gRPC (1)

Jae Hyeon Kim

Contents

1. 서론

2. HTTP

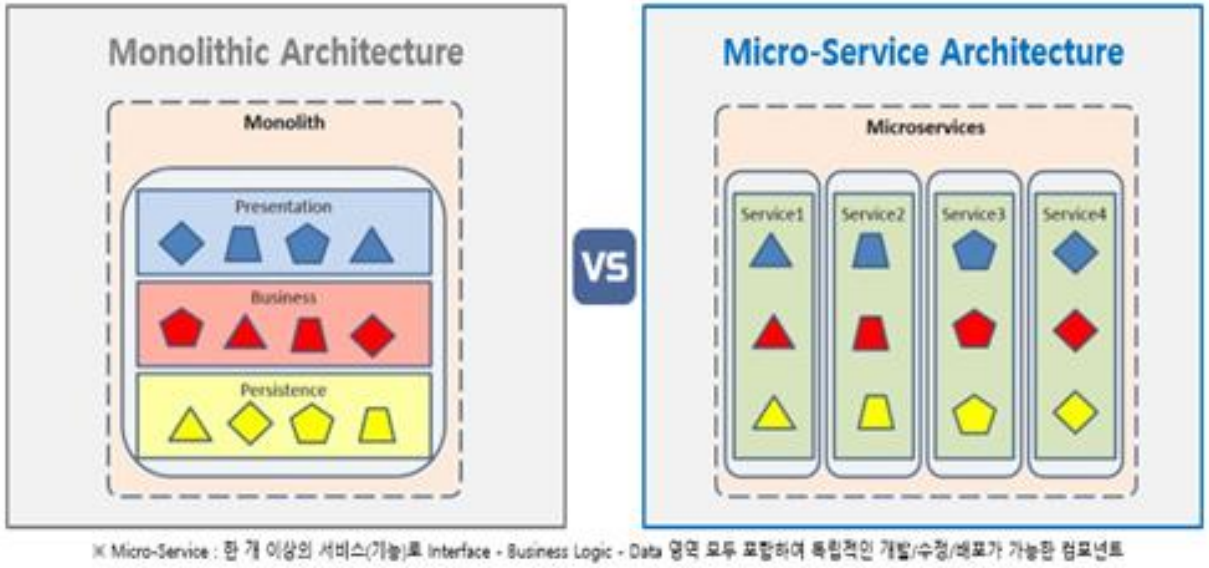
3. REST API의 문제점

4. gRPC Protobuf

서론

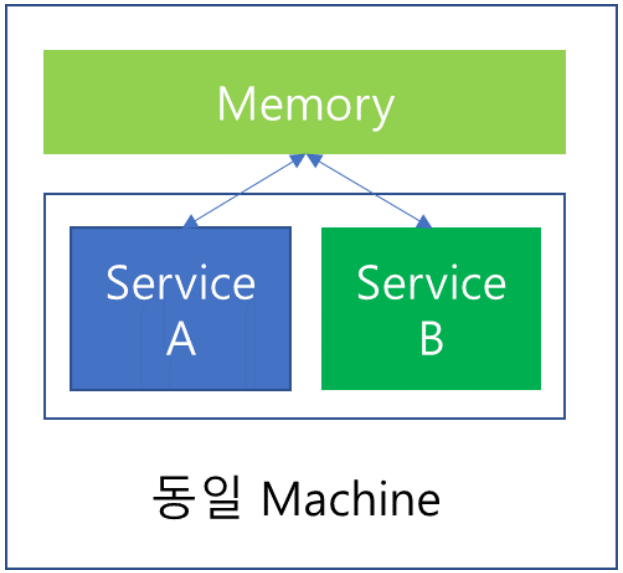
MSA & Network Communication

- 최근 많은 회사에서 Monolithic 구조를 여러 개의 Micro Service로 분리하고 있다
- MSA 구성은 다양한 장점을 가지지만, 그만큼 다양한 문제점 또한 존재한다
- 문제점 중 하나는 네트워크 통신에서의 overhead



MSA & Network Communication

- Monolithic 구조에서는 하나의 프로그램으로 동작하기 때문에 그 안에서 서비스 간의 데이터는 공유 메모리를 통해 주고받을 수 있다



MSA & Network Communication

- MSA는 여러 모듈로 분리되어 있기 때문에 데이터는 동일 머신에 존재하지 않을 수 있다
- 따라서 일반적으로 REST 통신을 통해 메시지를 주고 받는다

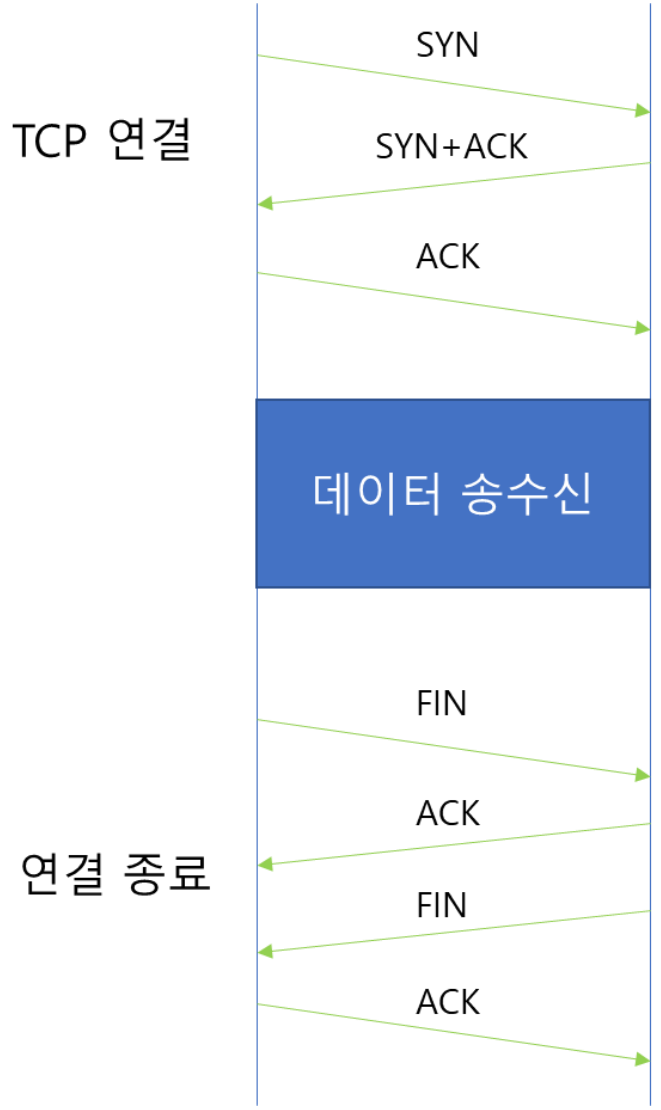
– MSA & Network Communication

- 문제는 하나의 응답을 만들어내기 위해 여러 마이크로 서비스 간의 협력이 필요하다면, 구간별 REST 통신에 따른 overhead로 인해 응답속도가 저하된다는 점이다

HTTP

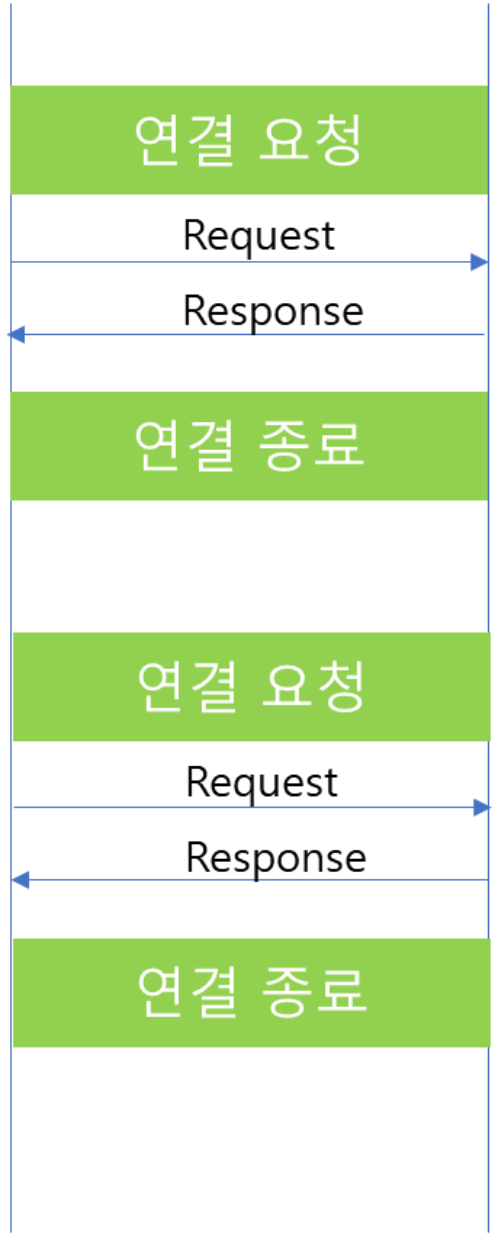
HTTP

- 하이퍼텍스트를 빠르게 교환하기 위한 프로토콜의 일종으로, 서버와 클라이언트 사이에서 어떻게 메시지를 교환할지를 정해 놓은 규칙
- TCP 기반에서 동작
- TCP 연결 시점에 3 way handshake 과정을 거치고, 연결을 종료할 땐 4 way handshake 방식으로 종료
- 이러한 경우 전송 응답을 반복해야 하는 상황이라면, 매번 연결을 맺고 종료하는 과정으로 비효율 발생



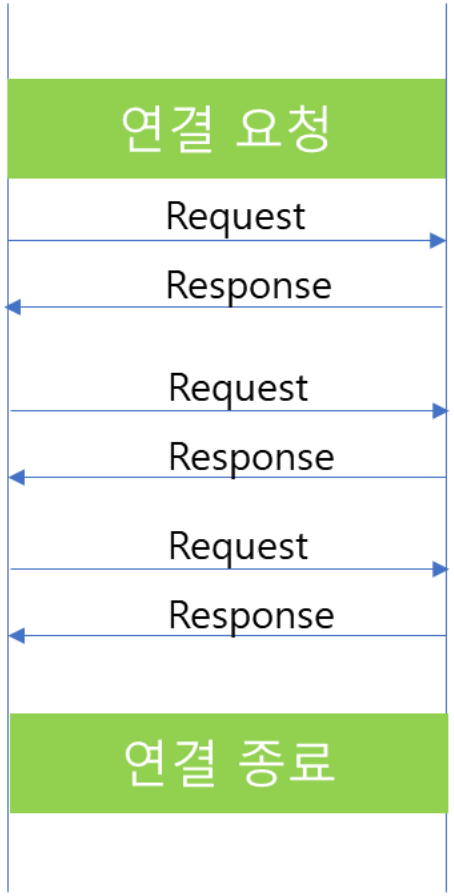
HTTP 1.0

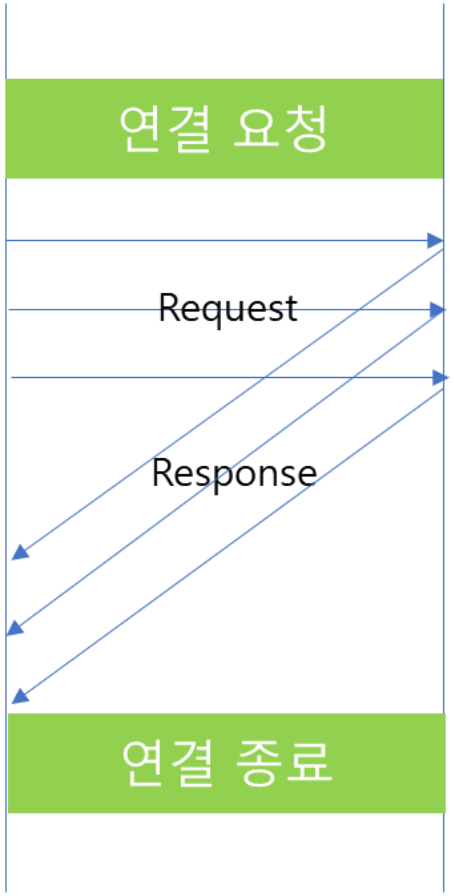
- HTTP 1.0은 요청/응답을 하기에 앞서 매번 connection을 맺고 끊어야 하기 때문에 연결 요청/해제 비용이 상당히 높다.
- 비효율적



HTTP 1.1

- HTTP 1.1에서 1.0의 문제점을 해결하고자 Persistent Connection과 Pipelining 기법을 제공





HTTP 1.1

- Pipelining 기법을 사용하면 요청 자체는 응답 여부와 관계없이 보낼 수 있다
- 하지만 여전히 순차적으로 응답을 받아야 한다
- HOLB(Head Of Line Blocking)문제 발생

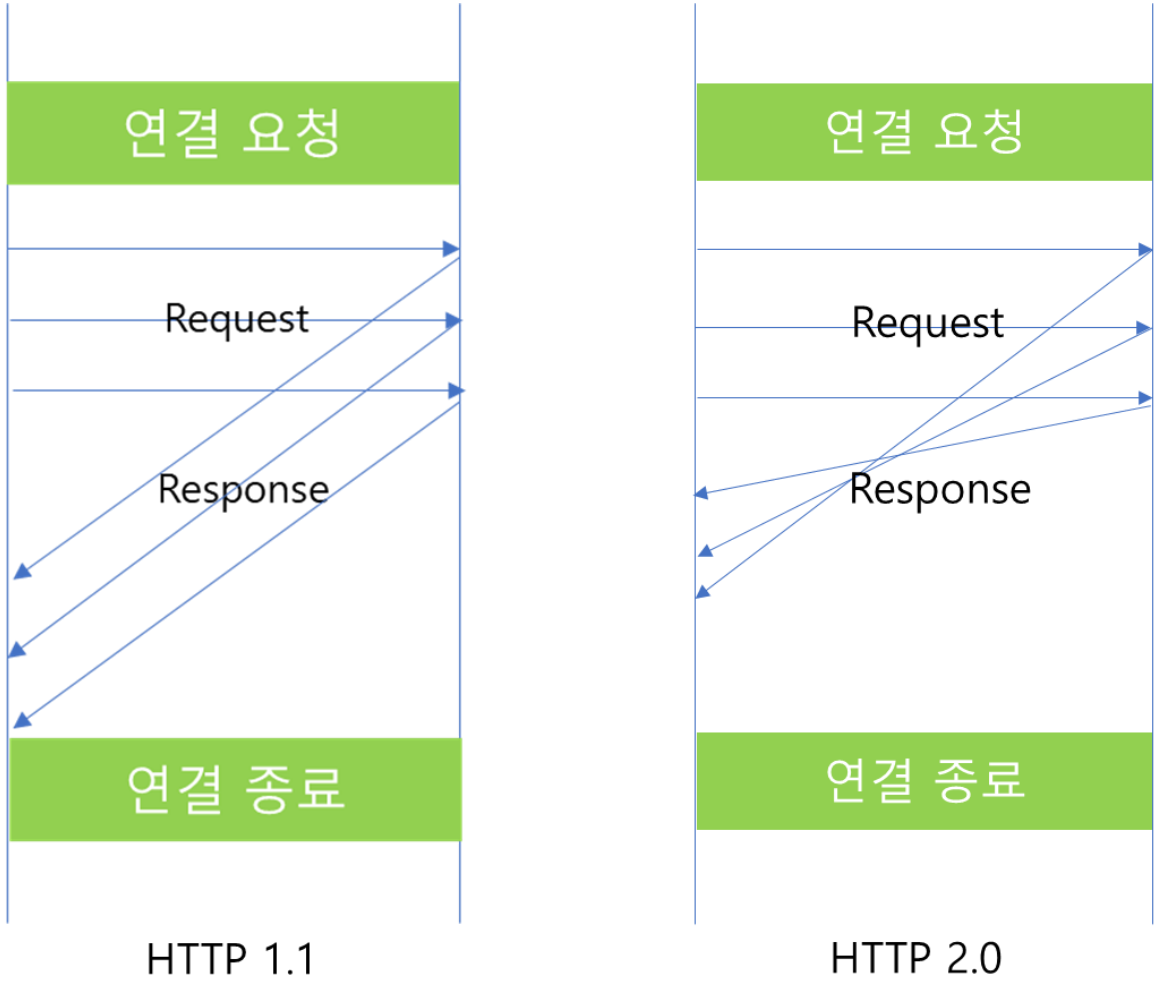
- 이러한 이슈를 해결하기 위해 대게 브라우저에서는 도메인당 기본 6개의 connection을 맺어 놓고 데이터를 병렬적으로 처리
- Domain Sharding을 통해 여러 도메인으로 데이터를 분산하고 병렬적으로 connection을 맺어 빠르게 많은 자원을 다운로드하도록 개선

HTTP 1.1

- HTTP 통신시 헤더에는 많은 메타 정보가 저장되어 있다
- 매 요청마다 중복된 헤더 값을 전달하며, 쿠키 또한 요청마다 포함되어 전송된다
- 더욱이 header 정보는 plain text로 전달되고 이는 binary에 비해 상대적으로 크기가 크기 때문에 전송 시 많은 비효율이 발생

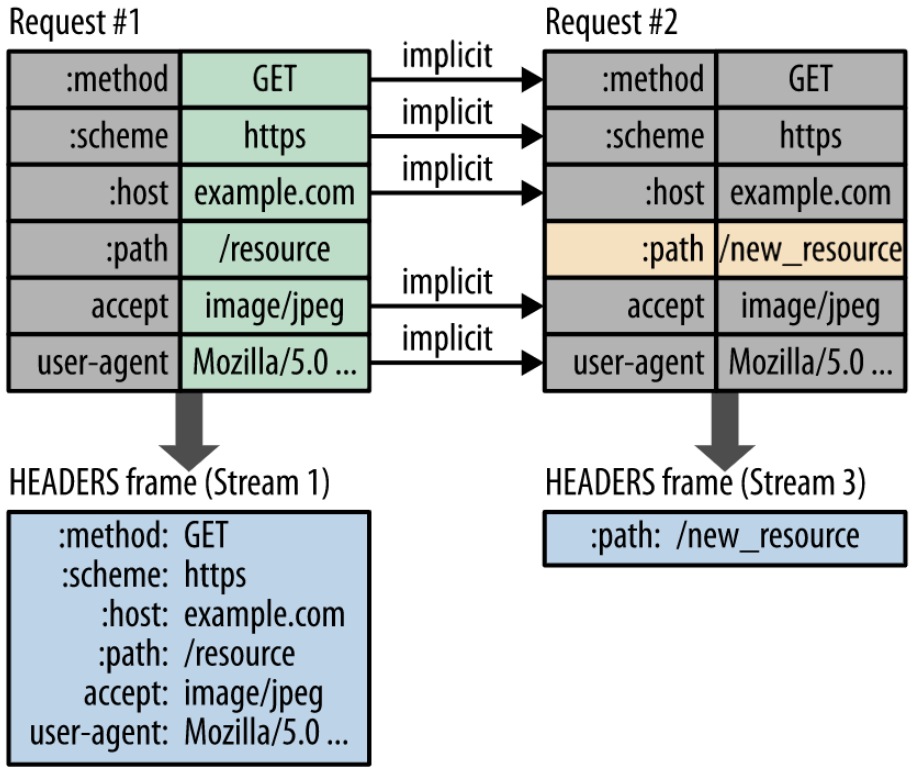
HTTP 2.0

- HTTP 2.0은 2014년에 표준안이 제안되고 15년에 공개된 프로토콜
- Multiplexed Streams을 통해 성능 개선
- 하나의 connection으로 여러 개의 데이터를 주고 받을 수 있도록 stream 처리가 가능
- 응답에 대해 우선순위가 주어져서 요청 순서와 관계없이 응답을 받을 수 있다



HTTP 2.0

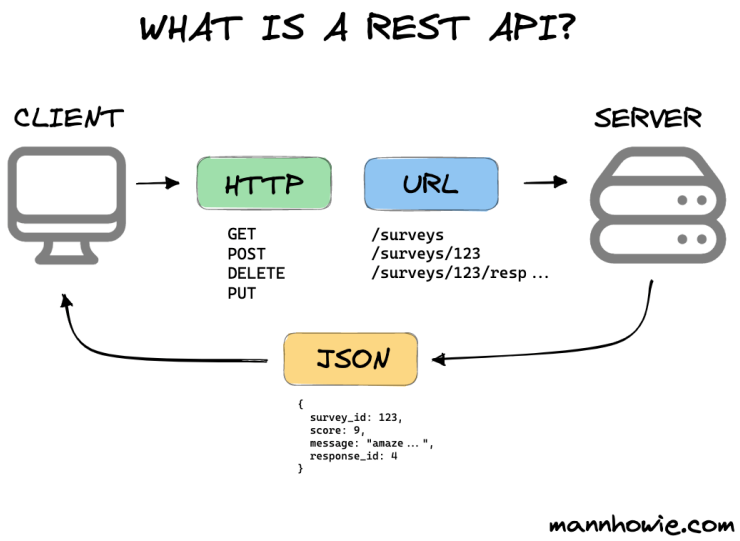
- Header 압축(huffman coding)을 통해 지속적인 데이터 요청에 대한 header 크기를 줄임
- 즉, 더 적은 connection으로 더 작은 header를 전송할 수 있으며 stream 통신으로 인해 여러 데이터를 주고 받을 수 있게 되었다

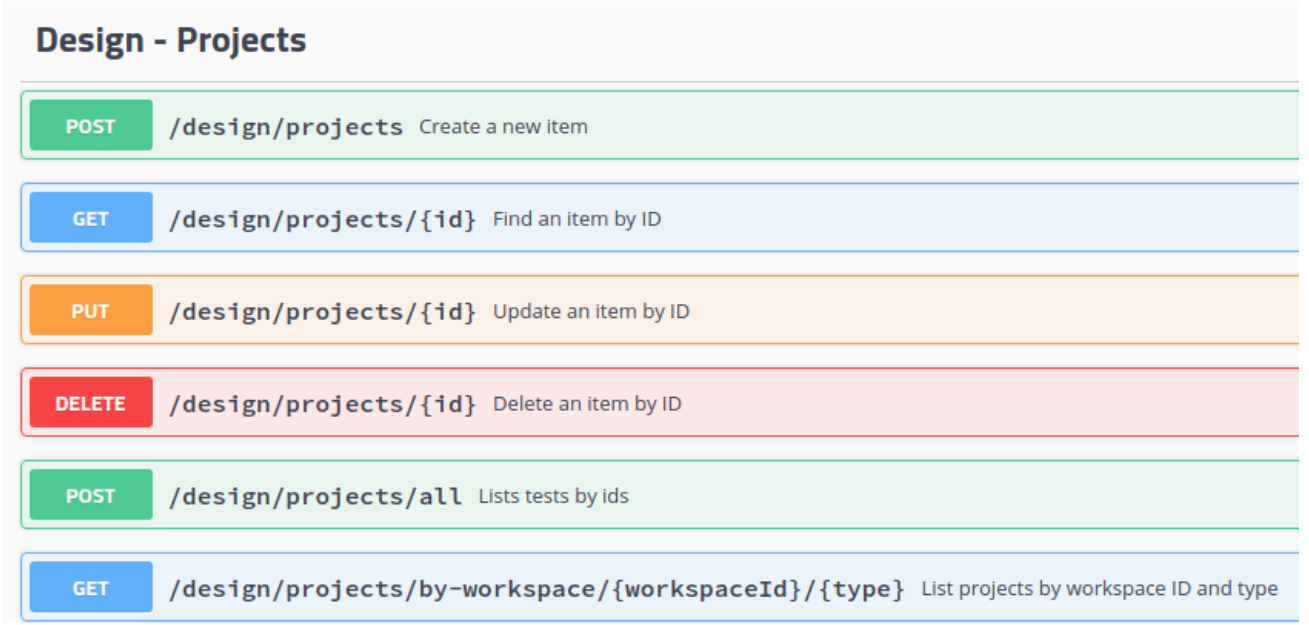


REST API의 문제점

REST API

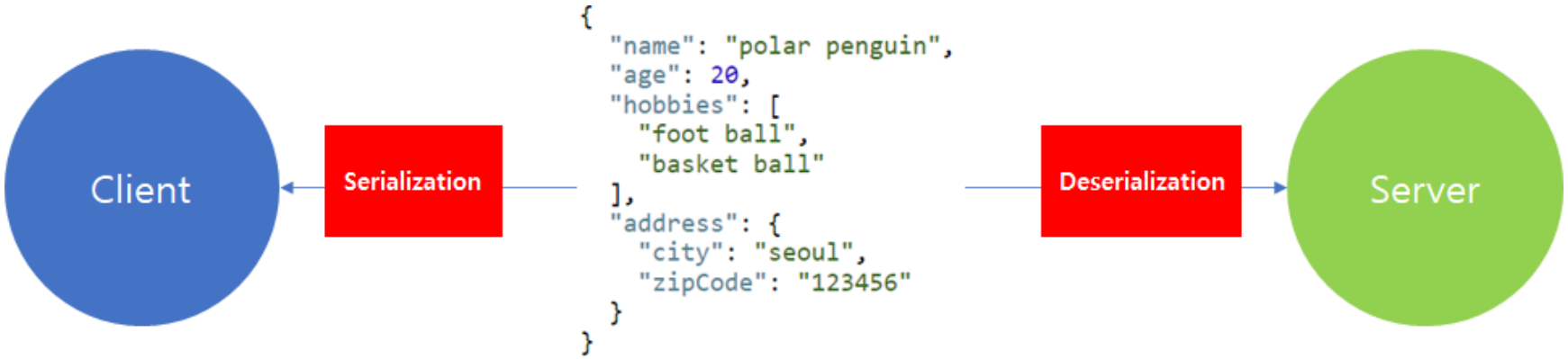
- URI를 통해 모든 자원을 명시하고 HTTP Method를 통해 처리하는 아키텍처
- 자원 그 자체를 표현하기에 직관적이고, HTTP를 그대로 계승하므로 별도 작업 없이도 쉽게 사용 가능





JSON Payload의 비효율

- REST 구조에서는 JSON 형태로 데이터를 주고 받는다
- JSON은 데이터 구조를 쉽게 표현할 수 있으며, 사람이 읽기 좋은 표현 방식
- 하지만 사람이 읽기 좋은 방식이란 머신 입장에서 자신이 읽을 수 있는 형태로 변환이 필요하다는 것을 의미
- 따라서 clien와 server 간의 데이터 송수신간에 JSON serialization과 deserialization 과정이 수반되어야 함



API Spec 정의 및 문서 표준화 부재

- REST API를 사용할 때 가장 큰 고민은 API 개발자와 사용자 간의 효율적인 커뮤니케이션 방법이다
- REST를 사용한다면 이를 위해서 자체적인 문서나 Swagger를 통해 API문서를 공유
- 하지만 이러한 방식은 REST와 관련된 표준은 아니다
- 또한 HTTP 메소드의 형태가 제한적이기 때문에 세부 기능 구현에는 제약이 있다

gRPC Protobuf

gRPC(Google Remote Procedure Call)

- Google이 개발한 고성능 오픈소스 범용 RPC 프레임워크
- Protocol Buffer 기반 HTTP 2.0을 사용

gRPC Protobuf

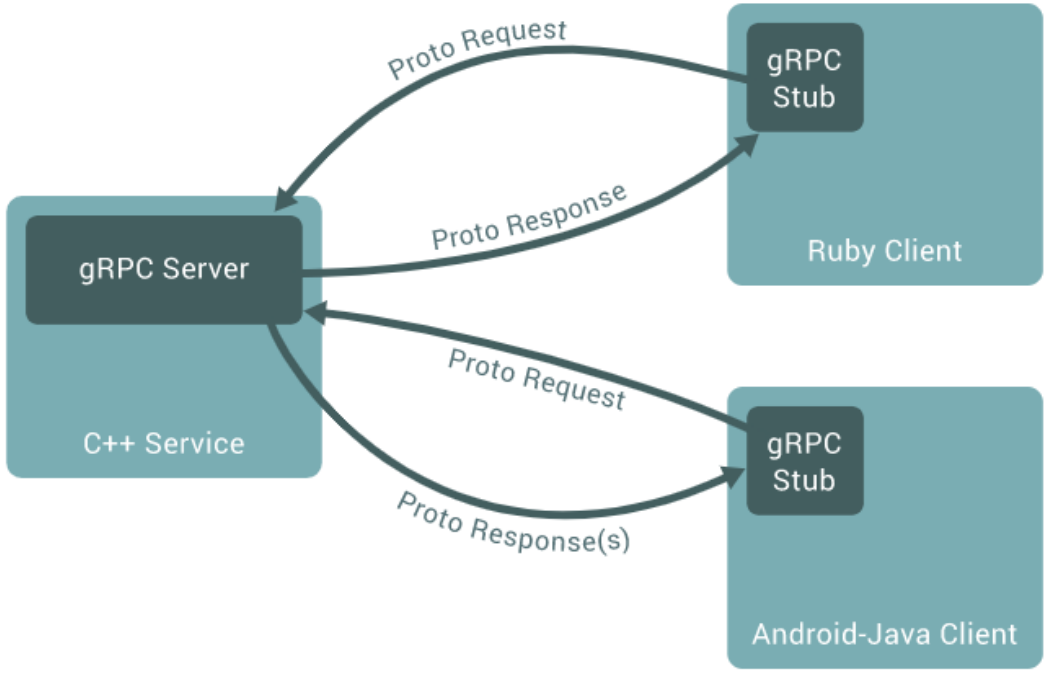
- Client에서 server측의 api를 호출하기 위해 기존에는 어떤 endpoint로 호출해야 할 지 그리고 전달 spec에 대해 api문서를 정의해야 했다
- 이를 해결하기 위한 방법은 serve의 기능을 사용할 수 있는 전용 library를 client에게 제공하는 것
- 그러면 client는 해당 library에서 제공하는 util 메소드를 활용해 호출하면 내부적으로는 server와 통신하여 올바를 결과를 제공한다
- 또한 serve에서 요구하는 spec에 부합되는 데이터만 보낼 수 있게 강제화 할 수 있다는 측면에서 스키마에 대한 제약을 가할 수 있다.

gRPC Protobuf

- Client에서 server측의 api를 호출하기 위해 기존에는 어떤 endpoint로 호출해야 할 지 그리고 전달 spec에 대해 api문서를 정의해야 했다
- 이를 해결하기 위한 방법은 server의 기능을 사용할 수 있는 전용 library를 client에게 제공하는 것
- 그러면 client는 해당 library에서 제공하는 util 메소드를 활용해 호출하면 내부적으로는 server와 통신하여 올바를 결과를 제공한다
- 또한 server에서 요구하는 spec에 부합되는 데이터만 보낼 수 있게 강제화 할 수 있다는 측면에서 스키마에 대한 제약을 가할 수 있다.

gRPC Protobuf

- gRPC에서는 stub 클래스를 client에게 제공한다
- Client는 stub을 통해서만 gRPC 서버와 통신을 수행할 수 있다

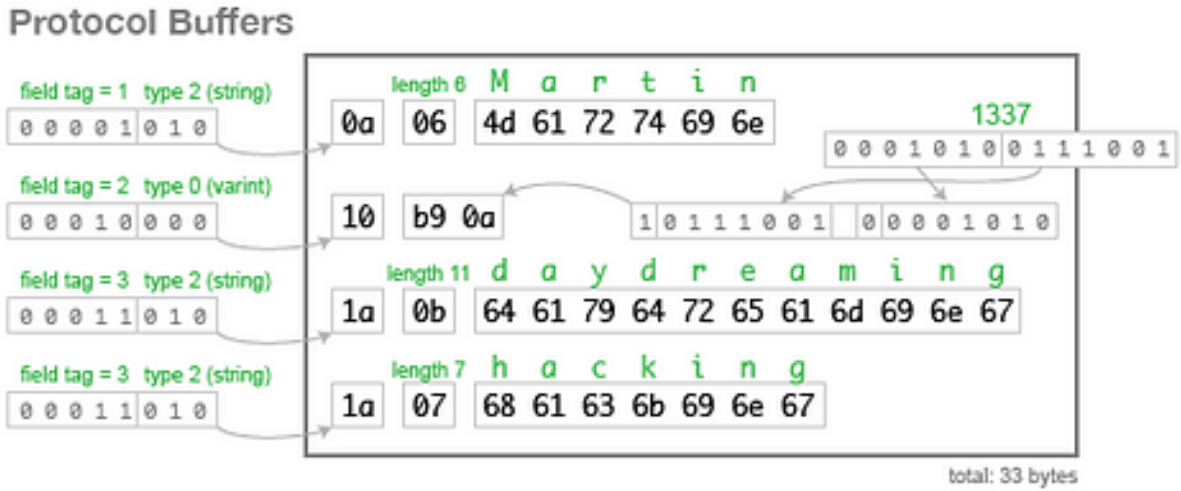


Protocol buffer

- Protocol buffer는 google이 공개한 데이터 구조로써, 특정 언어 혹은 플랫폼에 종속적이지 않은 데이터 표현 방식

```
{
    "userName": "Martin",
    "favouriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

```
message Person {
    required string user_name        = 1;
    optional int64  favourite_number = 2;
    repeated string interests        = 3;
}
```



gRPC Stub

- Stub 클래스를 생성하면, 해당 클래스 정보를 server와 client에 공유한 다음 양방향 통신 수행

```
stub
    .register(
        person {  this: PersonKt.Dsl
            name = "kevin"
            age = (1..50).random()
            address = address {  this: AddressKt.Dsl
                city = "seoul"
                zipCode = "123456"
            }
            hobbies.addAll(listOf("foot ball", "basket ball"))
        }
    )
```

gRPC

- gRPC 통신에서는 데이터를 송수신할 때 binary로 데이터를 encoding 해서 보내고 이를 decoding 해서 매핑한다
- 따라서 JSON에 비해 payload 크기가 상당히 적다