

HTTP 웹 기본 지식

모든 개발자를 위한

김영한

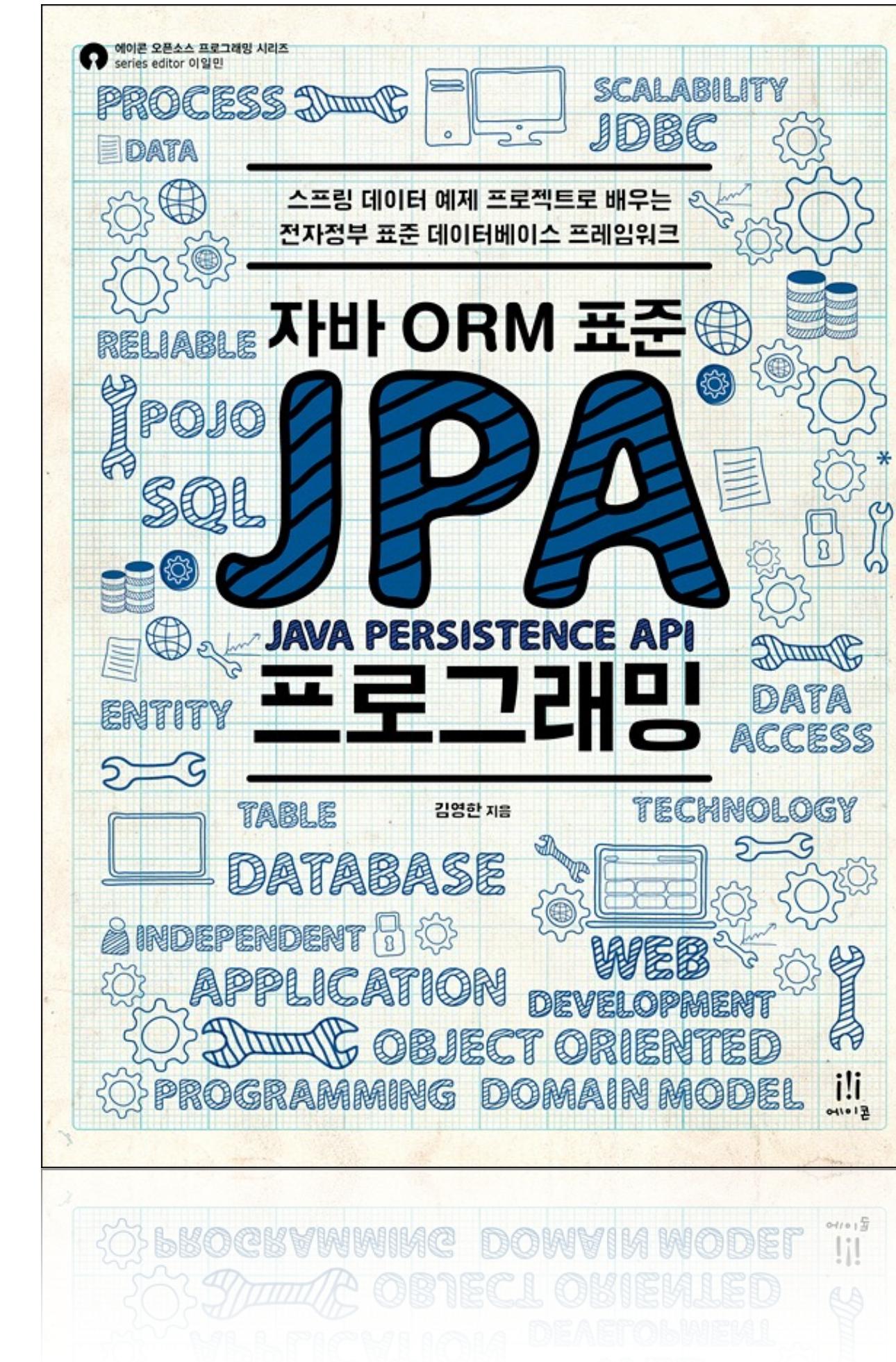
패치

- 2023-07-12
 - User-Agent 슬라이드 누락 해결(JIWON님 도움)
- 2023-06-06
 - 오태(b611219, haj1003, Woolly, jgam님 도움)
- 2022-02-21
 - 400ms 오태 (pj5016님 도움)

김영한

SI, Java EE 강사
카카오, SK 플래닛
우아한형제들 개발 팀장

저서: 자바 ORM 표준
JPA 프로그래밍

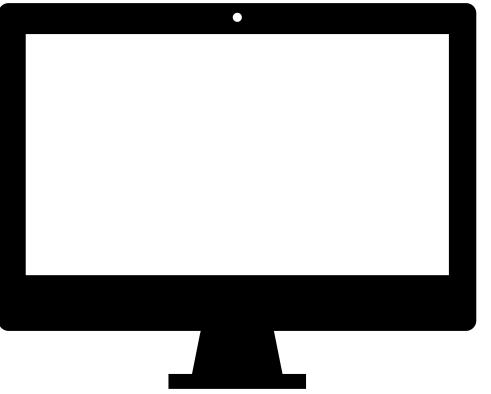


[인터넷 네트워크]

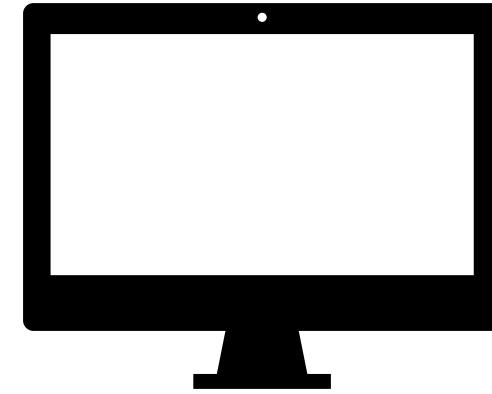
- 인터넷 통신
- IP(Internet Protocol)
- TCP, UDP
- PORT
- DNS

인터넷 통신

인터넷에서 컴퓨터들은 어떻게 통신할까?

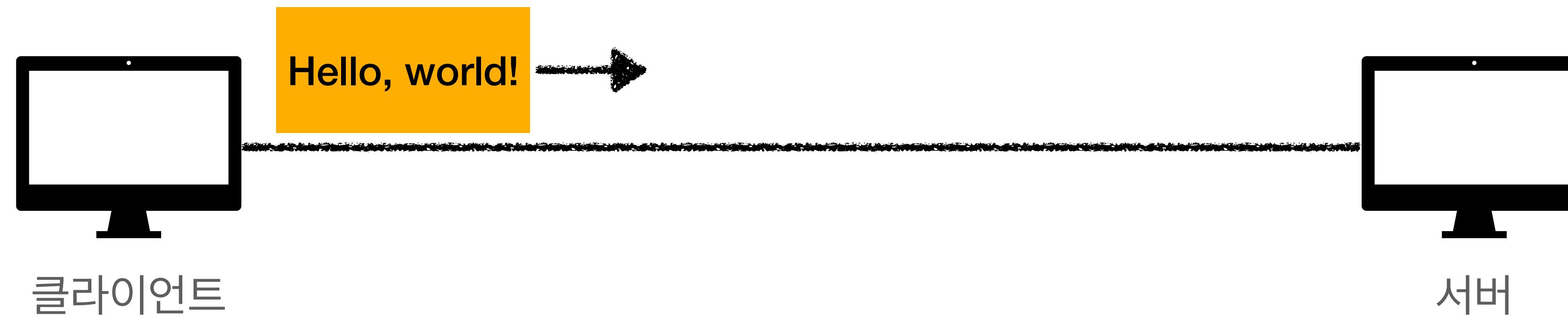


클라이언트

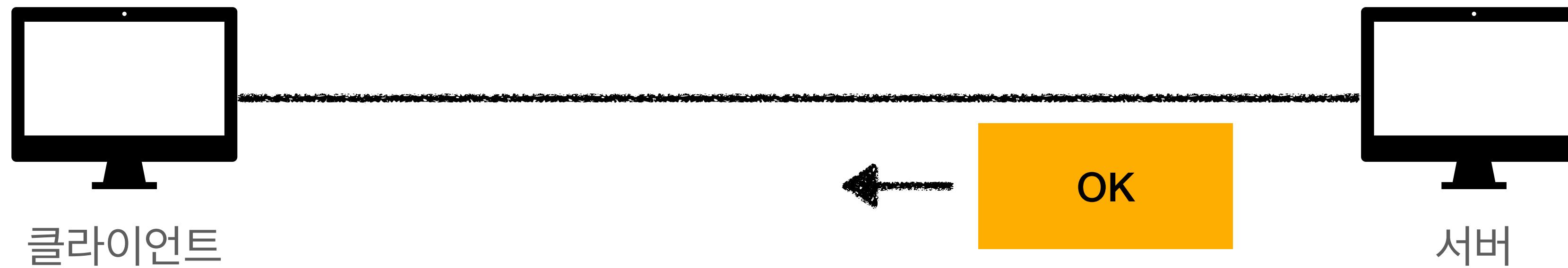


서버

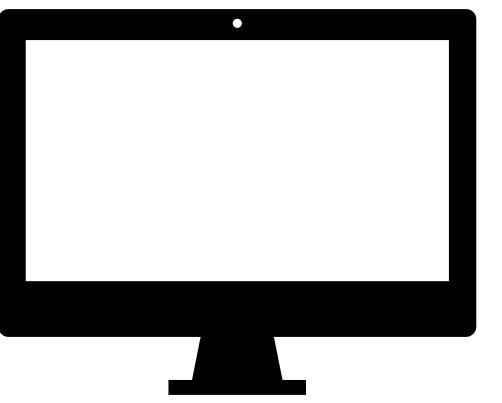
인터넷에서 컴퓨터들은 어떻게 통신할까?



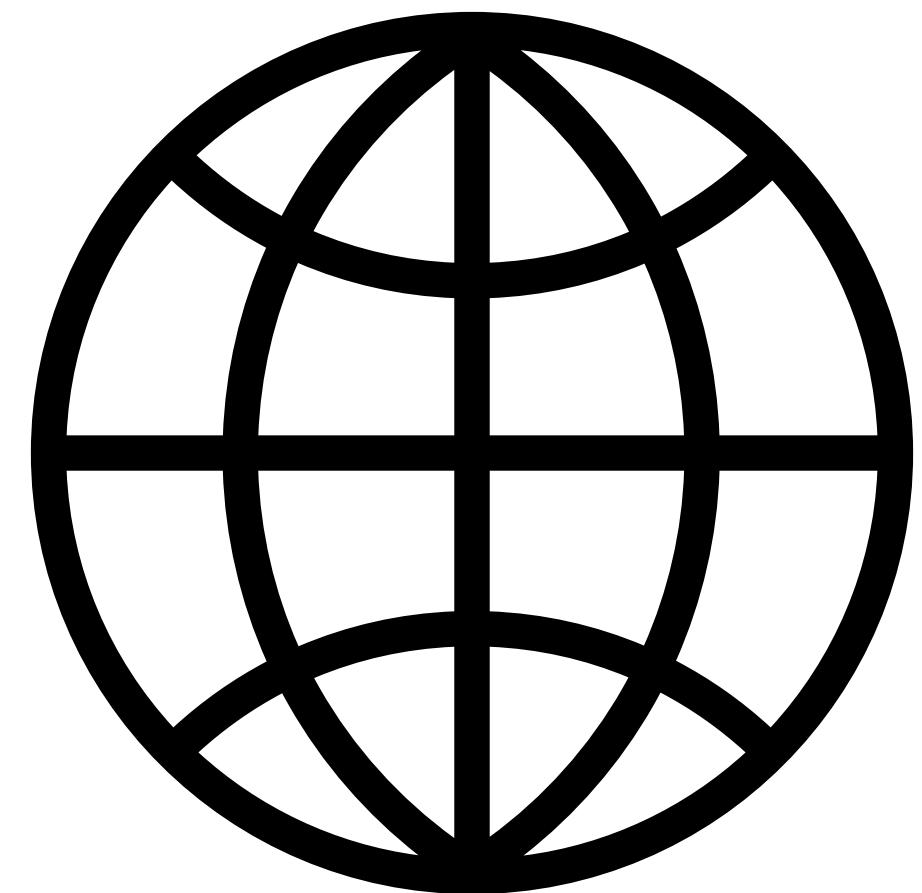
인터넷에서 컴퓨터들은 어떻게 통신할까?



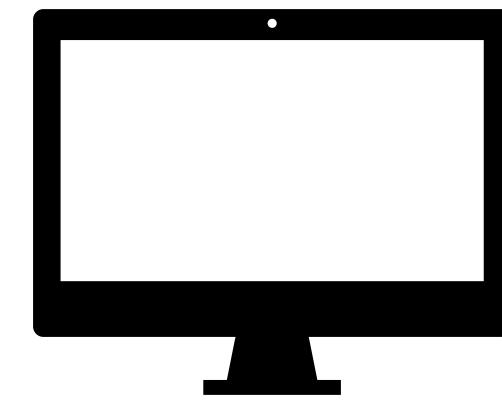
인터넷



클라이언트

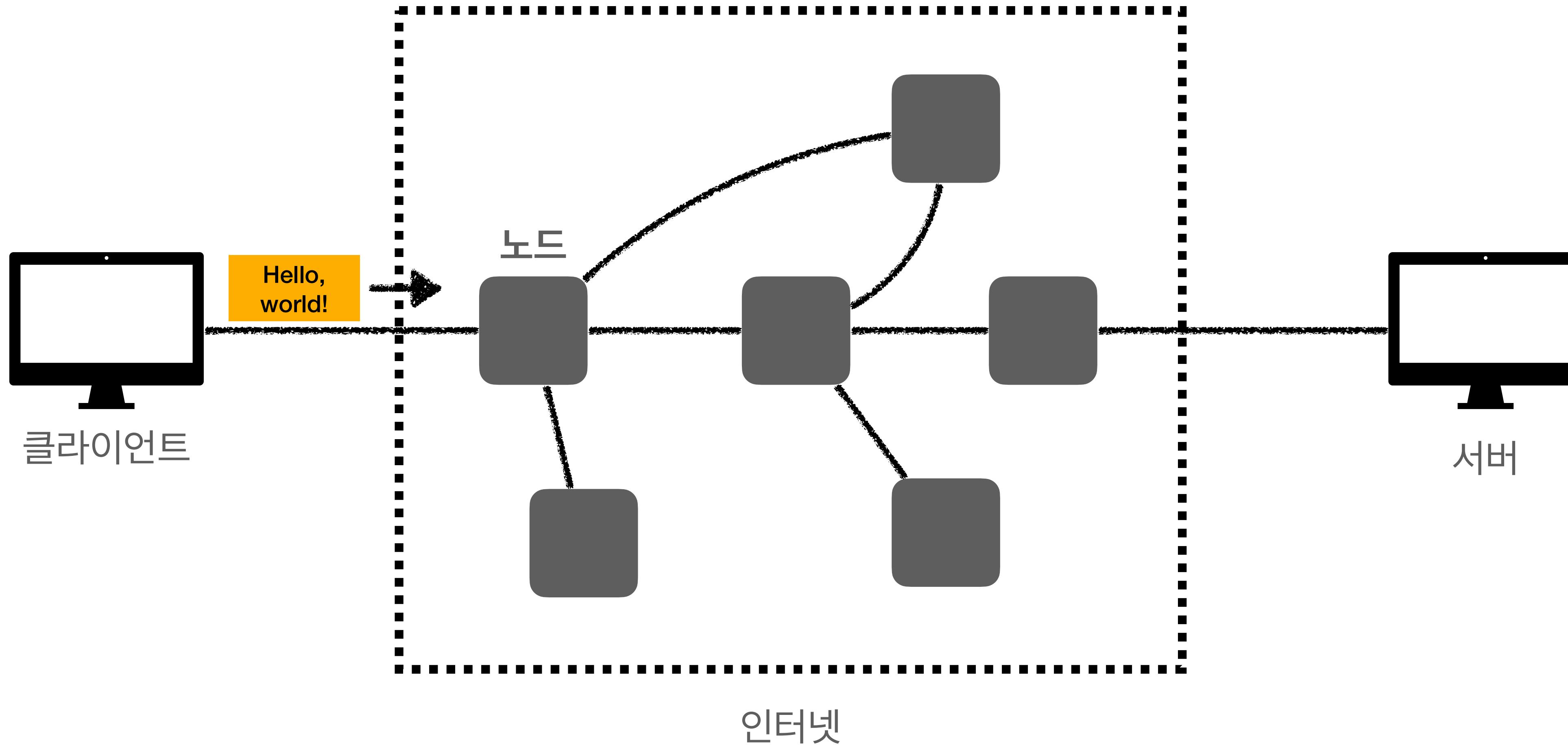


인터넷

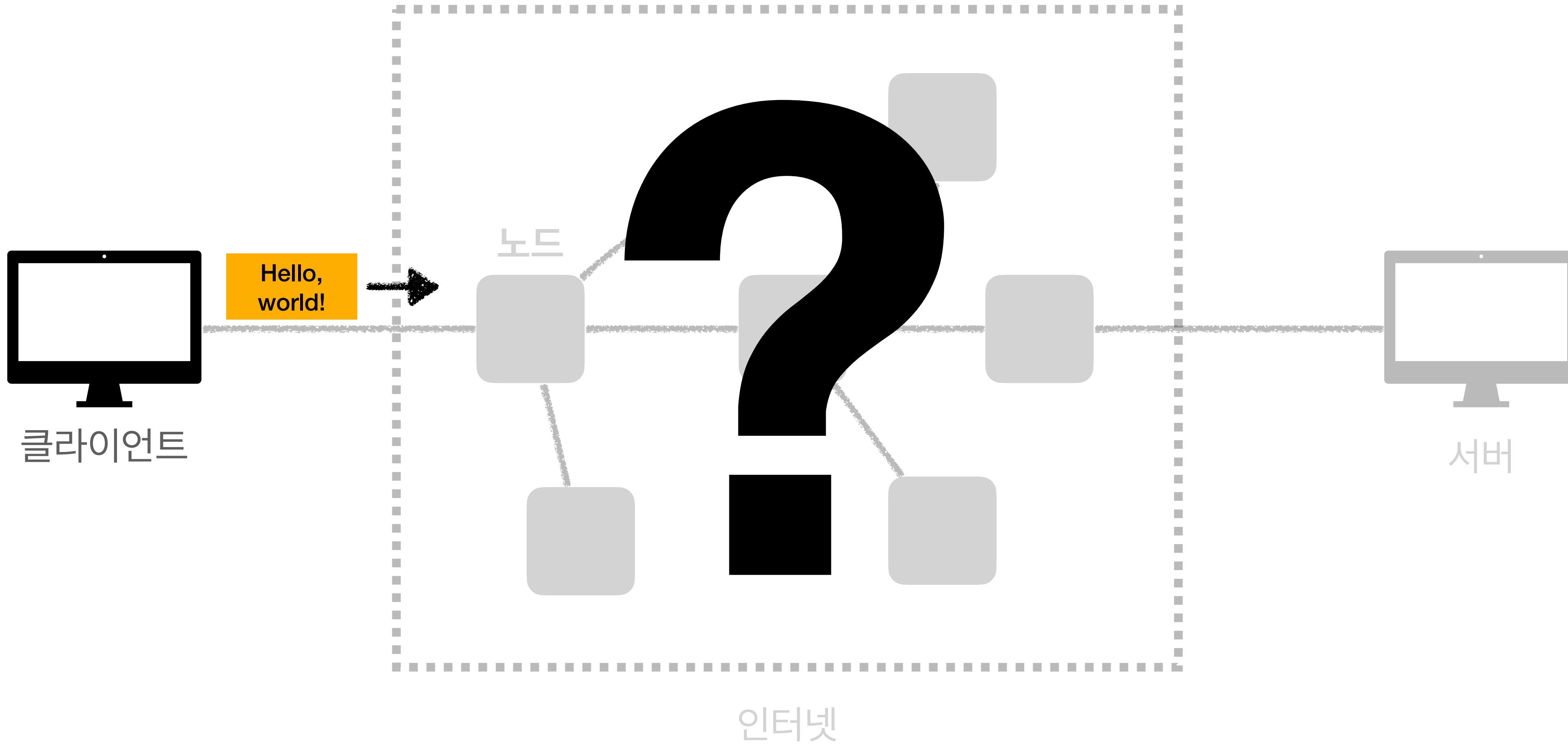


서버

복잡한 인터넷 망

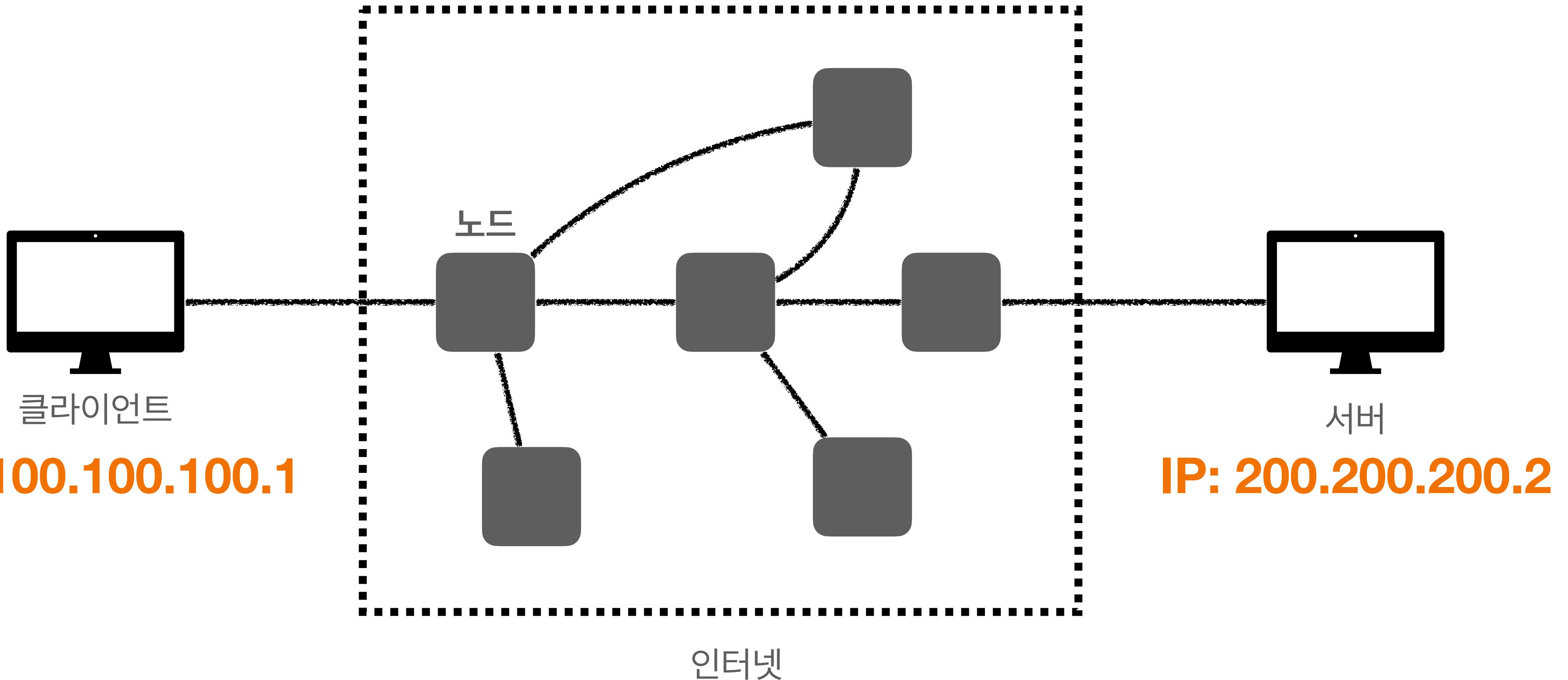


복잡한 인터넷 망



IP(인터넷 프로토콜)

IP 주소 부여



IP

인터넷 프로토콜 역할

- 지정한 IP 주소(IP Address)에 데이터 전달
- 패킷(Packet)이라는 통신 단위로 데이터 전달

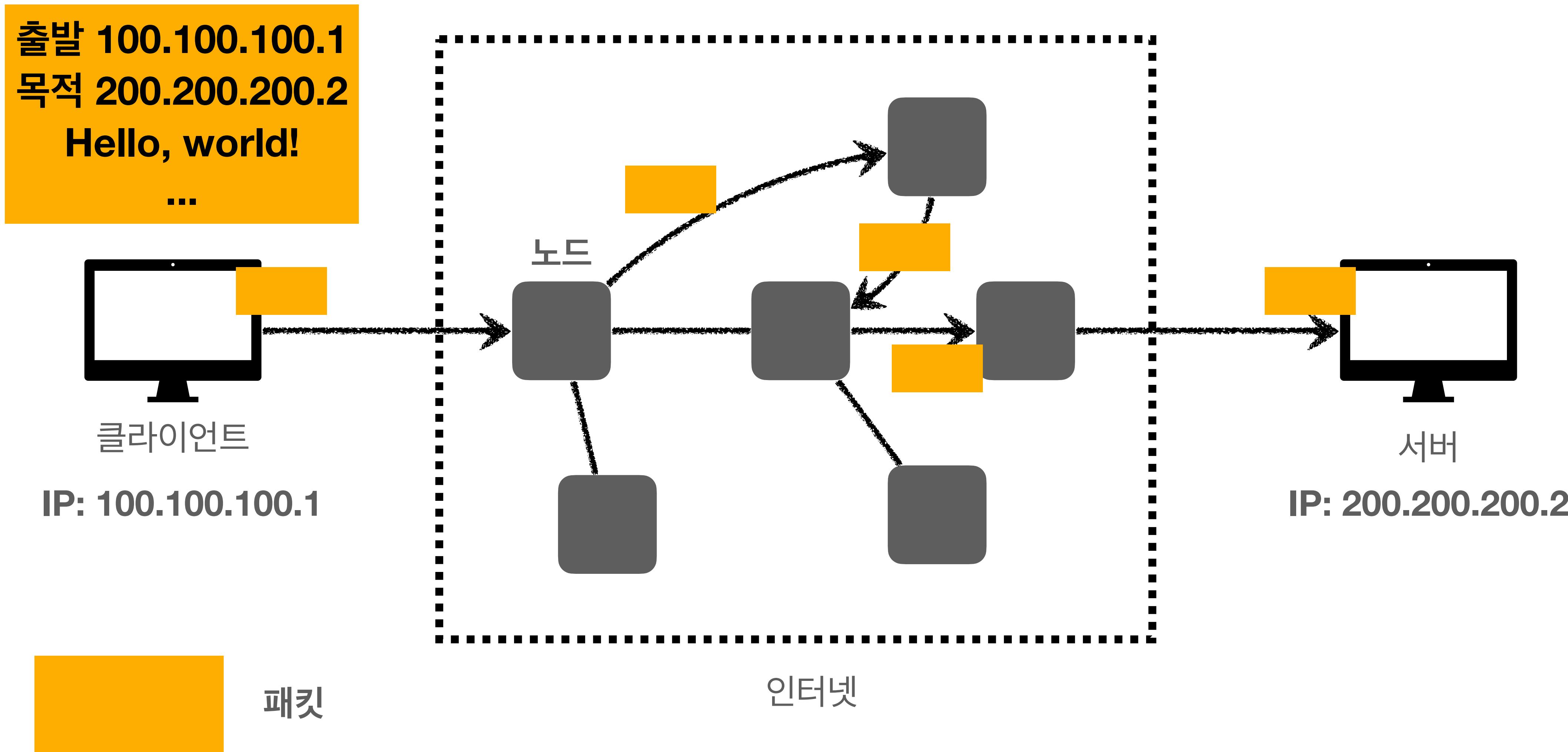
IP 패킷 정보

출발지 IP, 목적지 IP, 기타...

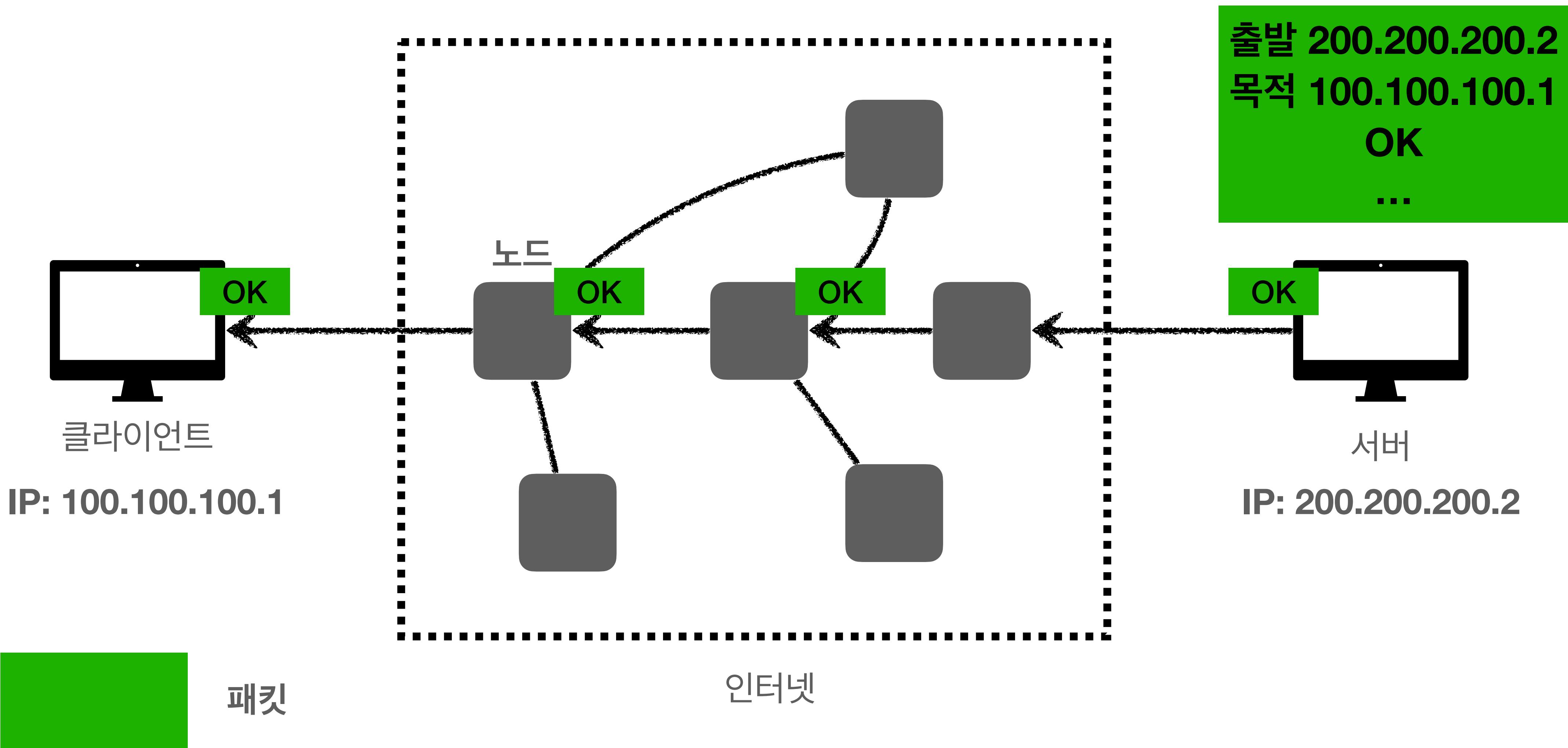
전송 데이터

IP 패킷

클라이언트 패킷 전달



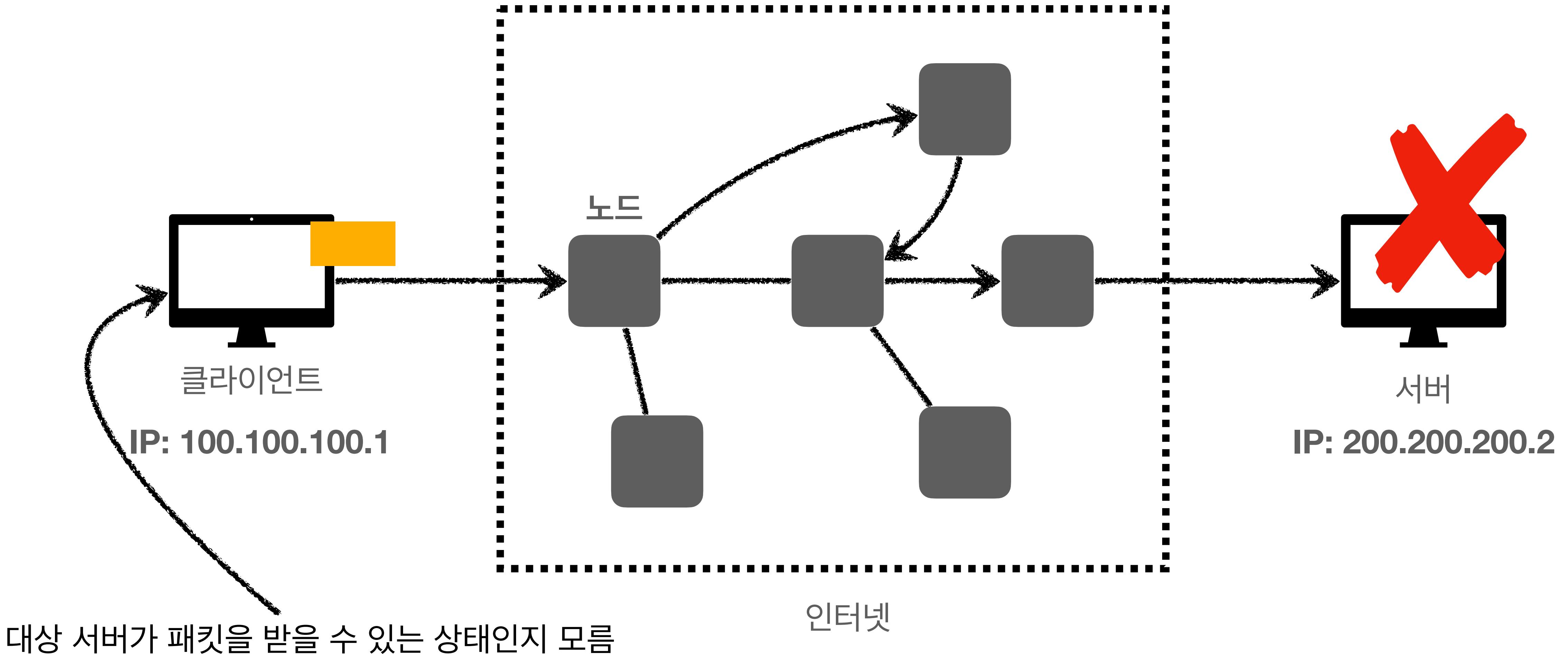
서버 패킷 전달



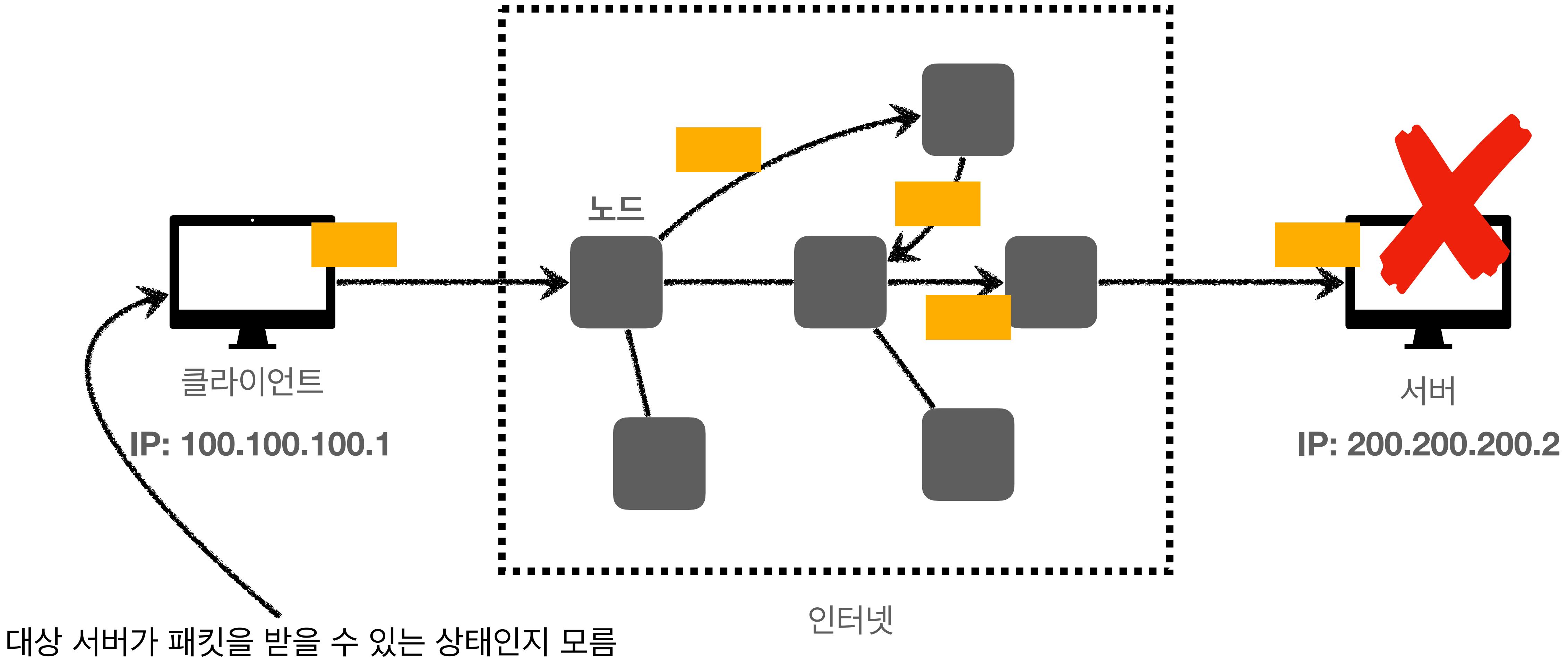
IP 프로토콜의 한계

- **비연결성**
 - 패킷을 받을 대상이 없거나 서비스 불능 상태여도 패킷 전송
- **비신뢰성**
 - 중간에 패킷이 사라지면?
 - 패킷이 순서대로 안오면?
- **프로그램 구분**
 - 같은 IP를 사용하는 서버에서 통신하는 애플리케이션이 둘 이상이면?

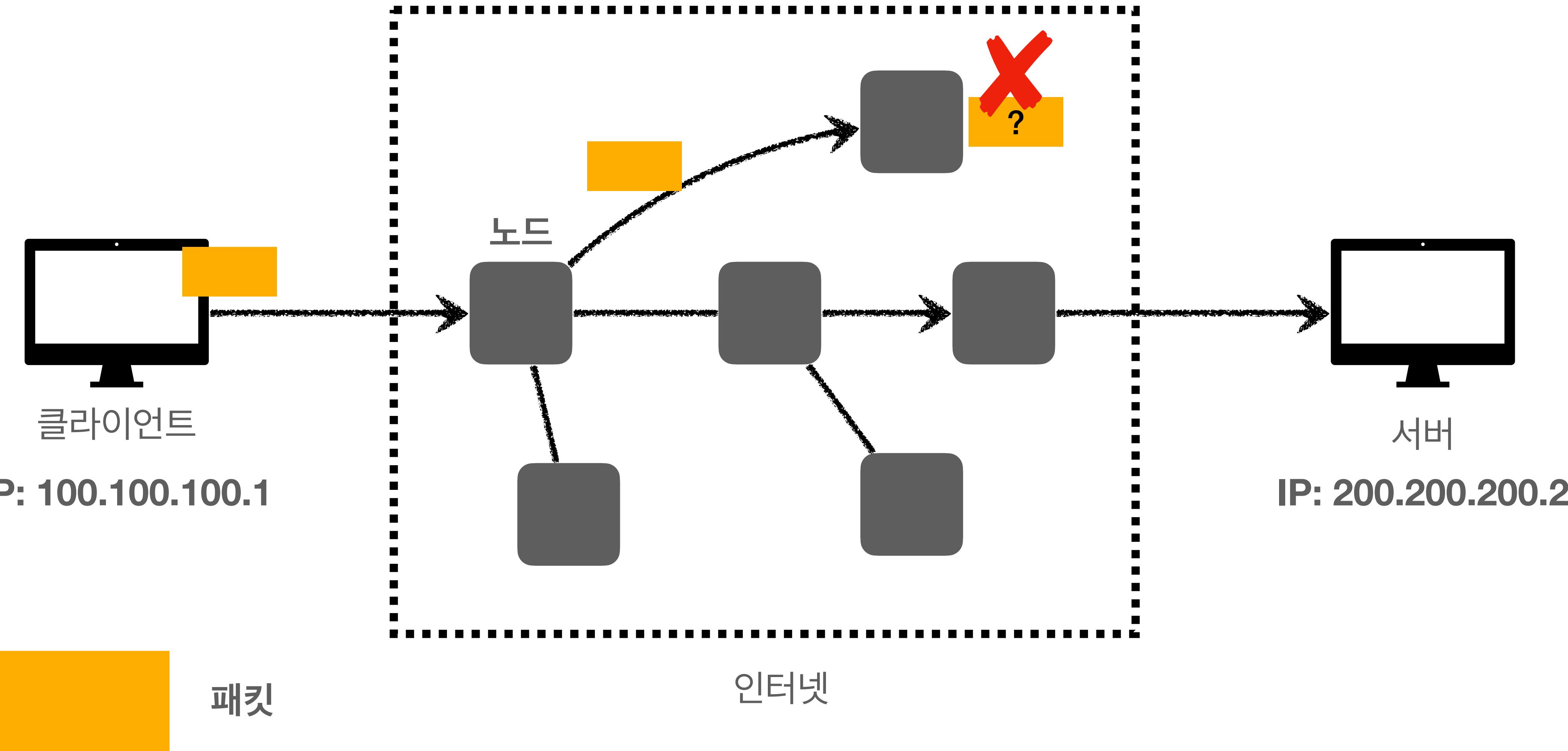
대상이 서비스 불능, 패킷 전송



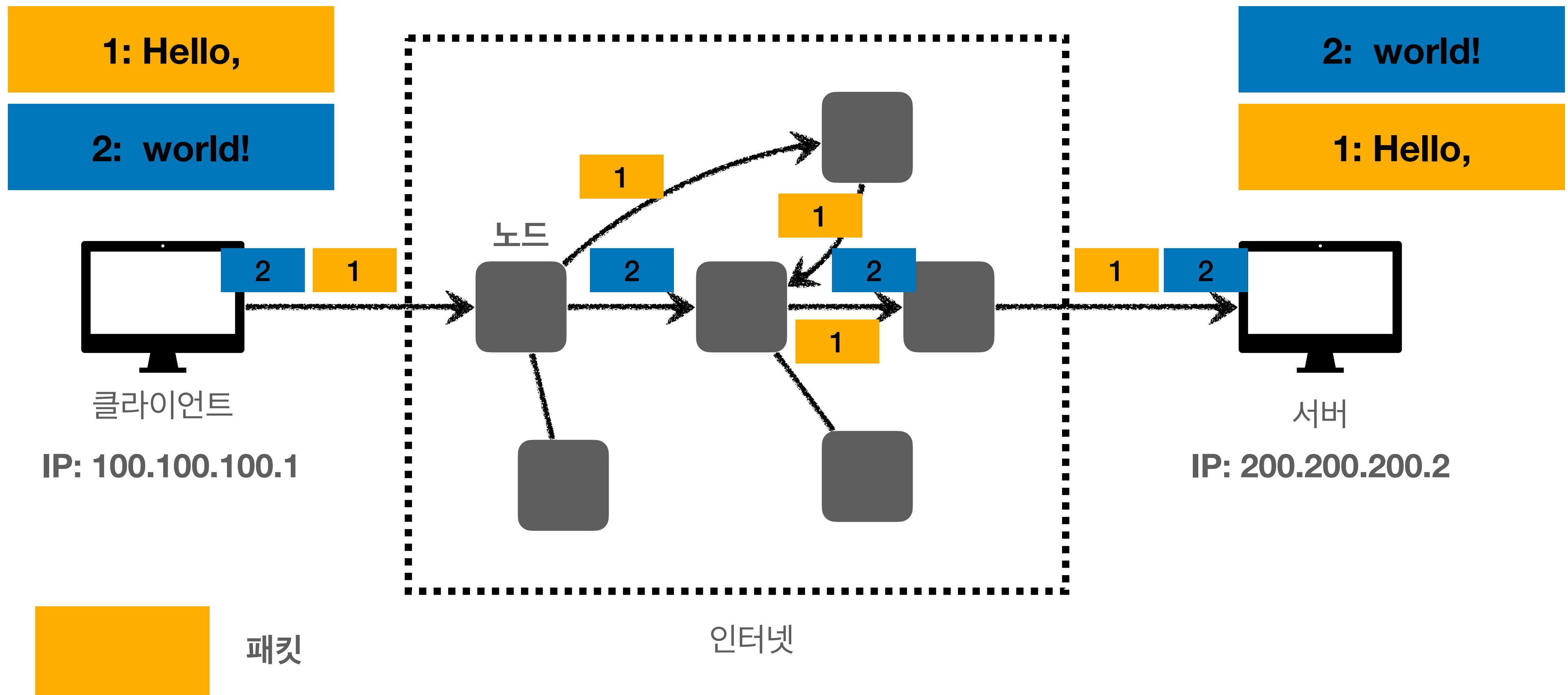
대상이 서비스 불능, 패킷 전송



패킷 소실



패킷 전달 순서 문제 발생



TCP UDP

인터넷 프로토콜 스택의 4계층

인터넷 프로토콜 스택의 4계층

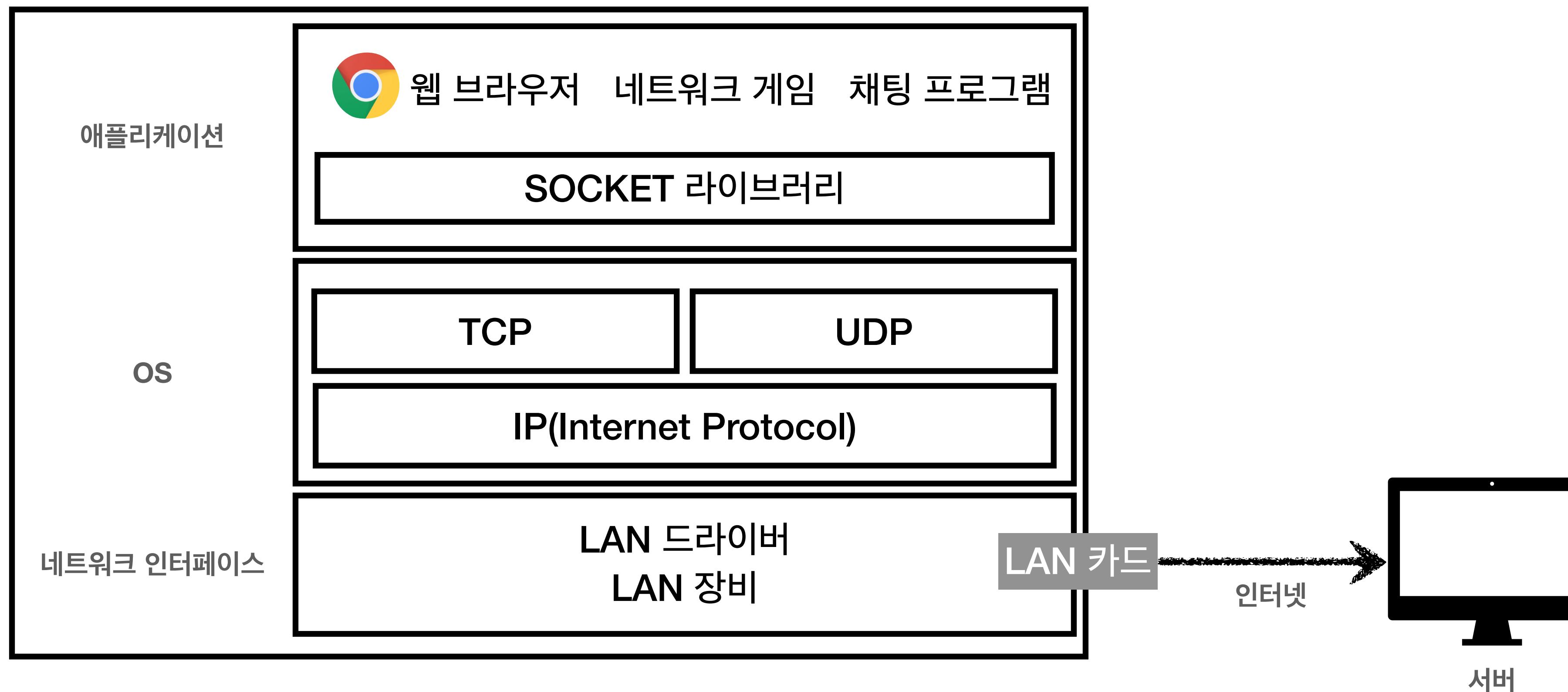
애플리케이션 계층 - HTTP, FTP

전송 계층 - TCP, UDP

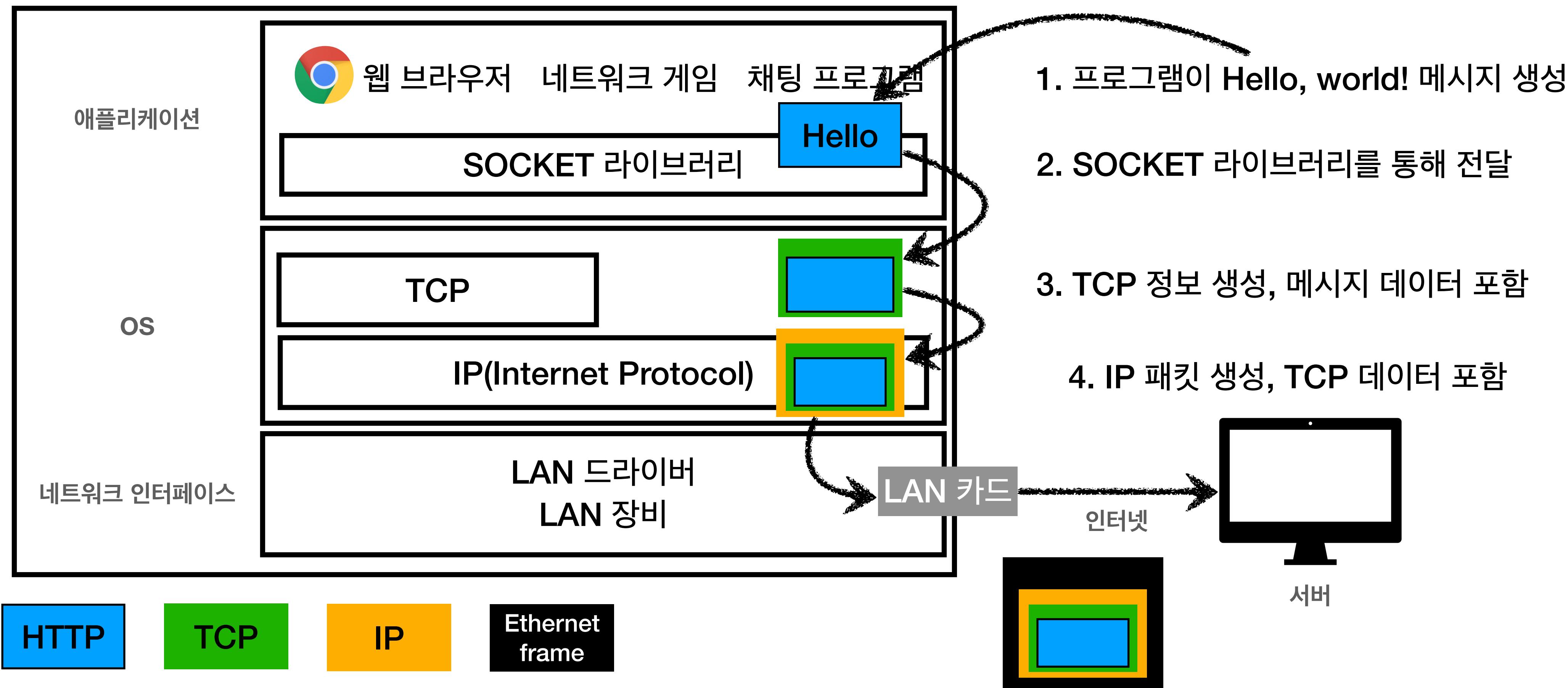
인터넷 계층 - IP

네트워크 인터페이스 계층

프로토콜 계층



프로토콜 계층



IP 패킷 정보

출발지 IP, 목적지 IP, 기타...

전송 데이터

IP 패킷

TCP/IP 패킷 정보

출발지 IP, 목적지 IP, 기타...

출발지 PORT, 목적지 PORT
전송 제어, 순서, 검증 정보...

전송 데이터



IP 패킷



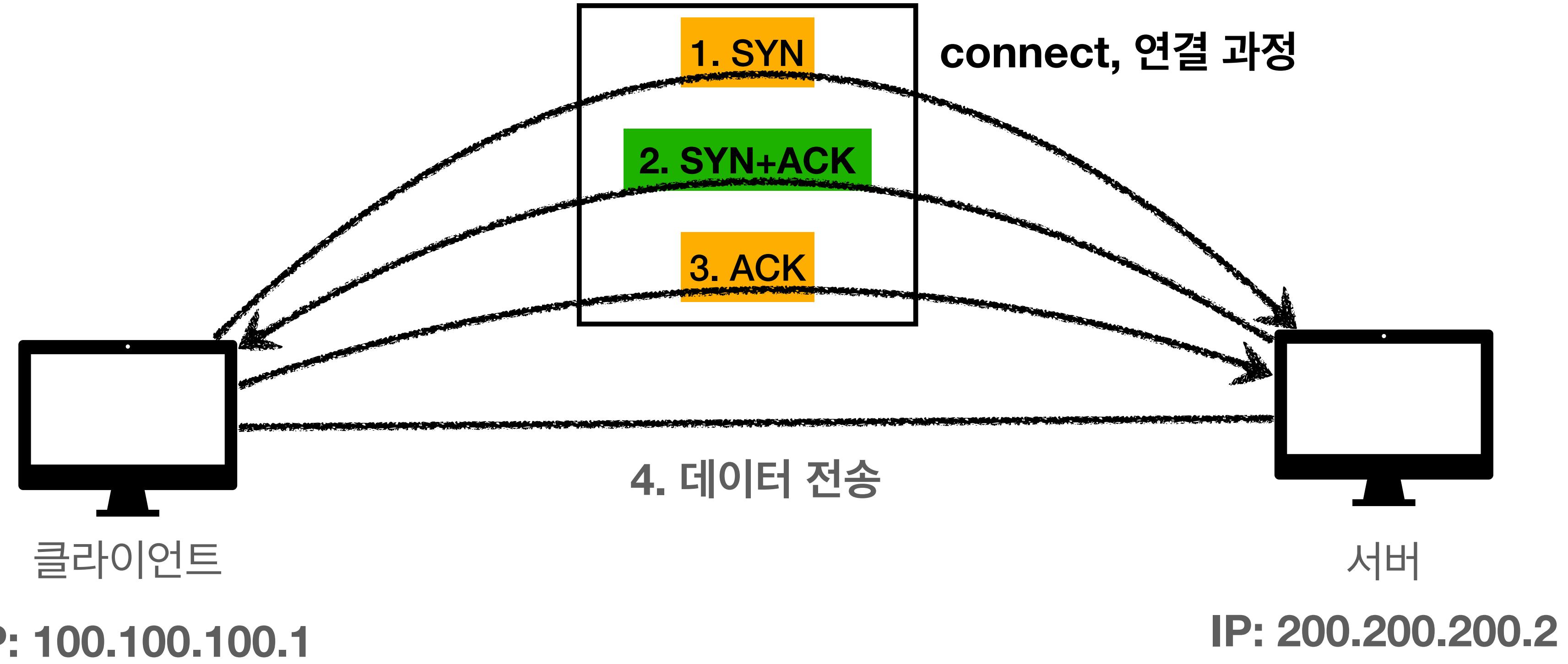
TCP 세그먼트

TCP 특징

전송 제어 프로토콜(Transmission Control Protocol)

- 연결지향 - TCP 3 way handshake (가상 연결)
- 데이터 전달 보증
- 순서 보장
- 신뢰할 수 있는 프로토콜
- 현재는 대부분 TCP 사용

TCP 3 way handshake

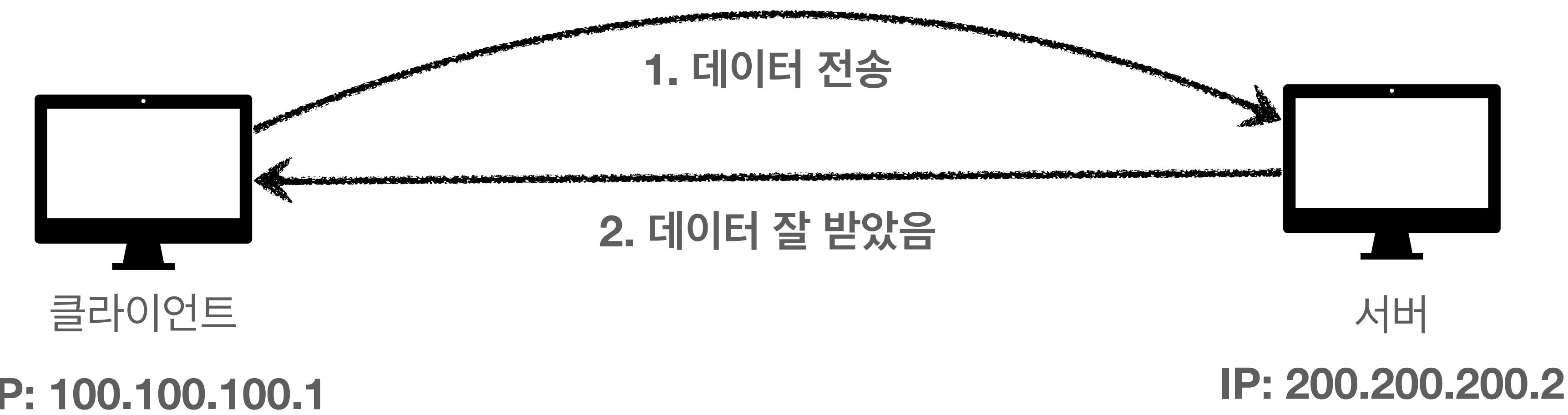


SYN: 접속 요청

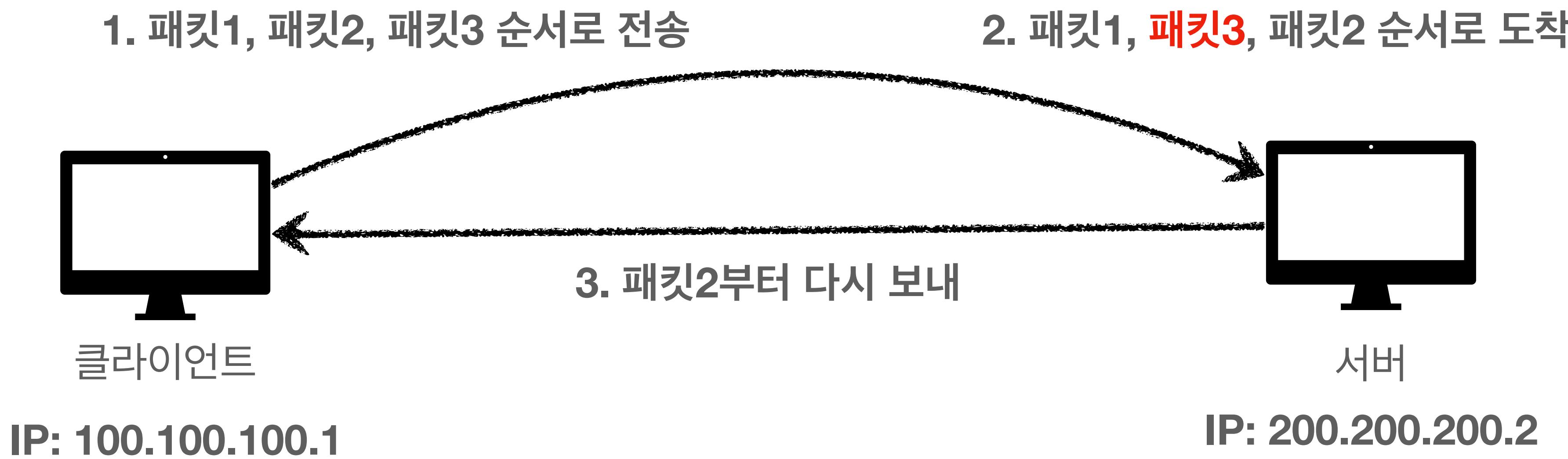
ACK: 요청 수락

참고: 3. ACK와 함께 데이터 전송 가능

데이터 전달 보증



순서 보장



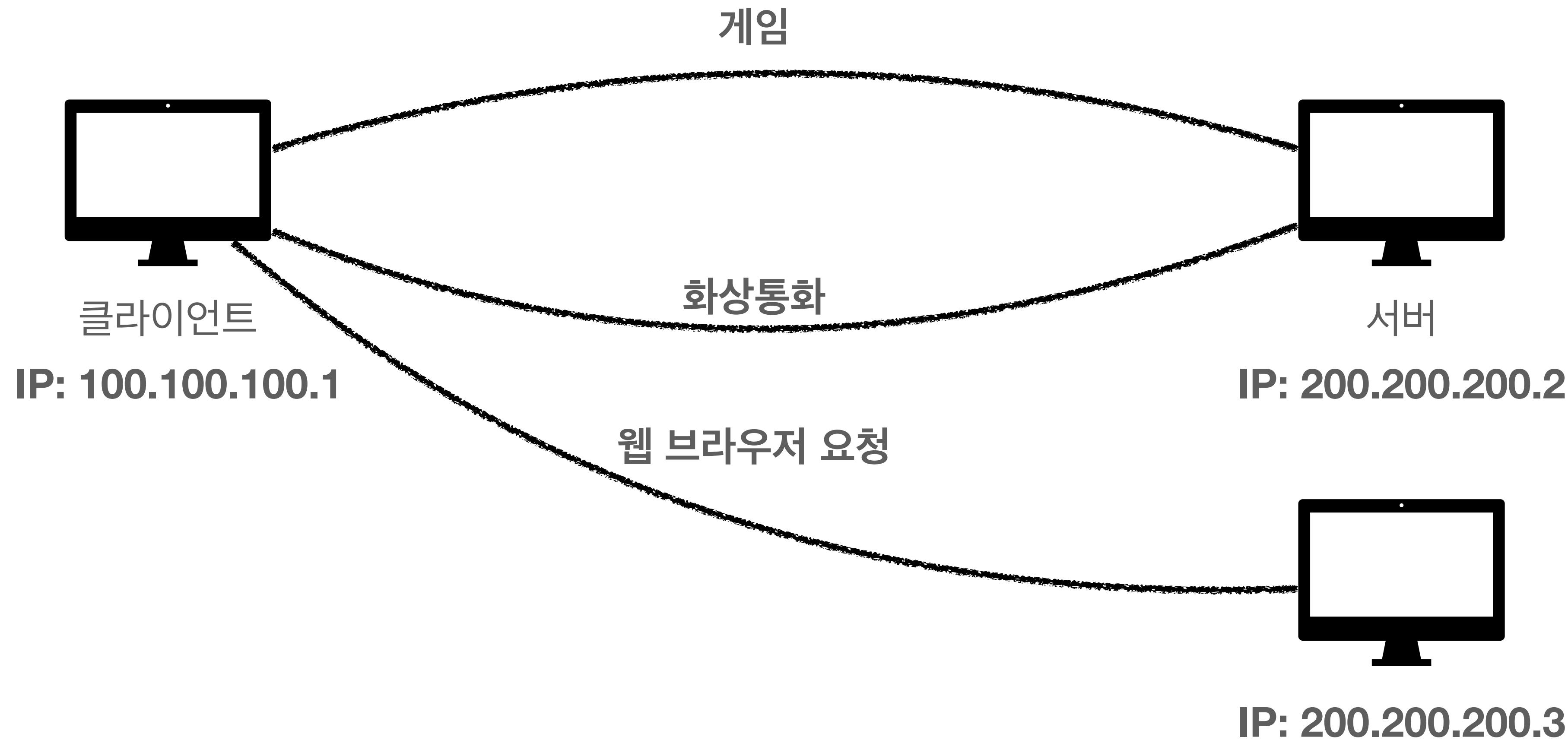
UDP 특징

사용자 데이터그램 프로토콜(User Datagram Protocol)

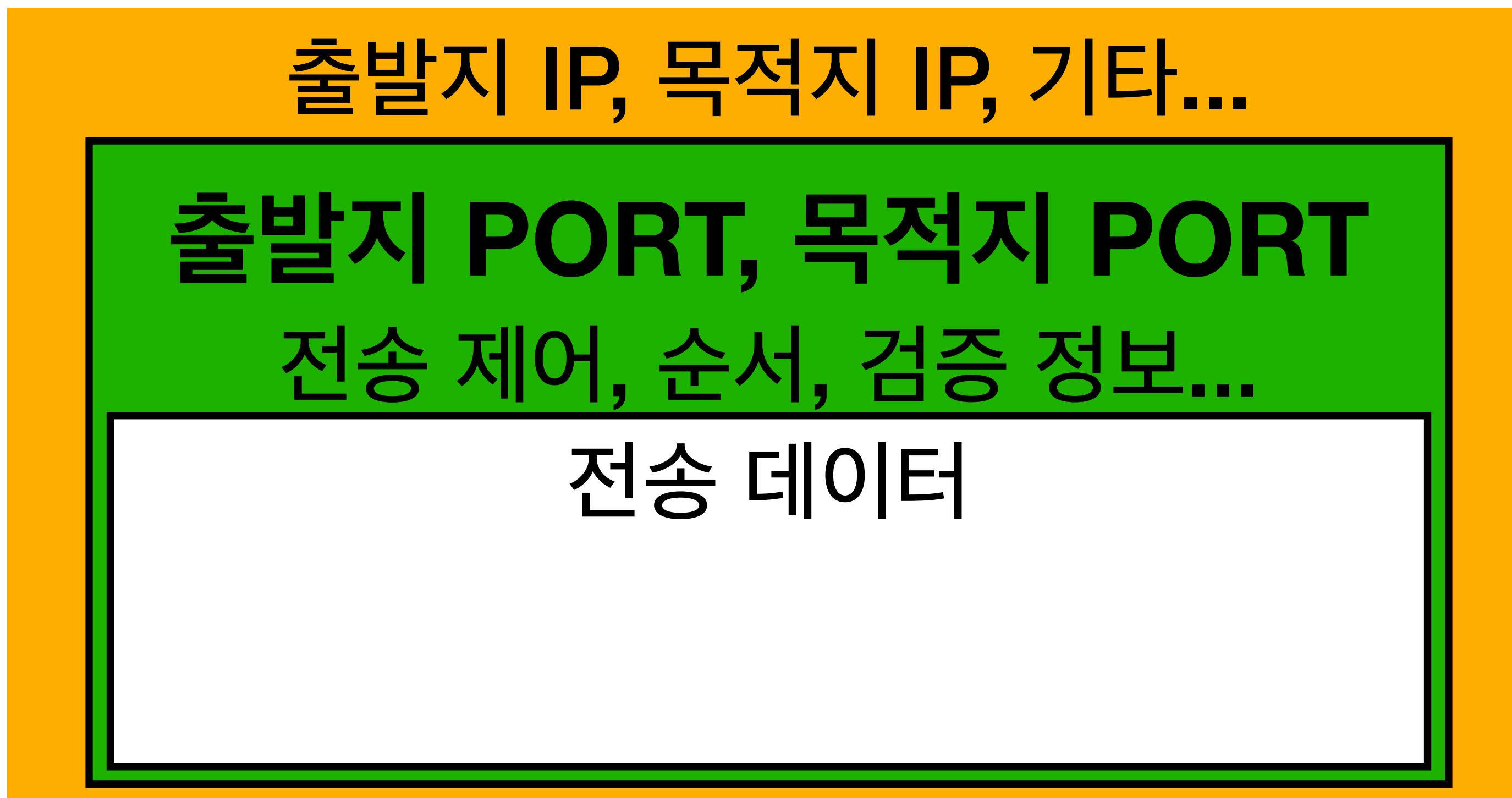
- 하양 도화지에 비유(기능이 거의 없음)
- 연결지향 X - TCP 3 way handshake X
- 데이터 전달 보증 X
- 순서 보장 X
- 데이터 전달 및 순서가 보장되지 않지만, 단순하고 빠름
- 정리
 - IP와 거의 같다. +PORT +체크섬 정도만 추가
 - 애플리케이션에서 추가 작업 필요

PORT

한번에 둘 이상 연결해야 하면?



TCP/IP 패킷 정보



IP 패킷



TCP 세그먼트

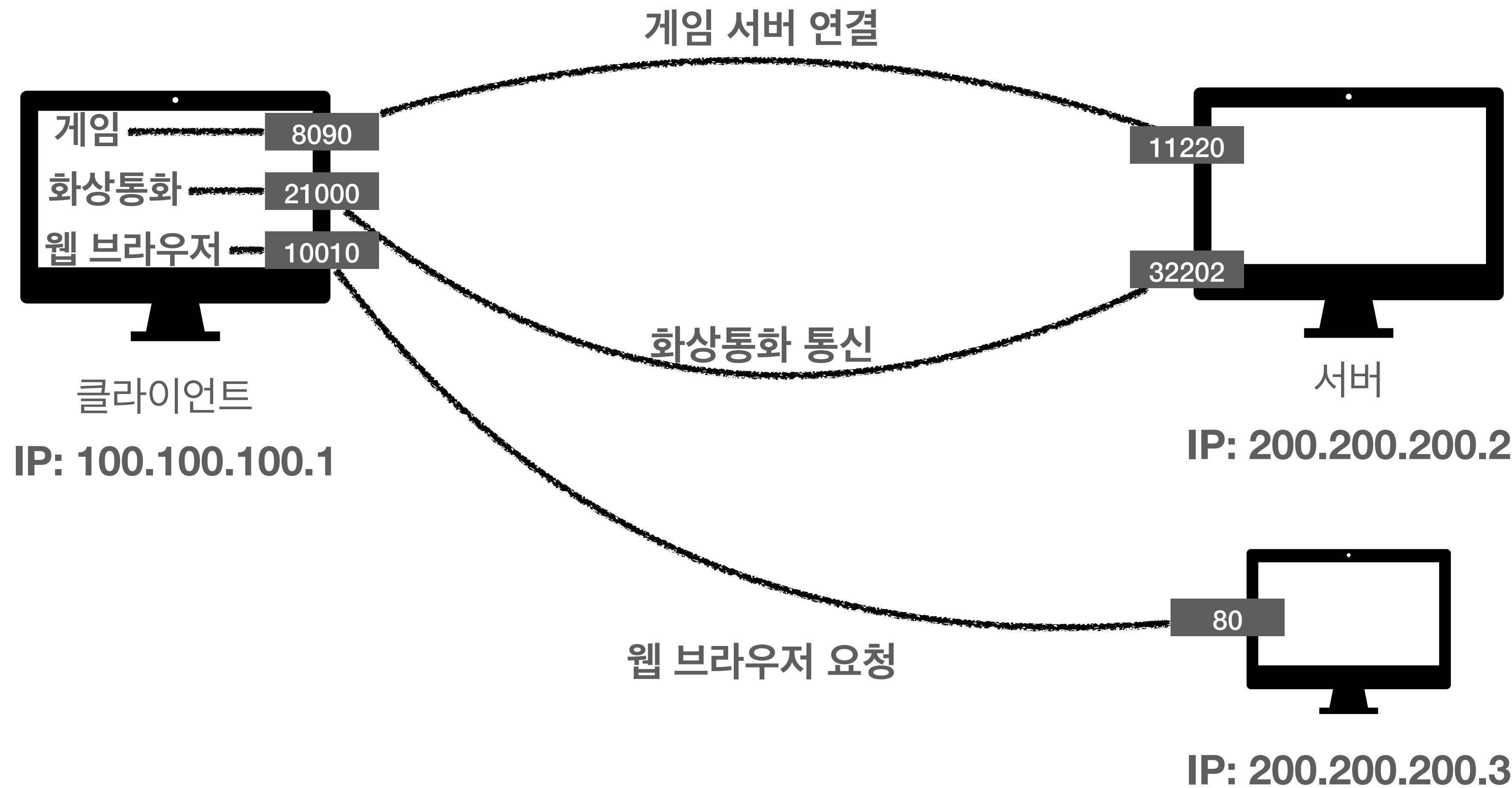
패킷 정보

출발지 IP, PORT
목적지 IP, PORT
전송 데이터

...

TCP/IP 패킷

PORT - 같은 IP 내에서 프로세스 구분

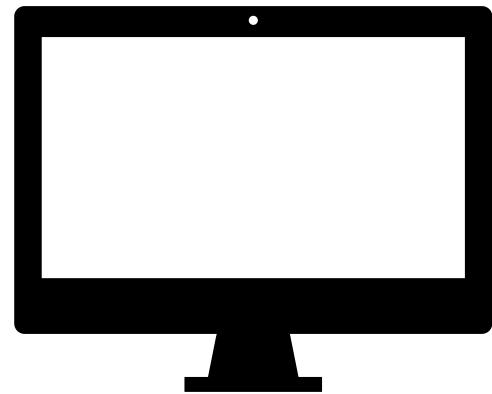


PORT

- 0 ~ 65535 할당 가능
- 0 ~ 1023: 잘 알려진 포트, 사용하지 않는 것이 좋음
 - FTP - 20, 21
 - TELNET - 23
 - HTTP - 80
 - HTTPS - 443

DNS

IP는 기억하기 어렵다.

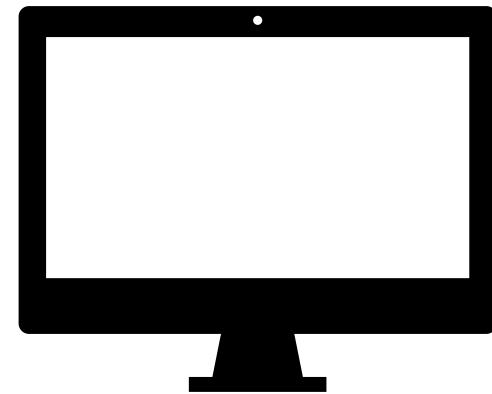


?

200.200.200.2???

클라이언트

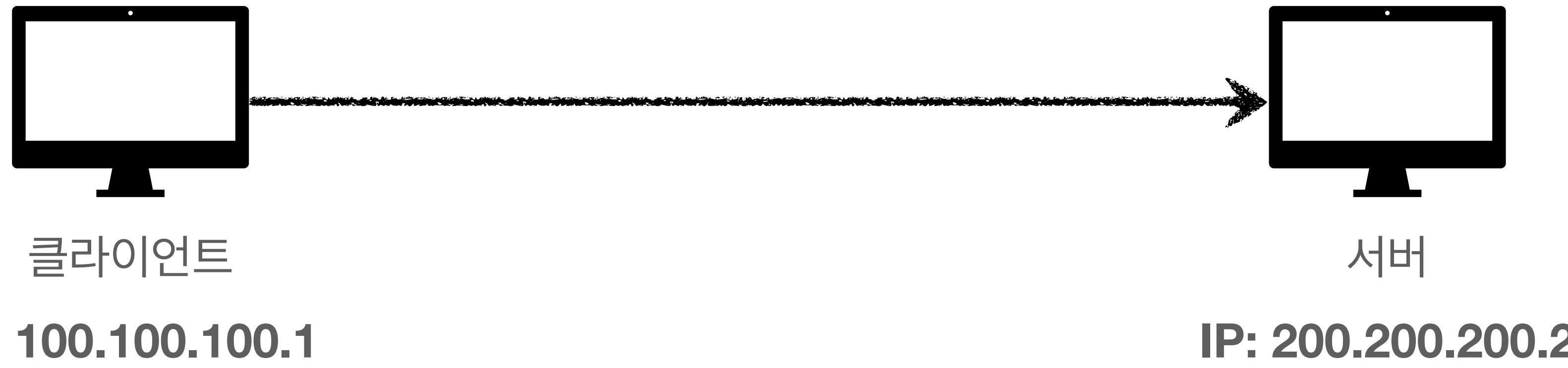
IP: 100.100.100.1



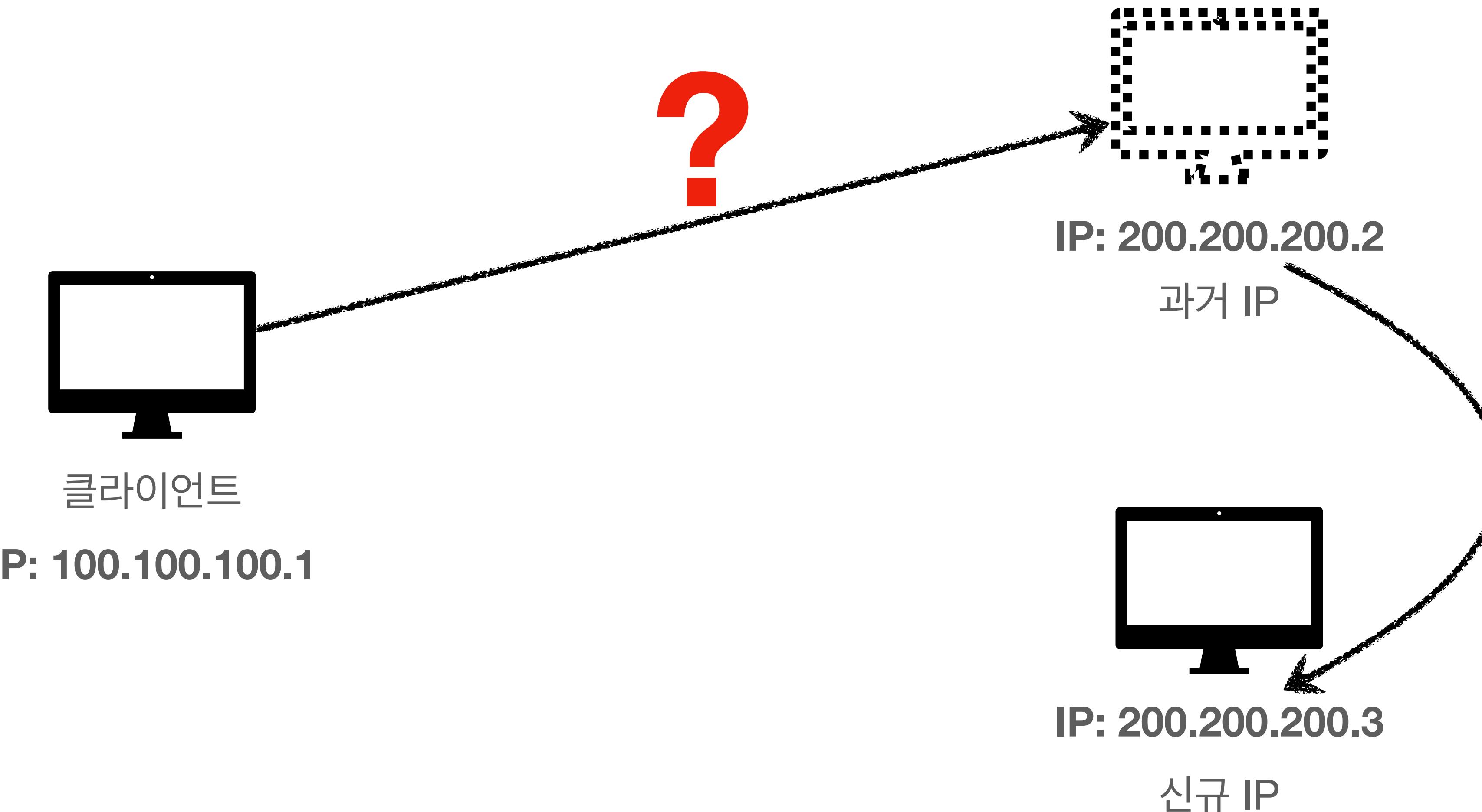
서버

IP: 200.200.200.2

IP는 변경될 수 있다.



IP는 변경될 수 있다.



DNS

도메인 네임 시스템(Domain Name System)

- 전화번호부
- 도메인 명을 IP 주소로 변환

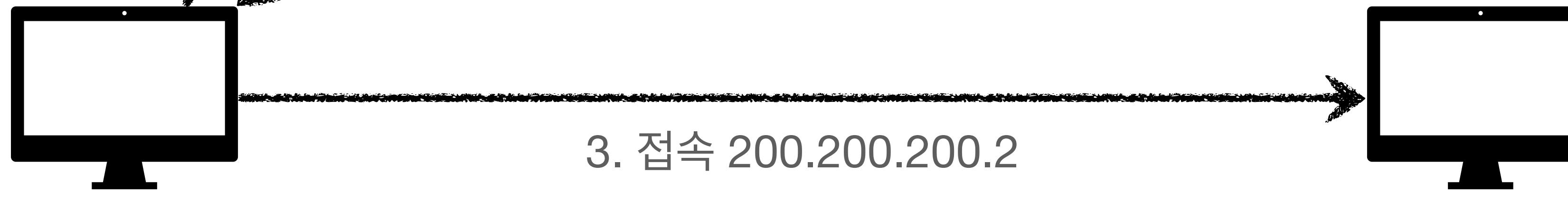
DNS 사용

DNS 서버

도메인 명	IP
<u>google.com</u>	200.200.200.2
<u>aaa.com</u>	210.210.210.3

1. 도메인 명 **google.com**

2. 응답: 200.200.200.2



3. 접속 200.200.200.2

IP: 100.100.100.1

IP: 200.200.200.2

인터넷 네트워크 정리

- 인터넷 통신
- IP(Internet Protocol)
- TCP, UDP
- PORT
- DNS

[URI와 웹 브라우저 요청 흐름]

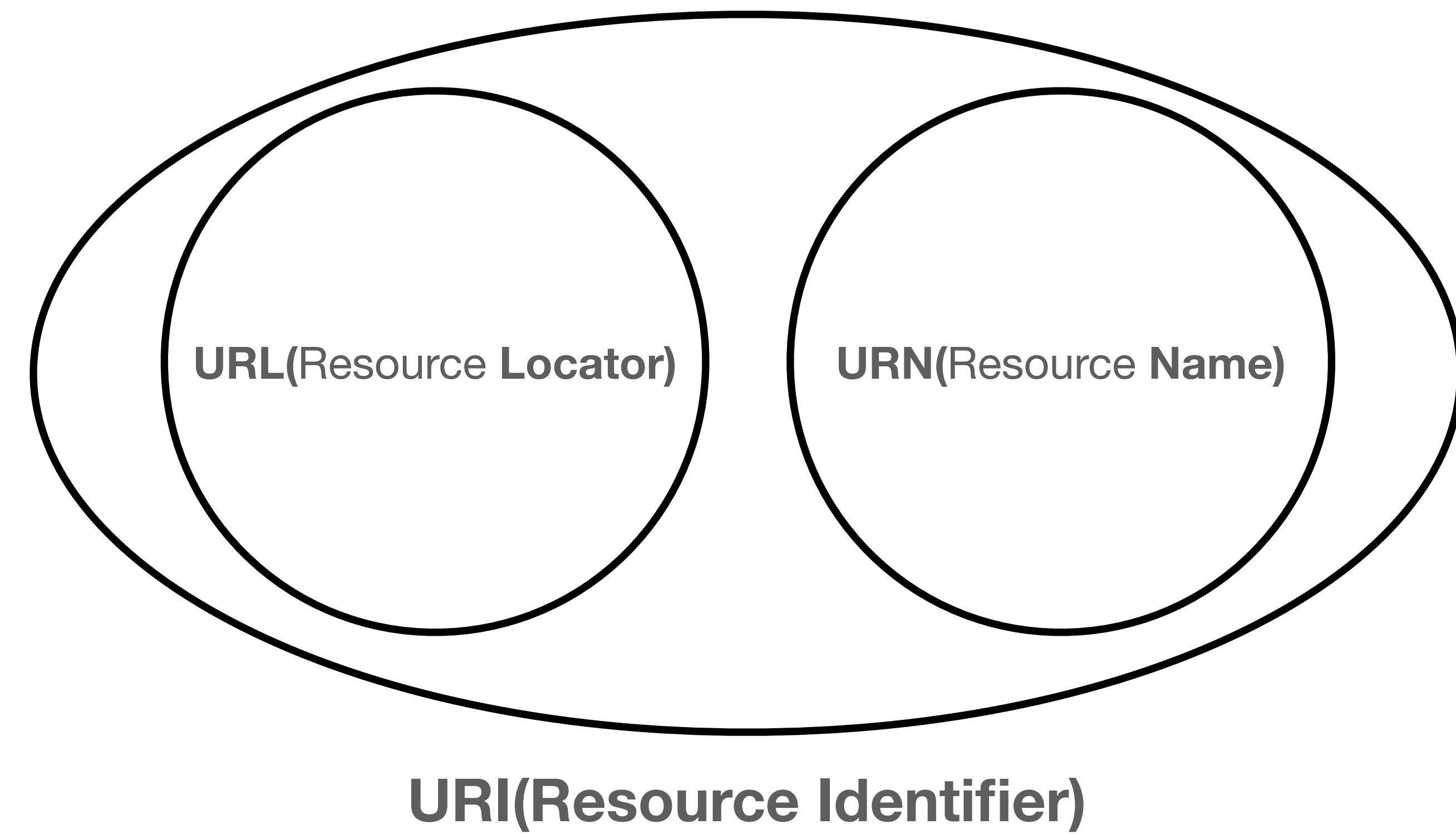
- URI
- 웹 브라우저 요청 흐름

URI(Uniform Resource Identifier)

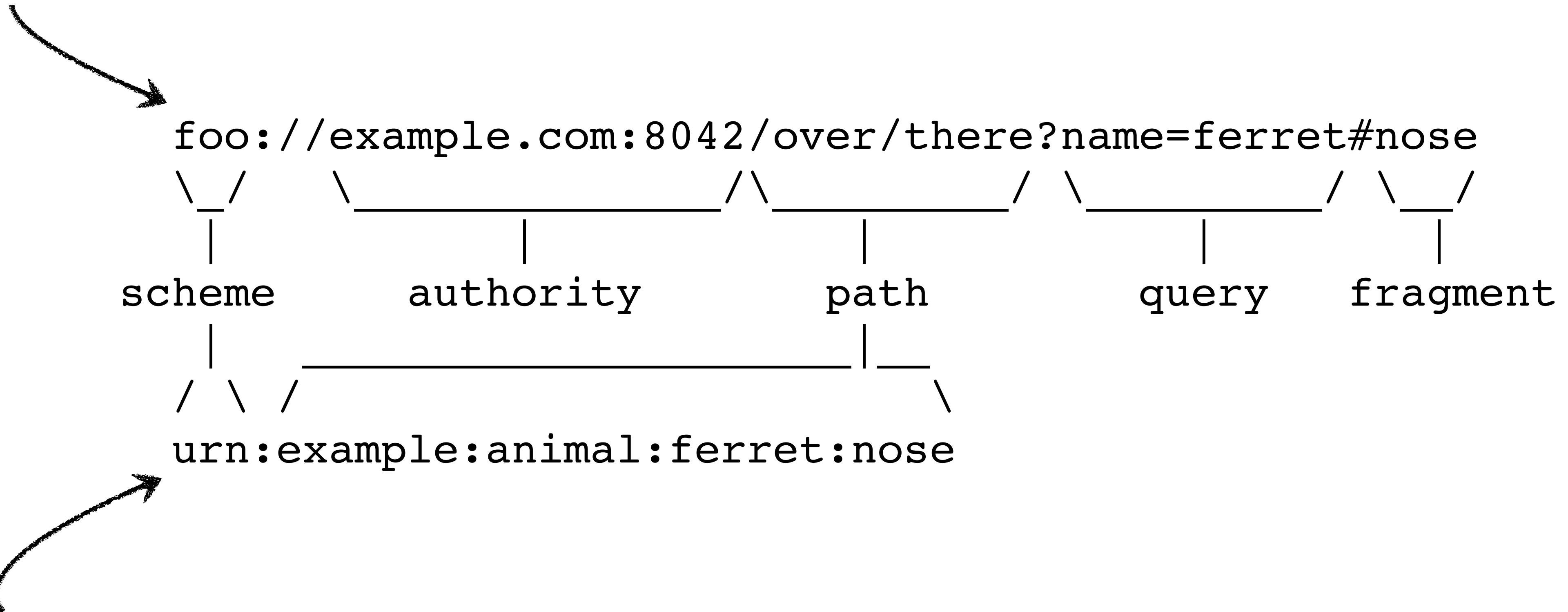
URI? URL? URN?

"URI는 로케이터(locator), 이름(name) 또는 둘
다 추가로 분류될 수 있다"

<https://www.ietf.org/rfc/rfc3986.txt> - 1.1.3. URI, URL, and URN



URL(Resource Locator)



URN(Resource Name)

URI

단어 뜻

- **Uniform:** 리소스 식별하는 통일된 방식
- **Resource:** 자원, URI로 식별할 수 있는 모든 것(제한 없음)
- **Identifier:** 다른 항목과 구분하는데 필요한 정보
- **URL:** Uniform Resource Locator
- **URN:** Uniform Resource Name

<https://www.ietf.org/rfc/rfc3986.txt>

URL, URN

단어 뜻

- URL - Locator: 리소스가 있는 위치를 지정
- URN - Name: 리소스에 이름을 부여
- 위치는 변할 수 있지만, 이름은 변하지 않는다.
- urn:isbn:8960777331 (어떤 책의 isbn URN)
- URN 이름만으로 실제 리소스를 찾을 수 있는 방법이 보편화 되지 않음
- 앞으로 URI를 URL과 같은 의미로 이야기하겠음

URL 분석

`https://www.google.com/search?q=hello&hl=ko`

URL

전체 문법

- scheme://[userinfo@]host[:port][/path][?query][#fragment]
- `https://www.google.com:443/search?q=hello&hl=ko`
- 프로토콜(https)
- 호스트명(www.google.com)
- 포트 번호(443)
- 패스(/search)
- 쿼리 파라미터(q=hello&hl=ko)

URL scheme

- **scheme://[userinfo@]host[:port][/path][?query][#fragment]**
- **https://www.google.com:443/search?q=hello&hl=ko**
- 주로 프로토콜 사용
- 프로토콜: 어떤 방식으로 자원에 접근할 것인가 하는 약속 규칙
 - 예) http, https, ftp 등등
- http는 80 포트, https는 443 포트를 주로 사용, 포트는 생략 가능
- https는 http에 보안 추가 (HTTP Secure)

URL

userinfo

- scheme://[**userinfo@**]host[:port][/path][?query][#fragment]
- <https://www.google.com:443/search?q=hello&hl=ko>
- URL에 사용자정보를 포함해서 인증
- 거의 사용하지 않음

URL

host

- scheme://[userinfo@]**host**[:port][/path][?query][#fragment]
- **https://www.google.com:443/search?q=hello&hl=ko**
- 호스트명
- 도메인명 또는 IP 주소를 직접 사용가능

URL

PORT

- scheme://[userinfo@]host[:port][/path][?query][#fragment]
- `https://www.google.com:443/search?q=hello&hl=ko`
- 포트(PORT)
- 접속 포트
- 일반적으로 생략, 생략시 http는 80, https는 443

URL

path

- scheme://[userinfo@]host[:port][**/path**][?query][#fragment]
- <https://www.google.com:443/search?q=hello&hl=ko>
- 리소스 경로(path), 계층적 구조
- 예)
 - /home/file1.jpg
 - /members
 - /members/100, /items/iphone12

URL

query

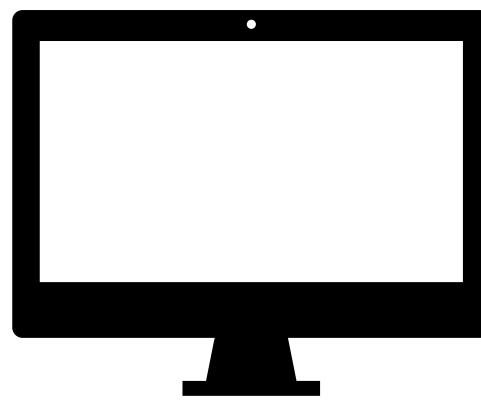
- scheme://[userinfo@]host[:port][/path]**[?query]**[#fragment]
- <https://www.google.com:443/search?q=hello&hl=ko>
- key=value 형태
- ?로 시작, &로 추가 가능 ?keyA=valueA&keyB=valueB
- query parameter, query string 등으로 불림, 웹서버에 제공하는 파라미터, 문자 형태

URL fragment

- scheme://[userinfo@]host[:port][/path][?query][#fragment]
- <https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started-introducing-spring-boot>
- fragment
- html 내부 북마크 등에 사용
- 서버에 전송하는 정보 아님

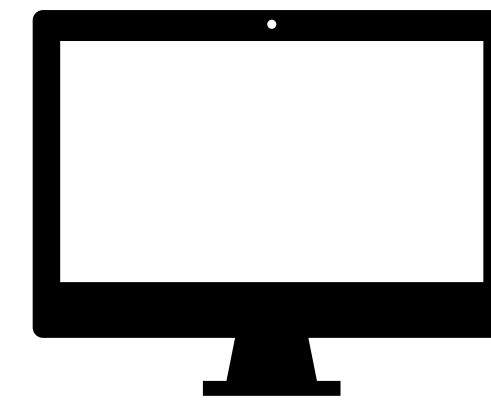
웹 브라우저 요청 흐름

https://www.google.com/search?q=hello&hl=ko



웹 브라우저

IP: 100.100.100.1



구글 서버

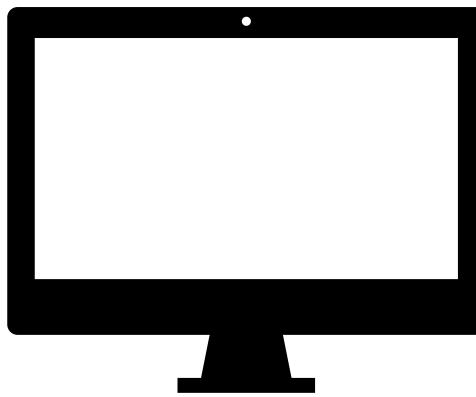
IP: 200.200.200.2

https://www.google.com:443/search?q=hello&hl=ko

DNS 조회

HTTPS PORT 생략, 443

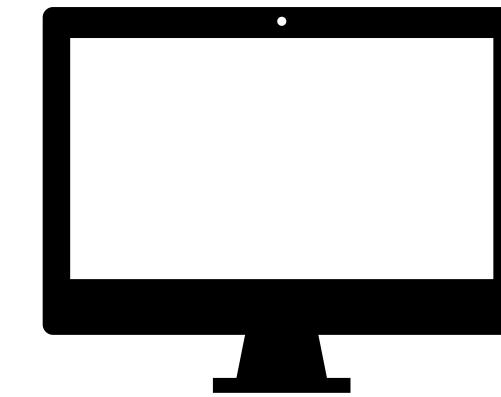
IP: 200.200.200.2



웹 브라우저

IP: 100.100.100.1

HTTP 요청 메시지 생성



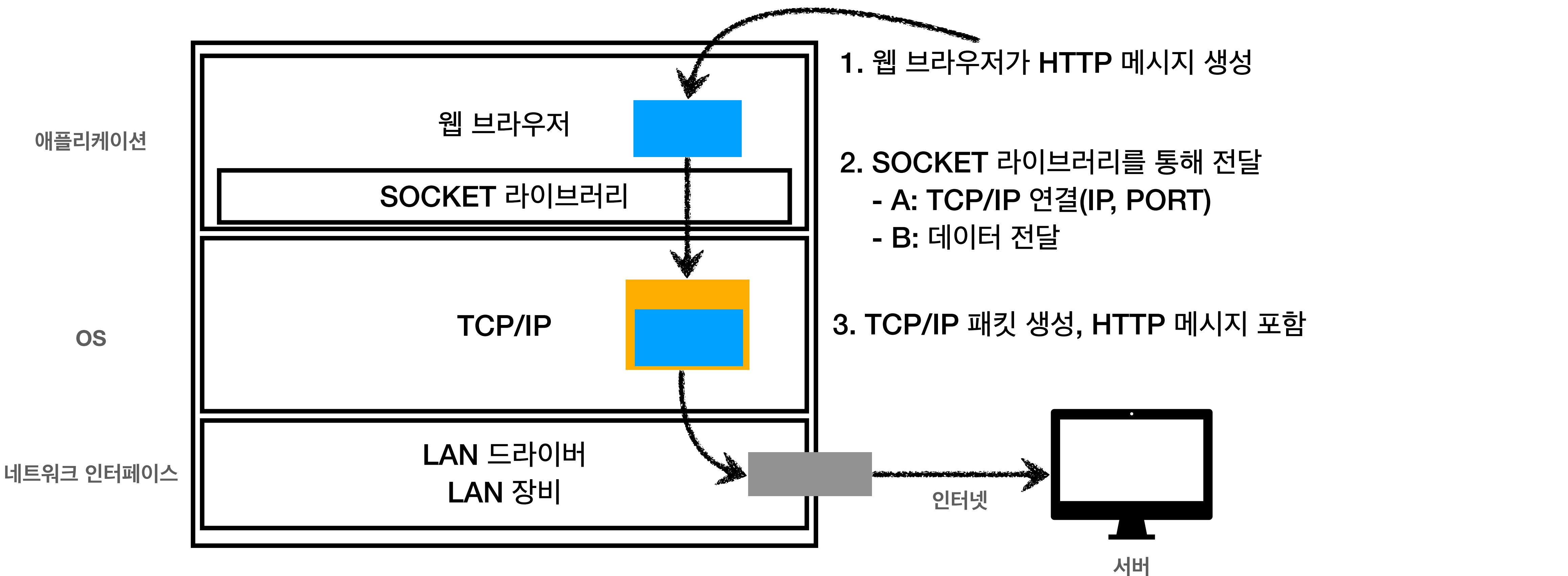
구글 서버

IP: 200.200.200.2

```
GET /search?q=hello&hl=ko HTTP/1.1
Host: www.google.com
```

HTTP 요청 메시지

HTTP 메시지 전송



패킷 생성

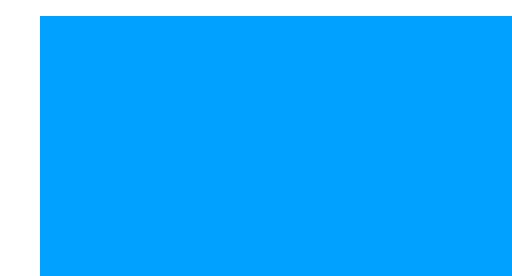
출발지 IP, PORT
목적지 IP, PORT
전송 데이터
...

TCP/IP 패킷

패킷 생성



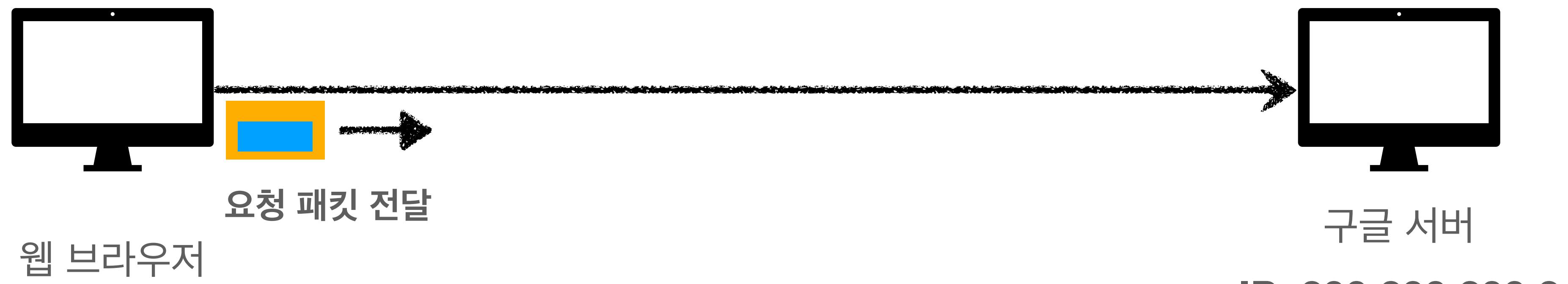
TCP/IP 패킷



HTTP 메시지

전송 데이터
HTTP 메시지

`https://www.google.com/search?q=hello&hl=ko`



TCP/IP 패킷

HTTP 메시지

`https://www.google.com/search?q=hello&hl=ko`



HTTP/1.1 200 OK

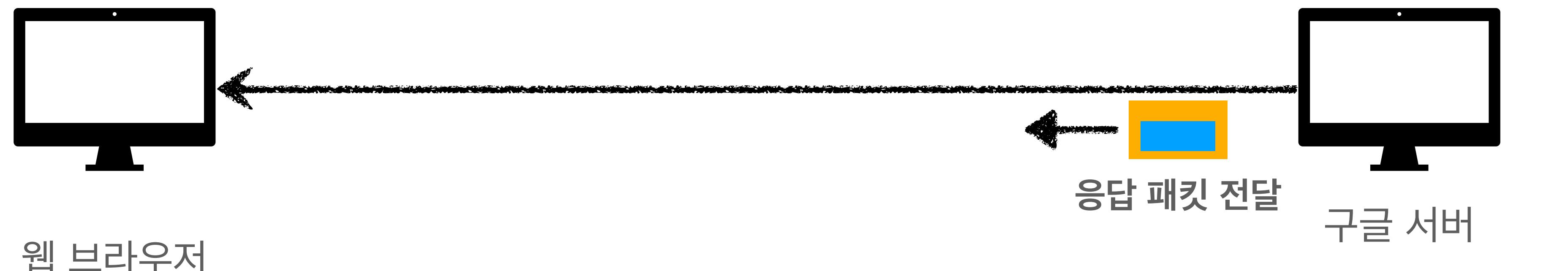
Content-Type: text/html; charset=UTF-8

Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

HTTP 응답 메시지

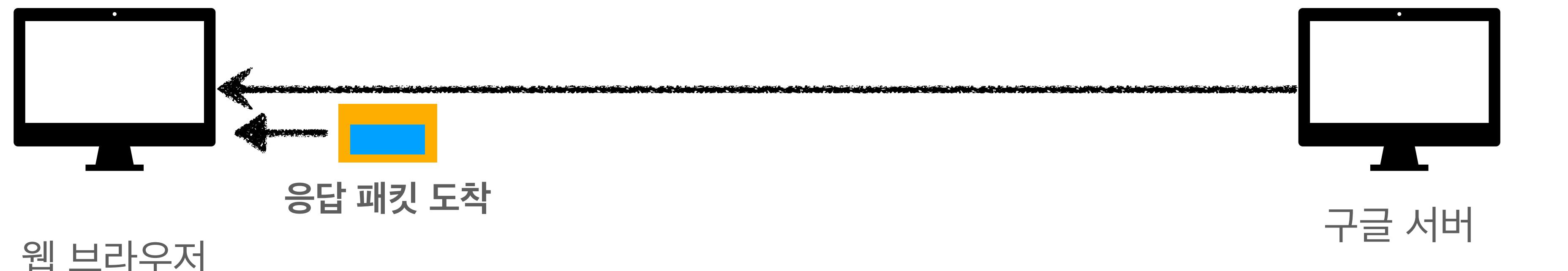
`https://www.google.com/search?q=hello&hl=ko`



TCP/IP 패킷

HTTP 메시지

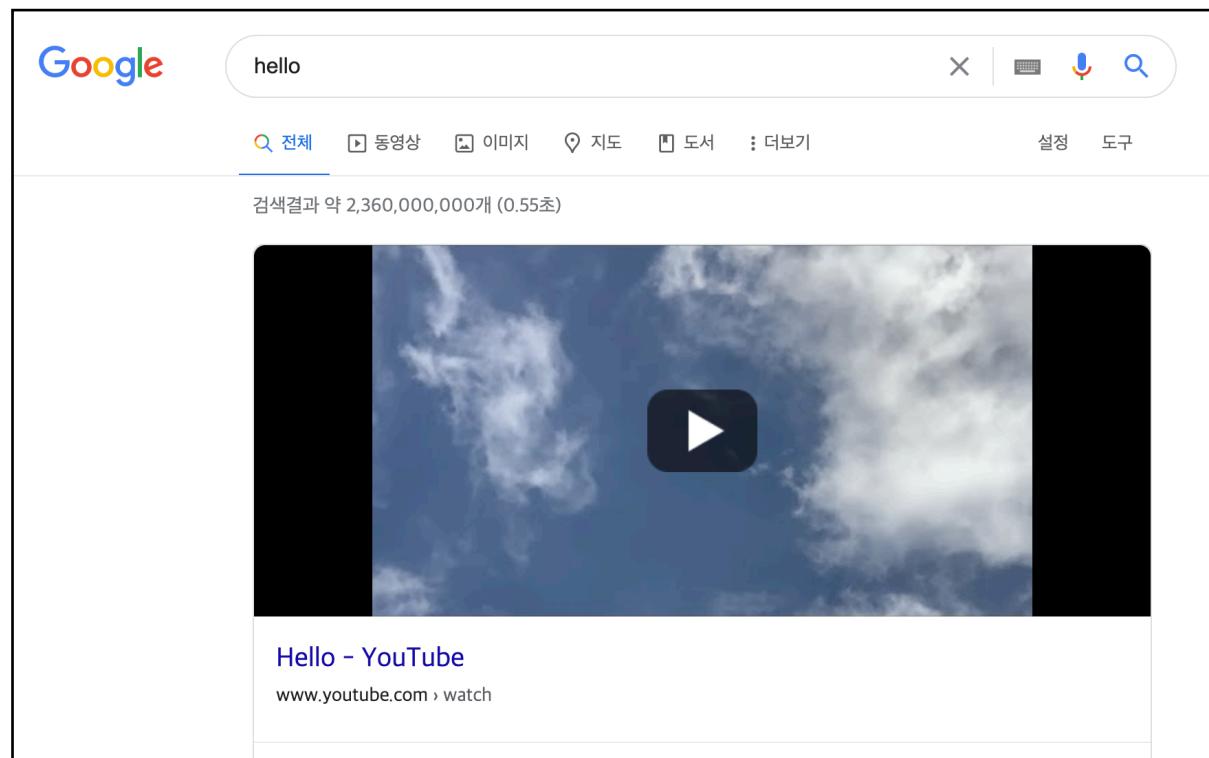
`https://www.google.com/search?q=hello&hl=ko`



TCP/IP 패킷

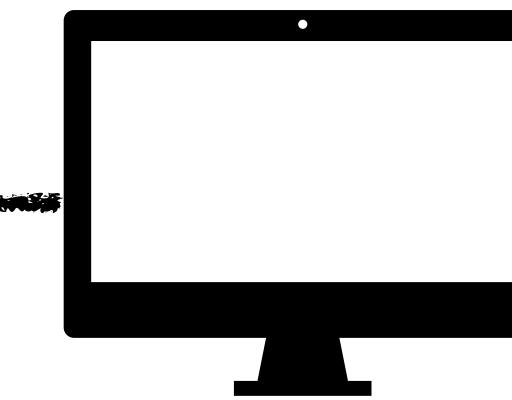
HTTP 메시지

https://www.google.com/search?q=hello&hl=ko



웹 브라우저 HTML 렌더링

IP: 100.100.100.1

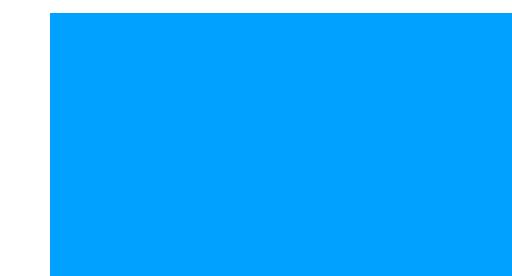


구글 서버

IP: 200.200.200.2



TCP/IP 패킷



HTTP 메시지

[HTTP]

- 모든 것이 HTTP
- 클라이언트 서버 구조
- Stateful, Stateless
- 비 연결성(connectionless)
- HTTP 메시지

HTTP
HyperText Transfer Protocol

모든 것이 HTTP

HTTP 메시지에 모든 것을 전송

- HTML, TEXT
- IMAGE, 음성, 영상, 파일
- JSON, XML (API)
- 거의 모든 형태의 데이터 전송 가능
- 서버간에 데이터를 주고 받을 때도 대부분 HTTP 사용
- 지금은 HTTP 시대!

HTTP 역사

- HTTP/0.9 1991년: GET 메서드만 지원, HTTP 헤더X
- HTTP/1.0 1996년: 메서드, 헤더 추가
- **HTTP/1.1 1997년: 가장 많이 사용, 우리에게 가장 중요한 버전**
 - RFC2068 (1997) -> RFC2616 (1999) -> RFC7230~7235 (2014)
- HTTP/2 2015년: 성능 개선
- HTTP/3 진행중: TCP 대신에 UDP 사용, 성능 개선

기반 프로토콜

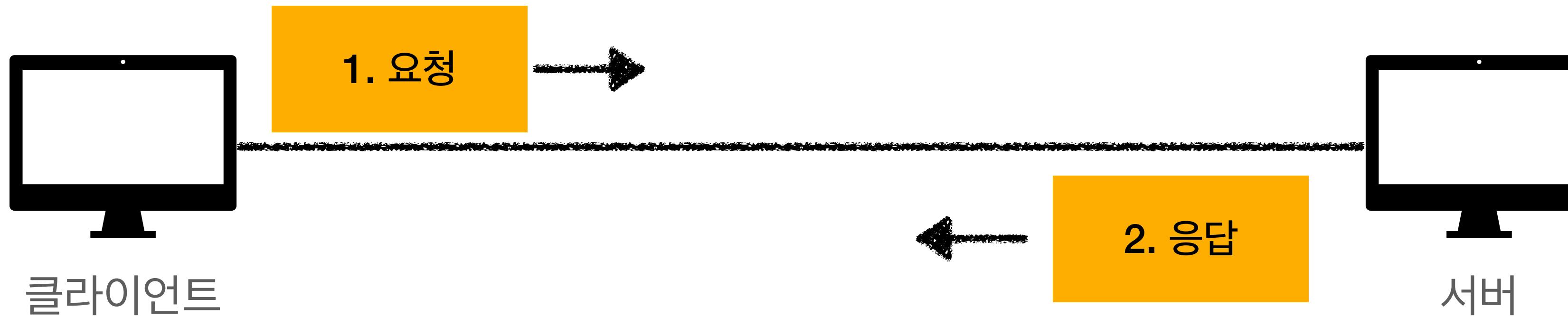
- **TCP:** HTTP/1.1, HTTP/2
- **UDP:** HTTP/3
- 현재 HTTP/1.1 주로 사용
 - HTTP/2, HTTP/3 도 점점 증가

HTTP 특징

- 클라이언트 서버 구조
- 무상태 프로토콜(스테이스리스), 비연결성
- HTTP 메시지
- 단순함, 확장 가능

클라이언트 서버 구조

- Request Response 구조
- 클라이언트는 서버에 요청을 보내고, 응답을 대기
- 서버가 요청에 대한 결과를 만들어서 응답



무상태 프로토콜

스테이스리스(Stateless)

- 서버가 클라이언트의 상태를 보존X
- 장점: 서버 확장성 높음(스케일 아웃)
- 단점: 클라이언트가 추가 데이터 전송

Stateful, Stateless 차이

상태 유지 - Stateful

- 고객: 이 노트북 얼마인가요?
- 점원: 100만원입니다.
- 고객: **2개** 구매하겠습니다.
- 점원: 200만원입니다. **신용카드, 현금** 중에 어떤 걸로 구매 하시겠어요?
- 고객: 신용카드로 구매하겠습니다.
- 점원: 200만원 결제 완료되었습니다.

Stateful, Stateless 차이

상태 유지 - Stateful, 점원이 중간에 바뀌면?

- 고객: 이 노트북 얼마인가요?
- 점원A: 100만원 입니다.
- 고객: **2개** 구매하겠습니다.
- 점원B: ? 무엇을 2개 구매하시겠어요?
- 고객: 신용카드로 구매하겠습니다.
- 점원C: ? 무슨 제품을 몇 개 신용카드로 구매하시겠어요?

Stateful, Stateless 차이

상태 유지 - Stateful, 정리

- 고객: 이 노트북 얼마인가요?
- 점원: 100만원 입니다. (노트북 상태 유지)
- 고객: 2개 구매하겠습니다.
- 점원: 200만원 입니다. 신용카드, 현금중에 어떤 걸로 구매 하시겠어요?
(노트북, 2개 상태 유지)
- 고객: 신용카드로 구매하겠습니다.
- 점원: 200만원 결제 완료되었습니다. (노트북, 2개, 신용카드 상태 유지)

Stateful, Stateless 차이

무상태 - Stateless

- 고객: 이 노트북 얼마인가요?
- 점원: 100만원입니다.
- 고객: 노트북 2개 구매하겠습니다.
- 점원: 노트북 2개는 200만원입니다. 신용카드, 현금중에 어떤 걸로 구매 하시겠어요?
- 고객: 노트북 2개를 신용카드로 구매하겠습니다.
- 점원: 200만원 결제 완료되었습니다.

Stateful, Stateless 차이

무상태 - Stateless, 점원이 중간에 바뀌면?

- 고객: 이 노트북 얼마인가요?
- 점원A: 100만원입니다.
- 고객: 노트북 2개 구매하겠습니다.
- 점원B: 노트북 2개는 200만원입니다. 신용카드, 현금중에 어떤 걸로 구매 하시겠어요?
- 고객: 노트북 2개를 신용카드로 구매하겠습니다.
- 점원C: 200만원 결제 완료되었습니다.

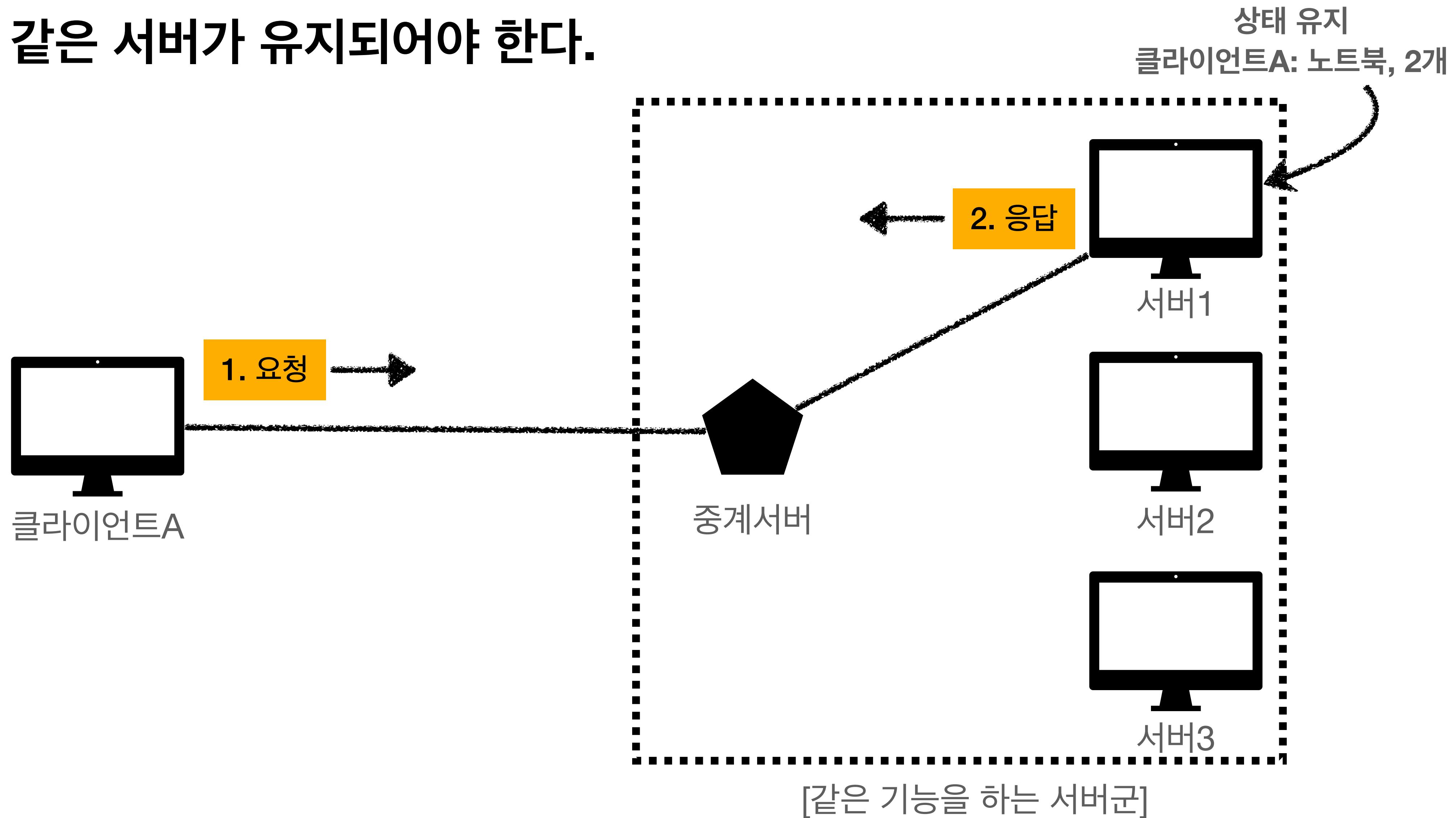
Stateful, Stateless 차이

정리

- **상태 유지:** 중간에 다른 점원으로 바뀌면 안된다.
(중간에 다른 점원으로 바뀔 때 상태 정보를 다른 점원에게 미리 알려줘야 한다.)
- **무상태:** 중간에 다른 점원으로 바뀌어도 된다.
 - 갑자기 고객이 증가해도 점원을 대거 투입할 수 있다.
 - 갑자기 클라이언트 요청이 증가해도 서버를 대거 투입할 수 있다.
- 무상태는 응답 서버를 쉽게 바꿀 수 있다. -> **무한한 서버 증설 가능**

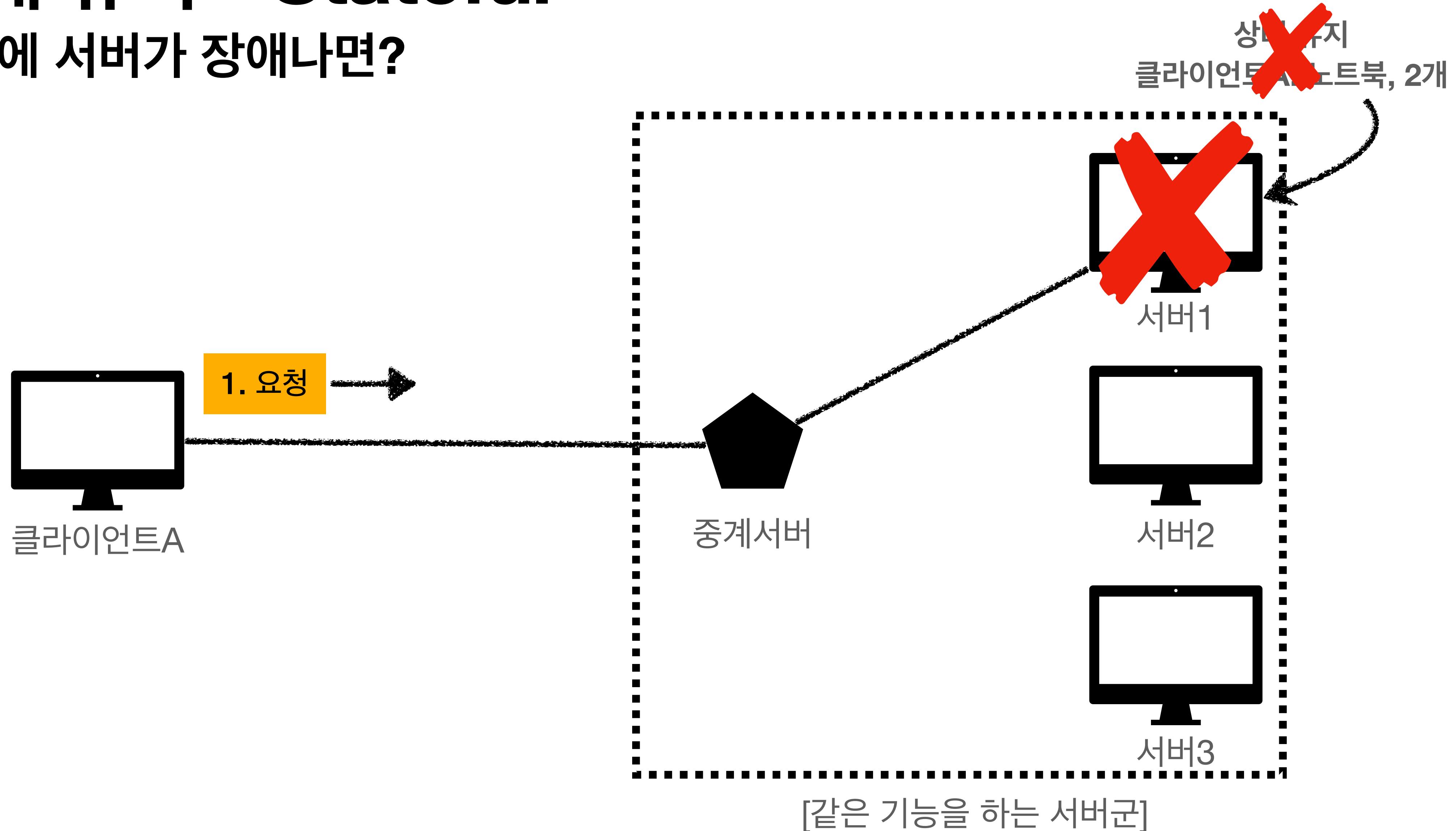
상태 유지 - Stateful

항상 같은 서버가 유지되어야 한다.



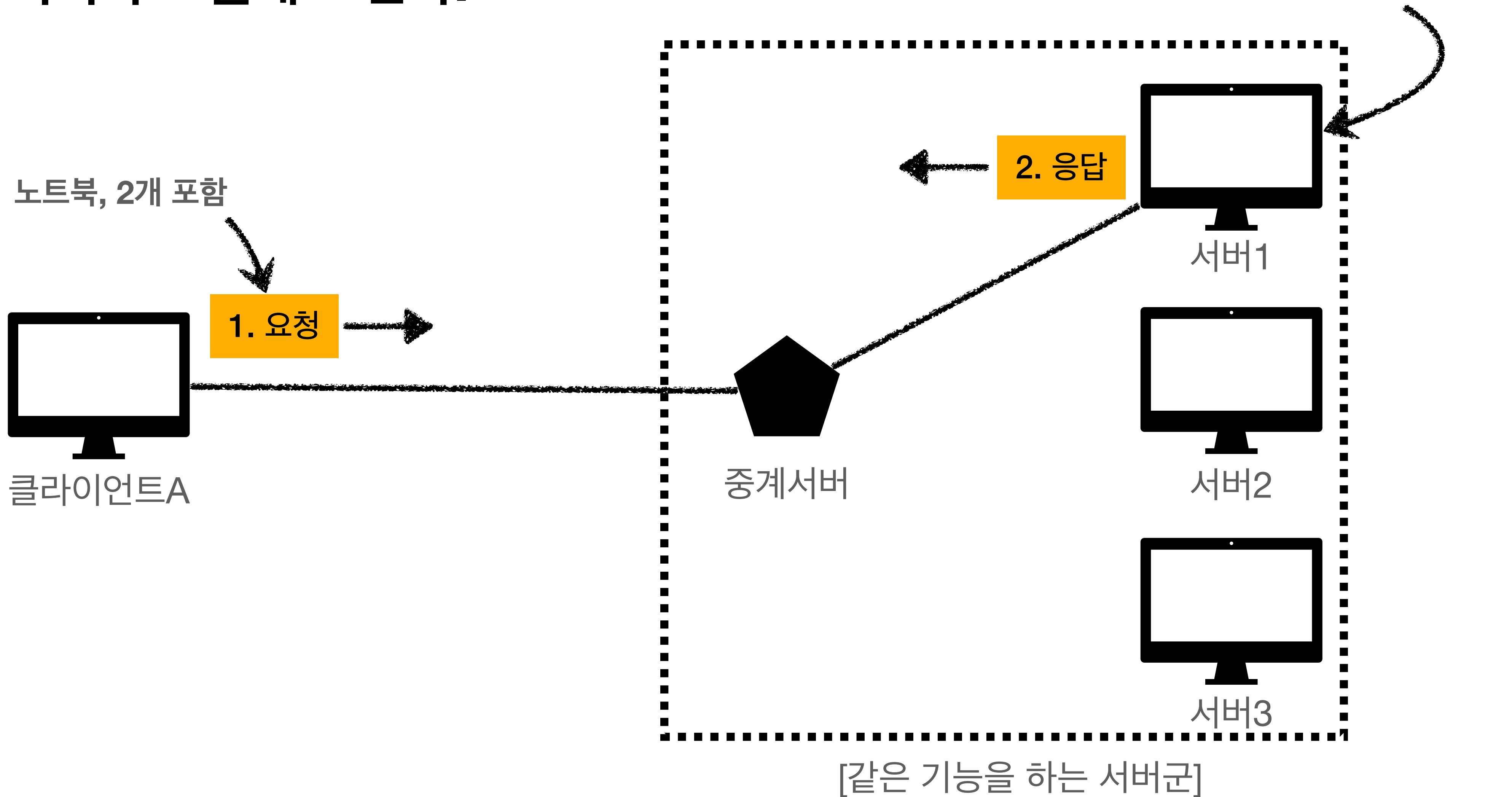
상태 유지 - Stateful

중간에 서버가 장애나면?



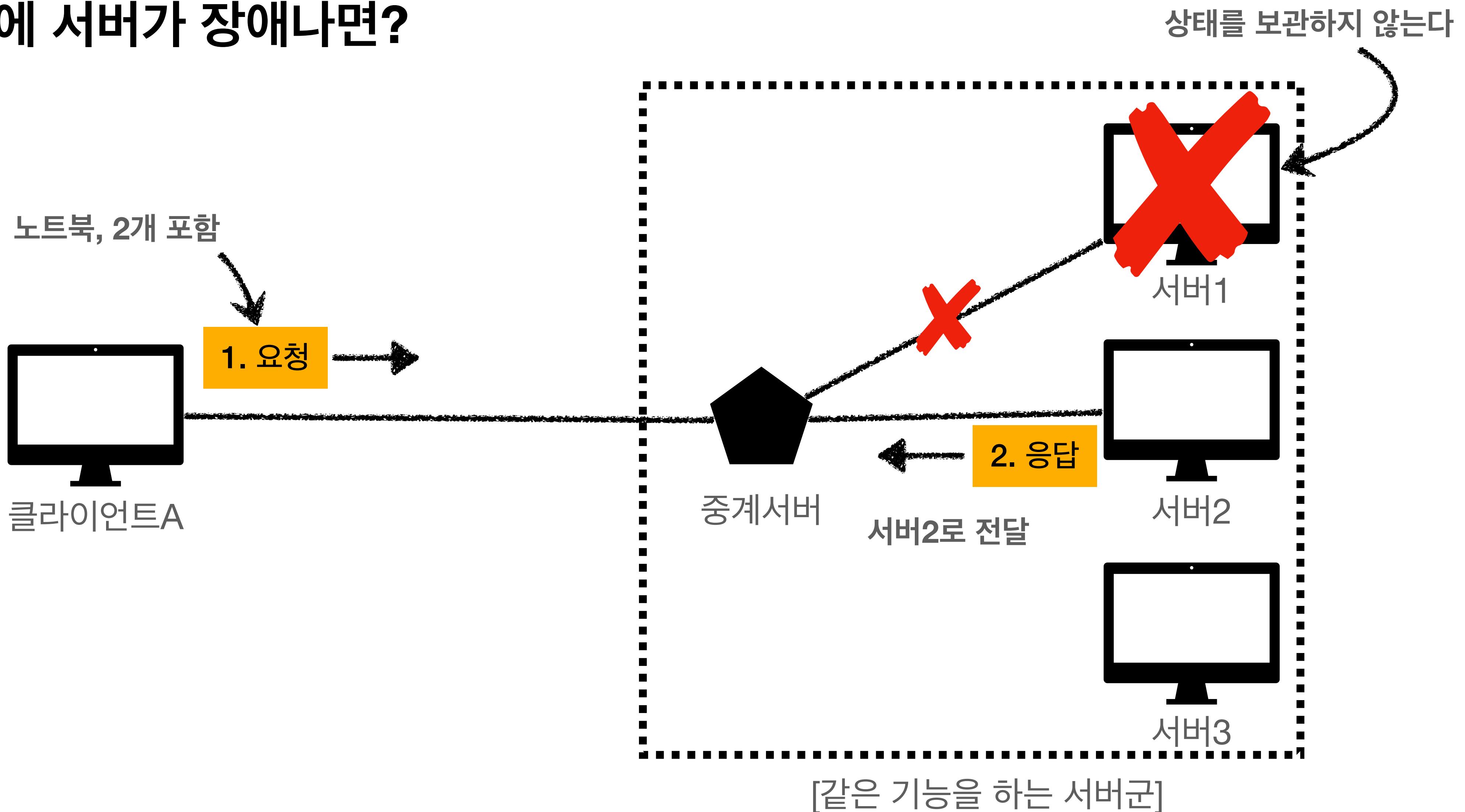
무상태 - Stateless

아무 서버나 호출해도 된다.



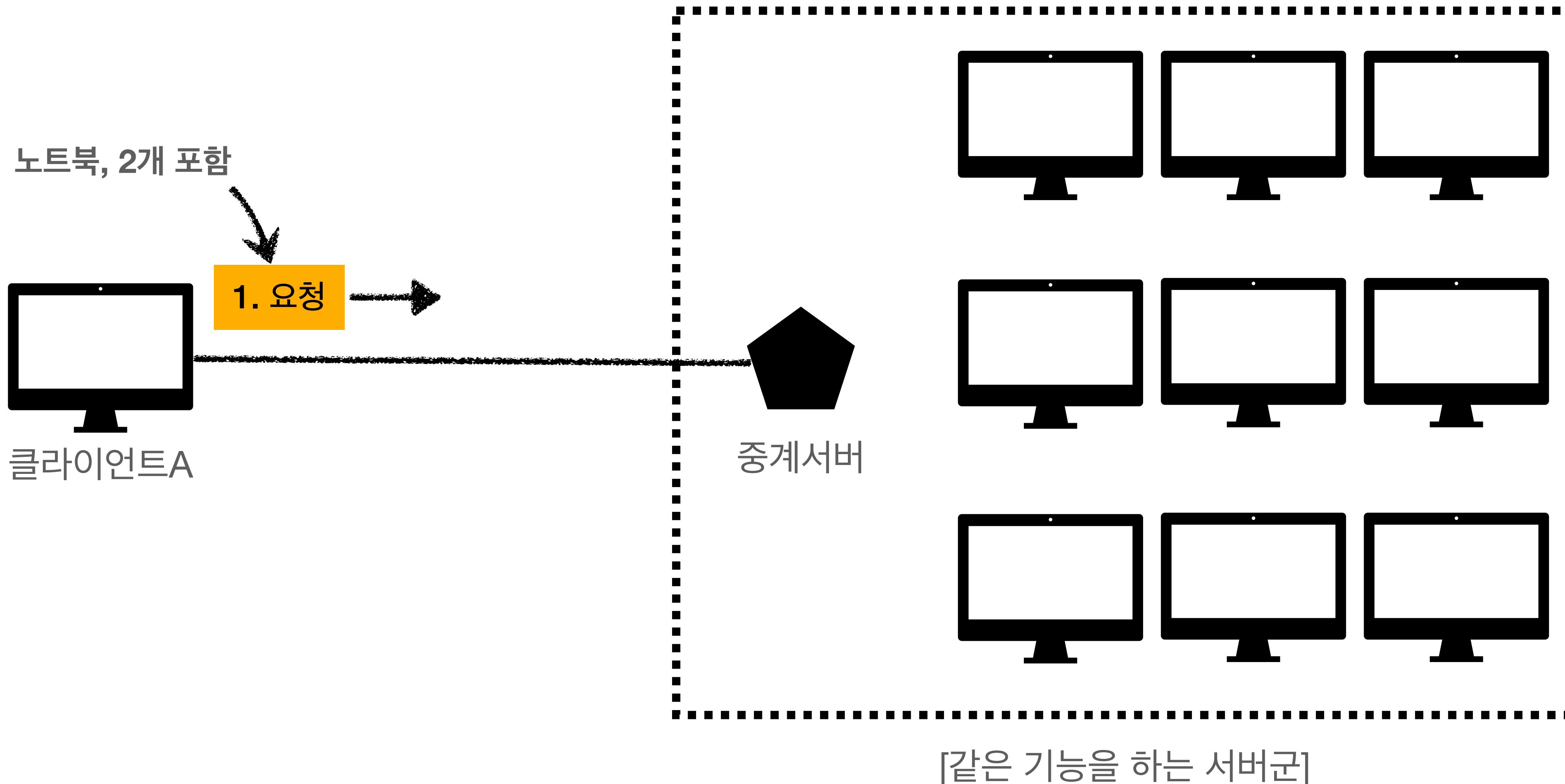
무상태 - Stateless

중간에 서버가 장애나면?



무상태 - Stateless

스케일 아웃 - 수평 확장 유리



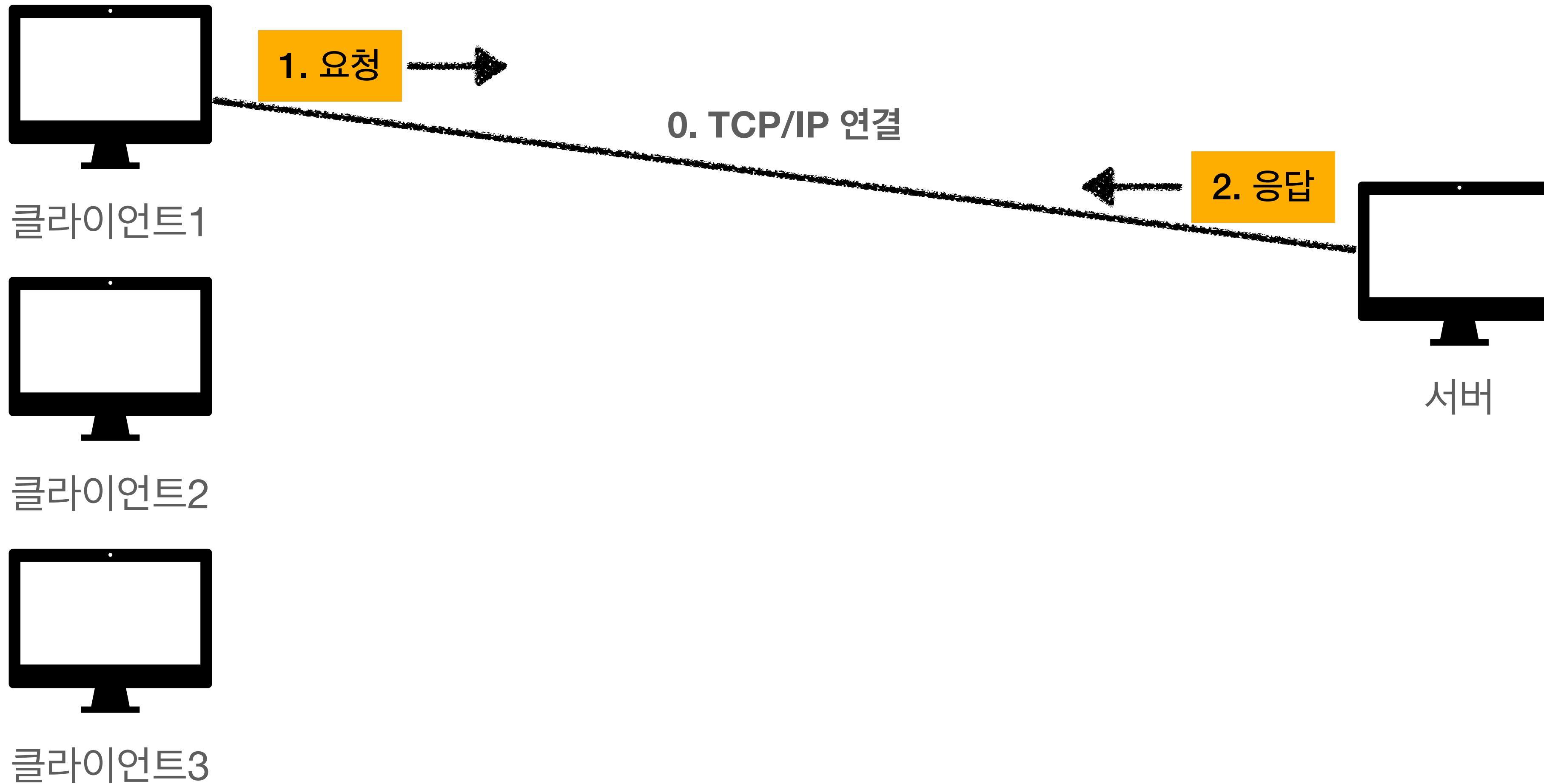
Stateless

실무 한계

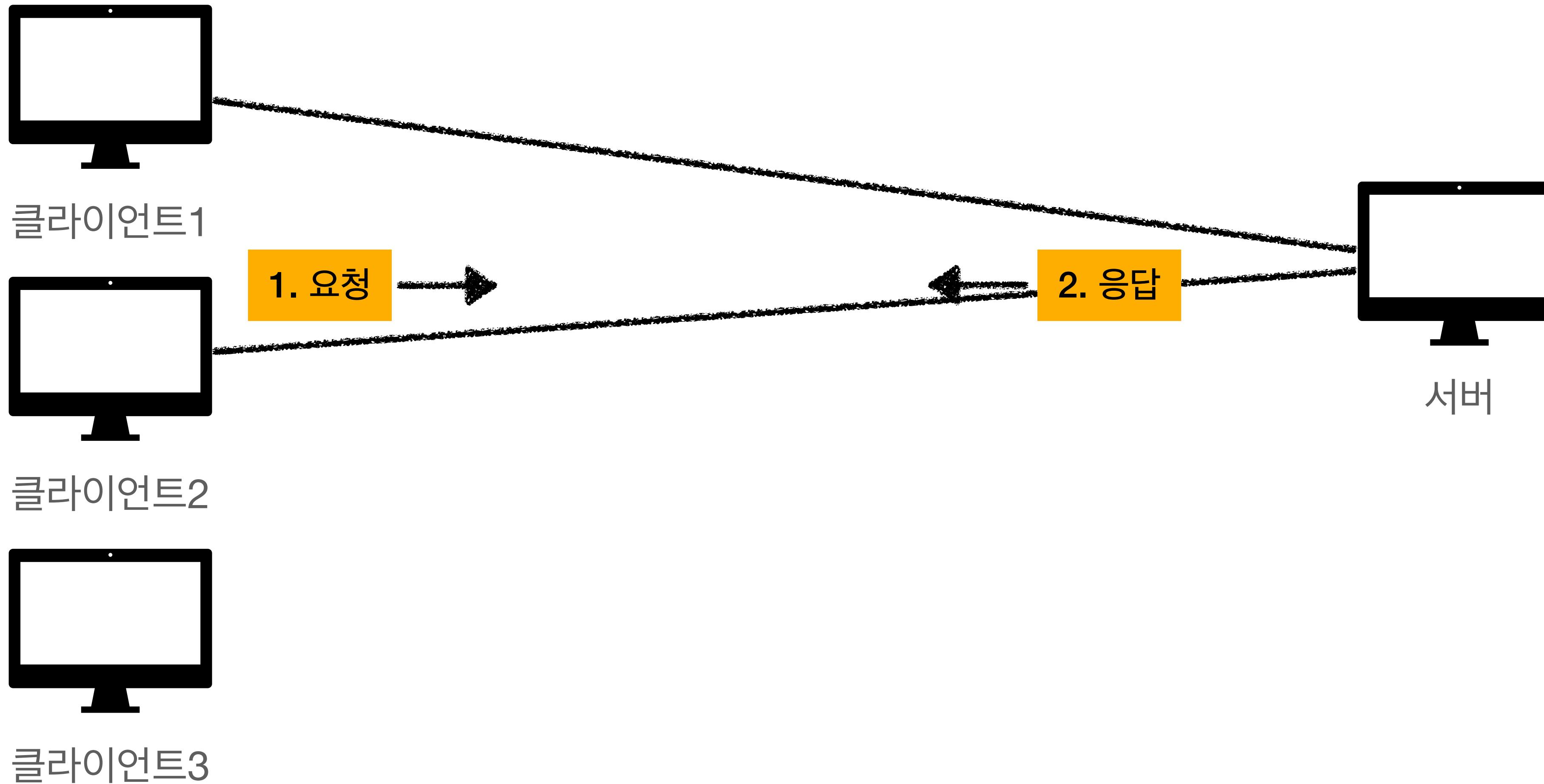
- 모든 것을 무상태로 설계 할 수 있는 경우도 있고 없는 경우도 있다.
- 무상태
 - 예) 로그인이 필요 없는 단순한 서비스 소개 화면
- 상태 유지
 - 예) 로그인
- 로그인한 사용자의 경우 로그인 했다는 상태를 서버에 유지
- 일반적으로 브라우저 쿠키와 서버 세션등을 사용해서 상태 유지
- 상태 유지는 최소한만 사용

비 연결성(connectionless)

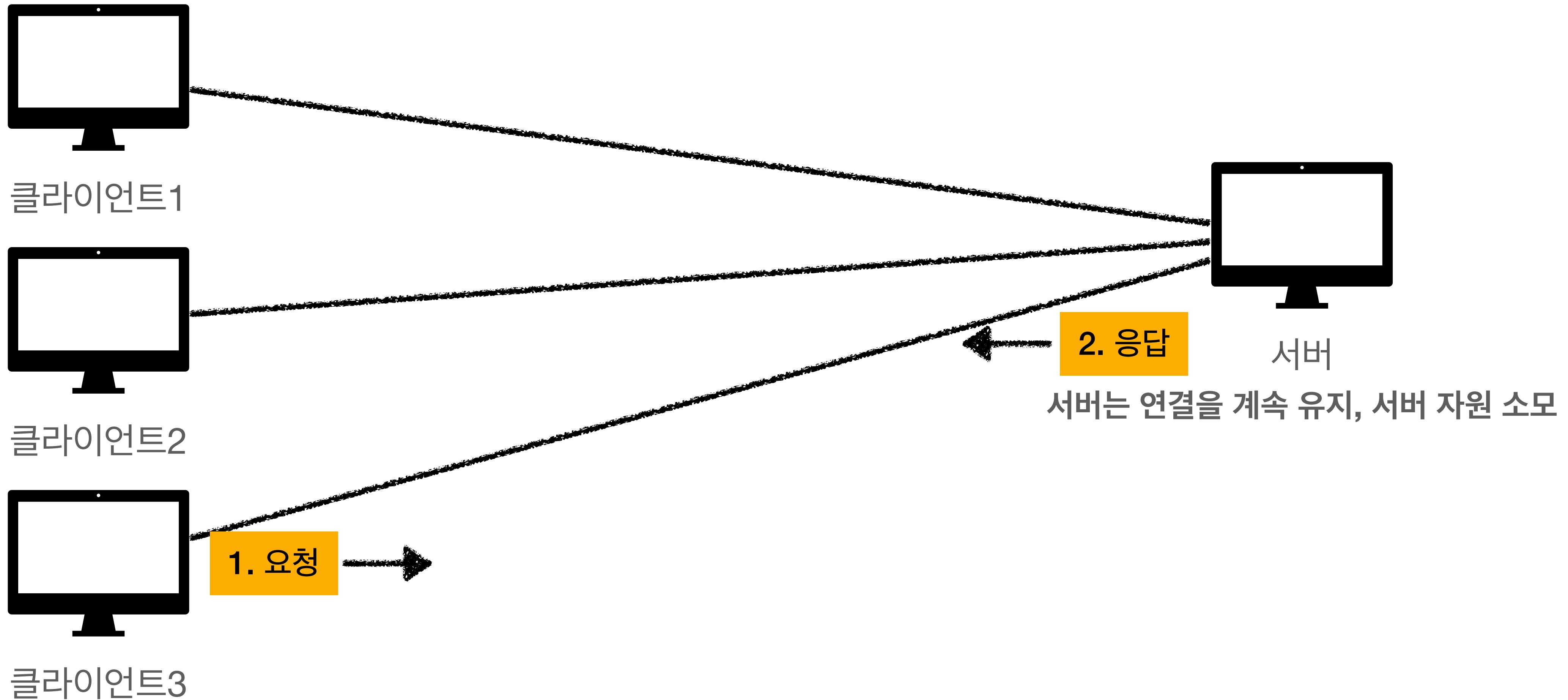
연결을 유지하는 모델



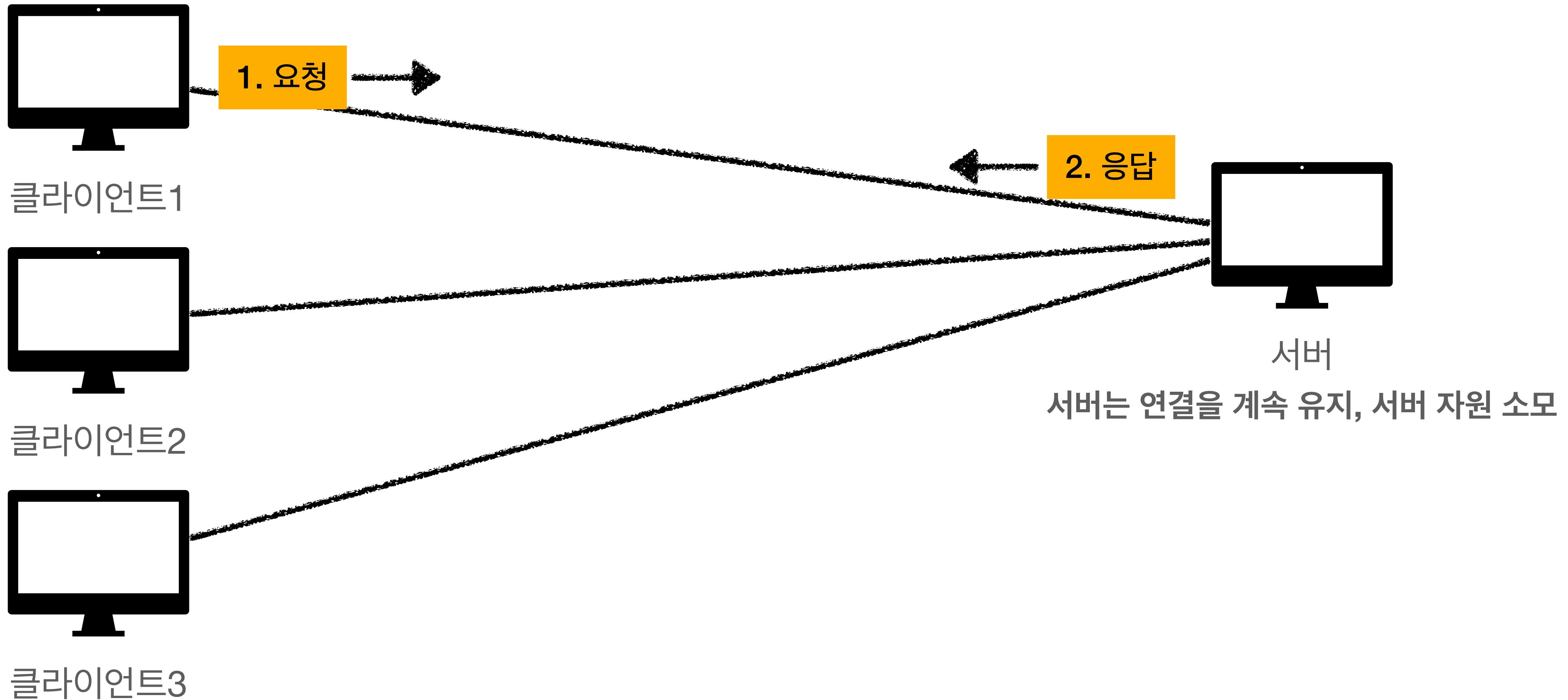
연결을 유지하는 모델



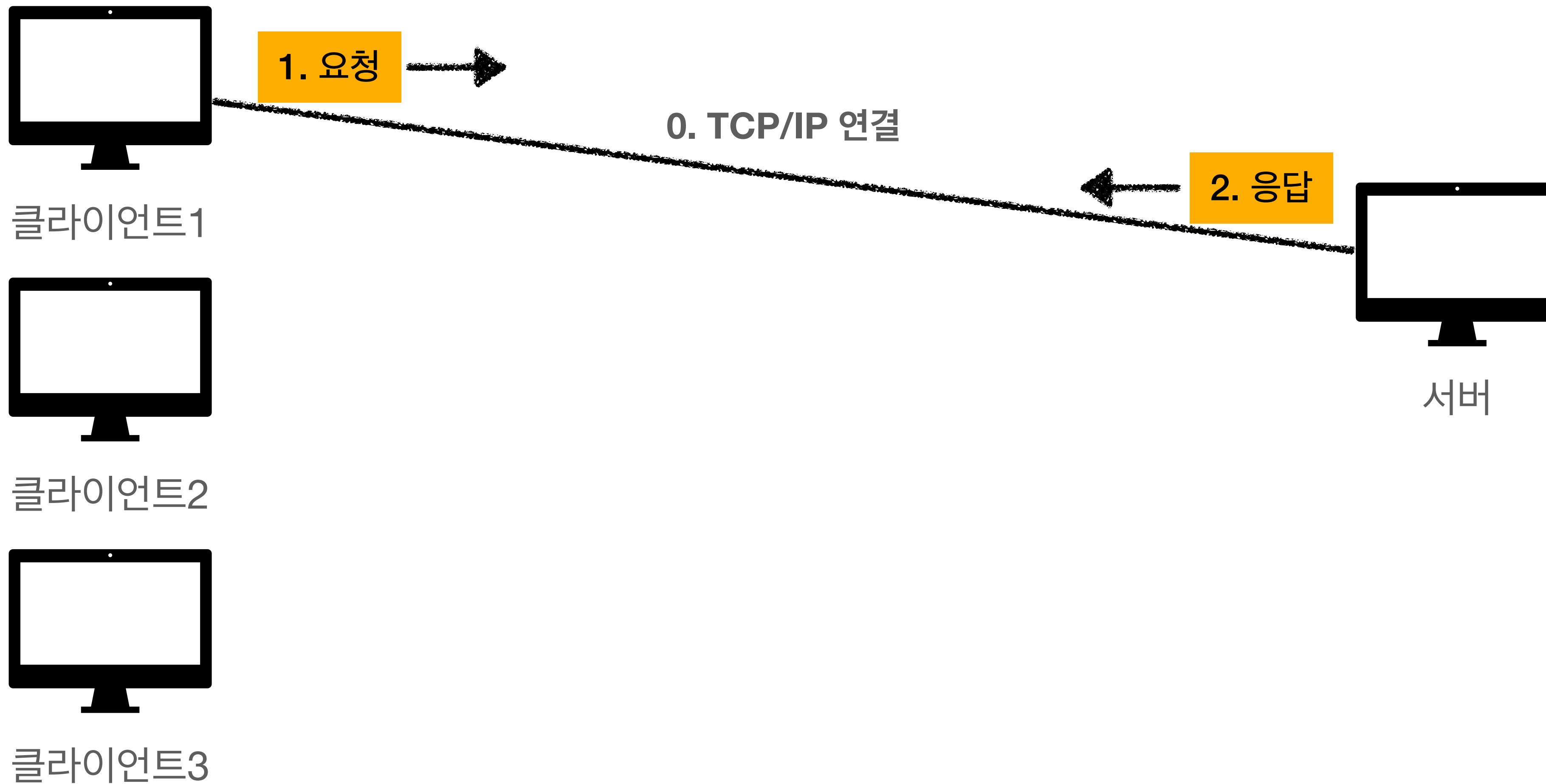
연결을 유지하는 모델



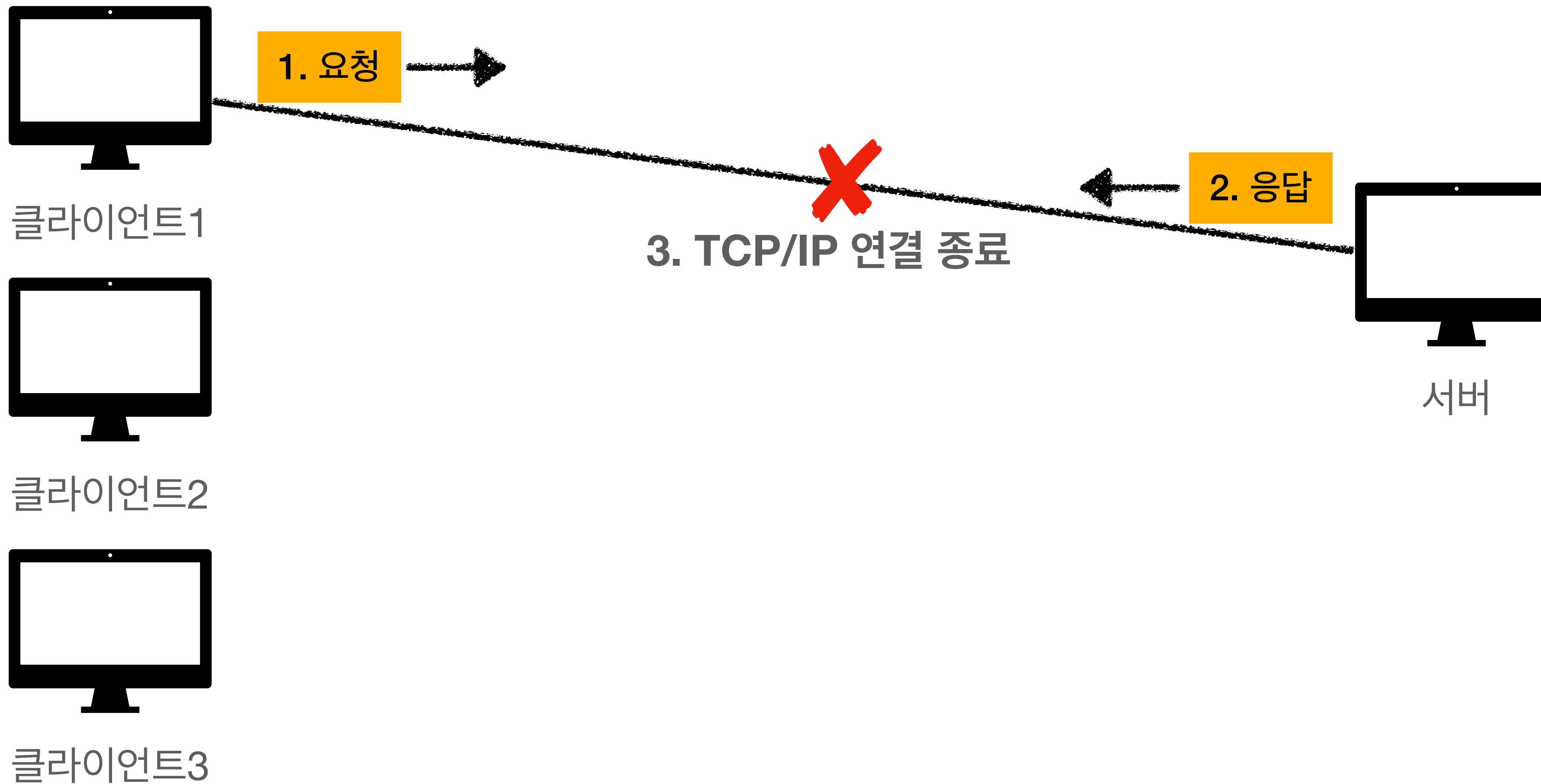
연결을 유지하는 모델



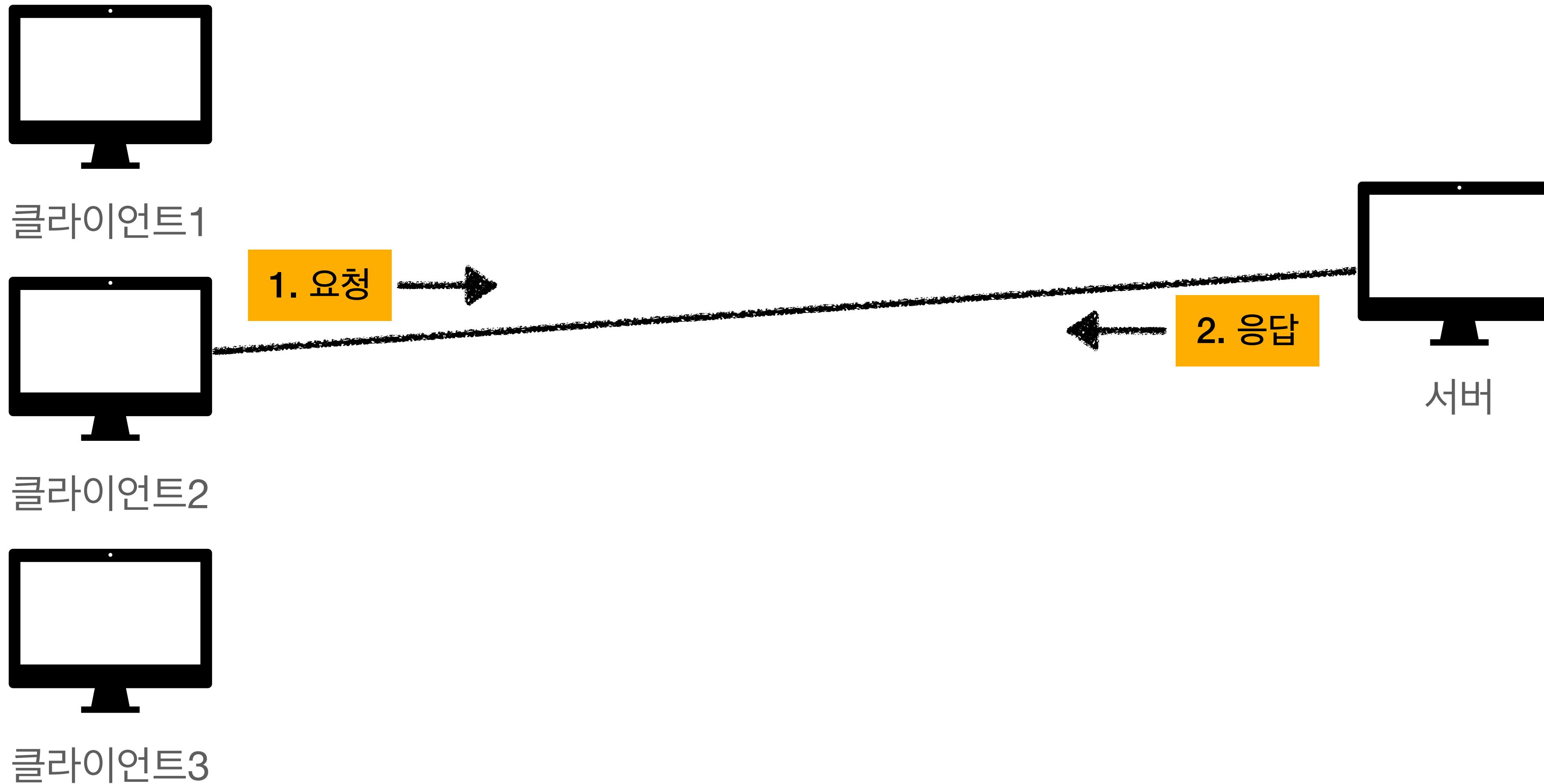
연결을 유지하지 않는 모델



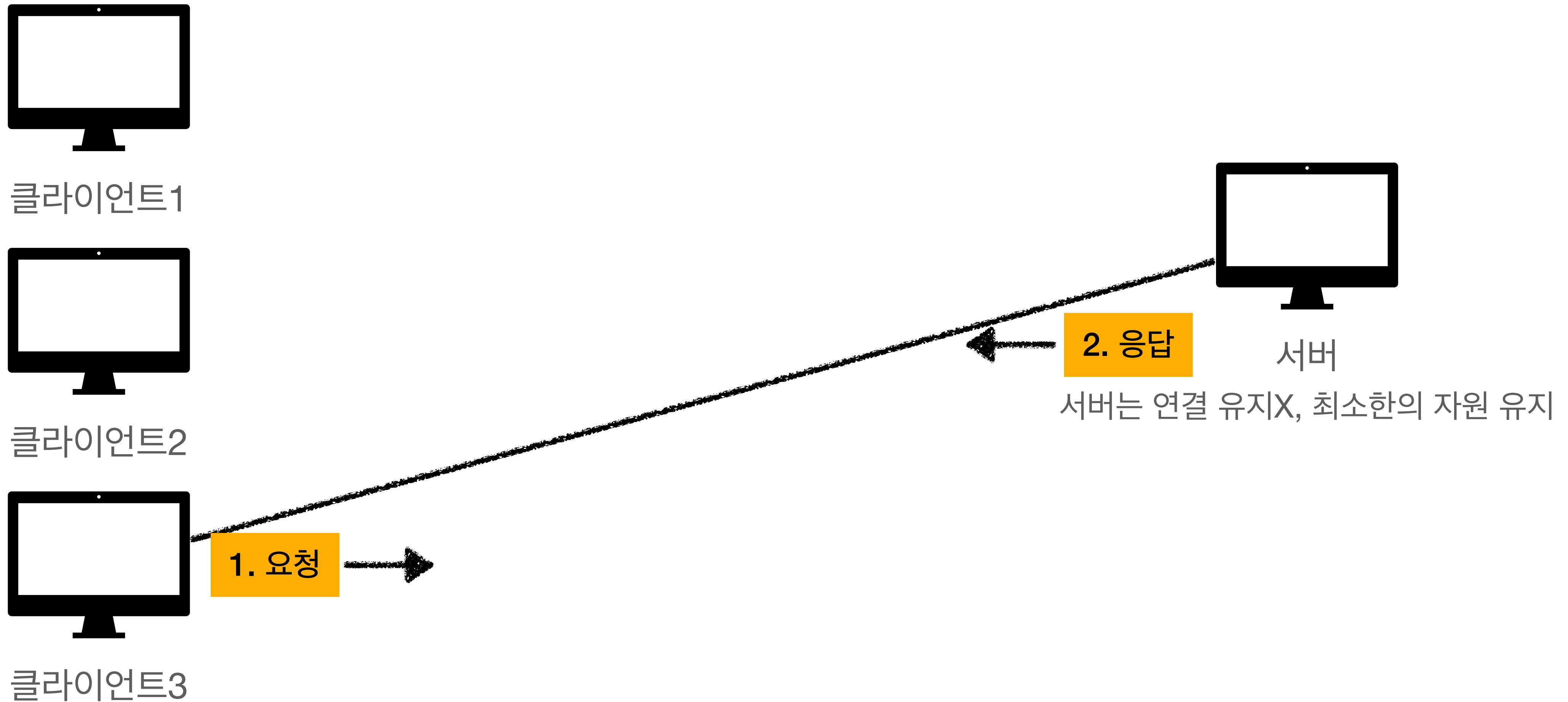
연결을 유지하지 않는 모델



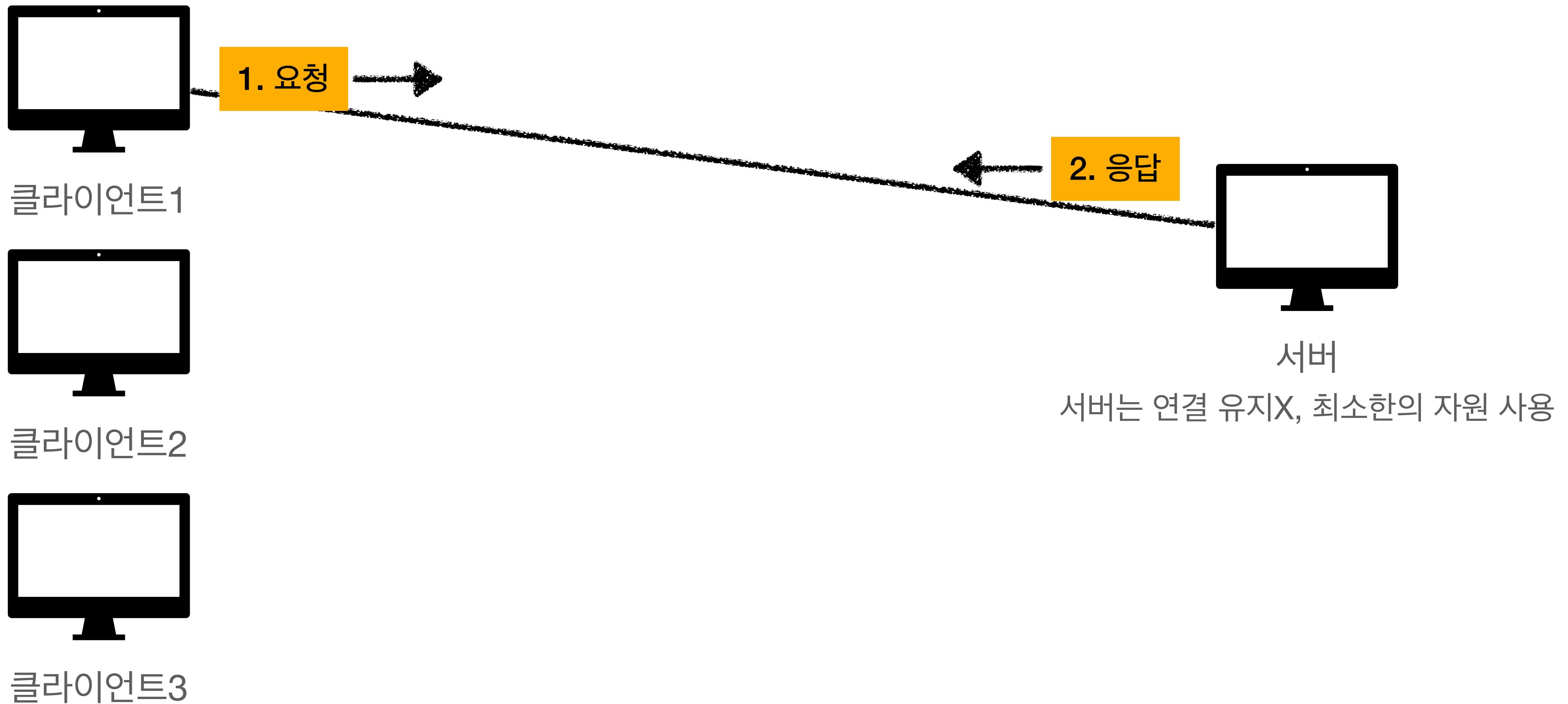
연결을 유지하지 않는 모델



연결을 유지하지 않는 모델



연결을 유지하지 않는 모델



비 연결성

- HTTP는 기본이 연결을 유지하지 않는 모델
- 일반적으로 초 단위의 이하의 빠른 속도로 응답
- 1시간 동안 수천명이 서비스를 사용해도 실제 서버에서 동시에 처리하는 요청은 수십개 이하로 매우 작음
 - 예) 웹 브라우저에서 계속 연속해서 검색 버튼을 누르지는 않는다.
- 서버 자원을 매우 효율적으로 사용할 수 있음

비 연결성

한계와 극복

- TCP/IP 연결을 새로 맺어야 함 - 3 way handshake 시간 추가
- 웹 브라우저로 사이트를 요청하면 HTML 뿐만 아니라 자바스크립트, css, 추가 이미지 등 수 많은 자원이 함께 다운로드
- 지금은 HTTP 지속 연결(Persistent Connections)로 문제 해결
- HTTP/2, HTTP/3에서 더 많은 최적화

HTTP 초기 - 연결, 종료 낭비



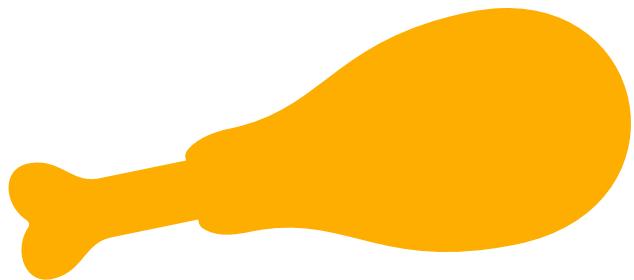
HTTP 지속 연결(Persistent Connections)



스테이스리스를 기억하자

서버 개발자들이 어려워하는 업무

- 정말 같은 시간에 딱 맞추어 발생하는 대용량 트래픽
- 예) 선착순 이벤트, 명절 KTX 예약, 학과 수업 등록
- 예) 저녁 6:00 선착순 1000명 치킨 할인 이벤트 -> 수만명 동시 요청



HTTP 메시지

모든 것이 HTTP - 한번 더!

HTTP 메시지에 모든 것을 전송

- HTML, TEXT
- IMAGE, 음성, 영상, 파일
- JSON, XML
- 거의 모든 형태의 데이터 전송 가능
- 서버간에 데이터를 주고 받을 때도 대부분 HTTP 사용
- 지금은 HTTP 시대!

GET /search?q=hello&hl=ko HTTP/1.1

Host: www.google.com

예) HTTP 요청 메시지

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

예) HTTP 응답 메시지

GET /search?q=hello&hl=ko HTTP/1.1

Host: www.google.com

예) HTTP 요청 메시지

요청 메시지도 body 본문을 가질 수 있음

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8
Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

예) HTTP 응답 메시지

start-line 시작 라인

header 헤더

empty line 공백 라인 (CRLF)

message body

HTTP 메시지 구조

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8
Content-Length: 3423

<html>
 <body>...</body>
</html>

HTTP-message = start-line
 *(header-field CRLF)
CRLF
[message-body]

<https://tools.ietf.org/html/rfc7230#section-3>

예) HTTP 응답 메시지

공식 스펙

시작 라인

요청 메시지

- start-line = **request-line** / status-line

```
GET /search?q=hello&hl=ko HTTP/1.1
```

Host: www.google.com

- **request-line** = method SP(공백) request-target SP HTTP-version CRLF(엔터)
- HTTP 메서드 (GET: 조회)
- 요청 대상 (/search?q=hello&hl=ko)
- HTTP Version

시작 라인

요청 메시지 - HTTP 메서드

- 종류: GET, POST, PUT, DELETE...
- 서버가 수행해야 할 동작 지정
 - GET: 리소스 조회
 - POST: 요청 내역 처리

```
GET /search?q=hello&hl=ko HTTP/1.1
```

```
Host: www.google.com
```

시작 라인

요청 메시지 - 요청 대상

- absolute-path[?query] (절대경로[?쿼리])
- 절대경로= "/" 로 시작하는 경로
- 참고: *, http://...?x=y 와 같이 다른 유형의 경로지정 방법도 있다.

```
GET /search?q=hello&hl=ko HTTP/1.1
```

```
Host: www.google.com
```

시작 라인

요청 메시지 - HTTP 버전

- HTTP Version

```
GET /search?q=hello&hl=ko HTTP/1.1
```

```
Host: www.google.com
```

시작 라인 응답 메시지

- start-line = request-line / **status-line**
- **status-line** = HTTP-version SP status-code SP reason-phrase CRLF
- HTTP 버전
- HTTP 상태 코드: 요청 성공, 실패를 나타냄
 - 200: 성공
 - 400: 클라이언트 요청 오류
 - 500: 서버 내부 오류
- 이유 문구: 사람이 이해할 수 있는 짧은 상태 코드 설명 글

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

HTTP 헤더

- header-field = field-name ":" OWS field-value OWS (OWS:띄어쓰기 허용)
- field-name은 대소문자 구분 없음

```
GET /search?q=hello&hl=ko HTTP/1.1  
Host: www.google.com
```

```
HTTP/1.1 200 OK  
Content-Type: text/html; charset=UTF-8  
Content-Length: 3423
```

```
<html>  
<body>...</body>  
</html>
```

HTTP 헤더 용도

- HTTP 전송에 필요한 모든 부가정보
- 예) 메시지 바디의 내용, 메시지 바디의 크기, 압축, 인증, 요청 클라이언트(브라우저) 정보, 서버 애플리케이션 정보, 캐시 관리 정보...
- 표준 헤더가 너무 많음
 - https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- 필요시 임의의 헤더 추가 가능
 - helloworld: hihi

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

HTTP 메시지 바디 용도

- 실제 전송할 데이터
- HTML 문서, 이미지, 영상, JSON 등등 byte로 표현할 수 있는 모든 데이터 전송 가능

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

단순함 확장 가능

- HTTP는 단순하다. 스펙도 읽어볼만...
- HTTP 메시지도 매우 단순
- 크게 성공하는 표준 기술은 단순하지만 확장 가능한 기술

HTTP 정리

- HTTP 메시지에 모든 것을 전송
- HTTP 역사 HTTP/1.1을 기준으로 학습
- 클라이언트 서버 구조
- 무상태 프로토콜(스테이스리스)
- HTTP 메시지
- 단순함, 확장 가능
- 지금은 HTTP 시대

HTTP 메서드

- HTTP API를 만들어보자
- HTTP 메서드 - GET, POST
- HTTP 메서드 - PUT, PATCH, DELETE
- HTTP 메서드의 속성

HTTP API를 만들어보자.

요구사항

회원 정보 관리 API를 만들어라.

- 회원 목록 조회
- 회원 조회
- 회원 등록
- 회원 수정
- 회원 삭제

API URI 설계

URI(Uniform Resource Identifier)

- 회원 목록 조회 /read-member-list
- 회원 조회 /read-member-by-id
- 회원 등록 /create-member
- 회원 수정 /update-member
- 회원 삭제 /delete-member

이것은 좋은 URI 설계일까?

가장 중요한 것은

리소스 씩별

API URI 고민

URI(Uniform Resource Identifier)

- 리소스의 의미는 뭘까?
 - 회원을 등록하고 수정하고 조회하는게 리소스가 아니다!
 - 예) 미네랄을 캐라 -> 미네랄이 리소스
 - 회원이라는 개념 자체가 바로 리소스다.
- 리소스를 어떻게 식별하는게 좋을까?
 - 회원을 등록하고 수정하고 조회하는 것을 모두 배제
 - 회원이라는 리소스만 식별하면 된다. -> 회원 리소스를 URI에 매핑

API URI 설계

URI(Uniform Resource Identifier)

- 회원 목록 조회
- 회원 조회
- 회원 등록
- 회원 수정
- 회원 삭제

API URI 설계

리소스 식별, URI 계층 구조 활용

- 회원 목록 조회 /members
- 회원 조회 /members/{id}
- 회원 등록 /members/{id}
- 회원 수정 /members/{id}
- 회원 삭제 /members/{id}
- 참고: 계층 구조상 상위를 컬렉션으로 보고 복수단어 사용 권장(member -> members)

API URI 설계

리소스 식별, URI 계층 구조 활용

- 회원 목록 조회 /members
- 회원 조회 /members/{id} -> 어떻게 구분하지?
- 회원 등록 /members/{id} -> 어떻게 구분하지?
- 회원 수정 /members/{id} -> 어떻게 구분하지?
- 회원 삭제 /members/{id} -> 어떻게 구분하지?

리소스와 행위을 분리

가장 중요한 것은 리소스를 식별하는 것

- **URI는 리소스만 식별!**
- 리소스와 해당 리소스를 대상으로 하는 행위을 분리
 - 리소스: 회원
 - 행위: 조회, 등록, 삭제, 변경
- 리소스는 명사, 행위는 동사 (미네랄을 캐라)
- 행위(메서드)는 어떻게 구분?

HTTP 메서드 - GET, POST

HTTP 메서드 종류

주요 메서드

Representation 설명 전

- GET: 리소스 조회
- POST: 요청 데이터 처리, 주로 등록에 사용
- PUT: 리소스를 대체, 해당 리소스가 없으면 생성
- PATCH: 리소스 부분 변경
- DELETE: 리소스 삭제

HTTP 메서드 종류

기타 메서드

- **HEAD:** GET과 동일하지만 메시지 부분을 제외하고, 상태 줄과 헤더만 반환
- **OPTIONS:** 대상 리소스에 대한 통신 가능 옵션(메서드)을 설명(주로 CORS에서 사용)
- **CONNECT:** 대상 리소스로 식별되는 서버에 대한 터널을 설정
- **TRACE:** 대상 리소스에 대한 경로를 따라 메시지 루프백 테스트를 수행

GET

```
GET /search?q=hello&hl=ko HTTP/1.1  
Host: www.google.com
```

- 리소스 조회
- 서버에 전달하고 싶은 데이터는 query(쿼리 파라미터, 쿼리 스트링)를 통해서 전달
- 메시지 바디를 사용해서 데이터를 전달할 수 있지만, 지원하지 않는 곳이 많아서 권장하지 않음

GET

리소스 조회1 - 메시지 전달

```
GET /members/100 HTTP/1.1  
Host: localhost:8080
```

/members/100

```
{  
    "username": "young",  
    "age": 20  
}
```



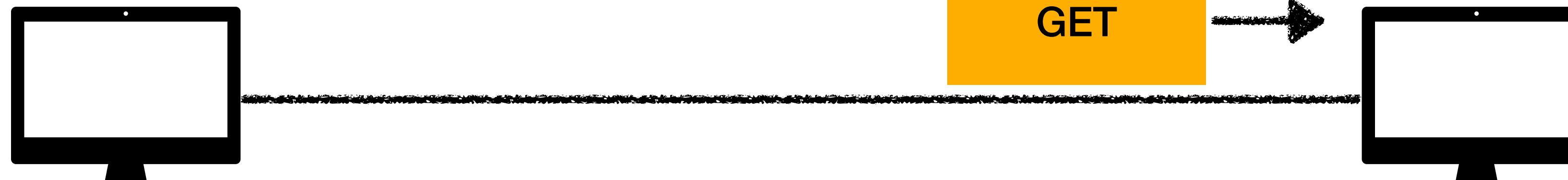
GET

리소스 조회2 - 서버도착

```
GET /members/100 HTTP/1.1  
Host: localhost:8080
```

/members/100

```
{  
    "username": "young",  
    "age": 20  
}
```



클라이언트

서버

GET

리소스 조회3 - 응답 데이터

응답 데이터

HTTP/1.1 200 OK

Content-Type: application/json

Content-Length: 34

{

 "username": "young",
 "age": 20

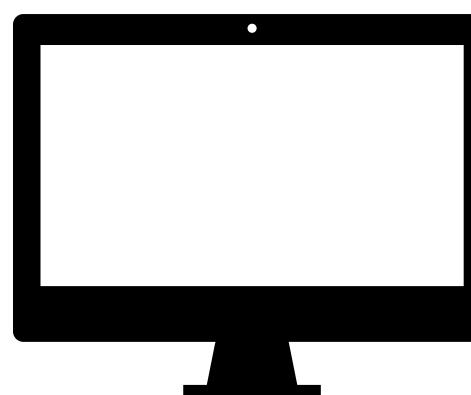
}

/members/100

{

 "username": "young",
 "age": 20

}



클라이언트



Response



서버

POST

```
POST /members HTTP/1.1  
Content-Type: application/json
```

```
{  
  "username": "hello",  
  "age": 20  
}
```

- 요청 데이터 처리
- 메시지 바디를 통해 서버로 요청 데이터 전달
- 서버는 요청 데이터를 처리
 - 메시지 바디를 통해 들어온 데이터를 처리하는 모든 기능을 수행한다.
- 주로 전달된 데이터로 신규 리소스 등록, 프로세스 처리에 사용

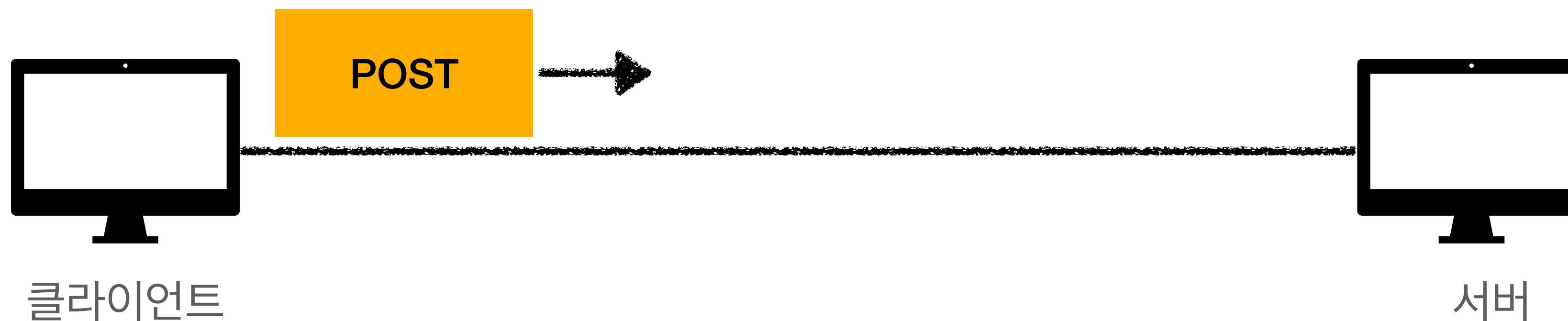
POST

리소스 등록1 - 메시지 전달

```
POST /members HTTP/1.1  
Content-Type: application/json
```

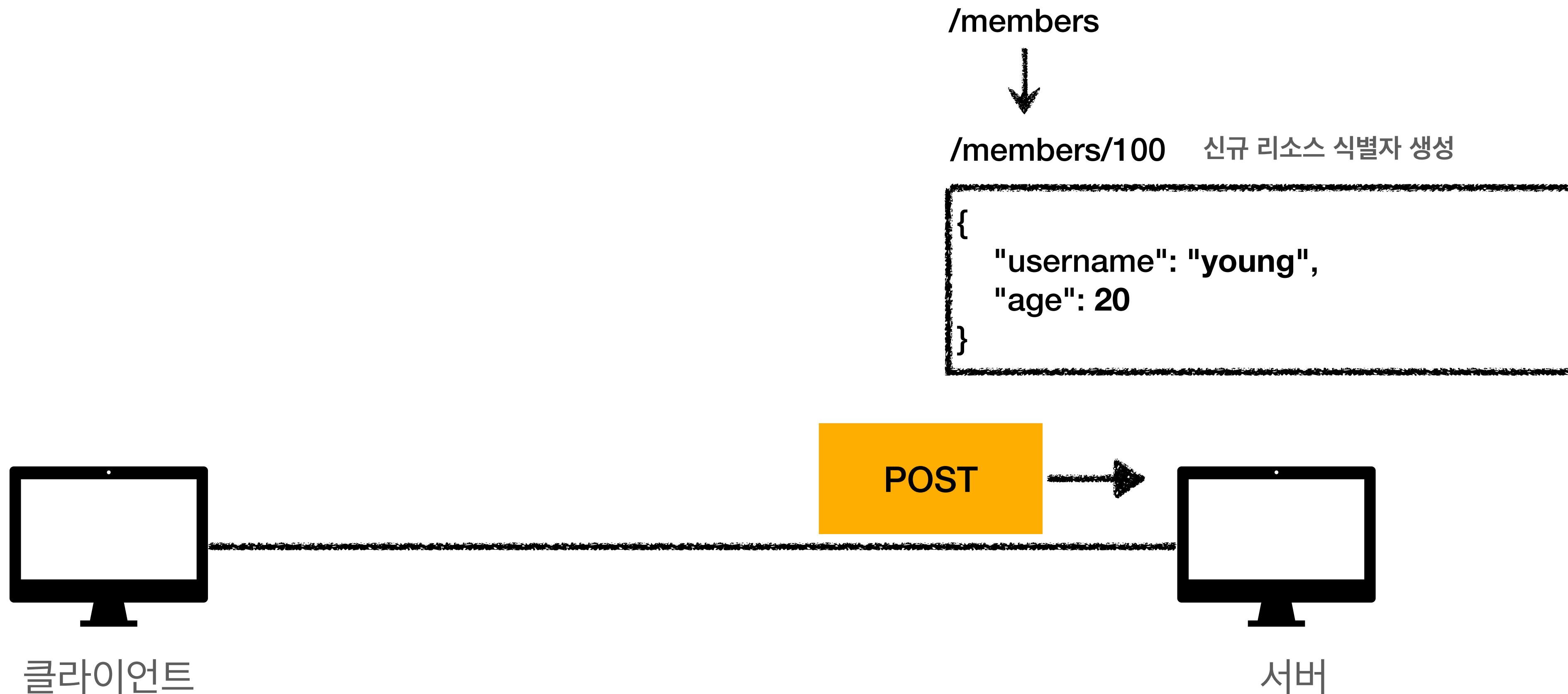
```
{  
  "username": "young",  
  "age": 20  
}
```

/members



POST

리소스 등록2 - 신규 리소스 생성



POST

리소스 등록3 - 응답 데이터

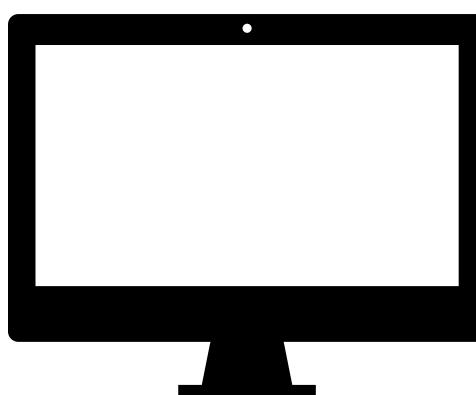
응답 데이터

```
HTTP/1.1 201 Created  
Content-Type: application/json  
Content-Length: 34  
Location: /members/100
```

```
{  
    "username": "young",  
    "age": 20  
}
```

/members/100

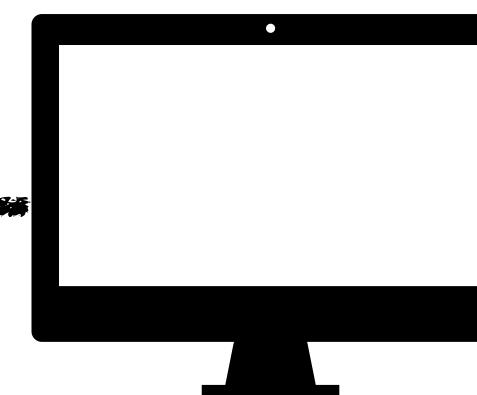
```
{  
    "username": "young",  
    "age": 20  
}
```



클라이언트



Response



서버

POST

요청 데이터를 어떻게 처리한다는 뜻일까? 예시

- 스펙: POST 메서드는 대상 리소스가 리소스의 고유 한 의미 체계에 따라 요청에 포함 된 표현을 처리하도록 요청합니다. (구글 번역)
- 예를 들어 POST는 다음과 같은 기능에 사용됩니다.
 - HTML 양식에 입력 된 필드와 같은 데이터 블록을 데이터 처리 프로세스에 제공
 - 예) HTML FORM에 입력한 정보로 회원 가입, 주문 등에서 사용
 - 게시판, 뉴스 그룹, 메일링 리스트, 블로그 또는 유사한 기사 그룹에 메시지 게시
 - 예) 게시판 글쓰기, 댓글 달기
 - 서버가 아직 식별하지 않은 새 리소스 생성
 - 예) 신규 주문 생성
 - 기존 자원에 데이터 추가
 - 예) 한 문서 끝에 내용 추가하기
- 정리: 이 리소스 URI에 POST 요청이 오면 요청 데이터를 어떻게 처리할지 리소스마다 따로 정해야 함 -> 정해진 것이 없음

POST

정리

- 1. 새 리소스 생성(등록)
 - 서버가 아직 식별하지 않은 새 리소스 생성
- 2. 요청 데이터 처리
 - 단순히 데이터를 생성하거나, 변경하는 것을 넘어서 프로세스를 처리해야 하는 경우
 - 예) 주문에서 결제완료 -> 배달시작 -> 배달완료 처럼 단순히 값 변경을 넘어 프로세스의 상태가 변경되는 경우
 - POST의 결과로 새로운 리소스가 생성되지 않을 수도 있음
 - 예) POST /orders/{orderId}/start-delivery (**컨트롤 URI**)
- 3. 다른 메서드로 처리하기 애매한 경우
 - 예) JSON으로 조회 데이터를 넘겨야 하는데, GET 메서드를 사용하기 어려운 경우
 - 애매하면 POST

HTTP 메서드 - PUT, PATCH, DELETE

PUT

```
PUT /members/100 HTTP/1.1  
Content-Type: application/json
```

```
{  
    "username": "hello",  
    "age": 20  
}
```

- 리소스를 대체
 - 리소스가 있으면 대체
 - 리소스가 없으면 생성
 - 쉽게 이야기해서 덮어버림
- 중요! 클라이언트가 리소스를 식별
 - 클라이언트가 리소스 위치를 알고 URI 지정
 - POST와 차이점

PUT

리소스가 있는 경우1

```
PUT /members/100 HTTP/1.1  
Content-Type: application/json  
  
{  
  "username": "old",  
  "age": 50  
}
```

/members/100

```
{  
  "username": "young",  
  "age": 20  
}
```



PUT

리소스가 있는 경우2

리소스 대체

/members/100

```
{  
  "username": "old",  
  "age": 50  
}
```



PUT

리소스가 없는 경우1

```
PUT /members/100 HTTP/1.1  
Content-Type: application/json  
  
{  
    "username": "old",  
    "age": 50  
}
```

이런 리소스는 없음

/members/100



PUT

리소스가 없는 경우2

신규 리소스 생성

/members/100

```
{  
    "username": "old",  
    "age": 50  
}
```



PUT

주의! - 리소스를 완전히 대체한다!

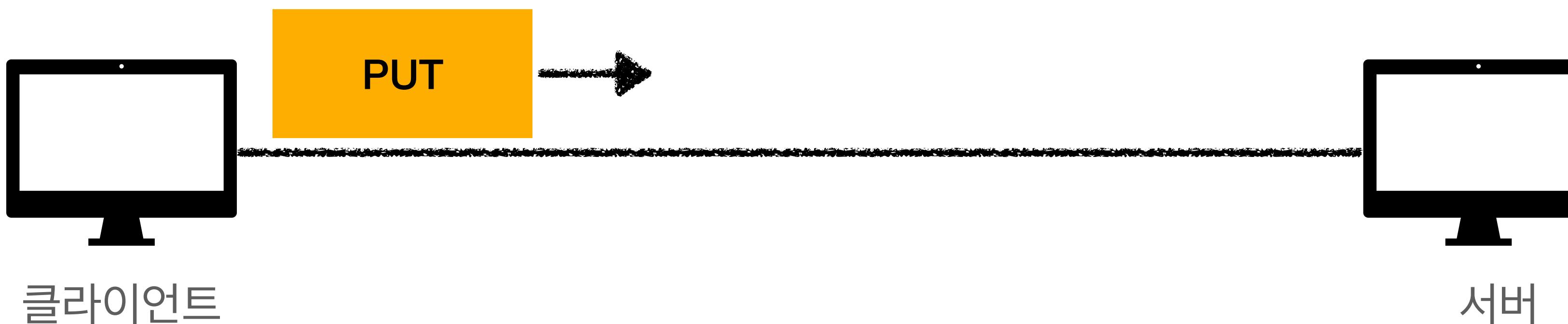
```
PUT /members/100 HTTP/1.1  
Content-Type: application/json
```

```
{  
    "age": 50  
}
```

/members/100

```
{  
    "username": "young",  
    "age": 20  
}
```

username 필드 없음



PUT

주의! - 리소스를 완전히 대체한다2

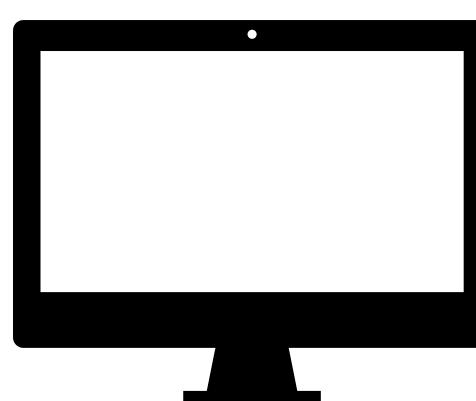
리소스 대체

/members/100

```
{  
  "age": 50,  
}
```



username 필드 삭제됨



클라이언트

PUT →



서버

PATCH

```
PATCH /members/100 HTTP/1.1  
Content-Type: application/json
```

```
{  
    "age": 50  
}
```

- 리소스 부분 변경

PATCH

리소스 부분 변경1

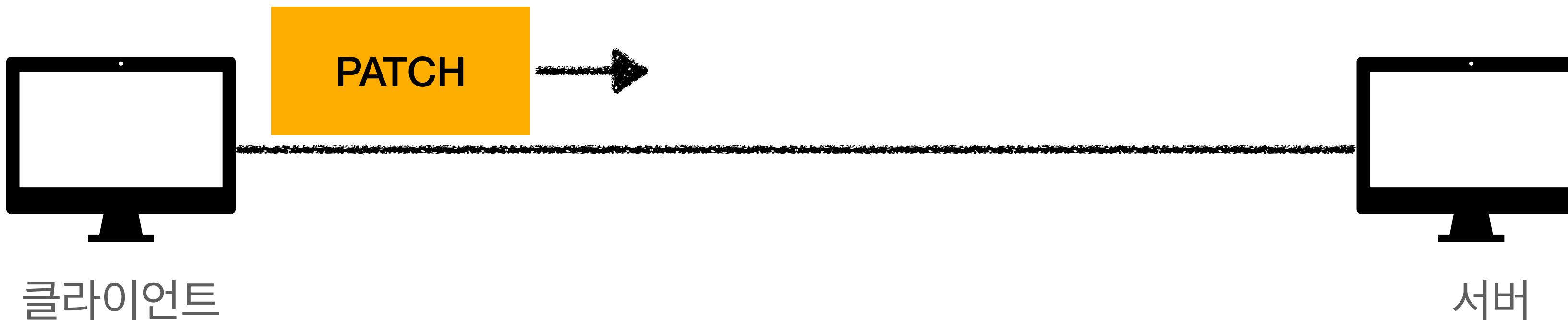
```
PATCH /members/100 HTTP/1.1  
Content-Type: application/json
```

```
{  
  "age": 50  
}
```

/members/100

```
{  
  "username": "young",  
  "age": 20  
}
```

username 필드 없음



PATCH

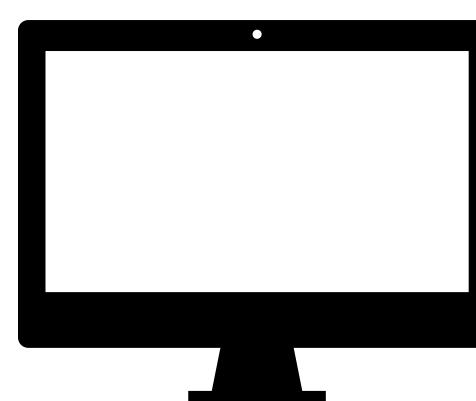
리소스 부분 변경2

리소스 부분 변경

/members/100

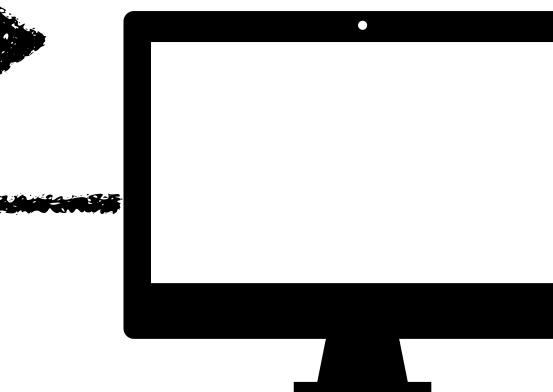
```
{  
  "username": "young",  
  "age": 50  
}
```

age만 50으로 변경



클라이언트

PATCH



서버

DELETE

```
DELETE /members/100 HTTP/1.1  
Host: localhost:8080
```

- 리소스 제거

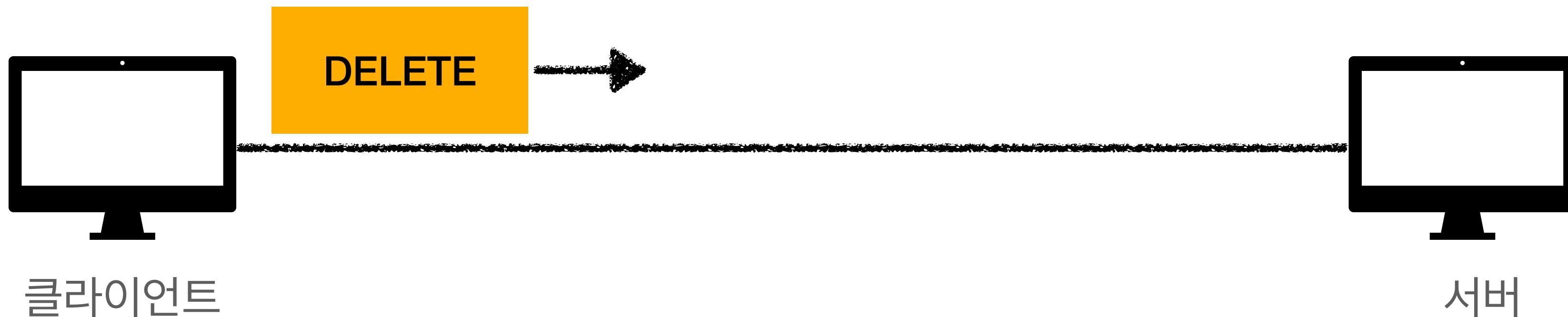
DELETE

리소스 제거1

```
DELETE /members/100 HTTP/1.1  
Host: localhost:8080
```

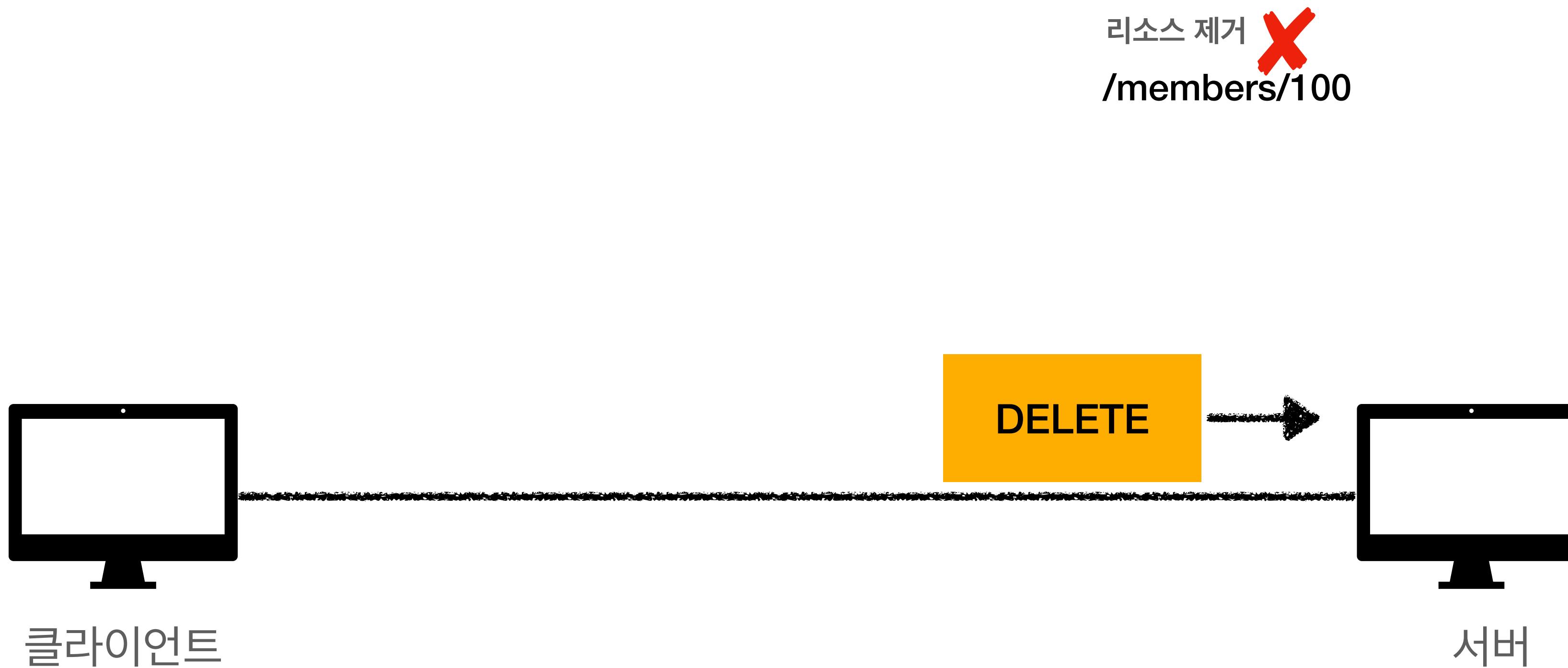
/members/100

```
{  
    "username": "young",  
    "age": 20  
}
```



DELETE

리소스 제거2



HTTP 메서드의 속성

- 안전(Safe Methods)
- 멱등(Idempotent Methods)
- 캐시가능(Cacheable Methods)

HTTP 메소드 ◆	RFC ◆	요청에 Body가 있음 ◆	응답에 Body가 있음 ◆	안전 ◆	멱등(Idempotent) ◆	캐시 가능 ◆
GET	RFC 7231 ↗	아니오	예	예	예	예
HEAD	RFC 7231 ↗	아니오	아니오	예	예	예
POST	RFC 7231 ↗	예	예	아니오	아니오	예
PUT	RFC 7231 ↗	예	예	아니오	예	아니오
DELETE	RFC 7231 ↗	아니오	예	아니오	예	아니오
CONNECT	RFC 7231 ↗	예	예	아니오	아니오	아니오
OPTIONS	RFC 7231 ↗	선택 사항	예	예	예	아니오
TRACE	RFC 7231 ↗	아니오	예	예	예	아니오
PATCH	RFC 5789 ↗	예	예	아니오	아니오	예

출처: <https://ko.wikipedia.org/wiki/HTTP>

안전

Safe

- 호출해도 리소스를 변경하지 않는다.
- Q: 그래도 계속 호출해서, 로그 같은게 쌓여서 장애가 발생하면요?
- A: 안전은 해당 리소스만 고려한다. 그런 부분까지 고려하지 않는다.

멱등

Idempotent

- $f(f(x)) = f(x)$
- 한 번 호출하든 두 번 호출하든 100번 호출하든 결과가 똑같다.
- 멱등 메서드
 - **GET**: 한 번 조회하든, 두 번 조회하든 같은 결과가 조회된다.
 - **PUT**: 결과를 대체한다. 따라서 같은 요청을 여러번 해도 최종 결과는 같다.
 - **DELETE**: 결과를 삭제한다. 같은 요청을 여러번 해도 삭제된 결과는 똑같다.
 - **POST**: 멱등이 아니다! 두 번 호출하면 같은 결제가 중복해서 발생할 수 있다.

멱등

Idempotent

- 활용
 - 자동 복구 메커니즘
 - 서버가 TIMEOUT 등으로 정상 응답을 못주었을 때, 클라이언트가 같은 요청을 다시 해도 되는가? 판단 근거

멱등

Idempotent

- Q: 재요청 중간에 다른 곳에서 리소스를 변경해버리면?
 - 사용자1: GET -> username:A, age:20
 - 사용자2: PUT -> username:A, age:30
 - 사용자1: GET -> username:A, age:30 -> 사용자2의 영향으로 바뀐 데이터 조회
- A: 멱등은 외부 요인으로 중간에 리소스가 변경되는 것 까지는 고려하지는 않는다.

캐시가능

Cacheable

- 응답 결과 리소스를 캐시해서 사용해도 되는가?
- GET, HEAD, POST, PATCH 캐시가능
- 실제로는 GET, HEAD 정도만 캐시로 사용
 - POST, PATCH는 본문 내용까지 캐시 키로 고려해야 하는데, 구현이 쉽지 않음

HTTP 메서드 활용

- 클라이언트에서 서버로 데이터 전송
- HTTP API 설계 예시

클라이언트에서 서버로 데이터 전송

데이터 전달 방식은 크게 2가지

- 쿼리 파라미터를 통한 데이터 전송
 - GET
 - 주로 정렬 필터(검색어)
- 메시지 바디를 통한 데이터 전송
 - POST, PUT, PATCH
 - 회원 가입, 상품 주문, 리소스 등록, 리소스 변경

클라이언트에서 서버로 데이터 전송

4가지 상황

- 정적 데이터 조회
 - 이미지, 정적 텍스트 문서
- 동적 데이터 조회
 - 주로 검색, 게시판 목록에서 정렬 필터(검색어)
- **HTML Form을 통한 데이터 전송**
 - 회원가입, 상품 주문, 데이터 변경
- **HTTP API를 통한 데이터 전송**
 - 회원가입, 상품 주문, 데이터 변경
 - 서버 to 서버, 앱 클라이언트, 웹 클라이언트(Ajax)

정적 데이터 조회

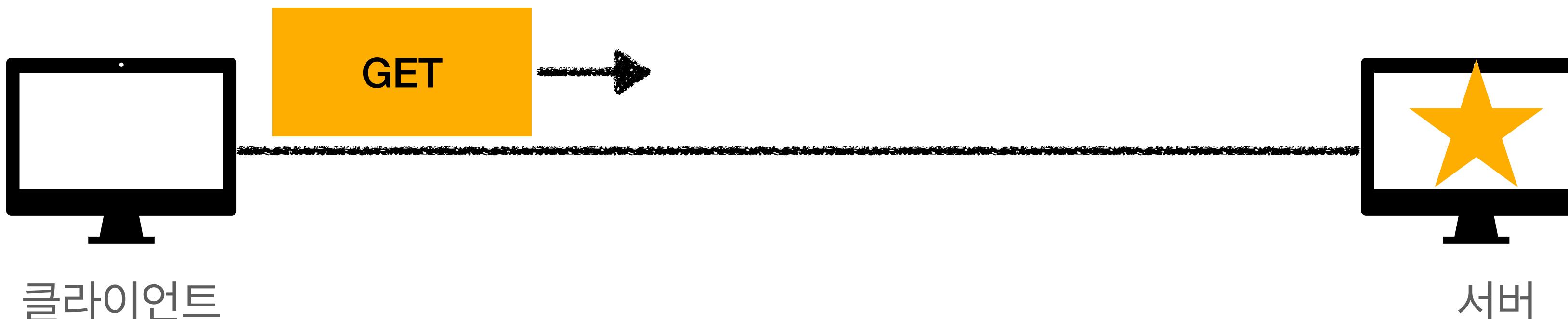
쿼리 파라미터 미사용

GET /static/star.jpg HTTP/1.1
Host: localhost:8080

/static/star.jpg

HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34lko98udjkla
slkjdf;lkqkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123



정적 데이터 조회

정리

- 이미지, 정적 텍스트 문서
- 조회는 GET 사용
- 정적 데이터는 일반적으로 쿼리 파라미터 없이 리소스 경로로 단순하게 조회 가능

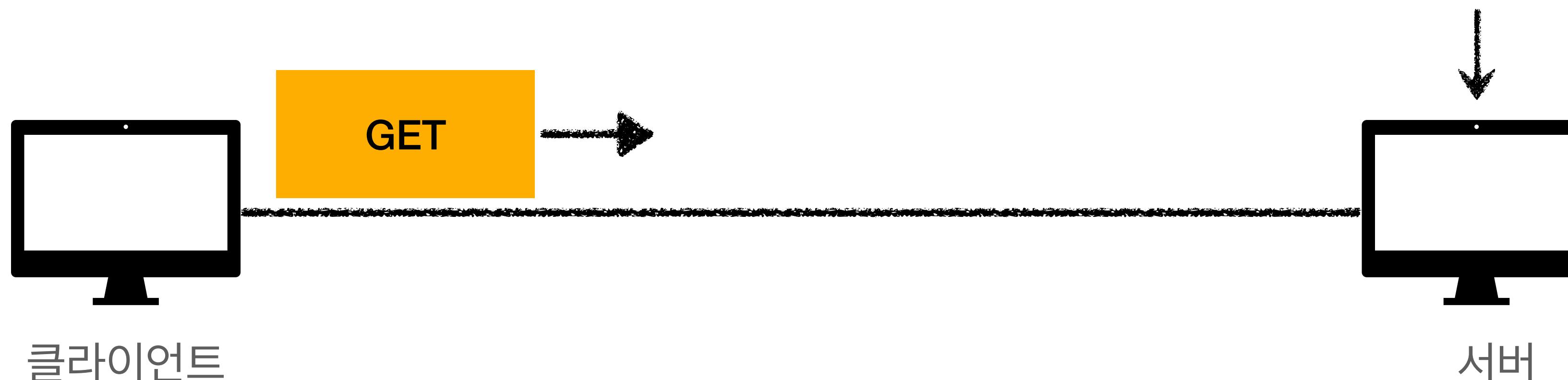
동적 데이터 조회

쿼리 파라미터 사용

```
https://www.google.com/search?q=hello&hl=ko
```

쿼리 파라미터
GET /search?q=hello&hl=ko HTTP/1.1
Host: www.google.com

쿼리 파라미터를 기반으로 정렬 필터해서 결과를 동적으로 생성



동적 데이터 조회 정리

- 주로 검색, 게시판 목록에서 정렬 필터(검색어)
- 조회 조건을 줄여주는 필터, 조회 결과를 정렬하는 정렬 조건에 주로 사용
- 조회는 GET 사용
- GET은 쿼리 파라미터 사용해서 데이터를 전달

HTML Form 데이터 전송

POST 전송 - 저장

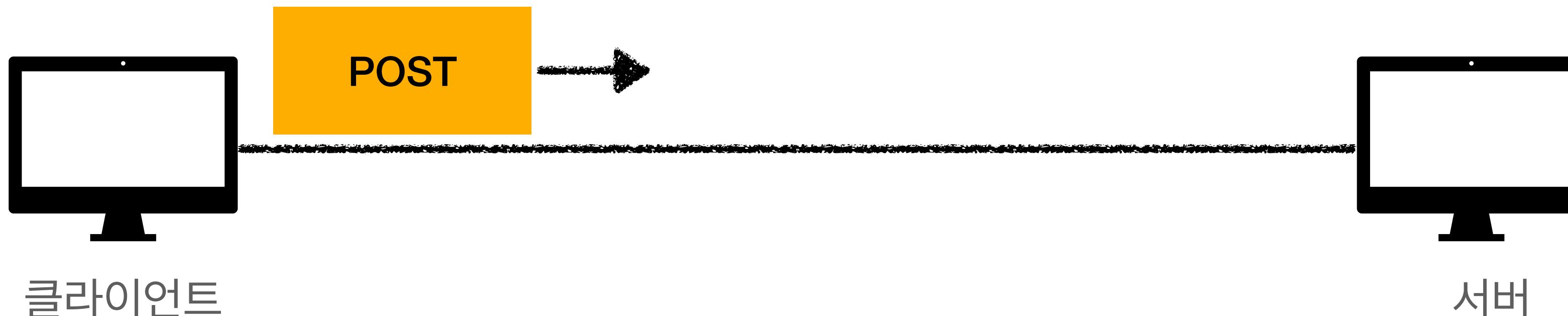
username: age:

```
<form action="/save" method="post">
  <input type="text" name="username" />
  <input type="text" name="age" />
  <button type="submit">전송</button>
</form>
```

웹 브라우저가 생성한 요청 HTTP 메시지

```
POST /save HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded

username=kim&age=20
```



HTML Form 데이터 전송

GET 전송 - 저장

username: age:

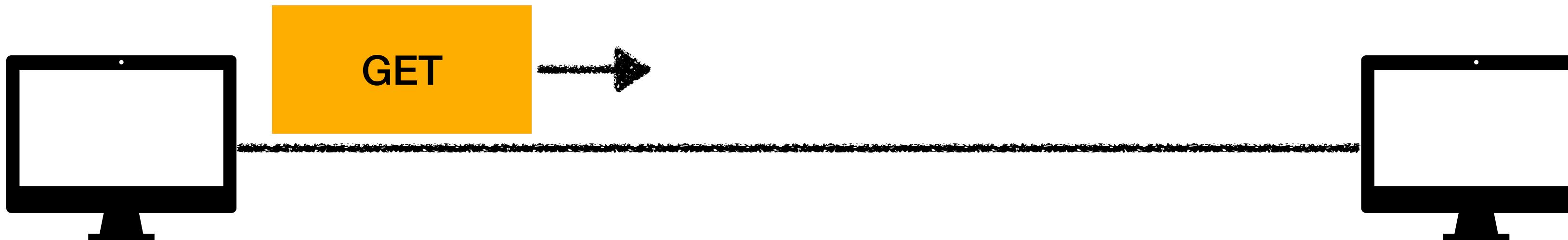
```
<form action="/save" method="get">  
  <input type="text" name="username" />  
  <input type="text" name="age" />  
  <button type="submit">전송</button>  
</form>
```

웹 브라우저가 생성한 요청 HTTP 메시지

GET /save?username=kim&age=20 HTTP/1.1
Host: localhost:8080



주의! GET은 조회에만 사용!
리소스 변경이 발생하는 곳에 사용하면 안됨!



클라이언트

서버

HTML Form 데이터 전송

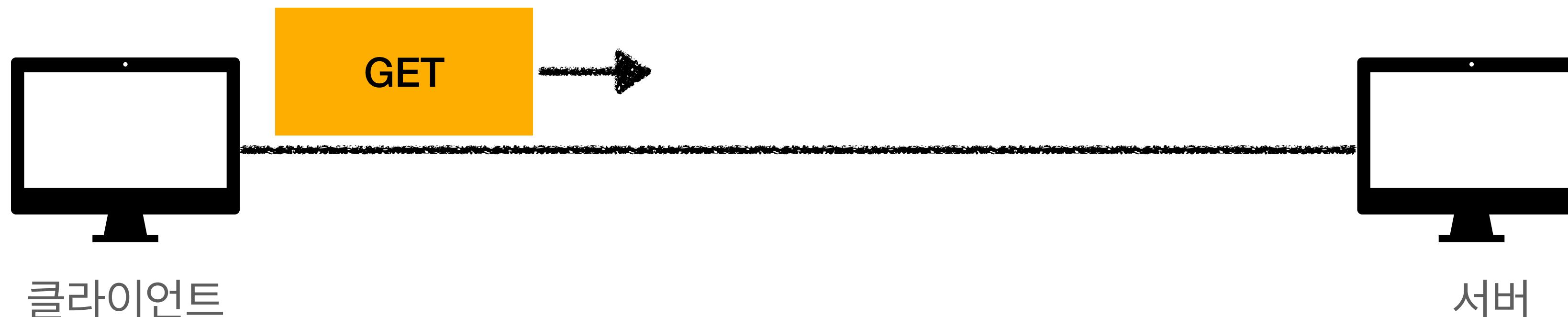
GET 전송 - 조회

username: age:

```
<form action="/members" method="get">
  <input type="text" name="username" />
  <input type="text" name="age" />
  <button type="submit">전송</button>
</form>
```

웹 브라우저가 생성한 요청 HTTP 메시지

```
GET /members?username=kim&age=20 HTTP/1.1
Host: localhost:8080
```



HTML Form 데이터 전송

multipart/form-data

웹 브라우저가 생성한 요청 HTTP 메시지

username:

age:

file: intro.png

```
<form action="/save" method="post" enctype="multipart/form-data">
<input type="text" name="username" />
<input type="text" name="age" />
<input type="file" name="file1" />
<button type="submit">전송</button>
</form>
```

POST /save HTTP/1.1
Host: localhost:8080
Content-Type: multipart/form-data; boundary=-----XXX
Content-Length: 10457

-----XXX
Content-Disposition: form-data; name="username"

kim
-----XXX
Content-Disposition: form-data; name="age"

20
-----XXX
Content-Disposition: form-data; name="file1"; filename="intro.png"
Content-Type: image/png

109238a9o0p3eqwokjasd09ou3oirjwoe9u34ouief...

-----XXX--

끝에는 -- 추가

HTML Form 데이터 전송

정리

- HTML Form submit시 POST 전송
 - 예) 회원 가입, 상품 주문, 데이터 변경
- Content-Type: application/x-www-form-urlencoded 사용
 - form의 내용을 메시지 바디를 통해서 전송(key=value, 쿼리 파라미터 형식)
 - 전송 데이터를 url encoding 처리
 - 예) abc김 -> abc%EA%B9%80
- HTML Form은 GET 전송도 가능
- Content-Type: multipart/form-data
 - 파일 업로드 같은 바이너리 데이터 전송시 사용
 - 다른 종류의 여러 파일과 폼의 내용 함께 전송 가능(그래서 이름이 multipart)
- 참고: HTML Form 전송은 **GET, POST**만 지원

HTTP API 데이터 전송

```
POST /members HTTP/1.1  
Content-Type: application/json  
  
{  
  "username": "young",  
  "age": 20  
}
```

/members



HTTP API 데이터 전송

정리

- 서버 to 서버
 - 백엔드 시스템 통신
- 앱 클라이언트
 - 아이폰, 안드로이드
- 웹 클라이언트
 - HTML에서 Form 전송 대신 자바 스크립트를 통한 통신에 사용(AJAX)
 - 예) React, VueJs 같은 웹 클라이언트와 API 통신
- POST, PUT, PATCH: 메시지 바디를 통해 데이터 전송
- GET: 조회, 쿼리 파라미터로 데이터 전달
- Content-Type: application/json을 주로 사용 (사실상 표준)
 - TEXT, XML, JSON 등등

HTTP API 설계 예시

- **HTTP API - 컬렉션**
 - **POST** 기반 등록
 - 예) 회원 관리 API 제공
- **HTTP API - 스토어**
 - **PUT** 기반 등록
 - 예) 정적 컨텐츠 관리, 원격 파일 관리
- **HTML FORM 사용**
 - 웹 페이지 회원 관리
 - GET, POST만 지원

회원 관리 시스템

API 설계 - POST 기반 등록

- 회원 목록 /members -> **GET**
- 회원 등록 /members -> **POST**
- 회원 조회 /members/{id} -> **GET**
- 회원 수정 /members/{id} -> **PATCH, PUT, POST**
- 회원 삭제 /members/{id} -> **DELETE**

회원 관리 시스템

POST - 신규 자원 등록 특징

- 클라이언트는 등록될 리소스의 URI를 모른다.
 - 회원 등록 /members -> POST
 - POST /members
- 서버가 새로 등록된 리소스 URI를 생성해준다.
 - HTTP/1.1 201 Created
Location: **/members/100**
- 컬렉션(Collection)
 - 서버가 관리하는 리소스 디렉토리
 - 서버가 리소스의 URI를 생성하고 관리
 - 여기서 컬렉션은 /members

파일 관리 시스템

API 설계 - PUT 기반 등록

- 파일 목록 /files -> **GET**
- 파일 조회 /files/{filename} -> **GET**
- 파일 등록 /files/{filename} -> **PUT**
- 파일 삭제 /files/{filename} -> **DELETE**
- 파일 대량 등록 /files -> **POST**

파일 관리 시스템

PUT - 신규 자원 등록 특징

- 클라이언트가 리소스 URI를 알고 있어야 한다.
 - 파일 등록 /files/{filename} -> PUT
 - PUT **/files/star.jpg**
- 클라이언트가 직접 리소스의 URI를 지정한다.
- 스토어(Store)
 - 클라이언트가 관리하는 리소스 저장소
 - 클라이언트가 리소스의 URI를 알고 관리
 - 여기서 스토어는 /files

HTML FORM 사용

- HTML FORM은 **GET, POST**만 지원
- AJAX 같은 기술을 사용해서 해결 가능 -> 회원 API 참고
- 여기서는 순수 HTML, HTML FORM 이야기
- GET, POST만 지원하므로 제약이 있음

HTML FORM 사용

- 회원 목록 /members -> **GET**
- 회원 등록 폼 /members/new -> **GET**
- 회원 등록 /members/new, /members -> **POST**
- 회원 조회 /members/{id} -> **GET**
- 회원 수정 폼 /members/{id}/edit -> **GET**
- 회원 수정 /members/{id}/edit, /members/{id} -> **POST**
- 회원 삭제 /members/{id}/delete -> **POST**

HTML FORM 사용

- HTML FORM은 **GET, POST**만 지원
- 컨트롤 URI
 - GET, POST만 지원하므로 제약이 있음
 - 이런 제약을 해결하기 위해 동사로 된 리소스 경로 사용
 - POST의 /new, /edit, /delete가 컨트롤 URI
 - HTTP 메서드로 해결하기 애매한 경우 사용(HTTP API 포함)

정리

- **HTTP API - 컬렉션**
 - POST 기반 등록
 - 서버가 리소스 URI 결정
- **HTTP API - 스토어**
 - PUT 기반 등록
 - 클라이언트가 리소스 URI 결정
- **HTML FORM 사용**
 - 순수 HTML + HTML form 사용
 - GET, POST만 지원

정리

참고하면 좋은 URI 설계 개념

- **문서(document)**

- 단일 개념(파일 하나, 객체 인스턴스, 데이터베이스 row)
 - 예) /members/100, /files/star.jpg

- **컬렉션(collection)**

- 서버가 관리하는 리소스 디렉터리
 - 서버가 리소스의 URI를 생성하고 관리
 - 예) /members

- **스토어(store)**

- 클라이언트가 관리하는 자원 저장소
 - 클라이언트가 리소스의 URI를 알고 관리
 - 예) /files

- **컨트롤러(controller), 컨트롤 URI**

- 문서, 컬렉션, 스토어로 해결하기 어려운 추가 프로세스 실행
 - 동사를 직접 사용
 - 예) /members/{id}/delete

HTTP 상태코드

상태 코드

클라이언트가 보낸 요청의 처리 상태를 응답에서 알려주는 기능

- 1xx (Informational): 요청이 수신되어 처리중
- 2xx (Successful): 요청 정상 처리
- 3xx (Redirection): 요청을 완료하려면 추가 행동이 필요
- 4xx (Client Error): 클라이언트 오류, 잘못된 문법등으로 서버가 요청을 수행할 수 없음
- 5xx (Server Error): 서버 오류, 서버가 정상 요청을 처리하지 못함

만약 모르는 상태 코드가 나타나면?

- 클라이언트가 인식할 수 없는 상태코드를 서버가 반환하면?
- 클라이언트는 상위 상태코드로 해석해서 처리
- 미래에 새로운 상태 코드가 추가되어도 클라이언트를 변경하지 않아도 됨
- 예)
 - 299 ??? -> 2xx (Successful)
 - 451 ??? -> 4xx (Client Error)
 - 599 ??? -> 5xx (Server Error)

1xx (Informational)

요청이 수신되어 처리중

- 거의 사용하지 않으므로 생략

2XX - 성공

2xx (Successful)

클라이언트의 요청을 성공적으로 처리

- 200 OK
- 201 Created
- 202 Accepted
- 204 No Content

200 OK

요청 성공

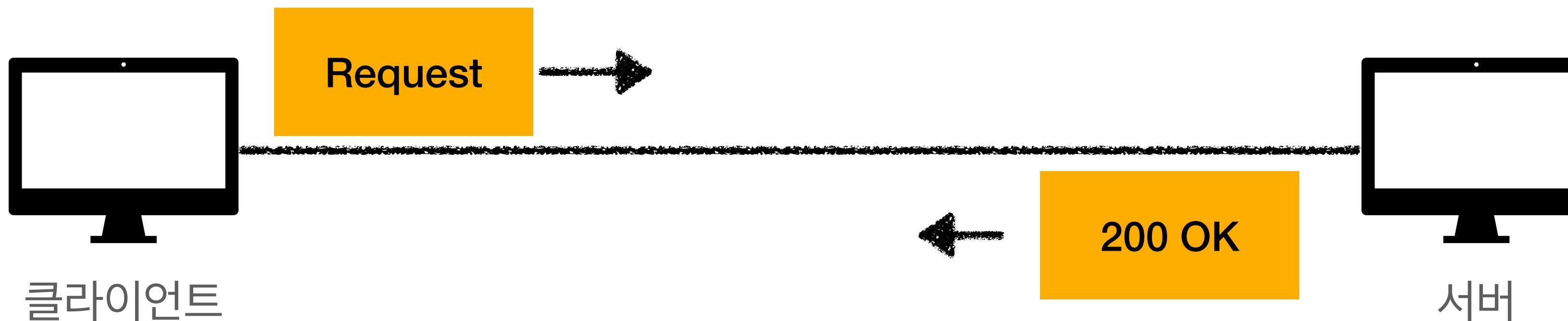
요청

```
GET /members/100 HTTP/1.1  
Host: localhost:8080
```

응답

```
HTTP/1.1 200 OK  
Content-Type: application/json  
Content-Length: 34
```

```
{  
    "username": "young",  
    "age": 20  
}
```



201 Created

요청 성공해서 새로운 리소스가 생성됨

생성된 리소스는 응답의 Location 헤더 필드로 식별

요청

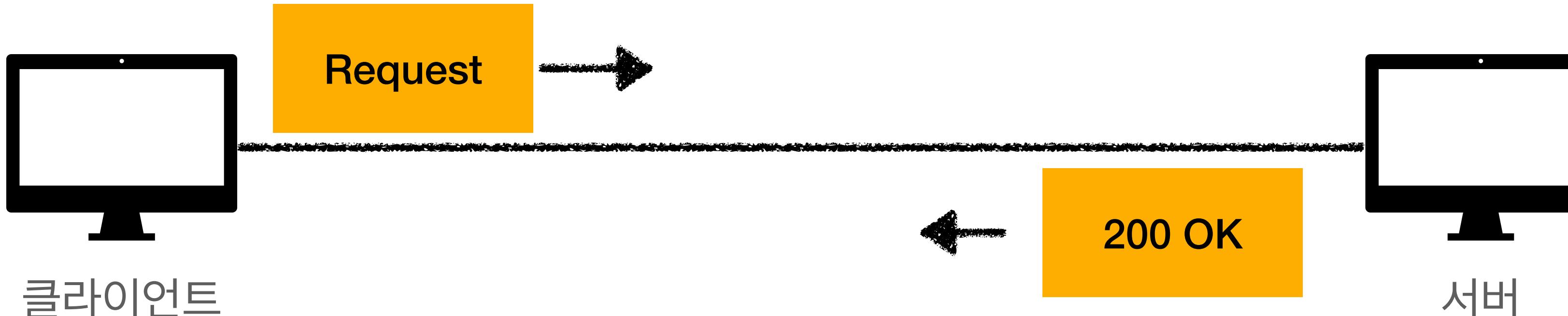
```
POST /members HTTP/1.1  
Content-Type: application/json
```

```
{  
  "username": "young",  
  "age": 20  
}
```

응답

```
HTTP/1.1 201 Created  
Content-Type: application/json  
Content-Length: 34  
Location: /members/100
```

```
{  
  "username": "young",  
  "age": 20  
}
```



202 Accepted

요청이 접수되었으나 처리가 완료되지 않았음

- 배치 처리 같은 곳에서 사용
- 예) 요청 접수 후 1시간 뒤에 배치 프로세스가 요청을 처리함

204 No Content

서버가 요청을 성공적으로 수행했지만, 응답 페이로드 본문에 보낼 데이터가 없음

- 예) 웹 문서 편집기에서 save 버튼
- save 버튼의 결과로 아무 내용이 없어도 된다.
- save 버튼을 눌러도 같은 화면을 유지해야 한다.
- 결과 내용이 없어도 204 메시지(2xx)만으로 성공을 인식할 수 있다.

3XX - 리다이렉션

3xx (Redirection)

요청을 완료하기 위해 유저 에이전트의 추가 조치 필요

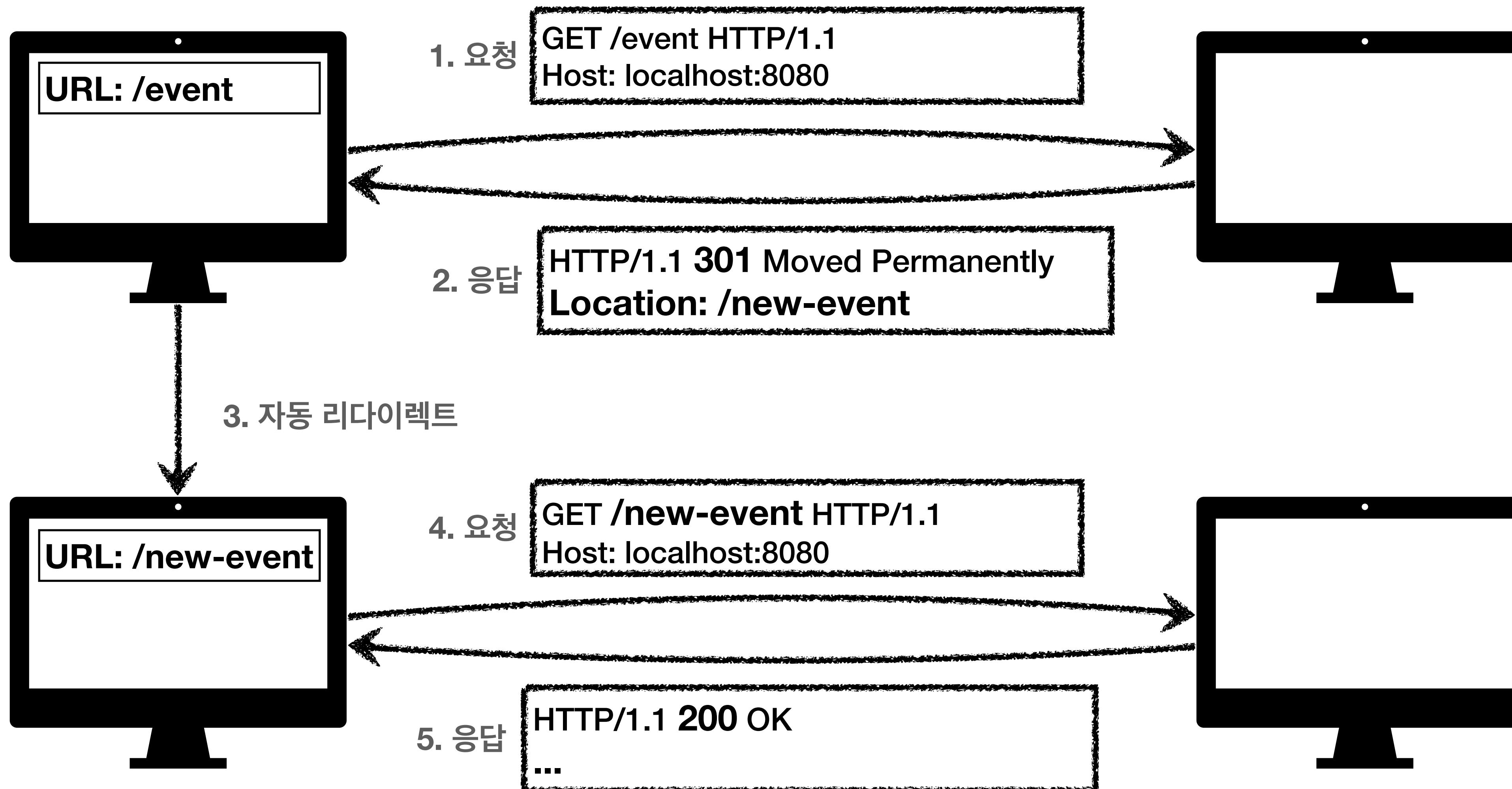
- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 303 See Other
- 304 Not Modified
- 307 Temporary Redirect
- 308 Permanent Redirect

리다이렉션 이해

- 웹 브라우저는 3xx 응답의 결과에 Location 헤더가 있으면, Location 위치로 자동 이동 (리다이렉트)

리다이렉션 이해

자동 리다이렉트 흐름



리다이렉션 이해

종류

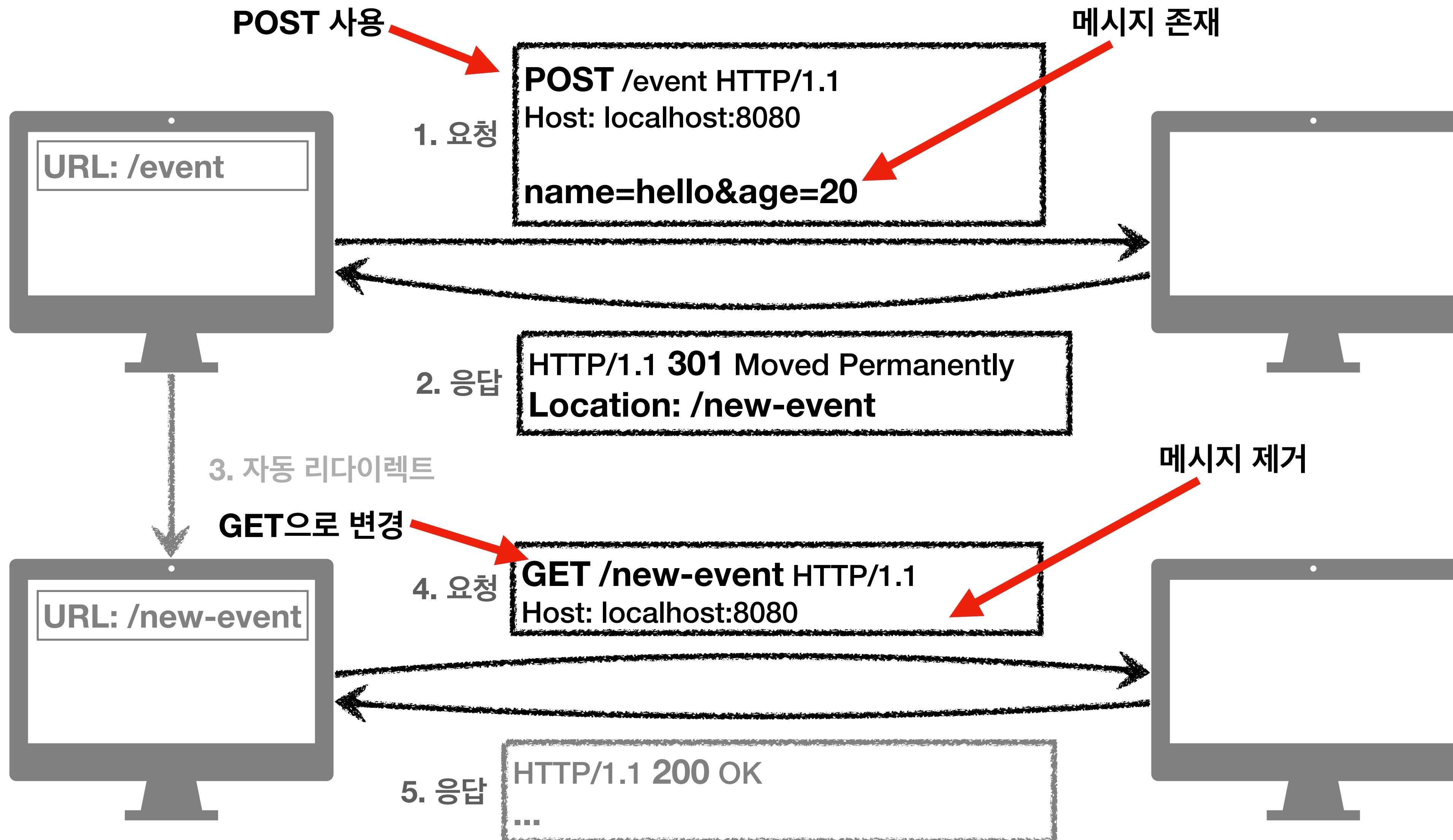
- 영구 리다이렉션 - 특정 리소스의 URI가 영구적으로 이동
 - 예) /members -> /users
 - 예) /event -> /new-event
- 일시 리다이렉션 - 일시적인 변경
 - 주문 완료 후 주문 내역 화면으로 이동
 - PRG: Post/Redirect/Get
- 특수 리다이렉션
 - 결과 대신 캐시를 사용

영구 리다이렉션

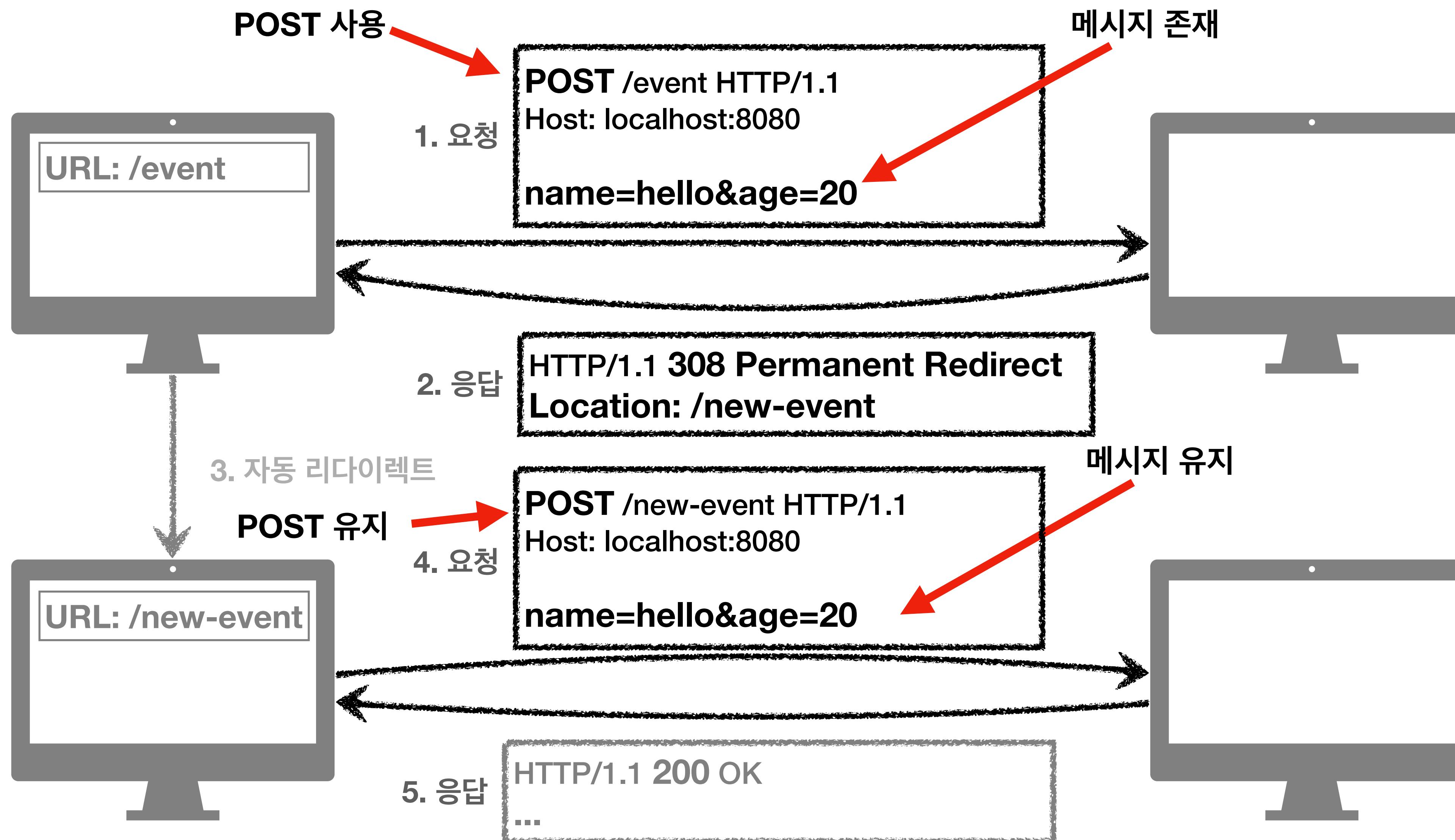
301, 308

- 리소스의 URI가 영구적으로 이동
- 원래의 URL를 사용X, 검색 엔진 등에서도 변경 인지
- **301 Moved Permanently**
 - 리다이렉트시 요청 메서드가 GET으로 변하고, 본문이 제거될 수 있음(MAY)
- **308 Permanent Redirect**
 - 301과 기능은 같음
 - 리다이렉트시 요청 메서드와 본문 유지(처음 POST를 보내면 리다이렉트도 POST 유지)

영구 리다이렉션 - 301



영구 리다이렉션 - 308



일시적인 리다이렉션

302, 307, 303

- 리소스의 URI가 일시적으로 변경
- 따라서 검색 엔진 등에서 URL을 변경하면 안됨
- **302 Found**
 - 리다이렉트시 요청 메서드가 **GET**으로 변하고, 본문이 제거될 수 있음(**MAY**)
- **307 Temporary Redirect**
 - 302와 기능은 같음
 - 리다이렉트시 요청 메서드와 본문 유지(요청 메서드를 변경하면 안된다. **MUST NOT**)
- **303 See Other**
 - 302와 기능은 같음
 - 리다이렉트시 요청 메서드가 **GET**으로 변경

PRG: Post/Redirect/Get

일시적인 리다이렉션 - 예시

- POST로 주문후에 웹 브라우저를 새로고침하면?
- 새로고침은 다시 요청
- 중복 주문이 될 수 있다.

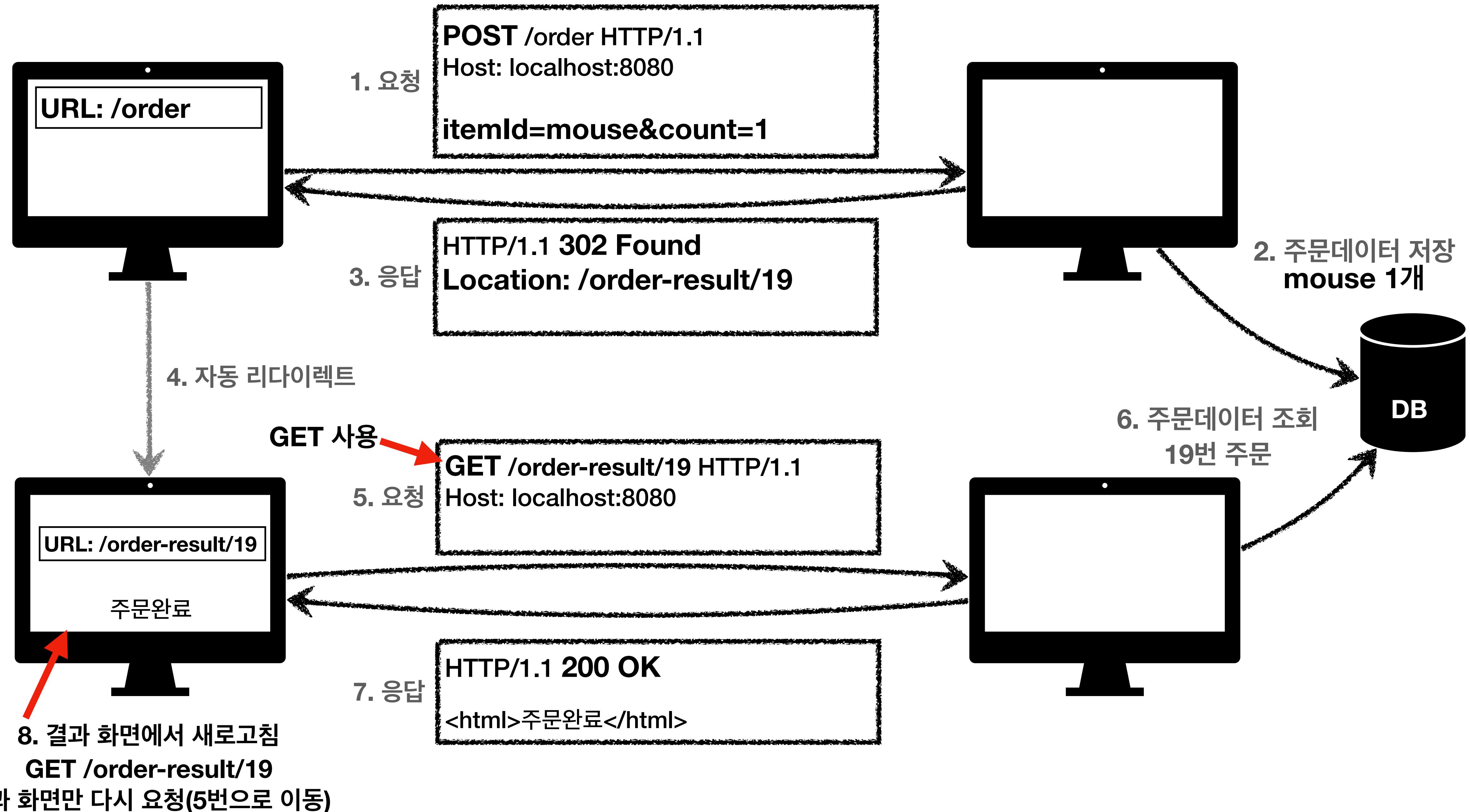


PRG: Post/Redirect/Get

일시적인 리다이렉션 - 예시

- POST로 주문후에 새로 고침으로 인한 중복 주문 방지
- POST로 주문후에 주문 결과 화면을 GET 메서드로 리다이렉트
- 새로고침해도 결과 화면을 GET으로 조회
- 중복 주문 대신에 결과 화면만 GET으로 다시 요청

PRG: Post/Redirect/Get



PRG: Post/Redirect/Get

일시적인 리다이렉션 - 예시

- PRG 이후 리다이렉트
 - URL이 이미 POST -> GET으로 리다이렉트 됨
 - 새로 고침 해도 GET으로 결과 화면만 조회

그래서 뭘 써야 하나요?

302, 307, 303

- 잠깐 정리
 - 302 Found -> GET으로 변할 수 있음
 - 307 Temporary Redirect -> 메서드가 변하면 안됨
 - 303 See Other -> 메서드가 GET으로 변경
- 역사
 - 처음 302 스펙의 의도는 HTTP 메서드를 유지하는 것
 - 그런데 웹 브라우저들이 대부분 GET으로 바꾸어버림(일부는 다르게 동작)
 - 그래서 모호한 302를 대신하는 명확한 307, 303이 등장함(301 대응으로 308도 등장)
- 현실
 - 307, 303을 권장하지만 현실적으로 이미 많은 애플리케이션 라이브러리들이 302를 기본값으로 사용
 - 자동 리다이렉션시에 GET으로 변해도 되면 그냥 302를 사용해도 큰 문제 없음

기타 리다이렉션

300 304

- 300 Multiple Choices: 안쓴다.
- 304 Not Modified
 - 캐시를 목적으로 사용
 - 클라이언트에게 리소스가 수정되지 않았음을 알려준다. 따라서 클라이언트는 로컬PC에 저장된 캐시를 재사용한다. (캐시로 리다이렉트 한다.)
- 304 응답은 응답에 메시지 바디를 포함하면 안된다. (로컬 캐시를 사용해야 하므로)
- 조건부 GET, HEAD 요청시 사용

4xx - 클라이언트 오류

5xx - 서버 오류

4xx (Client Error)

클라이언트 오류

- 클라이언트의 요청에 잘못된 문법등으로 서버가 요청을 수행할 수 없음
- 오류의 원인이 클라이언트에 있음
- 중요! 클라이언트가 이미 잘못된 요청, 데이터를 보내고 있기 때문에, 똑같은 재시도가 실패함

400 Bad Request

클라이언트가 잘못된 요청을 해서 서버가 요청을 처리할 수 없음

- 요청 구문, 메시지 등등 오류
- 클라이언트는 요청 내용을 다시 검토하고, 보내야함
- 예) 요청 파라미터가 잘못되거나, API 스펙이 맞지 않을 때

401 Unauthorized

클라이언트가 해당 리소스에 대한 인증이 필요함

- 인증(Authentication) 되지 않음
- 401 오류 발생시 응답에 WWW-Authenticate 헤더와 함께 인증 방법을 설명
- 참고
 - 인증(Authentication): 본인이 누구인지 확인, (로그인)
 - 인가(Authorization): 권한부여 (ADMIN 권한처럼 특정 리소스에 접근할 수 있는 권한, 인증이 있어야 인가가 있음)
- 오류 메시지가 Unauthorized 이지만 인증 되지 않음 (이름이 아쉬움)

403 Forbidden

서버가 요청을 이해했지만 승인을 거부함

- 주로 인증 자격 증명은 있지만, 접근 권한이 불충분한 경우
- 예) 어드민 등급이 아닌 사용자가 로그인은 했지만, 어드민 등급의 리소스에 접근하는 경우

404 Not Found

요청 리소스를 찾을 수 없음

- 요청 리소스가 서버에 없음
- 또는 클라이언트가 권한이 부족한 리소스에 접근할 때 해당 리소스를 숨기고 싶을 때

5xx (Server Error)

서버 오류

- 서버 문제로 오류 발생
- 서버에 문제가 있기 때문에 재시도 하면 성공할 수도 있음(복구가 되거나 등등)

500 Internal Server Error

서버 문제로 오류 발생, 애매하면 500 오류

- 서버 내부 문제로 오류 발생
- 애매하면 500 오류

503 Service Unavailable

서비스 이용 불가

- 서버가 일시적인 과부하 또는 예정된 작업으로 잠시 요청을 처리할 수 없음
- Retry-After 헤더 필드로 얼마뒤에 복구되는지 보낼 수도 있음

HTTP 헤더1

일반 헤더

HTTP 헤더 개요

HTTP 헤더

- header-field = field-name ":" OWS field-value OWS (OWS:띄어쓰기 허용)
- field-name은 대소문자 구분 없음

```
GET /search?q=hello&hl=ko HTTP/1.1  
Host: www.google.com
```

```
HTTP/1.1 200 OK  
Content-Type: text/html; charset=UTF-8  
Content-Length: 3423
```

```
<html>  
  <body>...</body>  
</html>
```

HTTP 헤더 용도

- HTTP 전송에 필요한 모든 부가정보
- 예) 메시지 바디의 내용, 메시지 바디의 크기, 압축, 인증, 요청 클라이언트, 서버 정보, 캐시 관리 정보...
- 표준 헤더가 너무 많음
 - https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- 필요시 임의의 헤더 추가 가능
 - helloworld: hihi

HTTP/1.1 200 OK

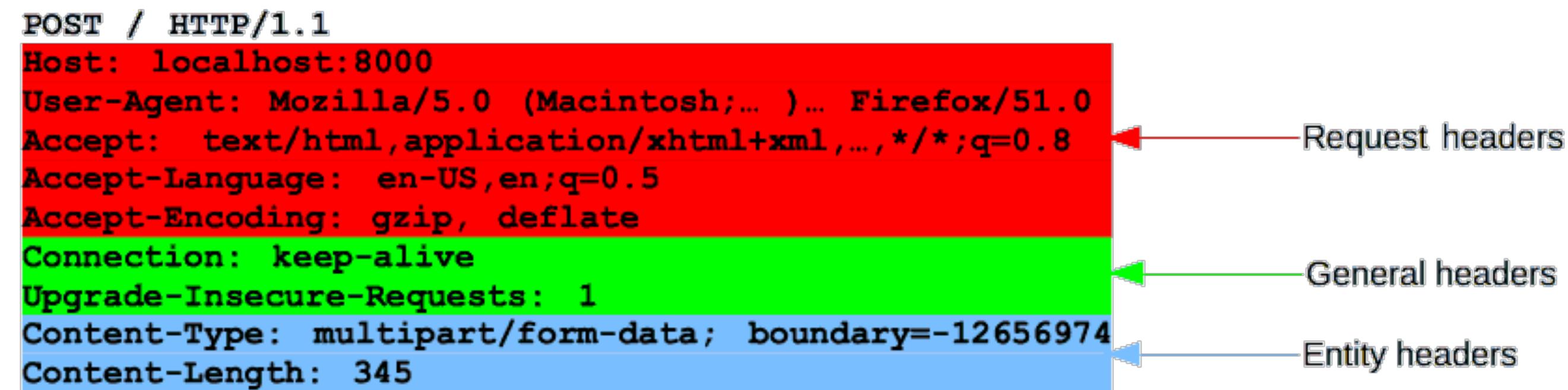
Content-Type: text/html; charset=UTF-8

Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

HTTP 헤더

분류 - RFC2616(과거)



-12656974
(more data)

출처: <https://developer.mozilla.org/docs/Web/HTTP/Messages>

- 헤더 분류
 - **General 헤더**: 메시지 전체에 적용되는 정보, 예) Connection: close
 - **Request 헤더**: 요청 정보, 예) User-Agent: Mozilla/5.0 (Macintosh; ..)
 - **Response 헤더**: 응답 정보, 예) Server: Apache
 - **Entity 헤더**: 엔티티 바디 정보, 예) Content-Type: text/html, Content-Length: 3423

HTTP BODY

message body - RFC2616(과거)

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 3423

메시지 본문

```
<html>
  <body>...</body>
</html>
```

엔티티 헤더

엔티티 본문

- 메시지 본문(message body)은 엔티티 본문(entity body)을 전달하는데 사용
- 엔티티 본문은 요청이나 응답에서 전달할 실제 데이터
- 엔티티 헤더는 엔티티 본문의 데이터를 해석할 수 있는 정보 제공
 - 데이터 유형(html, json), 데이터 길이, 압축 정보 등등

그런데...

HTTP 표준

1999년 RFC2616 

폐기됨

2014년 RFC7230~7235 등장

RFC723x 변화

- 엔티티(Entity) -> 표현(Representation)
- Representation = representation Metadata + Representation Data
- 표현 = 표현 메타데이터 + 표현 데이터

HTTP BODY

message body - RFC7230(최신)

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 3423

메시지 본문

```
<html>
  <body>...</body>
</html>
```

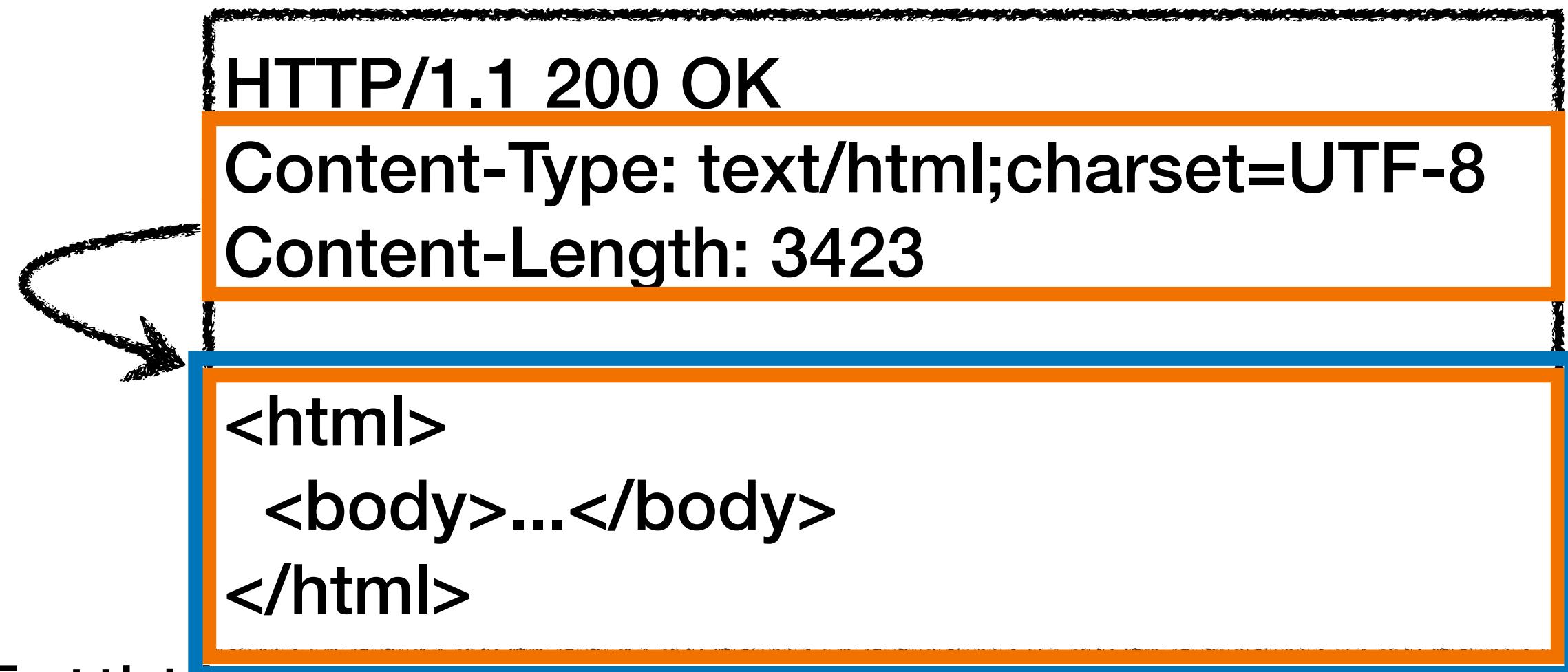
표현 헤더

표현 데이터

- 메시지 본문(message body)을 통해 표현 데이터 전달
- 메시지 본문 = 페이로드(payload)
- 표현은 요청이나 응답에서 전달할 실제 데이터
- 표현 헤더는 표현 데이터를 해석할 수 있는 정보 제공
 - 데이터 유형(html, json), 데이터 길이, 압축 정보 등등
- 참고: 표현 헤더는 표현 메타데이터와, 페이로드 메시지를 구분해야 하지만, 여기서는 생략

표현

- Content-Type: 표현 데이터의 형식
- Content-Encoding: 표현 데이터의 압축 방식
- Content-Language: 표현 데이터의 자연 언어
- Content-Length: 표현 데이터의 길이
- 표현 헤더는 전송, 응답 둘다 사용



Content-Type

표현 데이터의 형식 설명

- 미디어 타입, 문자 인코딩
- 예)
 - text/html; charset=utf-8
 - application/json
 - image/png

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 3423

```
<html>
  <body>...</body>
</html>
```

HTTP/1.1 200 OK

Content-Type: application/json

Content-Length: 16

```
{"data":"hello"}
```

Content-Encoding

표현 데이터 인코딩

- 표현 데이터를 압축하기 위해 사용
- 데이터를 전달하는 곳에서 압축 후 인코딩 헤더 추가
- 데이터를 읽는 쪽에서 인코딩 헤더의 정보로 압축 해제
- 예)
 - gzip
 - deflate
 - identity

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Encoding: gzip

Content-Length: 521

Ikj123kljoiasudlkjaweioluywlnfdo912u34lj
ko98udjkl

Content-Language

표현 데이터의 자연 언어

- 표현 데이터의 자연 언어를 표현
- 예)
 - ko
 - en
 - en-US

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Language: ko

Content-Length: 521

<html>

안녕하세요.

</html>

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Language: en

Content-Length: 521

<html>

hello

</html>

Content-Length

표현 데이터의 길이

- 바이트 단위
- Transfer-Encoding(전송 코딩)을 사용하면 Content-Length를 사용하면 안됨

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

Content-Length: 5

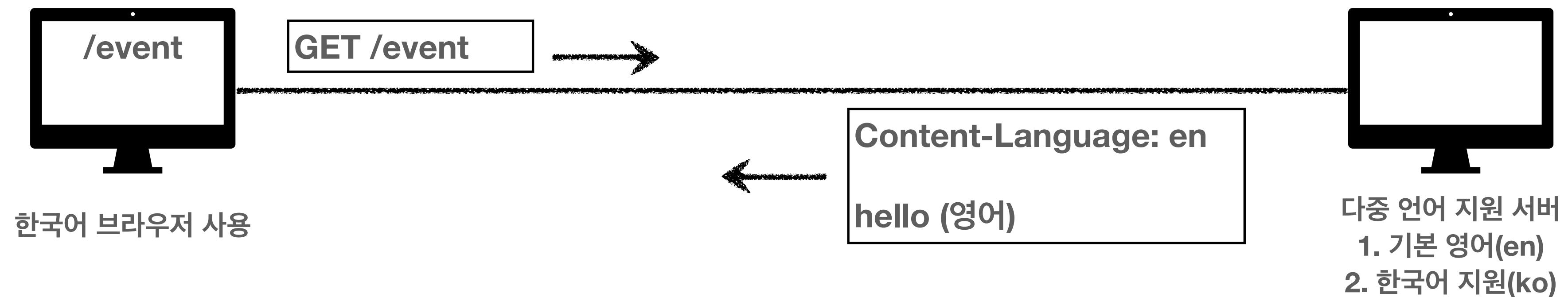
hello

협상(콘텐츠 네고시에이션)

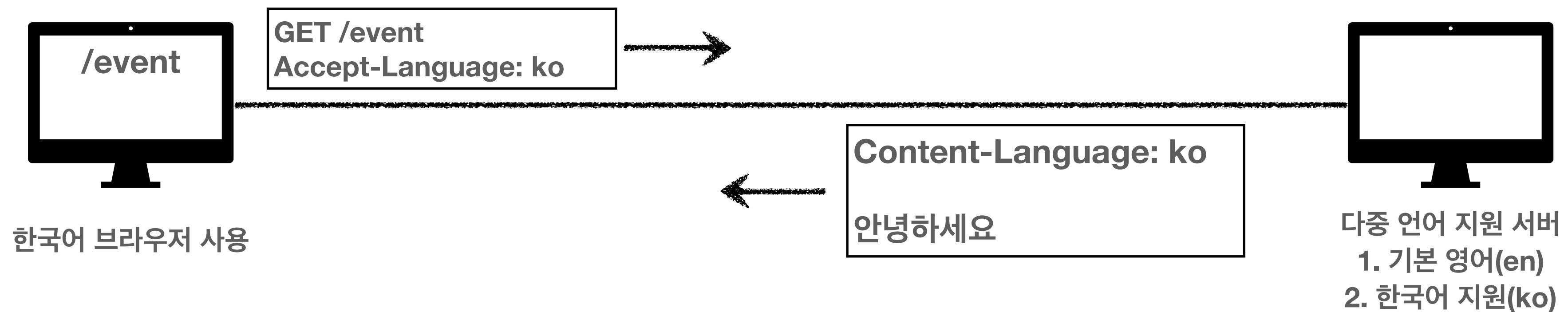
클라이언트가 선호하는 표현 요청

- Accept: 클라이언트가 선호하는 미디어 타입 전달
- Accept-Charset: 클라이언트가 선호하는 문자 인코딩
- Accept-Encoding: 클라이언트가 선호하는 압축 인코딩
- Accept-Language: 클라이언트가 선호하는 자연 언어
- 협상 헤더는 요청시에만 사용

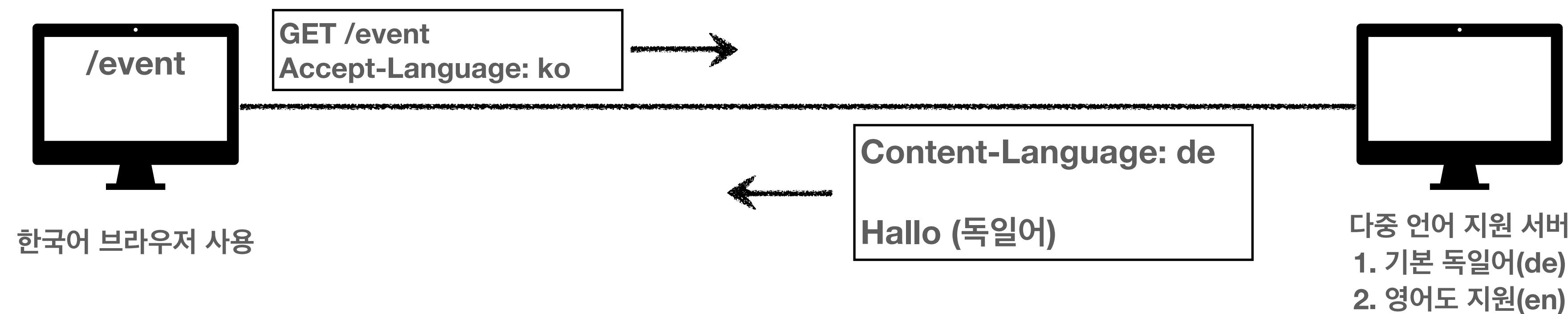
Accept-Language 적용 전



Accept-Language 적용 후



Accept-Language 복잡한 예시



협상과 우선순위1

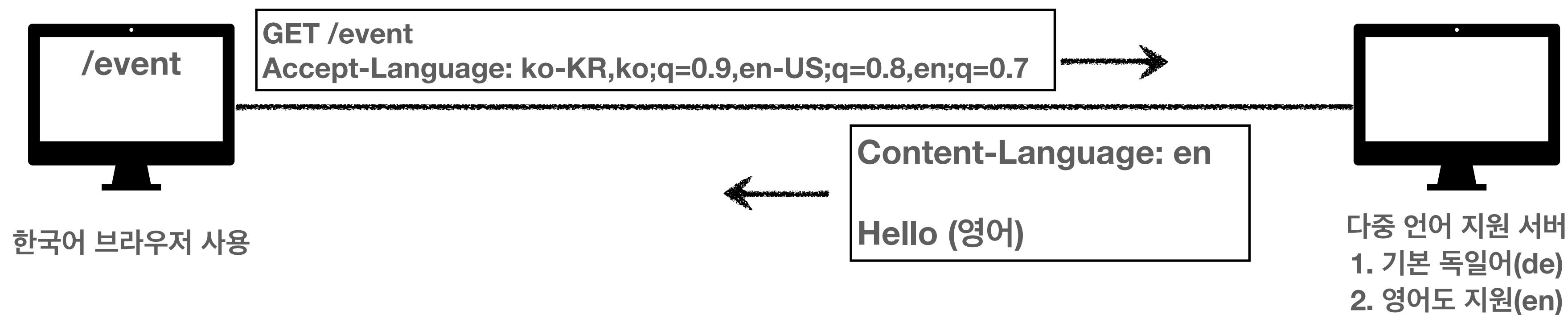
Quality Values(q)

GET /event

Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7

- Quality Values(q) 값 사용
- 0~1, 클수록 높은 우선순위
- 생략하면 1
- Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
 - 1. ko-KR;q=1 (q생략)
 - 2. ko;q=0.9
 - 3. en-US;q=0.8
 - 4. en;q=0.7

Accept-Language 복잡한 예시



협상과 우선순위2

Quality Values(q)

GET /event

Accept: **text/*, text/plain, text/plain;format=flowed, */***

- 구체적인 것이 우선한다.
- Accept: **text/*, text/plain, text/plain;format=flowed, */***
 1. text/plain;format=flowed
 2. text/plain
 3. text/*
 4. */*

협상과 우선순위3

Quality Values(q)

- 구체적인 것을 기준으로 미디어 타입을 맞춘다.
- Accept: **text/*;q=0.3, text/html;q=0.7, text/html;level=1, text/html;level=2;q=0.4, */*;q=0.5**

Media Type	Quality
text/html;level=1	1
text/html	0.7
text/plain	0.3
image/jpeg	0.5
text/html;level=2	0.4
text/html;level=3	0.7

전송 방식

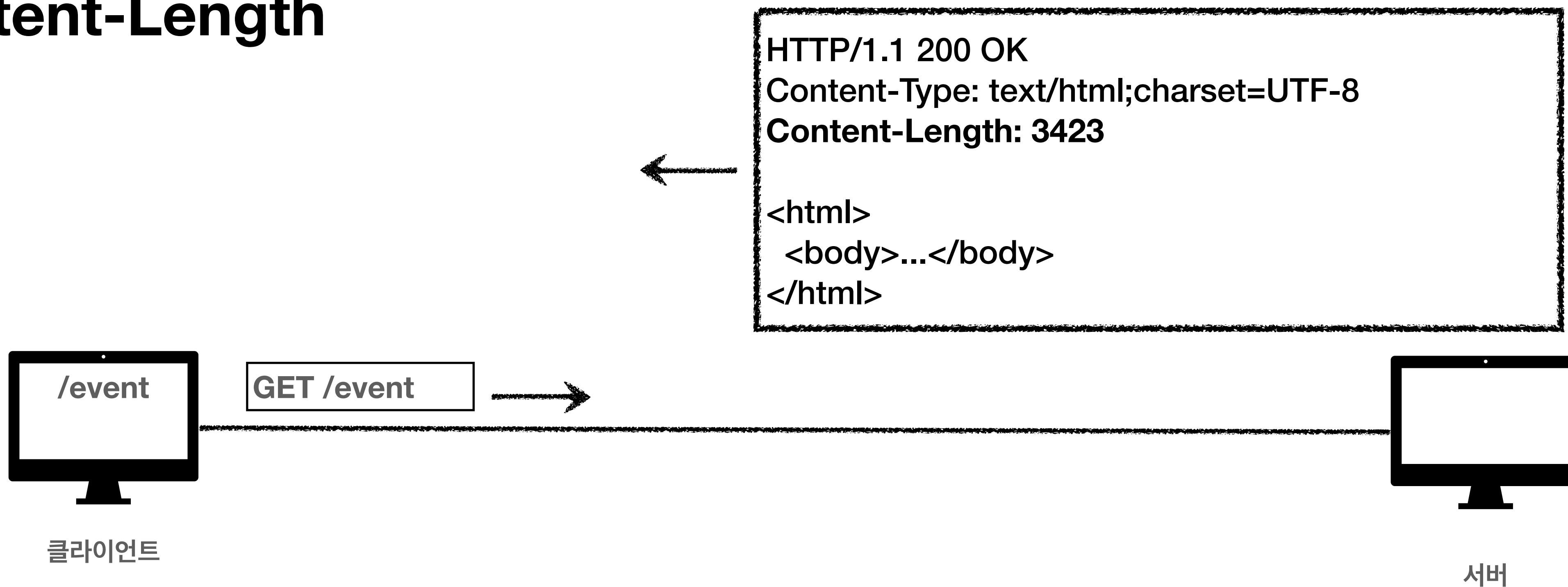
- Transfer-Encoding
- Range, Content-Range

전송 방식 설명

- 단순 전송
- 압축 전송
- 분할 전송
- 범위 전송

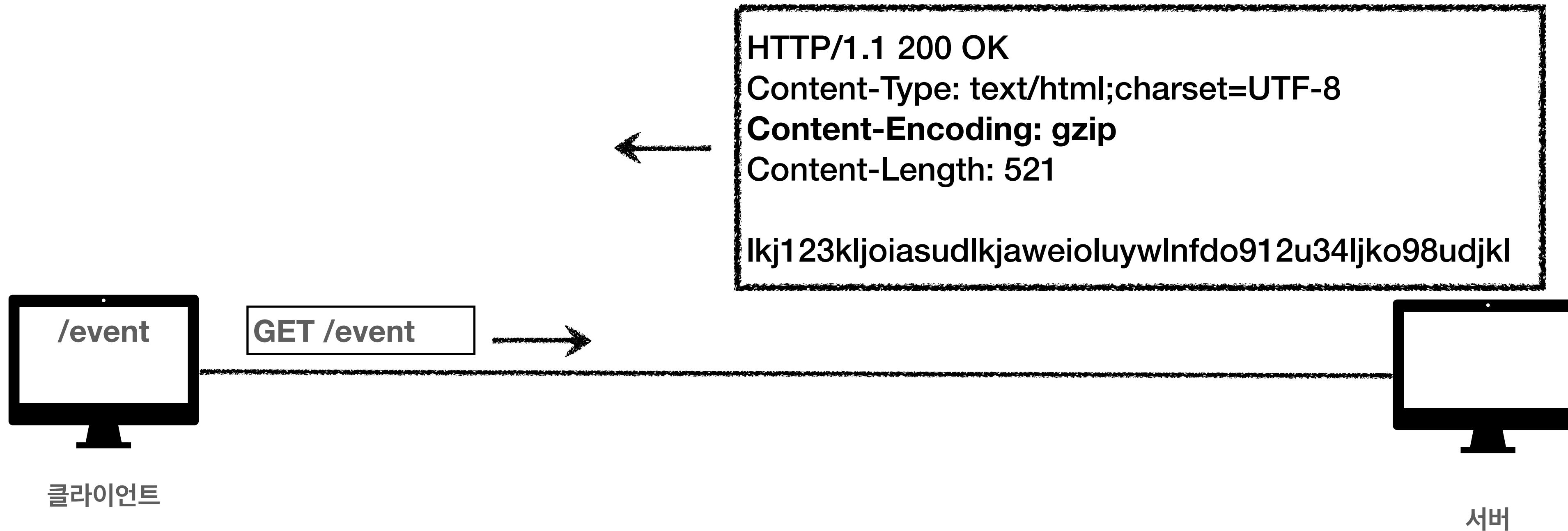
단순 전송

Content-Length



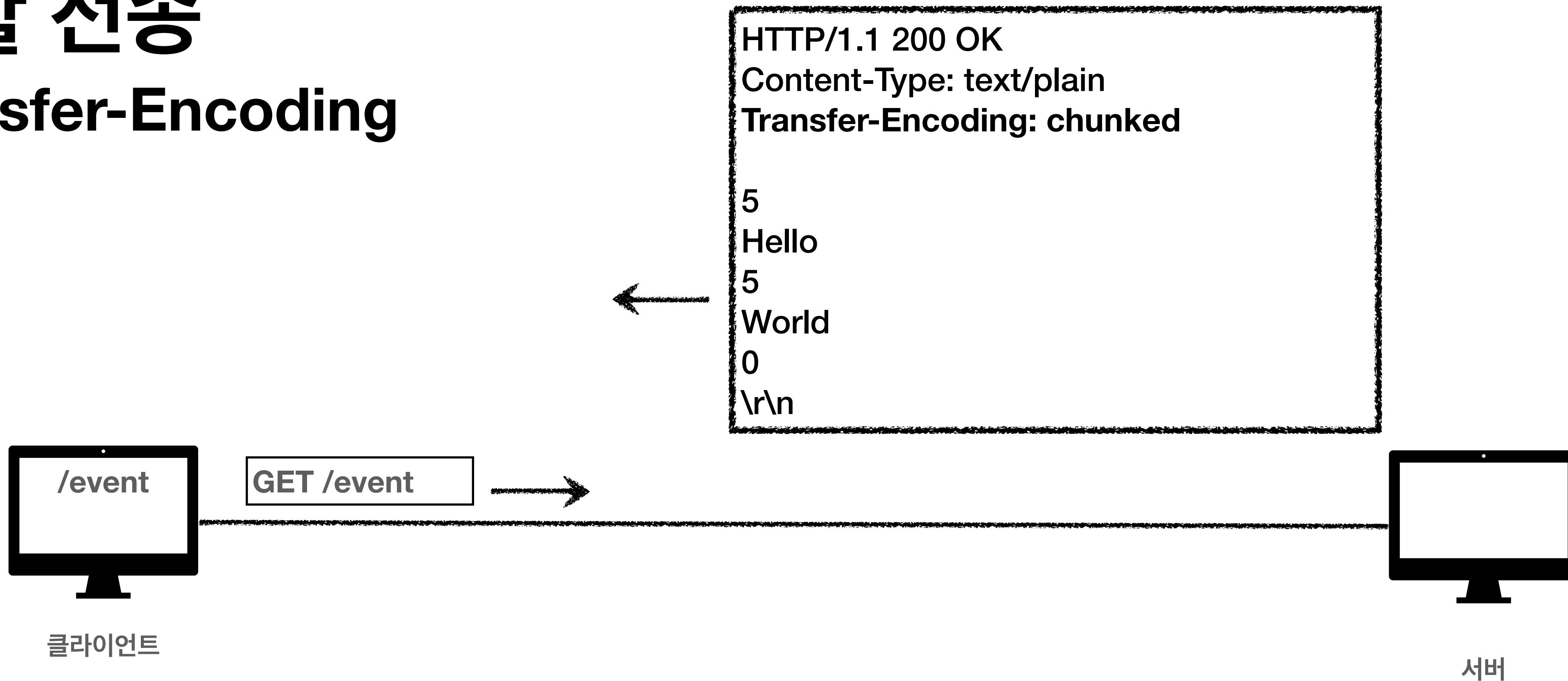
압축 전송

Content-Encoding



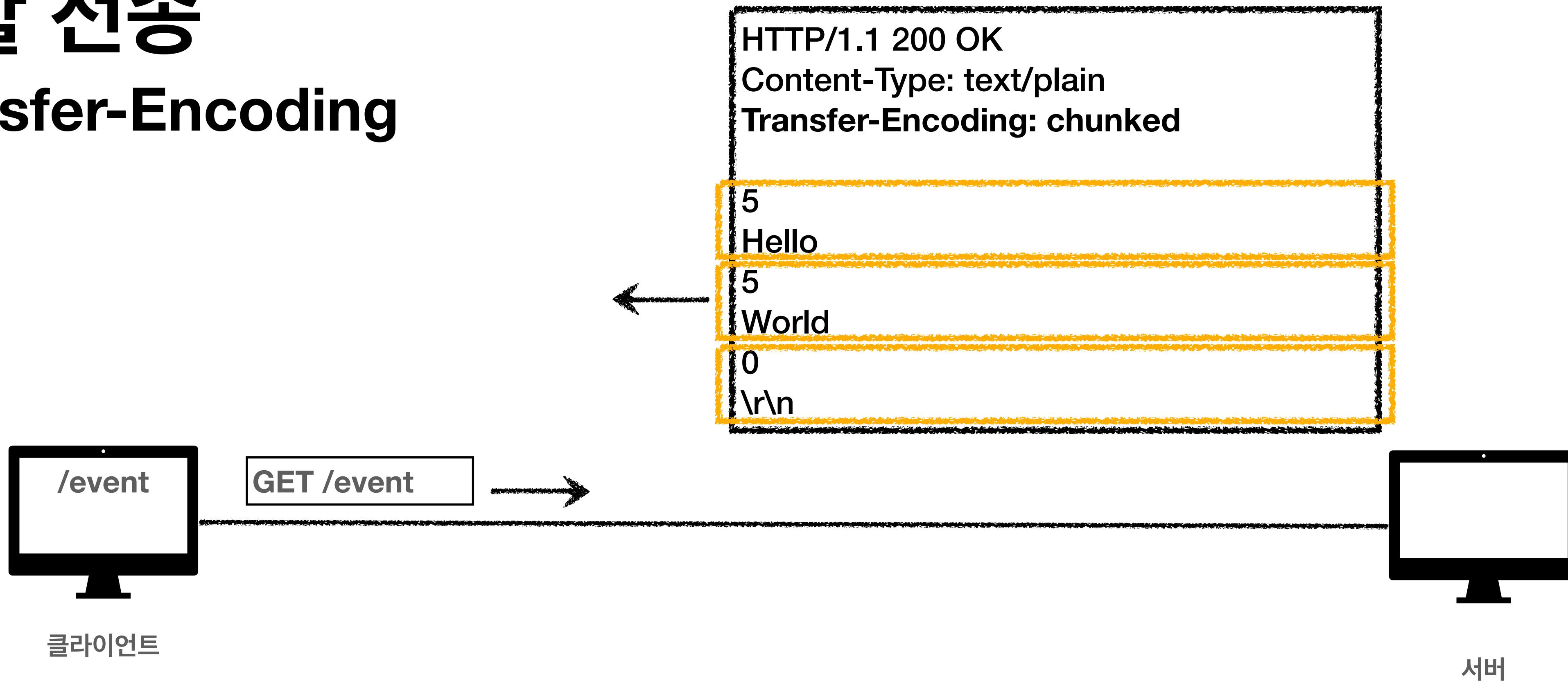
분할 전송

Transfer-Encoding



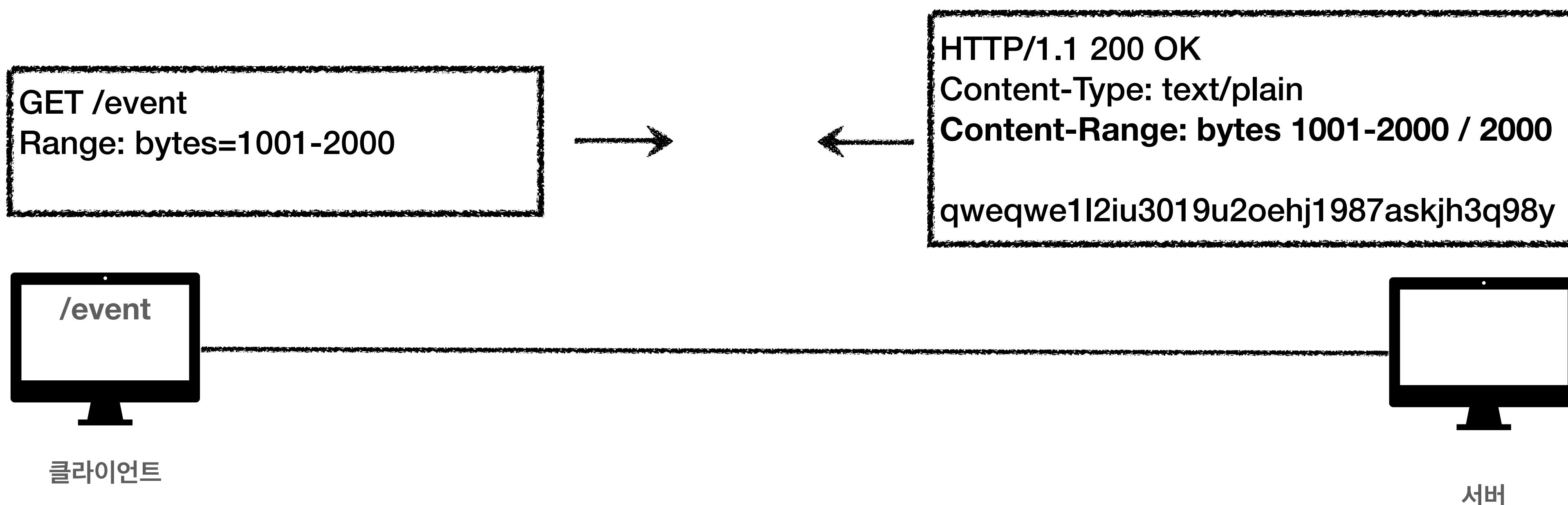
분할 전송

Transfer-Encoding



범위 전송

Range, Content-Range



일반 정보

- From: 유저 에이전트의 이메일 정보
- Referer: 이전 웹 페이지 주소
- User-Agent: 유저 에이전트 애플리케이션 정보
- Server: 요청을 처리하는 오리진 서버의 소프트웨어 정보
- Date: 메시지가 생성된 날짜

From

유저 에이전트의 이메일 정보

- 일반적으로 잘 사용되지 않음
- 검색 엔진 같은 곳에서, 주로 사용
- 요청에서 사용

Referer

이전 웹 페이지 주소

- 현재 요청된 페이지의 이전 웹 페이지 주소
- A -> B로 이동하는 경우 B를 요청할 때 Referer: A 를 포함해서 요청
- Referer를 사용해서 유입 경로 분석 가능
- 요청에서 사용
- 참고: referer는 단어referrer의 오타

User-Agent

유저 에이전트 애플리케이션 정보

- user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.183 Safari/537.36
- 클라이언트의 애플리케이션 정보(웹 브라우저 정보, 등등)
- 통계 정보
- 어떤 종류의 브라우저에서 장애가 발생하는지 파악 가능
- 요청에서 사용

Server

요청을 처리하는 ORIGIN 서버의 소프트웨어 정보

- Server: Apache/2.2.22 (Debian)
- server: nginx
- 응답에서 사용

Date

메시지가 발생한 날짜와 시간

- Date: Tue, 15 Nov 1994 08:12:31 GMT
- 응답에서 사용

특별한 정보

- Host: 요청한 호스트 정보(도메인)
- Location: 페이지 리다이렉션
- Allow: 허용 가능한 HTTP 메서드
- Retry-After: 유저 에이전트가 다음 요청을 하기까지 기다려야 하는 시간

Host

요청한 호스트 정보(도메인)

```
GET /search?q=hello&hl=ko HTTP/1.1  
Host: www.google.com
```

- 요청에서 사용
- 필수
- 하나의 서버가 여러 도메인을 처리해야 할 때
- 하나의 IP 주소에 여러 도메인이 적용되어 있을 때

Host

요청한 호스트 정보(도메인)

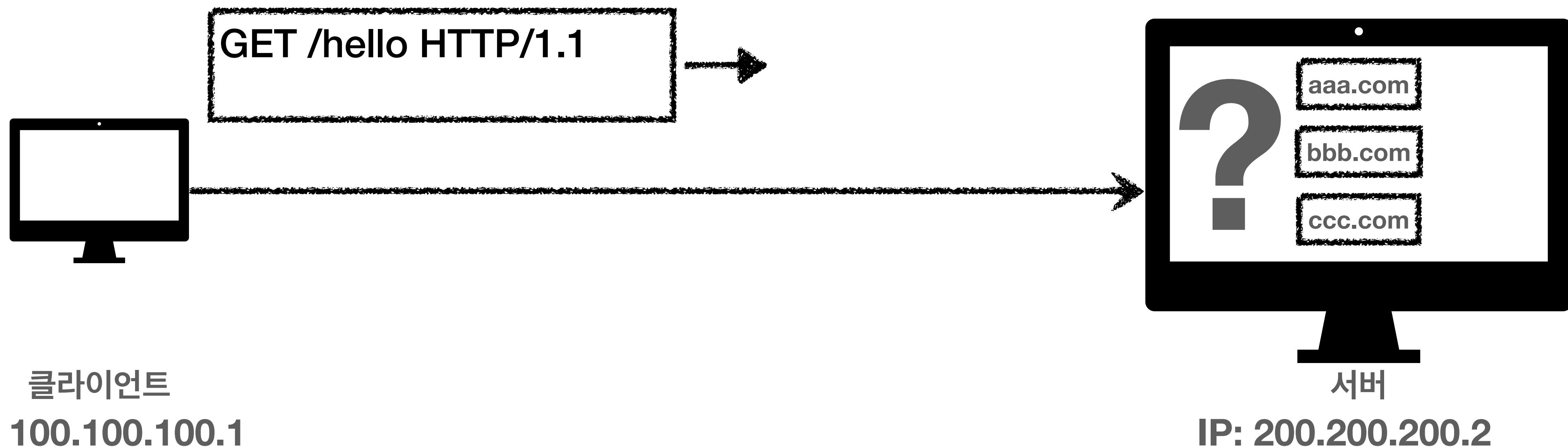
가상호스트를 통해 여러 도메인을 한번에 처리할 수 있는 서버
실제 애플리케이션이 여러개 구동될 수 있다.



IP: 200.200.200.2

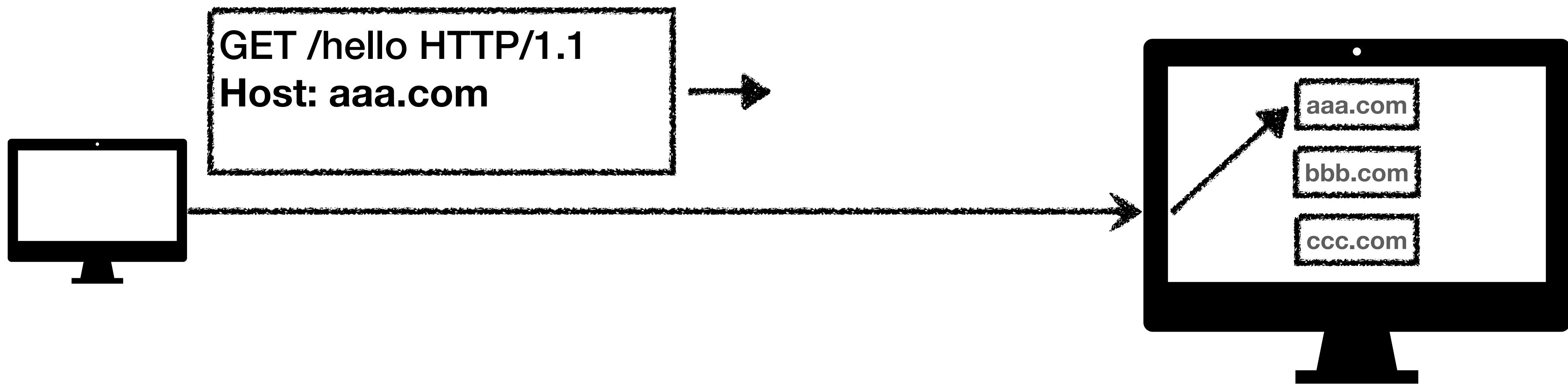
Host

요청한 호스트 정보(도메인)



Host

요청한 호스트 정보(도메인)



클라이언트

IP: 100.100.100.1

서버

IP: 200.200.200.2

Location

페이지 리다이렉션

- 웹 브라우저는 3xx 응답의 결과에 Location 헤더가 있으면, Location 위치로 자동 이동 (리다이렉트)
- 응답코드 3xx에서 설명
- 201 (Created): Location 값은 요청에 의해 생성된 리소스 URI
- 3xx (Redirection): Location 값은 요청을 자동으로 리디렉션하기 위한 대상 리소스를 가리킴

Allow

허용 가능한 HTTP 메서드

- 405 (Method Not Allowed) 에서 응답에 포함해야함
- Allow: GET, HEAD, PUT

Retry-After

유저 에이전트가 다음 요청을 하기까지 기다려야 하는 시간

- 503 (Service Unavailable): 서비스가 언제까지 불능인지 알려줄 수 있음
- Retry-After: Fri, 31 Dec 1999 23:59:59 GMT (날짜 표기)
- Retry-After: 120 (초단위 표기)

인증

- Authorization: 클라이언트 인증 정보를 서버에 전달
- WWW-Authenticate: 리소스 접근시 필요한 인증 방법 정의

Authorization

클라이언트 인증 정보를 서버에 전달

- Authorization: Basic xxxxxxxxxxxxxxxxx

WWW-Authenticate

리소스 접근시 필요한 인증 방법 정의

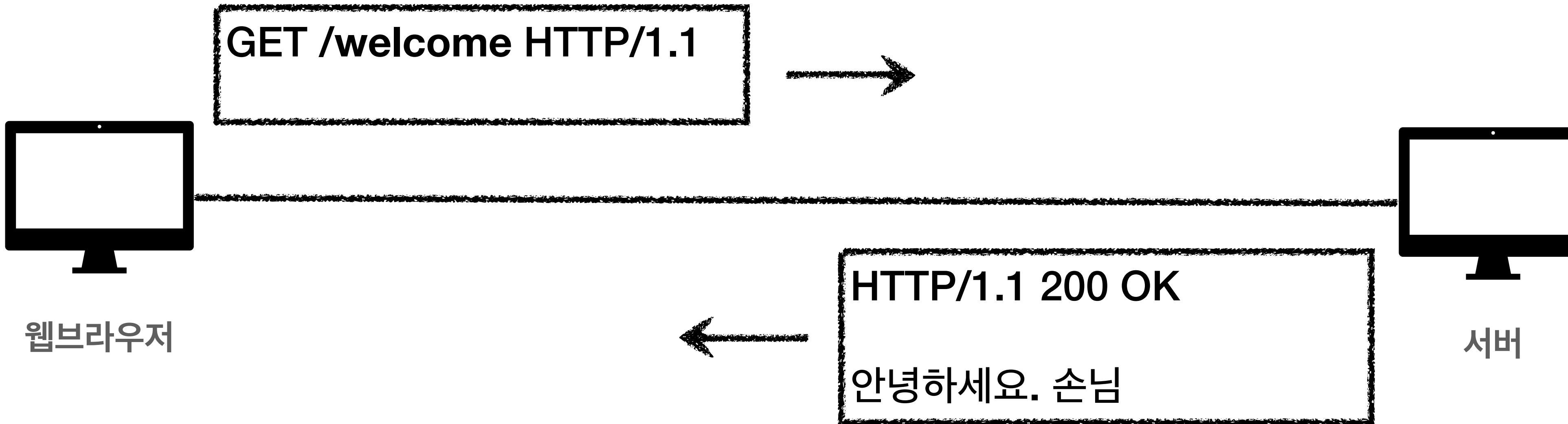
- 리소스 접근시 필요한 인증 방법 정의
- 401 Unauthorized 응답과 함께 사용
- WWW-Authenticate: Newauth realm="apps", type=1,
title="Login to \"apps\\\"", Basic realm="simple"

쿠키

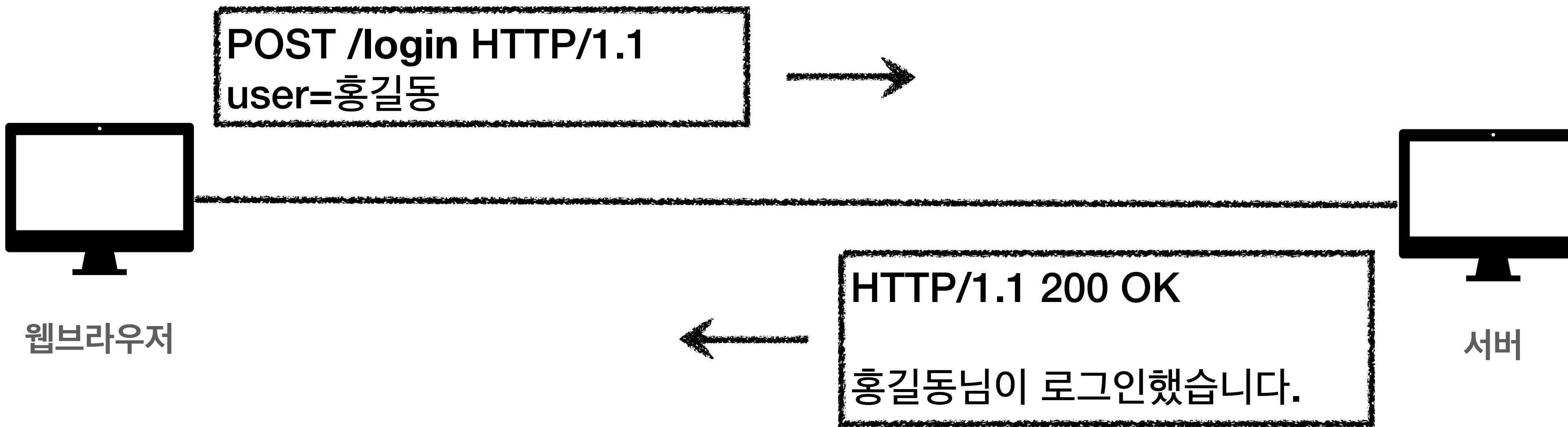
- Set-Cookie: 서버에서 클라이언트로 쿠키 전달(응답)
- Cookie: 클라이언트가 서버에서 받은 쿠키를 저장하고, HTTP 요청시 서버로 전달

쿠키 미사용

처음 welcome 페이지 접근

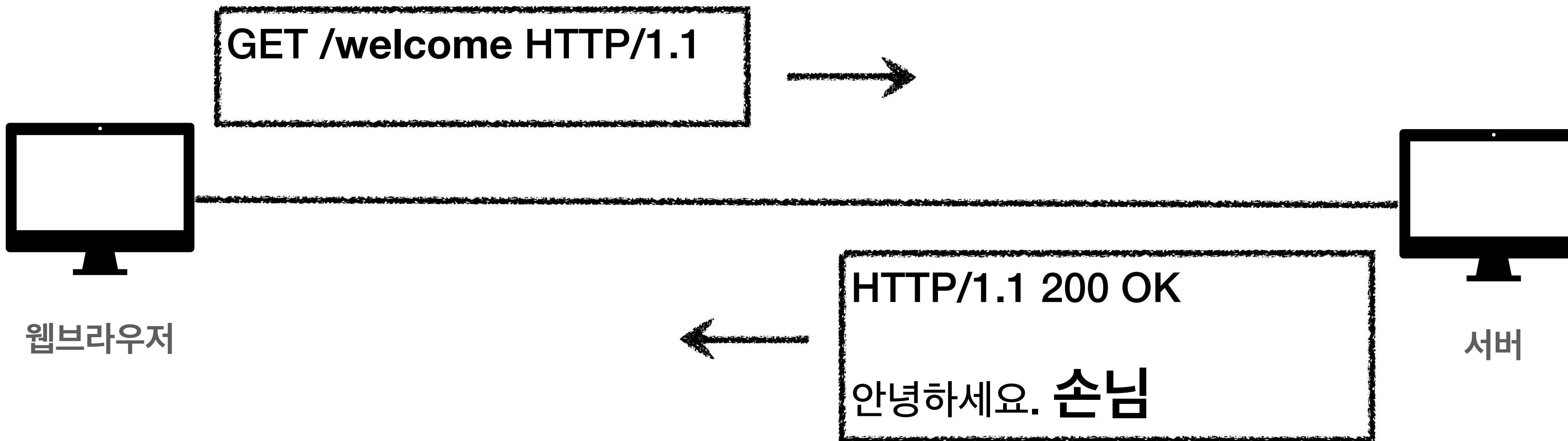


쿠키 미사용 로그인



쿠키 미사용

로그인 이후 welcome 페이지 접근

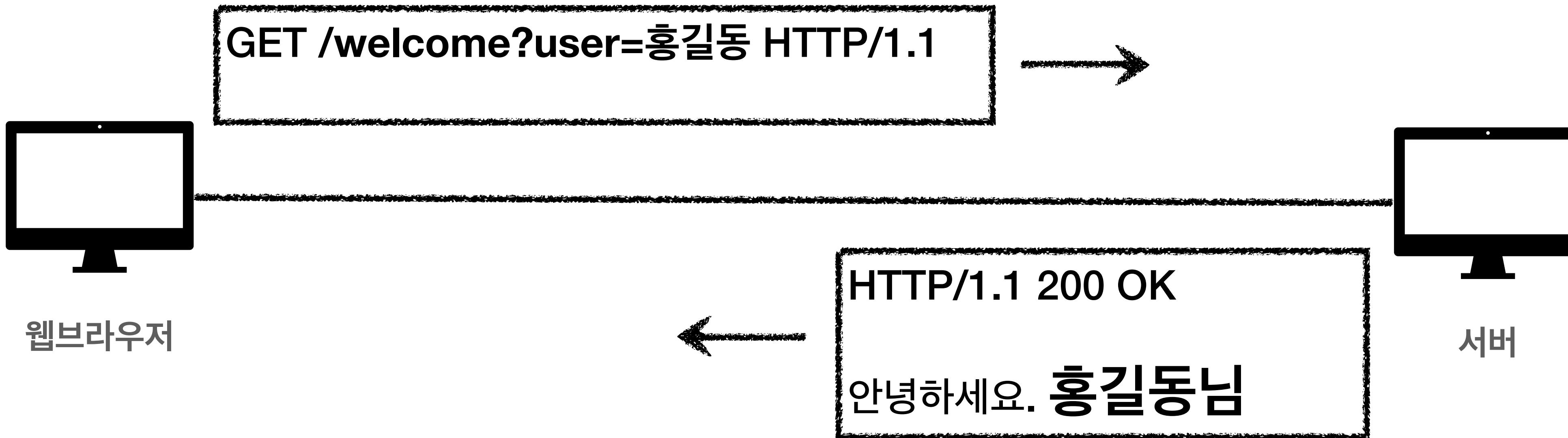


Stateless

- HTTP는 무상태(Stateless) 프로토콜이다.
- 클라이언트와 서버가 요청과 응답을 주고 받으면 연결이 끊어진다.
- 클라이언트가 다시 요청하면 서버는 이전 요청을 기억하지 못한다.
- 클라이언트와 서버는 서로 상태를 유지하지 않는다.

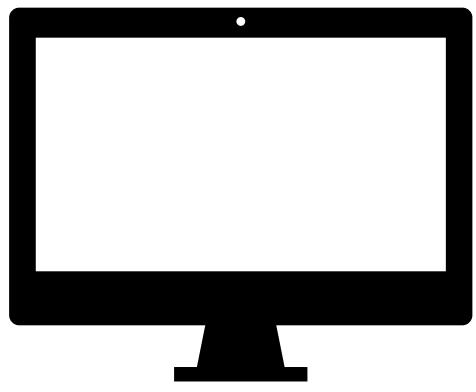
쿠키 미사용

대안 - 모든 요청에 사용자 정보 포함



쿠키 미사용

대안 - 모든 요청과 링크에 사용자 정보 포함?



웹브라우저

GET /welcome?user=홍길동 HTTP/1.1

GET /board?user=홍길동 HTTP/1.1

GET /order?user=홍길동 HTTP/1.1

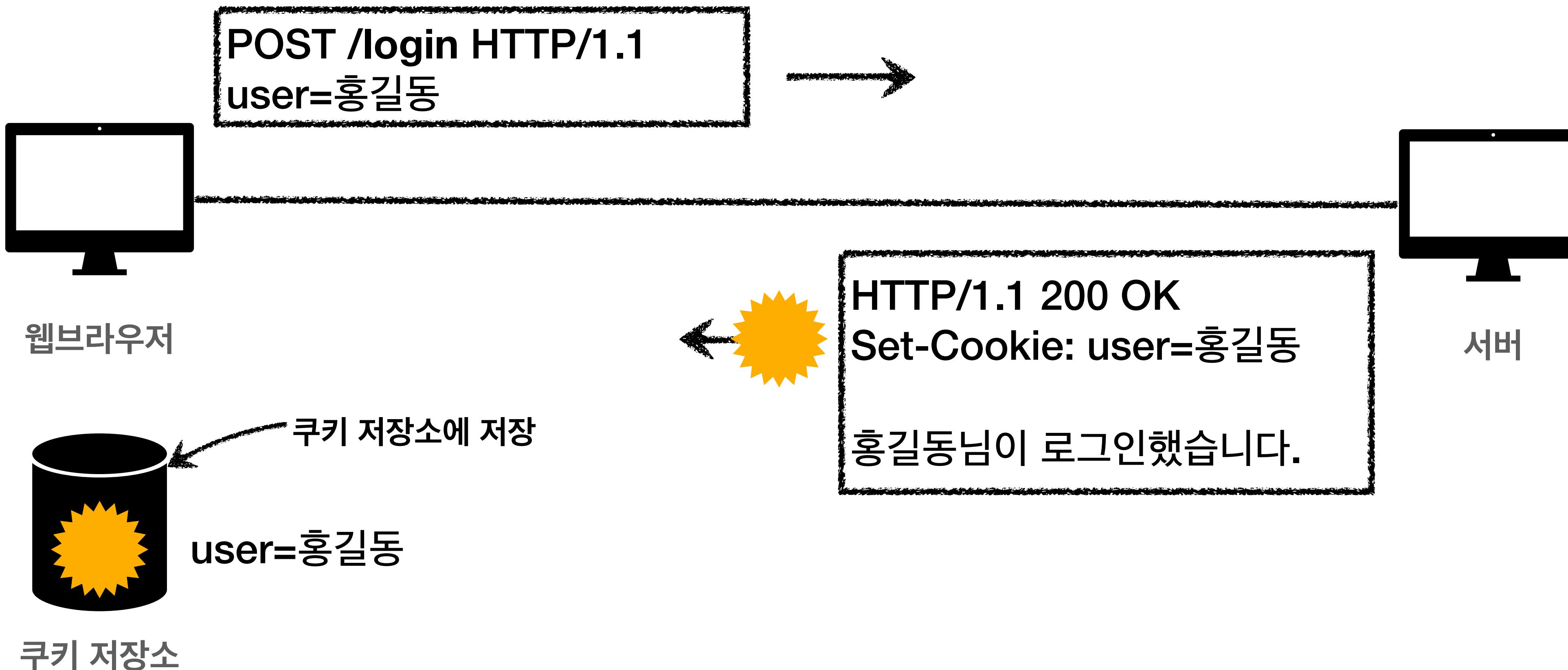
GET /xxx...?user=홍길동 HTTP/1.1

...

모든 요청에 정보를 넘기는 문제

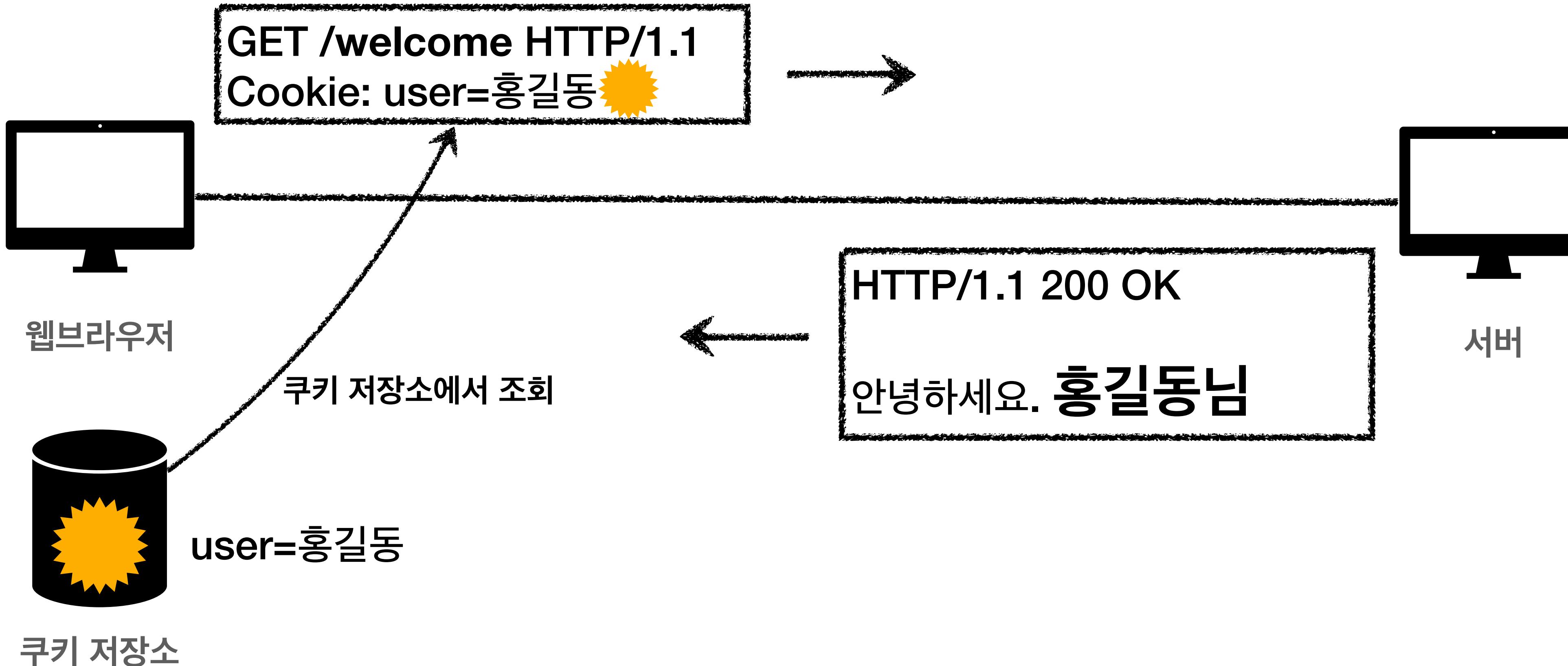
- 모든 요청에 사용자 정보가 포함되도록 개발 해야함
- 브라우저를 완전히 종료하고 다시 열면?

쿠키 로그인



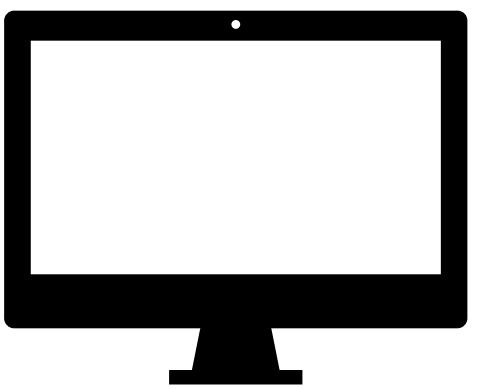
쿠키

로그인 이후 welcome 페이지 접근

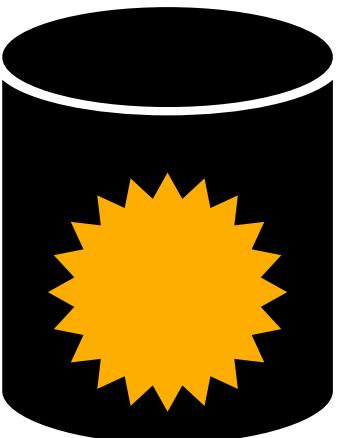


쿠키

모든 요청에 쿠키 정보 자동 포함



웹브라우저



user=홍길동

GET /welcome HTTP/1.1

Cookie: user=홍길동 

GET /board HTTP/1.1

Cookie: user=홍길동 

GET /order HTTP/1.1

Cookie: user=홍길동 

GET /xxx... HTTP/1.1

Cookie: user=홍길동 

...

쿠키 저장소

쿠키

- 예) set-cookie: **sessionId=abcde1234; expires=Sat, 26-Dec-2020 00:00:00 GMT; path=/; domain=.google.com; Secure**
- 사용처
 - 사용자 로그인 세션 관리
 - 광고 정보 트래킹
- 쿠키 정보는 항상 서버에 전송됨
 - 네트워크 트래픽 추가 유발
 - 최소한의 정보만 사용(세션 id, 인증 토큰)
 - 서버에 전송하지 않고, 웹 브라우저 내부에 데이터를 저장하고 싶으면 웹 스토리지 (localStorage, sessionStorage) 참고
- 주의!
 - 보안에 민감한 데이터는 저장하면 안됨(주민번호, 신용카드 번호 등등)

쿠키 - 생명주기

Expires, max-age

- Set-Cookie: **expires**=Sat, 26-Dec-2020 04:39:21 GMT
 - 만료일이 되면 쿠키 삭제
- Set-Cookie: **max-age**=3600 (3600초)
 - 0이나 음수를 지정하면 쿠키 삭제
- 세션 쿠키: 만료 날짜를 생략하면 브라우저 종료시 까지만 유지
- 영속 쿠키: 만료 날짜를 입력하면 해당 날짜까지 유지

쿠키 - 도메인

Domain

- 예) domain=example.org
- 명시: 명시한 문서 기준 도메인 + 서브 도메인 포함
 - domain=example.org를 지정해서 쿠키 생성
 - example.org는 물론이고
 - dev.example.org도 쿠키 접근
- 생략: 현재 문서 기준 도메인만 적용
 - example.org에서 쿠키를 생성하고 domain 지정을 생략
 - example.org에서만 쿠키 접근
 - dev.example.org는 쿠키 미접근

쿠키 - 경로

Path

- 예) path=/home
- 이 경로를 포함한 하위 경로 페이지만 쿠키 접근
- 일반적으로 path=/ 루트로 지정
- 예)
 - **path=/home** 지정
 - /home -> 가능
 - /home/level1 -> 가능
 - /home/level1/level2 -> 가능
 - /hello -> 불가능

쿠키 - 보안

Secure, HttpOnly, SameSite

- Secure
 - 쿠키는 http, https를 구분하지 않고 전송
 - Secure를 적용하면 https인 경우에만 전송
- HttpOnly
 - XSS 공격 방지
 - 자바스크립트에서 접근 불가(`document.cookie`)
 - HTTP 전송에만 사용
- SameSite
 - XSRF 공격 방지
 - 요청 도메인과 쿠키에 설정된 도메인이 같은 경우만 쿠키 전송

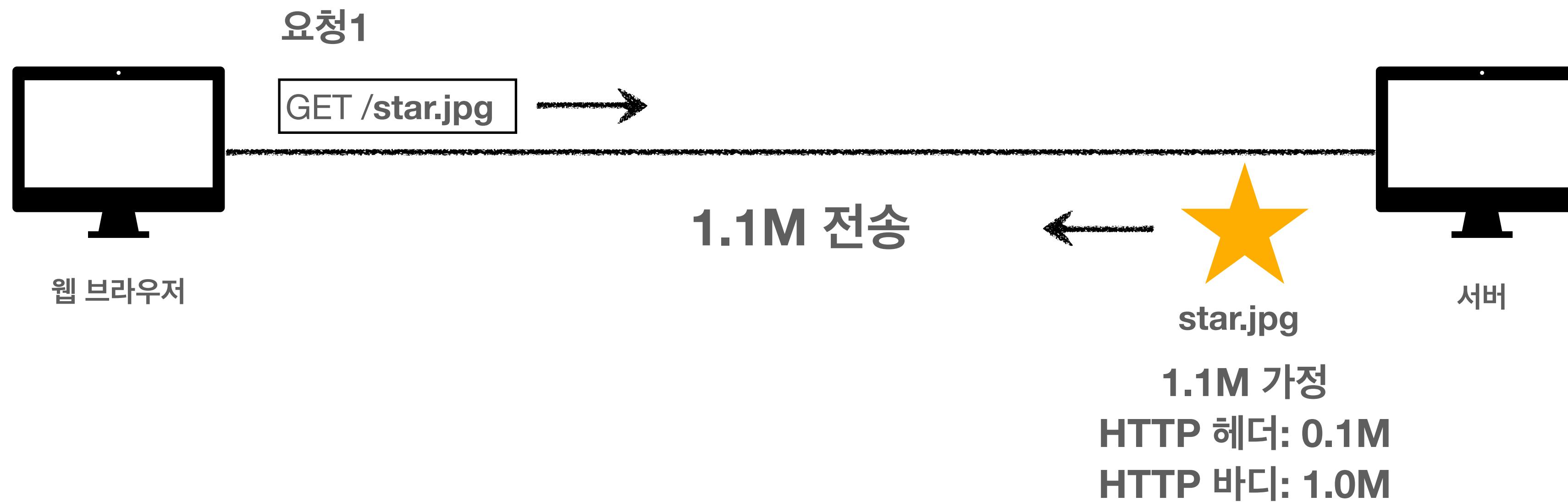
HTTP 헤더2 캐시와 조건부 요청

캐시 기본 동작

캐시가 없을 때 첫 번째 요청

HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34ljk098udjkla
slkjdf;lkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123

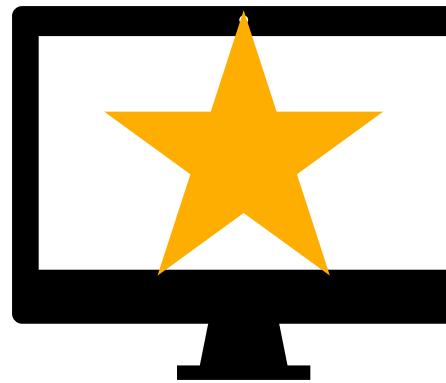


캐시가 없을 때

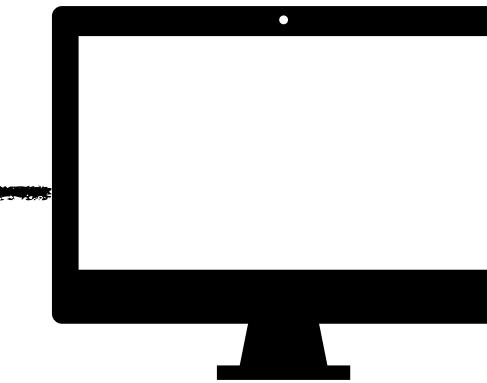
첫 번째 요청

HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34ljk098udjkla
slkjdf;lkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123



웹 브라우저



서버

캐시가 없을 때 두 번째 요청

HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 34012

lkj123kljoiasudlkjaweioluywlnfdo912u34ljk098udjkla
slkjdf;lkj9;04ruawsldkal;skdjfa;ow9ejkl3123123



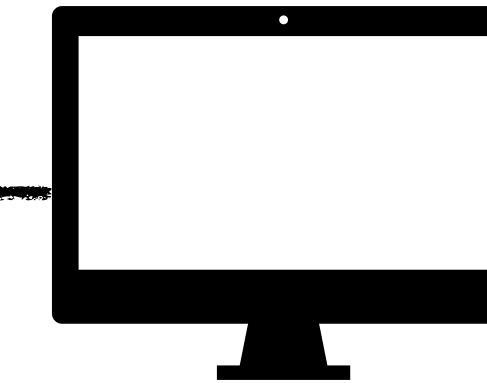
캐시가 없을 때 두 번째 요청

HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34ljk098udjkla
slkjdf;lkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123



웹 브라우저

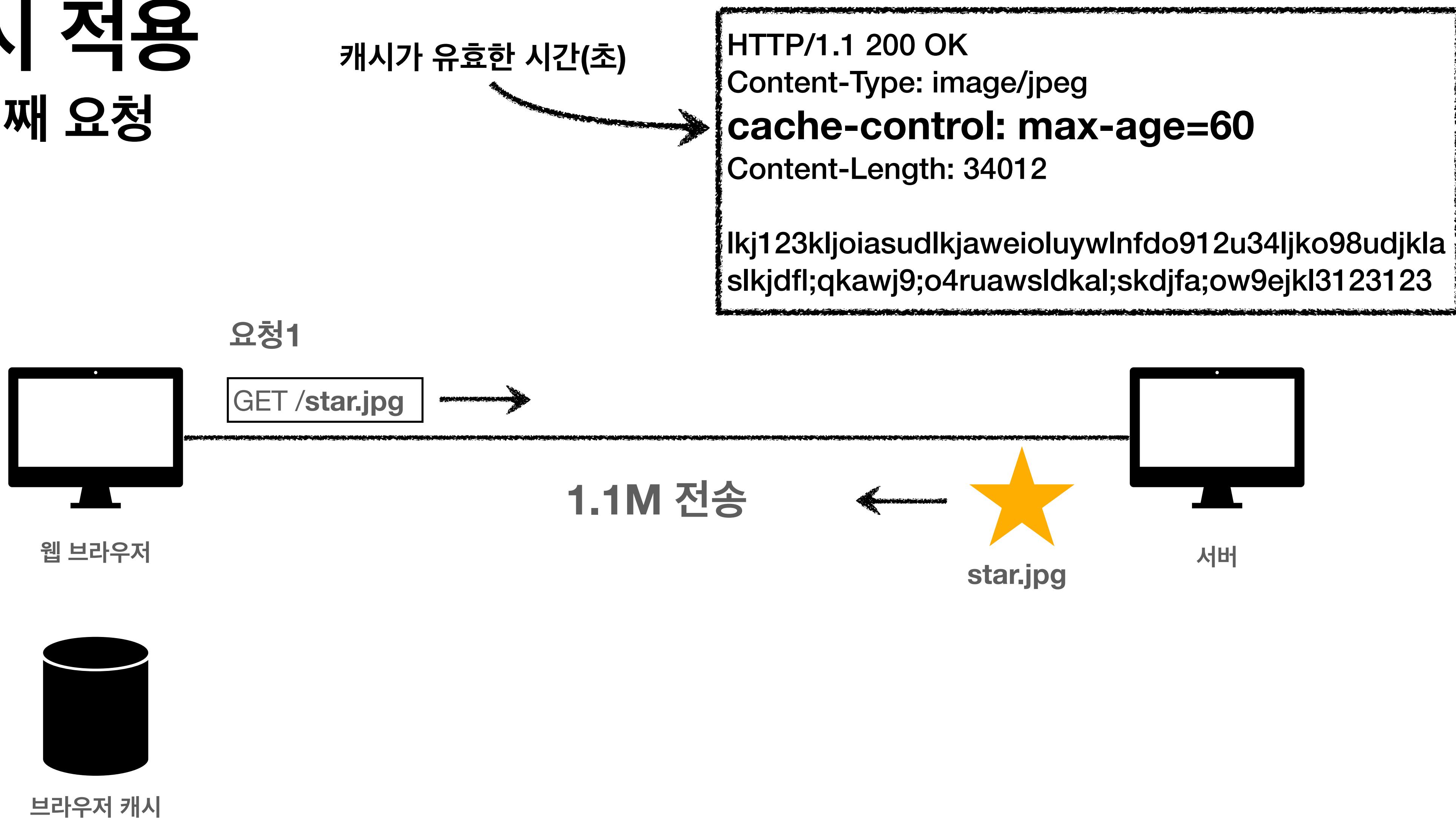


서버

캐시가 없을 때

- 데이터가 변경되지 않아도 계속 네트워크를 통해서 데이터를 다운로드 받아야 한다.
- 인터넷 네트워크는 매우 느리고 비싸다.
- 브라우저 로딩 속도가 느리다.
- 느린 사용자 경험

캐시 적용 첫 번째 요청

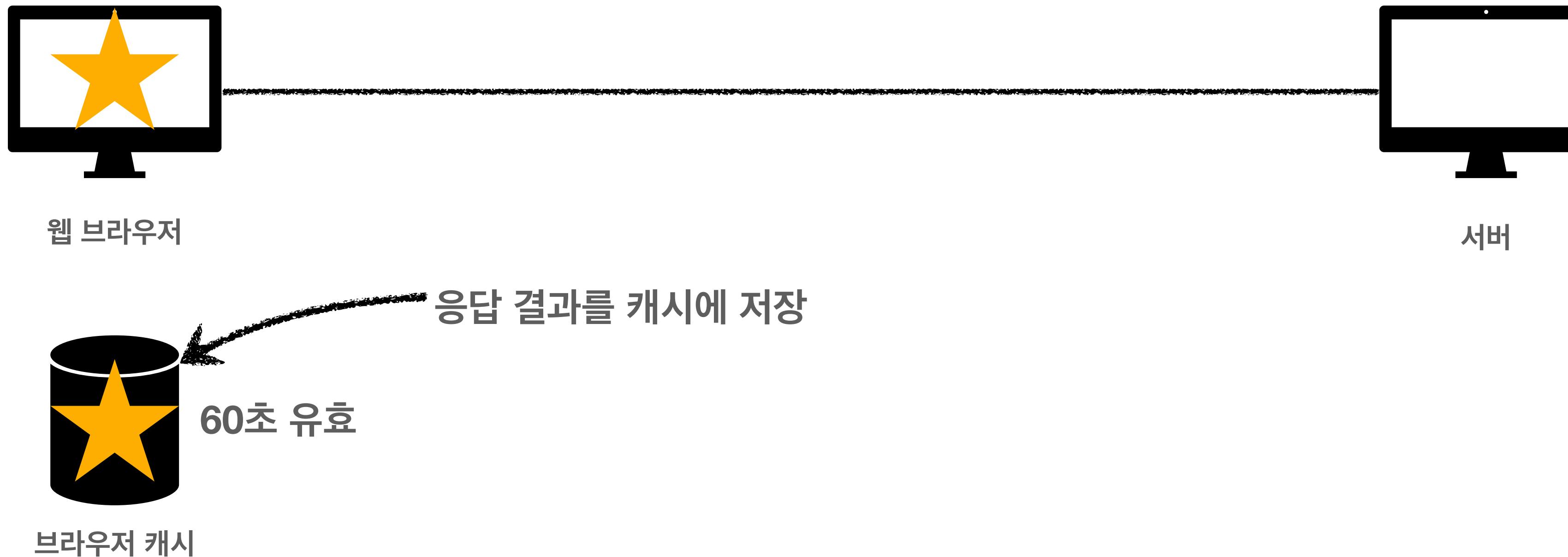


캐시 적용

첫 번째 요청

HTTP/1.1 200 OK
Content-Type: image/jpeg
cache-control: max-age=60
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34lko98udjklaslkjdfl;qkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123



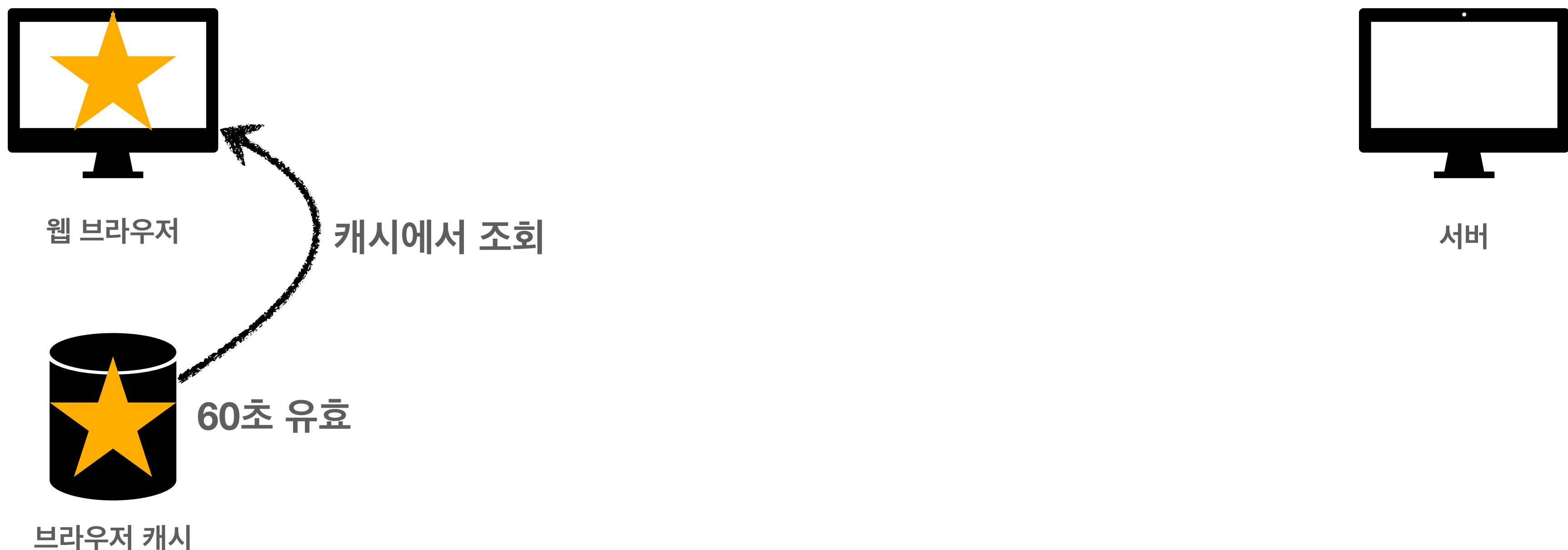
캐시 적용

두 번째 요청



캐시 적용

두 번째 요청



캐시 적용

- 캐시 덕분에 캐시 가능 시간동안 네트워크를 사용하지 않아도 된다.
- 비싼 네트워크 사용량을 줄일 수 있다.
- 브라우저 로딩 속도가 매우 빠르다.
- 빠른 사용자 경험

캐시 적용

세 번째 요청 - 캐시 시간 초과



캐시 적용

세 번째 요청 - 캐시 시간 초과

캐시가 유효한 시간(초)

HTTP/1.1 200 OK

Content-Type: image/jpeg

cache-control: max-age=60

Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34lko98udjkla
slkjdf;lkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123

요청3



60초 초과

브라우저 캐시

캐시 적용

세 번째 요청 - 캐시 시간 초과

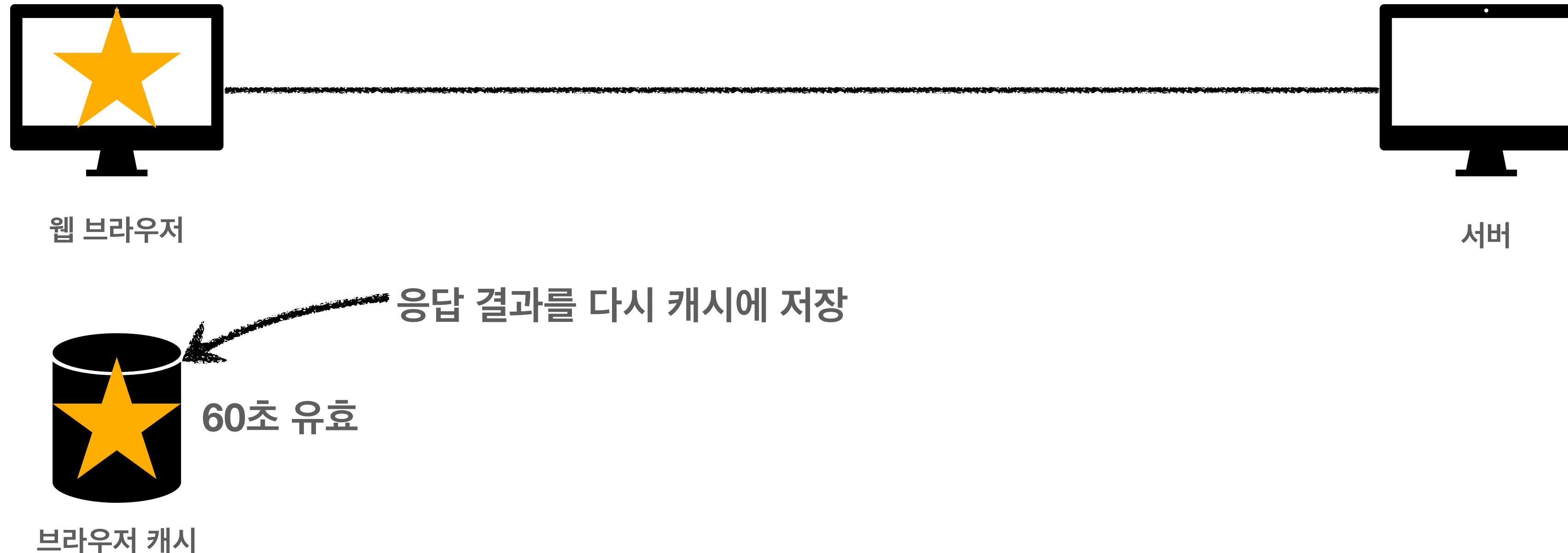
HTTP/1.1 200 OK

Content-Type: image/jpeg

cache-control: max-age=60

Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34lko98udjklaslkjdfl;qkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123



캐시 시간 초과

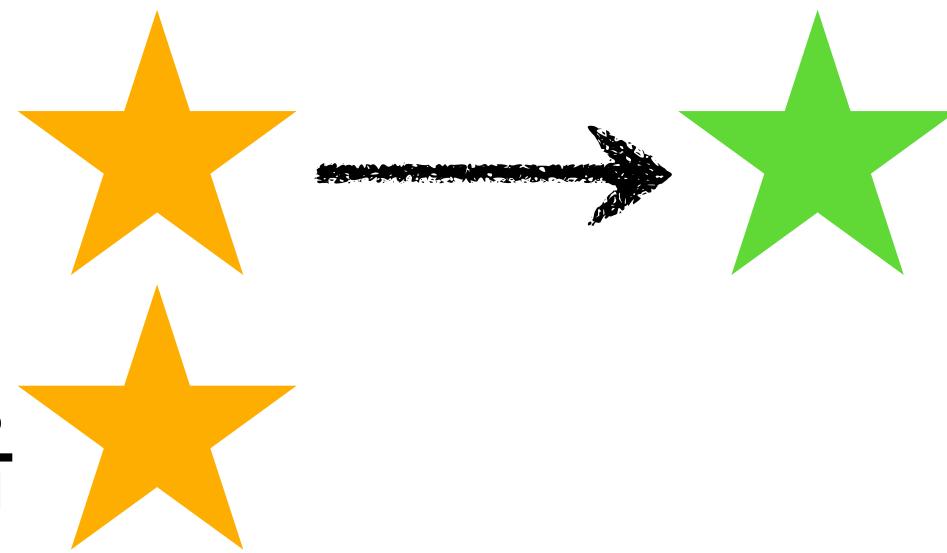
- 캐시 유효 시간이 초과하면, 서버를 통해 데이터를 다시 조회하고, 캐시를 갱신한다.
- 이때 다시 네트워크 다운로드가 발생한다.

검증 헤더와 조건부 요청1

캐시 시간 초과

- 캐시 유효 시간이 초과해서 서버에 다시 요청하면 다음 두 가지 상황이 나타난다.

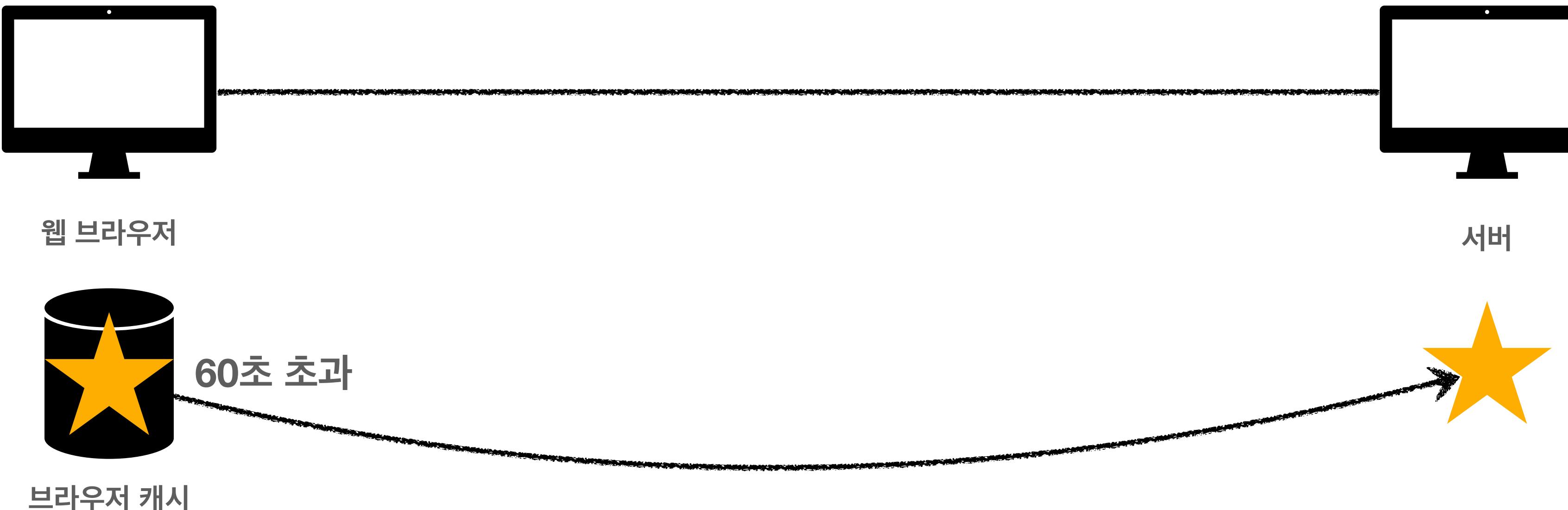
1. 서버에서 기존 데이터를 변경함



2. 서버에서 기존 데이터를 변경하지 않음

캐시 시간 초과

- 캐시 만료후에도 서버에서 데이터를 변경하지 않음 
- 생각해보면 데이터를 전송하는 대신에 저장해 두었던 캐시를 재사용 할 수 있다.
- 단 클라이언트의 데이터와 서버의 데이터가 같다는 사실을 확인할 수 있는 방법 필요



검증 헤더 추가

첫 번째 요청

데이터가 마지막에 수정된 시간

HTTP/1.1 200 OK
Content-Type: image/jpeg
cache-control: max-age=60
Last-Modified: 2020년 11월 10일 10:00:00
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfdo912u34lko98udjklaslkjdf;qkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123

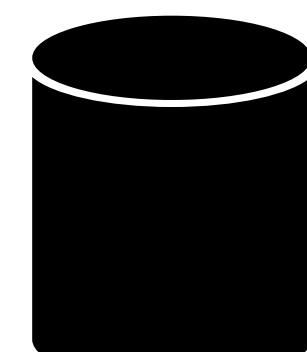
요청1



1.1M 전송

데이터 최종 수정일

2020년 11월 10일 10:00:00



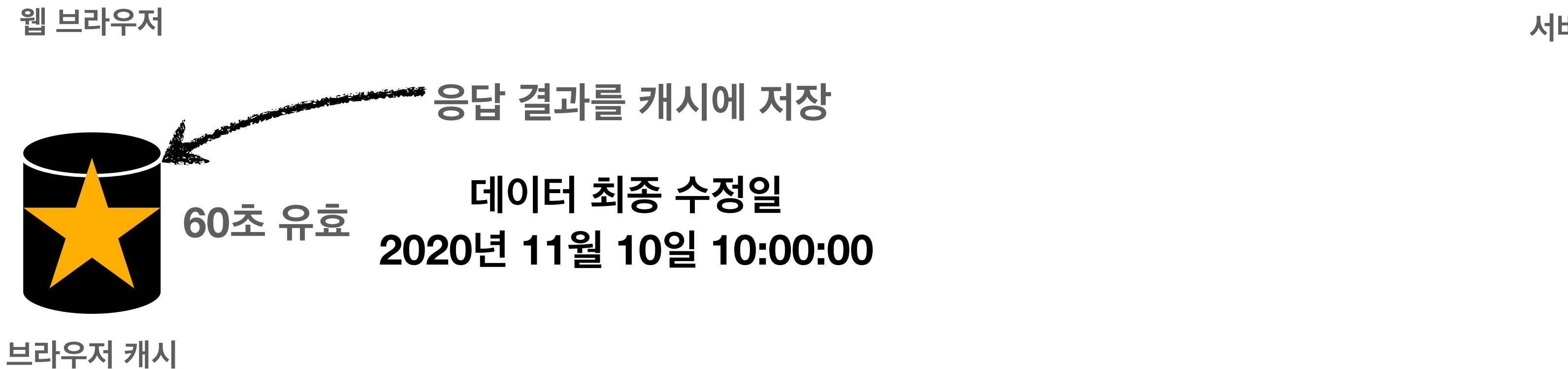
브라우저 캐시

검증 헤더 추가

첫 번째 요청

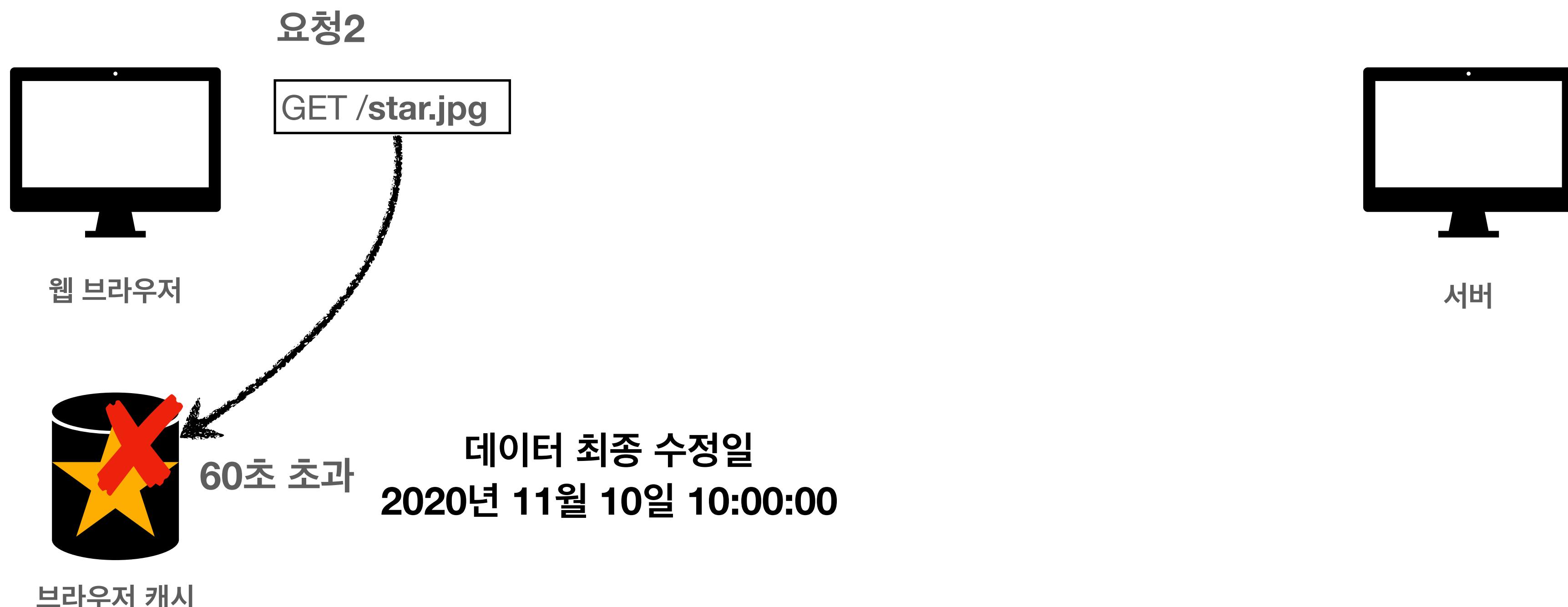
```
HTTP/1.1 200 OK
Content-Type: image/jpeg
cache-control: max-age=60
Last-Modified: 2020년 11월 10일 10:00:00
Content-Length: 34012

lkj123kljoiasudlkjaweioluywlnfdo912u34lko98udjklaslkjdf;qkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123
```



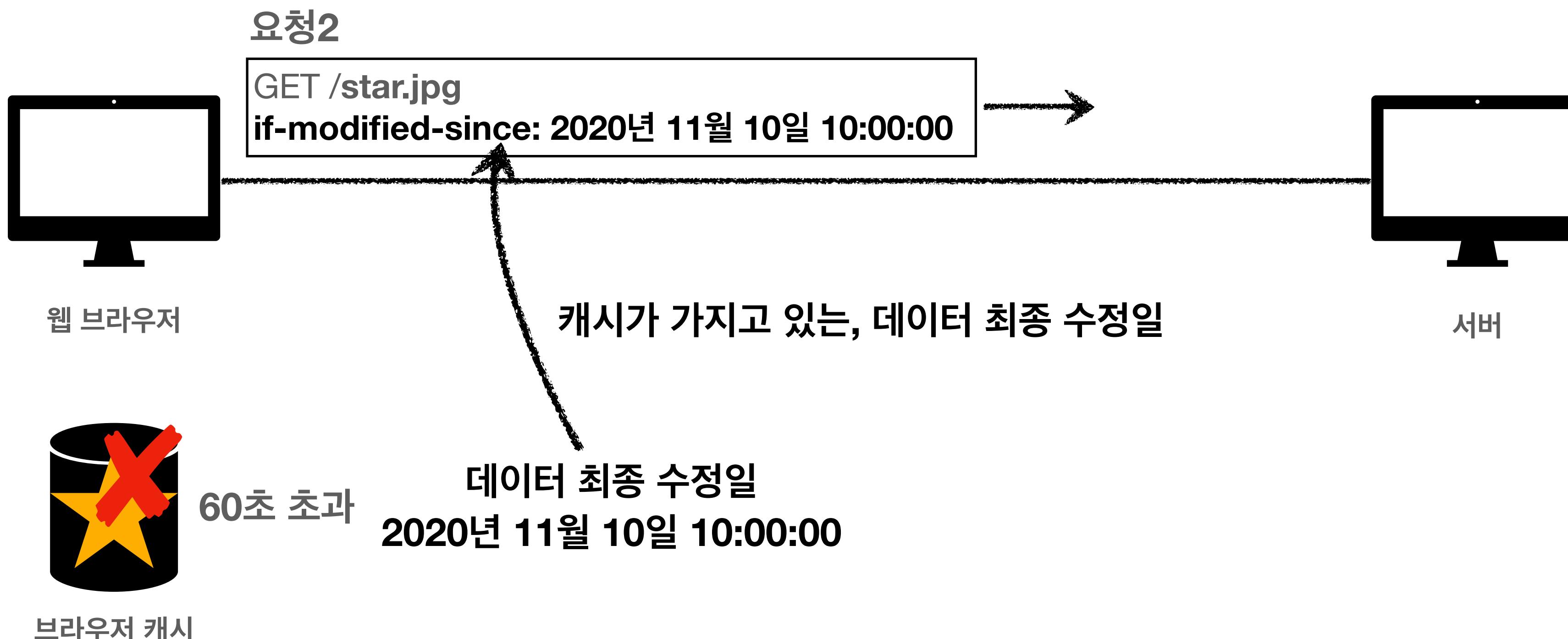
검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



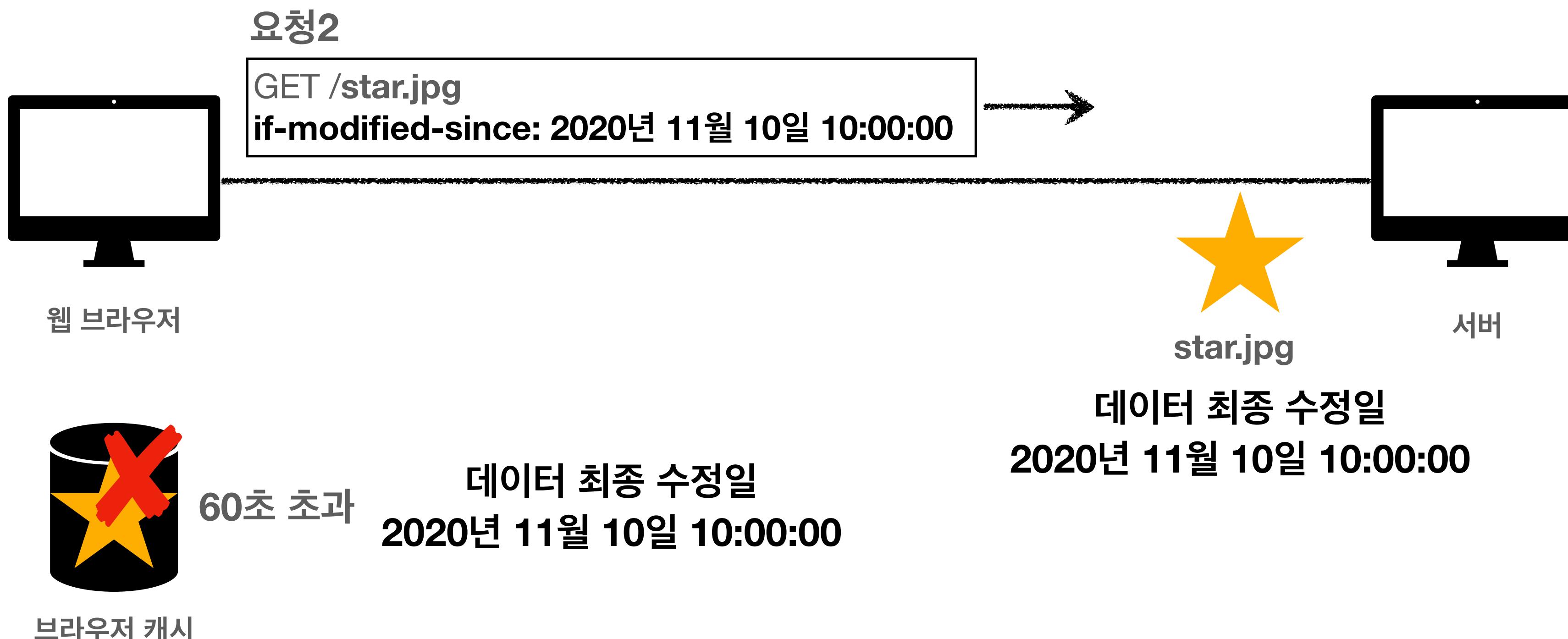
검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



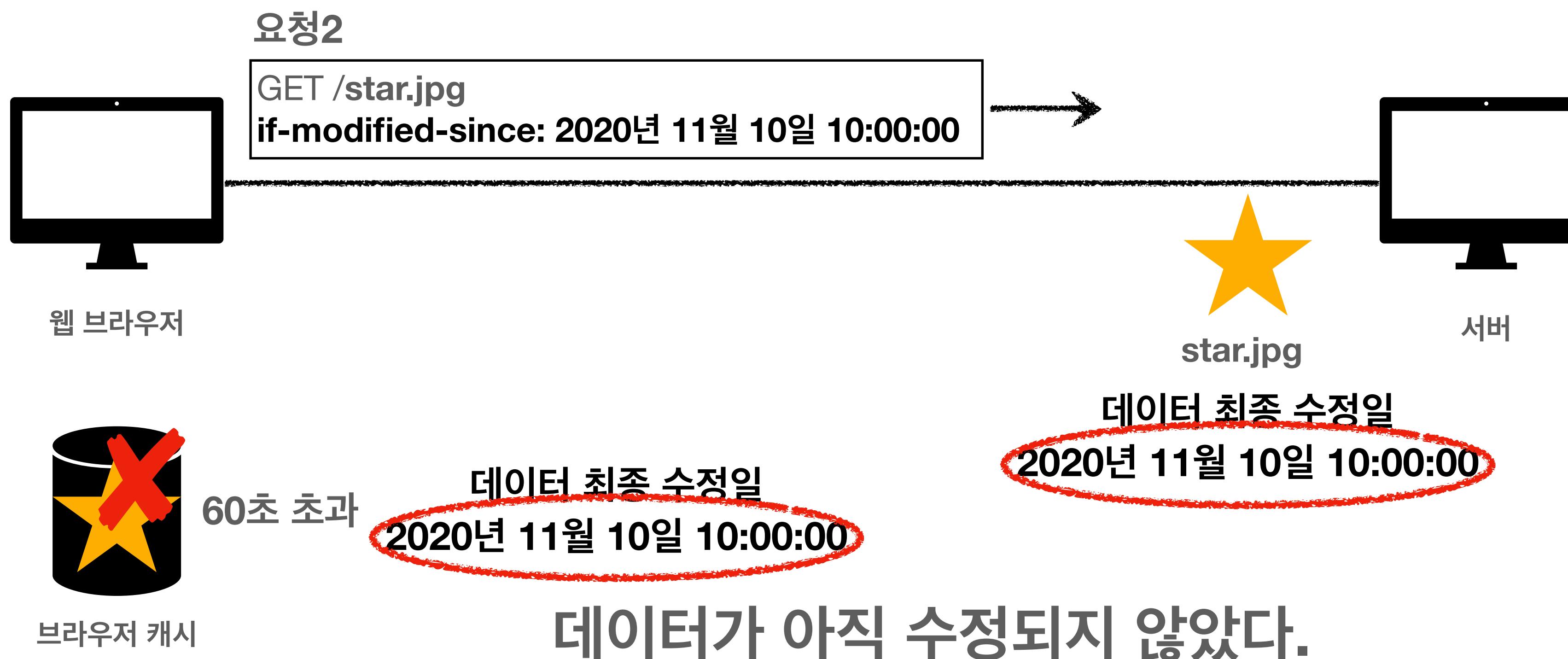
검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



검증 헤더 추가

두 번째 요청 - 캐시 시간 초과

HTTP/1.1 304 Not Modified

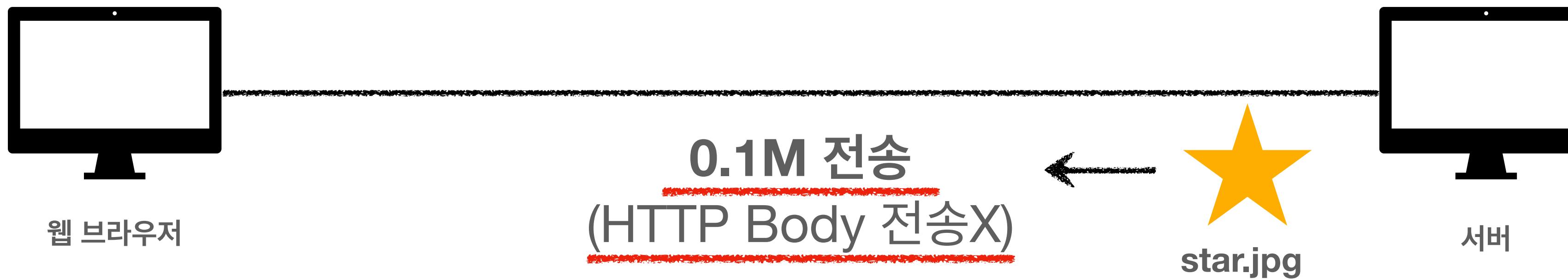
Content-Type: image/jpeg

cache-control: max-age=60

Last-Modified: 2020년 11월 10일 10:00:00

Content-Length: 34012

HTTP Body가 없음



60초 초과

데이터 최종 수정일
2020년 11월 10일 10:00:00

브라우저 캐시

1.1M 가정

HTTP 헤더: 0.1M

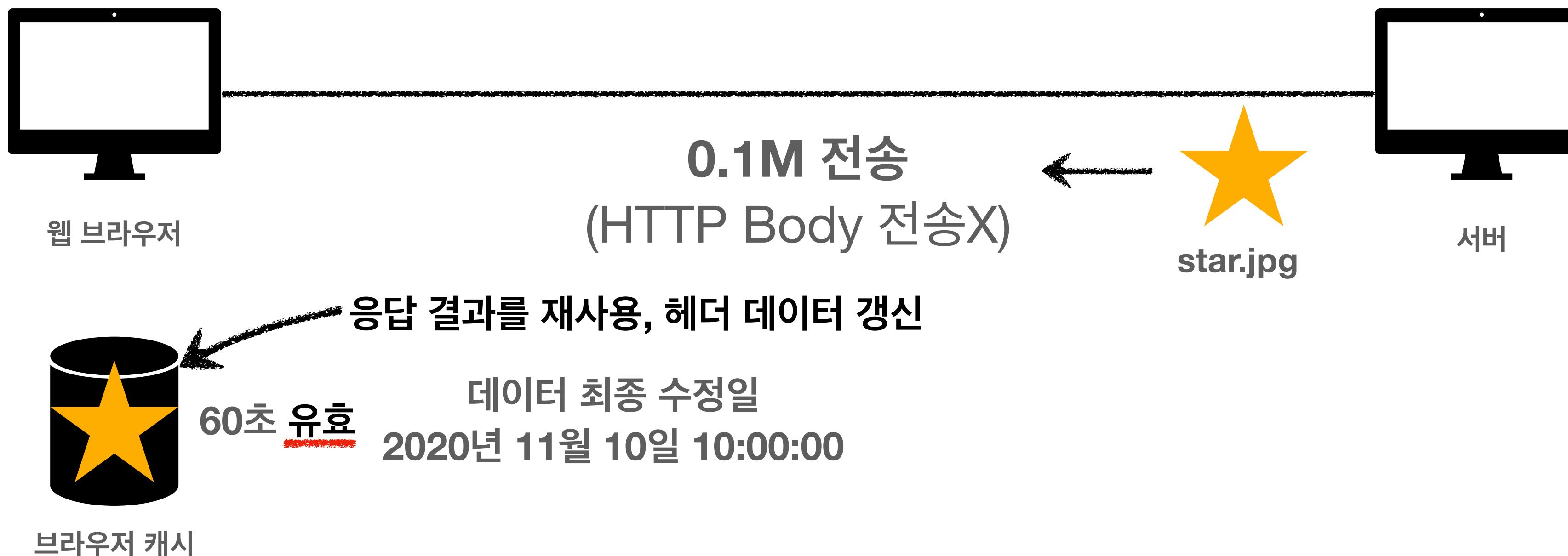
HTTP 바디: 1.0M

검증 헤더 추가

두 번째 요청 - 캐시 시간 초과

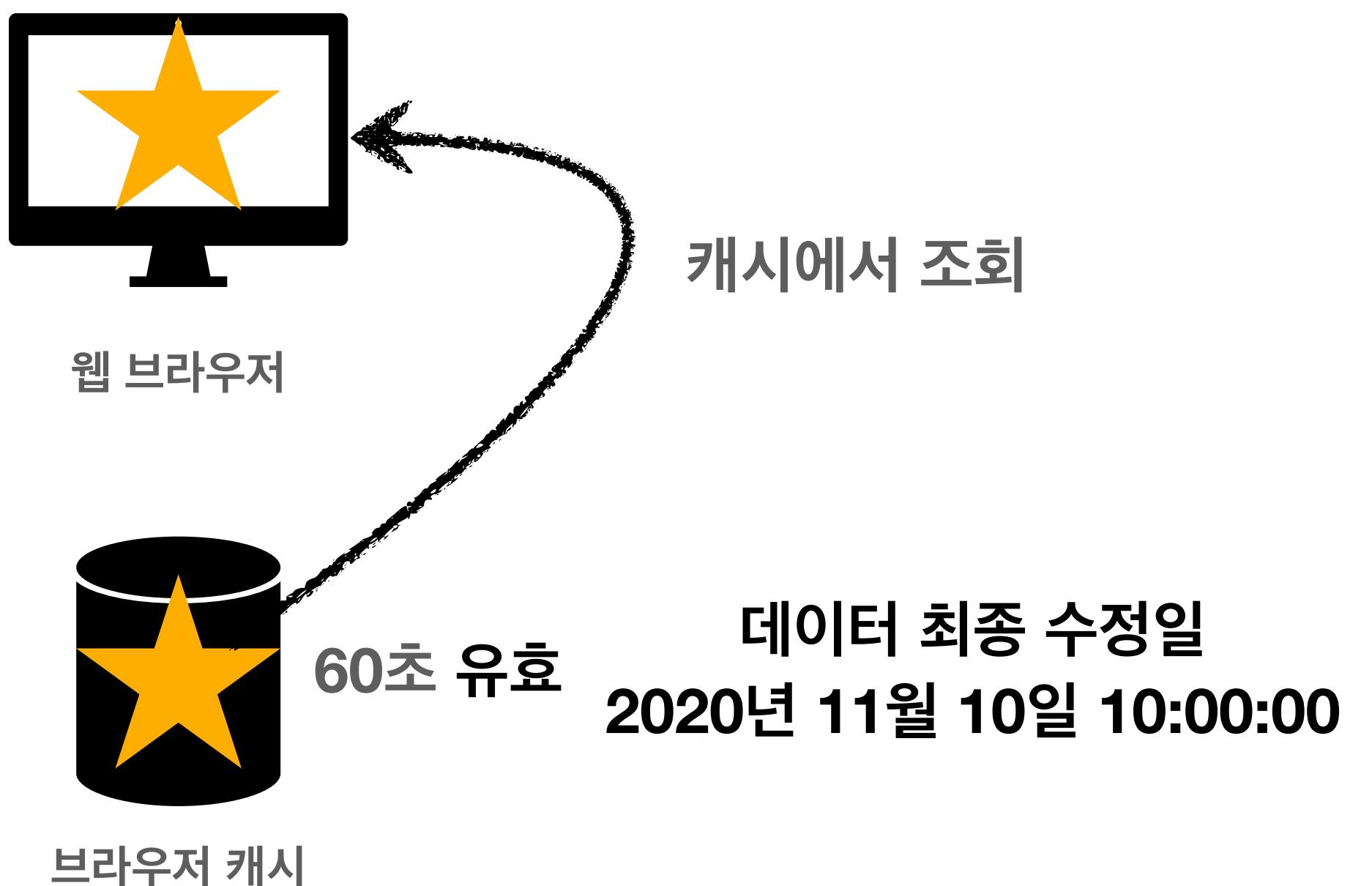
HTTP/1.1 304 Not Modified
Content-Type: image/jpeg
cache-control: max-age=60
Last-Modified: 2020년 11월 10일 10:00:00
Content-Length: 34012

HTTP Body가 없음



검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



검증 헤더와 조건부 요청

정리

- 캐시 유효 시간이 초과해도, 서버의 데이터가 갱신되지 않으면
- 304 Not Modified + 헤더 메타 정보만 응답(바디X)
- 클라이언트는 서버가 보낸 응답 헤더 정보로 캐시의 메타 정보를 갱신
- 클라이언트는 캐시에 저장되어 있는 데이터 재활용
- 결과적으로 네트워크 다운로드가 발생하지만 용량이 적은 헤더 정보만 다운로드
- 매우 실용적인 해결책

검증 헤더와 조건부 요청2

검증 헤더와 조건부 요청

- 검증 헤더
 - 캐시 데이터와 서버 데이터가 같은지 검증하는 데이터
 - Last-Modified , ETag
- 조건부 요청 헤더
 - 검증 헤더로 조건에 따른 분기
 - If-Modified-Since: Last-Modified 사용
 - If-None-Match: ETag 사용
 - 조건이 만족하면 200 OK
 - 조건이 만족하지 않으면 304 Not Modified

검증 헤더와 조건부 요청

예시

- If-Modified-Since: 이후에 데이터가 수정되었으면?
 - 데이터 미변경 예시
 - 캐시: 2020년 11월 10일 10:00:00 vs 서버: 2020년 11월 10일 10:00:00
 - **304 Not Modified**, 헤더 데이터만 전송(BODY 미포함)
 - 전송 용량 0.1M (헤더 0.1M)
 - 데이터 변경 예시
 - 캐시: 2020년 11월 10일 10:00:00 vs 서버: 2020년 11월 10일 **11:00:00**
 - **200 OK**, 모든 데이터 전송(BODY 포함)
 - 전송 용량 1.1M (헤더 0.1M, 바디 1.0M)

검증 헤더와 조건부 요청

Last-Modified, If-Modified-Since 단점

- 1초 미만(0.x초) 단위로 캐시 조정이 불가능
- 날짜 기반의 로직 사용
- 데이터를 수정해서 날짜가 다르지만, 같은 데이터를 수정해서 데이터 결과가 똑같은 경우
- 서버에서 별도의 캐시 로직을 관리하고 싶은 경우
 - 예) 스페이스나 주석처럼 크게 영향이 없는 변경에서 캐시를 유지하고 싶은 경우

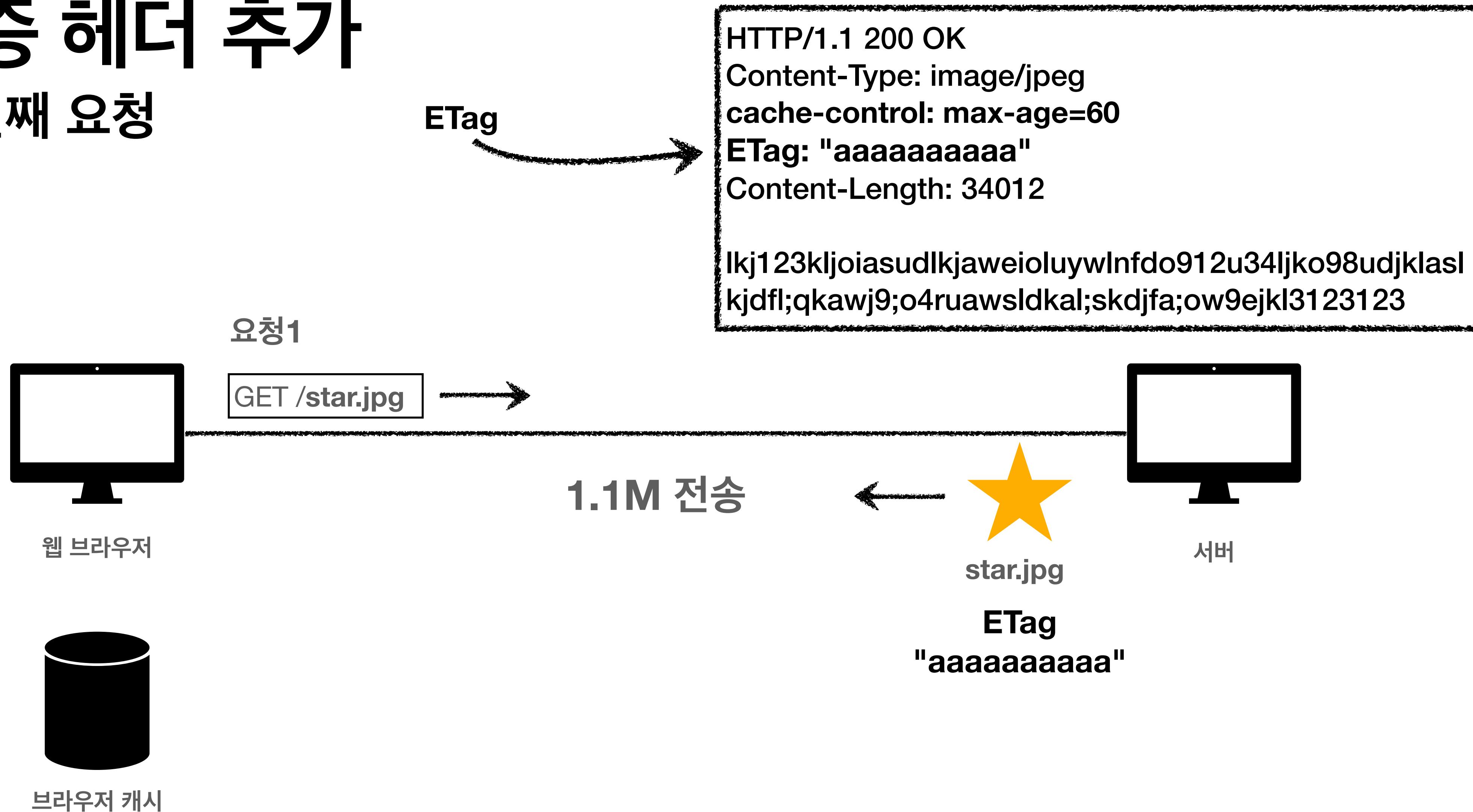
검증 헤더와 조건부 요청

ETag, If-None-Match

- ETag(Entity Tag)
- 캐시용 데이터에 임의의 고유한 버전 이름을 달아둠
 - 예) ETag: "v1.0", ETag: "a2jiodwjekjl3"
- 데이터가 변경되면 이 이름을 바꾸어서 변경함(Hash를 다시 생성)
 - 예) ETag: "aaaaaa" -> ETag: "bbbbbb"
- 진짜 단순하게 ETag만 보내서 같으면 유지, 다르면 다시 받기!

검증 헤더 추가

첫 번째 요청



검증 헤더 추가

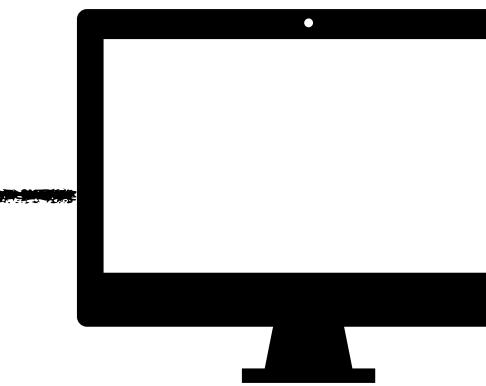
첫 번째 요청

HTTP/1.1 200 OK
Content-Type: image/jpeg
cache-control: max-age=60
ETag: "aaaaaaaaaa"
Content-Length: 34012

Ikj123kljoiasudlkjaweioluywlnfd0912u34lko98udjklaslkjdf; qkawj9;o4ruawsldkal;skdjfa;ow9ejkl3123123



웹 브라우저



서버



브라우저 캐시

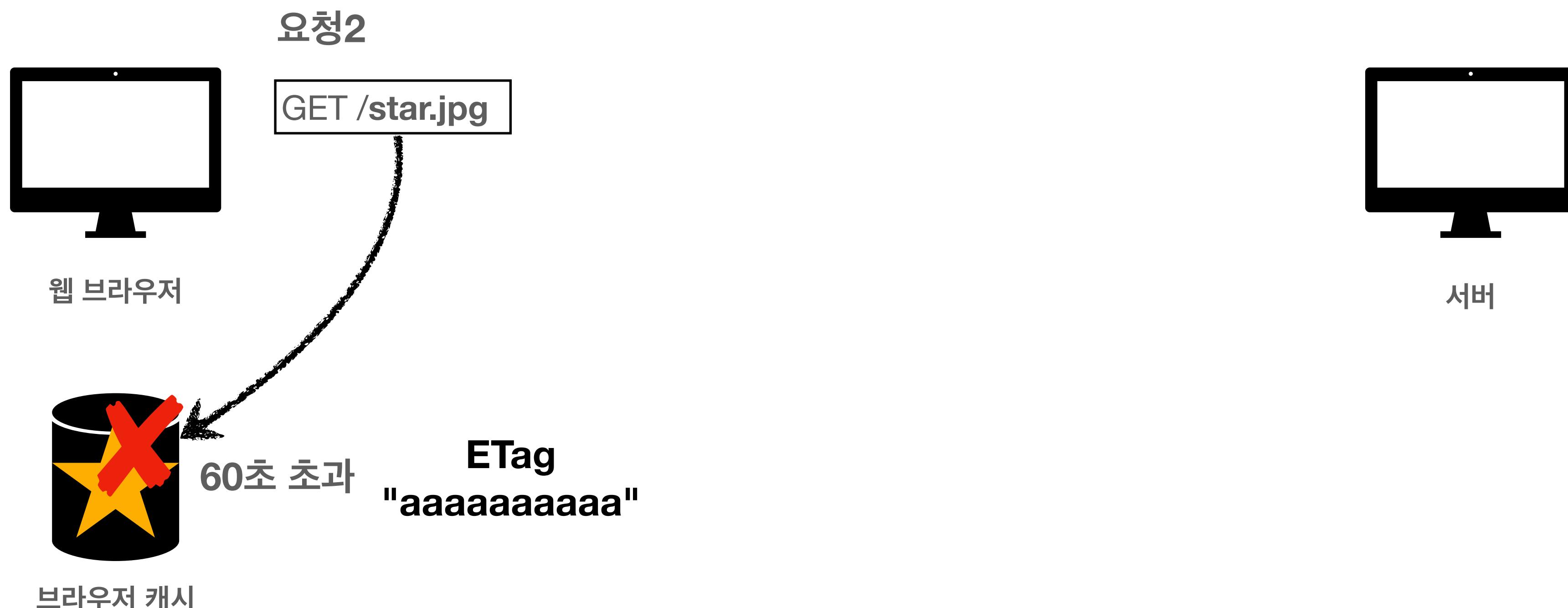
응답 결과를 캐시에 저장

60초 유효

ETag
"aaaaaaaaaa"

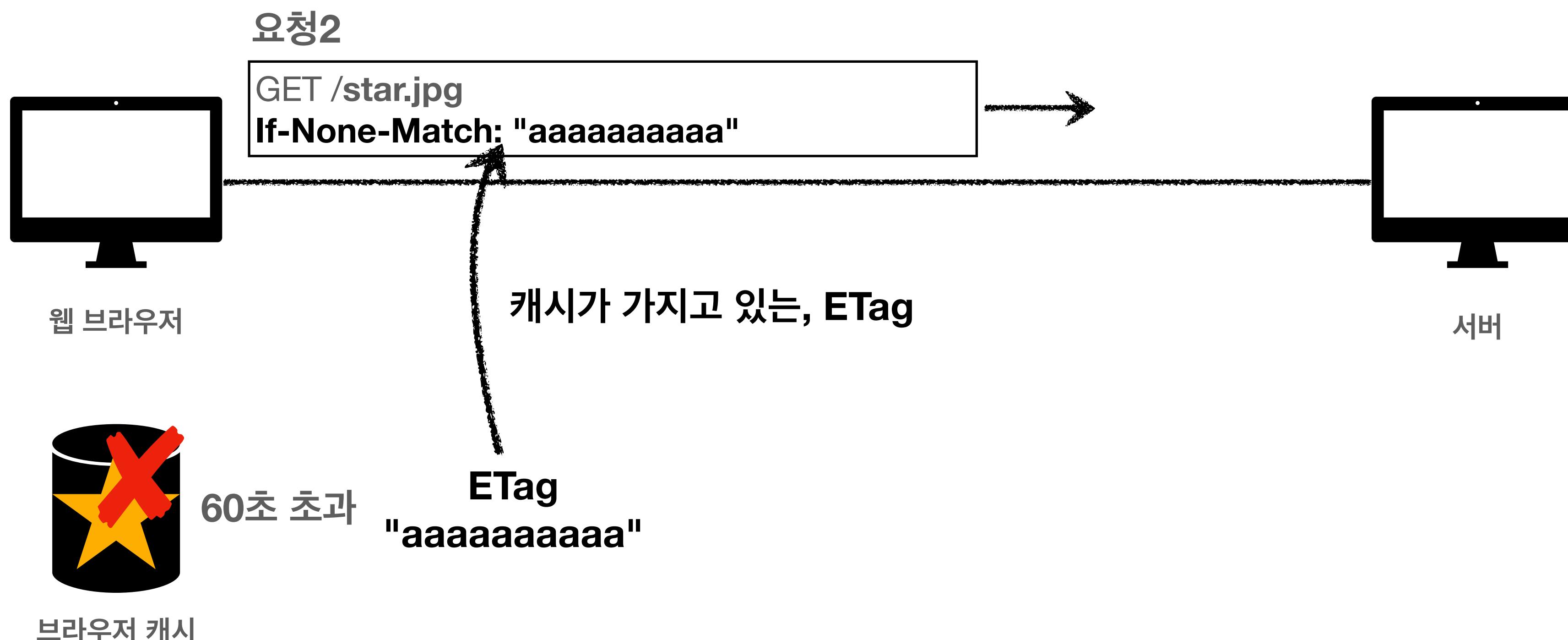
검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



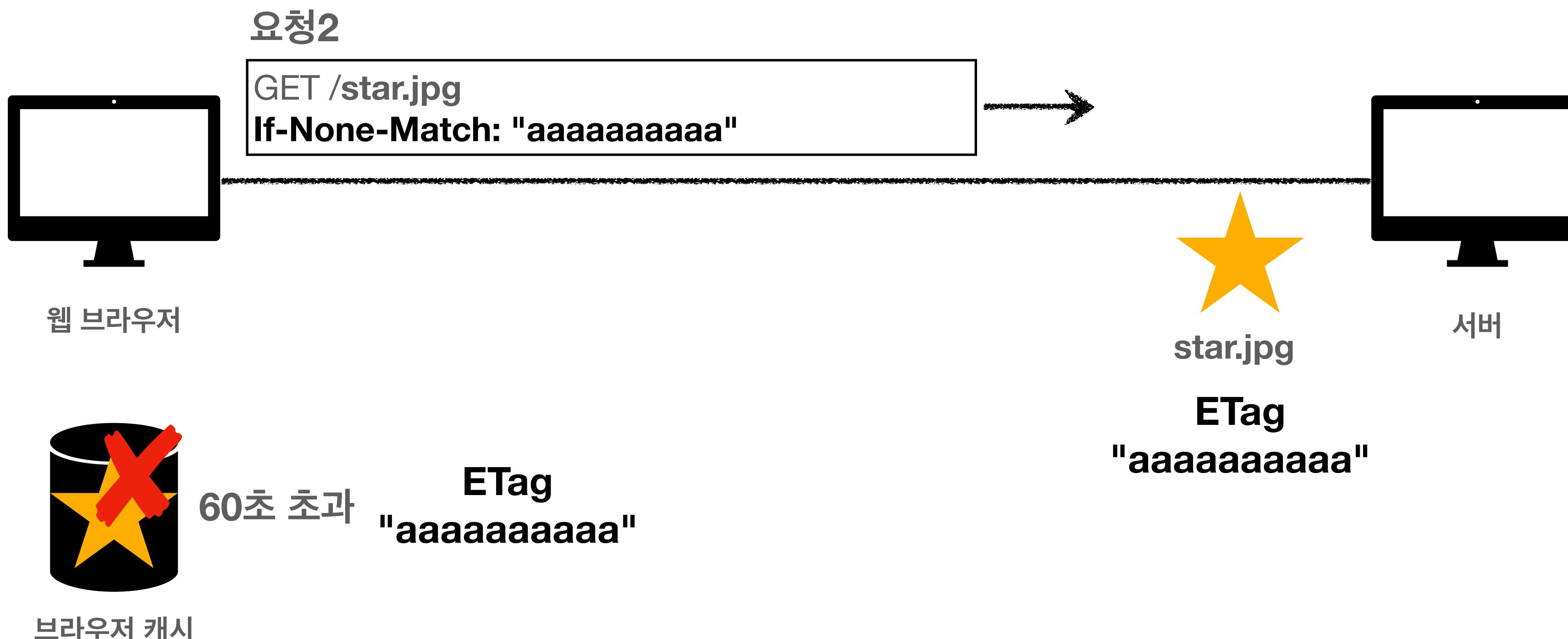
검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



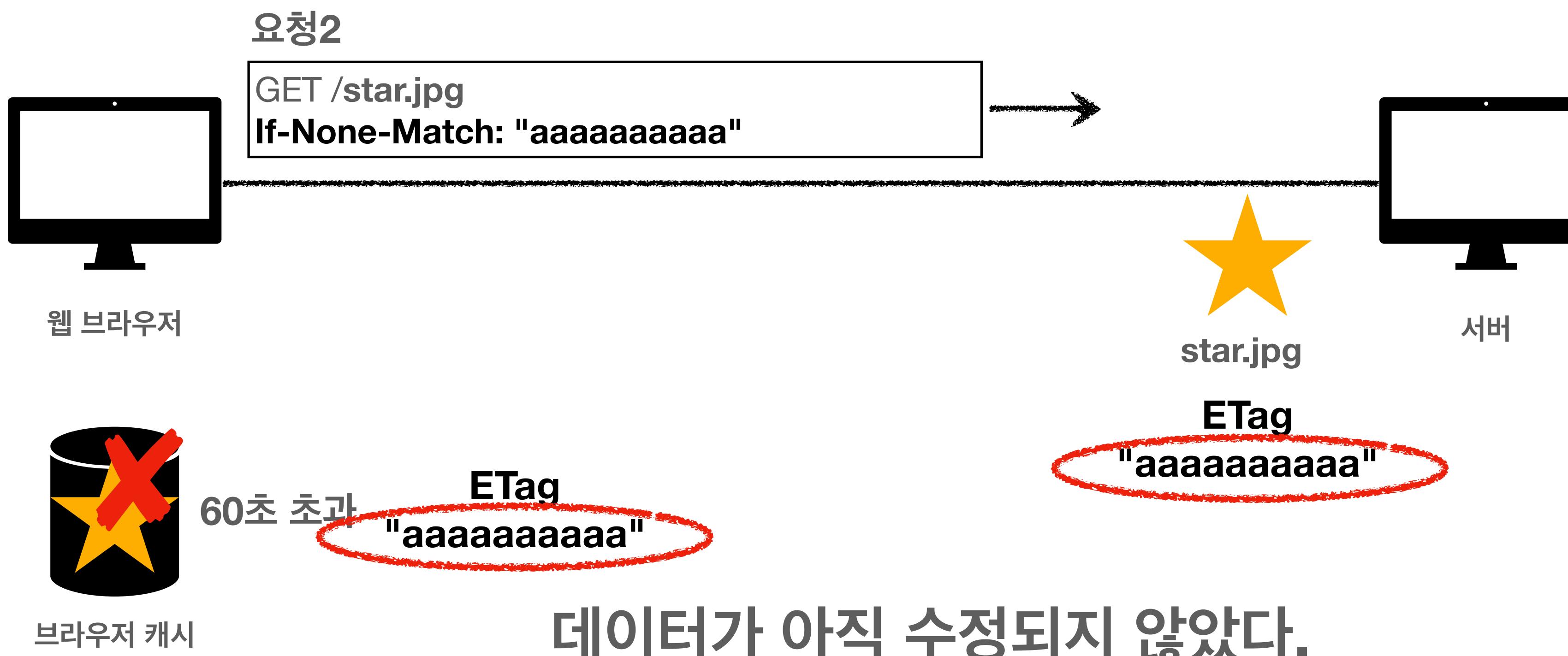
검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



검증 헤더 추가

두 번째 요청 - 캐시 시간 초과

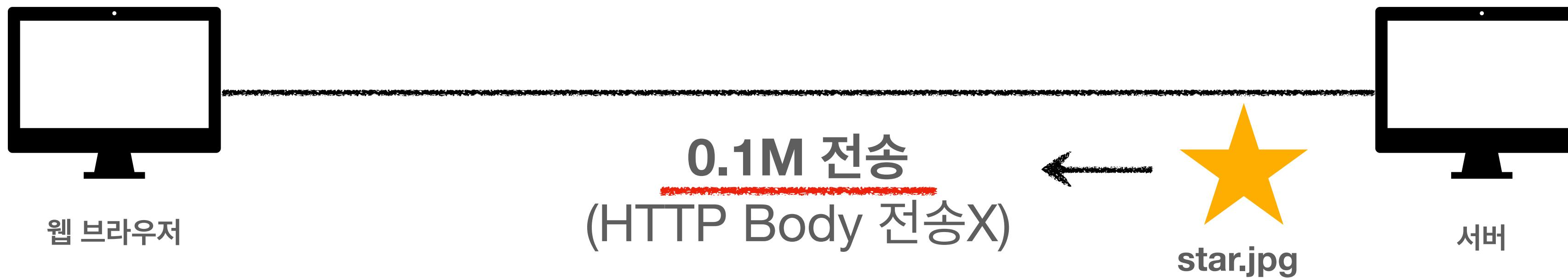


검증 헤더 추가

두 번째 요청 - 캐시 시간 초과

HTTP/1.1 304 Not Modified
Content-Type: image/jpeg
cache-control: max-age=60
ETag: "aaaaaaaaaa"
Content-Length: 34012

HTTP Body가 없음



60초 초과

ETag
"aaaaaaaaaa"

브라우저 캐시

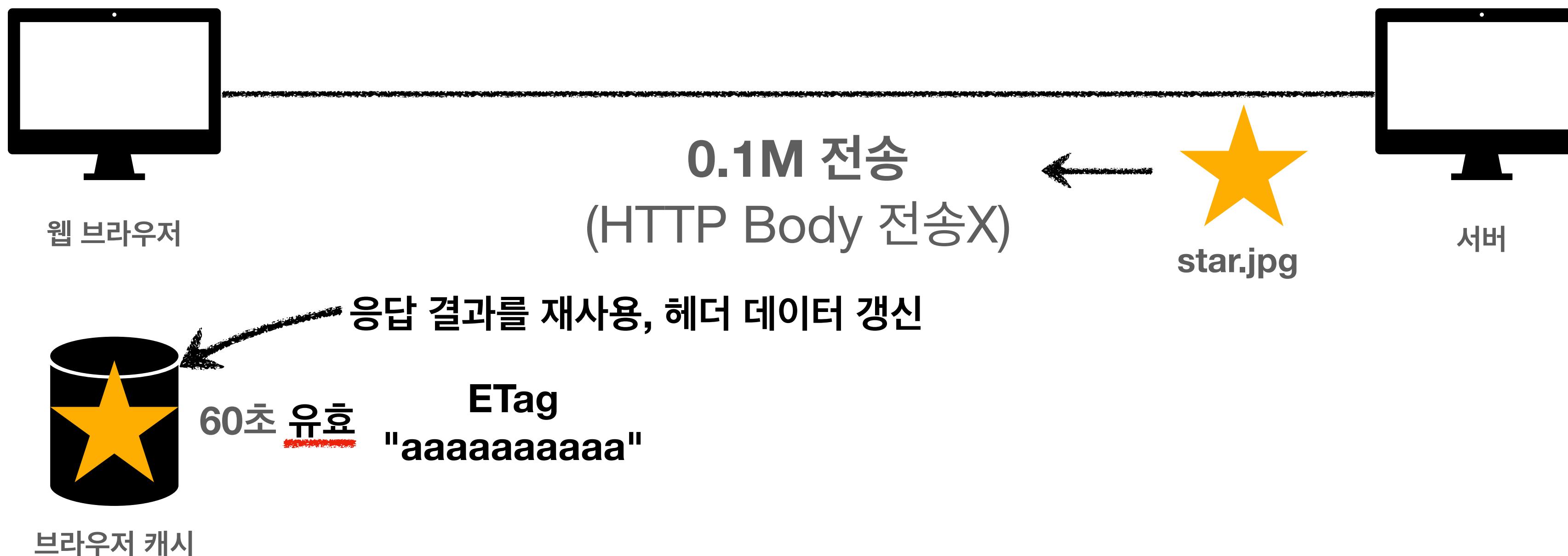
1.1M 가정
HTTP 헤더: 0.1M
HTTP 바디: 1.0M

검증 헤더 추가

두 번째 요청 - 캐시 시간 초과

HTTP/1.1 304 Not Modified
Content-Type: image/jpeg
cache-control: max-age=60
ETag: "aaaaaaaaaa"
Content-Length: 34012

HTTP Body가 없음



검증 헤더 추가

두 번째 요청 - 캐시 시간 초과



검증 헤더와 조건부 요청

ETag, If-None-Match 정리

- 진짜 단순하게 ETag만 서버에 보내서 같으면 유지, 다르면 다시 받기!
- 캐시 제어 로직을 서버에서 완전히 관리
- 클라이언트는 단순히 이 값을 서버에 제공(클라이언트는 캐시 메커니즘을 모름)
- 예)
 - 서버는 배타 오픈 기간인 3일 동안 파일이 변경되어도 ETag를 동일하게 유지
 - 애플리케이션 배포 주기에 맞추어 ETag 모두 갱신

캐시와 조건부 요청 헤더

캐시 제어 헤더

- Cache-Control: 캐시 제어
- Pragma: 캐시 제어(하위 호환)
- Expires: 캐시 유효 기간(하위 호환)

Cache-Control

캐시 지시어(directives)

- Cache-Control: max-age
 - 캐시 유효 시간, 초 단위
- Cache-Control: no-cache
 - 데이터는 캐시해도 되지만, 항상 원(origin) 서버에 검증하고 사용
- Cache-Control: no-store
 - 데이터에 민감한 정보가 있으므로 저장하면 안됨
(메모리에서 사용하고 최대한 빨리 삭제)

Pragma

캐시 제어(하위 호환)

- Pragma: no-cache
- HTTP 1.0 하위 호환

Expires

캐시 만료일 지정(하위 호환)

- expires: Mon, 01 Jan 1990 00:00:00 GMT
- 캐시 만료일을 정확한 날짜로 지정
- HTTP 1.0 부터 사용
- 지금은 더 유연한 Cache-Control: max-age 권장
- Cache-Control: max-age와 함께 사용하면 Expires는 무시

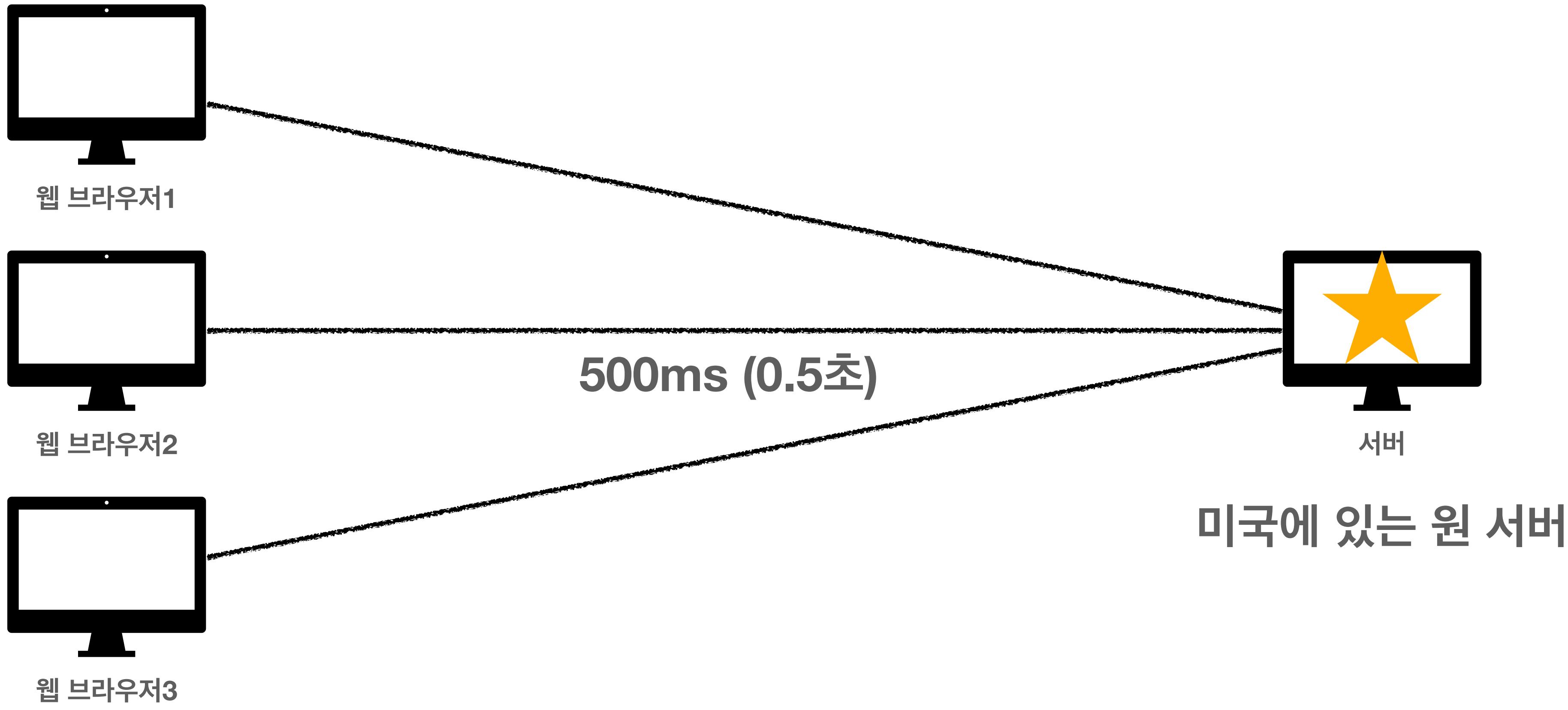
검증 헤더와 조건부 요청 헤더

- 검증 헤더 (**Validator**)
 - **ETag**: "v1.0", **ETag**: "asid93jkrh2l"
 - **Last-Modified**: Thu, 04 Jun 2020 07:19:24 GMT
- 조건부 요청 헤더
 - If-Match, If-None-Match: ETag 값 사용
 - If-Modified-Since, If-Unmodified-Since: Last-Modified 값 사용

프록시 캐시

원 서버 직접 접근

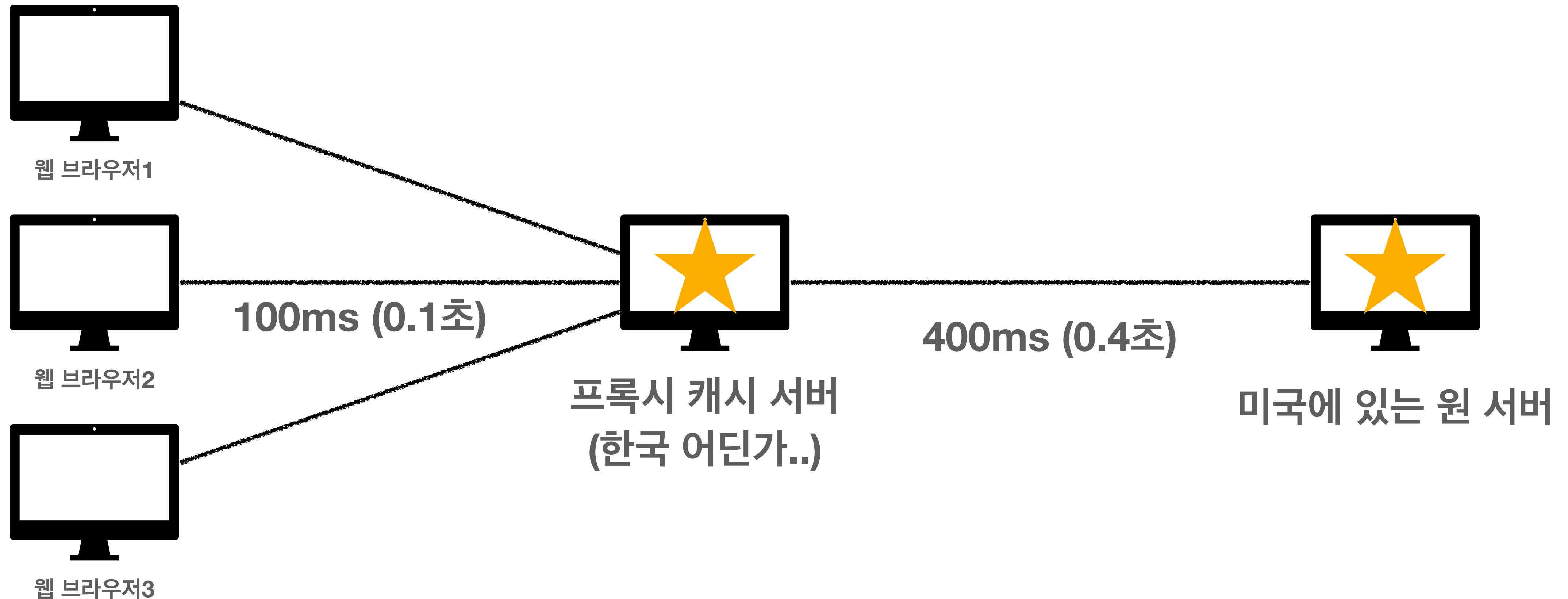
origin 서버



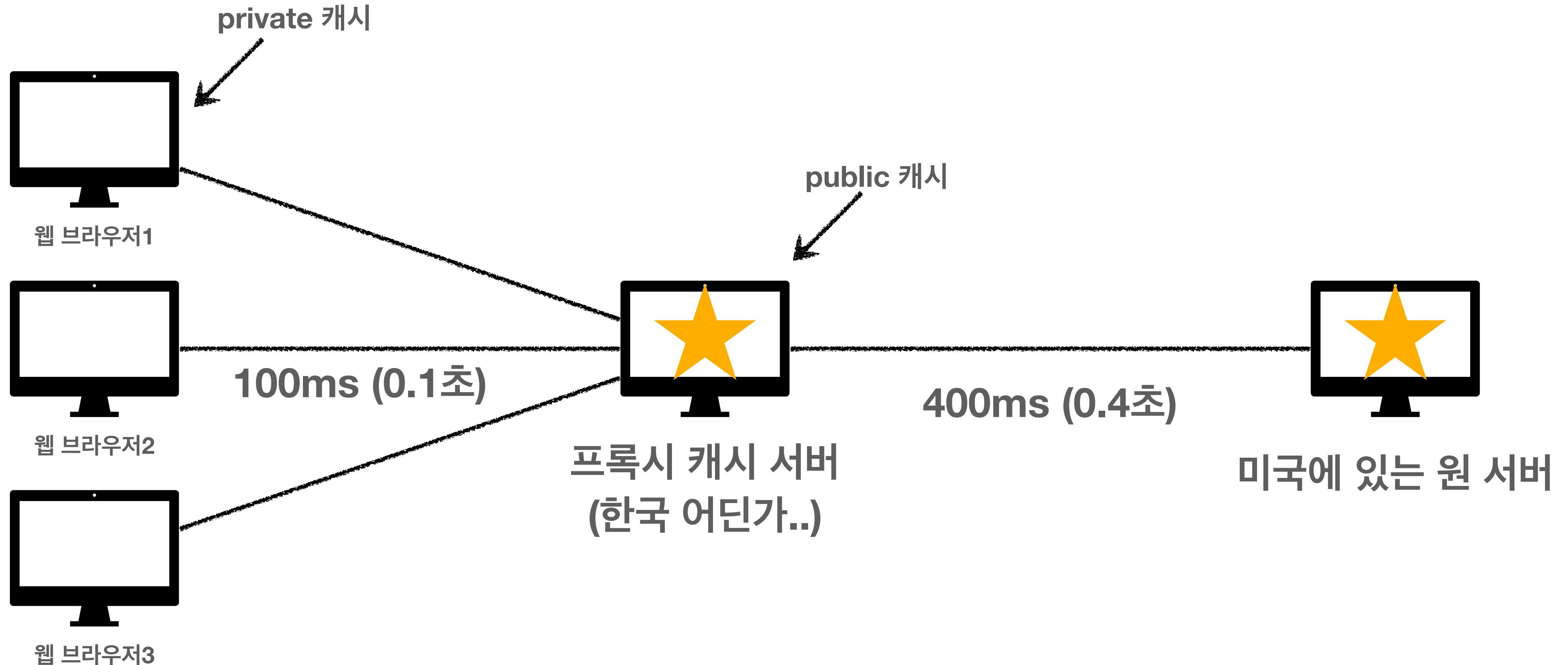
한국에 있는 클라이언트

프록시 캐시 도입

첫 번째 요청



프록시 캐시 도입



Cache-Control

캐시 지시어(directives) - 기타

- **Cache-Control: public**
 - 응답이 public 캐시에 저장되어도 됨
- **Cache-Control: private**
 - 응답이 해당 사용자만을 위한 것임, private 캐시에 저장해야 함(기본값)
- **Cache-Control: s-maxage**
 - 프록시 캐시에만 적용되는 max-age
- **Age: 60** (HTTP 헤더)
 - 오리진 서버에서 응답 후 프록시 캐시 내에 머문 시간(초)

캐시 무효화

Cache-Control

확실한 캐시 무효화 응답

- **Cache-Control: no-cache, no-store, must-revalidate**
- **Pragma: no-cache**
 - HTTP 1.0 하위 호환

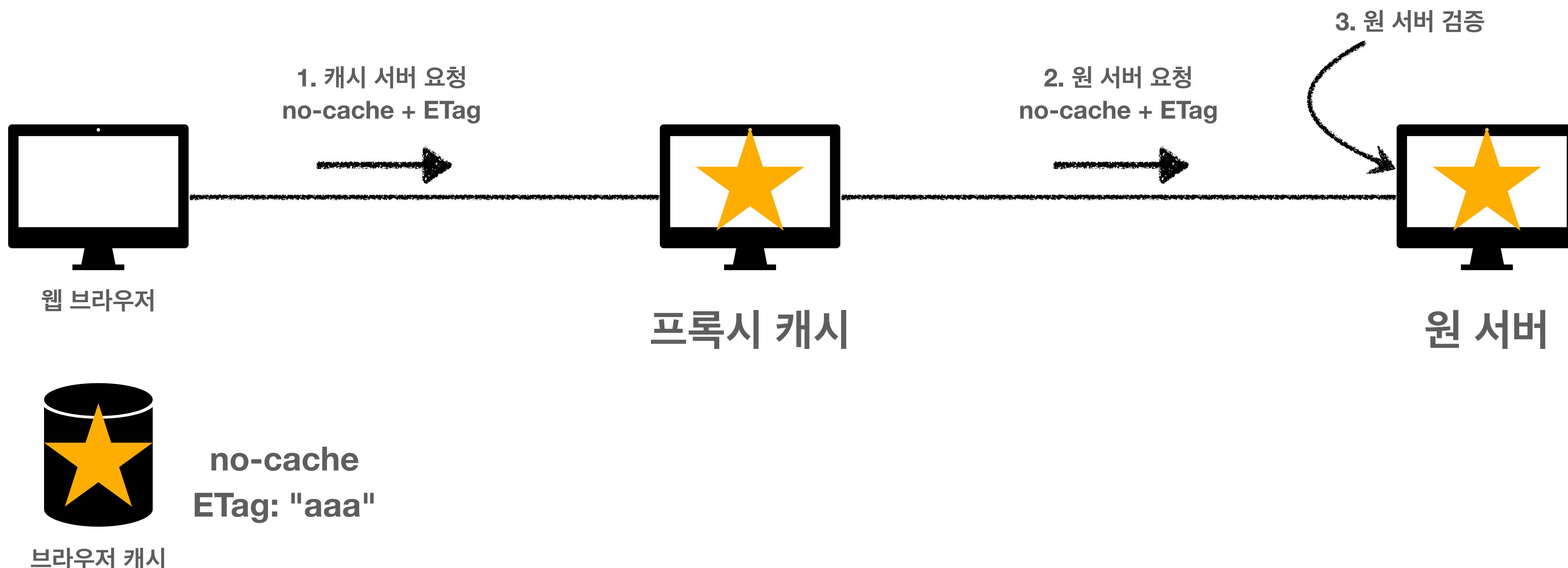
Cache-Control

캐시 지시어(directives) - 확실한 캐시 무효화

- **Cache-Control: no-cache**
 - 데이터는 캐시해도 되지만, 항상 원 서버에 검증하고 사용(이름에 주의!)
- **Cache-Control: no-store**
 - 데이터에 민감한 정보가 있으므로 저장하면 안됨
(메모리에서 사용하고 최대한 빨리 삭제)
- **Cache-Control: must-revalidate**
 - 캐시 만료후 최초 조회시 원 서버에 검증해야함
 - 원 서버 접근 실패시 반드시 오류가 발생해야함 - 504(Gateway Timeout)
 - must-revalidate는 캐시 유효 시간이라면 캐시를 사용함
- **Pragma: no-cache**
 - HTTP 1.0 하위 호환

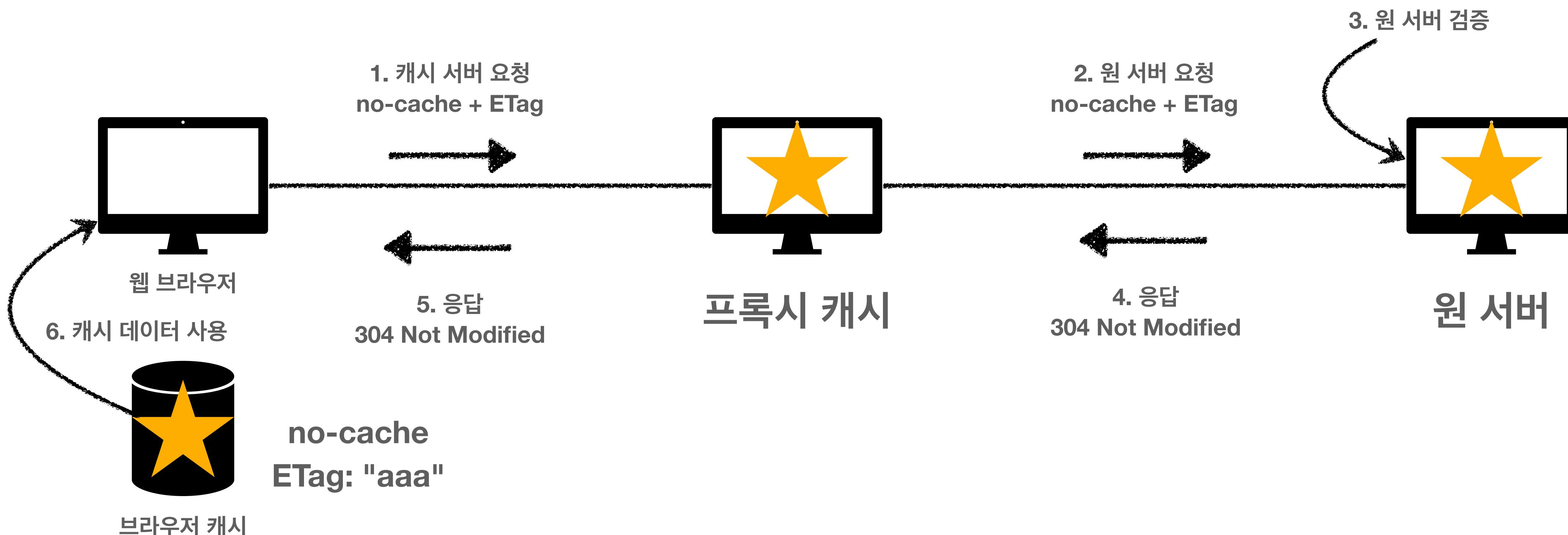
no-cache vs must-revalidate

no-cache 기본 동작



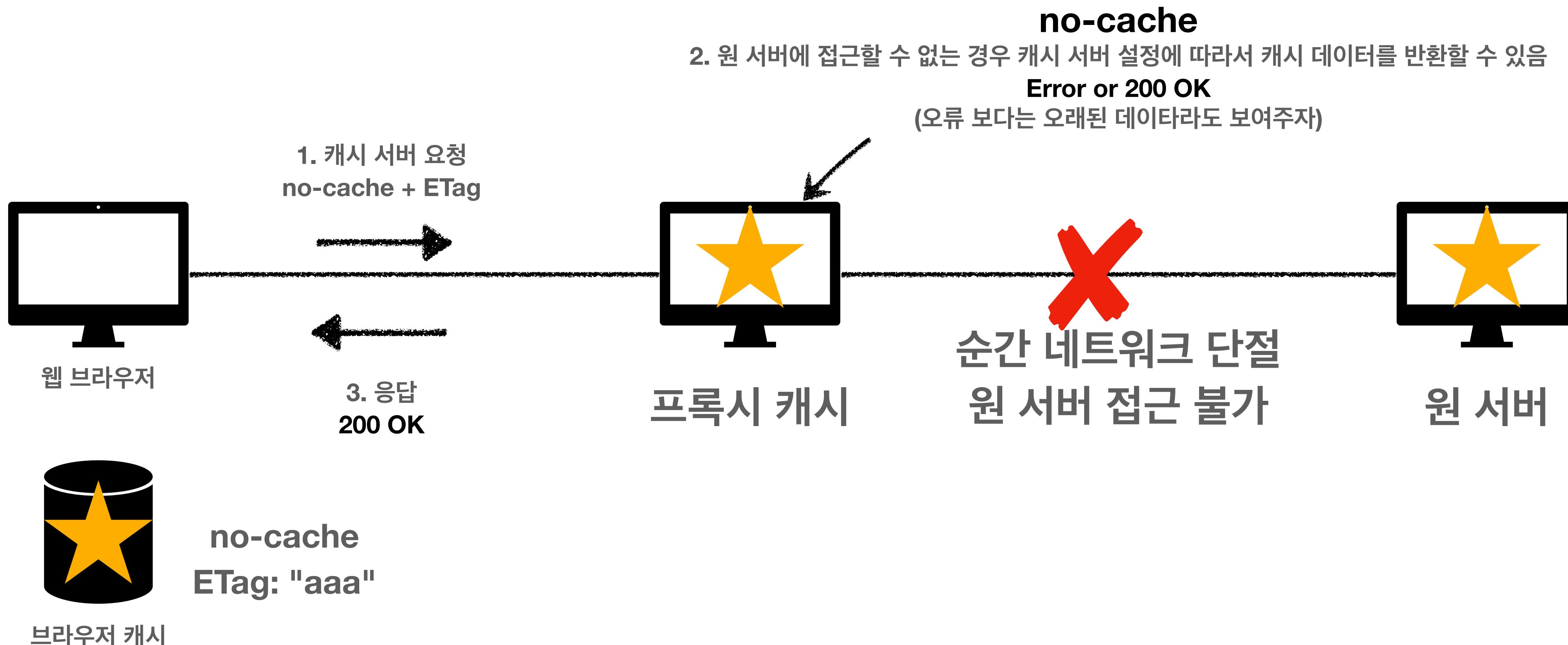
no-cache vs must-revalidate

no-cache 기본 동작



no-cache vs must-revalidate

no-cache



no-cache vs must-revalidate

must-revalidate

