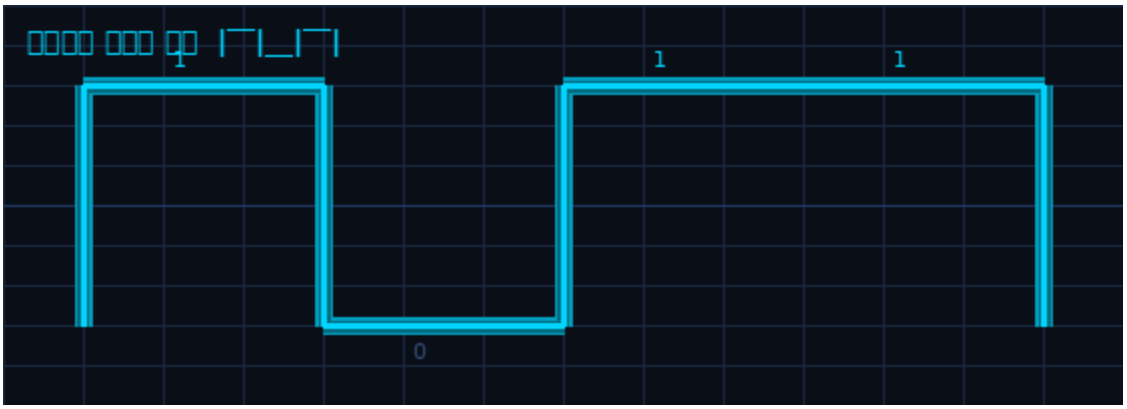
각 계층이 무엇을 해결하는지.
왜 그 위 계층이 필요한지.
이 두 질문을 중심으로 읽으면 흐름이 보입니다.

1계층: 물리 계층 (Physical) — 신호 문제

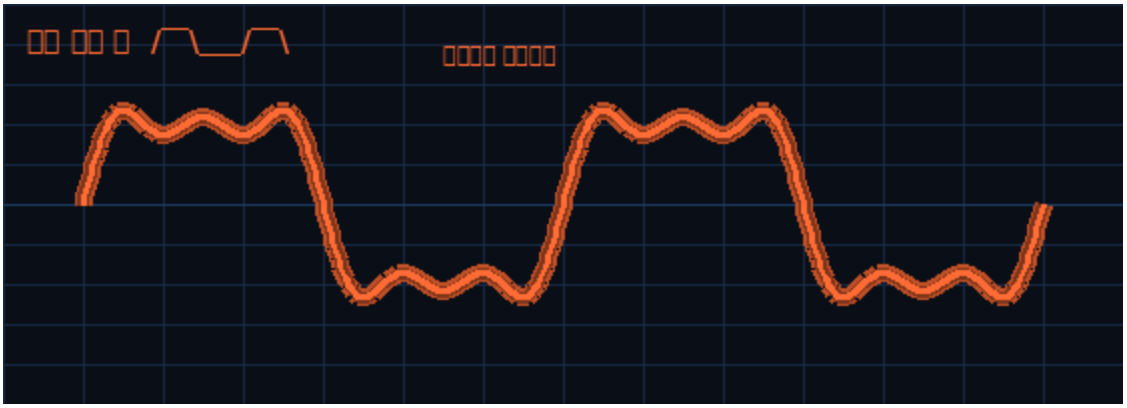
컴퓨터 내부 데이터는 0과 1의 나열입니다. 이를 전송하려면 전기적 신호로 표현해야 합니다.

이때 현실의 전선은 이상적인 사각파를 그대로 전달하지 못합니다. 대역폭 제한과 감쇠 때문에 신호는 완만하게 변형됩니다.

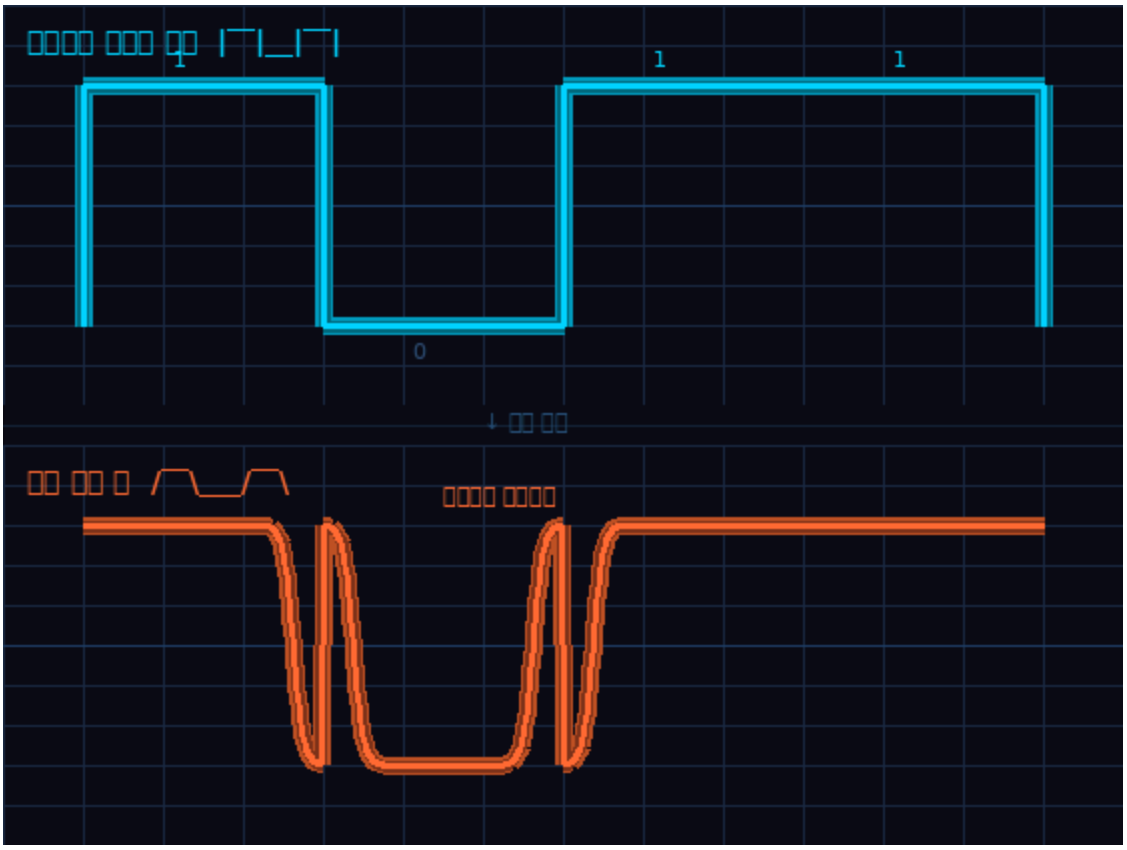
우리가 생각한 0과 1은 이렇게 생겼죠.



하지만 전선을 통과하면 실제로는 이렇게 됩니다:



모서리가 둥글어집니다. 속도를 빠르게 올리면 신호가 완전히 올라가기도 전에 다음 신호가 시작돼서



"이게 0이야? 1이야?" 판별이 어려워집니다.

따라서 비트는 직접 전압 변화로 표현하지 않고, 채널 특성에 맞는 파형으로 변환해 전송합니다. 이 변환/복원 기능을 수행하는 것이 **PHY 칩**입니다.

> **1계층의 본질: 신호 문제**
> 구현 위치: 거의 전적으로 하드웨어 — 송수신 양쪽 NIC의 **PHY 칩**

2계층: 데이터 링크 계층 (Data Link) — LAN 내부 전달 문제

여러 장치가 동일한 물리 매체를 공유하면, 전송 데이터는 모든 장치에 도달합니다.

이를 해결하기 위해 두 가지 규칙이 도입되었습니다.

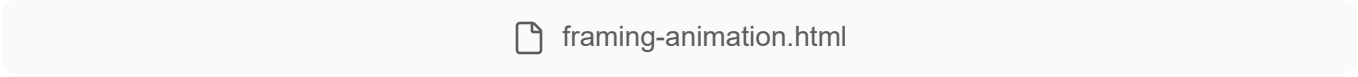
규칙 1: MAC 주소 기반 전달 → 스위치

스위치는 **MAC 주소(2계층 주소)** 기반으로 프레임을 전달하고, 목적지 MAC이 어느 포트에 있는지 테이블을 학습해서 **필요한 포트로만 포워딩**합니다.

switch-animation.html

규칙 2: 프레임 단위 구분

여러 장치가 동시에 보내면 수신 측에는 비트가 **쭉 붙어서** 들어옵니다. 어디서 끊어야 할지 알 수가 없죠. 그래서 데이터를 **프레임(Frame)** 단위로 포장합니다.



실제로는 앞뒤 플래그 비트열, 길이(length) 필드, 혹은 이스케이프(비트/바이트 스터핑) 같은 방식으로 경계를 표시합니다.

스위치는 이 규칙들을 하드웨어적으로 구현한 장비입니다. 다만 2계층 기능이 스위치에만 있는 건 아닙니다. 프레임을 **만드는 쪽(내 NIC)**, **전달하는 쪽(스위치)**, **받는 쪽(상대방 NIC)** 모두에 2계층이 있습니다.

> **2계층의 본질: LAN 내부 전달 문제**
> 구현 위치: 하드웨어 중심 + 일부 커널 — **NIC(MAC 로직) + OS 드라이버 + 스위치(장비)**

(참고) "스위치 하나 = 하나의 네트워크"는 단순화다

현실에서는 **VLAN**을 쓰면 스위치 하나 안에서도 여러 논리 네트워크가 생깁니다. 이 글에서는 이해를 위해 "같은 스위치(같은 VLAN)" = 같은 2계층 네트워크로 봅니다.

3계층: 네트워크 계층 (Network) — 네트워크 간 경로 문제

2계층은 단일 네트워크 내부 규칙입니다. 네트워크가 분리되면, 전역적인 주소 체계와 경로 선택이 필요합니다.

이를 위해 **IP 주소 체계**와 **라우팅 메커니즘**이 도입되었습니다.

IP 주소와 라우터

3계층은 데이터를 **패킷(Packet)** 으로 만들고, 목적지 **IP 주소**를 보고 라우터가 다음 길을 선택합니다.

- **라우팅(Routing)**: 어디로 가야 하는지 계산해서 **테이블을 만드는 과정**
- **포워딩(Forwarding)**: 그 테이블을 보고 패킷을 **실제로 다음으로 넘기는 동작**

이 기능은 어디서 수행되나?

3계층 기능은 두 주체 모두에서 수행됩니다.

호스트(커널) — 패킷 생성 시
커널이 IP 헤더를 붙이고, 라우팅 테이블을 조회해서 다음 홉을 결정한 뒤 NIC으로 넘깁니다.

라우터(장비) — 경로 전달 시
2계층 프레임을 벗기고, IP 헤더를 읽고, 라우팅 테이블을 조회해서 다음 인터페이스로 포워딩합니다.

패킷을 직접 소비하지는 않고 중간에서 경로만 전달합니다.

구분	호스트 (커널)	라우터 (장비)
IP 헤더 생성	✅	❌
IP 헤더 해석	✅	✅
라우팅 테이블 조회	✅	✅
패킷 최종 소비	✅	❌
단순 포워딩	일부	✅

3계층은 특정 장비의 소유물이 아니라, **IP를 이해하는 모든 시스템에 존재하는 기능**입니다.

> **3계층의 본질: 네트워크 간 경로 문제**
> 구현 위치: 커널 + 네트워크 장비 (둘 다 필수)
> — 호스트 → 운영체제 커널 / 네트워크 경계 장비(라우터) → 장비 OS

(참고) DNS와 NAT

- **DNS**: 도메인 → IP로 바꿔주는 "전화번호부"입니다.
- **NAT**: 공유기가 내부 사설 IP ↔ 외부 공인 IP를 매핑해줍니다.

3계층 : IP 프로토콜은 실제로 무엇을 담고 있을까?

3계층은 단순히 "IP 주소를 붙이는 단계"가 아닙니다. IP는 패킷이 여러 네트워크를 통과할 수 있도록 전달에 필요한 **최소한의 정보**를 제공합니다.

IP는 복잡한 품질 보장을 하지 않습니다. 대신, 네트워크 간 전달을 가능하게 하는 기본 규칙과 메타데이터만을 담당합니다.

1️⃣ 비연결성 (Connectionless)

IP는 전송 전에 연결을 설정하지 않습니다. 각 패킷은 독립적으로 처리되며, 이전 패킷과의 상태를 기억하지 않습니다.

즉, "이전에 무엇을 보냈는지"를 고려하지 않고 지금 도착한 패킷만 보고 처리합니다.

2️⃣ 비신뢰성 (Best Effort)

IP는 전달을 "시도"할 뿐, 결과를 보장하지 않습니다. 다음은 IP가 책임지지 않는 것들입니다:

- 패킷 손실 방지
- 순서 보장
- 중복 제거

- 재전송

이러한 기능은 TCP 같은 상위 계층이 담당합니다.

정리: IP는 무엇을 책임질까?

IP의 역할은 네트워크 간 전달에 한정됩니다.

IP가 책임지는 것:

- 전역 주소 체계 (IP 주소)
- 네트워크 간 패킷 전달
- 라우터 통과 횟수 제한 (TTL)
- 필요 시 단편화 (IPv4)

IP가 책임지지 않는 것:

- 도착 보장
- 순서 보장
- 중복 방지
- 재전송

> **한 문장 요약**: IP는 "전달을 위한 최소 규칙"만 제공하는, 비연결·비신뢰 네트워크 프로토콜이다.

ip-encap-demo.html

4계층: 전송 계층 (Transport) — 어느 프로세스에게 줄 건데?

문제: 같은 컴퓨터에 프로그램이 여러 개면?

3계층까지 성공하면 "목적지 컴퓨터"까지는 도착합니다. 그런데 그 컴퓨터 안에는 크롬, 카카오톡, 게임, 서버 프로세스가 동시에 돌고 있습니다. 받은 데이터를 **어느 프로세스에 전달**해야 할까요?

해결: 포트 번호(Port)

4계층은 IP 위에 **포트 번호**를 추가해 프로세스까지 배달합니다.

- IP → 컴퓨터(호스트) 식별
- Port → 그 컴퓨터 안의 프로세스 식별

서버 포트는 약속된 번호(HTTP 80, HTTPS 443 등)를 쓰고, 클라이언트 포트는 OS가 임시로 자동 배정합니다.

- **TCP**: 순서/재전송/흐름제어로 신뢰성 제공
- **UDP**: 그걸 빼서 가볍고 빠름

> **4계층의 본질: 프로세스 문제**
> 구현 위치: 운영체제 **커널(네트워크 스택)**

4계층 심화: 포트 매핑은 실제로 어떻게 동작하나?

"포트 번호로 프로세스를 구분한다"는 설명은 맞지만, 실제 동작은 단순한 포트 번호 비교가 아닙니다. 커널 내부에서 소켓이 생성되고, 연결이 수립되고, 데이터가 전달되는 전체 흐름을 따라가 보겠습니다.

1️⃣ 서버가 준비되는 과정 (소켓 생성)

```
int s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, 0.0.0.0:443);
listen(s, backlog);
```

socket() — 커널 내부에 `struct sock` 생성, 파일 디스크립터(fd) 할당. 아직 포트는 없습니다.

bind() — 해당 소켓을 로컬 IP + 포트(443)에 묶습니다. 커널 내부에 다음과 같이 등록됩니다:

```
(local_ip=*, local_port=443)
상태: BOUND
```

listen() — 상태를 LISTEN으로 변경합니다. 이제 커널은 포트 443으로 들어오는 SYN을 감시합니다.

2️⃣ 클라이언트가 connect() 호출

```
int c = socket(...);
connect(c, server_ip, 443);
```

`connect()` 내부에서 커널이 하는 일: 임시 포트(ephemeral port)를 자동 할당(예: 52341)하고, SYN 세그먼트를 생성해 IP, Ethernet 헤더를 붙여 전송합니다.

3️⃣ SYN 패킷이 서버에 도착하면

```
src: 192.168.0.10:52341
dst: 203.0.113.5:443
SYN
```

커널에서 일어나는 일: NIC → 프레임 수신 → IP 계층에서 목적지 IP 확인 → TCP 계층에서 destination port 확인(443)

4️⃣ 포트 매핑 (소켓 검색 과정)

TCP 스택은 다음 순서로 소켓을 찾습니다.

1단계 — 완전 일치 (4-tuple 매칭)

```
(src IP, src port, dst IP, dst port)
```

아직 연결이 없으므로 실패합니다.

2단계 — LISTEN 소켓 검색 (포트 매칭)

```
(local_port = 443)
상태: LISTEN
```

매칭 성공.

5️⃣ 새 연결 소켓 생성

커널은 새로운 `struct sock`을 생성하고 4-tuple로 등록합니다:

```
(192.168.0.10, 52341, 203.0.113.5, 443)
상태: SYN-RECEIVED
```

이 소켓은 LISTEN 소켓과 별도입니다. 이후 SYN-ACK를 전송합니다.

6️⃣ handshake 완료 후

ACK가 도착하면 상태가 ESTABLISHED로 변경되고, 이 소켓은 커널의 ESTABLISHED hash table에 들어갑니다.

7️⃣ accept()가 하는 일

```
int client_fd = accept(s, ...);
```

`accept()`는 ESTABLISHED 상태의 연결 큐에서 하나를 꺼내 새로운 파일 디스크립터(fd)를 생성하고, 그 fd를 해당 `struct sock`에 연결합니다.

- LISTEN 소켓 fd → 문 역할
- accept()가 반환한 fd → 개별 연결 전용 통로

8️⃣ 이후 데이터 패킷이 도착하면

```
192.168.0.10:52341 → 203.0.113.5:443
```

TCP는 4-tuple로 해시 테이블을 검색합니다:

```
(src IP, src port, dst IP, dst port)
```

정확히 매칭되는 `struct sock`을 발견하고, 해당 소켓의 receive buffer에 데이터를 저장합니다.

9️⃣ recv() 호출 시

```
recv(client_fd, buf, size);
```

커널 동작: fd → 내부 socket 구조체 참조 → receive buffer에서 데이터 읽기 → 사용자 공간으로 복사.

정리: 패킷 → fd까지 연결 흐름

```
패킷 도착
↓
TCP 헤더 분석
↓
4-tuple로 소켓 검색
↓
struct sock 발견
↓
해당 소켓의 receive buffer 저장
↓
파일 디스크립터(fd)와 연결된 소켓
↓
recv() 호출 시 사용자 공간 전달
```

포트 매핑은 단순히 "포트 번호 비교"가 아닙니다. Port는 4-tuple 키의 일부일 뿐입니다.

> **한 문장 핵심**: 포트 매핑은 커널 TCP 스택이 4-tuple을 키로 소켓 구조체를 조회하는 과정이며, 파일 디스크립터는 그 소켓 구조체를 사용자 공간에 연결하는 핸들이다.

5~7계층: 세션 / 프레젠테이션 / 응용

현대 인터넷은 실무적으로 **TCP/IP 스택**이 표준입니다. OSI의 5~7계층 기능들은 사라진 게 아니라 **애플리케이션/라이브러리 안으로 흡수**됐다고 보는 게 더 정확합니다.

- 프레젠테이션 성격 → 직렬화(JSON, Protobuf), 압축(gzip), 암호화(TLS)
- 세션 성격 → 로그인 상태, 토큰/쿠키, Keep-alive
- 응용 → HTTP, DNS, SMTP 등 실제 프로토콜

> **5~7계층의 본질: 애플리케이션 문제**
> 구현 위치: 사용자 공간 — **애플리케이션 / 라이브러리**

계층은 "물리적 위치"가 아니라 "역할 단위"다

OSI 모델은 하드웨어와 소프트웨어를 나누는 모델이 아닙니다. 통신 과정에서 발생하는 문제를 단계별로 분리한 **논리적 구조**입니다. 따라서 계층의 본질은 "구현 위치"가 아니라 **"해결하는 문제"**입니다.

계층	본질	주 구현 위치
1계층	신호 문제	NIC의 PHY 칩 (하드웨어)
2계층	LAN 내부 전달 문제	NIC + 커널 + 스위치
3계층	네트워크 간 경로 문제	커널 + 라우터 (둘 다 필수)
4계층	프로세스 문제	커널
5~7계층	애플리케이션 문제	애플리케이션 / 라이브러리

어떤 계층은 하드웨어에 가깝고, 어떤 계층은 커널에, 어떤 계층은 애플리케이션에 구현되어 있지만 — 그건 결과일 뿐입니다. 계층을 나눈 이유는 항상 "어떤 문제를 해결하느냐"입니다.

캡슐화(Encapsulation): 계층이 쌓이는 방식

각 계층은 상위 계층의 데이터를 "페이로드"로 보고 **헤더를 붙입니다.**

```
Application Data
→ (TCP) Segment
→ (IP) Packet
→ (Ethernet) Frame
→ Bits / Signal
```

수신은 역순으로 "까서" 올라갑니다(역캡슐화). 라우터는 주로 **3계층까지** 보고 넘기고, 4~7은 **종단 호스트**에서 처리됩니다.

마무리: 한 문장 요약

> **각 계층은 "이전 계층만으로 해결 못 하는 문제"를 해결하기 위해 추가된 규칙/모듈이다.**
