BACKEND CS · STUDY GUIDE

네트워크 7강

OSI · TCP/UDP · HTTP · TLS · DNS

7강 · 2부 구성

ttukttak-coding.dev

2026

이 책은

이 책은 백엔드 개발자가 알아야 할 네트워크 기초 7개 주제를 한 권으로 정리한 학습서입니다. OSI 7계층에서 시작해 IPv4/IPv6, TCP/UDP, TCP 핸드셰이크, HTTP 버전 차이, HTTPS · TLS, DNS까지 — 요청 한 번이 어떻게 흘러가는지를 차근히 따라갑니다.

각 강의는 "왜 이게 이렇게 동작하는가?" 를 묻습니다. RFC 정의를 외우는 게 아니라, 패킷이 실제로 어떤 경로로 어떤 헤더를 들고 가는지를 그림으로 그릴 수 있게 만드는 게 목표입니다.

1부는 네트워크 기초(OSI · TCP/IP), 2부는 HTTP와 보안 계층입니다. 면접 단골 주제이자 장애 대응에서 매번 다시 펴보게 되는 영역만 모았습니다.

> **구성**: 2부 7강 / **대상**: 네트워크 기초를 한 번에 정리하고 싶은 백엔드 개발자 / **블로그**: 더 많은 글은 ttukttak-coding.dev 에서 보실 수 있습니다.

목차

1부. 네트워크 기초 — OSI에서 TCP까지

2부. HTTP와 보안

PART 1

1부. 네트워크 기초 — OSI에서 TCP까지

Chapter 1

네트워크 계층 완전 정복 — OSI 7계층과 TCP/IP를 요청 흐름으로 이해하기

#TCP/IP #OSI 7계층

OSI 7계층과 TCP/IP 4계층이 무엇이 다른지, 브라우저 요청 하나가 각 계층을 어떻게 통과하는지, 실무에서 계층별로 장애를 어떻게 나눠 보는지 정리합니다.

네트워크 계층, 왜 알아야 하나요?

백엔드 개발을 하다 보면 "API가 느리다", "연결이 안 된다", "간헐적으로 timeout 난다" 같은 말을 자주 듣습니다. 그런데 이런 증상은 원인이 전혀 다를 수 있습니다.

- DNS 조회가 느린 것일 수 있습니다
- TCP 연결 자체가 늦게 맺히는 것일 수 있습니다
- TLS handshake에서 인증서나 암호 협상이 꼬인 것일 수 있습니다
- 라우팅이나 방화벽 때문에 패킷이 중간에서 막히는 것일 수 있습니다
- 애플리케이션은 멀쩡한데 HTTP 레벨에서 잘못된 응답을 보내는 것일 수 있습니다

이때 유용한 기준이 `OSI 7계층`과 `TCP/IP` 모델입니다.

중요한 점은 **계층 모델을 외우는 것 자체가 목적은 아니라는 것**입니다. 진짜 목적은 이것입니다.

> **통신 문제를 어느 층에서 봐야 하는지 나누는 기준을 갖는 것**

이 글에서는 `OSI 7계층`과 `TCP/IP 4계층`을 각각 따로 암기하지 않고, **브라우저에서 서버까지 요청 하나가 실제로 어떻게 내려가고 올라오는지**를 기준으로 정리하겠습니다.

> **기준:** 이 글은 일반적인 `HTTP/1.1` 또는 `HTTP/2` 기반 웹 요청, 즉 `HTTP over TLS over TCP/IP` 흐름을 기준으로 설명합니다. `HTTP/3`는 `QUIC`을 통해 `UDP` 위에서 동작하므로 전송 계층 해석이 조금 달라질 수 있습니다.

먼저 큰 그림부터 보면

`OSI 7계층`과 `TCP/IP`는 경쟁 관계가 아니라, **같은 통신 과정을 다른 해상도로 설명하는 모델**이라고 보면 이해가 쉽습니다.

관점	OSI 7계층	TCP/IP 4계층
목적	통신 과정을 계층별로 세분화해 설명하는 참조 모델	실제 인터넷 프로토콜 묶음을 설명하는 실용 모델
계층 수	7	4
강점	역할을 더 세밀하게 나눠서 설명하기 좋음	현실의 프로토콜 스택과 더 가깝고 실무 설명에 자주 쓰임
대표 키워드	Application, Presentation, Session, Transport, Network, Data Link, Physical	Application, Transport, Internet, Network Access
실무에서 자주 듣는 표현	`L4`, `L7`, `L2 스위치`, `L3 라우터`	HTTP, TCP, UDP, IP, Ethernet 같은 실제 스택 설명

핵심만 먼저 말하면:

- **OSI 7계층은 설명용 지도가 더 가깝습니다**
- **TCP/IP는 실제 인터넷이 돌아가는 방식에 더 가깝습니다**
- 그래서 실무에서는 두 모델을 섞어 말하는 경우가 많습니다

예를 들어:

- "이 로드밸런서는 `L4`에서 동작한다"
- "이 프록시는 `L7`까지 본다"
- "이 서비스는 `TCP/IP` 기준으로 `TCP` 위에 `HTTP`가 올라간다"

이런 표현은 모두 같은 통신 과정을 서로 다른 관점에서 설명하는 것입니다.

Phase 1. 왜 굳이 계층으로 나눌까?

계층 모델이 필요한 이유는 단순합니다. 통신은 한 번에 하나의 기술로 끝나지 않기 때문입니다.

브라우저에서 `https://example.com`에 요청을 보낸다고 가정해 보겠습니다. 이 요청 하나 안에는 사실 여러 역할이 겹쳐 있습니다.

- 어떤 형식으로 요청을 표현할 것인가? → `HTTP`
- 어떤 상대 프로세스와 통신할 것인가? → `TCP` 포트
- 목적지 호스트까지 어떻게 보낼 것인가? → `IP`
- 같은 네트워크 구간에서 실제로 어떤 장치에게 넘길 것인가? → `MAC`, `Ethernet`, `Wi-Fi`
- 전기 신호나 무선 신호로 어떻게 내보낼 것인가? → 물리 계층

이걸 하나의 거대한 프로토콜로 만들면 구현도 어렵고, 교체도 어렵고, 장애 분석도 매우 힘들어집니다. 그래서 역할을 층으로 나눕니다.

```
브라우저 요청
  ↓
HTTP 메시지 생성
  ↓
TCP가 포트와 순서를 관리
  ↓
IP가 목적지까지 라우팅
  ↓
Ethernet/Wi-Fi가 같은 링크의 다음 장치로 전달
  ↓
전기/무선 신호로 전송
```

계층을 나누면 얻는 장점은 분명합니다.

- **역할 분리** — 각 계층은 자기 책임에 집중합니다
- **표준화** — 아래 계층 구현이 바뀌어도 위 계층은 덜 흔들립니다
- **교체 가능성** — 애플리케이션은 `Ethernet`인지 `Wi-Fi`인지 몰라도 됩니다
- **장애 분리** — 어디서 문제가 났는지 더 체계적으로 좁힐 수 있습니다

즉, 계층 모델은 이론 장난이 아니라 **복잡한 통신을 분해해서 다루기 위한 공통 언어**입니다.

Phase 2. OSI 7계층을 한 층씩 보면

`OSI 7계층`은 통신 과정을 가장 세밀하게 나눈 모델입니다. 아래에서 위로 올라가며 보면 다음과 같습니다.

L1. Physical Layer

가장 아래 층입니다. 비트를 실제 신호로 보내는 역할을 합니다.

- 케이블, 커넥터, 전파, 광신호 같은 물리 매체
- 전기적/기계적 특성
- 실제 0과 1을 어떻게 흘려보낼지

예를 들어 랜 케이블이 불량이거나, Wi-Fi 신호가 너무 약하거나, 포트 자체가 죽어 있으면 이 층의 문제일 가능성이 큽니다.

L2. Data Link Layer

같은 네트워크 구간 안에서 **바로 다음 장치까지** 데이터를 전달하는 층입니다.

- `MAC` 주소 사용
- `Ethernet`, `Wi-Fi` 프레임 처리
- 스위치가 주로 이 계층에서 동작
- 오류 검출, 프레임 단위 전달

핵심은 L2가 "전 세계 목적지"를 찾는 것이 아니라, **현재 링크에서 누구에게 넘길지**를 다룬다는 점입니다.

예를 들어 같은 사내망 안에서 스위치나 VLAN 구성이 잘못되면 L2 문제가 될 수 있습니다.

L3. Network Layer

다른 네트워크까지 포함해서 **목적지 호스트까지 경로를 찾아가는 층**입니다.

- `IP` 주소 사용
- 라우터가 주로 이 계층에서 동작
- 패킷 전달과 라우팅
- `ICMP`, `TTL` (IPv6에서는 `Hop Limit`) 같은 개념

우리가 흔히 말하는 "서버 IP", "라우팅", "서브넷", "게이트웨이"가 이 층과 강하게 연결됩니다.

즉:

- L2는 같은 링크 안에서 다음 장치를 찾고
- L3는 여러 네트워크를 거쳐 최종 목적지까지 가는 길을 찾습니다

L4. Transport Layer

호스트와 호스트가 아니라, 더 정확히는 **프로세스와 프로세스 사이의 통신**을 다루는 층입니다.

- 포트 번호 사용
- `TCP`, `UDP`
- 신뢰성, 순서 보장, 재전송, 흐름 제어
- 어떤 애플리케이션 프로세스로 데이터를 전달할지 구분

예를 들어:

- `443` 포트는 HTTPS 서버 프로세스
- `3306` 포트는 MySQL 서버 프로세스

처럼 이해할 수 있습니다.

`TCP`는 연결 지향적이고, 순서 보장과 재전송을 제공합니다. 반면 `UDP`는 더 단순하고 가볍지만, 전송 보장은 애플리케이션이 직접 책임질 수 있습니다.

L5. Session Layer

통신 세션의 생성, 유지, 종료 같은 대화 흐름을 다루는 층입니다.

- 연결 상태 유지
- 대화 동기화
- 세션 재개 같은 개념

다만 현대 웹 애플리케이션에서는 이 층이 독립적인 프로토콜로 또렷하게 드러나기보다, 애플리케이션 프레임워크나 `TLS`, 인증 메커니즘, RPC 계층 안으로 흡수된 경우가 많습니다.

그래서 시험에서는 자주 보이지만, 실무에서는 "이건 정확히 세션 계층 문제다"라고 딱 잘라 말하는 경우는 많지 않습니다.

L6. Presentation Layer

데이터 표현 형식과 변환을 다루는 층입니다.

- 인코딩/디코딩
- 직렬화 포맷
- 압축
- 암호화

예를 들어:

- `UTF-8` 인코딩
- `JSON`, `XML`, `Protocol Buffers`
- 압축
- 암호화

같은 주제가 이 층과 연결됩니다.

`TLS`도 교육용 설명에서는 이 층과 연결해 다루는 경우가 많습니다. 다만 현대 인터넷 스택에는 "`TLS = OSI 6계층`"처럼 고정된 공식 매핑이 있는 것은 아니고, 보통은 **애플리케이션 위에서 동작하는 별도 보안 프로토콜**로 이해하는 편이 더 정확합니다.

다만 이것도 현대 시스템에서는 독립 계층이라기보다, 애플리케이션과 보안 계층에 섞여 있는 경우가 많습니다.

L7. Application Layer

사용자나 애플리케이션이 직접 다루는 가장 위 계층입니다.

- `HTTP`
- `DNS`

- `SMTP`
- `FTP`
- `SSH`

중요한 점은 여기서 말하는 Application이 "브라우저 앱", "서버 앱" 자체를 뜻한다기보다, **애플리케이션이 사용하는 네트워크 프로토콜 계층**이라는 것입니다.

예를 들어 `HTTP 500`, `404`, `Cookie`, `Header`, `Path`, `Method` 같은 것은 L7 수준 이야기입니다.

참고로 `HTTPS`는 실무에서 편의상 L7으로 묶어 부르기도 하지만, 엄밀히 보면 **`HTTP` 위에 `TLS`가 추가된 형태**입니다.

헷갈리기 쉬운 포인트

현실에서는 L5와 L6가 독립 장비나 독립 프로토콜 계층으로 선명하게 드러나지 않는 경우가 많습니다. 그래서 OSI 7계층은 이해용으로는 좋지만, 구현 관점에서는 다소 이상적인 모델로 느껴질 수 있습니다.

이 지점에서 `TCP/IP` 모델이 더 실용적으로 보이기 시작합니다.

Phase 3. TCP/IP 4계층은 실제로 어떻게 묶일까?

인터넷에서 실제로 자주 쓰는 설명은 `TCP/IP 4계층`입니다.

TCP/IP 계층	OSI 대응	대표 프로토콜	핵심 역할
Application	대체로 L7-L5에 걸침 (`RFC 1122` 관점에서는 L7/L6 중심)	HTTP, DNS, SMTP, TLS, SSH	애플리케이션 프로토콜과 데이터 표현
Transport	L4	TCP, UDP	종단 간 전송, 포트, 신뢰성
Internet	L3	IP, ICMP	라우팅과 패킷 전달
Network Access	L2, L1	Ethernet, Wi-Fi, ARP	링크 전달과 물리 전송

실무에서 `TCP/IP`가 더 자주 쓰이는 이유는 단순합니다.

- 실제 인터넷 프로토콜 스택이 이 구조에 더 가깝습니다
- L5, L6를 따로 떼지 않아도 대부분의 설명이 충분합니다
- 운영체제 네트워크 스택, 라우터, NIC, 실제 장비와 대응이 쉽습니다

즉, 현대 시스템에서는 보통 이렇게 이해하면 충분합니다.

```
Application  : HTTP, DNS, TLS
Transport    : TCP, UDP
Internet     : IP
Network Access: Ethernet, Wi-Fi
```

참고로 교재에 따라 `TCP/IP 5계층`처럼 Physical을 따로 떼어 설명하기도 합니다. 하지만 실무 감각에서는 보통 `4계층` 모델이 더 자주 쓰입니다.

Phase 4. 브라우저에서 서버까지 요청 하나를 따라가 보면

이제 이론 대신 실제 흐름으로 보겠습니다. 사용자가 브라우저에 `https://example.com`을 입력했다고 가정해 보겠습니다.

1. 먼저 DNS로 IP 주소를 찾습니다

브라우저는 도메인 이름만으로는 패킷을 보낼 수 없습니다. 먼저 `example.com`이 어떤 IP인지 알아야 합니다.

이 단계는 보통 애플리케이션 계층에서 다룹니다.

- 브라우저 캐시 확인
- OS 캐시 확인
- 필요하면 DNS resolver에 질의

즉, HTTP 요청을 보내기 전에도 이미 네트워크 통신이 한 번 일어날 수 있습니다.

2. 서버와 TCP 연결을 맺습니다

IP를 알게 되면 클라이언트는 서버의 `443` 포트와 TCP 연결을 시도합니다.

여기서 일어나는 것이 `3-way handshake`입니다.

```
Client → SYN
Server → SYN + ACK
Client → ACK
```

이 단계의 목적은:

- 서로 통신 가능한지 확인하고
- 초기 시퀀스 번호를 맞추고
- 신뢰성 있는 연결을 시작하는 것

입니다.

이때 timeout이 나면 흔히:

- 서버 포트가 안 열려 있거나
- 방화벽이 막고 있거나
- 중간 라우팅이 잘못되었거나
- 패킷 손실이 심한 상황

을 의심하게 됩니다.

3. HTTPS라면 TLS handshake가 이어집니다

TCP 연결이 맺어졌다고 바로 HTTP 본문을 보내는 것은 아닙니다. HTTPS라면 그 위에서 `TLS handshake`가 먼저 일어납니다.

여기서:

- 서버 인증서 확인
- 암호 스위트 협상
- 세션 키 생성

같은 작업이 수행됩니다.

OSI 관점으로 보면 `Presentation`이나 `Session`과 닿아 있는 역할처럼 설명할 수 있지만, 현대 실무에서는 보통 그냥 "`TLS`" 자체로 이해하는 경우가 많습니다.

4. 그 위에 HTTP 요청을 올립니다

이제서야 브라우저는 HTTP 메시지를 씁니다.

```
GET / HTTP/1.1
Host: example.com
User-Agent: ...
Accept: text/html
```

이 메시지는 애플리케이션 계층 데이터입니다.

5. HTTP 데이터는 TCP 세그먼트로 감싸집니다

TCP는 이 데이터를 받아:

- 출발지 포트
- 목적지 포트
- 시퀀스 번호

- ACK 번호

같은 정보를 헤더에 붙입니다.

이제 "HTTP 메시지"는 L4 입장에서는 단순한 payload가 됩니다.

6. TCP 세그먼트는 다시 IP 패킷이 됩니다

IP는 TCP 세그먼트를 받아:

- 출발지 IP
- 목적지 IP
- `TTL` (IPv6에서는 `Hop Limit`)
- 프로토콜 번호

같은 정보를 붙입니다.

이제 TCP 세그먼트는 L3 입장에서는 또 하나의 payload가 됩니다.

7. IP 패킷은 Ethernet 또는 Wi-Fi 프레임이 됩니다

마지막으로 링크 계층은 다음 홉으로 보내기 위해:

- 출발지 MAC
- 목적지 MAC

같은 정보를 붙입니다.

이 단계에서 목적지 MAC은 "최종 서버의 MAC"이 아니라, **현재 링크에서 다음으로 넘길 장치의 MAC**일 수 있습니다. 예를 들어 기본 게이트웨이의

MAC일 수 있습니다.

8. 중간 장비를 거치며 무엇이 바뀔까?

패킷이 라우터를 하나 지날 때마다:

- **L2 헤더는 보통 다시 씌워집니다**
- **L3의 `TTL` (IPv6에서는 `Hop Limit`)은 감소합니다**
- **L3의 출발지/목적지 IP는 대체로 유지됩니다** (`NAT` 같은 중간 장비가 없다는 전제)

즉, 프레임은 구간마다 바뀌고, IP 패킷은 종단까지 이어지는 축에 가깝습니다.

9. 서버는 반대로 역캡슐화합니다

서버는 받은 프레임에서 헤더를 하나씩 벗겨 냅니다.

```
Ethernet/Wi-Fi 프레임 제거
  ↓
IP 헤더 확인
  ↓
TCP 헤더 확인
  ↓
HTTP 메시지 전달
  ↓
웹 서버/애플리케이션 처리
```

이 과정을 **역캡슐화(decapsulation)** 라고 합니다.

Phase 5. 캡슐화만 이해해도 네트워크가 훨씬 덜 헷갈립니다

네트워크 입문에서 가장 중요한 개념 하나만 고르라면, 많은 경우 **캡슐화 (encapsulation)** 를 꼽을 수 있습니다.

각 계층은 자기 헤더를 붙이고, 아래 계층에 데이터를 넘깁니다.

계층	데이터 단위
Application	Data, Message
Transport	Segment (`TCP`), Datagram (`UDP`)
Internet	Packet
Data Link	Frame
Physical	Bit

예를 들어:

```
HTTP 메시지
  ↓
TCP Segment
  ↓
IP Packet
  ↓
Ethernet Frame
  ↓
Bits
```

이 순서로 감싸집니다.

이걸 이해하면 이런 문장이 자연스럽게 해석됩니다.

- "L4에서 재전송이 발생했다"
- "L3 라우팅은 정상인데 L7 응답이 이상하다"
- "L2는 붙는데 L3 게이트웨이로 못 나간다"

즉, 문제를 "패킷이 이상하다" 수준에서 끝내지 않고, **어느 계층의 헤더와 책임에서 문제가 생겼는지**로 더 구체적으로 볼 수 있습니다.

Phase 6. 실무에서는 계층별로 어떻게 장애를 나눠 볼까?

실무에서 계층 모델이 빛나는 순간은 장애 분석입니다. 증상만 보면 비슷해 보여도, 어느 층에서 보느냐에 따라 확인 순서가 달라집니다.

증상	먼저 볼 계층	흔한 원인
도메인으로 접속이 안 됨	L7(Application) 또는 DNS	DNS 레코드 오류, resolver 문제, 캐시 문제
IP는 보이는데 연결 timeout	L4/L3	포트 미오픈, 방화벽, 보안 그룹, 라우팅 문제
인증서 오류 발생	L6에 가까운 보안 표현 영역 또는 L7	인증서 만료, SNI 설정 오류, TLS 설정 불일치
응답은 오지만 404/500/502	L7	애플리케이션 버그, 프록시 설정 문제, 업스트림 장애
같은 서브넷 안 통신만 이상함	L2	VLAN, 스위치, ARP, MAC 학습 문제
케이블 교체 후 간헐적 패킷 손실	L1/L2	물리 매체 불량, NIC, 포트 문제

예를 들어 "서비스가 안 된다"는 말을 들었을 때, 바로 애플리케이션 로그부터 보는 것이 항상 정답은 아닙니다.

아래처럼 좁혀 가는 편이 더 빠를 때가 많습니다.

1. 도메인이 올바른 IP로 해석되는가?
2. 해당 IP의 해당 포트까지 TCP 연결이 맺어지는가?
3. TLS handshake는 정상인가?
4. HTTP 응답 코드는 무엇인가?
5. 그 뒤에야 애플리케이션 로직과 DB를 본다

즉, 네트워크 계층 모델은 **문제를 아래에서 위로, 또는 위에서 아래로 잘라 보는 틀**입니다.

Phase 7. 왜 실무에서는 OSI보다 TCP/IP가 더 자주 쓰일까?

실무에서 `TCP/IP`가 더 많이 등장하는 이유는 현실의 구현과 더 가깝기 때문입니다.

- 인터넷은 실제로 `IP` 위에서 동작합니다
- 전송은 보통 `TCP`나 `UDP`로 설명하면 충분합니다
- `Session`, `Presentation`은 현대 시스템에서 별도 층으로 분리되지 않는 경우가 많습니다

조금 더 엄밀히 말하면, `RFC 1122`는 인터넷 프로토콜 스위트의 application layer가 OSI의 top two layers, 즉 **presentation과 application 기능을 본질적으로 함께 가진다**고 설명합니다. session 성격의 기능은 별도 고정 계층보다는 개별 프로토콜 내부 동작으로 흡수되는 경우가 많습니다.

그래서 개발자나 인프라 엔지니어는 흔히 이렇게 말합니다.

- "이 문제는 `TCP` 연결 문제다"
- "이건 `IP` 라우팅 문제다"
- "이건 `HTTP` 레벨 문제다"

반면 `OSI 7계층`은 여전히 매우 유용합니다. 특히 장비와 역할을 구분할 때 그렇습니다.

`L4`와 `L7`이 왜 자주 나올까?

대표적인 예가 로드밸런서와 프록시입니다.

- **L4 로드밸런서** — 주로 IP와 포트 같은 전송 계층 정보로 트래픽을 분산합니다
- **L7 프록시/로드밸런서** — HTTP 헤더, URL 경로, Host, Cookie 같은 애플리케이션 계층 정보까지 보고 판단합니다

즉, 실무에서는:

- **구현은 TCP/IP 관점으로 이해하고**
- **문제 범위 설명은 OSI 용어를 빌려서 말하는 경우가 많다**

고 정리할 수 있습니다.

핵심만 다시 정리하면

마지막으로 `OSI 7계층`과 `TCP/IP`를 한 번 더 비교하면 이렇게 볼 수 있습니다.

구분	`OSI 7계층`	`TCP/IP 4계층`
성격	통신 과정을 세밀하게 설명하는 참조 모델	실제 인터넷 스택에 더 가까운 실용 모델
장점	역할을 층별로 나눠 이해하기 쉽습니다	실제 프로토콜과 장비 설명에 바로 연결됩니다
자주 쓰는 상황	`L4`, `L7`, 계층별 장애 분리 설명	`HTTP`, `TCP`, `IP`, `Ethernet` 같은 실제 스택 설명

핵심 포인트는 아래 다섯 가지입니다.

1. `OSI 7계층`은 암기용 표가 아니라, 통신 문제를 어느 층에서 봐야 하는지 나누는 기준입니다.
2. `TCP/IP 4계층`은 현대 인터넷 구현과 더 가깝기 때문에 실무 설명에서 더 자주 등장합니다.
3. 웹 요청 하나는 보통 `HTTP/TLS → TCP → IP → Ethernet/Wi-Fi` 순서로 내려가며 캡슐화됩니다.
4. 반대로 서버는 프레임, 패킷, 세그먼트 헤더를 벗기며 역캡슐화한 뒤 최종적으로 애플리케이션에 메시지를 전달합니다.
5. 장애 분석에서는 "네트워크가 안 된다"보다 **DNS 문제인지, `TCP` 연결 문제인지, `TLS` 문제인지, `HTTP` 문제인지**를 나눠 보는 것이 더 중요합니다.

결국 네트워크 계층 모델은 암기 과목이라기보다, **복잡한 통신을 구조적으로 보는 방법**에 가깝습니다.

Chapter 2

IPv4 vs IPv6 완전 정복 — 주소, `NAT`, 헤더, `NDP`까지

#TCP/IP

IPv4와 IPv6가 무엇이 다른지, 왜 주소 체계가 바뀌었는지, `NAT`, 헤더 구조, `ARP`와 `NDP`, 단편화와 주소 설정 관점에서 실무 기준으로 정리합니다.

`IPv4`와 `IPv6`, 왜 따로 알아야 하나요?

네트워크 계층 글, HTTP 버전 글까지 읽고 나면 이런 질문이 자연스럽게 남습니다.

- `IPv4`와 `IPv6`는 주소 길이만 다른 걸까요?
- 왜 아직도 많은 서비스는 `IPv4`를 쓰는데, `IPv6`는 계속 도입하라고 할까요?
- `IPv6`를 쓰면 `NAT`이 사라진다고 하는데, 그게 정확히 무슨 뜻일까요?
- `ARP`, `DHCP`, `broadcast`처럼 익숙한 개념은 `IPv6`에서 어떻게 바뀔까요?

핵심은 이것입니다.

> **`IPv6`는 단순히 주소를 길게 늘린 프로토콜이 아니라, 주소 부족과 확장성 문제를 해결하려고 네트워크 계층 구조를 다시 정리한 후속 프로토콜입니다**

즉:

- `IPv4`는 오랫동안 인터넷의 기본 프로토콜이었고
- 주소 부족, `NAT`, 복잡한 확장성 문제가 쌓였으며
- `IPv6`는 이를 더 큰 주소 공간, 단순한 기본 헤더, 새로운 주소 설정과 이웃 탐색 방식으로 재정리했습니다

이 글에서는 `IPv4`와 `IPv6`를 "32비트 vs 128비트" 비교에서 끝내지 않고, **실무에서 체감되는 차이인 주소 배분, `NAT`, 헤더 처리, `ARP`와 `NDP`, 단편화**를 기준으로 정리하겠습니다.

> **기준:** 이 글은 `IPv4`는 `RFC 791`, `IPv6`는 `RFC 8200`, 주소 구조는 `RFC 4291`, 이웃 탐색은 `RFC 4861`, 자동 주소 설정은 `RFC 4862`를 기준으로 설명합니다.

먼저 가장 큰 차이부터 보면

실무에서는 아래 정도만 먼저 잡아도 큰 틀은 흔들리지 않습니다.

관점	IPv4	IPv6
주소 길이	32-bit	128-bit
주소 부족 대응	사설 주소 + NAT 가 매우 흔함	훨씬 넓은 주소 공간으로 NAT 필요성이 크게 줄어듦
주소 종류	unicast, multicast, broadcast	unicast, multicast, anycast
기본 헤더	가변 길이, 헤더 체크섬 포함	고정 40 bytes, 헤더 체크섬 없음
옵션/확장	IPv4 옵션 필드	extension headers
단편화	중간 라우터도 가능	출발지에서만 Fragment header 사용
이웃 탐색	ARP	NDP (ICMPv6)
주소 설정	수동, DHCP, 사설망 중심	SLAAC, DHCPv6, link-local 주소 기본 사용

가장 짧게 줄이면 이렇습니다.

- **IPv4 는 부족한 주소를 아껴 쓰기 위해 NAT 와 사설망을 적극 활용해 온 체계입니다**
- **IPv6 는 더 큰 주소 공간과 더 단순한 기본 헤더를 바탕으로 구조를 다시 정리한 체계입니다**

- **그래서 실무 차이는 주소 표기보다 `NAT`, 라우팅, 이웃 탐색, 운영 방식에서 더 크게 드러납니다**

Phase 1. 왜 `IPv6`가 필요해졌을까?

가장 먼저 이해해야 할 것은 "`IPv6`가 왜 나왔는가"입니다.

`IPv4`는 주소가 부족해졌다

`IPv4` 주소는 `32-bit`입니다. 이론상 주소 개수는 많아 보이지만, 인터넷이 커지고 서버, PC, 모바일, IoT 장치가 폭증하면서 공인 주소만으로 모든 장치를 직접 식별하기 어려워졌습니다.

그래서 실무에서는 보통 이런 구조가 익숙합니다.

```
인터넷 공인 IPv4 1개
  ↓
공유기 / 방화벽
  ↓
사설 IPv4 대역의 내부 장치 여러 대
```

이때 핵심 역할을 하는 것이 `NAT`입니다.

`NAT`는 문제를 "없앤" 것이 아니라 "우회"한 것이다

`NAT` 덕분에 사설 주소를 내부에서 재사용할 수 있게 되었고, 주소 부족 문제를 오래 버틸 수 있었습니다. 하지만 그 대가도 있었습니다.

- 종단 간 직접 연결이 더 복잡해졌습니다

- 포트 포워딩, 세션 상태 관리, `NAT traversal` 같은 추가 고민이 생겼습니다
- 주소와 포트가 중간 장비에서 바뀌므로 디버깅이 어려워질 수 있습니다

즉, `NAT`는 매우 실용적인 해법이었지만, **인터넷 계층 자체를 단순하게 만든 것은 아니었습니다.**

`IPv6`는 더 넓은 주소 공간을 전제로 다시 설계됐다

`IPv6`는 `128-bit` 주소를 사용합니다. 이 차이는 단순히 "주소가 많아졌다"에서 끝나지 않습니다.

- 주소를 매우 넉넉하게 배분할 수 있고
- 네트워크마다 일관된 주소 계획을 세우기 쉬워지며
- 주소 부족 때문에 반드시 주소 공유용 `NAT`를 써야 하는 압박이 크게 줄어듭니다

즉, `IPv6`의 출발점은 이것입니다.

> **주소 부족을 애플리케이션과 운영 복잡도로 버티지 말고, 네트워크 계층에서 더 크게 풀자**

Phase 2. 주소는 어떻게 달라졌을까?

주소 체계 차이는 가장 눈에 띄지만, 실무에서는 "길이"보다 "의미"를 같이 보는 편이 중요합니다.

1. 표기 방식이 다르다

`IPv4`는 보통 이렇게 씁니다.

```
192.168.0.10
10.0.0.15
203.0.113.20
```

`IPv6`는 16진수 블록을 `:`로 구분합니다.

```
2001:db8:abcd:1::10
fe80::1
```

`::`는 연속된 `0` 구간을 압축한 표기입니다.

2. 주소 종류도 다르다

`IPv4`에서는 `unicast`, `multicast`, `broadcast`를 자주 떠올립니다.

반면 `IPv6`는 `broadcast`가 없습니다. `RFC 4291`도 **`IPv6`에는 `broadcast` 주소가 없고, 그 역할은 `multicast`가 대체한다**고 설명합니다.

즉:

- `IPv4` : `broadcast`가 있음
- `IPv6` : `broadcast` 없음, 대신 `multicast` 중심

또 하나는 `anycast`입니다.

`anycast`는 같은 주소를 여러 인터페이스에 할당하고, 그중 **라우팅 관점에서 가장 가까운 하나**로 보내는 방식입니다. CDN이나 분산 서비스 이야기에서 자주 만나는 개념입니다.

3. `link-local` 주소가 훨씬 중요하다

`IPv6`에서는 `link-local` 주소가 기본적으로 매우 중요합니다. `RFC 4291`도 **모든 인터페이스는 최소 하나의 `link-local unicast` 주소를 가져야 한다**고 설명합니다.

- 같은 링크 안의 이웃과 통신할 때 사용하고
- 라우터 탐색과 이웃 탐색에도 쓰이며
- 인터페이스가 글로벌 주소를 아직 받기 전에도 기본 통신에 참여할 수 있습니다

즉, `IPv6`에서는 "공인 주소를 받기 전에는 아무것도 못 한다"보다, **링크 단위 기본 주소를 먼저 갖고 시작한다**는 감각이 더 중요합니다.

Phase 3. 헤더 구조는 무엇이 달라졌을까?

주소만 바뀐 것이 아니라, 패킷 헤더 설계도 꽤 달라졌습니다.

`IPv4` 헤더는 가변 길이이고 체크섬이 있다

`RFC 791`의 `IPv4` 헤더는:

- 기본 `20 bytes`
- 옵션이 있으면 더 길어질 수 있고

- 헤더 체크섬이 있으며
- 라우터가 `TTL` 감소 등으로 헤더를 바꿀 때 체크섬도 다시 계산해야 합니다

이 구조는 유연하지만, 중간 장비가 처리해야 할 일이 더 있습니다.

`IPv6` 기본 헤더는 더 단순하다

`RFC 8200`의 `IPv6` 기본 헤더는 고정 `40 bytes`입니다.

그리고 실무에서 중요한 차이는 아래입니다.

- 헤더 체크섬이 없습니다
- 옵션성 정보는 `extension headers`로 분리합니다
- `TTL` 대신 `Hop Limit`을 씁니다

핵심은 이것입니다.

> **`IPv6`는 자주 필요한 필드는 기본 헤더에 남기고, 추가 정보는 확장 헤더로 분리해서 기본 처리를 단순하게 만들었습니다**

헤더가 더 단순하다고 무조건 "더 빠르다"는 뜻은 아니다

여기서 조심할 점이 있습니다.

`IPv6` 헤더가 더 정돈되어 있다고 해서, 모든 환경에서 `IPv4`보다 무조건 빠르다고 말할 수는 없습니다.

실제 성능은:

- 장비 구현
- 라우팅 경로
- `ACL`/방화벽 정책
- 애플리케이션 병목

같은 운영 요소에 더 크게 좌우될 수 있습니다.

즉, 헤더 설계 차이는 중요하지만 **항상 체감 속도 차이로 직결된다고 보면 과합니다.**

Phase 4. 단편화는 왜 운영 포인트가 될까?

`IPv4`와 `IPv6`의 중요한 차이 중 하나가 단편화입니다.

`IPv4`는 중간 라우터도 단편화할 수 있다

`IPv4`에서는 패킷이 경로 중간의 MTU보다 크면, 라우터가 패킷을 더 작은 조각으로 나눌 수 있습니다.

이 방식은 유연해 보이지만:

- 중간 장비가 추가 작업을 해야 하고
- 조각 손실 시 재조립 비용이 생기며
- 운영상 예측이 어려워질 수 있습니다

`IPv6`는 출발지에서만 단편화한다

`RFC 8200` 기준으로 `IPv6`에서는 **중간 라우터가 패킷을 단편화하지 않습니다.**

필요하다면:

- 출발지가 `Path MTU`를 고려해서 크기를 줄이거나
- 정말 필요한 경우 출발지가 `Fragment header`를 사용해 단편화합니다

즉, `IPv6`에서는 "중간 라우터가 알아서 잘게 쪼개 주겠지"라고 기대하면 안 됩니다.

그래서 `Path MTU`가 더 중요해진다

실무에서 이 차이는 꽤 크게 느껴집니다.

- 중간 장비가 조용히 잘라 주지 않으므로
- 송신 측이 경로 MTU를 더 의식해야 하고
- 잘못하면 큰 패킷이 기대대로 전달되지 않을 수 있습니다

그래서 `IPv6` 운영에서는 `Path MTU Discovery`와 관련된 방화벽/`ICMPv6` 정책까지 같이 봐야 할 때가 많습니다.

Phase 5. `ARP`와 `DHCP`는 `IPv6`에서 어떻게 바뀔까?

이 부분이 현장에서 가장 실무적으로 헷갈립니다.

`IPv4`에서는 보통 `ARP`와 `DHCP`가 익숙하다

같은 L2 링크 안에서:

- IP 주소에 대응하는 MAC 주소를 찾는 데는 `ARP`
- 주소를 자동으로 받는 데는 `DHCP`

가 익숙합니다.

`IPv6`는 `ARP` 대신 `NDP`를 쓴다

`RFC 4861`의 `Neighbor Discovery`는 `ICMPv6` 기반으로:

- 이웃 존재 확인
- 링크 계층 주소 확인
- 라우터 발견
- 다음 홉 결정
- 경로 도달 가능성 관리

를 함께 처리합니다.

즉, `IPv4`에서 `ARP`가 맡던 일부 역할을 넘어, **`IPv6`의 `NDP`는 라우터 발견과 이웃 상태 관리까지 더 넓게 다룹니다.**

또 `IPv6`는 `broadcast`가 없기 때문에, `ARP`처럼 전체 브로드캐스트에 기대는 구조 대신 `multicast`를 적극 활용합니다. 그래서 `ICMPv6`와 `NDP`를 막연히 "부가 기능"처럼 다루면 실제 통신 자체가 흔들릴 수 있습니다.

주소 설정도 더 유연하다

`RFC 4862`는 `SLAAC`를 정의합니다. 즉, `IPv6` 호스트는 라우터 광고 정보를 바탕으로 **상태 없는 자동 주소 설정**을 할 수 있습니다.

물론 여기서 중요한 점은:

- `IPv6`가 항상 `DHCPv6` 없이만 동작한다는 뜻은 아니고
- 환경에 따라 `SLAAC`, `DHCPv6`, 둘의 조합이 모두 가능하다는 점입니다

즉:

- `IPv4` : `DHCP` 중심이 익숙함
- `IPv6` : `SLAAC`와 `DHCPv6`를 함께 설계 대상으로 봄

으로 이해하는 편이 실무 감각에 가깝습니다.

Phase 6. 실무에서는 무엇을 가장 크게 체감할까?

실무에서 많이 생기는 오해를 먼저 정리하면 이렇습니다.

1. `IPv6`라고 `NAT`이 완전히 금지되는 것은 아니다

`IPv6`의 큰 장점은 주소 공유용 `NAT` 필요성이 크게 줄어든다는 점입니다. 하지만 이것이 "`IPv6`에서는 어떤 형태의 주소 변환도 절대 없다"는 뜻은 아닙니다.

실무에서는:

- 주소 부족 때문에 강제되는 `NAT`의 압박이 줄고
- 더 자연스러운 `end-to-end` 주소 설계가 쉬워지며
- 방화벽 정책과 주소 계획을 더 직접적으로 볼 수 있게 됩니다

정도가 핵심입니다.

2. `IPv6`라고 자동으로 더 안전해지지는 않는다

주소가 많고 구조가 새롭다고 해서 보안이 자동으로 해결되지는 않습니다.

여전히:

- 어떤 주소를 외부에 노출할지
- 인바운드 정책을 어떻게 걸지
- `ICMPv6`를 어디까지 허용할지
- 이웃 탐색과 라우터 광고를 어떻게 통제할지

같은 운영 설계가 중요합니다.

즉, `IPv6`는 **보안 정책을 없애는 기술이 아니라, 다른 전제 위에서 다시 설계해야 하는 기술**에 가깝습니다.

3. 실제 전환은 보통 `dual-stack`으로 간다

많은 서비스가 하루아침에 `IPv4`를 버리고 `IPv6`만 쓰지는 않습니다.

보통은:

- `IPv4`와 `IPv6`를 함께 운영하는 `dual-stack`

- 외부는 `IPv6`, 내부 일부는 아직 `IPv4`
- CDN/LB까지만 `IPv6`, 뒤쪽은 `IPv4`

같은 점진적 구성이 더 현실적입니다.

즉, 실무에서 중요한 질문은 "`IPv4`냐 `IPv6`냐"의 이분법보다:

- 어디까지 `IPv6`를 열 것인가
- 어떤 구간이 아직 `IPv4`에 남아 있는가
- `DNS`, 로드밸런서, 보안 정책이 둘 다를 일관되게 처리하는가

에 더 가깝습니다.

핵심만 다시 정리하면

마지막으로 `IPv4`와 `IPv6`를 한 번 더 비교하면 이렇게 볼 수 있습니다.

구분	IPv4	IPv6
출발점	제한된 주소 공간 안에서 확장	넓은 주소 공간을 전제로 재설계
운영 체감	사설망, `NAT`, `ARP`, `DHCP` 중심	`link-local`, `NDP`, `SLAAC`, 더 직접적인 주소 계획
패킷 처리	가변 헤더, 체크섬, 라우터 단편화 가능	고정 헤더, 체크섬 없음, 출발지 단편화 중심
주소 전달 방식	`broadcast` 사용 가능	`broadcast` 없음, `multicast`와 `anycast` 활용

핵심 포인트는 다섯 가지입니다.

1. `IPv4`와 `IPv6`의 가장 큰 차이는 주소 길이만이 아니라, **주소 부족을 다루는 철학과 운영 구조**에 있습니다.
2. `IPv4`는 주소 부족 때문에 사설망과 `NAT`가 매우 흔해졌고, `IPv6`는 더 넓은 주소 공간으로 그 압박을 크게 줄였습니다.
3. `IPv6`는 `broadcast`를 없애고 `multicast`와 `anycast`를 더 적극적으로 사용하는 구조입니다.
4. `IPv6`는 기본 헤더를 단순화하고, 중간 라우터 단편화를 없애서 `Path MTU`와 송신 측 조절이 더 중요해졌습니다.
5. 실무에서는 `ARP` / `DHCP`만 보지 말고, **`NDP`, `SLAAC`, `dual-stack`, 방화벽 정책까지 함께** 봐야 `IPv6`를 제대로 이해할 수 있습니다.

결국 `IPv4` vs `IPv6`는 "주소 표기가 다르다" 수준의 차이가 아니라, **인터넷 주소 부족을 어떻게 감당할지, 네트워크 계층을 얼마나 단순하게 유지할지에 대한 설계 철학 차이**에 가깝습니다.

Chapter 3

TCP vs UDP 완전 정복 — 연결, 신뢰성, 순서 보장, `QUIC`까지

#TCP/IP

TCP와 UDP가 무엇이 다른지, 왜 하나는 연결형이고 다른 하나는 비연결형인지, 신뢰성, 순서 보장, 혼잡 제어, 메시지 경계, `QUIC` 관점에서 실무 기준으로 정리합니다.

`TCP`와 `UDP`, 왜 따로 알아야 하나요?

네트워크 계층 글, HTTP 버전 글, IPv4 vs IPv6 글까지 읽고 나면 이런 질문이 남습니다.

- 웹은 왜 보통 `TCP` 위에서 동작할까요?
- `UDP`는 연결을 안 맺으니 무조건 더 빠르다고 해도 될까요?
- 게임, 음성 통화, 스트리밍은 왜 `UDP`를 많이 쓸까요?
- `HTTP/3`는 `UDP` 위에서 동작하는데, 그럼 신뢰성이 없는 걸까요?

핵심은 이것입니다.

> **`TCP`와 `UDP`의 가장 큰 차이는 단순히 "빠르냐 느리냐"가 아니라, 애플리케이션에게 어떤 전송 계약을 제공하느냐에 있습니다**

즉:

- `TCP`는 **신뢰성 있는, 순서 보장 바이트 스트림**
- `UDP`는 **최소 기능의, 비연결형 데이터그램 전송**

에 가깝습니다.

이 글에서는 `TCP`와 `UDP`를 "둘 다 포트 번호를 쓰는 전송 계층 프로토콜"에서 끝내지 않고, **연결, 신뢰성, 순서, 흐름 제어, 혼잡 제어, 메시지 경계, `QUIC` 과의 관계**를 기준으로 정리하겠습니다.

> **기준:** 이 글은 `TCP`는 `RFC 9293`, `UDP`는 `RFC 768`, `UDP` 사용 지침은 `RFC 8085`를 기준으로 설명합니다.

먼저 선택 기준부터 보면

실무에서는 보통 아래 기준으로 보면 크게 틀리지 않습니다.

상황	먼저 볼 선택지	이유
정확한 전달과 순서 보장이 중요함	`TCP`	재전송, 순서 보장, 흐름 제어가 기본 제공됩니다
일부 손실보다 지연이 더 중요함	`UDP`	연결 설정과 재전송 대기 없이 빠르게 보낼 수 있습니다
메시지 경계 자체가 중요함	`UDP`	한 번 보낸 데이터그램 경계가 유지됩니다
바이트 스트림으로 길게 주고받음	`TCP`	메시지 경계 대신 연속 스트림 처리에 적합합니다
`QUIC`처럼 신뢰성과 혼잡 제어를 직접 얹고 싶음	`UDP` 위의 상위 프로토콜	필요한 기능을 상위 프로토콜 쪽에서 설계할 수 있습니다

가장 짧게 줄이면 이렇습니다.

- **정확하게 도착해야 하면 `TCP`부터 봅니다**
- **조금 잃어도 실시간성이 더 중요하면 `UDP`를 봅니다**
- **`UDP`가 항상 더 빠른 것은 아니고, 빠져 있는 기능을 누가 대신 구현할지가 더 중요합니다**

Phase 1. 전송 계층은 무엇을 맡을까?

먼저 큰 그림부터 다시 잡아야 합니다.

전송 계층의 핵심 역할은:

- 어떤 프로세스와 통신할지 포트로 구분하고
- 애플리케이션 데이터를 네트워크에 실어 보내며
- 상대에게 어떤 형태로 전달할지 계약을 정하는 것

입니다.

즉, 같은 IP 통신이라도:

- `TCP`를 쓰면 애플리케이션은 **연결된 스트림**처럼 느끼고
- `UDP`를 쓰면 애플리케이션은 **독립된 메시지 묶음**처럼 느낍니다

이 차이가 상위 프로토콜 설계에 매우 크게 영향을 줍니다.

Phase 2. `TCP`는 무엇을 보장할까?

`RFC 9293`는 `TCP`를 **`reliable, in-order, byte-stream service`**로 설명합니다. 이 한 문장에 핵심이 거의 다 들어 있습니다.

1. `TCP`는 연결형이다

`TCP`는 데이터를 주고받기 전에 연결을 맺습니다. 흔히 말하는 `3-way handshake`가 여기서 나옵니다.

아주 단순하게 그리면:

```
Client → Server : SYN
Client ← Server : SYN + ACK
Client → Server : ACK
```

이 과정을 거쳐 양쪽은:

- 서로 통신할 준비가 되었는지 확인하고
- 초기 시퀀스 번호를 맞추고
- 연결 상태를 만듭니다

즉, `TCP`는 "그냥 보내면 된다"가 아니라 **연결 상태를 만든 뒤 그 위에서 데이터를 흐르게 하는 프로토콜**입니다.

2. `TCP`는 신뢰성과 순서를 보장한다

`TCP`가 중요한 이유는 단순히 연결을 맺기 때문이 아닙니다.

- 시퀀스 번호로 순서를 관리하고
- `ACK`로 도착 여부를 확인하며
- 손실이 나면 재전송하고
- 중복 세그먼트를 걸러냅니다

즉, 애플리케이션은 보통:

- 일부 패킷이 중간에서 빠졌는지
- 순서가 뒤바뀌었는지
- 다시 보내야 하는지

를 직접 다루지 않아도 됩니다.

이 때문에:

- 파일 전송

- 데이터베이스 연결
- 로그인/결제 같은 API 요청

처럼 **정확도가 먼저인 작업**과 잘 맞습니다.

3. 하지만 `TCP`는 메시지 프로토콜이 아니라 스트림이다

이 부분이 가장 자주 헷갈립니다.

> **`TCP`는 메시지 경계를 보장하지 않고, 연속된 바이트 스트림을 전달합니다**

예를 들어 애플리케이션이 두 번 `send()` 했다고 해서, 상대가 반드시 두 번 `recv()`로 정확히 같은 단위로 받는 것은 아닙니다.

```
보낸 쪽:
  "HELLO"
  "WORLD"

받는 쪽:
  "HELLOWORLD"
```

또는:

```
받는 쪽:
  "HEL"
  "LOW"
  "ORLD"
```

처럼 보일 수 있습니다.

즉, `TCP` 위에서는 애플리케이션이:

- 길이 프리픽스
- 구분자
- 자체 프레이밍 규칙

으로 메시지 경계를 따로 정의해야 합니다.

4. `TCP`는 흐름 제어와 혼잡 제어도 함께 본다

`TCP`는 단순히 "다시 보내 주는 프로토콜"이 아닙니다.

- **흐름 제어**로 상대 수신 버퍼가 감당할 수 있는 양을 넘기지 않으려 하고
- **혼잡 제어**로 네트워크가 감당하지 못할 만큼 무작정 밀어 넣지 않으려 합니다

즉, `TCP`는 단일 애플리케이션의 편의만이 아니라 **네트워크 전체의 안정성**도 같이 고려하는 전송 계층 프로토콜입니다.

Phase 3. `UDP`는 무엇을 제공할까?

`RFC 768`은 `UDP`를 `minimum of protocol mechanism`을 제공하는 프로토콜이라고 설명합니다. 그리고 `delivery`와 `duplicate protection`이 보장되지 않는다고 명시합니다.

1. `UDP`는 비연결형이다

`UDP`는 `TCP`처럼 연결을 맺는 절차가 없습니다.

즉, 개념적으로는:

```
메시지 하나를 바로 보냄
```

에 가깝습니다.

그래서:

- 연결 설정 지연이 없고
- 상태가 단순하며
- 짧은 요청/응답이나 실시간 트래픽에 유리할 수 있습니다

2. `UDP`는 메시지 경계를 유지한다

`UDP`의 중요한 장점 중 하나는 **데이터그램 단위가 유지된다는 점**입니다.

예를 들어 100바이트를 한 번 보내면, 받는 쪽은:

- 그 데이터그램 전체를 한 묶음으로 받거나
- 너무 큰 버퍼 부족 등으로 일부만 보더라도
- 기본 개념 자체는 "하나의 메시지"입니다

즉, `UDP`는 `TCP`와 달리 **메시지 지향적**입니다.

이 차이는 다음 같은 경우에 특히 중요합니다.

- 실시간 게임 입력 이벤트
- DNS 질의/응답
- 음성 프레임
- 센서/텔레메트리 샘플

3. 대신 순서, 재전송, 중복 제거는 기본 제공이 아니다

`UDP`는 애플리케이션에 이런 보장을 기본 제공하지 않습니다.

- 안 도착할 수 있습니다
- 순서가 바뀔 수 있습니다
- 중복 도착할 수 있습니다

즉, 애플리케이션이 필요하다면:

- 시퀀스 번호
- 타임아웃
- 재전송
- 중복 제거

를 직접 구현해야 합니다.

그래서 `UDP`를 쓴다는 말은 종종:

> **전송 계층이 안 해 주는 일을 상위 프로토콜이나 애플리케이션이 대신 떠안는다**

는 뜻이기도 합니다.

4. `UDP`에는 혼잡 제어가 내장되어 있지 않다

이 부분도 매우 중요합니다.

`RFC 8085`는 `UDP`가 **`no inherent congestion control mechanisms`**를 가진다고 설명합니다.

즉, 마음만 먹으면 애플리케이션이 매우 빠르게 `UDP` 데이터그램을 밀어 넣을 수 있습니다. 하지만 그렇게 하면:

- 경로 용량을 초과할 수 있고
- 혼잡 붕괴에 기여할 수 있으며
- 다른 트래픽과의 공정성도 해칠 수 있습니다

그래서 인터넷에서 `UDP`를 쓰는 애플리케이션은 **혼잡 제어를 따로 고려해야 합니다.**

Phase 4. 왜 `UDP`가 항상 더 빠르지는 않을까?

실무에서 가장 흔한 오해가 이것입니다.

> **`UDP`는 단순하니까 무조건 `TCP`보다 빠르다**

이건 절반만 맞습니다.

`UDP`가 유리할 수 있는 이유

- 연결 설정이 없습니다
- 재전송 대기가 기본 동작에 없습니다
- 헤더가 단순합니다
- 메시지 단위 전송이 쉽습니다

그래서 짧고 빠른 단방향 전송이나 실시간성 높은 트래픽에서는 유리할 수 있습니다.

하지만 손실이 생기면 이야기가 달라진다

패킷 손실이 있는 환경에서 애플리케이션이 결국:

- 재전송을 직접 구현하고
- 순서를 다시 맞추고
- 혼잡 제어도 추가하고
- 연결 생명주기도 추적한다면

처음의 "단순함"은 빠르게 사라집니다.

즉, `UDP` 자체가 빠르다기보다:

- **버릴 수 있는 것을 버릴 수 있을 때**
- 또는 **필요한 기능을 더 잘게 직접 설계할 수 있을 때**

강점이 생깁니다.

반대로 정확히 도착해야 하는 데이터를 `UDP`로 보내면서 위 기능을 다시 다 구현한다면, `TCP`를 쓰는 것보다 복잡하기만 할 수 있습니다.

Phase 5. 그럼 언제 `TCP`, 언제 `UDP`를 쓸까?

실무에서는 보통 아래처럼 판단하면 크게 틀리지 않습니다.

`TCP`가 잘 맞는 경우

- 웹 API 요청/응답
- 로그인, 결제, 주문 생성
- 데이터베이스 연결
- 파일 업로드/다운로드
- 메일, SSH, 일반적인 백엔드 간 통신

이런 경우는 대체로:

- 순서가 중요하고
- 일부 손실을 허용하기 어렵고
- 애플리케이션이 전송 신뢰성을 다시 구현할 이유가 적습니다

`UDP`가 잘 맞는 경우

- 실시간 음성/영상
- 온라인 게임 상태 업데이트
- DNS 같은 짧은 질의/응답
- 일부 텔레메트리, 모니터링 이벤트

이런 경우는 대체로:

- 늦게 도착한 데이터가 쓸모없어질 수 있고
- 일부 손실을 허용할 수 있으며
- 지연을 더 민감하게 봅니다

다만 여기서도 "무조건 `UDP`"라고 말하면 과합니다. 예를 들어 오늘날 스트리밍과 실시간 통신은 `UDP` 위의 상위 프로토콜 설계, 적응형 비트레이트, 혼잡 제어까지 함께 봐야 하기 때문입니다.

Phase 6. `QUIC`은 왜 `UDP` 위에서 동작할까?

HTTP/3 글과 연결되는 지점입니다.

많은 분이 여기서 이렇게 생각합니다.

- `UDP`는 신뢰성이 없는데
- 왜 `HTTP/3`는 굳이 `UDP` 위에 올라갈까?

핵심은 이것입니다.

> **`QUIC`은 `UDP` 위에 올라가지만, `UDP`의 빈 기능을 그대로 방치하지 않고 그 위에 신뢰성, 순서, 혼잡 제어, `TLS 1.3` 통합을 다시 설계한 프로토콜입니다**

즉, `QUIC`은:

- `UDP`의 데이터그램 전송을 바탕으로
- 필요한 재전송과 혼잡 제어를 직접 구현하고
- 스트림 단위 다중화도 제공하며
- `TCP` 위 `HTTP/2`에서 남던 전송 계층 병목 일부를 다르게 풀려는 접근입니다

그래서 "`UDP` = 신뢰성 없음"이라는 문장을 그대로 `HTTP/3`에 적용하면 틀립니다. 정확히는:

- **`UDP` 자체는 신뢰성과 혼잡 제어를 보장하지 않고**
- **`QUIC`이 그 위에 필요한 기능을 다시 얹습니다**

핵심만 다시 정리하면

마지막으로 `TCP`와 `UDP`를 한 번 더 비교하면 이렇게 볼 수 있습니다.

구분	TCP	UDP
전송 성격	연결형 바이트 스트림	비연결형 데이터그램
순서/재전송	기본 제공	기본 제공 안 함
메시지 경계	유지 안 됨	유지됨
흐름/혼잡 제어	기본 제공	애플리케이션이 고려해야 함
대표 사용처	웹, DB, 파일 전송	DNS, 게임, 음성, `QUIC`

핵심 포인트는 다섯 가지입니다.

1. `TCP`와 `UDP`의 가장 큰 차이는 속도 그 자체보다 **전송 계약의 차이**에 있습니다.
2. `TCP`는 **신뢰성 있는, 순서 보장 바이트 스트림**이고, 메시지 경계는 애플리케이션이 직접 만들어야 합니다.
3. `UDP`는 **메시지 지향 데이터그램 전송**이지만, 전달 보장, 중복 제거, 혼잡 제어를 기본 제공하지 않습니다.
4. `UDP`는 단순해서 유리할 수 있지만, 빠진 기능을 다시 구현해야 하면 반드시 더 좋은 선택이 되는 것은 아닙니다.
5. `HTTP/3`의 `QUIC`처럼, 현대 프로토콜은 `UDP` 위에 필요한 신뢰성과 혼잡 제어를 다시 설계해 사용하는 경우도 많습니다.

결국 `TCP` vs `UDP`는 "`UDP`가 더 빠른가?"의 문제가 아니라, **우리 애플리케이션이 순서와 신뢰성을 어디까지 필요로 하고, 그 비용을 전송 계층이 맡을지 애플리케이션이 맡을지**를 고르는 문제에 가깝습니다.

Chapter 4

TCP `4-way handshaking` 완전 정복 — `FIN`, `ACK`, `TIME_WAIT`까지

#TCP/IP

TCP `4-way handshaking`이 무엇인지, 왜 연결 종료에 `FIN`과 `ACK`가 네 번 오가는지, `half-close`, `TIME_WAIT`, `CLOSE_WAIT`, `RST`와의 차이까지 실무 기준으로 정리합니다.

`4-way handshaking`, 왜 따로 알아야 하나요?

TCP vs UDP 글까지 읽고 나면 이런 질문이 남습니다.

- `3-way handshake`는 많이 들어봤는데, `4-way handshaking`은 정확히 무엇일까요?
- 왜 연결을 닫는 데는 `SYN`, `ACK`처럼 세 번이 아니라 네 번이 필요할까요?
- 서버에 `CLOSE_WAIT`가 많이 쌓였다는 말은 무슨 뜻일까요?
- `TIME_WAIT`는 왜 마지막에 꼭 남고, 언제 문제처럼 보일까요?
- `FIN`으로 닫는 것과 `RST`로 끊는 것은 무엇이 다를까요?

핵심은 이것입니다.

> **`TCP` 연결 종료는 "한 번에 끊는 동작"이 아니라, 양방향 스트림을 각각 독립적으로 닫아 가는 과정입니다**

즉:

- `TCP`는 양방향 통신이고
- 내가 보내는 방향과 상대가 보내는 방향은 따로 닫을 수 있으며
- 그래서 연결 종료도 보통 `FIN`과 `ACK`가 나뉘어 오가게 됩니다

이 글에서는 `4-way handshaking`을 단순 암기 절차로 보지 않고, **왜 `FIN`과 `ACK`가 네 번 오가는지, `half-close`, `TIME_WAIT`, `CLOSE_WAIT`, `RST`와 어떤 관계가 있는지**를 기준으로 정리하겠습니다.

> **기준:** 이 글은 `TCP` 연결 종료 동작을 `RFC 9293` 기준으로 설명합니다. 표현상 `4-way handshaking`이라고 부르지만, RFC는 보통 **normal close sequence using a `FIN` handshake**라고 설명합니다.

먼저 가장 짧은 답부터 보면

실무에서는 아래 정도만 먼저 잡아도 큰 틀은 흔들리지 않습니다.

질문	짧은 답
`4-way handshaking`이란?	`TCP` 연결을 정상 종료할 때 `FIN` / `ACK`를 교환하는 대표 흐름
왜 네 번인가?	양방향 스트림을 각각 닫아야 해서 `ACK`와 반대편 `FIN`이 분리될 수 있기 때문
누가 `TIME_WAIT`에 들어가나?	보통 **active close**를 시작한 쪽
`CLOSE_WAIT`는 왜 쌓이나?	원격 `FIN`은 받았지만 로컬 애플리케이션이 자기 쪽 `close`를 안 했기 때문
`RST`와 차이는?	`FIN`은 정상 종료, `RST`는 연결을 즉시 중단하는 abort에 가깝습니다

가장 짧게 줄이면 이렇습니다.

- **`4-way handshaking`은 `TCP`의 정상 종료 절차입니다**
- **핵심은 "양쪽이 각자 보내던 방향을 따로 닫는다"는 점입니다**
- **그래서 `TIME_WAIT`, `CLOSE_WAIT` 같은 상태도 종료 과정의 일부로 이해해야 합니다**

Phase 1. 왜 연결 종료는 시작보다 더 복잡해 보일까?

`3-way handshake`는 연결을 시작하는 절차입니다. 이때는 양쪽이 "이제 연결을 만들자"는 데 동의하면 됩니다.

반면 종료는 조금 다릅니다.

> **연결 종료는 "앞으로 더 보낼 데이터가 없다"는 사실을 각 방향마다 따로 합의해야 합니다**

즉, `TCP`에서는:

- 내가 보내는 스트림
- 상대가 보내는 스트림

이 독립적으로 닫힐 수 있습니다.

그래서 한쪽이 먼저:

```
나는 더 이상 보낼 데이터가 없습니다
```

라고 말해도, 상대는 아직:

```
나는 보낼 데이터가 남아 있습니다
```

라고 할 수 있습니다.

이 특성 때문에 종료는 보통 한 번에 끝나지 않고, **한 방향 종료 확인 + 반대 방향 종료 확인** 으로 나뉩니다.

Phase 2. `FIN`과 `ACK`는 각각 무엇을 뜻할까?

`4-way handshaking`을 이해하려면 `FIN`과 `ACK`를 정확히 나눠 봐야 합니다.

`FIN`은 "더 보낼 데이터가 없다"는 뜻이다

`FIN`은 대충 "연결 끊자"가 아닙니다. 더 정확히는:

> **이 방향으로는 더 이상 보낼 사용자 데이터가 없습니다**

라는 뜻입니다.

그래서 한쪽이 `FIN`을 보냈다고 해서, 연결 전체가 즉시 사라지는 것은 아닙니다.

상대는 여전히:

- 지금까지 받은 데이터는 읽어야 하고
- 자기가 남은 데이터를 보낼 수도 있습니다

`ACK`는 "네 `FIN`은 받았다"는 뜻이다

상대가 `ACK`를 보내면, 이는 보통:

```
네가 더 이상 안 보낸다는 뜻은 알겠다
```

는 의미입니다.

하지만 이것만으로 상대도 자기 송신을 끝냈다는 뜻은 아닙니다.

즉:

- `FIN` = 나의 송신 종료 선언
- `ACK` = 네 종료 선언 수신 확인

으로 보는 편이 정확합니다.

Phase 3. `4-way handshaking`은 실제로 어떻게 흐를까?

가장 흔한 정상 종료 흐름은 아래처럼 그릴 수 있습니다.

```
Client → Server : FIN
Client ← Server : ACK
Client ← Server : FIN
Client → Server : ACK
```

이걸 단계별로 풀면 이렇습니다.

1. 먼저 닫고 싶은 쪽이 `FIN`을 보낸다

예를 들어 클라이언트가 먼저 요청/응답을 다 마쳤고 더 보낼 데이터가 없다고 가정해 보겠습니다.

그러면:

```
Client → Server : FIN
```

을 보냅니다.

이 순간 클라이언트는 보통 `FIN-WAIT-1` 같은 상태로 들어가며, **자기 송신 방향을 닫기 시작한 상태**가 됩니다.

2. 상대는 `ACK`로 받았다고 답한다

서버는 이 `FIN`을 받으면:

```
Client ← Server : ACK
```

를 보냅니다.

이 뜻은:

- 클라이언트의 송신 종료 선언은 받았고
- 하지만 서버 자신은 아직 보낼 것이 남아 있을 수 있다는 뜻

입니다.

이 시점이 바로 `half-close`를 이해하는 핵심입니다.

3. 상대가 자기 송신도 끝낼 때 `FIN`을 보낸다

서버 애플리케이션도 응답을 다 마쳤거나 더 보낼 데이터가 없어지면:

```
Client ← Server : FIN
```

을 보냅니다.

즉, 서버도 이제 자기 방향 송신을 닫습니다.

4. 마지막으로 `ACK`를 보내고 종료를 확정한다

클라이언트는 마지막으로:

```
Client → Server : ACK
```

를 보내고, 정상 종료 시퀀스가 마무리됩니다.

다만 여기서 끝난 것처럼 보여도, active close를 시작한 쪽은 보통 바로 사라지지 않고 `TIME_WAIT`에 들어갑니다.

Phase 4. 왜 정확히 네 번이어야 할까?

여기서 자주 나오는 질문이 이것입니다.

> **왜 `3-way handshake`는 세 번인데 종료는 네 번일까?**

핵심은 시작과 종료의 대칭성이 완전히 같지 않기 때문입니다.

연결 시작에서는:

- 서로 연결을 만들 준비가 되었는지
- 초기 시퀀스 상태를 맞추는지

를 빠르게 합의하면 됩니다.

반면 연결 종료에서는:

- 내 송신 종료
- 네 송신 종료

가 분리될 수 있습니다.

그래서 상대는 내 `FIN`을 받자마자:

- `ACK`로 먼저 응답하고
- 자기 애플리케이션이 실제로 닫을 준비가 되었을 때 나중에 `FIN`을 보낼 수 있습니다

즉, **`ACK`와 반대편 `FIN`이 논리적으로 분리될 수 있어서 네 단계처럼 보이는 것**입니다.

꼭 항상 패킷 네 개가 보이느냐?

엄밀히 말하면, 캡처에서는 상황에 따라 `ACK`와 `FIN`이 같은 세그먼트에 함께 실려 보일 수도 있습니다.

예를 들어 RFC 9293의 normal close 예시에서도 `FIN,ACK` 형태가 등장합니다.

즉:

- 흔히 설명할 때는 `FIN → ACK → FIN → ACK` 네 단계로 이해하고
- 실제 패킷 단위에서는 `ACK`가 다른 제어 비트와 합쳐질 수 있으며
- 동시에 닫으면 `CLOSING` 같은 다른 상태 흐름도 나올 수 있습니다

그래서 `4-way handshaking`은 **개념적 종료 절차**로 이해하는 편이 정확합니다.

Phase 5. `half-close`는 정확히 무엇일까?

`TCP` 종료를 이해할 때 중요한 개념이 `half-close`입니다.

한쪽이 `FIN`을 보내면, 그쪽은:

- 더 이상 보내지는 않지만
- 상대가 보내는 데이터는 아직 받을 수 있습니다

즉, 연결 전체가 죽은 것이 아니라 **한 방향만 먼저 닫힌 상태**가 됩니다.

예를 들어:

1. 클라이언트가 요청 바디를 다 보냄
2. `FIN` 전송
3. 서버는 그 요청을 다 읽고 처리
4. 응답을 보낸 뒤 자기 `FIN` 전송

같은 흐름이 가능합니다.

즉, `half-close`는 이상한 예외가 아니라, **`TCP` 종료가 양방향 독립이라는 사실의 자연스러운 결과**입니다.

Phase 6. `TIME_WAIT`는 왜 남을까?

실무에서 가장 많이 보이는 종료 관련 상태는 `TIME_WAIT`입니다.

`TIME_WAIT`는 active close 쪽에서 주로 보인다

RFC 9293 기준으로 normal close sequence에서 먼저 닫기를 시작한 쪽은 마지막 `ACK`를 보낸 뒤 `TIME_WAIT`에 들어갑니다.

이 상태의 목적은 크게 두 가지입니다.

1. **마지막 `ACK`가 상대에게 도달했는지 여유를 둔다**
2. **이전 연결의 지연 세그먼트가 새 연결과 섞이지 않게 한다**

RFC 9293도 `TIME-WAIT`를, 원격 피어가 종료 요청에 대한 `ACK`를 받았는지 충분한 시간을 기다리고 지연 세그먼트의 영향을 피하기 위한 상태로 설명합니다.

그래서 왜 2MSL 이야기가 나올까?

보통 `TIME_WAIT`는 `2MSL`과 함께 설명됩니다.

- `MSL`은 최대 세그먼트 생존 시간 개념이고
- `2MSL`은 패킷 왕복 지연과 지연 세그먼트 정리를 고려한 대기 시간입니다

즉, `TIME_WAIT`는 "쓸데없이 남는 상태"가 아니라, **정상 종료를 안전하게 마무리하기 위한 완충 구간**입니다.

언제 문제처럼 보일까?

실무에서는:

- 짧은 연결이 매우 많고
- active close를 애플리케이션이 계속 하고
- 포트 재사용 압박이 있는 환경

에서 `TIME_WAIT`가 많아 보일 수 있습니다.

하지만 상태가 많다는 사실 자체가 곧 버그는 아닙니다. 오히려 **정상 종료가 잘 되고 있다는 흔적**일 수도 있습니다.

Phase 7. `CLOSE_WAIT`는 왜 쌓일까?

이건 `TIME_WAIT`보다 더 자주 진짜 문제입니다.

`CLOSE_WAIT`는 원격 `FIN`은 받았는데, 로컬이 아직 안 닫았다는 뜻이다

상대가 `FIN`을 보내면 로컬 커널은 이를 `ACK`하고, 애플리케이션에게 연결이 닫히는 중이라고 알립니다.

이후 애플리케이션이 자기 쪽 `close`를 호출해야:

- `FIN`을 보내고

- `LAST-ACK`를 거쳐
- 완전히 종료됩니다

그런데 애플리케이션이 이 종료를 미루거나 누락하면 `CLOSE_WAIT`가 오래 남습니다.

즉:

> **`CLOSE_WAIT`가 많이 쌓인다는 것은 대개 네트워크 문제보다 애플리케이션이 소켓을 제대로 닫지 않는 문제에 가깝습니다**

이 때문에 서버 운영에서 `CLOSE_WAIT`는 진짜 점검 대상이 되는 경우가 많습니다.

Phase 8. `FIN` 종료와 `RST` 종료는 어떻게 다를까?

RFC 9293는 `TCP` 연결 종료를 크게 두 가지로 설명합니다.

1. 정상적인 `FIN` 기반 종료
2. `RST`를 보내고 상태를 즉시 버리는 abort

`FIN`은 정상 종료다

`FIN` 기반 종료는:

- 남아 있는 데이터 순서를 보존하고
- 상대에게 종료 사실을 예고하며

- 정상적으로 정리할 기회를 줍니다

즉, graceful close에 가깝습니다.

`RST`는 즉시 중단에 가깝다

반면 `RST`는 보통:

- 연결이 비정상 상태이거나
- 더 이상 정상 종료 절차를 밟지 않고
- 즉시 연결을 끊어야 할 때

사용됩니다.

즉, `RST`는:

- "정상적으로 다 보냈으니 닫겠습니다"가 아니라
- **"이 연결은 더 이상 유효하지 않으니 즉시 버리겠습니다"**

에 가깝습니다.

그래서 애플리케이션 관점에서도 `FIN` 종료와 `RST` 종료는 같은 일이 아닙니다.

핵심만 다시 정리하면

마지막으로 `4-way handshaking`을 한 번 더 정리하면 이렇게 볼 수 있습니다.

구분	의미
첫 번째 `FIN`	한쪽이 자기 송신 종료를 선언
첫 번째 `ACK`	상대가 그 종료 선언을 확인
두 번째 `FIN`	상대도 자기 송신 종료를 선언
마지막 `ACK`	종료 절차를 최종 확인
`TIME_WAIT`	active close 쪽이 마지막 정리를 위해 대기
`CLOSE_WAIT`	원격은 닫았지만 로컬 애플리케이션이 아직 안 닫음

핵심 포인트는 다섯 가지입니다.

1. `4-way handshaking`은 `TCP` 연결을 **정상 종료**할 때 가장 대표적으로 나타나는 절차입니다.
2. 종료가 네 단계처럼 보이는 이유는 **양방향 스트림을 각각 독립적으로 닫기 때문**입니다.
3. 한쪽이 `FIN`을 보냈다고 해서 연결 전체가 즉시 끝나는 것은 아니며, 이 사이에 `half-close`가 가능합니다.
4. `TIME_WAIT`는 정상 종료를 안전하게 마무리하기 위한 상태이고, `CLOSE_WAIT`는 대개 애플리케이션의 소켓 정리 누락과 더 관련이 큽니다.
5. `FIN` 종료와 `RST` 종료는 다르며, `RST`는 정상적인 마무리보다 **즉시 abort**에 가깝습니다.

결국 `4-way handshaking`은 "`FIN`, `ACK` 네 개를 외우는 절차"가 아니라, **`TCP`가 양방향 스트림을 어떻게 안전하게 접고, 종료 후 지연 세그먼트까지 어떻게 정리하는지**를 보여 주는 상태 전이 과정에 가깝습니다.

PART 2

2부. HTTP와 보안

Chapter 5

HTTP/1.1 vs HTTP/2 vs HTTP/3 완전 정복 — `keep-alive`, `multiplexing`, `QUIC`까지

#TCP/IP

HTTP/1.1, HTTP/2, HTTP/3가 무엇이 다른지, 왜 같은 API라도 체감 성능이 달라지는지, `keep-alive`와 `multiplexing`, `QUIC`와 `HOL blocking` 관점에서 실무 기준으로 정리합니다.

HTTP 버전, 왜 따로 알아야 하나요?

네트워크 계층 글까지 읽고 나면 이런 질문이 자연스럽게 남습니다.

- 어차피 API는 `GET /users` 처럼 똑같은데, 왜 `HTTP/1.1`, `HTTP/2`, `HTTP/3` 를 구분해서 봐야 할까요?
- `HTTP/2` 는 멀티플렉싱이라서 무조건 빠르다고 해도 될까요?
- `HTTP/3` 는 `UDP` 위에서 동작한다는데, 그러면 신뢰성이 약해지는 걸까요?
- 로드밸런서나 CDN이 `HTTP/3` 를 지원한다는 말은 정확히 어디까지를 뜻할까요?
- 브라우저에서 체감이 없는데도 서버는 굳이 `HTTP/3` 를 켜야 할까요?

핵심은 이것입니다.

> **HTTP 버전 차이는 "API 의미"보다 "연결을 맺고, 데이터를 잘게 나누고, 손실을 복구하는 방식"의 차이에서 성능과 안정성을 바꿉니다**

즉:

- `HTTP` 의 **`semantics`** 는 대체로 유지되지만
- 그 메시지를 **어떻게 전송하느냐** 가 달라지고
- 그 차이가 지연 시간, 병렬 처리, 패킷 손실 대응, 모바일 네트워크 체감에 영향을 줍니다

이 글에서는 `HTTP/1.1`, `HTTP/2`, `HTTP/3`를 암기식 표가 아니라, **브라우저가 서버와 연결을 재사용하고 여러 요청을 동시에 보내는 방식이 어떻게 바뀌었는지** 를 기준으로 정리하겠습니다.

> **기준:** 이 글은 일반적인 브라우저-웹 서버-프록시/CDN 환경을 기준으로 설명합니다. `gRPC`, CDN `edge`, 모바일 네트워크, TLS 종료 지점처럼 실무에서 자주 부딪히는 포인트도 함께 다룹니다.

먼저 선택 기준부터 보면

실무에서는 보통 아래 기준으로 보면 크게 틀리지 않습니다.

상황	먼저 눈여겨볼 버전	이유
레거시 프록시, 단순 API, 내부망 중심	`HTTP/1.1`	호환성이 가장 넓고 운영 이해도가 높습니다
일반적인 브라우저 웹 서비스	`HTTP/2`	다중 요청 병렬 처리와 헤더 압축 이점이 큽니다
모바일 네트워크, 패킷 손실, CDN `edge` 최적화	`HTTP/3`	TCP 레벨 `HOL blocking` 영향을 줄이고 재연결 체감이 더 좋을 수 있습니다
외부 공개 서비스에서 "기본 최적화"	`HTTP/2` + 가능하면 `HTTP/3`	대부분 환경에서 무난하고, 하위 호환도 자연스럽습니다
비멱등 `POST` 재시도 정책이 민감함	`HTTP/3` 의 `0-RTT` 는 신중히	성능 이점과 `replay` 위험을 함께 봐야 합니다

가장 짧게 줄이면 이렇습니다.

- **`HTTP/1.1` 은 단순하고 익숙하지만 병렬 전송에 약합니다**
- **`HTTP/2` 는 대부분의 웹 서비스에서 가장 먼저 체감되는 개선을 줍니다**
- **`HTTP/3` 는 특히 손실과 네트워크 전환이 잦은 환경에서 더 빛납니다**

Phase 1. 먼저 분리해서 봐야 할 것 - HTTP의 의미와 전송 방식

가장 먼저 정리해야 할 오해가 있습니다.

> **HTTP 버전이 바뀐다고 해서 `GET`, `POST`, 상태 코드, 헤더라는 개념이 통째로 새로 생기는 것은 아닙니다**

예를 들어 아래 요청의 의미 자체는 크게 달라지지 않습니다.

```
GET /posts/25 HTTP/1.1
Host: example.com
Accept: text/html
```

`HTTP/2`, `HTTP/3`에서도 여전히:

- `GET`, `POST`, `PUT`, `DELETE`가 있고
- `200`, `404`, `500` 같은 상태 코드가 있고
- `Host`, `Content-Type`, `Cache-Control` 같은 헤더가 있고
- 캐시, 쿠키, 인증 같은 상위 개념도 계속 유지됩니다

달라지는 핵심은 주로 아래입니다.

관점	`HTTP/1.1`	`HTTP/2`	`HTTP/3`
메시지 표현	텍스트 기반	바이너리 프레이밍	바이너리 프레이밍
전송 계층	`TCP`	`TCP`	`QUIC` over `UDP`
다중 요청 처리	연결 여러 개에 의존	한 연결에서 다중 스트림	한 연결에서 다중 스트림
손실 전파	연결 단위로 큼	TCP 레벨 영향 남음	스트림 단위로 더 잘 분리
연결 설정	TCP + TLS	TCP + TLS	QUIC + TLS 1.3 통합

즉, "HTTP의 의미"보다 **"연결을 얼마나 효율적으로 쓰는가"** 가 버전 차이의 핵심입니다.

Phase 2. `HTTP/1.1` 은 왜 느리다고 느껴질까?

`HTTP/1.1` 자체가 무조건 느린 것은 아닙니다. 다만 현대 웹 페이지처럼 작은 리소스를 많이 한꺼번에 가져오는 상황에서 구조적 한계가 잘 드러납니다.

1. 초창기에는 요청마다 연결 비용이 반복됐다

브라우저가 HTML, CSS, JS, 이미지, 폰트를 가져와야 한다고 가정해 보겠습니다.

연결 재사용이 없다면 요청마다 이런 비용이 반복됩니다.

```
DNS 조회
  ↓
TCP `handshake`
  ↓
TLS `handshake`
  ↓
HTTP 요청/응답
```

HTTPS에서는 특히 이 연결 설정 비용이 큽니다. MDN의 TTFB 설명도 브라우저가 첫 바이트를 받기 전 시간에 `DNS lookup`, `TCP handshake`, `TLS handshake`가 포함된다고 설명합니다.

2. 그래서 `keep-alive`가 중요해졌다

`HTTP/1.1`에서 가장 먼저 체감되는 개선은 **`persistent connection`**, 흔히 말하는 `keep-alive`입니다.

한 번 맺은 TCP 연결을 재사용하면:

- 같은 `origin`에 대한 후속 요청에서 `handshake` 비용을 줄일 수 있고
- 리소스 여러 개를 한 연결에서 순차적으로 요청할 수 있으며
- 특히 이미지, CSS, JS처럼 자잘한 요청이 많은 페이지에서 효과가 큽니다

하지만 여기에도 한계가 있습니다.

3. 한 연결 안에서는 결국 순서 대기가 생긴다

`HTTP/1.1`은 기본적으로 한 연결에서 요청/응답이 직렬적으로 흘러가기 쉽습니다.

예를 들어 첫 응답이 늦어지면 뒤 요청도 줄줄이 대기하게 됩니다.

```
Connection A
  Request 1 → 느린 응답
  Request 2 → 대기
  Request 3 → 대기
```

이 문제를 줄이려고 브라우저는 보통 같은 `origin`에 TCP 연결을 여러 개 열었습니다.

즉:

- 연결 1개는 직렬 처리 한계가 있고
- 이를 보완하려고 연결 수를 늘렸고
- 그 결과 서버 입장에서는 소켓 수, TLS 세션 수, 커널 자원 부담이 커졌습니다

4. `pipelining`은 왜 널리 안 쓰였을까?

이론적으로는 응답을 기다리지 않고 요청을 연달아 보내는 `pipelining`이 있었지만, 중간 프록시 호환성 문제와 응답 순서 제약 때문에 실무에서 널리 쓰이지 않았습니다.

결국 `HTTP/1.1`의 핵심 한계는 이것입니다.

> **연결 재사용은 되지만, "한 연결에서 여러 요청을 자연스럽게 동시에 흘리는 능력"은 약합니다**

Phase 3. `HTTP/2`는 무엇을 바꿨을까?

`HTTP/2`가 가져온 가장 큰 변화는 **한 TCP 연결 안에서 여러 요청/응답을 스트림 단위로 잘게 나눠 동시에 흘릴 수 있게 만든 것**입니다.

1. 텍스트 대신 바이너리 프레임으로 나눴다

`HTTP/1.1`에서는 요청 메시지가 텍스트 줄 단위로 보였다면, `HTTP/2`에서는 내부적으로 프레임이라는 단위로 잘게 쪼개집니다.

이렇게 되면 브라우저는 하나의 연결 안에서:

- 요청 A 일부
- 요청 B 일부
- 요청 C 일부

를 번갈아 보낼 수 있습니다.

즉, 더 이상 "응답 하나 끝나야 다음 요청" 구조에 묶일 필요가 줄어듭니다.

2. `multiplexing`이 핵심 이점이다

예를 들어 브라우저가 CSS, JS, 이미지 여러 개를 동시에 가져올 때:

```
HTTP/1.1:
  연결 여러 개를 열어서 병렬성 확보

HTTP/2:
  연결 하나에서 여러 스트림을 동시에 흘림
```

이 변화로 얻는 이점은 분명합니다.

- 연결 수를 덜 늘려도 됩니다
- 같은 `origin` 내 다중 리소스 로딩이 더 효율적입니다
- TLS 세션/소켓 관리 부담이 상대적으로 줄어듭니다
- 헤더를 압축(`HPACK`)해 중복 헤더 비용도 줄일 수 있습니다

특히 쿠키나 공통 헤더가 큰 서비스에서 헤더 압축은 생각보다 유의미할 수 있습니다.

3. 그런데 왜 `HTTP/2` 도 완전한 해답은 아닐까?

여기서 자주 생기는 오해가 있습니다.

> **`HTTP/2` 는 애플리케이션 레벨 `HOL blocking` 을 줄였지만, TCP 위에 있는 한 전송 계층 수준의 병목은 여전히 남아 있습니다**

하나의 TCP 연결 위에서 여러 스트림을 흘리더라도, 어떤 TCP 세그먼트가 유실되면 재전송과 순서 보장은 TCP가 처리합니다.

그러면 그 손실의 영향이 연결 전체에 퍼질 수 있습니다.

즉:

- `HTTP/2`는 요청 간 인터리빙은 잘하지만
- 패킷 손실이 있는 네트워크에서는 TCP 특성상 전체 스트림 체감이 흔들릴 수 있습니다

MDN도 `HTTP/2`가 `HTTP/1.x`의 HOL blocking을 해결했지만, TCP 레벨의 병목은 남아 있고, `HTTP/3`가 이를 `QUIC`으로 보완한다고 설명합니다.

4. Server Push는 왜 주인공이 아니게 됐을까?

`HTTP/2`를 이야기할 때 `server push`가 함께 나오곤 합니다. 다만 실전에서는:

- 브라우저 캐시 상태를 서버가 정확히 예측하기 어렵고
- 잘못 push하면 오히려 대역폭만 낭비하며
- 운영 복잡도 대비 이득이 제한적인 경우가 많았습니다

그래서 지금 `HTTP/2`의 핵심 가치는 보통:

- **`multiplexing`**
- **`header compression`**
- **연결 수 감소**

이 세 가지에 더 가깝습니다.

Phase 4. `HTTP/3`는 왜 `UDP` 위로 올라갔을까?

여기서부터가 가장 많이 헷갈리는 지점입니다.

`HTTP/3`는 단순히 "`HTTP/2`를 더 빠르게 튜닝한 버전"이 아닙니다. **전송 기반 자체를 `TCP`에서 `QUIC`으로 바꿨습니다.**

1. `HTTP/3`는 `QUIC` 위에서 동작한다

구조를 단순하게 그리면 이렇습니다.

```
HTTP/1.1 → HTTP over TCP
HTTP/2   → HTTP over TCP
HTTP/3   → HTTP over QUIC over UDP
```

여기서 오해하면 안 되는 점은:

- `UDP`를 쓴다고 해서 무조건 비신뢰 전송으로 내려가는 것은 아니고
- `QUIC`이 그 위에서 순서, 손실 감지, 재전송, 흐름 제어, 스트림 개념을 직접 제공합니다

즉, `HTTP/3`는 "TCP를 버리고 무책임해진 HTTP"가 아니라, **TCP가 맡던 기능 일부를 더 현대적인 형태로 사용자 공간 프로토콜로 옮긴 구조**에 가깝습니다.

2. 왜 성능상 이점이 생길까?

핵심은 두 가지입니다.

첫째, 스트림 간 손실 전파를 더 잘 분리한다

`HTTP/2`는 논리적으로는 다중 스트림이지만, 실제 전송은 하나의 TCP 연결에 묶여 있습니다.

반면 `HTTP/3`의 `QUIC`은 스트림 단위 전송을 더 직접적으로 다룹니다. 어떤 패킷 손실이 발생해도 그 영향이 다른 스트림 전체를 동일하게 막지 않도록 설계되어 있습니다.

그래서:

- 패킷 손실이 있는 네트워크
- 지하철, 와이파이 전환, 셀룰러 이동
- 모바일 환경처럼 RTT와 손실률이 불안정한 상황

에서 차이가 더 잘 드러날 수 있습니다.

둘째, 연결 설정 지연을 줄일 수 있다

`QUIC`은 `TLS 1.3`을 통합적으로 사용합니다. 그래서 신규 연결이나 재연결에서 지연을 줄이기 유리합니다.

특히 세션 재개가 가능한 상황에서는 `QUIC`과 `TLS 1.3` 조합에서 `0-RTT`가 가능할 수 있습니다. 다만 이건 성능만 보고 켜면 안 됩니다.

왜냐하면:

- 재전송과 `replay` 관점에서 주의가 필요하고
- 비멱등 요청은 같은 요청 재수행 위험을 더 엄격하게 봐야 하며
- 이 부분은 멱등성 글과 직접 연결됩니다

즉, `HTTP/3`는 `QUIC` 기반 연결 재개 이점이 장점이지만, **모든 `POST`가 자동으로 안전해지는 것은 아닙니다.**

3. `HTTP/3`는 어떻게 발견될까?

브라우저는 보통 서버가 `Alt-Svc`로 "`이 origin은 h3도 가능하다`"는 신호를 광고하면, 이후 해당 `origin`에 대해 `HTTP/3` 연결을 시도합니다.

즉, 실무에서 `HTTP/3 켰다`는 말은 보통:

- 브라우저가 항상 첫 요청부터 바로 `HTTP/3`로 시작한다는 뜻이라기보다
- 먼저 다른 버전의 응답에서 `Alt-Svc`를 보거나 이미 학습한 대체 서비스 정보를 바탕으로
- 이후 동일 `origin`에 대해 `h3`를 시도할 수 있게 만든다는 의미에 가깝습니다

여기서 `ALPN`도 같이 중요합니다. `HTTP/2`의 `h2`, `HTTP/3`의 `h3` 같은 식별자가 협상에 쓰이기 때문입니다.

Phase 5. 그래서 체감 성능은 언제 달라질까?

많은 경우 팀이 원하는 답은 이것입니다.

> **"그래서 우리 서비스에서 진짜 빨라지나요?"**

정답은 "환경에 따라 다르다"이지만, 실무에서는 아래처럼 보면 됩니다.

`HTTP/2` 체감이 큰 경우

- 한 페이지에서 CSS, JS, 이미지, 폰트, API 요청이 많이 발생함
- 같은 `origin`으로 작은 요청이 많이 몰림
- 기존에 연결 수가 많아 소켓 관리와 `handshake` 비용이 부담이었음
- 브라우저 중심 서비스라서 다중 리소스 로딩이 중요함

`HTTP/3` 체감이 큰 경우

- 모바일 사용자가 많음
- 네트워크 전환이 잦음
- 패킷 손실이나 `jitter`가 있는 환경에서 서비스 품질이 흔들림
- CDN `edge`를 적극 사용하고 글로벌 사용자 비중이 큼

체감이 크지 않을 수 있는 경우

- 대부분 요청이 길고 무거운 서버 처리 시간에 묻힘
- 정적 파일보다 API 한두 개가 지연 시간을 지배함
- 내부망 트래픽이라 손실률이 낮고 RTT도 짧음
- 애초에 병목이 DB나 애플리케이션 로직에 있음

즉, 프로토콜 업그레이드는 중요하지만, **DB 쿼리가 800ms인데 HTTP 버전만 바꿔서 20ms 줄이는 일**이 항상 우선순위는 아닙니다.

Phase 6. 실무에서는 어디까지 지원하면 될까?

운영 관점에서는 보통 "`브라우저 -> CDN/LB → origin`" 경로를 나눠서 봐야 합니다.

1. `edge` 까지만 `HTTP/3` 여도 체감은 좋아질 수 있다

예를 들어:

```
브라우저 -> CDN: HTTP/3
CDN → origin: HTTP/1.1 또는 HTTP/2
```

여기서도 사용자 체감은 좋아질 수 있습니다. 특히 마지막 마일, 모바일 무선 구간, 국제망 구간에서는 `edge` 까지의 프로토콜 차이가 더 크게 보일 수 있기 때문입니다.

즉, `origin` 서버가 직접 `HTTP/3` 를 처리하지 않아도:

- CDN이 `edge` 에서 `HTTP/3` 를 받고
- 내부적으로는 `origin` 과 다른 프로토콜로 통신하는 구조가
- 충분히 실용적일 수 있습니다

2. 로드밸런서와 프록시의 실제 지원 범위를 봐야 한다

"우리 서비스는 `HTTP/3` 지원합니다"라고 말하기 전에 확인할 것은 보통 이 정도입니다.

1. 브라우저와 맞닿는 CDN/LB가 `h3`를 광고하는가?
2. TLS 종료 지점이 어디인가?
3. `Alt-Svc`, `ALPN` 협상이 실제로 보이는가?
4. 프록시 체인 중간에서 다운그레이드되는가?
5. 로그와 모니터링에서 실제 프로토콜 버전을 확인할 수 있는가?

겉으로 `HTTP/3 enabled`라고 체크돼 있어도, 실제 트래픽 대부분이 `HTTP/2`로 끝나는 경우는 꽤 흔합니다.

3. gRPC나 양방향 통신은 별도로 봐야 한다

`gRPC`는 기본적으로 `HTTP/2`에 강하게 기대는 경우가 많습니다. 따라서 서비스 전반이 `HTTP/3`로 간다고 해도:

- 브라우저 정적 자산 배포
- 일반 REST/HTML 요청
- 내부 서비스 간 `gRPC`

는 동일한 전략으로 보지 않는 편이 낫습니다.

즉, 프로토콜 전략도 **트래픽 성격별로 분리해서** 보는 것이 맞습니다.

Phase 7. 장애 분석은 어떻게 달라질까?

네트워크 계층 글과 연결해서 보면, HTTP 버전 차이는 장애 분석 순서에도 영향을 줍니다.

`HTTP/1.1` / `HTTP/2` 에서 자주 보는 질문

- 연결 재사용이 제대로 되고 있는가?
- keep-alive timeout이 너무 짧지 않은가?
- 프록시가 연결을 자주 끊지 않는가?
- `HTTP/2` 인데도 연결 수가 비정상적으로 많지 않은가?
- 패킷 손실 구간에서 TCP 재전송 때문에 전체 응답이 흔들리지 않는가?

`HTTP/3` 에서 추가로 보는 질문

- 브라우저가 실제로 `h3` 를 사용했는가?
- `Alt-Svc` 캐시와 만료가 어떻게 동작하는가?
- UDP 차단 또는 중간 장비 정책 때문에 `fallback` 이 발생하는가?
- 일부 지역/망에서만 `HTTP/3` `handshake` 실패가 나는가?

즉, "`HTTP/3 켰는데 별 차이 없다`"는 말은 아래 셋 중 하나일 때가 많습니다.

1. 실제로는 `HTTP/2` 로 `fallback` 중이다
2. 병목이 애플리케이션/DB 쪽이다
3. 사용자 네트워크 특성상 프로토콜 차이가 크게 드러나지 않는다

핵심만 다시 정리하면

마지막으로 세 버전을 한 번 더 비교하면 이렇게 정리할 수 있습니다.

구분	HTTP/1.1	HTTP/2	HTTP/3
강점	단순함, 넓은 호환성	multiplexing, header compression	손실 환경 대응, 더 나은 연결 설정
약점	병렬 처리 비효율, 연결 수 증가	TCP 레벨 HOL 영향 잔존	운영 가시성, 장비/정책 지원 확인 필요
잘 맞는 곳	단순 서비스, 레거시 호환	대부분의 현대 웹 서비스	모바일/글로벌/손실 민감 환경

핵심 포인트는 다섯 가지입니다.

1. `HTTP/1.1`, `HTTP/2`, `HTTP/3`의 가장 큰 차이는 API 의미보다 **전송 방식과 연결 관리 방식**에 있습니다.
2. `HTTP/2`의 핵심 가치는 **`multiplexing`과 `header compression`**이며, 일반적인 웹 서비스에서 가장 먼저 체감되는 개선을 줍니다.
3. `HTTP/3`는 `QUIC`을 통해 **TCP 레벨 HOL blocking 영향을 줄이고**, 손실이 있는 환경에서 더 안정적인 체감을 줄 수 있습니다.
4. `QUIC`과 `TLS 1.3` 기반의 `0-RTT`는 성능 장점이 있지만, **비멱등 요청에서는 `replay` 위험을 함께 고려해야 합니다.**

5. 실무에서는 `origin` 만 볼 것이 아니라, **브라우저-프록시/CDN-`edge`-`origin` 전체 경로에서 실제 어느 버전이 쓰이는지** 확인해야 합니다.

결국 HTTP 버전 선택은 "최신 버전이 무조건 정답인가?"의 문제가 아니라, **우리 트래픽의 병목이 `handshake` 인지, 병렬 요청인지, 패킷 손실인지, 운영 호환성인지** 를 구분하는 문제에 가깝습니다.

Chapter 6

HTTPS 요청 과정 완전 정복 — `DNS`, `TCP`, `TLS handshake`, 인증서 검증까지

#TCP/IP

브라우저에서 `https://` 주소를 입력하면 어떤 일이 일어나는지, `DNS`, `TCP handshake`, `TLS 1.3 handshake`, 인증서 검증, `ALPN`, 실제 HTTP 요청 흐름까지 실무 기준으로 정리합니다.

`HTTPS` 과정, 왜 따로 알아야 하나요?

네트워크 계층 글, HTTP 버전 글, TCP vs UDP 글, TCP `4-way handshaking` 글까지 읽고 나면 이런 질문이 남습니다.

- 브라우저 주소창에 `https://example.com`을 입력하면 실제로 어떤 순서로 일이 진행될까요?
- `HTTPS`는 그냥 "암호화된 HTTP"라고만 보면 충분할까요?
- 인증서는 정확히 언제 확인하고, 틀리면 어디서 막힐까요?
- `HTTP/2`, `HTTP/3` 이야기를 하다 보면 `ALPN`, `SNI`, `TLS handshake`가 같이 나오는데 각각 무슨 역할일까요?

핵심은 이것입니다.

> **`HTTPS`는 HTTP 메시지 하나만의 문제가 아니라, 이름 해석, 연결 수립, `TLS` 협상, 인증서 검증, 애플리케이션 프로토콜 선택이 연속해서 이어지는 과정입니다**

즉:

- 먼저 목적지 이름을 IP로 찾고
- 전송 계층 연결을 만들고
- 그 위에서 `TLS`로 서버를 인증하고 키를 합의한 뒤
- 비로소 HTTP 요청/응답이 안전하게 오갑니다

이 글에서는 `HTTPS`를 "자물쇠 아이콘이 뜬다" 수준에서 끝내지 않고, **브라우저에서 `https://` 요청 하나가 `DNS → TCP → TLS → HTTP` 순서로 어떻게 흐르는지**를 기준으로 정리하겠습니다.

> **기준:** 이 글은 일반적인 `HTTPS` 요청, 즉 **`HTTP/1.1` 또는 `HTTP/2` over `TLS` over `TCP`** 흐름을 기준으로 설명합니다. `https` URI와 기본 흐름은 `RFC 9110`, `TLS 1.3`은 `RFC 8446`, `ALPN`은 `RFC 7301`, `SNI`는 `RFC 6066`, 브라우저 관점의 `TLS` 개요와 `TTFB`는 MDN 문서를 기준으로 설명합니다. `HTTP/3`는 마지막에 별도 차이점으로 짚습니다.

먼저 가장 짧은 흐름부터 보면

브라우저에서 `https://example.com`을 여는 과정을 아주 짧게 줄이면 이렇습니다.

단계	핵심 질문	하는 일
1	어디로 갈까?	`DNS`로 도메인을 IP 주소로 해석
2	어떻게 연결할까?	대상 IP의 `443` 포트로 `TCP handshake` 수행
3	누구와 말하는지 믿을 수 있을까?	`TLS handshake`로 서버 인증과 키 합의
4	HTTP 버전은 무엇을 쓸까?	`ALPN`으로 `http/1.1` 또는 `h2` 같은 프로토콜 선택
5	이제 실제로 무엇을 요청할까?	암호화된 채널 위에서 HTTP 요청/응답 교환

가장 짧게 줄이면 이렇습니다.

- **`HTTPS`는 `HTTP over TLS over TCP`입니다**
- **HTTP 요청은 `TLS handshake`가 끝난 뒤에야 안전하게 보내집니다**
- **인증서 검증에 실패하면 보통 HTTP 요청 자체로 넘어가지 못합니다**

Phase 1. 브라우저는 먼저 무엇을 확인할까?

브라우저가 `https://example.com`을 열 때 가장 먼저 하는 일은 "이 서버와 바로 HTTP부터 얘기하자"가 아닙니다.

먼저 아래 같은 정보를 정리합니다.

- 스킴이 `https`인지
- 호스트가 무엇인지
- 포트가 명시됐는지

`RFC 9110`은 `https` URI의 기본 포트가 `443`이라고 설명합니다. 즉:

```
https://example.com
```

이라면 기본적으로:

- 호스트: `example.com`
- 포트: `443`

으로 봅니다.

`HTTPS`는 보통 바로 HTTP부터 보내지 않는다

`RFC 9110`은 `https` URI에 접근할 때 클라이언트가 보통:

1. 호스트 이름을 IP로 해석하고

2. 그 주소의 `TCP` 연결을 만들고
3. `TLS`를 성공적으로 시작한 뒤
4. 그 위로 HTTP 요청을 보낸다고 설명합니다

즉, `HTTPS`의 핵심은:

> **HTTP 요청보다 먼저 연결 수립과 보안 협상이 선행된다는 점**

입니다.

> **참고:** 실제 브라우저는 캐시된 `DNS`, 기존 연결 재사용, `HSTS`, 프리커넥트 같은 최적화 때문에 이 흐름을 단축할 수 있습니다. 이 글은 기본 원리를 설명하기 위해 가장 대표적인 흐름으로 단순화합니다.

Phase 2. 먼저 `DNS`로 IP 주소를 찾는다

브라우저는 우선 `example.com`의 IP 주소를 알아야 합니다.

왜 `DNS`가 먼저일까?

네트워크는 결국 IP 주소 기준으로 목적지를 찾아갑니다. 그래서 도메인 이름만으로는 아직 연결을 열 수 없습니다.

즉:

```
example.com
  ↓
DNS 조회
  ↓
203.0.113.10 또는 2001:db8::10
```

처럼 **이름을 주소로 바꾸는 과정**이 먼저 필요합니다.

`TTFB`에도 이 시간이 들어간다

MDN의 `TTFB` 설명도, `HTTPS` 요청의 초기 지연에는:

- `DNS lookup`
- `TCP handshake`
- `TLS handshake`

가 포함될 수 있다고 설명합니다.

즉, 사용자가 "페이지가 느리다"고 느낄 때 원인은:

- 애플리케이션 처리
- DB 조회

만이 아니라, **아직 HTTP 요청을 보내기 전 단계**에 있을 수도 있습니다.

Phase 3. `TCP handshake`로 연결을 만든다

대상 IP를 알게 되면, 브라우저는 보통 그 서버의 `443` 포트로 `TCP` 연결을 만듭니다.

왜 `TCP`가 먼저일까?

이 글의 기본 흐름은 `HTTP/1.1` 또는 `HTTP/2` 기준이므로, `TLS`는 보통 `TCP` 위에서 동작합니다.

따라서 순서는:

```
DNS
  ↓
TCP handshake
  ↓
TLS handshake
  ↓
HTTP request
```

가 됩니다.

여기서는 아직 암호화되지 않는다

이 시점의 `TCP handshake`는:

- 서로 연결 가능한지 확인하고
- 시퀀스 번호를 맞추고

- 전송 경로를 여는 절차

에 가깝습니다.

즉, **보안은 아직 `TLS` 단계 전**입니다.

Phase 4. `TLS handshake`에서는 무엇이 오갈까?

이제부터가 `HTTPS`의 핵심입니다.

`TCP` 연결이 만들어지면, 그 위에서 `TLS handshake`가 시작됩니다.

`TLS`가 제공하는 것은 무엇일까?

MDN은 `TLS`가 연결을 세 가지 측면에서 보호한다고 설명합니다.

- **기밀성**: 중간에서 내용을 읽기 어렵게 함
- **무결성**: 중간에서 몰래 바꾸기 어렵게 함
- **인증**: 적어도 서버가 주장하는 주체인지 확인할 수 있게 함

즉, `HTTPS`는 단순히 "암호화"만이 아니라 **서버 인증과 무결성 보호까지 포함**합니다.

`TLS 1.3` 기준으로 아주 단순화하면

실제 메시지는 더 복잡하지만, 대표 흐름을 줄이면 대체로 이렇게 볼 수 있습니다.

```
Client → Server : ClientHello
Client ← Server : ServerHello
Client ← Server : EncryptedExtensions
Client ← Server : Certificate
Client ← Server : CertificateVerify
Client ← Server : Finished
Client → Server : Finished
```

이걸 역할별로 보면:

- 클라이언트는 지원 가능한 암호 스위트, 키 교환 정보 등을 제안하고
- 서버는 그중 사용할 매개변수를 고르고
- 서버는 `EncryptedExtensions`로 `ALPN` 결과 같은 추가 협상 정보를 전달하며
- 서버 인증서를 보내며
- 자신이 그 인증서의 개인키를 실제로 가지고 있음을 증명하고
- 양쪽이 이후 사용할 키로 보호된 `Finished`를 교환합니다

> **참고:** 실제 `TLS 1.3`에는 `CertificateRequest`처럼 상황에 따라 추가되는 메시지도 있고, `PSK`나 `0-RTT`처럼 다른 흐름도 있습니다. 여기서는 일반적인 브라우저의 서버 인증 기반 `HTTPS` 흐름만 단순화해 설명합니다.

여기서 `SNI`와 `ALPN`도 같이 등장한다

`ClientHello`에는 실무적으로 중요한 정보가 더 들어갑니다.

`SNI`

`RFC 6066`의 `server_name` 확장은, 클라이언트가 **어느 호스트 이름으로 접속하려는지** 서버에 알려 주기 위한 장치입니다.

이게 왜 중요할까요?

- 같은 IP 주소 하나에 여러 도메인이 올라가 있을 수 있고
- 서버는 어떤 인증서를 보여줘야 할지 알아야 하기 때문입니다

즉, `SNI`는:

> **"나는 지금 `example.com`에 접속하려는 중입니다"**

를 `TLS handshake` 초반에 알려 주는 역할에 가깝습니다.

`ALPN`

`RFC 7301`의 `ALPN`은 **이 `TLS` 연결 위에서 어떤 애플리케이션 프로토콜을 쓸지** 협상하는 확장입니다.

예를 들어 클라이언트는:

- `http/1.1`
- `h2`

같은 후보를 제안할 수 있고, 서버는 그중 하나를 고릅니다.

즉, 브라우저는 `TLS handshake` 안에서:

- 서버가 어떤 인증서를 제시해야 하는지(`SNI`)
- `HTTP/1.1` 을 쓸지 `HTTP/2` 를 쓸지(`ALPN`)

까지 같이 정할 수 있습니다.

Phase 5. 인증서는 정확히 언제, 어떻게 검증할까?

많은 분이 `HTTPS` 를 "자물쇠 아이콘이 있으면 끝"처럼 느끼지만, 실제로는 이 단계가 매우 중요합니다.

브라우저는 인증서를 그냥 보기만 하지 않는다

서버가 인증서를 보냈다고 해서 무조건 믿는 것은 아닙니다.

브라우저는 보통 아래를 확인합니다.

1. 이 인증서 체인이 신뢰할 수 있는 루트까지 이어지는가
2. 현재 시간 기준으로 유효 기간이 맞는가
3. 접속하려는 호스트 이름과 인증서 이름이 맞는가
4. 서명이 올바른가

즉, `Certificate` 를 받는 순간 중요한 것은:

> **"인증서가 왔다"가 아니라 "이 인증서를 브라우저가 신뢰해도 되는가"**

입니다.

여기서 틀리면 어떻게 될까?

이 검증이 실패하면 브라우저는 보통:

- 경고 페이지를 띄우거나
- 연결을 중단하고
- 실제 HTTP 요청 전송으로 넘어가지 않습니다

즉, `HTTPS` 오류는 많은 경우 **애플리케이션이 응답을 못 줘서가 아니라, `TLS handshake` 단계에서 신뢰를 만들지 못해서** 발생합니다.

Phase 6. `TLS handshake`가 끝나면 그다음은 무엇일까?

`TLS handshake`가 성공하면 이제 양쪽은:

- 서버를 인증했고
- 암호화 키를 합의했고
- 이후 데이터를 보호할 준비가 끝났습니다

그다음에야 비로소 HTTP 요청이 오갑니다.

이때부터 HTTP는 암호화된 채널 위에서 흐른다

즉, 브라우저는 이제:

```
GET / HTTP/1.1
Host: example.com
```

같은 요청을 보내지만, 이 내용이 네트워크 위에서 평문 HTTP처럼 그대로 노출되는 것은 아닙니다.

실제로는 **`TLS`가 보호하는 기록 계층 위로 실려서** 전달됩니다.

응답도 같은 방식으로 보호된다

서버 응답도 마찬가지입니다.

- 상태 코드
- 헤더
- 본문

모두 `TLS`로 보호된 채널 위에서 오갑니다.

즉, `HTTPS`는 "요청만 암호화"가 아니라 **요청과 응답 전체를 보호하는 연결**입니다.

Phase 7. 연결은 요청 하나마다 새로 만들까?

여기서도 자주 오해가 생깁니다.

항상 매 요청마다 처음부터 다시 하지는 않는다

실무에서는 다음이 매우 중요합니다.

- 기존 `TCP` 연결 재사용
- 기존 `TLS` 세션 재개

- `HTTP/2`의 연결 재사용과 다중화

즉, 첫 요청은:

```
DNS → TCP → TLS → HTTP
```

가 길게 보이지만, 이후 요청은:

- 이미 열린 연결을 재사용하거나
- `HTTP/2` 하나의 연결 안에서 여러 요청을 보내거나
- 세션 재개로 `TLS` 비용을 줄일 수 있습니다

그래서 실제 체감 성능은 **첫 연결 비용과 이후 재사용 전략**에 크게 좌우됩니다.

Phase 8. 그럼 `HTTP/3`에서는 무엇이 달라질까?

HTTP 버전 글과 연결되는 부분입니다.

이 글의 기본 흐름은 `HTTP over TLS over TCP`입니다. 하지만 `HTTP/3`는 예외가 있습니다.

`HTTP/3`는 `TCP` 대신 `QUIC` 위에서 동작한다

즉:

- `HTTP/1.1`, `HTTP/2`: 보통 `TCP → TLS → HTTP`

- `HTTP/3` : `QUIC` 위에서 동작하고, `TLS 1.3` handshake가 그 안에 통합됩니다

그래서 `HTTP/3`에서는 이 글에서 설명한 "먼저 `TCP handshake`, 그다음 `TLS handshake`" 흐름이 그대로 적용되지는 않습니다.

하지만 핵심은 같습니다.

- 상대를 인증하고
- 암호화 키를 합의하고
- 그 위에서 HTTP를 보낸다

즉, **보안 협상과 애플리케이션 프로토콜 협상이 먼저 온다**는 큰 원리는 유지됩니다.

핵심만 다시 정리하면

마지막으로 `HTTPS` 과정을 한 번 더 정리하면 이렇게 볼 수 있습니다.

단계	핵심 포인트
`DNS`	도메인을 IP 주소로 바꿔야 실제 연결을 열 수 있습니다
`TCP handshake`	`HTTP/1.1` / `HTTP/2` 기준으로 전송 경로를 먼저 엽니다
`TLS handshake`	서버 인증, 키 합의, 보호 채널 생성이 여기서 일어납니다
인증서 검증	신뢰 체인, 유효 기간, 호스트 이름 일치를 확인합니다
`ALPN`	`http/1.1`과 `h2` 같은 상위 프로토콜을 고릅니다
HTTP 요청/응답	모든 HTTP 메시지는 보호된 채널 위에서 오갑니다

핵심 포인트는 다섯 가지입니다.

1. `HTTPS`는 단순히 "암호화된 HTTP"가 아니라, **`DNS`, `TCP`, `TLS`, HTTP가 이어지는 전체 연결 과정**입니다.
2. `https` URI에 접근할 때는 보통 **이름 해석 -> `TCP` 연결 -> `TLS` 성공 -> HTTP 요청** 순서로 진행됩니다.
3. `TLS handshake`에서는 서버 인증, 키 합의, `SNI`, `ALPN` 같은 중요한 협상이 함께 일어납니다.
4. 인증서 검증에 실패하면 보통 **HTTP 요청 자체로 넘어가지 못합니다.**
5. `HTTP/3`는 전송 기반이 다르지만, **보안 협상과 프로토콜 협상이 먼저 온다**는 큰 원리는 같습니다.

결국 `HTTPS` 과정은 "자물쇠 아이콘이 뜬다" 수준의 이야기가 아니라, **브라우저가 누구와 연결하는지 확인하고, 어떤 프로토콜로 말할지 합의한 뒤, 그 위에서 HTTP를 안전하게 흘리게 만드는 단계적 절차**에 가깝습니다.

Chapter 7

DNS 완전 정복 — 재귀 질의, `TTL`, `A`/`AAAA`/`CNAME`, 권한 DNS까지

#TCP/IP

DNS가 무엇인지, 도메인 이름이 어떻게 IP 주소로 바뀌는지, 재귀 질의와 반복 질의, Root/TLD/권한 DNS, `TTL`, 주요 레코드 타입까지 실무 기준으로 정리합니다.

`DNS`, 왜 따로 알아야 하나요?

HTTPS 요청 과정 글까지 읽고 나면 이런 질문이 자연스럽게 남습니다.

- 브라우저가 `https://example.com`에 접속하기 전에 `DNS`에서는 정확히 무슨 일이 일어날까요?
- `DNS`는 그냥 "도메인을 IP로 바꿔 주는 시스템"이라고만 이해해도 충분할까요?
- `TTL`은 왜 바꿨는데도 적용이 늦어지는 원인이 될까요?
- `A`, `AAAA`, `CNAME`, `MX`, `NS`, `TXT` 레코드는 각각 언제 쓰일까요?
- 장애가 났을 때 `DNS` 문제인지, 네트워크나 `TLS` 문제인지 어떻게 구분해야 할까요?

핵심은 이것입니다.

> **`DNS`는 단순한 주소록이 아니라, 계층적이고 분산된 이름 시스템 위에서 도메인 이름을 IP 주소와 각종 자원 정보로 해석하는 인프라입니다**

즉:

- 사람은 `example.com` 같은 이름을 기억하고
- 네트워크는 결국 IP 주소로 목적지를 찾아가며
- `DNS`는 그 사이에서 이름을 주소와 자원 정보로 바꾸는 역할을 합니다

이 글에서는 `DNS`를 "도메인 -> IP 변환"에서 끝내지 않고, **재귀 질의, 반복 질의, Root/TLD/권한 DNS, 캐시와 `TTL`, 주요 레코드 타입, `UDP`와 `TCP` 사용 지점**을 기준으로 정리하겠습니다.

> **기준:** 이 글은 DNS 개념과 계층 구조는 `RFC 1034`, 프로토콜과 레코드 형식은 `RFC 1035`, 일부 동작과 `TTL` 해석의 보충은 `RFC 2181`, 고수준 개요는 MDN DNS 문서를 기준으로 설명합니다.

먼저 가장 짧은 답부터 보면

브라우저가 `example.com`에 접속하기 전에 일어나는 DNS 흐름을 아주 짧게 줄이면 이렇습니다.

단계	핵심 질문	하는 일
1	이 이름은 이미 알고 있나?	브라우저/OS/리졸버 캐시 확인
2	모르면 누구에게 물을까?	로컬 또는 ISP의 `recursive resolver`에 질의
3	최종 답은 어디에 있나?	Root -> TLD -> `authoritative name server` 순서로 추적
4	얼마나 믿고 저장할까?	응답의 `TTL`을 최대 상한으로 삼아 캐시
5	실제로 무엇을 얻을까?	`A`, `AAAA`, `CNAME`, `MX` 같은 레코드 응답

가장 짧게 줄이면 이렇습니다.

- **`DNS`는 이름을 IP 주소나 다른 자원 정보로 바꾸는 시스템입니다**
- **클라이언트는 보통 `recursive resolver`에 묻고, resolver가 대신 끝까지 찾아옵니다**
- **한 번 찾은 결과는 보통 `TTL`을 최대 상한으로 삼아 캐시되므로, 설정을 바꿔도 바로 안 바뀔 수 있습니다**

Phase 1. 왜 `DNS`가 필요한가?

먼저 가장 기본부터 보면, 인터넷은 결국 IP 주소 기반으로 동작합니다.

즉, 라우터와 호스트는:

- `93.184.216.34`
- `2001:db8::10`

같은 주소를 기준으로 목적지를 찾습니다.

하지만 사람은 이런 숫자를 기억하기 어렵습니다. 그래서:

```
example.com
```

같은 이름을 쓰고, `DNS`가 이를 실제 네트워크 주소로 바꿉니다.

MDN도 DNS의 가장 대표적인 기능을, 사람이 읽기 쉬운 도메인 이름을 수치 IP 주소로 변환하는 것이라고 설명합니다.

`DNS`는 IP 주소만 주는 시스템은 아니다

여기서 자주 놓치는 점이 있습니다.

`DNS`는 단순히:

```
도메인 -> IP
```

만 저장하는 시스템이 아닙니다.

실제로는:

- 메일 서버 정보
- 다른 이름으로 가리키는 별칭

- 어떤 네임서버가 이 도메인을 책임지는지
- 검증용 텍스트 정보

같은 자원 정보도 함께 다룹니다.

즉, `DNS`는 더 정확히 말하면:

> **도메인 이름 공간에 다양한 자원 레코드를 매핑하는 분산 데이터베이스**

에 가깝습니다.

Phase 2. `DNS` 이름 공간은 어떻게 나뉠까?

`DNS`를 제대로 이해하려면 계층 구조를 먼저 봐야 합니다.

`DNS`는 계층적 이름 공간이다

`RFC 1034`는 DNS를 계층적인 이름 공간으로 설명합니다.

예를 들어:

```
www.example.com.
```

이라는 이름은:

- 최상위의 `.` (root)
- `com`

- `example`
- `www`

처럼 라벨이 계층적으로 이어진 구조입니다.

즉, `example.com`은 `com` 아래의 하위 도메인이고, `www.example.com`은 다시 `example.com` 아래의 하위 이름입니다.

그래서 관리도 나눠 맡을 수 있다

이 계층 구조 덕분에 전 세계 모든 이름을 한 서버가 들고 있을 필요가 없습니다.

예를 들어:

- Root는 최상위 위임 정보만 알고
- `.com`은 `example.com` 같은 하위 도메인의 위임 정보를 알고
- `example.com`의 권한 DNS는 그 안의 실제 레코드를 관리합니다

즉, DNS는 **계층적 이름 구조 + 분산 관리 구조**가 결합된 시스템입니다.

Phase 3. 조회는 실제로 어떻게 진행될까?

많은 분이 이 부분에서 가장 헷갈립니다.

브라우저가 `example.com`을 조회한다고 해서, 곧바로 root 서버에 직접 물어보는 것은 아닙니다.

보통은 `recursive resolver`에 먼저 묻는다

클라이언트는 보통:

- 운영체제가 설정한 DNS 서버
- 회사/집 공유기 DNS
- ISP DNS
- 퍼블릭 리졸버

같은 `recursive resolver`에게 먼저 물어봅니다.

즉, 클라이언트 입장에서는:

```
example.com의 `A` 또는 `AAAA`를 알려 줘
```

라고 한 번 묻고, resolver가 대신 나머지 과정을 수행합니다.

resolver는 보통 반복적으로 추적한다

resolver가 캐시에 답이 없으면, 보통 이런 식으로 따라갑니다.

```
Resolver → Root
Resolver → .com TLD
Resolver → example.com 권한 DNS
```

그리고 최종 응답을 클라이언트에게 돌려줍니다.

즉, 실무에서 흔한 흐름을 단순화하면:

- 클라이언트와 resolver 사이에서는 **재귀 질의**
- resolver는 상위 DNS들의 referral을 따라가며 **반복적으로 추적**

처럼 이해하면 실무 감각에 가깝습니다.

Phase 4. 재귀 질의와 반복 질의는 무엇이 다를까?

이 둘은 말은 비슷하지만 역할이 다릅니다.

재귀 질의

재귀 질의는:

> **"최종 답을 찾아서 나에게 가져와 주세요"**

에 가깝습니다.

즉, 클라이언트는 보통 `recursive resolver`에게:

- 최종 IP
- 또는 `NXDOMAIN` 같은 최종 결과

를 기대합니다.

반복 질의

반복 질의는:

> **"내가 가진 범위 안에서는 이렇게 알고 있고, 다음은 저쪽에 물어보세요"**

에 가깝습니다.

예를 들어 root 서버는 보통:

- "`example.com`의 `A`는 직접 모르지만"
- "`.com`을 담당하는 네임서버는 여기다"

같은 응답을 줍니다.

즉, 반복 질의에서는 서버가 최종 답을 끝까지 대신 찾아오지 않고, **다음 단계의 힌트**를 주는 쪽에 가깝습니다.

그래서 실무에서 보이는 흐름은 보통 이렇다

```
브라우저/OS
  ↓ 재귀 질의
recursive resolver
  ↓ 반복 질의
Root
  ↓ 반복 질의
TLD
  ↓ 반복 질의
권한 DNS
```

이 구조를 이해하면 "`DNS`가 느리다"는 말을 들었을 때:

- 로컬 캐시 문제인지

- resolver 문제인지
- 권한 DNS 문제인지

를 더 잘 나눠 볼 수 있습니다.

Phase 5. Root, TLD, 권한 DNS는 각각 무엇을 들고 있을까?

이 부분도 역할이 섞여 보이면 금방 헷갈립니다.

Root 서버

Root는 최상위 이름 공간의 시작점입니다.

Root 서버가 보통 직접 알려 주는 것은:

- `.com`
- `.net`
- `.org`
- `.kr`

같은 TLD를 어디가 담당하는지입니다.

즉, Root는 모든 도메인의 최종 IP를 아는 서버가 아니라, **최상위 위임 정보를 아는 서버**에 가깝습니다.

TLD 서버

`.com` 같은 TLD 서버는 그 아래의:

- `example.com`
- `openai.com`

같은 도메인에 대한 위임 정보를 관리합니다.

즉, TLD 서버는 보통 "`example.com`을 누가 authoritative하게 관리하는가"를 알려 주는 단계입니다.

권한 DNS 서버

실제 도메인의 레코드를 들고 있는 곳은 보통 `authoritative name server` 입니다.

예를 들어:

- `example.com`의 `A`
- `www.example.com`의 `CNAME`
- 메일용 `MX`

같은 실제 레코드는 여기서 내려옵니다.

즉, 최종적으로 진짜 답을 갖고 있는 곳은 보통 **권한 DNS**입니다.

Phase 6. `TTL`은 왜 중요할까?

실무에서 가장 많이 문제처럼 보이는 개념 중 하나가 `TTL`입니다.

`TTL`은 캐시 수명이다

`TTL`은 응답이 **얼마 동안 캐시될 수 있는지의 최대 상한**을 나타냅니다.

예를 들어 `TTL = 300`이면, 보통 최대 300초 범위 안에서 같은 응답을 캐시해서 재사용할 수 있습니다.

이 덕분에:

- 같은 도메인을 매번 root부터 다시 물을 필요가 없고
- 지연 시간을 줄이며
- 권한 DNS 부하도 줄일 수 있습니다

왜 바꿨는데 바로 안 보일까?

가장 흔한 현상이 이것입니다.

- DNS 레코드를 바꿨는데
- 어떤 사용자는 이미 새 값이 보이고
- 어떤 사용자는 한동안 예전 값이 계속 보입니다

이때 가장 먼저 의심할 것이 `TTL`입니다.

즉, 레코드를 바꿨더라도:

- 브라우저 캐시
- OS 캐시
- resolver 캐시

어딘가에 이전 값이 남아 있으면, 그 캐시가 만료되기 전까지는 예전 값이 계속 보일 수 있습니다.

`TTL`은 짧다고 항상 좋은 것도 아니다

`TTL`을 너무 짧게 잡으면:

- 변경 전환은 빨라질 수 있지만
- 캐시 이점이 줄고
- 권한 DNS 조회 빈도가 늘 수 있습니다

즉, `TTL`은 **전환 민첩성과 캐시 효율 사이의 절충점**에 가깝습니다.

Phase 7. 자주 쓰는 레코드 타입은 무엇일까?

레코드 타입은 DNS를 실무적으로 이해하는 핵심입니다.

`A`

도메인을 `IPv4` 주소에 매핑합니다.

```
example.com → 93.184.216.34
```

AAAA

도메인을 IPv6 주소에 매핑합니다.

```
example.com → 2001:db8::10
```

CNAME

한 이름을 다른 이름의 별칭으로 가리킵니다.

즉:

```
www.example.com → example.com
```

처럼 "이 이름의 최종 답은 저 이름을 따라가라"는 의미입니다.

MX

이 도메인의 메일을 어느 서버가 받을지 지정합니다.

즉, 메일 시스템이 목적지 메일 서버를 찾을 때 주로 씁니다.

NS

어떤 네임서버가 이 도메인 또는 존을 책임지는지 나타냅니다.

즉, 위임 구조를 이해할 때 핵심입니다.

TXT

임의 텍스트 정보를 담는 레코드입니다.

실무에서는:

- 도메인 소유권 검증
- 메일 정책
- 외부 서비스 연동 검증

같은 용도로 매우 자주 씁니다.

PTR

역방향 조회에 쓰입니다.

즉:

```
IP → 도메인 이름
```

방향의 조회에 사용됩니다.

Phase 8. `DNS`는 `UDP`만 쓸까?

많은 분이 `DNS`를 `UDP` 전용처럼 기억하지만, 실제로는 조금 더 정확히 볼 필요가 있습니다.

일반 조회는 보통 `UDP`가 흔하다

`RFC 1035`는 DNS가 `UDP`와 `TCP`를 모두 사용할 수 있음을 정의합니다. 전통적으로 일반 질의/응답은 짧고 빠른 응답이 많아서 `UDP`가 흔했습니다.

즉, `DNS`는:

- 짧은 질의/응답
- 연결 상태 단순성

측면에서 `UDP`가 잘 맞는 경우가 많았습니다.

하지만 `TCP`도 중요하다

예를 들어:

- 응답이 커지는 경우
- 존 전송 같은 작업

에서는 `TCP`가 필요할 수 있습니다.

즉, "`DNS = UDP`"라고 외우면 반쯤만 맞습니다. 더 정확히는:

> **일반 조회는 `UDP`가 흔하지만, DNS 자체는 `TCP`도 사용합니다**

Phase 9. 장애 상황에서는 무엇부터 보면 될까?

실무에서 `DNS` 문제는 네트워크, 인증서, 로드밸런서 문제처럼 보이기도 합니다.

자주 보는 증상

- 어떤 지역에서만 접속이 안 됨
- 배포 후 일부 사용자만 예전 서버로 감
- `A`는 맞는데 `AAAA` 때문에 일부 환경에서 이상함
- 인증서가 맞는데도 엉뚱한 서버로 감
- 도메인은 살아 있는데 메일만 안 됨

이럴 때 먼저 나눠 봐야 할 것

1. 이름이 올바른 IP로 해석되는가?
2. 캐시 때문에 예전 값이 남아 있지 않은가?
3. `A`와 `AAAA`가 의도대로 관리되고 있는가?
4. 권한 DNS와 resolver 중 어디서 값이 달라지는가?
5. 문제 레코드가 `A`인지, `CNAME`인지, `MX`인지 구분했는가?

즉, "`접속이 안 된다`"는 말은:

- `DNS`가 잘못된 것인지
- IP는 맞는데 그다음 `TCP` / `TLS`가 깨지는 것인지

를 먼저 나눠 보는 것이 중요합니다.

핵심만 다시 정리하면

마지막으로 `DNS`를 한 번 더 정리하면 이렇게 볼 수 있습니다.

구분	핵심 의미
`DNS`	이름을 IP 주소와 자원 정보로 바꾸는 계층적 분산 시스템
`recursive resolver`	클라이언트 대신 최종 답을 찾아오는 역할
Root / TLD / 권한 DNS	위임 구조를 따라 최종 답을 찾게 해 주는 계층
`TTL`	응답을 얼마나 캐시할지 정하는 시간
레코드 타입	`A`, `AAAA`, `CNAME`, `MX`, `NS`, `TXT`, `PTR` 등 목적별 정보

핵심 포인트는 다섯 가지입니다.

1. `DNS`는 단순한 전화번호부가 아니라, **도메인 이름 공간을 계층적으로 관리하는 분산 시스템**입니다.
2. 클라이언트는 보통 `recursive resolver`에 재귀 질의를 보내고, resolver가 Root, TLD, 권한 DNS를 반복적으로 따라가며 최종 답을 찾습니다.

3. `TTL`은 성능과 캐시 효율에 중요하지만, 변경 전파가 늦어지는 원인이 되기도 합니다.
4. `DNS`는 `A` / `AAAA`만 다루는 것이 아니라, 별칭, 메일, 위임, 검증용 텍스트 같은 다양한 자원 정보를 저장합니다.
5. 장애 분석에서는 "`DNS`가 틀렸는지"와 "이후 `TCP` / `TLS`가 틀렸는지"를 먼저 분리해서 봐야 합니다.

결국 `DNS`는 "이름을 주소로 바꾼다"는 한 문장으로 시작할 수는 있지만, 실제로는 **이름 공간의 위임 구조, 캐시 전략, 레코드 타입, 질의 방식이 함께 맞물려 인터넷의 첫 단계 응답을 만들어 내는 시스템**에 가깝습니다.