Chap 02 네트워크_2.5 HTTP

2.5 HTTP

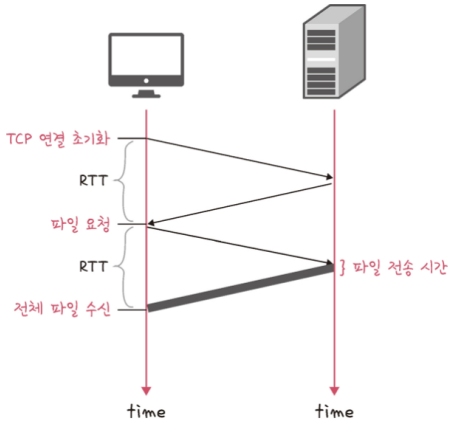
> HTTP는 애플리케이션 계층.
> 웹 서비스 통신에 사용
> HTTP/1.0 부터 시작, 현재는 HTTP/3

2.5.1 HTTP/1.0

= 하나의 연결당 하나의 요청 처리

→ RTT 증가

✅ RTT(Round Trip Time) 증가



▲ 그림 2-54 RTT 증가

= 패킷이 목적지에 도달하고 나서 다시 출발지로 돌아오기까지 걸리는 시간

= 패킷 왕복 시간

⇒ RTT 증가 → 서버 부담, 사용자 응답 시간 길어짐
⇒ 해결 방안 : 스플리팅, 코드 압축, 이미지 Base64 인코딩

✅ 해결1) 이미지 스플리팅

: 작은 아이콘들을 각각의 이미지 파일로 만들 경우 과부하 발생

→ 하나의 이미지 파이로 합쳐 사용

```
import java.awt.image.BufferedImage;
import java.io.File;
import javax.imageio.ImageIO;

public class ImageSplitting {
    public static void main(String[] args) throws Exception {
        File file = new File("large_image.jpg"); // 원본 이미지 파일 경로
        BufferedImage image = ImageIO.read(file);

        int rows = 4; // 행 수
        int cols = 4; // 열 수
        int chunks = rows * cols;

        int chunkWidth = image.getWidth() / cols;
        int chunkHeight = image.getHeight() / rows;

        int count = 0;
        BufferedImage[] imgs = new BufferedImage[chunks];
        for (int x = 0; x < rows; x++) {
            for (int y = 0; y < cols; y++) {
                imgs[count] = new BufferedImage(chunkWidth, chunkHeight, imag

                // 이미지 부분을 잘라서 배열에 저장
                int dx = chunkWidth * y;
                int dy = chunkHeight * x;
                for (int i = 0; i < chunkWidth; i++) {
                    for (int j = 0; j < chunkHeight; j++) {
                        imgs[count].setRGB(i, j, image.getRGB(dx + i, dy + j)
                    }
                }
                count++;
```

```
            }
        }

        // 분할된 이미지 저장
        for (int i = 0; i < imgs.length; i++) {
            ImageIO.write(imgs[i], "jpg", new File("img" + i + ".jpg"));
        }

        System.out.println("이미지 분할 완료!");
    }
}
```

✅ 해결2) 코드 압축

: 코드 압축을 통해 개행문자, 빈칸 없앰 → 코드의 용량 자체를 줄이는 방식

자바스크립트

```
const express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

앞의 코드를 다음과 같은 코드로 바꾸는 방법입니다.

자바스크립트

```
const express=require("express"),app=express(),port=3e3;app.get("/",(e,p)=>{p.send("Hello World!")}),app.listen(3e3,()=>{console.log("Example app listening on port 3000")});
```

✅ 해결3) 이미지 Base64 인코딩

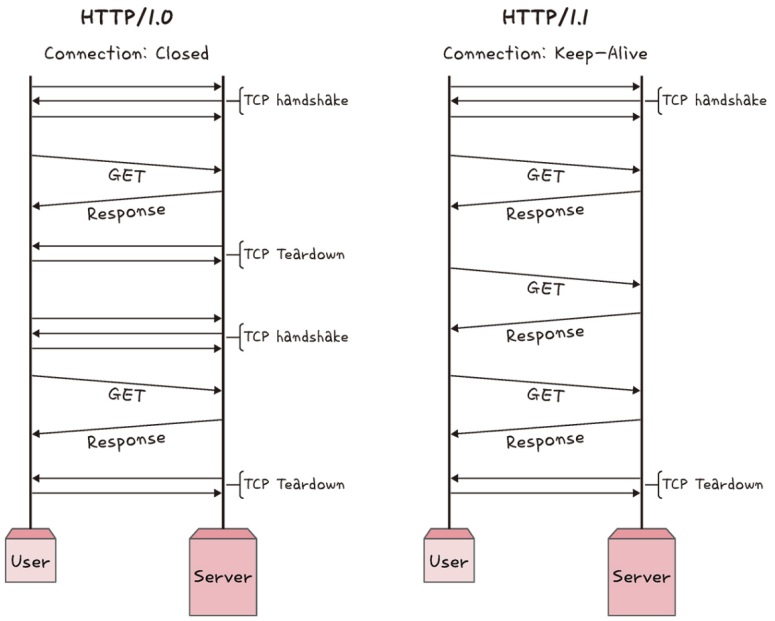
: 이미지 파일을 64진법으로 이루어진 문자열로 인코딩하는 방법

: [장점] 서버와의 연결을 열고 이미지에 대해 서버에 HTTP 요청 필요 없음

: [단점] Base64 문자열로 변환할 경우 37%정도 크기가 커짐

2.5.2 HTTP/1.1

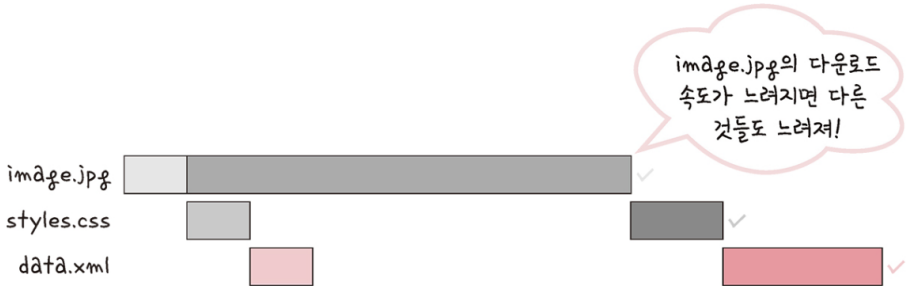
: 매번 TCP 연결 x → 한번 TCP 초기화 이후 keep-alive라는 옵션으로 여러 개의 파일을 송수신할 수 있도록함



▲ 그림 2-55 HTTP/1.0과 HTTP/1.1의 비교

→ 문서 안에 포함된 다수의 리소스를 처리 → 요청할 리소스 개수에 비례해서 대기 시간 길어질 수 있음
ex) HOL Blocking, 무거운 헤더 구조

✅ 문제점1) HOL Blocking



▲ 그림 2-56 HOL Blocking

= Head of Line Blocking

= 네트워크에서 같은 큐에 있는 패킷이 그 첫 번째 패킷에 의해 지연될 때 발생하는 성능 저항 현상

✅ 문제점2) 무거운 헤더 구조

헤더에 쿠키 등등 많은 메타 데이터 들어 있음 → 압축 되지 않아 무거움

2.5.3 HTTP/2

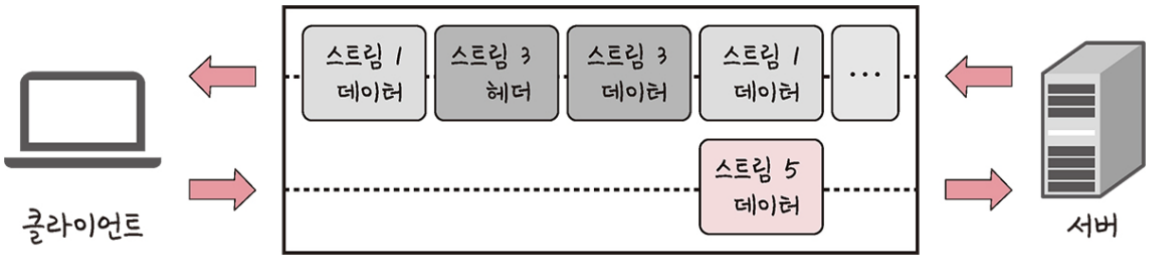
HTTP/1.x 보다 지연시간 단축, 응답 시간 축소, 멀티플렉싱/헤더압축/서버푸시/요청의 우선순위 처리 지원

✅ 멀티플렉싱 (HOL Blocking 해결)

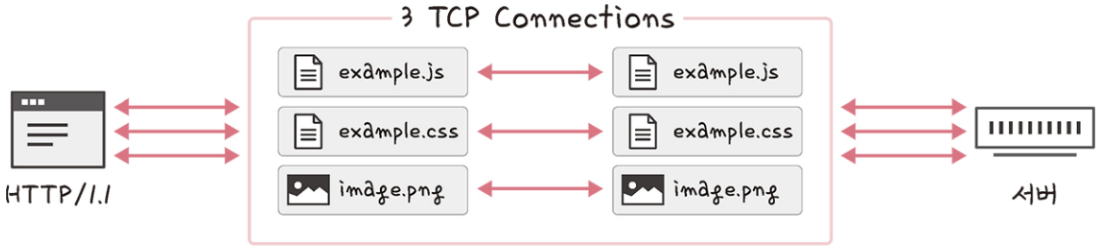
= 여러개의 스트림을 사용하여 송수신

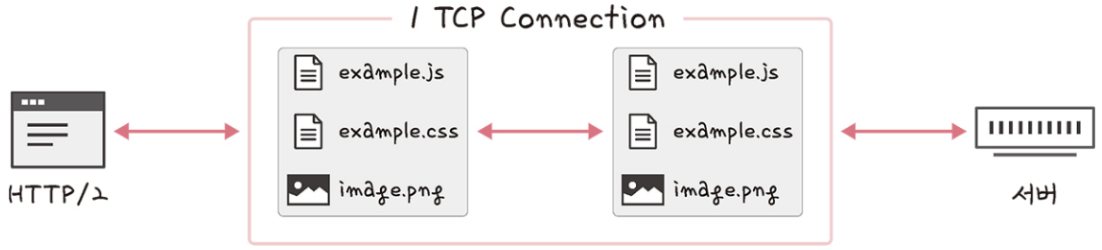
= 특정 스트림의 패킷 손실 → 나머지 스트림은 멀쩡하게 동작

*스트림 : 시간이 지남에 따라 사용할 수 있게 되는 일련의 데이터 요소를 가리키는 데이터 흐름



→ 하나의 연결 내 여러 스트림





→ 애플리케이션에서 받아온 메세지를 독립된 프레임으로 조각내어 서로 송수신한 이후 다시 조립

→ HOL Blocking 문제 해결

*HOL Blocking이란?

`HTTP/1.1`의 요청-응답 쌍은 항상 순서를 유지하고 동기적으로 수행되어야 한다.

구체적으로 1개의 TCP 커넥션 상에서 3개의 이미지 (`a.png`, `b.png`, `c.png`)를 받는 경우, HTTP 리퀘스트는 다음과 같이 된다.

```
|---a.png---|
            |---b.png---|
                        |---c.png---|
```

하나의 요청이 처리되고 응답을 받은 후에 다음 요청을 보낸다.

이전의 요청이 처리되지 않았다면 그 다음 요청은 보낼 수 없다는 것이다.

만약 `a.png`의 요청이 막혀버리게 되면 `b,c`가 아무리 빨리 처리될 수 있더라도 전체적으로 느려지게 된다.

```
|------------a.png------------|
                              |-b.png-|
                                      |---c.png---|
```

이것이 바로 `HTTP/1.1`의 HOL Blocking이다.

`HTTP/1.1`의 `pipelining`이라는 사양은 (조건부로) 요청만 먼저 보내버리는 것으로, 이 문제를 회피하는 것처럼 보인다.

그러나 응답을 보낸 순서대로 무조건 받아야 하므로 `a.png`가 막혔을 경우에 생각보다 큰 효과를 보기 어렵다.

`HTTP/2`의 경우 요청은 하나의 연결에서 병렬적으로 보내질 수 있다.

즉, `a ~ c.png`가 모두 병렬적으로 요청되고, 응답된다는 것이다.

따라서 `a.png`가 시간이 걸리는 처리에서도, `b,c.png`는 먼저 받아서 보여줄 수 있다는 것이다.

```
|------------a.png------------|
|-b.png-|
|---c.png---|
```

따라서 `HTTP/1.1`에서의 HOL Blocking은 `HTTP/2`에서는 발생하지 않는다고 말할 수 있다.

또한 `HTTP/2`는 접속의 `Flow Control`과 중요한 자원의 우선순위를 부여하는 `Priority`를 가지고 있기 때문에 세세한 제어가 가능하다.

✅ 헤더 압축

▲ 그림 2-59 헤더 압축

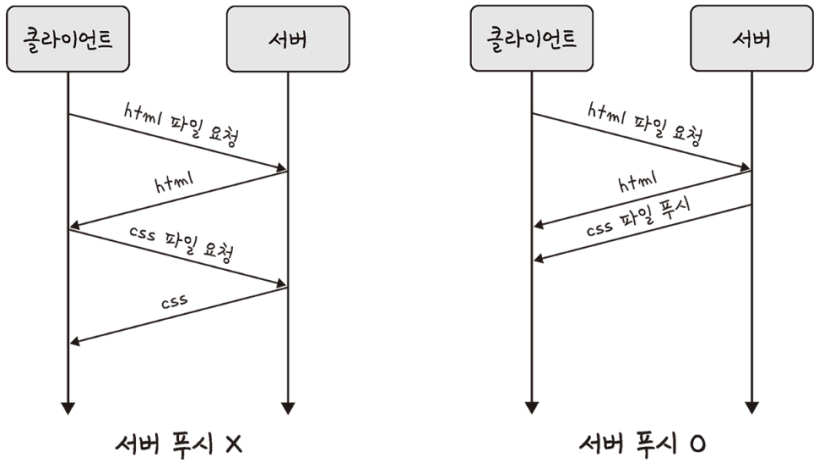
HTTP/1.x ⇒ 헤더 크기가 큼 (대역폭 사용 증가, 응답 지연, 리소스 낭비)

HTTP/2 ⇒ 헤더 압축 사용 (허프만 코딩 압축 알고리즘을 사용하는 HPACK 압축 형식 가짐)

허프만 코딩이란?

입력 파일의 문자 빈도 수를 가지고 최소힙을 이용해 파일을 압축하는 과정

✅ 서버 푸시



▲ 그림 2-60 서버 푸시

HTTP/1.1 ⇒ 클라이언트가 서버에 요청해야 파일 다운로드 가능

HTTP/2 ⇒ 클라이언트 요청 없이 서버가 바로 리소스 푸쉬 가능

클라이언트가 리소스를 요청하기 전에 서버가 이를 미리 전송 → 리소스 로딩 시간 단축, 페이지 로딩 성능 향상

2.5.4 HTTPs

= TCP위에서 돌아감

= 애플리케이션 계층과 전송 계층 사이에 신뢰 계층인 SSL/TLS 계층을 넣은 신뢰할 수 있는 HTTP 요청

= 통신 암호화

= 스누핑 방지

✅ SSL/TLS

= 전송 계층에서 보안을 제공한느 프로토콜

= 클라이언트와 서버가 통신시 제3자가 메시지 도청하거나 변조하지 못하게 함

→ 인터셉터 방지

= 보안 세션 기반으로 데이터 암호화 → 핸드 세이크를 통해 보안 세션 생성, 상태 정보 공유

보안 세션

인증 메너키즘

: CA에서 발급한 잉ㄴ증서 기반으로 이루어짐

→ 안전한 연결을 위해 공개키를 클라이언트에 제공

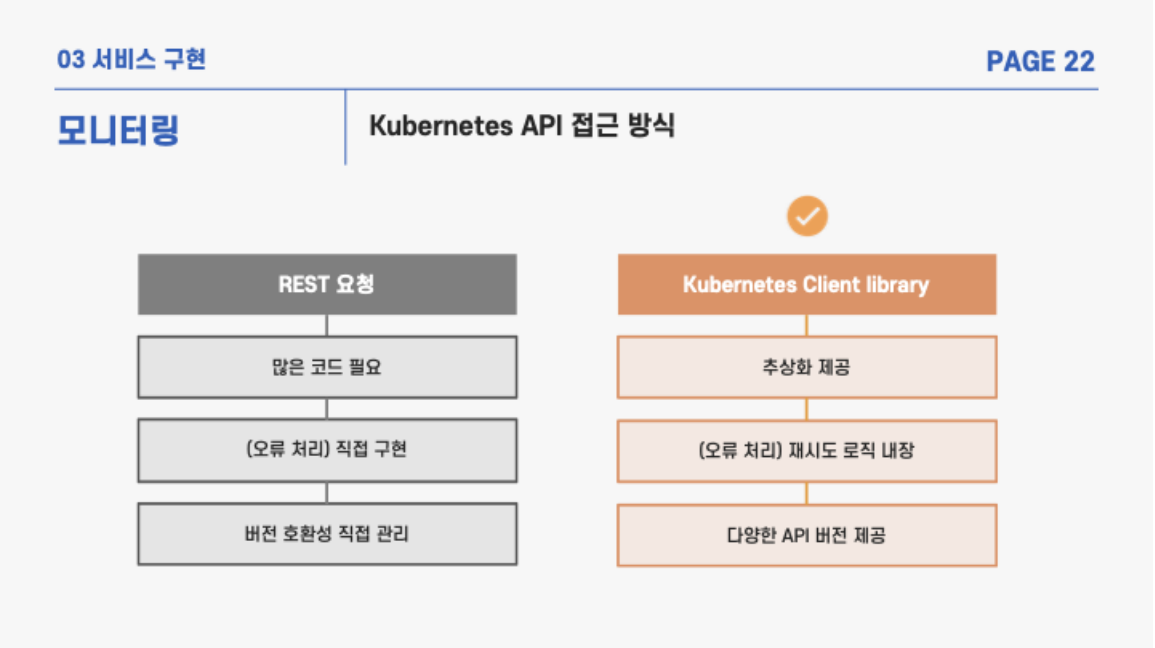
→ 인증서 = 서비스 정보 + 공개키 + 지문 + 디지털 서명

→ 공인 기업만 참여 가능

CA 발급 과정

1. 자신의 사이트 정보와 공개키를 CA에 제출
2. CA는 공개키를 해시한 값인 지문을 사용하는 CA의 비밀키 등을 기반으로 CA 인증서 발급

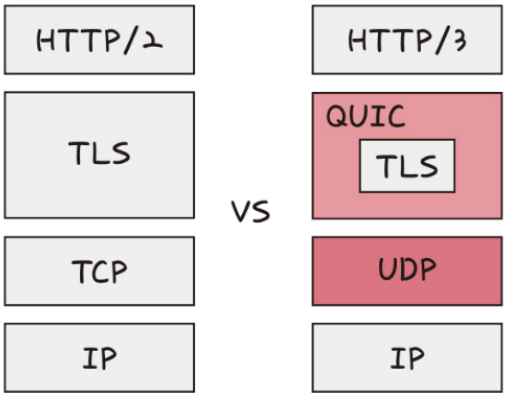
CA 발급 사용



2.5.5 HTTP

= QUIC 계층 위에서 돌아감

= UDP 기반



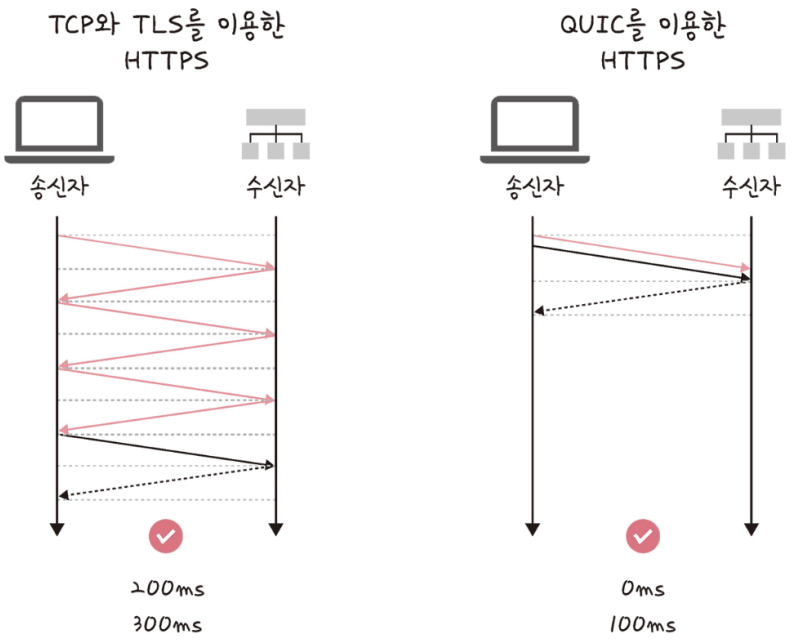
▲ 그림 2-68 UDP 기반으로 돌아가는 HTTP/3

= 멀티 플렉싱 기능 가짐

= 초기 연결 설정 시 지연 시간 감소

✅ 초기 연결 설정 시 지연 시간 감소

QUIC는 TCP를 사용하지 않기 때문에 통신 시작시 번거로운 3-웨이 핸드셰이크 과정 거치지 않음



▲ 그림 2-69 RTT의 감소