마이크로서비스 자료조사

목차

1. Service discovery

Service discovery

마이크로 서비스 아키텍처와 같은 분산 환경은 서비스 간의 원격 호출로 구성이 된다. 원격 서비스 호출은 IP 주소와 포트를 이용하는 방식이 된다.

클라우드 환경이 되면서 서비스가 오토 스케일링등에 의해서 동적으로 생성되거나 컨테이너 기반의 배포로 인해서, 서비스의 IP가 동적으로 변경되는 일이 잦아졌다.

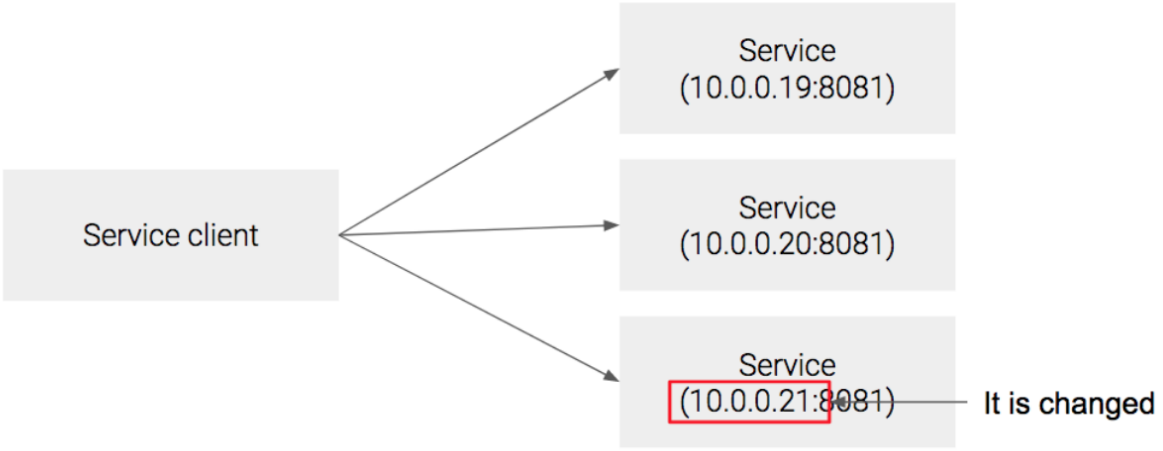

그림 1 IP 변경 예시

그래서 서비스 클라이언트가 서비스를 호출 시 서비스의 위치 (IP와 포트)를 알아낼 수 있는 기능이 필요한데, 이것이 Service discovery이다.

Client side discovery vs Server side discovery

이러한 Service discovery 기능을 구현하는 방법으로는 크게 client discovery 방식과 server side discovery 방식이 있다. 앞에서 설명한 service client가 service registry에서 서비스의 위치를 찾아서 호출하는 방식을 client side discovery 라고 한다.

다른 접근 방법으로는 호출이 되는 서비스 앞에 일종의 proxy 서버 (로드밸런서)를 넣는 방식인데, 서비스 클라이언트는 이 로드밸런서를 호출하면 로드밸런서가 Service registry로 부터 등록된 서비스의 위치를 리턴하고, 이를 기반으로 라우팅을 하는 방식이다.

가장 흔한 예제로는 클라우드에서 사용하는 로드밸런서로 AWS의 ELB나 구글 클라우드의 로드밸런서가 대표적인 Server side discovery 방식에 해당한다.

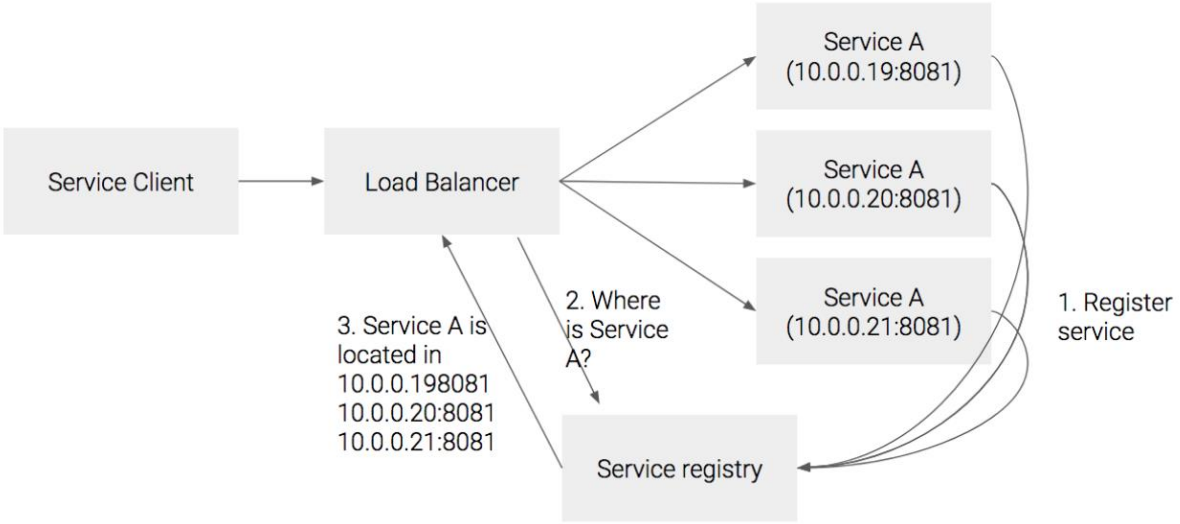

그림 2 로드 밸런서를 통한 Service discovery

Service registry

구현 방식으로는 가장 쉬운 방법으로는 DNS 레코드에 하나의 호스트명에 여러개의 IP를 등록하는 방식으로 구현이 가능하다. 그러나 DNS는 레코드 삭제시 업데이트 되는 시간등이 소요되기 때문에, 그다지 적절한 방법은 아니기 때문에, 솔루션을 사용하는 방법이 있는데, ZooKeeper나 etcd 와 같은 서비스를 이용할 수 있고 또는 Service discovery에 전문화된 솔루션으로는 Netflix의 Eureka나 Hashcorp의 Consul과 같은 서비스가 있다. Api gateway OSS에서도 이 기능들을 제공하고 있다.

2. Circuit Breaker

Circuit Breaker의 필요성

마이크로서비스 아키텍처 패턴은 시스템을 여러개의 서비스 컴포넌트로 나눠서 서비스 컴포넌트 간에 호출하는 개념을 가지고 있다. 이 아키텍처의 단점 중 하나는 하나의 컴포넌트가 느려지거나 장애가 나면 그 컴포넌트를 호출하는 종속된 컴포넌트까지 장애가 전파되는 특성을 가지고 있다

Circuit Breaker의 패턴

이런 문제를 해결하는 디자인 패턴이 Circuit breaker 라는 패턴이 있다.

기본적인 원리는 다음과 같다. 서비스 호출 중간 즉 위의 예제에서는 Service A와 Service B에 Circuit Breaker를 설치한다. Service B로의 모든 호출은 이 Circuit Breaker를 통하게 되고 Service B가 정상적인 상황에서는 트래픽을 문제없이 bypass 한다.

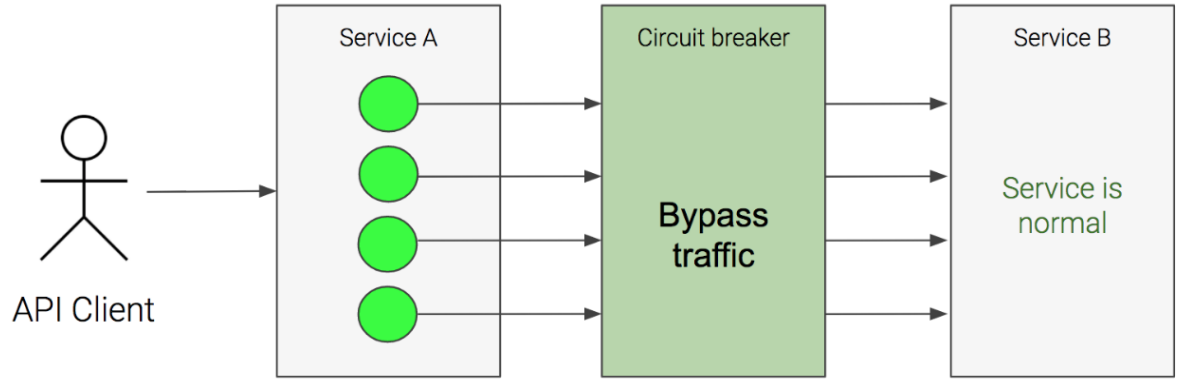

그림 3 서비스에 문제가 없을 경우

만약에 Service B가 문제가 생겼음을 Circuit breaker가 감지한 경우에는 Service B로의 호출을 강제적으로 끊어서 Service A에서 쓰레드들이 더 이상 요청을 기다리지 않도록 해서 장애가 전파하는 것을 방지한다. 강제적으로 호출을 끊으면 에러 메세지가 Service A에서 발생하기 때문에 장애 전파는 막을 수 있지만, Service A에서 이에 대한 장애 처리 로직이 별도로 필요하다.

이를 조금 더 발전 시킨것이 Fall-back 메시징인데, Circuit breaker에서 Service B가 정상적인 응답을 할 수 없을 때, Circuit breaker가 룰에 따라서 다른 메세지를 리턴하게 하는 방법이다.

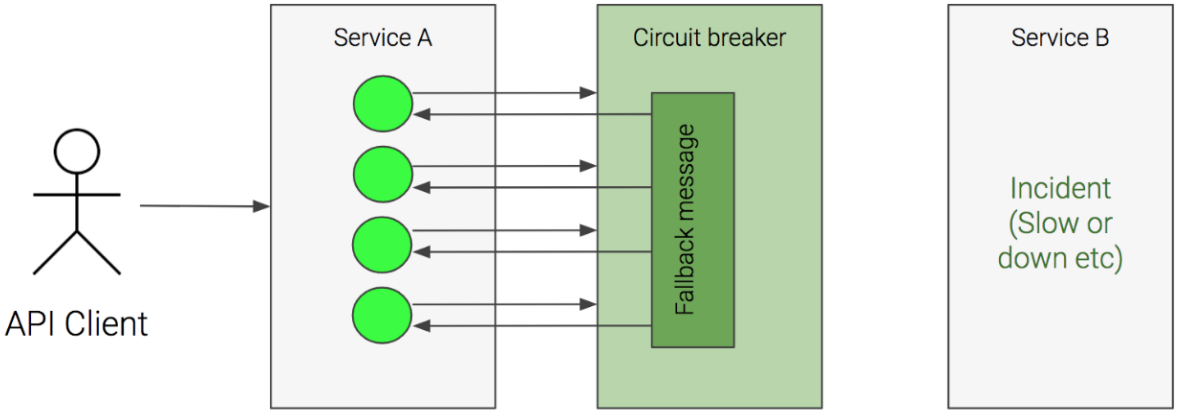

그림 4 서비스에 문제가 있을 경우

이 패턴은 넷플릭스에서 자바 라이브러리인 Hystrix로 구현이 되었으며, Spring 프레임웍을 통해서도 손쉽게 적용할 수 있다.

Hystrix Circuit Breaker 구현(spring)

우선 pom.xml 에 hystrix의 의존성을 추가시켜준다.

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

그림 5 pom.xml(project of model file)

springboot main 클래스에는 @EnableCircuitBreaker 어노테이션을 추가시켜준다. 또한 다른 서비스와 통신을 하기 위해 RestTemplate bean을 생성한다.

```
@EnableCircuitBreaker
@EnableEurekaClient
@SpringBootApplication
public class EurekaClient1Application {

    public static void main(String[] args) {
        SpringApplication.run(EurekaClient1Application.class, args);
    }

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

그림 6 Service A

동작 수행 서비스 파일을 작성해준다. 정상 동작시에는 간단한 String 하나를 return 해준다.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@RefreshScope
@RestController
public class SampleContoller {

    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand
    @RequestMapping("/hello")
    public String printHelloWorld() {

        String result = restTemplate.getForObject("http://eurekaclient2/hello", String.class);
        return result;
    }
}
```

그림 7 Hello 동작 수행 서비스 파일

만약 서비스가 정상동작을 하면 그림 8과 같은 동작을 수행하는 것을 볼 수 있다.

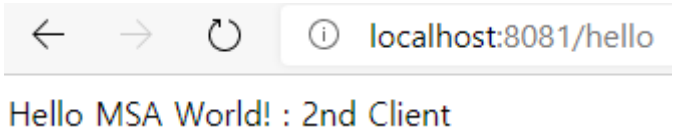

그림 8 정상 서비스 동작화면

Hystrix는 기본적으로 timeout을 1000ms로 설정되어 있다. 이를 1ms로 수정하여 반드시 timeout이 발생하는 조건으로 만들어주면 Circuit Breaker가 동작을 할 것이다.

```
@HystrixCommand(
    commandProperties = {
        @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="1")
    }
)
@RequestMapping("/hello")
public String printHelloWorld() {

    String result = restTemplate.getForObject("http://eurekaclient2/hello", String.class);
    return result;
}
```

그림 9 time out 값을 수정 (1ms)

Time out이 발생하여 Circuit Breaker가 동작하는 것을 알 수 있다.

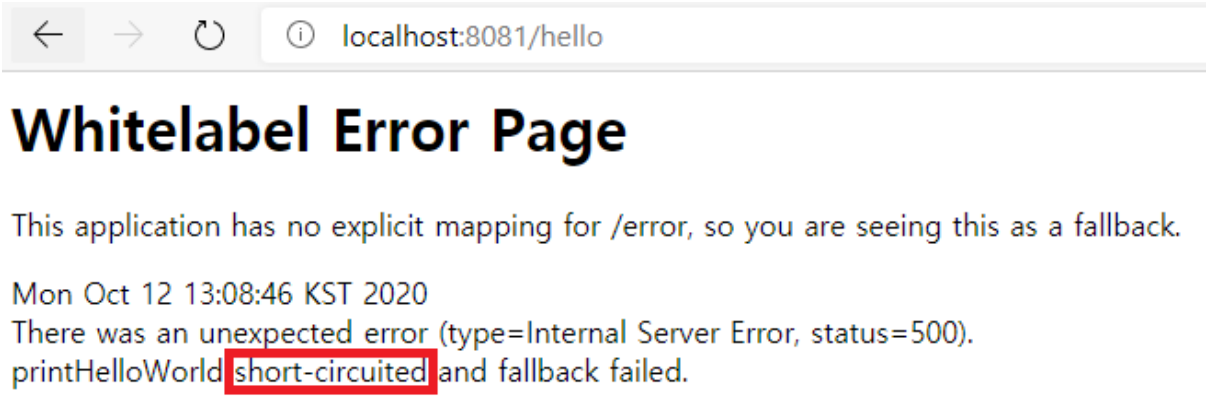

그림 10 Circuit Braker 동작

다음으로 Circuit Breaker가 동작할 때, 로깅 정보만을 보여주는 것이 아닌 다른 동작을 수행하게 하기위해서 Fallback 함수를 추가하여 서비스에 문제가 생길 시 다른 동작을 수행하게 서비스 A 파일을 수정한다.

```
@HystrixCommand(
    commandProperties = {
        @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="1")
    },
    fallbackMethod = "sampleFallback"
)
@RequestMapping("/hello")
public String printHelloWorld() {

    String result = restTemplate.getForObject("http://eurekaclient2/hello", String.class);
    return result;
}

private String sampleFallback() {
    return "circuit breaker on";
}
```

그림 11 Fallback 함수 추가

수정 이후 circuit breaker on이라는 문구를 출력하는 함수가 동작 하는 것을 확인 할 수 있다.

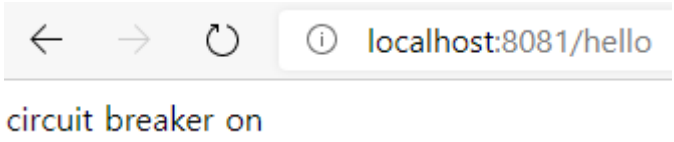

그림 12 Fallback 메시징

3. API gateway 패턴

트랜잭션 ID 추적 패턴

마이크로서비스 아키텍처를 기반으로 하게 되면, 클라이언트에서 호출된, 하나의 API 요청은 API 게이트웨이와 여러 서버를 거쳐서 처리된 후에, 최종적으로 클라이언트에 전달된다.

만약에 중간에 에러가 났을 경우, 어떤 호출이 어떤 서버에서 에러가 났는지를 연결해서 판단해야 할 수 가 있어야 한다. 예를 들어 서버 A,B,C를 거쳐서 처리되는 호출의 경우 서버 C에서 에러가 났을때, 이 에러가 어떤 메세지에 의해서 에러가 난 것이고, 서버 A,B에서는 어떻게 처리되었는 찾으려면, 각 서버에서 나오는 로그를 해당 호출에 대한 것인지 묶을 수 있어야 한다.

하나의 API 호출을 트랜잭션이라고 정의하며, 각 트랜잭션에 ID를 부여한다. API 호출시, HTTP Header에 이 트랜잭션 ID를 넣어 서버간의 호출시에 전달하면 하나의 트랜잭션을 구별할 수 있다.

여기에 추가적인 개념이 필요한데, 서버 A,B,C가 있을때, API 서버 B가 하나의 API 호출에서 두번

호출된다고 가정하시. 그러면 에러가 났을 경우 B 서버에 있는 로그중에, 몇 번째 호출에서 에러가 발생하였는지 알 수 없다.

아래 그림을 서버 A->B로의 첫번째 호출과, 두번째 호출 모두 트랜잭션 ID가 txid:1로, 이 txid로는 구별이 불가하다

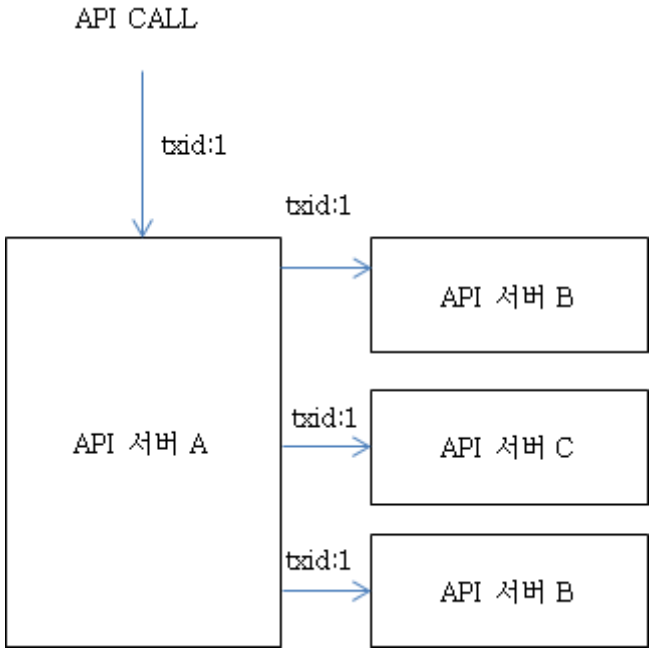

그림 13 하나의 트랜잭션 ID를 가진 패턴

그래서 이러한 문제를 해결하기 위해서는 글로벌 트랜잭션 ID(gtxid)와, 로컬 트랜잭션 ID (ltxid)의 개념을 도입할 수 있다.

API 호출을 하나의 트랜잭션으로 정의하면 이를 글로벌 트랜잭션 gtx라고 하고, 개별 서버에 대한 호출을 로컬 트랜잭션 ltx라고 한다. 이렇게 하면 아래 그림과 같이 하나의 API호출은 gtxid로 모두 연결될 수 있고 각 서버로의 호출은 ltxid로 구분될 수 있다

※ 사실 이 개념은 2개 이상의 데이터 베이스를 통한 분산 트랜잭션을 관리하기 위한 개념으로, 글로벌 트랜잭션과 로컬 트랜잭션의 개념을 사용하는데, 그 개념이 차용된 것이다

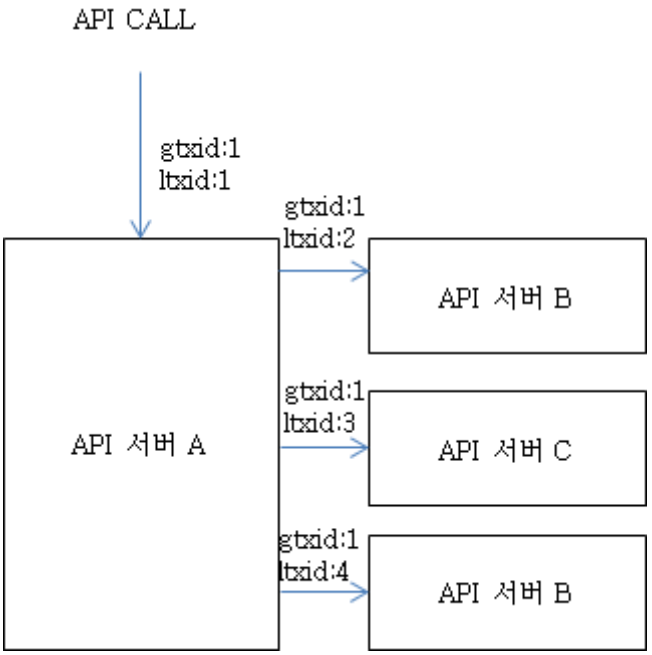

그림 14 두 종류의 트랜잭션 ID를 가진 패턴

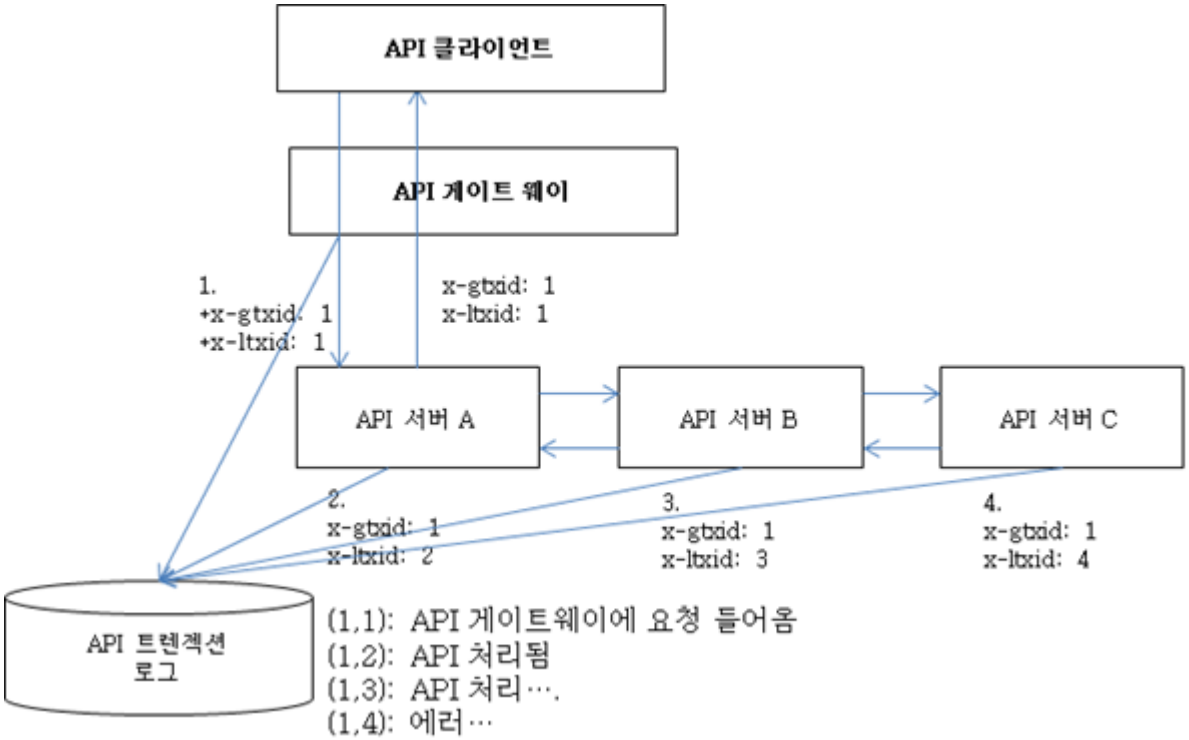

그림 15 API gateway와 함께 구현된 패턴

API 클라이언트는 API를 호출한다. API 클라이언트는 트랜잭션 ID에 대한 개념이 없다.

그림 15는 API gateway에서, HTTP 헤더를 체크해서 x-gtxid가 없으면 신규 API호출로 판단하고

트랜잭션 ID를 생성해서 HTTP 헤더에 채워 넣는다. 로컬 트랜잭션 ID로 1로 세팅한다.

2번에서, API 서버 A가 호출을 받으면, 서버 A는 헤더를 체크하여 ltxid를 체크하고, ltxid를 하나 더 더해서, 로그 출력에 사용한다. 같은 gtxid를 이용해서 같은 API호출임을 알 수 있고, ltxid가 다르기 때문에 해당 API서버로는 다른 호출임을 구별할 수 있다.

동일한 방식으로 서버B,C도 동일한 gtxid를 가지지만, 다른 ltxid를 갖게 된다.

각 API 서버는 이 gtxid와 ltxid로 로그를 출력하고, 중앙에서 로그를 수집해서 보면, 만약에 에러가 발생한 경우, 그 에러가 발생한 gtxid로 검색을 하면, 어떤 어떤 서버를 거쳐서 트랜잭션이 수행되었고 어떤 서버의 몇번째 호출에서 에러가 발생하였는지 쉽게 판별이 가능하다.

4. 결론

트랜잭션 ID와 마찬가지로 HTTP Header의 내용 부분에 특정 서비스의 name을 넣고, Fallback method에 error를 return 해주거나 특정 error message file을 작성해주는 함수와 다른 서비스로 연결해주는 함수를 넣어준다면 error control에 보다 나은 결과를 보여줄 것으로 예상된다.