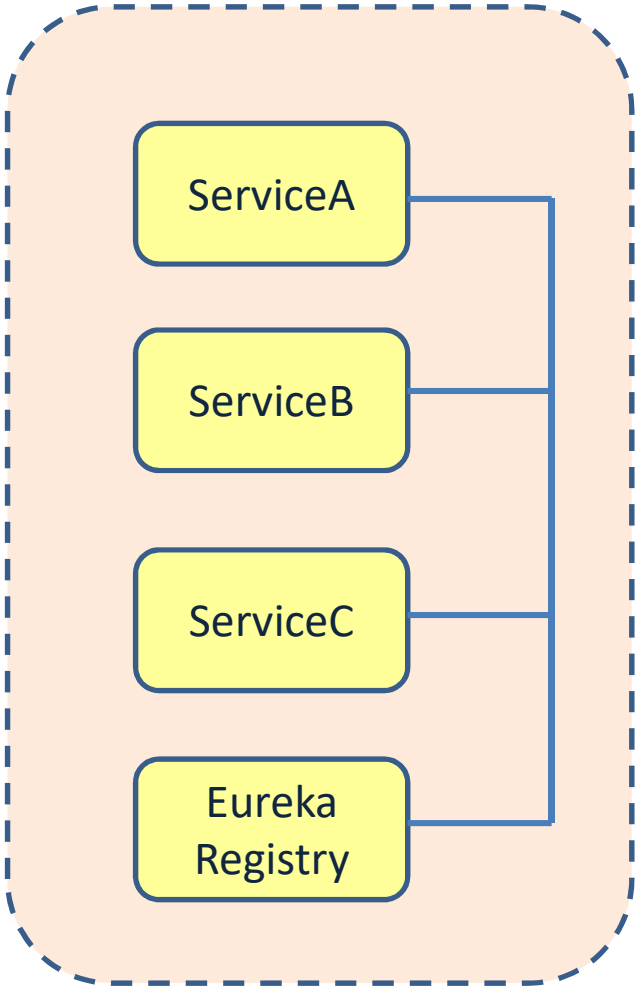
ZUUL

API GATEWAY: ZUUL

is the enter to the syste, which allow different clients like browser, app, etc to consume microservices

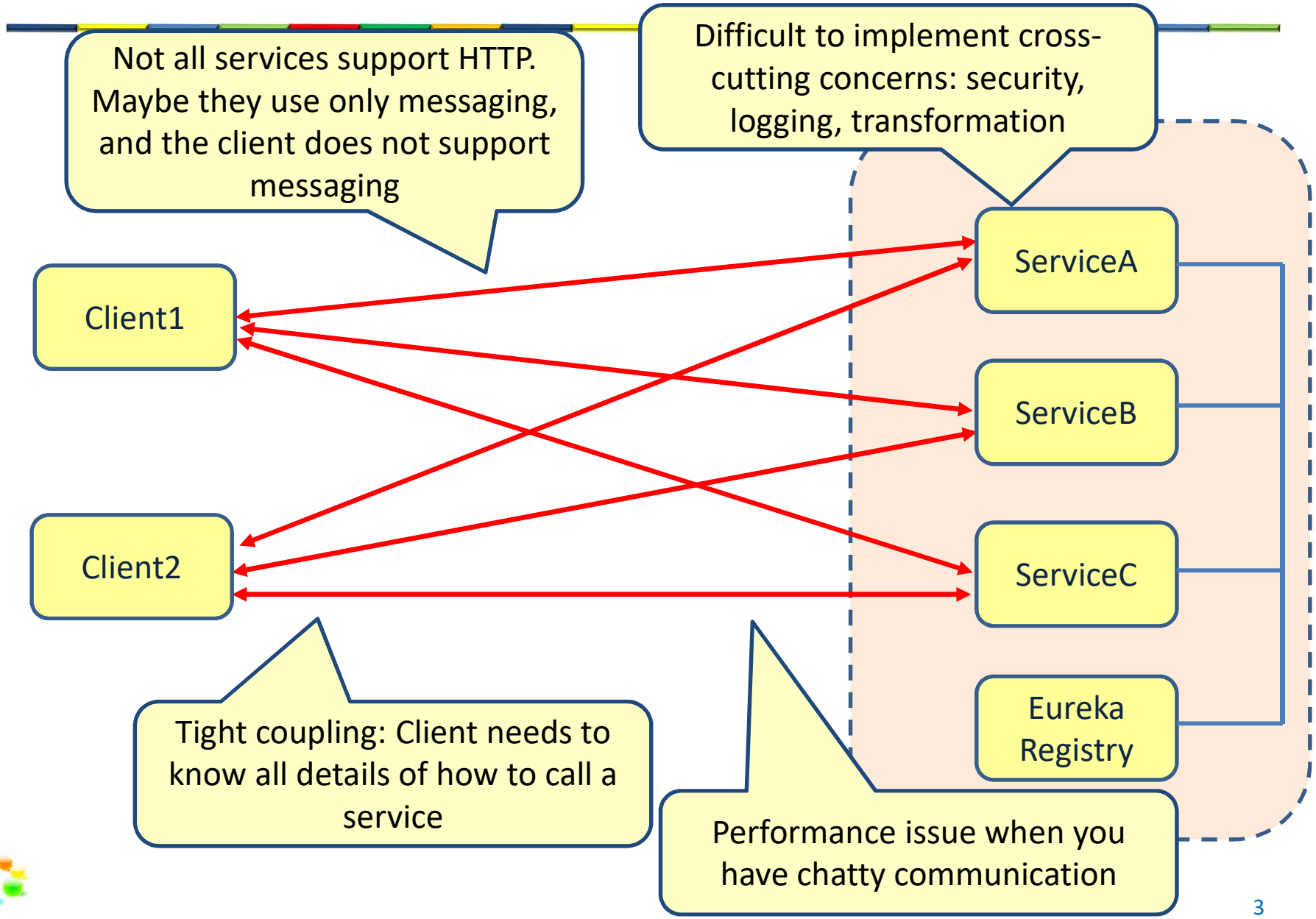
It has the tasks of routing, filterins, logging, security

Microservice architecture

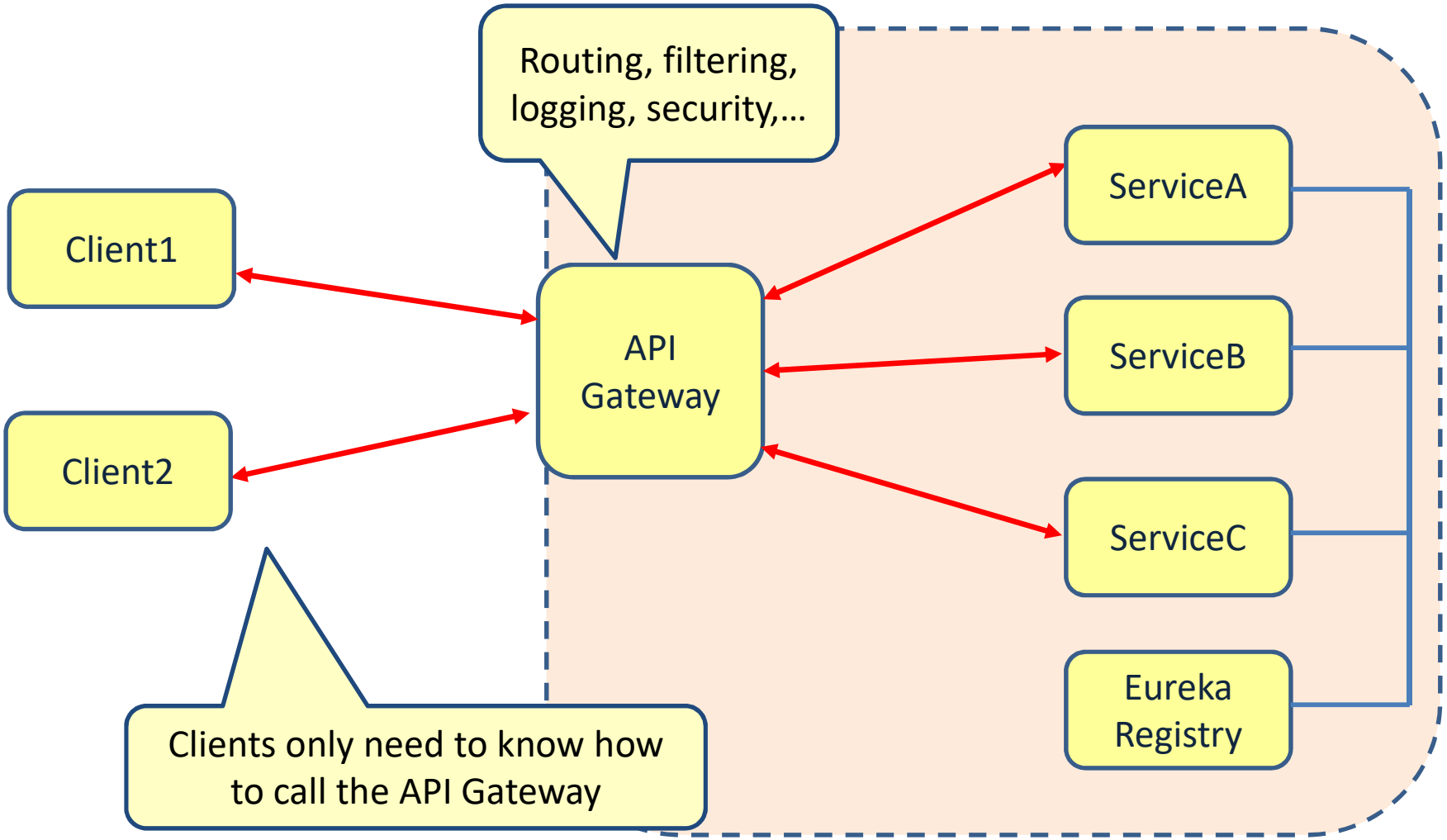


Services talk to each other using the registry

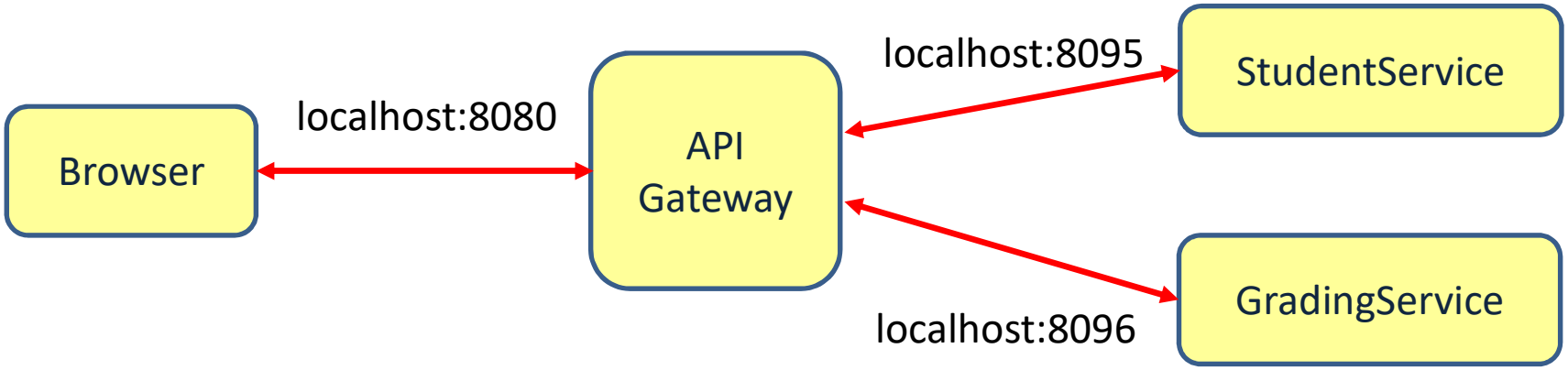
Adding clients



Api Gateway


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

Api Gateway example



Zuul dependency

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

StudentService

```java
@SpringBootApplication
@EnableDiscoveryClient
public class StudentServiceApplication {

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

application.yml

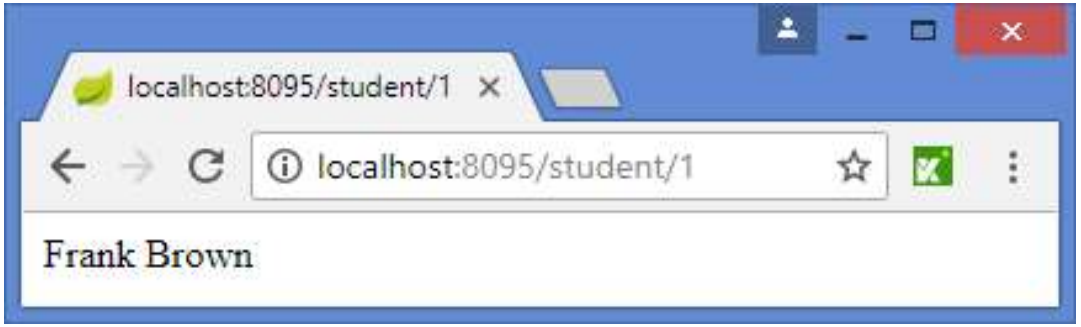
```yaml
server:
  port: 8095

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

bootstrap.yml

```yaml
spring:
  application:
    name: StudentService
```

StudentService: the controller

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



GradingService

```java
@SpringBootApplication
@EnableDiscoveryClient
public class GradingServiceApplication {

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

application.yml

```yaml
server:
  port: 8096

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

bootstrap.yml

```yaml
spring:
  application:
    name: GradingService
```

GradingService: the controller

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

API Gateway: Zuul

```
@SpringBootApplication
@EnableZuulProxy
public class ApiGatewayApplication {

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

bootstrap.yml

```
spring:
  application:
    name: ZuulService
```

API Gateway: Zuul

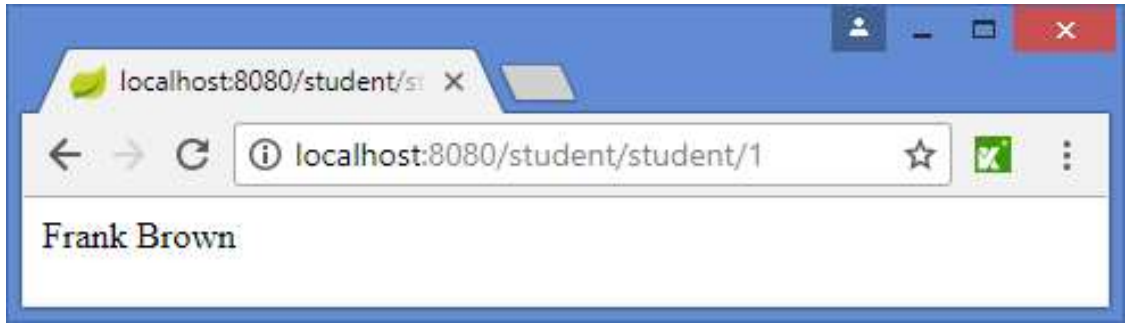
application.yml

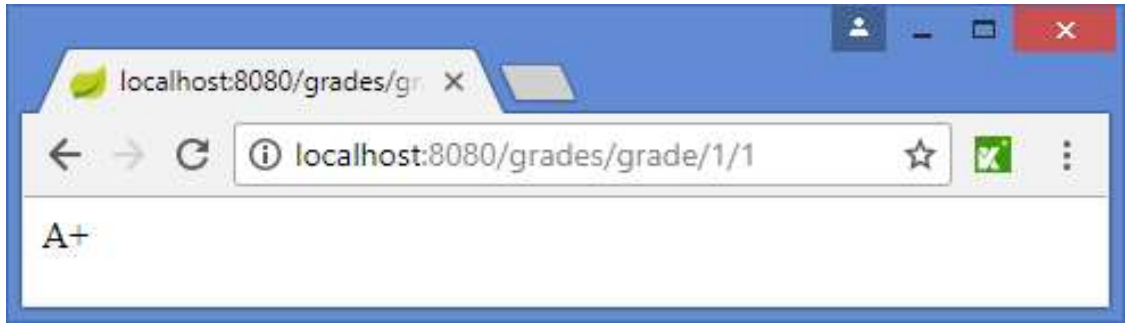
```yaml
server:
  port: 8080

zuul:
  routes:
    student:
      url: http://localhost:8095
    grades:
      url: http://localhost:8096
```

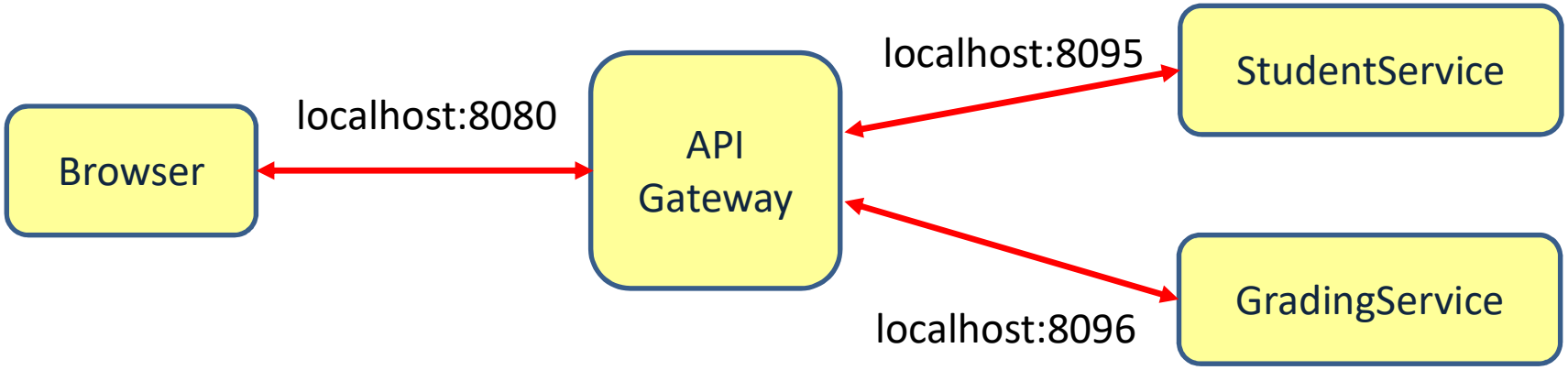
Route localhost:8080/student to localhost:8095

Route localhost:8080/grades to localhost:8096

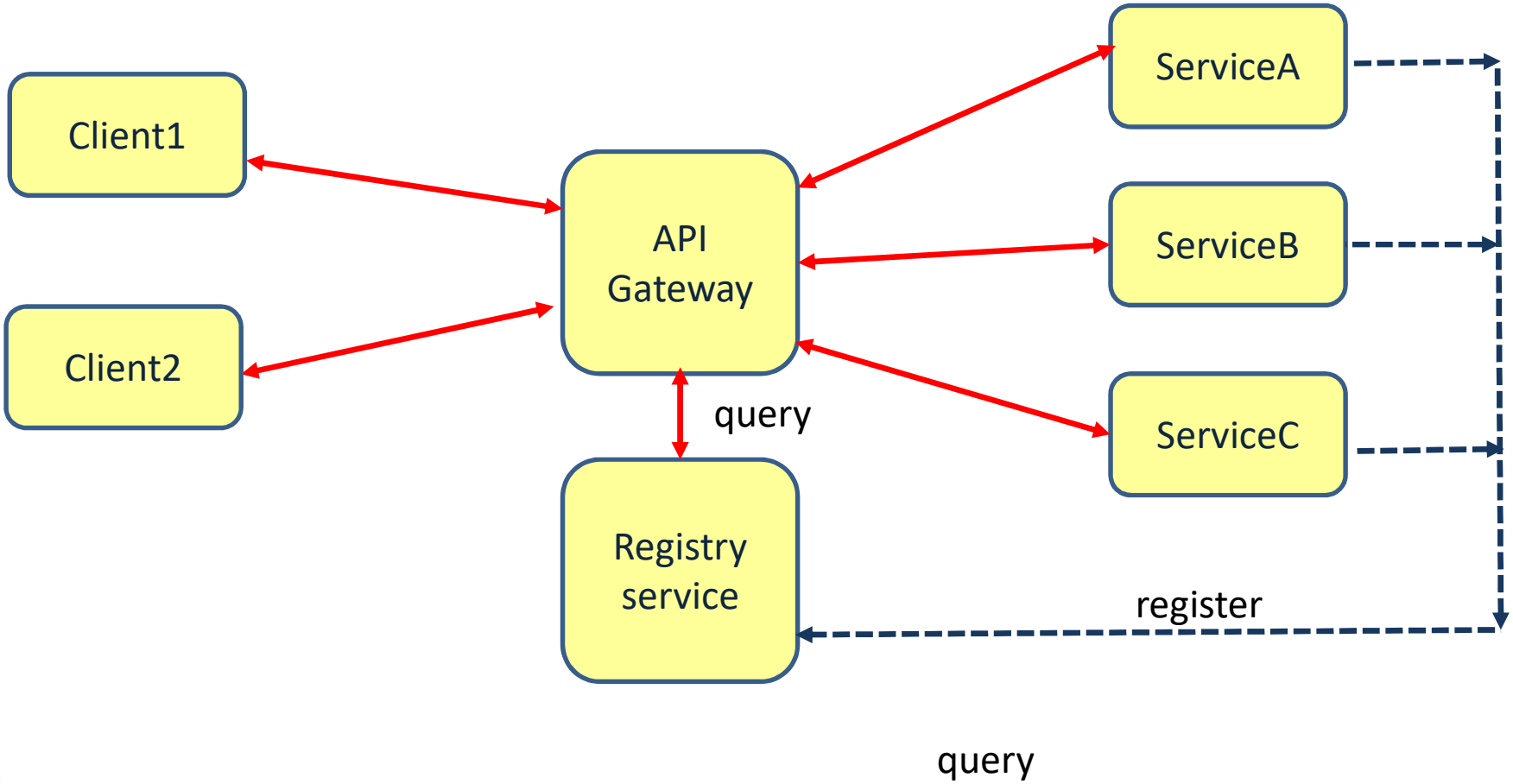
Using the API Gateway







Api Gateway and registry service



API Gateway: Zuul

```
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ApiGatewayApplication {

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

Give The API server access to Eureka

bootstrap.yml

```
spring:
  application:
    name: ZuulService
```

API Gateway: Zuul

application.yml

```yaml
server:
  port: 8080

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    registerWithEureka: true
    fetchRegistry: true

zuul:
  routes:
    student:
      serviceId: StudentService
    grades:
      serviceId: GradingService
```

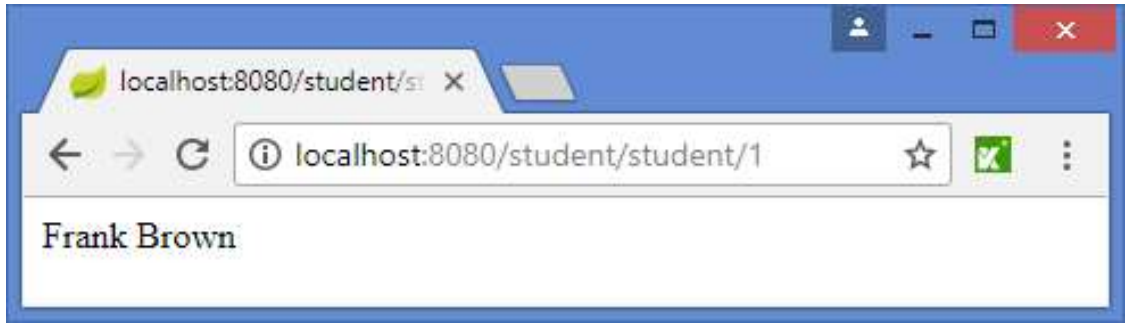
Register with Eureka

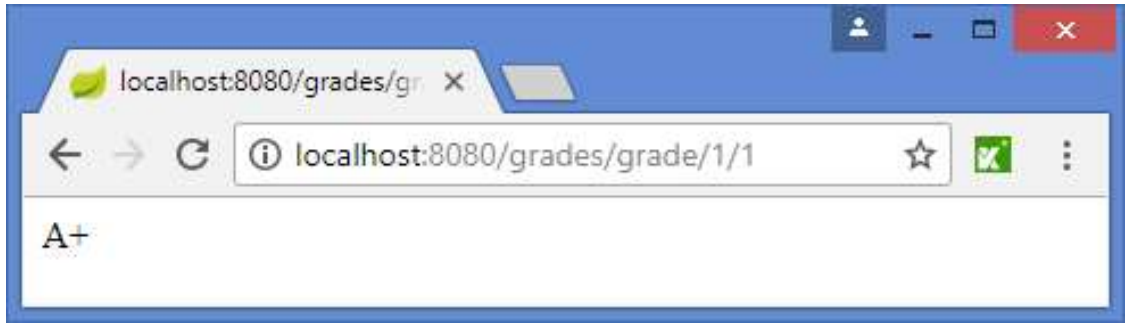
Fetch from Eureka

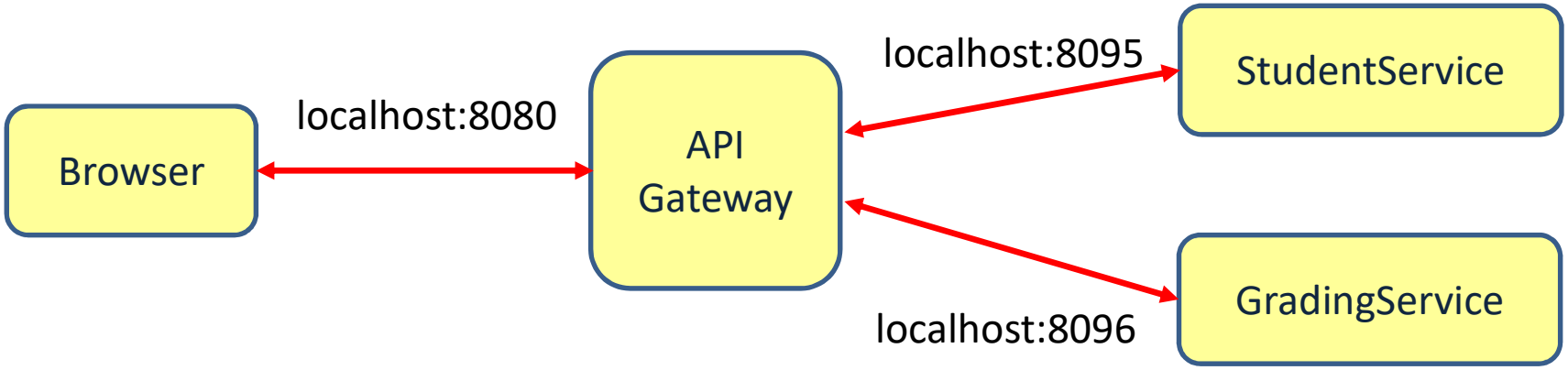
Route localhost:8080/student to the service that is registered in Eureka with the name StudentService

Route localhost:8080/grades to the service that is registered in Eureka with the name GradingService

Using the API Gateway



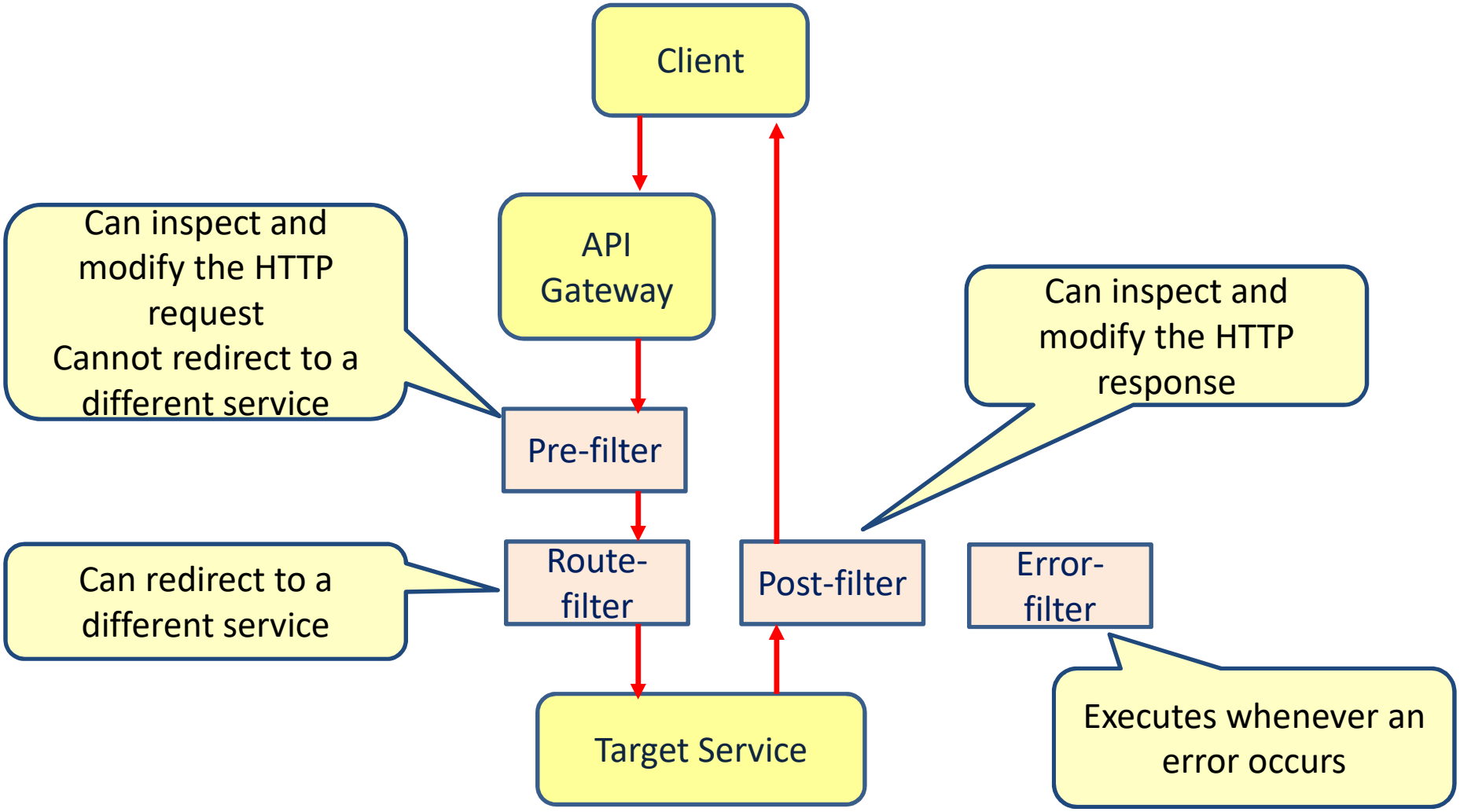




Cross cutting concerns

- Security, logging, tracking, transformations
- Implemented with filters
  - Pre-filters
  - Post-filters
  - Route-filters

API Gateway Filters



Pre-filter

```
@Component
public class SimpleFilter extends ZuulFilter {
  @Override
  public String filterType() {
    return "pre";
  }
  @Override
  public int filterOrder() {
    return 1;
  }
  @Override
  public boolean shouldFilter() {
    return true;
  }
  @Override
  public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();

    System.out.println(request.getMethod() + " request to " +
                       request.getRequestURL().toString());
    return null;
  }
}
```

Type of filter

Order of nested filters

Should the filter be active?

Functionality of the filter

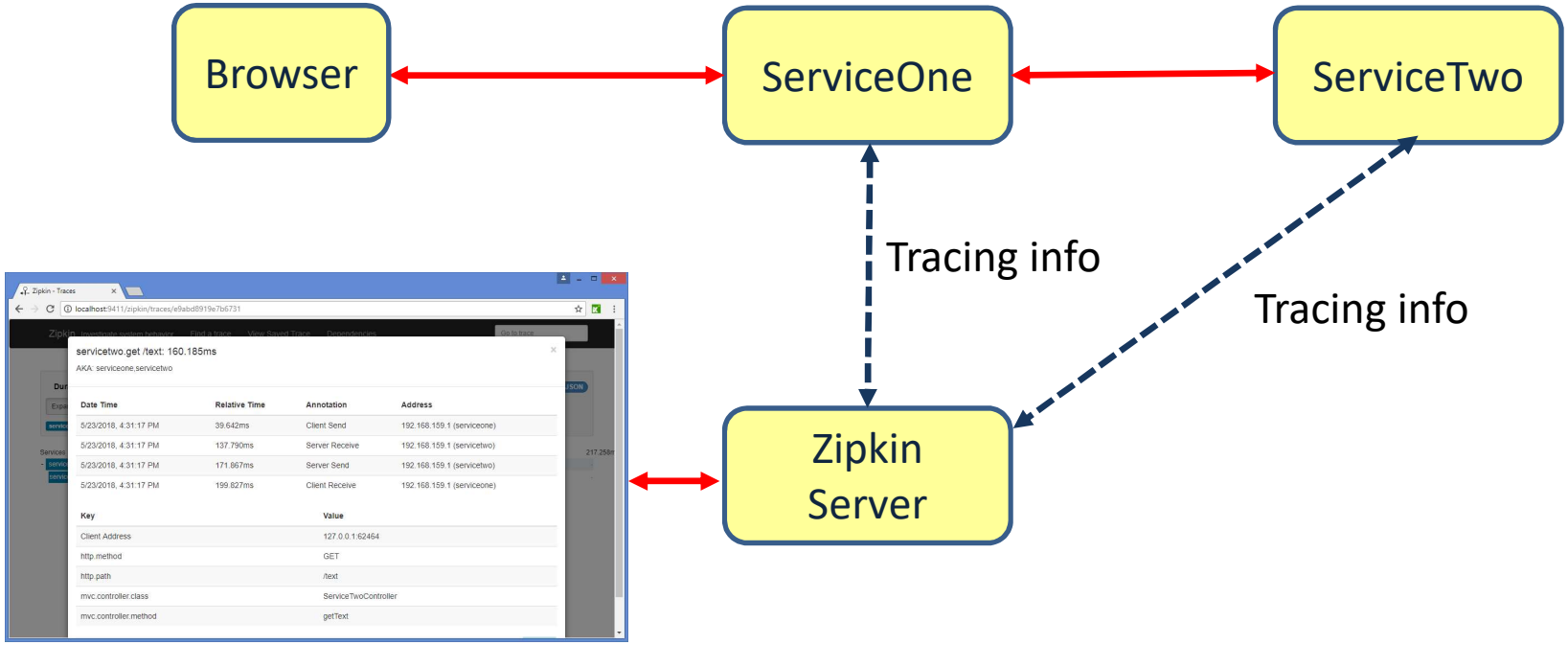
Main point

- The API gateway sits between the client applications and the microservices so that we get loose coupling between them.
- Pure Consciousness provides a unified interface to all aspects of creation, and the daily experience of Pure Consciousness makes life much more enjoyable.

DISTRIBUTED TRACING: ZIPKIN

Distributed Tracing

- One central place where one can see the end-to-end tracing of all communication between services



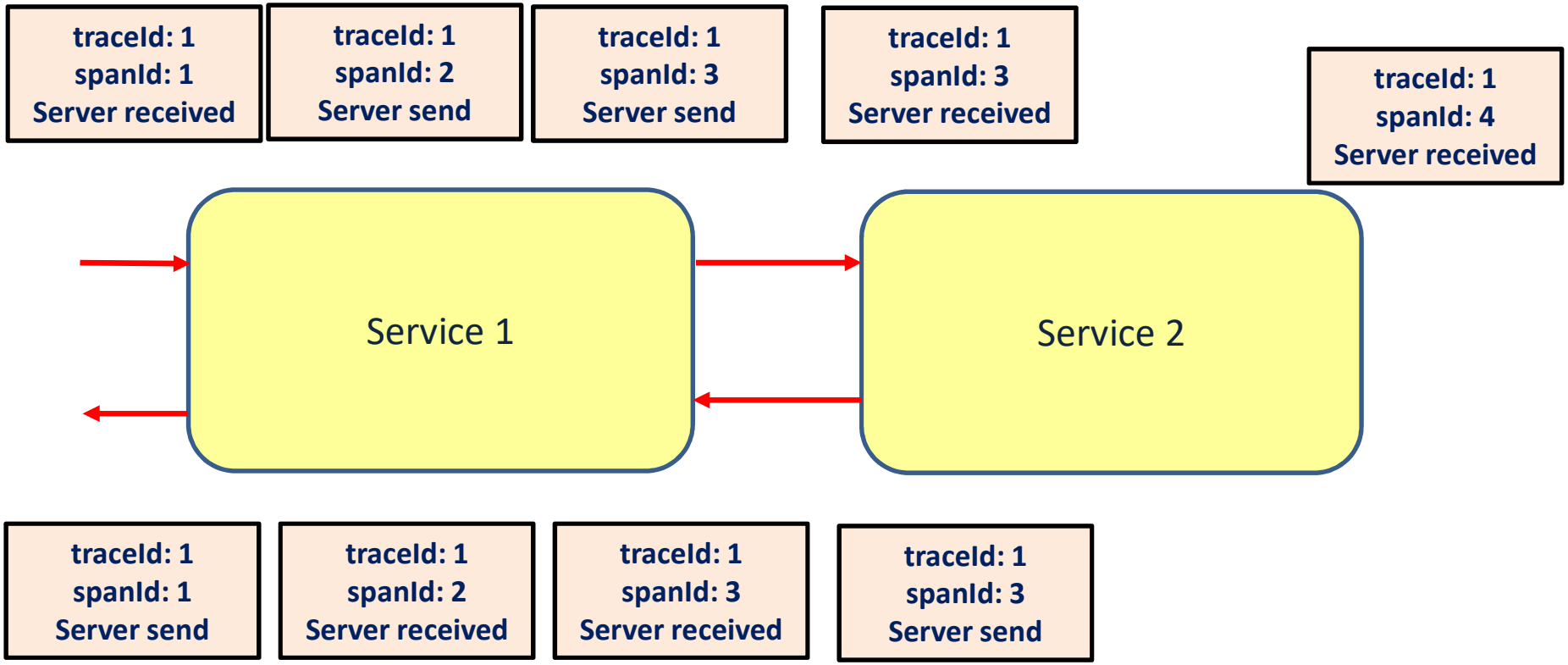
Spring cloud Sleuth

- Adds unique id's to a request so we can trace the request
  - Span id: id for an individual operation
  - Trace id: id for a set of spans

- Also embeds these unique id's to log messages

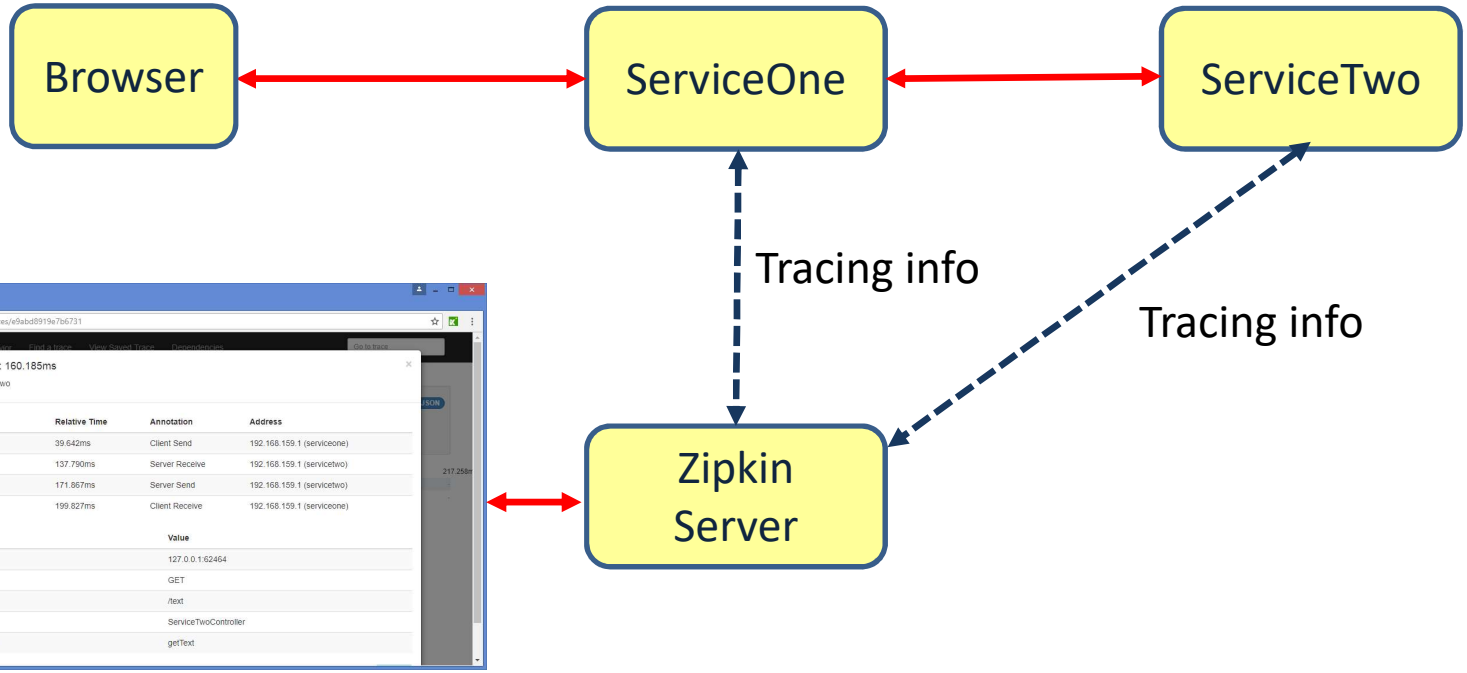
Spring cloud Sleuth

- Span: an individual operation
- Trace: a set of spans



Zipkin

- Centralized tracing server
  - Collects tracing information
- Zipkin console shows the data



Zipkin and Sleuth dependency

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

Service1

```
@SpringBootApplication
public class Service1Application {
  public static void main(String[] args) {
    SpringApplication.run(Service1Application.class, args);
  }
}
```

```
@RestController
public class ServiceOneController {

  @Autowired
  RestTemplate restTemplate;

  @RequestMapping("/text")
  public String getText() {
    String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                                                    String.class);
    return "Hello "+ service2Text;
  }

  @Bean
  RestTemplate getRestTemplate() {
    return new RestTemplate();
  }
}
```

Service1

application.yml

```yaml
server:
  port: 9090

spring:
  zipkin:
    base-url: http://localhost:9411/

sleuth:
    sampler:
      probability: 1 #100% (default = 10%)
```

bootstrap.yml

```yaml
spring:
  application:
    name: ServiceOne
```

Service1

pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

Service2

```
@SpringBootApplication
public class Service2Application {
  public static void main(String[] args) {
    SpringApplication.run(Service2Application.class, args);
  }
}
```

```
@RestController
public class ServiceTwoController {

  @RequestMapping("/text")
  public String getText() {
    return "World";
  }
}
```

Service1

application.yml

```yaml
server:
  port: 9091

spring:
  zipkin:
    base-url: http://localhost:9411/

  sleuth:
    sampler:
      probability: 1 #100% (default = 10%)
```

bootstrap.yml

```yaml
spring:
  application:
    name: ServiceTwo
```

Zipkin console

Zipkin - Traces

localhost:9411/zipkin/traces/e9abd8919e7b6731

Zipkin Investigate system behavior Find a trace View Saved Trace Dependencies Go to trace

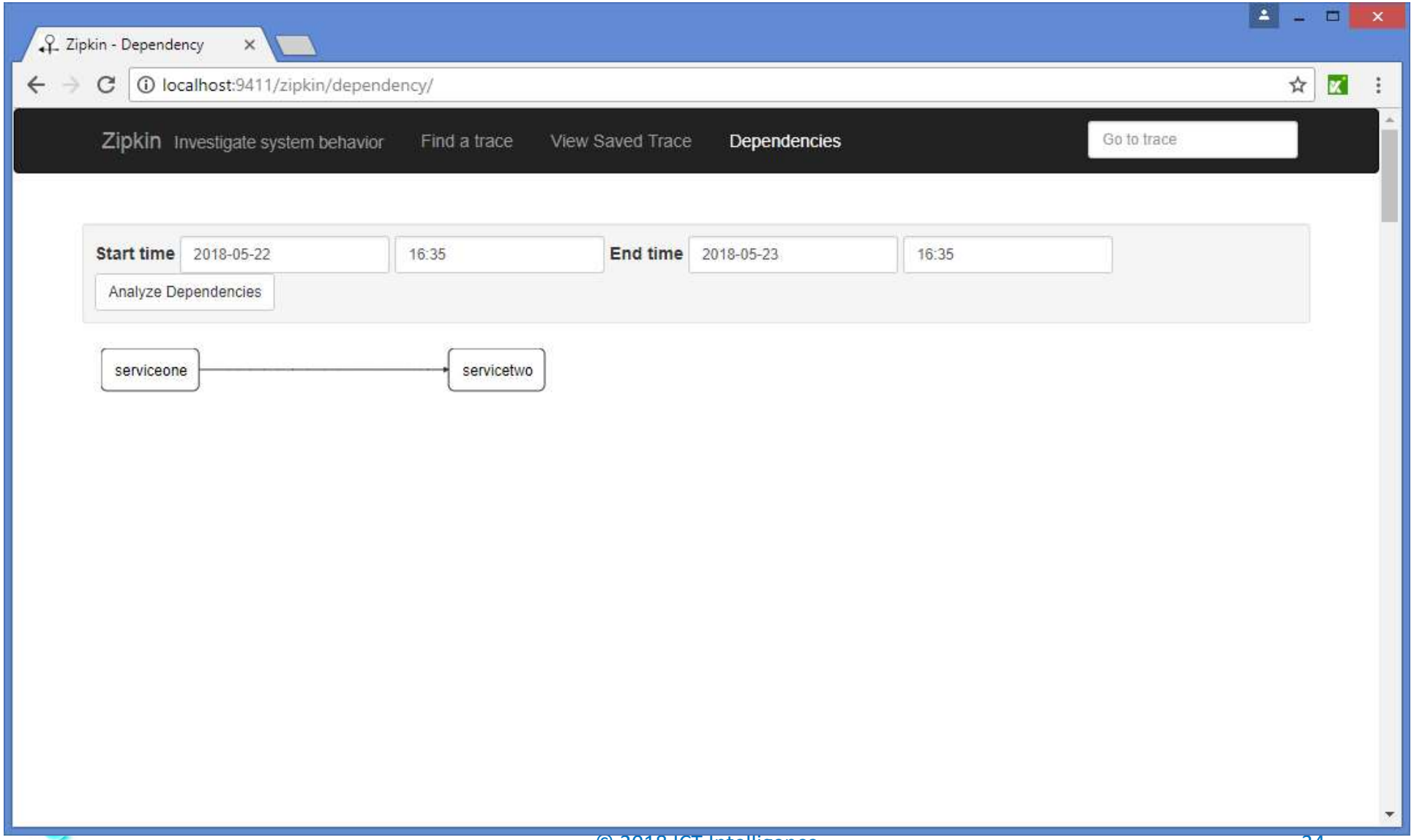
servicetwo.get /text: 160.185ms

AKA: serviceone,servicetwo

Date Time	Relative Time	Annotation	Address
5/23/2018, 4:31:17 PM	39.642ms	Client Send	192.168.159.1 (serviceone)
5/23/2018, 4:31:17 PM	137.790ms	Server Receive	192.168.159.1 (servicetwo)
5/23/2018, 4:31:17 PM	171.867ms	Server Send	192.168.159.1 (servicetwo)
5/23/2018, 4:31:17 PM	199.827ms	Client Receive	192.168.159.1 (serviceone)

Key	Value
Client Address	127.0.0.1:62464
http.method	GET
http.path	/text
mvc.controller.class	ServiceTwoController
mvc.controller.method	getText

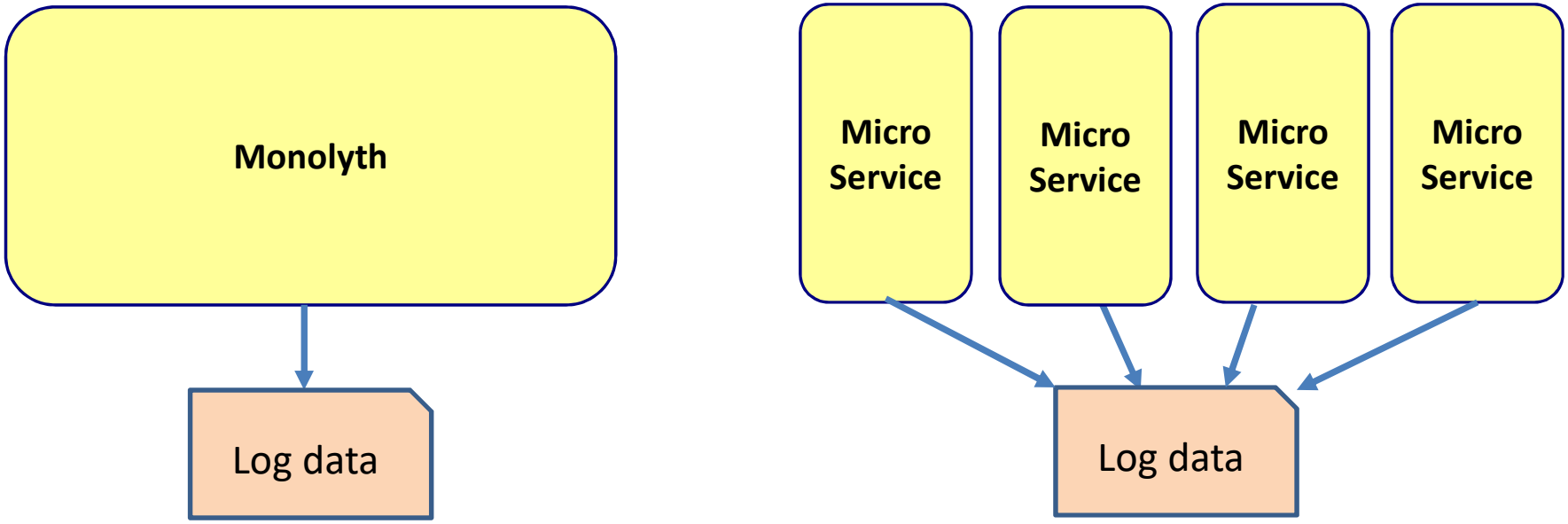
Zipkin console



DISTRIBUTED LOGGING: ELK

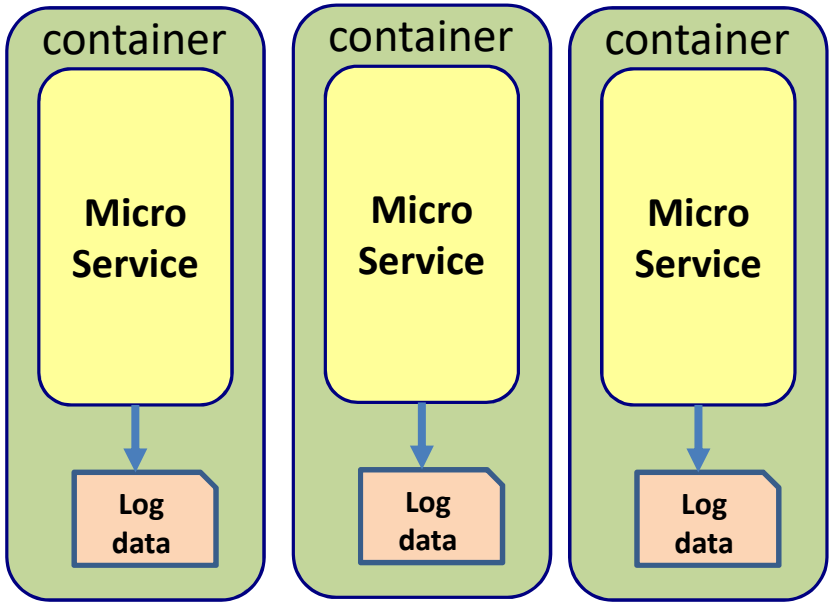
We need collect all log data from all services to know what has happened. It is centralized

Logging



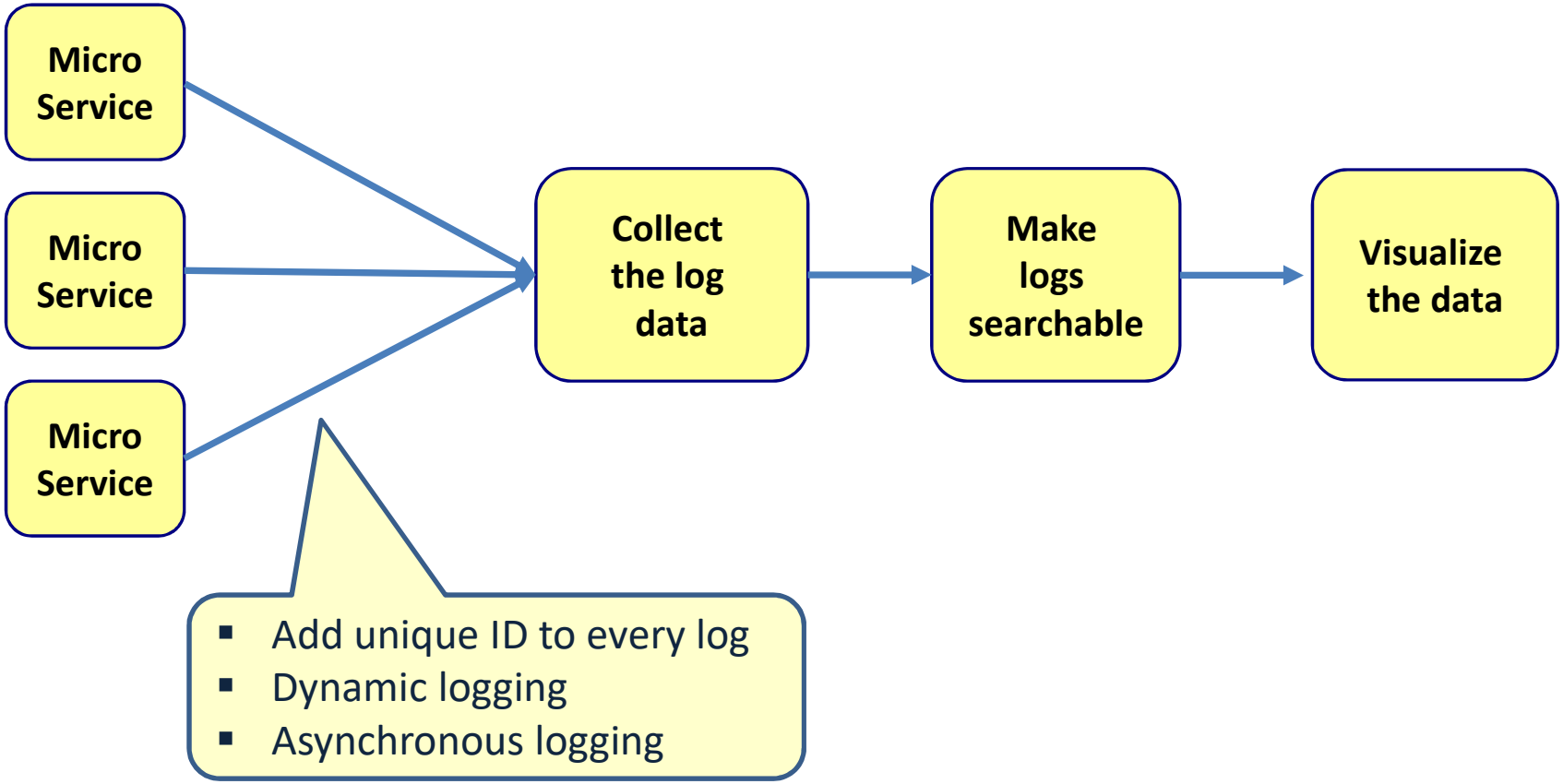
- We need to collect all log data from all services to know what has happened

The need for centralized logging



- Local logging does not work
  - Containers come and go
  - Containers have no fixed address

Microservice logging architecture



ELK stack

Micro Service

Micro Service

Micro Service

Collect the log data

Make logs searchable

Visualize the data

logstash

elasticsearch

kibana

Collect logs
Transform
Time normalization

Schema less search DB
Highly scalable
Fast searching

Main point

- In a microservice architecture, we need centralized tracing and logging to monitor our systems
- The Unified Field is the abstract field that unites all diversity in creation.

RESILIENCE

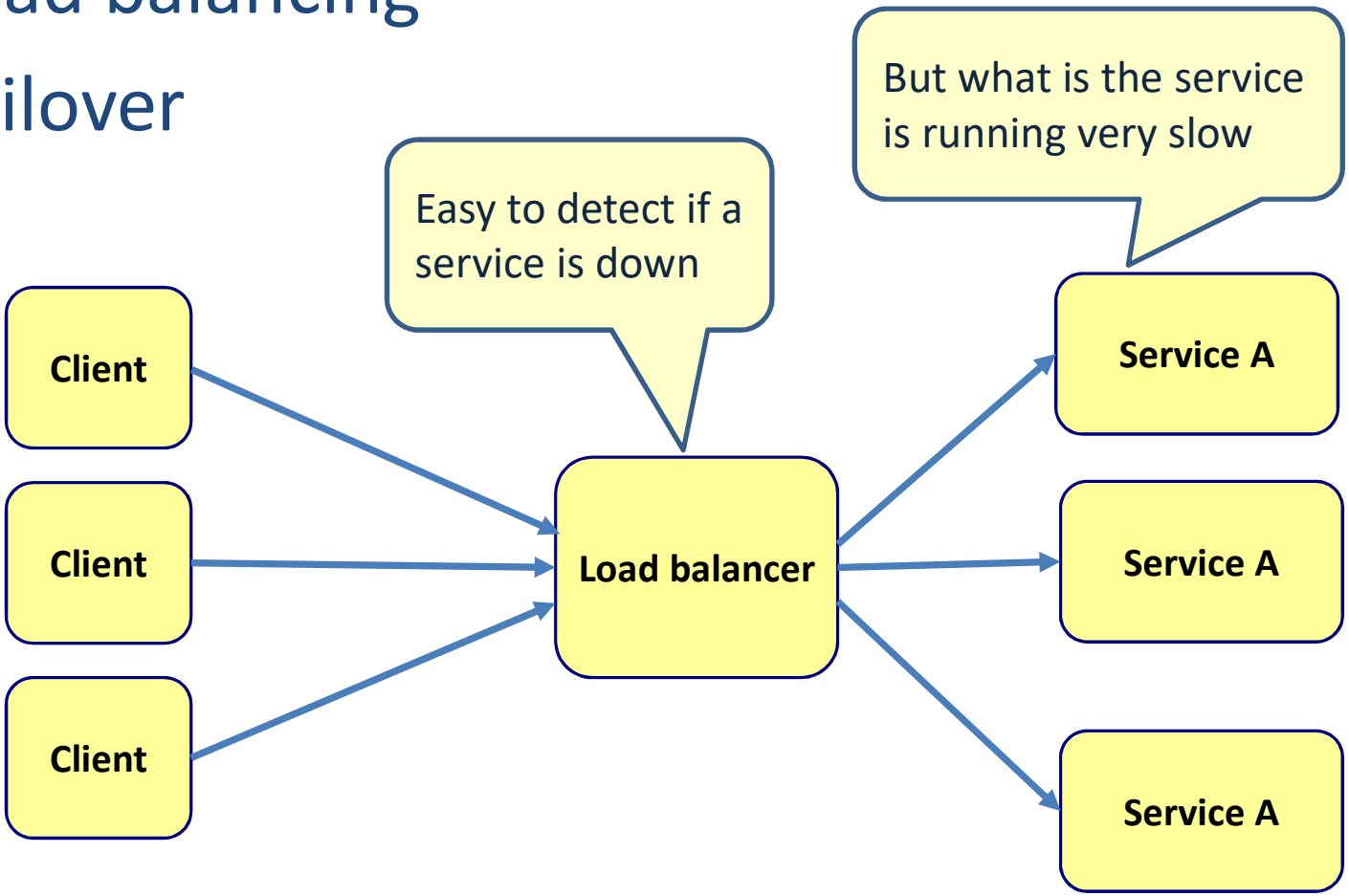
The ability to recover from failures

Fallacies of distributed computing

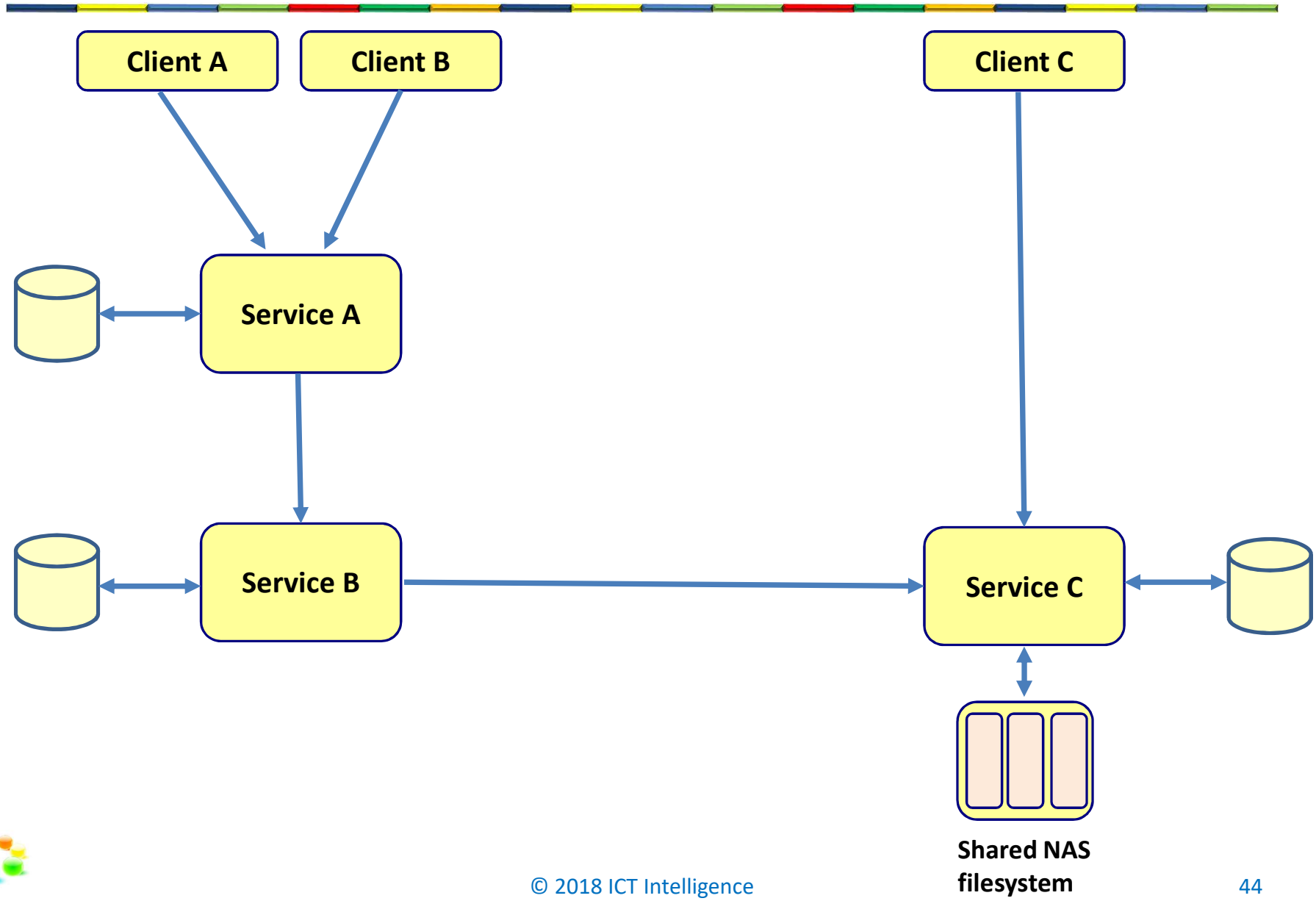
- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn’t change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Clustering

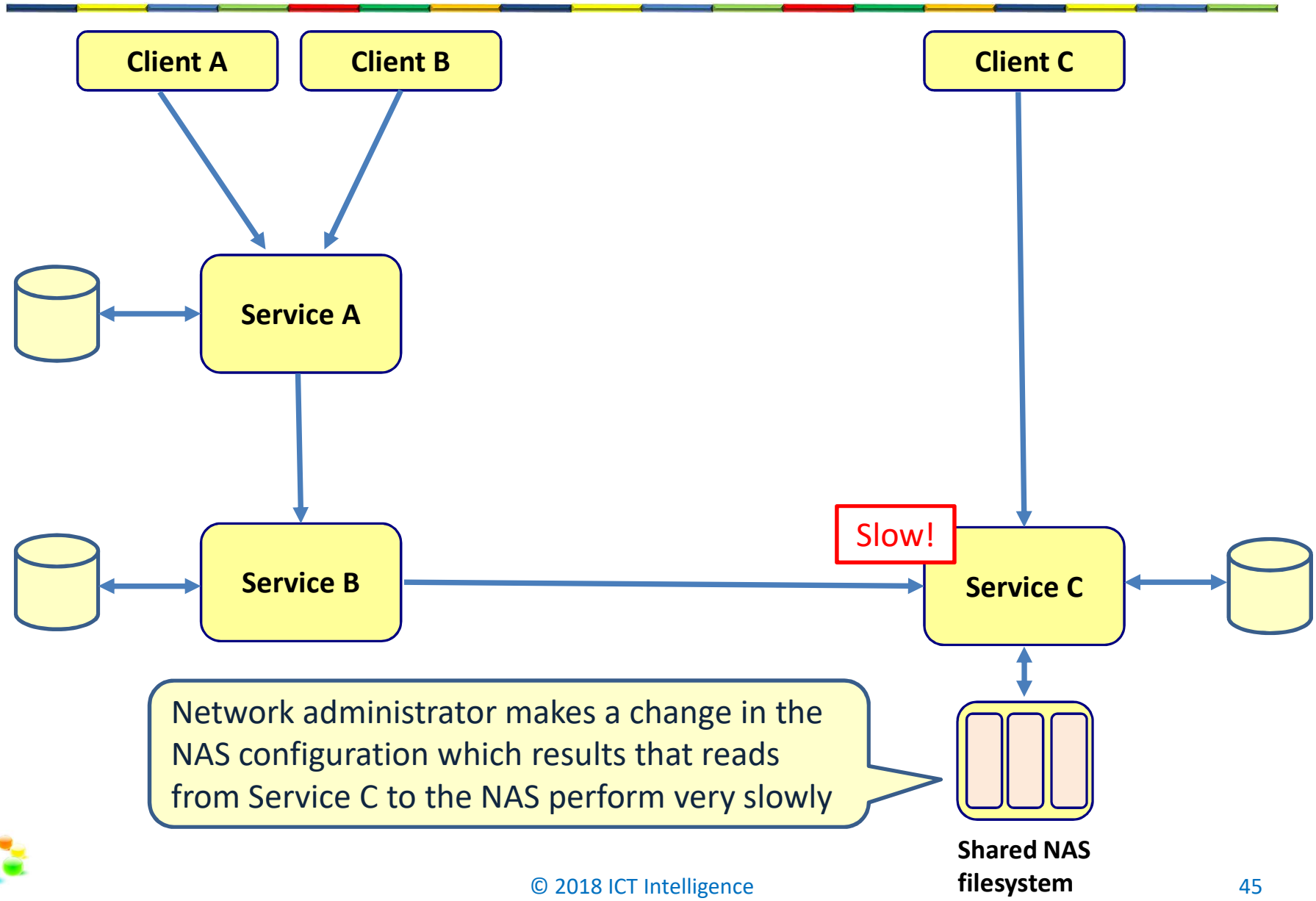
- Load balancing
- Failover



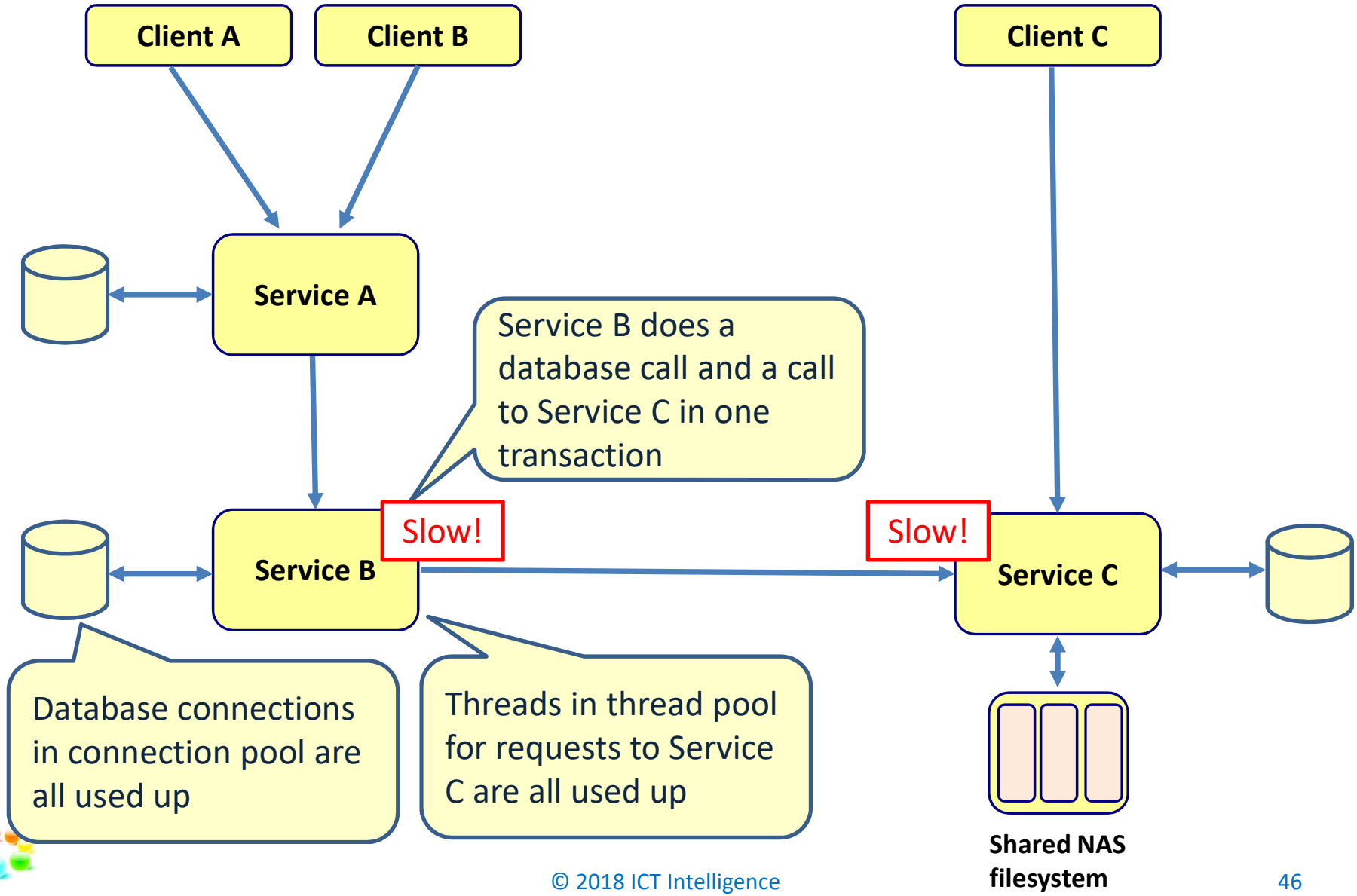
Example


Client A
Client B
Client C
Service A
Service B
Service C
Shared NAS filesystem

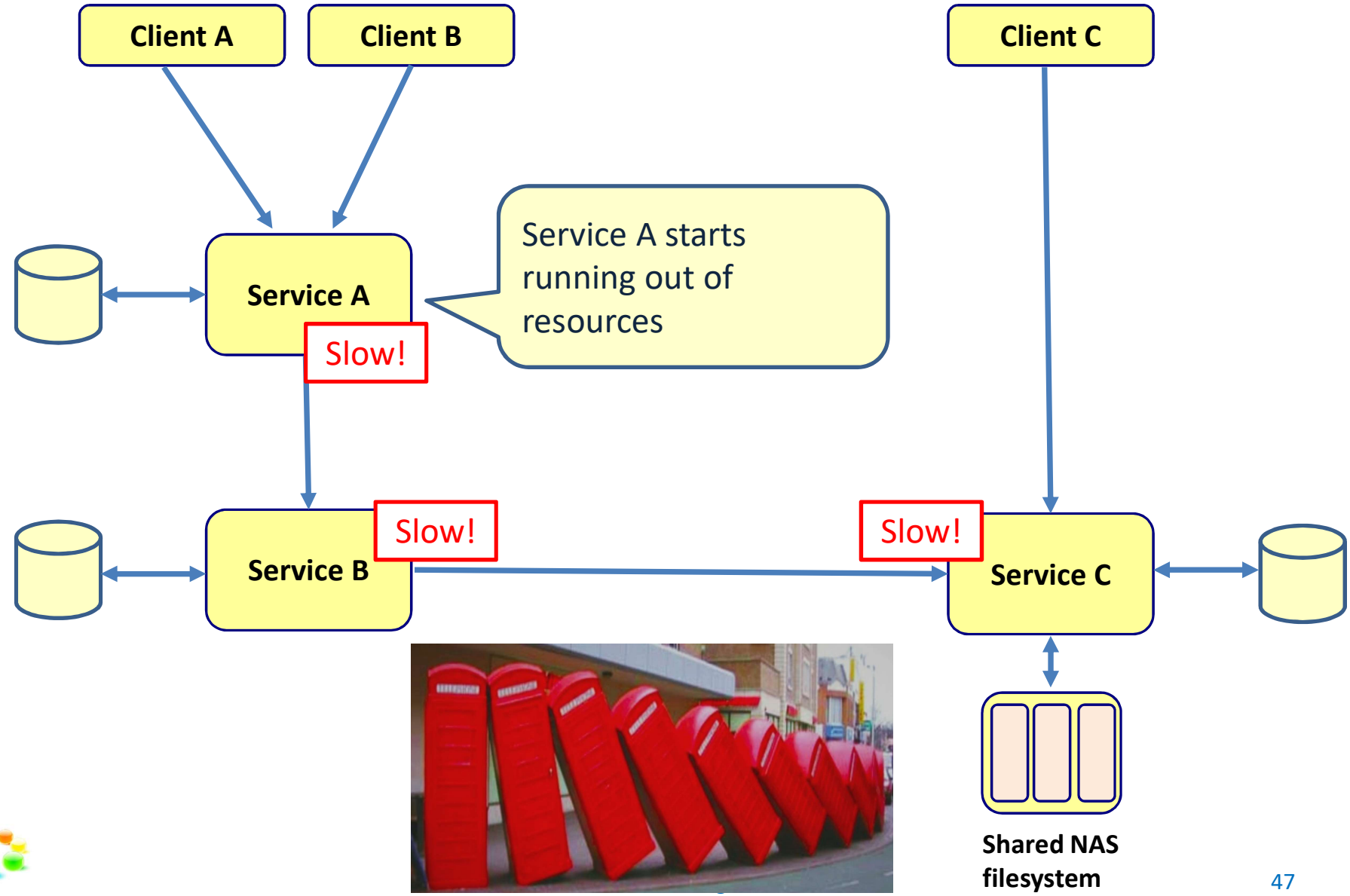
Example


Client A
Client B
Client C
Service A
Service B
Slow!
Service C
Network administrator makes a change in the NAS configuration which results that reads from Service C to the NAS perform very slowly
Shared NAS filesystem

Example


Client A
Client B
Client C
Service A
Service B does a database call and a call to Service C in one transaction
Slow!
Slow!
Service B
Service C
Database connections in connection pool are all used up
Threads in thread pool for requests to Service C are all used up
Shared NAS filesystem

Example


Client A
Client B
Client C
Service A
Slow!
Service A starts running out of resources
Service B
Slow!
Slow!
Service C
Shared NAS filesystem

RESILLIENCE: HYSTRIX

tool to mange resilence

Hystrix dependency

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

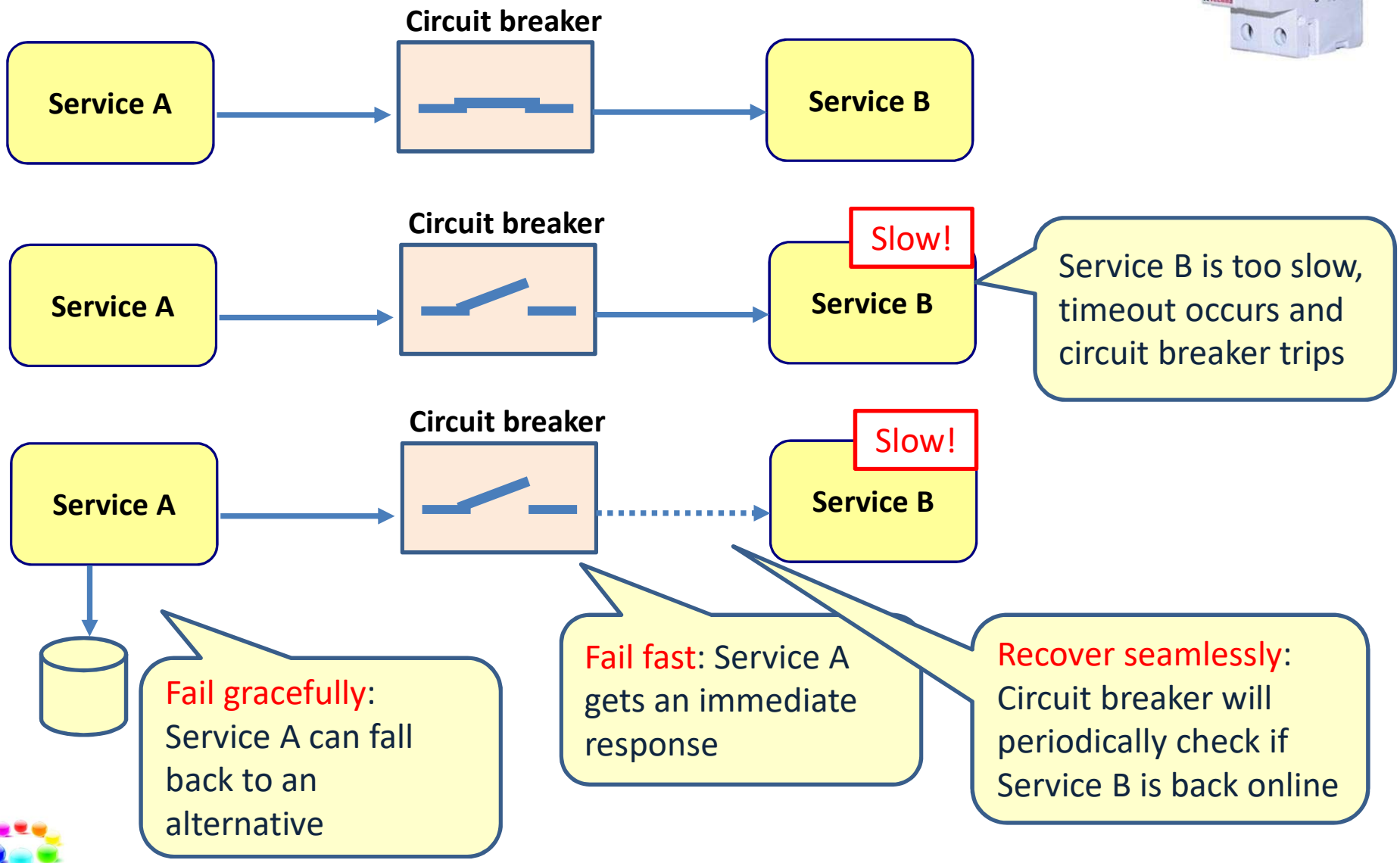
Resilience patterns

- Timeouts
- Circuit breaker
- Bulkheads

Timeouts

- Put timeouts on all out-of-process calls.
  - Other services
  - Database
  - File system
- Log when timeouts occur
  1. Pick a default timeout
  2. Monitor
  3. Adjust

Circuit breaker



Enable Hystrix


```
@SpringBootApplication
@EnableCircuitBreaker
public class ServiceOneApplication {

  public static void main(String[] args) {
    SpringApplication.run(Service2Application.class, args);
  }
}
```

Enable Histrix

```
@SpringBootApplication
@EnableCircuitBreaker
public class ServiceTwoApplication {

  public static void main(String[] args) {
    SpringApplication.run(Service2Application.class, args);
  }
}
```

Enable Histrix

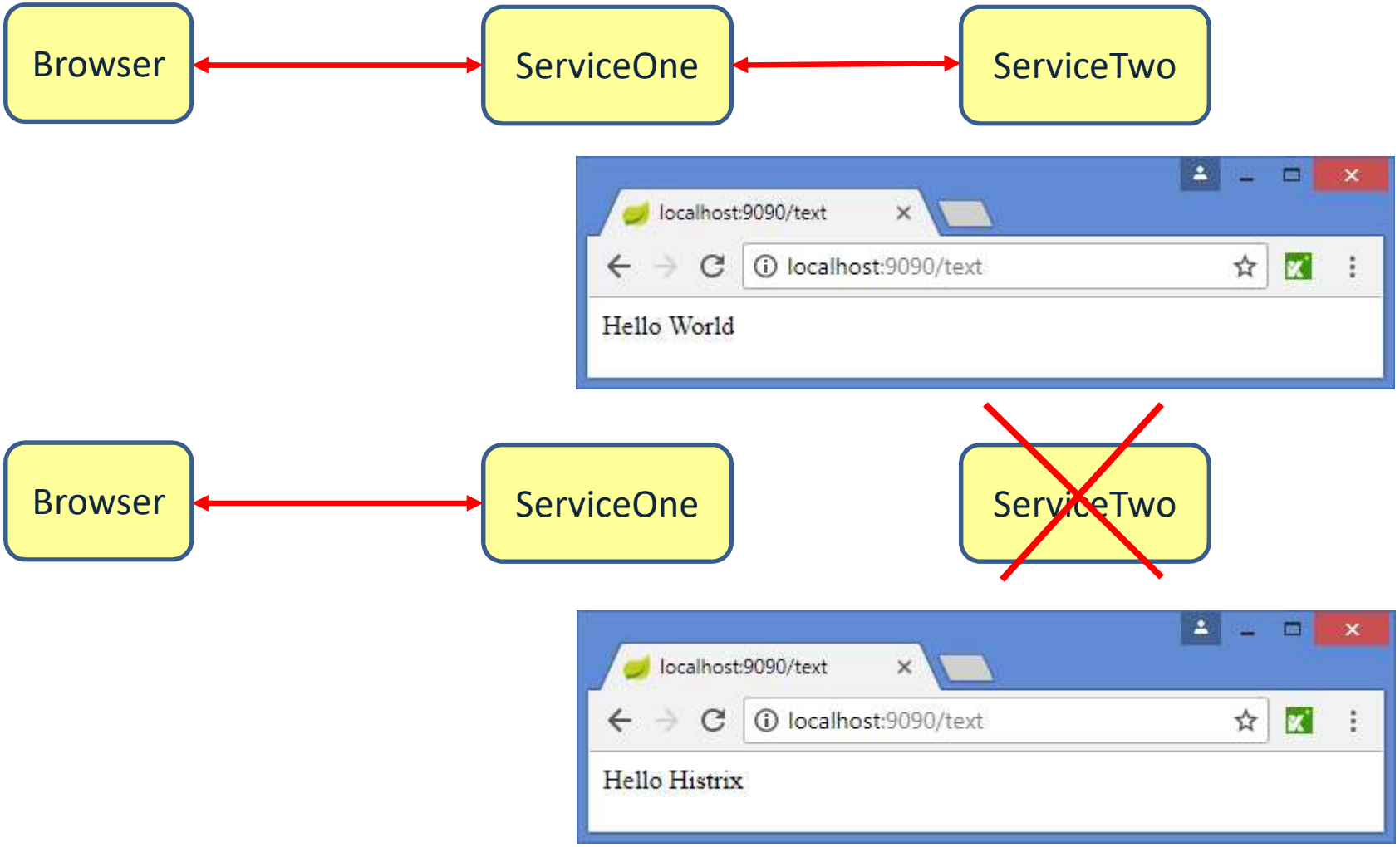
Using the circuit breaker

```
public class ServiceOneController {

  @Autowired
  RestTemplate restTemplate;

  @RequestMapping("/text")
  @HystrixCommand(fallbackMethod = "getTextFallback")
  public String getText() {
    String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                                                    String.class);
    return "Hello "+ service2Text;
  }

  public String getTextFallback() {
   return "Hello Histrix";
  }

  @Bean
  RestTemplate getRestTemplate() {
    return new RestTemplate();
  }
}
```

If this method throws an exception or takes longer than 2 seconds, call the fallback method

Fallback method

Using Histrix



Setting the timeout

```
public class ServiceOneController {

  @Autowired
  RestTemplate restTemplate;

  @RequestMapping("/text")
  @HystrixCommand(fallbackMethod = "getTextFallback", commandProperties=
  {@HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",
                    value="4000")})
  public String getText() {
    String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                                                    String.class);
    return "Hello "+ service2Text;
  }

  public String getTextFallback() {
   return "Hello Histrix";
  }

  @Bean
  RestTemplate getRestTemplate() {
    return new RestTemplate();
  }
}
```

Set timeout to 4 seconds

Setting the timeout

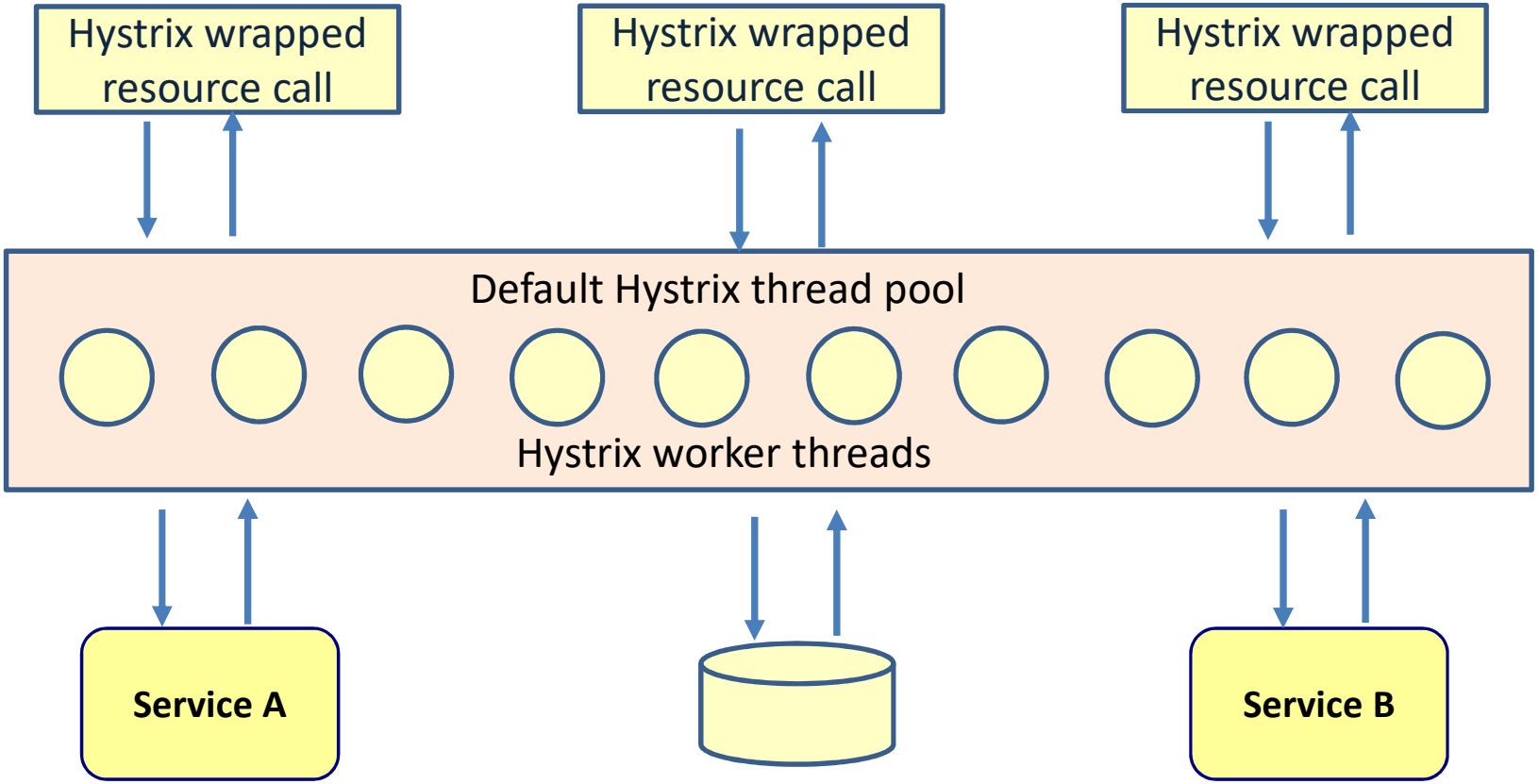
```
@RestController
public class ServiceTwoController {

  @RequestMapping("/text")
  public String getText() throws InterruptedException {
    Thread.sleep(5000);
    return "World";
  }
}
```

Sleep of 5 seconds

Bulkhead

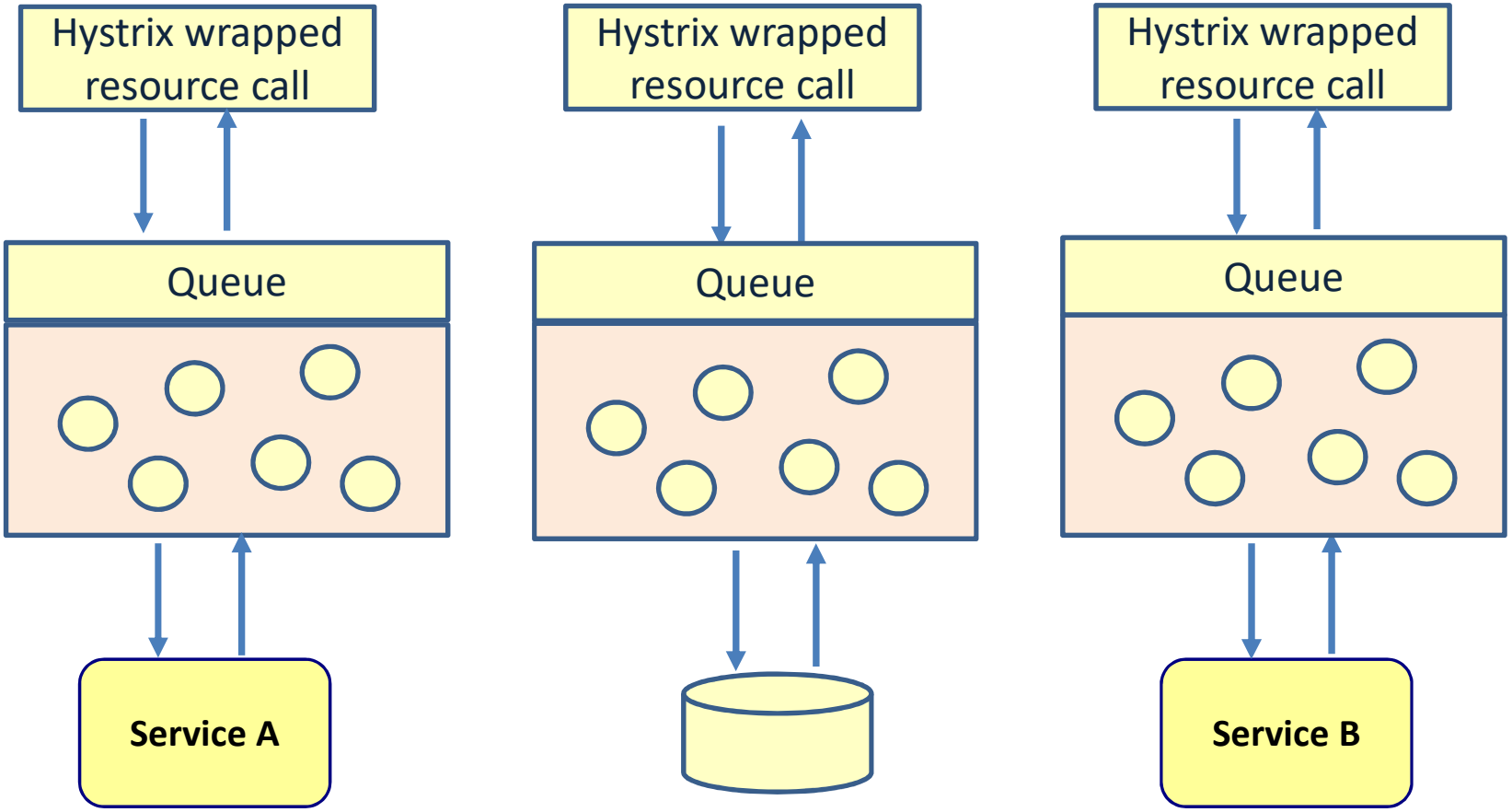
Hystrix thread pool

- Hystrix uses a common thread pool for all remote calls



Hystrix bulkheads

- Hystrix uses a common thread pool for all remote calls



Hystrix bulhead

```
@RequestMapping("/text")
@HystrixCommand(fallbackMethod = "getTextFallback",
  threadPoolKey = "Service2ThreadPool",
  threadPoolProperties = {
    @HystrixProperty(name = "coreSize", value = "30"),
    @HystrixProperty(name = "maxQueueSize", value = "10")
  })
public String getText() {
  String service2Text = restTemplate.getForObject("http://localhost:9091/text",
                        String.class);
  return "Hello "+ service2Text;
}

public String getTextFallback() {
  return "Hello Histrix";
}
```

Name of the thread pool

Maximum number of threads

Maximum queue size

Main point

- To make a microservice architecture resilient, we need to think of fallback scenarios for distributed calls
- Daily contact with Pure Consciousness is the fallback scenario for many challenges in life. Bring light into the darkness.

Connecting the parts of knowledge with the wholeness of knowledge

1. The API gateway is “a layer of indirection” between clients and microservices.
2. In a distributed microservice architecture you need to program defensively, because things will go wrong.

---

3. **Transcendental consciousness** is the source of all activity.
4. **Wholeness moving within itself:** In Unity Consciousness, one experiences that one self (rishi), and all other objects (chhandas) and the operations between oneself and all other objects (devata) are expressions of one’s own Self.