1. 네트워크 기초 이론

1.1 컴퓨터 네트워크의 개요

✅ 컴퓨터 네트워크란?

컴퓨터 네트워크(Computer Network)란, **두 대 이상의 컴퓨터 또는 장치들이 데이터를 주고받기 위해 연결된 시스템**을 말한다. 이 연결은 물리적으로 유선 또는 무선으로 이루어질 수 있으며, 다양한 프로토콜과 구조를 이용하여 통신한다.

🧱 네트워크 구성 기본 요소

구성 요소	설명
호스트 (Host)	네트워크에 연결된 컴퓨터, 스마트폰, 서버, IoT 디바이스 등
NIC (Network Interface Card)	네트워크 통신을 담당하는 하드웨어 장치 (유선 LAN, 무선 Wi-Fi)
라우터 (Router)	다른 네트워크 간의 연결을 담당. IP 주소 기반으로 경로 설정
스위치 (Switch)	같은 네트워크 내 여러 장치를 연결. MAC 주소 기반
모뎀 (Modem)	인터넷 서비스 제공자(ISP)와의 연결을 위해 사용하는 장비
AP (Access Point)	무선 네트워크 접속을 위한 장치. 유선 ↔ 무선 브리지 역할

🌐 네트워크의 주요 목적

- **자원 공유**: 프린터, 파일 서버, 인터넷 연결 등
- **통신**: 메일, 메신저, VoIP, 영상통화 등
- **정보 접근**: 웹브라우징, 원격 데스크탑
- **분산 처리**: 클러스터링, 분산 컴퓨팅, 클라우드 등

📊 네트워크 분류

① 범위에 따른 분류

종류	설명
PAN (Personal Area Network)	개인용 네트워크. 블루투스, USB, NFC
LAN (Local Area Network)	집, 회사, 학교 등 한정된 공간
MAN (Metropolitan Area Network)	도시 수준의 네트워크. 대형 캠퍼스, 기업
WAN (Wide Area Network)	인터넷처럼 전세계 규모의 네트워크

② 구성 방식에 따른 분류

방식	설명
클라이언트-서버	중앙 서버에 여러 클라이언트가 연결 (ex. 웹서버)
P2P (Peer-to-Peer)	모든 노드가 동등한 위치에서 직접 통신 (ex. 토렌트)
혼합형	위 두 가지를 혼합한 하이브리드 구조

🔁 네트워크 통신의 기본 개념

1. 통신 방향성

- **단방향(Simplex)**: 한쪽에서만 전송 (예: 라디오)
- **반이중(Half-Duplex)**: 양방향이지만 동시에 불가 (예: 무전기)
- **전이중(Full-Duplex)**: 양방향 동시에 가능 (예: 전화)

2. 주소 체계

- **MAC 주소**: 물리적 주소, 48비트
- **IP 주소**: 논리적 주소, IPv4(32bit), IPv6(128bit)
- **포트 번호**: 프로세스 식별, 0~65535 범위

3. 데이터 전송 단위

- 비트 (bit): 0 또는 1
- 바이트 (Byte): 8비트
- 프레임, 패킷, 세그먼트: 계층별 데이터 단위

💡 실세계 예시: 웹 페이지 요청 과정

```
[브라우저] → www.example.com 입력
  ↓
[DNS 요청] → IP 주소 확인
  ↓
[TCP 연결 수립] → 3-way handshake
  ↓
[HTTP 요청 전송] → GET /index.html
  ↓
[서버 응답] → HTTP/1.1 200 OK
  ↓
[HTML 수신] → 브라우저에 렌더링
```

📚 정리: C 프로그래머가 반드시 이해해야 할 점

항목	이유
IP / 포트	소켓 주소 설정 (`struct sockaddr_in`)
TCP/UDP	소켓 타입 선택 (`SOCK_STREAM`, `SOCK_DGRAM`)
DNS, 라우팅	`gethostbyname`, `getaddrinfo`
프로토콜 계층	어떤 계층에서 동작하는지 명확히 이해
통신 방향성	블로킹, 논블로킹 처리에 영향

1.2 OSI 7계층 vs TCP/IP 4계층

🧠 OSI 7계층: 이론적 완성체, 역할 기반 분리

OSI(Open Systems Interconnection) 모델은 **국제표준화기구(ISO)**가 정의한 통신 구조 모델로, **기능별로 7단계로 나눈 통신 계층 구조**이다.
이 모델은 실제 통신을 설명하는 데 가장 많이 쓰이며, 각 계층은 **직접 아래 계층과만 통신**하고, **상호 독립적인 책임**을 가진다.

📚 OSI 7계층 요약표

계층	이름	주요 기능	대표 예시
7	응용 계층 (Application)	사용자 인터페이스, 응용프로토콜 구동	HTTP, FTP, SMTP
6	표현 계층 (Presentation)	데이터 형식 변환, 인코딩, 압축, 암호화	TLS, JPEG, UTF-8
5	세션 계층 (Session)	세션 연결, 유지, 동기화	TLS Handshake, NetBIOS
4	전송 계층 (Transport)	종단 간 통신, 흐름 제어, 신뢰성 확보	TCP, UDP
3	네트워크 계층 (Network)	라우팅, 논리 주소(IP), 경로 결정	IP, ICMP, ARP
2	데이터링크 계층 (Data Link)	MAC 주소, 프레임, 오류 검출	Ethernet, ARP
1	물리 계층 (Physical)	전기적 신호, 물리 매체, 비트 전송	UTP, 광섬유, 전압 레벨

🧪 TCP/IP 4계층: 실전 중심의 인터넷 구조

OSI는 이론적 완성체지만, **실제 운영체제와 인터넷에서 사용되는 것은 TCP/IP 모델**이다.
이 모델은 **4계층으로 단순화**되어 있으며, **OSI 계층을 일부 통합한 구조**야.

🌐 TCP/IP 4계층 요약표

계층	이름	OSI 계층 매핑	주요 프로토콜 예
4	응용 계층 (Application)	7 + 6 + 5	HTTP, FTP, DNS, TLS

계층	이름	OSI 계층 매핑	주요 프로토콜 예
3	전송 계층 (Transport)	4	TCP, UDP
2	인터넷 계층 (Internet)	3	IP, ICMP
1	네트워크 인터페이스 계층 (Link)	2 + 1	Ethernet, ARP

🔄 OSI vs TCP/IP 계층 비교 (매핑 표)

OSI 7계층	TCP/IP 4계층	설명
7. 응용 계층	4. 응용 계층	사용자 서비스 제공
6. 표현 계층	〃	데이터 형식, 인코딩
5. 세션 계층	〃	세션 연결 및 동기화
4. 전송 계층	3. 전송 계층	포트 번호 기반 종단 간 통신
3. 네트워크 계층	2. 인터넷 계층	IP 주소 기반 경로 결정
2. 데이터링크 계층	1. 링크 계층	MAC 주소, 프레임 처리
1. 물리 계층	〃	전기 신호, 케이블 등

📦 캡슐화 구조: 실제 통신 시 데이터 흐름

```
css코드 복사[응용 계층]    HTTP 헤더 + 데이터
 ↓
[전송 계층]    TCP 헤더 추가
 ↓
[인터넷 계층]  IP 헤더 추가
 ↓
[링크 계층]    MAC 헤더 + 실제 전송
```

- 즉, 각 계층은 **상위 계층의 데이터를 감싸서** 전달하며, 이를 **Encapsulation (캡슐화)**라고 해.
- 수신 측에서는 이 역과정인 **Decapsulation (캡슐 해제)**를 수행해.

🧰 리눅스 C 프로그래밍 관점에서의 계층 위치

계층	프로그래밍 관점
응용 계층	직접 구현 (ex. HTTP 요청 처리)
전송 계층	`socket()`, `send()`, `recv()`
인터넷 계층	`struct sockaddr_in`, `inet_pton()`

계층	프로그래밍 관점
링크 계층	`libpcap`, `RAW socket`, NIC ioctl
물리 계층	직접 제어 X, 장치 드라이버 또는 NIC 전용 API

🧩 실전 적용 요약

- **TCP/IP는 OSI를 현실화한 모델**로, 네트워크 프로그래밍의 모든 기반
- **C에서 사용하는 소켓 API는 TCP/IP 모델을 기준**으로 설계되어 있음
- **TCP/IP의 4계층 중, 주로 응용 ~ 인터넷 계층까지만** 프로그래머가 직접 다룸
- **RAW 소켓**을 사용할 경우 데이터링크 계층까지 접근 가능

🔍 정리 요약

구분	OSI 모델	TCP/IP 모델
계층 수	7 계층	4 계층
주 목적	이론적 분석, 설계 기준 제공	실제 인터넷 구현 기반
사용 범위	학습, 표준화 문서, 장비 설계 등	운영체제, 리눅스 네트워크 구현
직접 구현 가능 계층	주로 7~4 계층	4~1 계층 중 4~2계층 실전 구현

1.3 OSI 1계층: 물리 계층 (Physical Layer)

✅ 개요

물리 계층은 **네트워크 통신에서 가장 하드웨어적인 계층**으로, **"0과 1의 비트"를 실제 물리적인 신호로 바꿔 전송하는 계층**이다.
즉, 논리적인 데이터가 실제 **전압, 광신호, 전자기파**로 바뀌는 층이 바로 이곳이다.

🧱 주요 역할

기능	설명
비트 전송	디지털 데이터를 전기/광학/무선 신호로 변환하고 전송
동기화	송신자와 수신자의 클럭 타이밍 맞추기
인코딩	NRZ, Manchester 등으로 0과 1을 신호화
전송 매체 정의	유선(UTP, 광섬유), 무선(Wi-Fi, RF) 결정
전송 모드	simplex, half-duplex, full-duplex
물리 인터페이스 정의	커넥터 종류, 핀 배열, 전압 규격 등

🌐 전송 매체 종류

종류	설명
UTP/STP 케이블	이더넷 LAN 케이블, 흔히 사용하는 Cat.5e/Cat.6
광섬유 (Fiber)	레이저 기반 고속 통신 (Gbps~Tbps)
동축 케이블	케이블 TV, 고정 회선망
무선 (Wireless)	Wi-Fi, Bluetooth, 5G 등 전자기파 기반

🔌 인터페이스 예: NIC, PHY 칩, 커넥터

- **NIC (Network Interface Card)**: 컴퓨터와 물리적 네트워크를 연결하는 장치
- **PHY (Physical Layer Transceiver)**: MAC에서 생성된 비트스트림을 실제 신호로 변환
- **RJ45**: 일반적인 LAN 케이블 커넥터
- **SFP 모듈**: 광섬유용 트랜시버

🔍 인코딩 방식 (신호화)

방식	특징
NRZ (Non-Return-to-Zero)	0과 1을 전압 High/Low로 표현
Manchester	비트마다 신호 전환 포함, 클럭 동기화 쉬움
4B/5B, 8B/10B	효율 및 동기화 목적의 고급 인코딩

🧪 동작 예시: 이더넷 통신 흐름

1. TCP/IP 계층에서 전송할 데이터가 내려옴
2. MAC 계층이 프레임 생성
3. **PHY가 0과 1을 Manchester 코드로 변환**
4. RJ45 포트를 통해 신호가 케이블로 나감
5. 상대편 PHY가 신호를 **디지털 비트로 복원**
6. MAC → IP → TCP → 응용 계층으로 전달

🛠️ 리눅스 C 프로그래머 관점

C 코드에서 직접적으로 **물리 계층을 제어하지는 않지만**, 간접적으로 다음을 통해 연관이 있다:

1. `ioctl()`을 통한 NIC 상태 확인

```c
#include <sys/ioctl.h>
#include <net/if.h>
#include <stdio.h>
#include <string.h>

int main() {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    struct ifreq ifr;
    strcpy(ifr.ifr_name, "eth0");

    ioctl(sock, SIOCGIFFLAGS, &ifr);
    if (ifr.ifr_flags & IFF_UP)
        printf("인터페이스 eth0: UP\n");
    else
        printf("인터페이스 eth0: DOWN\n");
}
```

2. `ethtool`로 PHY 상태 확인 (외부 도구)

```
sudo ethtool eth0
```

결과 예:

```
Speed: 1000Mb/s
Duplex: Full
Link detected: yes
```

3. `dmesg` 또는 `/sys/class/net/eth0/` 접근

```
cat /sys/class/net/eth0/speed
cat /sys/class/net/eth0/duplex
```

🧬 물리 계층이 중요한 이유

- 느려지면 전체 네트워크가 병목됨 (케이블 문제, 접속 불량 등)
- **오류의 시작점**: CRC 에러, frame drop, Link down
- 고속 네트워크 튜닝 시 PHY의 역할 중요
- **IoT/임베디드 장비**에서는 PHY 직접 제어가 필요할 수 있음 (SPI/Ethernet PHY)

📌 프로그래머가 기억해야 할 핵심 요약

키포인트	설명
제어 불가능	물리 계층은 보통 커널/드라이버/하드웨어가 관리
정보 조회	`ioctl`, `ethtool`, `/sys/class/net/` 경로로 확인
오류 진단	패킷 안 잡히면 물리 계층부터 점검 (link, speed, duplex)
개발자 입장	PHY 모듈 설정, 링크 상태 확인, 속도 조정 정도까지 접근

• 신호, 케이블, 커넥터, NIC, 전송 매체(1계층 구성 요소)

📡 1. 신호 (Signal)

🔷 역할

네트워크에서 `신호`란, **디지털 데이터(0과 1)를 실제 전송 가능한 물리적 형태로 변환한 것**이다. 이건 반드시 전송 매체를 통해 전달돼야 하고, 송신 측과 수신 측이 **정확하게 인식 가능한 규격**이어야 한다.

🔶 종류

신호 유형	설명	사용 예
전기 신호	전압의 High/Low 전환	이더넷, RS-232
광 신호	레이저 빛의 On/Off	광 케이블 기반 인터넷
무선 신호	라디오파 (전자기파)	Wi-Fi, Bluetooth

🔷 신호 인코딩

방식	설명
NRZ (Non-Return to Zero)	1: 고전압, 0: 저전압
Manchester	1비트당 전압 전환으로 클럭 내장
PWM, ASK, FSK	무선용 주파수/진폭 기반 변조

🔌 2. 케이블 (Cable)

🧱 역할

송수신 장치 간 물리적 신호를 전달하는 경로.
전달 속도, 안정성, 거리 등에 큰 영향을 미친다.

🔧 유선 케이블 종류

종류	설명	최대 속도/거리
UTP (Unshielded Twisted Pair)	보통 우리가 쓰는 LAN 케이블 (Cat.5e, Cat.6 등)	최대 1~10 Gbps / ~100m
STP (Shielded Twisted Pair)	외부 전자기 간섭 차단	고노이즈 환경에서 안정적
Coaxial (동축 케이블)	TV, 오래된 이더넷	최대 수백 Mbps
광섬유 (Fiber Optic)	레이저 기반, 매우 긴 거리와 빠른 속도	수십 km, 수 Gbps 이상

🪝 3. 커넥터 (Connector)

🧩 역할

케이블을 장비나 NIC에 물리적으로 연결해주는 **종단 인터페이스**이다.

🔗 주요 커넥터 종류

커넥터	설명	사용 매체
RJ45	UTP용, 흔한 이더넷 커넥터 (8핀)	Cat.5/6/7 케이블
LC, SC	광섬유용 커넥터	Fiber Optic
BNC	동축 케이블용	오래된 네트워크, CCTV
SFP	Hot-pluggable 광/전기 트랜시버 모듈용	기가비트 스위치, 라우터

🧩 4. NIC (Network Interface Card)

💡 정의

NIC는 **호스트(PC/임베디드 보드 등)와 네트워크를 연결**하는 하드웨어 장치야. OSI 모델에서 1계층과 2계층(데이터링크) 사이를 담당한다.

🔩 구성 요소

컴포넌트	역할
MAC	MAC 주소 보관 및 프레임 처리
PHY	물리 계층 → 전기/광 신호 변환
버퍼	전송 큐, 수신 큐
DMA 인터페이스	메모리 직접 접근 지원

컴포넌트	역할
PCIe 또는 USB 인터페이스	CPU/메인보드와 연결

🔧 드라이버 예

- 리눅스에선 `eth0`, `enp0s3` 등으로 인식됨
- 드라이버는 `/sys/class/net/`, `ethtool`, `ioctl`로 정보 확인 가능

🌐 5. 전송 매체 (Transmission Medium)

📦 개요

데이터를 실제로 전달하는 **물리적 또는 무선적 경로**. 신호가 이 매체를 타고 전달됨.

📶 분류

분류	매체 예	특징
유선 (Guided)	UTP, STP, 동축, 광섬유	보안/속도 높고 간섭 적음
무선 (Unguided)	전자기파, 적외선, 위성	설치 자유로움, 간섭 많음

📈 성능 요소

- **대역폭**: 전송 가능한 최대 속도
- **감쇠 (Attenuation)**: 신호 세기 감소
- **잡음 (Noise)**: 외부 간섭
- **지연 (Latency)**: 도달 시간

🎯 정리 요약

요소	핵심 요약
신호	0과 1을 전기/광/무선으로 바꿈
케이블	물리적 매체 (Cat.5, 광섬유 등)
커넥터	장치와 케이블 연결 인터페이스
NIC	컴퓨터와 네트워크의 중개자
전송 매체	신호가 실제로 전달되는 경로

🧠 C언어 프로그래머 입장에서 중요한 점

항목	예시/도구
NIC 설정	`ifconfig`, `ip link`, `ioctl()`
속도/링크 상태	`ethtool`, `/sys/class/net/eth0/`
장애 원인 분석	`ping`, `tcpdump`, `dmesg`
PHY 제어 (임베디드)	SPI/Ethernet PHY 레지스터 직접 제어

- # 실제 데이터 흐름 및 디바이스 관점

📦 시나리오: PC A → PC B로 데이터 전송

> 전송 목표: PC A가 PC B에게 TCP/IP 기반으로 HTTP 요청 전송
> 전송 경로: PC A → 스위치 → PC B
> 물리 계층의 역할: 비트를 전기 신호로 바꿔 실제 케이블을 통해 전송

⛓️ 전체 흐름 요약

① 상위 계층에서 데이터 생성

1. 애플리케이션 계층에서 HTTP 요청 생성
2. TCP 헤더, IP 헤더가 붙으며 하나의 패킷으로 캡슐화
3. MAC 헤더가 붙으면서 **프레임**이 완성됨 → **NIC로 전달됨**

⚙️ 물리 계층 동작 흐름 상세

⬇️ 송신 측 (PC A 기준)

단계	디바이스	동작 내용
1	**운영체제 커널**	완성된 Ethernet 프레임을 NIC로 전달 (`DMA`)
2	**NIC의 MAC**	MAC 주소 확인 후 프레임을 `PHY`로 전달
3	**PHY (Physical Layer)**	0과 1의 비트를 **전기 신호 또는 광 신호**로 변환
4	**RJ45 포트 + 케이블**	실제로 신호를 케이블로 전송 (twisted pair 전압 차)

🔧 실제 신호 예

- 1비트: 0 → 전압 0V
- 1비트: 1 → 전압 2.5V
  (인코딩 방식에 따라 전환 구조 다름, 예: Manchester)

⬆️ 수신 측 (PC B 기준)

단계	디바이스	동작 내용
1	**RJ45 + 케이블**	전기 신호 수신
2	**PHY**	신호를 **디지털 비트 스트림**으로 복원
3	**MAC**	프레임을 확인하고 IP 계층으로 전달
4	**커널**	상위 계층으로 전달되어 TCP/IP 분해 후 HTTP 요청 처리

🪢 디바이스별 관여 범위

디바이스	물리 계층에서의 역할
NIC (MAC + PHY)	비트 → 신호 변환 및 반대 방향 수신 처리
PHY (Transceiver)	물리적 변환의 핵심. 전기 ↔ 디지털 변환 담당
RJ45 / 포트	매체와 장치 연결. 신호 손실 최소화 구조
케이블 (Cat.6 등)	신호 전달 매체. 거리, 감쇠율, 대역폭에 영향
스위치	물리 계층 + 데이터링크 계층 역할. 프레임 전달 담당

🧪 리눅스에서의 물리 계층 확인 실습 예

🧑‍💻 1. 링크 상태 확인

```
cat /sys/class/net/eth0/carrier
# 1이면 연결됨, 0이면 끊김
```

🧠 2. 속도 및 듀플렉스 모드

```
sudo ethtool eth0
```

```
Speed: 1000Mb/s
Duplex: Full
Link detected: yes
```

🧰 3. 패킷이 NIC까지 도달했는지 확인

```
sudo tcpdump -i eth0
```

> 아무것도 안 잡힌다면 → 소켓 프로그래밍 문제
> 패킷이 들어오지 않음 → 링크 불량 or PHY 문제

🧬 PHY 칩 예시 (임베디드 개발자용)

칩 이름	인터페이스	용도
DP83848	MII, RMII	STM32, Cortex-M MCU용
KSZ8081	RMII	이더넷 PHY 모듈
RTL8211	RGMII	고속 기가비트 PHY, 라즈베리파이/보드에서 사용

👉 이들은 **SPI, I2C, MDIO** 등의 버스를 통해 제어 가능하고, 저수준 제어가 가능한다.

🎯 요약 정리

요소	흐름 상 역할
OSI 1계층	신호 수준의 데이터 전송
NIC	MAC + PHY 포함, 1~2계층 담당
PHY	디지털 ↔ 아날로그 신호 변환
케이블	실제 신호 이동 경로
RJ45	포트 연결 인터페이스
전송	0과 1 → 신호 → 케이블 → 신호 → 0과 1

1.4 OSI 2계층: 데이터 링크 계층 (Data Link Layer)

✅ 개요

데이터 링크 계층은 **물리 계층에서 송수신된 신호를 신뢰성 있는 프레임으로 포장하거나 해석하는 계층**이다.
 즉, 이 계층은 실제 네트워크 장치 간에 **프레임 단위로 통신**을 하게 만들어주고, **MAC 주소를 사용하여 식별**하고, 오류 검출까지 담당한다.

🔍 주요 역할

기능	설명
프레이밍 (Framing)	상위 계층에서 전달된 데이터를 프레임 단위로 나눔
주소 지정 (Addressing)	**MAC 주소** 사용. 물리적 주소 기반 통신
오류 검출 (Error Detection)	CRC 사용하여 오류 탐지 (수정은 안함)
흐름 제어 (Flow Control)	송수신 속도 조절 (일부 프로토콜만)
접근 제어 (Media Access Control)	여러 장치가 공유하는 매체 사용 시 충돌 방지 (CSMA/CD 등)

📦 프레임 구조 (이더넷 기준)

```
[프리앰블][목적지 MAC][출발지 MAC][타입][데이터][FCS]
```

필드명	설명
프리앰블	101010... 패턴으로 수신 동기화
목적지 MAC 주소	수신 대상
출발지 MAC 주소	송신자
타입 (Type)	상위 프로토콜 식별 (ex: 0x0800 = IPv4)
데이터	최대 1500바이트
FCS (Frame Check Sequence)	CRC32 기반 오류 검출

🧠 MAC 주소란?

- NIC 제조사에서 하드웨어에 부여된 **48비트 물리 주소**
- 16진수 12자리: `00:1A:2B:3C:4D:5E`
- 네트워크 내에서 **장치 식별**을 담당
- 변경도 가능 (리눅스에서 `ip link set` 명령 등)

🛰️ 통신 유형

방식	설명
유니캐스트	특정 MAC 주소만 지정
브로드캐스트	`FF:FF:FF:FF:FF:FF` → 네트워크 전체
멀티캐스트	특정 그룹 지정 (IPv6에서 많이 사용)

🔁 브로드캐스트 기반의 ARP 동작 예

예: `ping 192.168.0.23`을 수행할 때

1. 내 컴퓨터는 **192.168.0.23의 MAC 주소를 모름**
2. ARP 요청 프레임을 브로드캐스트로 보냄
   (목적지 MAC: `FF:FF:FF:FF:FF:FF`)
3. 대상 컴퓨터가 자신의 IP에 해당하면 **ARP 응답 전송**
4. MAC 주소 캐싱 후, 이후엔 유니캐스트로 통신

→ 이 전체 과정은 **2계층에서 MAC 주소 기반으로 처리**

⚙️ 리눅스 C 프로그래머 관점

✅ 1. 프레임 캡처 (libpcap 사용)

```
#include <pcap.h>
pcap_t *handle = pcap_open_live("eth0", BUFSIZ, 1, 1000, errbuf);
pcap_loop(handle, -1, callback, NULL);
```

콜백 안에서는 이더넷 헤더를 다음처럼 파싱 가능:

```
struct ether_header {
  u_char ether_dhost[6];
  u_char ether_shost[6];
  u_short ether_type;
};
```

✅ 2. RAW 소켓을 이용한 직접 프레임 수신

```
int sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

- 이 소켓은 **2계층의 전체 프레임**을 직접 읽어들인다.
- `recvfrom()`으로 이더넷 헤더 포함 프레임을 수신
- `sendto()`로 직접 프레임을 송신하는 것도 가능

🛠️ 도구 기반 확인

1. MAC 주소 확인

```
ip link show dev eth0
```

2. ARP 테이블 확인

```
ip neigh
```

3. 브로드캐스트 프레임 확인

```
sudo tcpdump -i eth0 arp
```

📌 고급 프로토콜과의 연관

프로토콜	설명
ARP	IP ↔ MAC 주소 매핑
LLDP	장비 정보 교환
VLAN (802.1Q)	프레임에 VLAN 태그 삽입
PPP	직렬 통신 기반 2계층 프로토콜

🎯 요약 정리

항목	키포인트
MAC 주소	하드웨어 단위 주소. 유일 식별자
프레임	2계층의 전송 단위
오류 검출	FCS (CRC32)로 수행
브로드캐스트	네트워크 전체에게 전달 (ARP 등)
C 프로그래머 접근	`libpcap`, RAW socket, ioctl

• MAC 주소, 이더넷, ARP, 오류 검출

📍 1. MAC 주소 (Media Access Control Address)

✅ 정의

- **NIC에 내장된 고유 식별자**
- **48비트 (6바이트)** → 16진수 12자리로 표현됨
  - 예: `00:1A:2B:3C:4D:5E`
- **제조사 코드(OUI) + 일련번호**

✅ 분류

구분	예시	설명
Unicast MAC	`00:11:22:33:44:55`	특정 장치 하나를 식별
Broadcast MAC	`FF:FF:FF:FF:FF:FF`	같은 네트워크의 모든 장치 대상
Multicast MAC	`01:00:5E:xx:xx:xx`	특정 그룹 식별 (IPv6 등에서 활용)

✅ 리눅스에서 확인/변경

```
ip link show dev eth0
sudo ip link set dev eth0 address 00:11:22:33:44:55
```

🧱 2. 이더넷 (Ethernet)

✅ 정의

- 2계층의 대표적인 프레임 기반 통신 기술
- MAC 주소 기반
- **CSMA/CD (Carrier Sense Multiple Access with Collision Detection)**: 충돌 감지 방식

✅ 이더넷 프레임 구조 (IEEE 802.3)

필드	크기	설명
프리앰블	7B	수신 동기화 (`101010...`)
SFD	1B	프레임 시작 알림 (`10101011`)
목적지 MAC	6B	수신 대상
출발지 MAC	6B	송신자 주소
Type/Length	2B	상위 프로토콜 또는 데이터 길이
데이터	46~1500B	실제 전송할 정보
FCS (Frame Check Sequence)	4B	CRC 오류 검출용

> 프리앰블 + SFD는 일반적인 프레임 분석 툴에서 보이지 않음 (PHY 단에서 제거됨)

✅ 이더넷 프로토콜 Type 예

Type	설명
`0x0800`	IPv4
`0x0806`	ARP
`0x86DD`	IPv6

📡 3. ARP (Address Resolution Protocol)

✅ 개요

- **IP 주소 → MAC 주소로 변환**
- 네트워크 계층과 데이터링크 계층의 연결자
- 브로드캐스트 방식 사용 (`FF:FF:FF:FF:FF:FF`)

✅ 동작 과정

1. 송신자: "192.168.0.10의 MAC 주소 누구야?" → ARP 요청
2. 모든 장치 수신 후, 대상 장치가 "내가 192.168.0.10야!" → ARP 응답
3. 응답을 받은 송신자는 MAC 주소를 **ARP 캐시 테이블**에 저장

✅ ARP 테이블 확인

```
ip neigh
```

```
192.168.0.10 dev eth0 lladdr 00:1a:2b:3c:4d:5e REACHABLE
```

✅ ARP 패킷 구조 (간단 요약)

필드	내용
Hardware type	Ethernet (1)
Protocol type	IPv4 (0x0800)
Opcode	요청(1), 응답(2)
Sender MAC, IP	송신자의 주소
Target MAC, IP	수신자의 주소

🛡️ 4. 오류 검출 (FCS - Frame Check Sequence)

✅ 개요

- **이더넷 프레임의 마지막 4바이트**는 CRC32 기반으로 계산된 **오류 검출 코드**
- **수정은 못 하고, 검출만 함**

✅ 동작 흐름

1. 송신 측: MAC에서 프레임 데이터 기반으로 CRC32 계산 → FCS에 삽입
2. 수신 측: 동일한 방식으로 CRC 재계산
    - 다르면 **프레임 폐기**

> FCS는 리눅스 사용자 공간에서는 **보통 볼 수 없음**
> `RAW socket`이나 `libpcap`을 **드라이버 설정에 따라** 일부 볼 수 있음

🧪 실전 코드/도구 기반 확인

✅ 1. 프레임 분석 (tcpdump)

```
sudo tcpdump -i eth0 -e
```

```
08:00:27:12:34:56 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42
```

✅ 2. `libpcap`을 이용한 C 코드 내 MAC 파싱

```
struct ether_header {
    u_char dst_mac[6];
    u_char src_mac[6];
    u_short ether_type;
};
```

📌 정리 요약표

요소	설명	리눅스 도구/코드 접근
MAC 주소	NIC의 고유 식별자	`ip link`, `ioctl()`
이더넷 프레임	2계층 데이터 단위	`RAW socket`, `libpcap`
ARP	IP ↔ MAC 변환 프로토콜	`ip neigh`, `tcpdump`
FCS (CRC32)	프레임 오류 검출	일반적으론 커널/드라이버 레벨

- # 프레임 구조, 브로드캐스트/유니캐스트

📦 1. 이더넷 프레임 구조 (Ethernet Frame Structure)

✅ 기본 프레임 형식 (IEEE 802.3)

```
+----------------+-----------+------------+------------------+------------+
| Destination MAC| Source MAC| Type/Length| Payload (Data)   | FCS (CRC32)|
| (6 bytes)      | (6 bytes) | (2 bytes)  | (46 ~ 1500 bytes)| (4 bytes)  |
+----------------+-----------+------------+------------------+------------+
```

> ⚠️ 실제로는 앞에 **프리앰블(7B)** + **SFD(1B)**가 더 존재하지만, 이는 **PHY에서 처리**되며 커널 레벨 이상에서는 보통 볼 수 없어.

✅ 각 필드 설명

필드	길이	설명
DST MAC	6B	프레임의 목적지 MAC 주소
SRC MAC	6B	프레임을 보낸 송신자의 MAC 주소
Type / Length	2B	IP(0x0800), ARP(0x0806) 등
Payload	46~1500B	실질적인 데이터 (TCP/IP 등 포함)
FCS	4B	오류 검출용 CRC32 값 (드라이버/PHY에서 처리)

✅ MTU (Maximum Transmission Unit)

- 일반 Ethernet의 최대 Payload: **1500 bytes**
- 이를 초과하면 **IP 계층에서 분할(fragmentation)** 처리됨

📡 2. 유니캐스트 / 브로드캐스트 / 멀티캐스트

📍 A. 유니캐스트 (Unicast)

- **특정 하나의 MAC 주소**를 대상으로 프레임 전송
- 가장 일반적인 통신 방식

```
// DST MAC: 00:1A:2B:3C:4D:5E
```

💡 예: 웹 브라우저 → 특정 서버, SSH 접속, ARP 응답 등

📍 B. 브로드캐스트 (Broadcast)

- 모든 네트워크 장비에게 전송
- **DST MAC: FF:FF:FF:FF:FF:FF**
- **스위치가 해당 프레임을 모든 포트로 복사해서 전달**

💡 예: ARP 요청, DHCP Discover

```
Sender: 12:34:56:78:9A:BC
Target: FF:FF:FF:FF:FF:FF
```

> 브로드캐스트는 **서브넷 내부에서만 유효**함. 라우터는 일반적으로 전달하지 않음.

📍 C. 멀티캐스트 (Multicast)

- 하나의 **특정 그룹**을 대상으로 전달
- 예를 들어: **01:00:5E:00:00:FB** → mDNS 전용 멀티캐스트

범위	MAC 시작 주소
IPv4 멀티캐스트	01:00:5e:xx:xx:xx
IPv6 멀티캐스트	33:33:xx:xx:xx:xx

💡 예: IPTV, mDNS, OSPF, VRRP

🧪 실전 분석: tcpdump 프레임 캡처 예

```
sudo tcpdump -e -i eth0
```

유니캐스트 예:

```
12:34:56:78:9a:bc > 00:1a:2b:3c:4d:5e, ethertype IPv4 (0x0800)
```

브로드캐스트 예 (ARP 요청):

```
12:34:56:78:9a:bc > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806)
```

멀티캐스트 예 (mDNS):

```
12:34:56:78:9a:bc > 01:00:5e:00:00:fb, ethertype IPv4 (0x0800)
```

⚙️ C언어에서 프레임 수신하기 (RAW 소켓)

```
int sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

수신한 데이터에서 다음처럼 프레임 파싱:

```
struct ether_header {
    uint8_t dst[6];
    uint8_t src[6];
    uint16_t ethertype;
};
```

이후 `dst`가 `0xFF...`이면 브로드캐스트, `01:00:5E...`면 멀티캐스트, 그 외는 유니캐스트로 판단할 수 있음.

🧠 정리 요약

항목	설명	예시
프레임	이더넷의 전송 단위	MAC + Type + Data + FCS
유니캐스트	특정 1명에게만	일반 데이터 전송
브로드캐스트	모든 장치에게	ARP 요청, DHCP Discover
멀티캐스트	그룹에게만	mDNS, IPTV, VRRP

1.5 OSI 3계층: 네트워크 계층 (Network Layer)

✅ 핵심 개념

네트워크 계층의 가장 중요한 목적은 단 하나이다:

> **"발신지에서 수신지까지 IP 패킷을 목적지에 정확히 전달하는 것"**

📌 주요 기능 정리

기능	설명
논리 주소 지정	IP 주소 (IPv4, IPv6) 부여 및 처리
라우팅(Routing)	목적지 IP 주소에 따라 패킷 전송 경로 선택
패킷 포워딩	패킷을 다음 홉(next hop)으로 전송
패킷 분할/재조립	MTU보다 큰 데이터는 조각내어 처리
에러 제어	ICMP를 통한 경로 오류 통지

🧱 네트워크 계층 데이터 단위: 패킷(Packet)

- 3계층은 전송 계층의 세그먼트를 **IP 헤더를 붙여서 "IP 패킷"**으로 만든다.
- 이 IP 패킷은 MAC 계층에서 프레임으로 감싸져 전송된다.

🧩 주요 구성요소

---

1. IP 주소 (IPv4/IPv6)

유형	구조	비트 수	주소 예
IPv4	4 옥텟	32비트	`192.168.1.100`

유형	구조	비트 수	주소 예
IPv6	8 그룹	128비트	`2001:0db8::1`

CIDR: `192.168.1.1/24` → 24비트는 네트워크, 나머지 호스트

---

2. 라우팅 (Routing)

> 라우터는 목적지 IP를 보고 **자신의 라우팅 테이블**을 참고해 다음 홉을 결정함

라우팅 테이블 확인

```
ip route
```

```
default via 192.168.0.1 dev eth0
192.168.0.0/24 dev eth0 proto kernel src 192.168.0.23
```

---

3. TTL (Time To Live)

- IP 패킷은 **통과한 라우터마다 TTL이 1씩 감소**
- TTL = 0이 되면 폐기되고 **ICMP Time Exceeded** 발생

```
ping -t 1 google.com
traceroute google.com
```

---

4. ICMP (Internet Control Message Protocol)

- IP 계층의 **제어/오류 메시지용 프로토콜**
- **Ping, Traceroute**의 기반
- 에코 요청/응답, 목적지 없음, TTL 초과 등

ping 사용 예

```
ping 8.8.8.8
```

C 코드에서 사용

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

---

5. 패킷 분할(Fragmentation) / 재조립(Reassembly)

- IPv4는 MTU(보통 1500바이트)를 넘는 패킷을 **라우터 또는 송신자**가 조각내어 전송
- IPv6는 송신자만 분할 가능, 라우터는 하지 않음

분할 확인

```
ping -s 1472 -M do 8.8.8.8  # DF 비트 설정
```

🧠 네트워크 계층 실전에서의 위치 (캡슐화 흐름)

```
[응용 데이터]
   ↓
[전송 계층: TCP 헤더]
   ↓
[네트워크 계층: IP 헤더 추가 → 패킷 완성]
   ↓
[데이터링크 계층: MAC 헤더 추가 → 프레임]
   ↓
[물리 계층: 신호 전송]
```

⚙️ 리눅스/C 프로그래머 관점

항목	방법
IP 확인	`ip addr show`
라우팅 확인	`ip route`
TTL 조정	`setsockopt()` + `IP_TTL`
ICMP 전송	RAW 소켓 + `IPPROTO_ICMP`
분할 여부 실험	`ping -M do -s SIZE`
주소 해석	`getaddrinfo()`, `inet_pton()` 등 사용

🔍 요약표

개념	설명
IP 주소	논리 주소 체계, 장치 식별자
패킷	IP 계층의 데이터 단위
TTL	패킷 생존 시간
라우팅	목적지에 따라 경로 결정

개념	설명
ICMP	제어/에러 메시지용
분할/재조립	큰 데이터를 작은 조각으로 쪼개 전송

- # IP 주소, 라우팅, ICMP, NAT

📍 1. IP 주소 (Internet Protocol Address)

✅ 개념

- **논리 주소**로, **호스트를 식별하고** 서로 통신할 수 있도록 지정
- **MAC 주소는 물리적인 위치**, **IP 주소는 논리적인 위치**
- IPv4 (32비트) / IPv6 (128비트)

🧮 IPv4 구조

```
192.168.001.045
= 8비트 x 4 옥텟 = 총 32비트
```

- 1. 네트워크 주소 (IP + 서브넷 마스크)
- 1. 브로드캐스트 주소
- 1. 게이트웨이 주소

✅ CIDR 표기법

- `192.168.0.1/24` → 네트워크 마스크 `255.255.255.0`
- `2^(32-24) = 256`개의 주소 사용 가능

🛠️ 리눅스에서 확인

```
ip addr show dev eth0
```

```
inet 192.168.0.23/24 brd 192.168.0.255 scope global eth0
```

📍 2. 라우팅 (Routing)

✅ 개념

- **목적지 IP에 따라 다음 경로를 선택하는 기능**
- **라우터/호스트**가 목적지에 도달하기 위해 **게이트웨이**를 선택

✅ 라우팅 테이블 구성 요소

항목	예시	의미
Destination	`0.0.0.0/0`	기본 경로
Gateway	`192.168.0.1`	다음 홉
Interface	`eth0`	사용 인터페이스
Metric	`100`	우선 순위 (낮을수록 우선)

🛠️ 확인 명령어

```
ip route
```

```
default via 192.168.0.1 dev eth0
192.168.0.0/24 dev eth0 proto kernel scope link src 192.168.0.23
```

→ **0.0.0.0/0**은 "기본 라우팅 경로" (인터넷 방향)

📍 3. ICMP (Internet Control Message Protocol)

✅ 개념

- **IP 계층의 부속 프로토콜**
- **에러 메시지, 진단 메시지** 전송
- Ping, Traceroute, TTL 초과, 목적지 도달 불가 등에 사용

✅ 주요 메시지 타입

Type	코드	설명
0	0	Echo Reply (ping 응답)
8	0	Echo Request (ping 요청)
3	x	Destination Unreachable
11	x	Time Exceeded (Traceroute 중간 노드 응답)

🧪 리눅스에서 Ping 사용

```
ping 8.8.8.8
```

→ 내부적으로는 **ICMP Echo Request/Reply** 전송

🔧 C 코드로 Ping (ICMP)

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

> 일반 사용자 권한에서는 불가 → `sudo` 필요

📍 4. NAT (Network Address Translation)

✅ 개념

- **사설 IP ↔ 공인 IP** 변환
- **라우터에서 수행, IP 충돌 없이 여러 장치를 인터넷에 연결**

🧱 주요 형태

유형	설명
Static NAT	내부 IP ↔ 외부 IP 1:1
Dynamic NAT	풀(pool)에서 외부 IP 동적 할당
PAT (Port Address Translation)	가장 흔함. 내부 포트를 외부 포트에 매핑 (ex: `192.168.0.2:5001 → 203.0.113.2:40001`)

🧠 NAT의 흐름

```
[PC1] 192.168.0.2 → [NAT 라우터] → 203.0.113.2
     (src 변경)              (인터넷으로 전송)
```

- 외부 응답이 오면 NAT 테이블을 참조해 다시 내부로 전달

📌 요약 정리

항목	설명	관련 명령어/코드
IP 주소	논리적 주소, 계층 간 식별	`ip addr`
라우팅	목적지 선택	`ip route`, `traceroute`

항목	설명	관련 명령어/코드
ICMP	제어/진단 메시지	`ping`, `traceroute`, RAW 소켓
NAT	사설 ↔ 공인 IP 변환	`iptables`, 라우터 내부

🧠 프로그래머가 기억해야 할 것

- IP는 논리 주소 → MAC과 다름
- 목적지 MAC은 ARP로 결정, IP는 패킷 전송용
- ICMP는 네트워크 상태 진단에 필수
- NAT 환경에서는 서버로 외부 접속 시 **포트포워딩**이 필요함

• IPv4/IPv6, TTL, 패킷 분할/재조립

🧩 1. IPv4 구조

✅ IPv4 헤더 구조

```
+--------+--------+----------+--------+--------+
| Version|  IHL   | DSCP/ECN | Total| Length   |
+--------+--------+----------+--------+--------+
| Identification  |
+--------+--------+--------+
| Flags  |Fragment| Offset |
+--------+--------+--------+--------+
| TTL    |Protocol| Header |Checksum|
+--------+--------+--------+--------+
| Source IP Address        |
+--------+-----------------+
| Destination IP Address   |
+--------+--------+--------+
| Options (optional)       |
+--------+--------+--------+
```

✅ 주요 필드 설명

필드	의미
Version	IPv4는 항상 4
IHL (Header Length)	IP 헤더 길이 (기본값: 20바이트)
Total Length	전체 패킷 길이 (헤더 + 데이터)
TTL	패킷의 생존 시간
Protocol	상위 계층 프로토콜 (TCP=6, UDP=17, ICMP=1)
Source / Destination IP	발신자/수신자 IP 주소

필드	의미
Flags + Fragment Offset	패킷 분할/재조립용 정보

🧪 2. IPv6 구조

✅ IPv6 헤더 구조 (고정 40바이트)

```
+-------------------------------------------------+
| Version (4) | Traffic Class (8) | Flow Label (20) |
+-------------------------------------------------+
| Payload Length (16) | Next Header (8) | Hop Limit (8) |
+-------------------------------------------------+
| Source IP Address (128 bits)                    |
+-------------------------------------------------+
| Destination IP Address (128 bits)               |
+-------------------------------------------------+
```

✅ IPv6 특징 요약

항목	IPv4	IPv6
주소 길이	32비트	128비트
NAT	필요	불필요
브로드캐스트	있음	없음 (멀티캐스트 대체)
옵션 처리	가변 길이 옵션	확장 헤더로 분리
헤더 크기	가변 (20~60B)	고정 (40B)
재조립	라우터 가능	수신자만 가능 (라우터는 하지 않음)

🧭 3. TTL (Time To Live)

✅ 정의

- 패킷이 **몇 개의 라우터를 거칠 수 있는지** 제한하는 필드
- 초기값: 일반적으로 64, 128, 255 등
- **매 홉(hop)마다 1씩 감소**, 0이 되면 폐기됨

> ICMP 타입 11(Time Exceeded)이 이때 발생

🧪 실습: traceroute

```
traceroute google.com
```

- 내부적으로 **TTL = 1 → 2 → 3...** 순차 증가
- 경유지에서 TTL 초과 메시지를 회신함 → 경로 확인 가능

📦 4. 패킷 분할 (Fragmentation) & 재조립 (Reassembly)

✅ 왜 필요한가?

- Ethernet MTU: 1500바이트
- TCP/IP 헤더 포함 시 **데이터 크기 제한** 발생
- 큰 IP 패킷은 MTU보다 작게 **조각(Fragment)** 처리

✅ IPv4의 분할 구조

필드	설명
Identification	조각들의 그룹 식별
Flags	DF (Don't Fragment), MF (More Fragment)
Fragment Offset	조각의 순서 결정 (8바이트 단위)

🧬 조각 처리 흐름

```
패킷 A (4000 bytes) →
  조각1: offset 0, length 1480, MF=1
  조각2: offset 1480, length 1480, MF=1
  조각3: offset 2960, length 1040, MF=0
```

→ 수신 측은 **Identification + Offset + MF** 조합으로 재조립

❌ IPv6에서는?

- **라우터는 Fragmentation을 하지 않음**
- **송신자 측**에서 MTU 고려하여 분할
- 분할 시 **확장 헤더(Fragment Extension Header)** 사용

🛠️ MTU 확인 및 설정 (리눅스)

```
ip link show eth0
```

```
ip link set dev eth0 mtu 1400
```

🧪 ping -M do 테스트

```
ping -c 1 -M do -s 1472 8.8.8.8
```

- `-M do`: Don't Fragment 설정
- `-s 1472` + 28바이트 IP+ICMP 헤더 = 1500

📌 정리 요약표

항목	핵심 내용
IPv4	32비트 주소, 가변 헤더, NAT 사용
IPv6	128비트 주소, 고정 헤더, 확장성 향상
TTL	홉 제한 (라우팅 루프 방지)
분할/재조립	IPv4는 라우터도 분할, IPv6는 발신자만

🧠 리눅스 C 프로그래머 관점 요약

기능	접근 방식
IP 헤더 직접 보기	`RAW socket` (`SOCK_RAW`, `IPPROTO_RAW`)
TTL 설정	`setsockopt()`로 `IP_TTL` 조절
분할 여부 확인	Wireshark, `ip a`, `ping -M do`
IPv6 지원 여부 확인	`getaddrinfo()`, `sockaddr_in6` 사용

1.6 OSI 4계층: 전송 계층 (Transport Layer)

📌 개요

전송 계층(Transport Layer)은 OSI 7계층 중 네 번째 계층으로, **종단 간(end-to-end)의 신뢰성 있는 데이터 전송을 보장하거나, 반대로 빠른 전송만을 제공하는 기능을 수행**한다.
이 계층은 호스트 간 통신이 아닌, **프로세스 간 통신**을 중재한다는 점에서 하위 계층들과 명확히 구분된다.

🎯 주요 기능

기능	설명
Multiplexing/Demultiplexing	여러 응용 프로그램 간에 포트 번호로 구분하여 데이터를 전달하고 수신함
신뢰성 보장 (TCP)	데이터 손실, 순서 어긋남, 중복 등을 방지하고 보정
흐름 제어 (Flow Control)	송신자가 수신자의 처리 능력을 초과하지 않도록 조절
혼잡 제어 (Congestion Control)	네트워크 혼잡 발생 시 송신 속도를 조절
에러 제어	손상되거나 누락된 세그먼트에 대한 재전송 수행 (TCP 한정)

🧱 전송 계층의 대표 프로토콜

1. TCP (Transmission Control Protocol)

항목	설명
연결형(Connection-oriented)	통신을 시작하기 전에 세션을 설정함 (3-way Handshake)
신뢰성 보장	수신 확인(ACK), 재전송(Retransmission), 순서 제어(Ordering) 제공
흐름 제어	수신자의 버퍼 상태를 고려한 윈도우(Window) 기반 제어
혼잡 제어	AIMD, Slow Start, Fast Retransmit, Fast Recovery 등의 알고리즘 사용
용도 예시	HTTP/HTTPS, FTP, SMTP, SSH 등

2. UDP (User Datagram Protocol)

항목	설명
비연결형(Connectionless)	핸드셰이크 없이 전송 수행
신뢰성 없음	순서 보장, 오류 수정 없음 (단순 에러 체크 있음)
빠름	헤더가 작고, 처리 로직이 단순하여 빠름
용도 예시	DNS, DHCP, TFTP, VoIP, 게임 통신 등

🔄 TCP 연결: 3-Way Handshake

1. **SYN**: 클라이언트 → 서버, 연결 요청
2. **SYN-ACK**: 서버 → 클라이언트, 수락 + 응답
3. **ACK**: 클라이언트 → 서버, 연결 완료

이 과정을 통해 송·수신 양측은 서로의 초기 시퀀스 번호(ISN)를 공유하고 신뢰성 있는 세션을 구축한다.

🔚 연결 종료: 4-Way Handshake

1. **FIN**: 종료 요청
2. **ACK**: 수신 확인
3. **FIN**: 반대 방향 종료 요청
4. **ACK**: 최종 확인

🔄 흐름 제어 (Flow Control)

⏳ Sliding Window 방식

- 송신자는 수신자의 **수신 윈도우 크기(Window Size)** 를 고려하여 데이터를 전송
- 수신자는 `ACK` 패킷을 통해 윈도우 크기를 조정

효과

- 수신자의 버퍼 초과 방지
- 네트워크 자원 낭비 최소화

📉 혼잡 제어 (Congestion Control)

TCP는 네트워크 혼잡 시 패킷 손실을 방지하기 위해 송신 속도를 조절한다.

알고리즘	설명
Slow Start	처음에는 느리게 시작하며, 수신 확인마다 전송 윈도우를 2배로 증가
Congestion Avoidance	일정 임계점 이후 선형 증가
Fast Retransmit	중복 ACK 3개 수신 시, 재전송 수행
Fast Recovery	손실 발생 후 느리게 감소, 빠르게 회복

🔢 포트 번호 (Port Number)

전송 계층은 IP 주소와 함께 **포트 번호(port)** 를 이용하여 애플리케이션을 식별한다.

범위	의미
0 ~ 1023	Well-Known Ports (HTTP 80, HTTPS 443 등)
1024 ~ 49151	Registered Ports
49152 ~ 65535	Dynamic/Private Ports (Ephemeral Ports)

📎 세그먼트 구조

TCP 헤더 구조 (요약)

필드	설명
Source Port, Dest Port	송/수신 포트 번호
Sequence Number	데이터 순서 추적
Acknowledgment Number	받은 데이터의 다음 기대 번호
Flags	SYN, ACK, FIN, RST 등 제어 비트
Window Size	흐름 제어용 윈도우 크기
Checksum	오류 검출
Options	MSS, Timestamp 등

🧪 전송 계층의 디버깅 포인트

도구	설명
`netstat -anp`	포트 상태 및 연결 추적
`ss -tulnp`	TCP/UDP 소켓 상태 확인
`tcpdump`, `Wireshark`	핸드셰이크, 재전송, 혼잡제어 분석 가능
`strace`	`send`, `recv`, `connect`, `accept` syscall 추적

✅ 요약 정리

항목	TCP	UDP
연결 방식	연결형	비연결형
신뢰성	높음	낮음
순서 보장	O	X
오류 보정	O	X
속도	느림	빠름
대표 프로토콜	HTTP, FTP	DNS, VoIP

- # TCP/UDP, 포트 번호, 흐름 제어, 재전송

✅ 1. TCP / UDP 비교

📌 TCP (Transmission Control Protocol)

- **연결 지향형(Connected-Oriented)** 프로토콜
- 통신 시작 전에 **3-Way Handshake** 수행
- 신뢰성 있는 데이터 전송 보장:
  - 데이터 순서 보장
  - 손실 발생 시 자동 재전송
  - 흐름 제어 및 혼잡 제어 수행
- 애플리케이션에서 **전송이 안전해야 하는 경우** 사용됨

TCP 주요 사용 예:

- 웹 통신 (HTTP/HTTPS)
- 이메일 전송 (SMTP)
- 파일 전송 (FTP)
- 원격 접속 (SSH)

📌 UDP (User Datagram Protocol)

- **비연결형(Connectionless)** 프로토콜
- 핸드셰이크 없음, 즉시 전송
- 신뢰성 없음 (패킷 손실, 순서 변경 가능)
- 헤더 크기 작고, 처리 속도 빠름
- 애플리케이션에서 **속도가 중요하고 일부 손실을 허용할 수 있는 경우** 사용됨

UDP 주요 사용 예:

- 도메인 질의 (DNS)
- 실시간 스트리밍 (VoIP, IPTV)
- 온라인 게임

항목	TCP	UDP
연결 방식	연결형	비연결형
신뢰성	O (순서 + 재전송)	X
흐름 제어	O	X
혼잡 제어	O	X
속도	느림	빠름

항목	TCP	UDP
헤더 크기	최소 20바이트	8바이트
대표 용도	HTTP, FTP, SSH	DNS, RTP, 게임

✅ 2. 포트 번호 (Port Number)

📌 개념

- 전송 계층에서 사용하는 **논리적 식별자**
- 하나의 IP 주소에 여러 개의 애플리케이션을 구분하여 바인딩하기 위해 사용됨
- 포트 번호는 16비트 정수 (0 ~ 65535)

📌 분류

범위	구분	설명
0 ~ 1023	Well-Known Ports	특정 서비스 예약 (예: HTTP 80, HTTPS 443)
1024 ~ 49151	Registered Ports	특정 기업, 사용자 등록 포트
49152 ~ 65535	Dynamic/Ephemeral Ports	클라이언트 측 임시 포트

📌 예시

프로토콜	포트 번호	설명
HTTP	80	웹 서버 기본 포트
HTTPS	443	보안 웹 통신
SSH	22	원격 로그인
DNS	53	도메인 이름 질의
FTP	20 (Data), 21 (Control)	파일 전송 프로토콜
SMTP	25	이메일 전송
DHCP	67 (서버), 68 (클라이언트)	IP 자동 할당 프로토콜

✅ 3. 흐름 제어 (Flow Control)

📌 개념

- **송신 측**이 수신 측의 **처리 능력을 초과하지 않도록** 데이터를 전송 속도를 조절하는 메커니즘
- 수신자의 버퍼가 가득 차면 데이터를 더 받지 않도록 신호를 보냄

📌 TCP의 흐름 제어 방식: Sliding Window

- 수신자는 매 ACK 패킷에 **수신 가능 윈도우 크기(window size)** 를 포함시켜 송신자에게 전달
- 송신자는 이 윈도우 내에서만 데이터를 전송

예시

1. 수신자: "윈도우 크기 4096바이트"
2. 송신자는 최대 4096바이트까지 전송
3. 수신자가 처리하면 윈도우 크기 다시 증가 → 전송 재개

📌 작동 목적

- 네트워크 트래픽 제어가 아니라 **엔드포인트 사이의 처리량** 조절
- 네트워크 혼잡과는 무관 (혼잡 제어와는 별개임)

✅ 4. 재전송 (Retransmission)

📌 개념

- 전송된 데이터가 손실되거나 손상된 경우, **송신 측이 다시 데이터를 전송**하는 기능
- 주로 **TCP에서만 제공**되는 기능

📌 TCP 재전송 트리거

조건	설명
ACK 타임아웃	일정 시간 내에 ACK가 도착하지 않으면 재전송
중복 ACK 3회 수신	동일한 ACK 번호가 3번 연속 도착하면 손실로 판단 (Fast Retransmit)

📌 재전송 방식 예

📍 Timeout 기반

```
setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, ...)
```

- 수신 대기시간이 초과되면 재전송 시도

📍 Fast Retransmit

- 순서대로 받은 ACK이 아니라, **중복된 ACK**가 여러 번 도착 → 네트워크 혼잡 아님 → 재전송 즉시 실행

🧪 C 코드 예시 스니펫: 흐름 제어 및 재전송 관찰용

```c
// TCP socket 생성 및 send/recv 동작 관찰
int sock = socket(AF_INET, SOCK_STREAM, 0);
connect(sock, ...);

// 의도적으로 작은 버퍼 전송
char buffer[1024];
send(sock, buffer, sizeof(buffer), 0);

// Wireshark로 "ACK", "Window Update", "Retransmission" 필터 적용 관찰
```

📝 결론 요약

- **TCP는 신뢰성과 흐름 제어, 재전송을 제공**하며 연결형 프로토콜임
- **UDP는 단순하고 빠르지만, 신뢰성 없음**
- **포트 번호는 애플리케이션 간 논리적 통신을 위한 주소 체계**
- **흐름 제어는 수신자 보호**, **재전송은 신뢰성 보장**을 위한 핵심 기술

• 3-way handshake, 혼잡 제어

✅ 1. TCP 3-Way Handshake

📌 개념

TCP는 신뢰성 있는 연결을 제공하기 위해 **세션을 맺기 전 3단계의 핸드셰이크 절차**를 수행한다.
이 과정을 통해 **양측의 통신 가능성 확인**, **초기 순서 번호(ISN, Initial Sequence Number)** 설정이 이루어진다.

🧭 동작 흐름

단계	송신자(Client)	수신자(Server)
① SYN	연결 요청 (SYN=1, ISN=x)	대기 상태 (LISTEN)
② SYN+ACK	수신자 수락 + 응답 (SYN=1, ACK=1, ISN=y, ACK=x+1)	연결 설정
③ ACK	수신자의 응답 확인 (ACK=1, ACK=y+1)	연결 완료

📊 상태 전이

클라이언트 상태	서버 상태
CLOSED → SYN_SENT → ESTABLISHED	LISTEN → SYN_RCVD → ESTABLISHED

🔐 목적

- **전이중(Full-Duplex)** 통신 준비
- **시퀀스 넘버(ISN)** 동기화
- **패킷 손실 가능성 점검** (재전송 가능)

🖼️ 시각적 요약

```
Client                          Server
  | -------- SYN (SEQ=x) --------> |
  | <---- SYN + ACK (SEQ=y) ------ |
  | --------- ACK (ACK=y+1) -----> |
```

✅ 2. TCP 혼잡 제어 (Congestion Control)

📌 개념

혼잡 제어는 네트워크의 혼잡 상태를 감지하고, 트래픽 과부하를 줄이기 위해 송신 속도를 동적으로 조절하는 알고리즘이다.

TCP는 기본적으로 "네트워크 상태를 정확히 알 수 없기 때문에, 간접적 신호(패킷 손실, 중복 ACK 등)를 활용하여 혼잡을 감지하고 대응"한다.

🚦 주요 혼잡 제어 알고리즘

1. Slow Start

- 초기에 `Congestion Window(cwnd)`를 매우 작게 설정
- 매 ACK 수신마다 `cwnd` **2배로 증가** (지수 증가)
- **임계점(ssthresh)** 도달 시 Congestion Avoidance로 전환

2. Congestion Avoidance

- 혼잡 위험이 감지되면 `cwnd`를 천천히 **선형 증가**
  - `cwnd += MSS * (MSS / cwnd)` (AIMD: Additive Increase)

3. Fast Retransmit

- **중복 ACK**가 3개 수신되면, 손실로 간주하고 즉시 재전송

4. Fast Recovery

- 손실 발생 시 `ssthresh = cwnd / 2`
- `cwnd = ssthresh + 3 * MSS`로 설정
- ACK 도착 시 다시 증가

📈 혼잡 제어 흐름 예시

```
Time →
cwnd
 ↑
 |          /
 |         /
 |        /
 |       /
 |      /
 |     /
 |    /    <- Slow Start (exponential)
 |--------- <- ssthresh (threshold)
 |        \     <- Congestion Avoidance (linear)
 |         \___ <- Packet loss detected → cwnd drop → Fast Recovery
```

🔧 혼잡 제어 관련 변수 요약

변수	설명
cwnd	Congestion Window (혼잡 제어를 위한 송신 윈도우 크기)
ssthresh	Slow Start Threshold (임계점, 이 지점을 넘으면 선형 증가)
MSS	Maximum Segment Size (세그먼트 당 최대 데이터 크기)
RTT	Round Trip Time (왕복 시간) – 재전송 타이머에 영향

🧪 실제 분석 도구

도구	혼잡 제어 관찰 항목
Wireshark	Duplicate ACK, TCP Retransmission, Window Full 등 필터 가능
tcpdump	RTT 및 재전송 이벤트 추적
netstat -s	TCP 통계 항목: 재전송 횟수, 혼잡 상태 등
ss -i	각 TCP 소켓의 `cwnd`, `rto`, `rtt` 등의 상세 정보 확인 가능

✅ 혼잡 제어 요약

알고리즘	특징
Slow Start	빠른 시작, 지수 증가
Congestion Avoidance	천천히 증가, 혼잡 예방

알고리즘	특징
Fast Retransmit	중복 ACK로 손실 감지, 즉시 재전송
Fast Recovery	선형 회복, 임계치 재설정

1.7 OSI 5계층: 세션 계층 (Session Layer)

✅ 개요

세션 계층(Session Layer)은 OSI 7계층 중 **5번째 계층**으로,
통신 세션(Session)을 **설정, 유지, 종료**하는 역할을 한다.
즉, 단순히 데이터를 보내는 것뿐 아니라 **대화의 논리적 문맥(context)**을 관리한다.

> 📌 통신 세션이란?
> 한 클라이언트와 서버 또는 두 호스트 간의 **연결 유지 기간 전체를 의미**하며, 여러 데이터 전송 동작이 포함됨.

🎯 핵심 기능

기능	설명
세션 설정 (Establishment)	통신 시작 시 서로의 상태 동기화
세션 유지 (Maintenance)	데이터 흐름 중 동기화 지점 삽입, 오류 발생 시 복원 가능
세션 종료 (Termination)	전송이 끝난 후 자원 정리
동기화 (Synchronization)	중간 복구를 위한 체크포인트 설정
토큰 관리 (Token Management)	다중 접속 환경에서 제어권을 한 쪽에 할당
중복 제어	동일 세션 내에서의 요청 반복 방지 (RPC 환경 등)

🧱 세션 계층의 실제 적용 예시

1. TLS/SSL Handshake 과정

- **SSL/TLS Handshake**는 세션 계층 수준에서 **보안 연결 세션**을 설정하는 예시
- 인증서 교환, 키 교환, 대칭키 협의, 암호화 방식 합의 등 수행
- 세션 재활용(Session Resumption) 기능도 이 계층의 연장선

2. 원격 프로시저 호출 (RPC)

- 클라이언트가 원격 서버에 함수 호출을 요청할 때, **세션 식별자**를 사용하여 각 호출을 구분
- 예: gRPC, DCOM, Java RMI

3. NetBIOS Sessions

- SMB 프로토콜에서 **NetBIOS 세션 서비스**는 세션 계층에서 동작
- 연결 설정 / 데이터 전송 / 연결 종료 프레임을 사용

🔐 TLS와 세션 계층의 관계

항목	설명
TLS Record Layer	전송 계층 역할 (데이터 암호화 단위)
TLS Handshake Layer	세션 계층 기능: 키 교환, 인증, 세션 설정
세션 식별자(Session ID)	클라이언트와 서버가 동일 암호키로 복구 가능한 세션 식별자 생성

🧭 OSI 상의 위치적 의미

세션 계층은 아래 전송 계층(TCP/UDP)과 위의 표현 계층 사이에 존재하며,
전송 계층이 데이터 전송을 보장하는 동안, **세션 계층은 논리적 대화를 관리**한다.

```
Application Layer       ┐
Presentation Layer      | 사용자 요청 처리
Session Layer           ┘ 논리적 연결 관리 ← 이 계층
Transport Layer         ┐ 전송 보장 (TCP/UDP)
Network Layer           | 라우팅
Data Link Layer         | MAC 주소 기반 전송
Physical Layer          ┘ 실제 신호 전달
```

🔍 세션 계층이 중요한 이유

관점	설명
장시간 연결 보장	상태 기반 통신에서 연결 상태를 유지하기 위함 (예: 스트리밍, 채팅)
보안 세션 구성	TLS 세션, SSH 세션 등
문맥(Context) 유지	REST와 다르게 상태 기반 요청 처리를 위한 세션 유지 필요
분산 시스템	멀티 노드 환경에서 RPC, 세션 클러스터링 등이 필요함

📜 관련 프로토콜 (실제 또는 논리적으로 세션 계층에서 동작)

프로토콜	설명
TLS/SSL	보안 세션 설정 및 재개
NetBIOS Session Service	SMB 통신에서 사용

프로토콜	설명
RPC (Remote Procedure Call)	클라이언트 ↔ 서버 간 함수 호출 기반 세션
RTSP (Real Time Streaming Protocol)	스트리밍 제어 세션 관리
SIP (Session Initiation Protocol)	VoIP에서 세션 생성, 수정, 해제
X.225 / ISO 8327	OSI에서 정의한 공식 세션 계층 프로토콜 (실제 사용 드묾)

🚫 주의할 오해

- ❌ **TCP는 세션 계층이 아니다.**
  TCP는 전송 계층이며, 세션은 TCP 위에서 **논리적으로 구현됨**
- ❌ **세션 계층은 항상 별도의 프로토콜로 구현되지 않는다.**
  많은 애플리케이션이 TCP 위에서 세션 논리를 직접 구현 (예: 로그인 상태 유지)

✅ 요약 정리

항목	설명
위치	OSI 5계층
역할	통신 세션의 설정, 유지, 종료
특징	동기화, 토큰 관리, 세션 복구
대표 기술	TLS Handshake, NetBIOS, RPC, SIP
오해 방지	TCP는 전송 계층, 세션 관리는 그 위 계층의 책임

- # 세션 연결, 동기화, TLS Handshake

  ## ✅ 1. 세션 연결 (Session Establishment)

  ### 📌 개요

  세션 연결은 **양쪽 애플리케이션이 통신을 시작하기 전**,
  논리적으로 연결이 성립되었음을 확인하고 **세션 상태를 생성**하는 과정이다.
  TCP의 3-way handshake 이후에도, **애플리케이션 레벨에서 또 한 번의 논리적 연결 합의**가 필요한 경우가 많다.

  ### 📊 예시: RPC 기반 시스템

  1. 클라이언트가 서버에 `Session-Init` 메시지를 보냄
  2. 서버가 `Session-Ack`로 응답
  3. 세션 ID가 부여되고 세션 테이블에 등록됨

```
Client                        Server
  | ---- Session Init -----> |
  | <---- Session Ack ------ |
  | ===> 세션 ID 할당 완료     |
```

📁 세션 관리 항목

항목	설명
Session ID	세션 식별자 (쿠키, 토큰, UUID 등)
Timeout	비활성 상태로 세션 종료되는 시간
State	연결 상태: Established, Suspended, Closed 등
Context	사용자 인증, 위치, 권한 등 상태 정보 포함 가능

✅ 2. 동기화 (Synchronization)

📌 개요

세션 도중, 통신이 중단되거나 중간에 복구가 필요할 수 있음.
이때 필요한 것이 **세션 동기화(Synchronization)**, 즉 **체크포인트 기반의 세션 상태 기록**이다.

📊 예시: 대용량 파일 전송 중 중단 발생

- 70%까지 전송 후 연결 끊김
- 세션 동기화 지점이 기록되어 있으면 70% 지점부터 재전송 가능

🧩 동기화 방식

방식	설명
Sequence Number 기반	TCP와 유사한 방식으로 각 메시지에 일련번호 부여
체크포인트 저장	주기적으로 서버에 현재 상태(예: 파일의 오프셋)를 저장
Ack 기반 로직	동기화 시점까지 수신된 마지막 응답에 대한 확인 메시지 전송

✅ 3. TLS Handshake (세션 계층 실전 적용)

📌 개요

TLS Handshake는 **보안 세션을 설정하기 위한 협상 과정**이며,
세션 계층 기능 중 가장 실전적으로 구현된 사례다.

> TLS는 실제로는 전송 계층(TCP) 위에서 동작하지만, 그 중 `Handshake Protocol` 은 세션 계층의 논리를 직접 수행한다.

🔐 TLS Handshake 주요 목적

1. **서버/클라이언트 인증**
2. **암호화 알고리즘 협상**
3. **세션 키 교환**
4. **세션 재개 여부 확인**

📊 전체 흐름

```
Client                                  Server
  | -------- ClientHello -------->       | ① 지원 암호 알고리즘 목록 전송
  | <------- ServerHello ---------       | ② 암호화 방식 선택
  | <------ Certificate -----------      | ③ 서버 인증서 전송 (X.509)
  | <-- ServerKeyExchange (optional)     | ④ Ephemeral 키 제공 (DH 등)
  | <-- ServerHelloDone ---------------- |
  | -------- ClientKeyExchange -------->  | ⑤ Pre-Master Key 전송
  | -------- ChangeCipherSpec -------->   | ⑥ 암호화 시작 알림
  | -------- Finished ------------->      | ⑦ MAC 포함 최종 확인
  | <------- ChangeCipherSpec ---------- |
  | <------- Finished ------------------- |
  | ===> 세션 설정 완료, 암호화 통신 시작     |
```

🔐 TLS 세션의 결과물

항목	내용
세션 ID / Session Ticket	암호 키 세트의 재사용 가능 여부
Cipher Suite	사용된 암호화/해시/키 교환 알고리즘 세트
Master Secret	공유된 대칭키 파생의 기준
서버 인증 여부	유효한 인증서 검사 결과

📌 TLS Handshake이 세션 계층인 이유

기능	TLS Handshake에서 수행 방식
세션 설정	ClientHello ~ Finished 교환
동기화	MAC(Msg Auth Code)을 통한 상태 검증
복구	세션 재개 (Session Resumption) 메커니즘
인증	서버 또는 클라이언트 인증서 확인

✅ 정리 요약

항목	설명
세션 연결	양측 통신 논리 연결 수립 및 세션 ID 발급
세션 동기화	오류나 중단에 대비한 복구 지점 설정
TLS Handshake	인증, 키 협상, 세션 설정 과정을 통합한 보안 세션 수립 절차
세션 계층의 특징	TCP 위에 존재하며, 애플리케이션 간의 상태 기반 대화를 유지

- # 연결 지향 세션 유지 기술

✅ 개요

연결 지향 세션 유지란, 클라이언트와 서버 또는 두 노드 간의 **논리적 연결 상태(session context)** 를 **지속적으로 유지**하며,
통신이 끊기지 않고 안정적으로 이루어지도록 보장하는 기술 집합을 말한다.

이는 단순 TCP 연결 유지뿐 아니라, **애플리케이션 레벨에서 세션 식별, 상태 저장, 시간 초과 관리, 실패 복구 등을 포함**한다.

🎯 세션 유지의 필요성

상황	이유
로그인 상태 유지	사용자의 인증 상태 지속
장시간 작업 (파일 업로드, 스트리밍)	중단 없이 전송 보장
장기 커넥션 (WebSocket, VPN)	지속적인 상호작용 또는 터널링
분산 시스템 세션 클러스터링	장애 발생 시에도 세션 정보 복구 가능

🧱 핵심 구성 요소

구성 요소	설명
Session ID	각 사용자/연결마다 고유한 식별자 (쿠키, 토큰 등)
Session Table	서버 측에 저장된 세션 상태 정보 테이블
Timeout 관리	일정 시간 활동이 없으면 세션 종료
Heartbeat/Ping	주기적으로 상태 확인 및 연결 유지
Keep-Alive 옵션	TCP 수준에서의 연결 유지 지원
Session Resumption	재연결 시 기존 세션 복원 (TLS, WebSocket 등)

📌 구현 기술 및 기법

1. TCP Keep-Alive

- **OS 레벨에서 TCP 연결 상태를 주기적으로 확인**
- 일정 시간동안 응답이 없으면 연결 종료
- `setsockopt()` 를 통해 `SO_KEEPALIVE` 옵션 설정 가능

```
int optval = 1;
setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof(optval));
```

---

2. Application-level Heartbeat (Ping/Pong)

- 애플리케이션이 **직접 일정 주기로 신호 전송** (예: `"ping"` → `"pong"`)
- WebSocket, MQTT 등 실시간 연결에서 많이 사용

```
Client:   PING
Server:   PONG
(매 30초 주기)
```

---

3. 세션 타임아웃 및 재설정

- 서버는 세션 생성 시 **마지막 활동 시각**을 저장
- 사용자가 재요청할 때마다 갱신
- 일정 시간 동안 무응답이면 세션 제거 (ex: `session.timeout = 30min`)

---

4. 세션 클러스터링 / 상태 저장소 사용

- 서버 간 로드밸런싱 또는 장애 복구 대비
- **Redis, Memcached** 등을 이용한 세션 공유
- 또는 **JWT (JSON Web Token)** 방식으로 클라이언트가 상태 보유

방법	특징
상태 저장 서버 메모리	빠르지만 단일 장애점 발생 가능
외부 공유 저장소	고가용성 가능, 약간 느림
JWT	서버 상태 없이 토큰으로 인증 상태 유지

---

5. TLS Session Resumption

- 클라이언트가 `Session ID` 또는 `Session Ticket`을 재전송하여
  암호화 세션을 새로 협상하지 않고 복원

방식

방식	설명
Session ID	서버가 발급한 세션 ID를 다시 보내 연결 복원
Session Ticket	서버가 세션 정보를 암호화한 티켓을 클라이언트가 보관

🧭 실전 적용 예시

📡 WebSocket

- 지속적 양방향 통신
- **ping/pong frame**으로 연결 상태 감시
- 연결 끊김 감지 시 자동 재접속 로직 구현

🔐 TLS (HTTPS)

- 장시간 연결 유지 시 **TLS 세션 재사용** 지원
- 성능 최적화: 핸드쉐이크 생략 → 지연 최소화

🧳 SSH

- ControlMaster 옵션을 통해 여러 연결을 하나의 세션으로 공유
- 타임아웃이 발생하지 않도록 서버/클라이언트 모두 KeepAlive 설정

📝 요약 정리

기술	계층	주요 목적
TCP Keep-Alive	전송 계층	연결 상태 유지
Application Heartbeat	세션 계층	상태 감시 및 응답 확인
세션 타임아웃 관리	애플리케이션 계층	리소스 해제 및 보안
세션 클러스터링	분산 시스템	세션 일관성 유지
TLS Resumption	세션/전송 계층	보안 세션 재활용

1.8 OSI 6계층: 표현 계층 (Presentation Layer)

✅ 개요

표현 계층(Presentation Layer)은 OSI 7계층 중 **6번째 계층**으로,
전송된 데이터를 수신 측 애플리케이션이 이해할 수 있는 형식으로 변환하거나, 그 반대의 역할을 수행한다.
이 계층은 "**데이터의 구조와 표현 방식**"을 중재하는 브릿지 역할을 하며, 흔히 말하는 **구문(Syntax) 계층**이라고도 불린다.

🎯 주요 역할

역할	설명
데이터 인코딩/디코딩	애플리케이션에서 사용하는 형식을 통일
압축/해제	전송 효율을 높이기 위한 데이터 압축 처리
암호화/복호화	보안을 위한 데이터 보호 및 복원
형식 변환	서로 다른 시스템 간의 데이터 표현 차이를 해소 (예: UTF-8 ↔ UTF-16)

🔧 기능별 상세 설명

1. 데이터 포맷 변환

- 서로 다른 플랫폼/OS 간의 표현 방식 차이를 해결
- 예: Windows는 UTF-16, Unix는 UTF-8 사용
- Big Endian ↔ Little Endian 변환
- ASN.1, XML, JSON, YAML 등의 구문 파싱도 포함됨

```
Ex: JSON → 내부 객체로 변환 → 애플리케이션에 전달
```

---

2. 압축 (Compression)

- 전송 데이터를 압축하여 **네트워크 대역폭 절감**
- 수신 측에서 다시 해제
- 예시: Gzip, Brotli, DEFLATE

```
[Hello Hello Hello Hello] → [H(4)x]
```

실제 적용 예:

- HTTP 헤더의 `Content-Encoding: gzip`
- SSH 연결 시 압축 옵션 사용 가능

---

3. 암호화/복호화 (Encryption/Decryption)

- TLS/SSL, SSH 등에서 데이터를 **암호화하여 기밀성 보장**
- 표현 계층에서 **실제 암호화 알고리즘(AES, RSA 등)** 을 수행
- 전송 계층은 암호화 여부를 알지 못함

적용 위치

위치	내용
애플리케이션 내부	JWT 서명, 쿠키 암호화
표현 계층	TLS, SSH, S/MIME
전송 계층	TCP/UDP – 암호화 처리 없음 (단순 전송)

🧩 프로토콜 및 포맷 예시

기술/포맷	설명
MIME (Multipurpose Internet Mail Extensions)	이메일 내 데이터 형식 식별 (텍스트, 바이너리 등)
JSON, XML	데이터를 구조화된 문자열 형식으로 표현
ASN.1 (Abstract Syntax Notation One)	통신 장비 간 이진 표현 규약
TLS Record Layer	암호화된 데이터 블록을 구조화하여 전달
XDR, BER, DER	형식 표준화된 바이너리 표현 방식

🧪 실전 적용 예

📁 예1: 웹에서 JSON 송수신

```
{
  "name": "Alice",
  "age": 30
}
```

- 표현 계층: JSON 문자열 → 내부 객체로 디코딩
- 사용자에게는 객체로 표현되지만, 실제 네트워크에는 문자열 전송됨

🔐 예2: HTTPS 암호화 처리

```
[Plaintext] → TLS Handshake → [암호화된 Record Block] → TCP로 전송
```

- 표현 계층에서 암호화

- 전송 계층은 데이터 구조나 의미를 알지 못함

🎧 예3: 오디오/비디오 압축

- MP3, AAC, H.264 등은 모두 표현 계층에서 적용되는 **인코딩 규약**
- 실시간 스트리밍에서도 동일한 압축/해제 작업이 수행됨

📝 요약 정리

항목	설명
계층	OSI 6계층
주된 역할	데이터 구조, 포맷, 보안 처리
기능	인코딩/디코딩, 암호화/복호화, 압축/해제, 포맷 변환
관련 기술	TLS, JSON/XML, MIME, ASN.1, gzip
계층 구분	전송 계층과 애플리케이션 사이의 변환 계층

🚫 오해 방지

- ❌ **TLS는 전송 계층이 아니다.**
  → TLS는 전송 계층(TCP) 위에서 동작하지만, 그 중 암호화/복호화 처리는 표현 계층 기능에 해당
- ❌ **표현 계층은 별도의 네트워크 패킷을 만들지 않는다.**
  → 표현 계층은 데이터의 내용 자체를 다루며, TCP/UDP 계층처럼 헤더를 가지는 것은 아님

• 데이터 인코딩, 압축, 암호화

✅ 1. 데이터 인코딩 (Encoding)

📌 개념

데이터 인코딩은 문자나 숫자, 구조화된 정보를 **전송 가능한 형식(바이트 시퀀스)** 으로 변환하는 과정이다.
이는 수신 측이 데이터를 **정확히 해석하고 복원**할 수 있도록 하기 위한 작업이다.

🧩 주요 인코딩 방식

인코딩 방식	설명
문자 인코딩	UTF-8, UTF-16, ASCII 등
바이너리 인코딩	Base64, Hex (이진 데이터를 문자로 표현)
구조화 인코딩	JSON, XML, YAML, Protocol Buffers, MessagePack 등

🎯 예시 1: 문자열 인코딩

```
// UTF-8로 인코딩된 문자열 예
const char* utf8 = "안녕하세요";  // UTF-8 기준 15바이트
```

- 표현 계층에서는 이 바이트들을 네트워크 전송용 버퍼로 처리
- 수신 측은 같은 인코딩 스킴으로 복호화 필요

🎯 예시 2: 바이너리 → 텍스트 인코딩 (Base64)

```
Binary:   01001101 01100001 01101110
          (Man)
Base64:   TWFu
```

- 텍스트 기반 전송(JSON, XML) 시 이진 데이터 삽입용으로 사용됨
- 이메일 첨부파일, HTTP 헤더 등에 활용

✅ 2. 압축 (Compression)

📌 개념

압축은 데이터의 중복을 제거하거나 통계적 구조를 활용하여 데이터 크기를 줄이는 작업이다.
전송 전 데이터를 압축하고, 수신 후 복원하는 과정을 통해 **전송 시간과 대역폭을 절약**할 수 있다.

🧰 압축 알고리즘 종류

종류	알고리즘	특징
손실 압축	MP3, JPEG, H.264	정보 일부 제거 (미디어 전용)
무손실 압축	Gzip, Brotli, LZ77, DEFLATE	원본 데이터 완전 복원 가능

🎯 예시: Gzip 압축

```
# 서버에서 gzip 압축 활성화 예
Content-Encoding: gzip
```

HTTP 흐름 예

```
[JSON 데이터] → gzip 압축 → TCP 전송
↓
수신자: 압축 해제 후 JSON 디코딩
```

```
#include <zlib.h>  // C에서 Gzip 압축을 사용할 수 있는 라이브러리
```

📁 실전 사용 예

위치	설명
웹 서버	HTTP Response 압축 (Nginx, Apache, Spring 등)
SSH	압축 옵션으로 트래픽 감소
VPN	터널링 데이터 압축
프로토콜	MQTT v5에서 payload 압축 옵션 등장 중

✅ 3. 암호화 (Encryption)

📌 개념

암호화는 평문 데이터를 암호 알고리즘을 이용해 읽을 수 없는 형식으로 변환하는 과정이다.
복호화는 이 암호문을 다시 원래 평문으로 되돌리는 과정이다.

🔐 암호화 목적

목적	설명
기밀성 (Confidentiality)	제3자가 내용을 볼 수 없음
무결성 (Integrity)	데이터가 변경되지 않았음을 보장
인증 (Authentication)	통신 주체의 신원 확인
부인 방지 (Non-repudiation)	송신자가 행위를 부정하지 못함 (전자서명 등)

🔐 암호화 방식

구분	방식	예시
대칭키	AES, ChaCha20	빠름, 키 공유 필요
비대칭키	RSA, ECC	느림, 키 교환 가능
해시기반 인증	HMAC	데이터 위조 방지
AEAD	AES-GCM	암호화 + 인증 동시 제공 (TLS 1.3 기본)

🎯 예시: TLS 암호화 흐름

```
[애플리케이션 데이터]
↓
암호화 (AES-GCM)
↓
TLS 레코드 구조로 래핑
↓
TCP 패킷으로 전송
```

- 이 전체 흐름은 **표현 계층 + 전송 계층** 조합
- 표현 계층이 실제 암호화 처리를 담당

📁 구현 라이브러리 예

라이브러리	언어	설명
OpenSSL	C	TLS, 암호화, 해시 등 종합 지원
GnuTLS	C	보안 커뮤니케이션 라이브러리
libsodium	C	고수준 암호화, 안전한 기본 제공
mbedTLS	C	임베디드/경량 환경용 TLS 라이브러리

📝 정리 요약

항목	핵심 기능	대표 기술
인코딩	데이터 → 바이트 변환	UTF-8, JSON, Base64
압축	전송 전 데이터 축소	Gzip, Brotli, LZ77
암호화	보안 전송을 위한 변환	AES, RSA, TLS

🧭 OSI 계층 관점 비교

계층	기능	관련 내용
표현 계층 (6계층)	인코딩, 압축, 암호화	구조적 처리, 내용 변환
전송 계층 (4계층)	세그먼트 전송	암호화 여부 알지 못함
응용 계층 (7계층)	사용자 처리	포맷 지정, JSON/XML 사용 등

- # TLS/SSL, JSON/XML 변환

✅ 1. TLS/SSL — 표현 계층의 암호화 구현

📌 개요

TLS (Transport Layer Security) 와 **SSL (Secure Sockets Layer)** 는
애플리케이션 데이터의 기밀성과 무결성을 보장하기 위해 사용되는 보안 프로토콜이다.

> 비록 이름은 "전송 계층 보안"이지만,
> **암호화, 복호화, 인증, 키 협상 등의 로직은 표현 계층의 역할**에 해당한다.

🔐 TLS vs SSL

항목	SSL	TLS
최신 버전	SSL 3.0 (Deprecated)	TLS 1.3 (Active)
보안성	취약함	강화됨
알고리즘 선택	느슨함	엄격한 협상
사용 권장 여부	❌ 사용 금지	✅ 표준 보안 프로토콜

🧱 TLS 구조 요약

```
+--------------------------+
| Application Data         |
+--------------------------+
| TLS Handshake Protocol   | ← 세션 설정 (세션 계층 성격)
| TLS Record Protocol      | ← 암호화/복호화 (표현 계층 성격)
+--------------------------+
| TCP                      |
+--------------------------+
```

🧭 TLS 주요 기능 (표현 계층 관점)

기능	설명
암호화	대칭키 (AES-GCM 등)로 애플리케이션 데이터 암호화
키 교환	비대칭키 (RSA, ECDHE) 사용
서버 인증	X.509 인증서 기반
세션 재개	암호화된 세션 식별자/티켓 기반 복원

기능	설명
AEAD	인증된 암호화 (기밀성 + 무결성 동시 제공)

🔒 TLS 실전 흐름

```
[ClientHello]          → 알고리즘 목록, 랜덤값, SNI
[ServerHello]          ← 선택된 알고리즘, 인증서, 키
[Key Exchange]         → pre-master secret
[ChangeCipherSpec]     → 이후 암호화 시작
[Encrypted Application Data] ← TLS 암호화된 컨텐츠
```

💡 TLS 사용 예 (C 기반 OpenSSL)

```
SSL_library_init();
SSL_CTX *ctx = SSL_CTX_new(TLS_server_method());
SSL *ssl = SSL_new(ctx);
SSL_set_fd(ssl, client_fd);
SSL_accept(ssl);  // TLS 핸드셰이크 수행
SSL_read(ssl, buf, sizeof(buf));  // 암호화된 데이터 수신
```

✅ 2. JSON/XML 변환 — 데이터 표현 포맷 처리

📌 개요

JSON과 **XML**은 서로 다른 구조화된 텍스트 기반 데이터 표현 방식이며,
표현 계층에서 **데이터를 직렬화/역직렬화하여 전송 가능한 형태로 바꾸는 작업**을 담당한다.

🧩 구조적 차이

항목	JSON	XML
문법	JavaScript 객체 표기	태그 기반
가독성	높음	낮음
파서	경량, 빠름	무겁고 느림
확장성	제한적	무한 확장 가능 (속성 등)
사용 분야	웹 API, 모바일	문서, 설정파일, SOAP 등

📂 예시 변환

JSON

```
{
  "name": "Alice",
  "age": 30
}
```

XML

```
<user>
  <name>Alice</name>
  <age>30</age>
</user>
```

🧰 변환 동작 흐름

```
[JSON 객체] → stringify → UTF-8 문자열 → TCP 전송
↓
수신 측: 문자열 → parse → 내부 구조 객체
```

- 이 모든 변환이 **표현 계층의 책임**
- 전송 계층은 단순히 바이트 단위로 운반만 함

💻 실전 예 (C 기반 JSON 변환)

```
#include <json-c/json.h>

struct json_object *obj = json_tokener_parse("{\"name\":\"Alice\"}");
const char* name = json_object_get_string(json_object_object_get(obj, "name"));
```

⚙️ XML 파싱 예 (libxml2)

```
xmlDocPtr doc = xmlParseFile("data.xml");
xmlNodePtr root = xmlDocGetRootElement(doc);
```

- XML은 태그 계층 구조를 유지하므로 **복잡한 표현도 가능**

📝 정리 요약

항목	설명	계층
TLS/SSL	보안 통신 세션, 암호화/인증	표현 계층 + 세션 계층
JSON/XML 변환	구조화 데이터의 직렬화/역직렬화	표현 계층
공통점	데이터 내용의 형식/보안 관리를 담당	OSI 6계층

🧭 계층 비교 정리

계층	역할
OSI 7계층 (응용 계층)	사용자가 요청한 기능 처리
OSI 6계층 (표현 계층)	데이터 변환, 암호화, 포맷 처리
OSI 5계층 (세션 계층)	세션 연결 유지 및 상태 관리

1.9 OSI 7계층: 응용 계층 (Application Layer)

✅ 개요

응용 계층(Application Layer)은 **OSI 7계층 중 가장 상위 계층**으로,
사용자 애플리케이션이 네트워크 서비스를 이용할 수 있도록 하는 인터페이스 계층이다.
이 계층은 실제 데이터를 생성하고 해석하는 소프트웨어(ex. 웹 브라우저, 메일 클라이언트)와 직접 연결된다.

🎯 핵심 역할

기능	설명
사용자 인터페이스 제공	애플리케이션과 네트워크 간 인터페이스
프로토콜 정의	HTTP, FTP, SMTP 등 사용 목적별 통신 규약
요청/응답 처리	서버와 클라이언트 간의 논리적 데이터 교환
오류 처리	프로토콜 수준의 상태 코드 및 메시지 처리

🧱 주요 프로토콜 비교

프로토콜	목적	전송 계층	기본 포트
HTTP	웹 문서 송수신	TCP	80 (HTTP), 443 (HTTPS)
DNS	도메인 이름 → IP 주소 변환	UDP/TCP	53

프로토콜	목적	전송 계층	기본 포트
FTP	파일 송수신	TCP	20 (Data), 21 (Control)
SMTP	이메일 발송	TCP	25
POP3/IMAP	이메일 수신	TCP	110 / 143
Telnet	원격 터미널 접속	TCP	23
SSH	보안 원격 접속	TCP	22

🔍 주요 프로토콜 상세

🌐 HTTP (HyperText Transfer Protocol)

📌 목적

- **웹 기반 데이터 전송**
- 클라이언트(웹 브라우저) ↔ 서버 간 **요청/응답(Request/Response)** 구조

📋 요청 메시지 구조

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

📋 응답 메시지 구조

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1256

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

💡 특징

항목	설명
무상태성	요청 간 상태가 유지되지 않음 (→ 쿠키/세션 필요)
버전	HTTP/1.0, 1.1, 2, 3 (QUIC 기반)
보안	HTTPS는 TLS 위에서 HTTP 동작

🌐 DNS (Domain Name System)

📌 목적

- **도메인 이름을 IP 주소로 변환**
- 예: `www.google.com` → `142.250.196.4`

📋 메시지 구조

필드	설명
Header	요청 ID, 플래그, 질문 수 등
Question	도메인 이름, 질의 유형 (A, AAAA, MX 등)
Answer	IP 주소 응답
Authority	권한 있는 서버 정보
Additional	부가 정보

💡 특징

항목	설명
프로토콜	UDP 53 (보통), TCP 53 (Zone Transfer 등)
타입	A (IPv4), AAAA (IPv6), MX (메일), CNAME 등
캐시	운영체제 및 DNS 서버에서 응답을 캐시함

📂 FTP (File Transfer Protocol)

📌 목적

- **파일 업로드/다운로드**
- 두 개의 채널 사용:
  - **Control Channel (포트 21)**: 명령 전송
  - **Data Channel (포트 20)**: 실제 파일 데이터

💡 특징

항목	설명
연결형	항상 TCP 기반
로그인 필요	기본적으로 사용자 인증 필요 (익명 허용 가능)
보안	평문 전송 (→ FTPS, SFTP로 대체됨)
명령어	`USER`, `PASS`, `LIST`, `RETR`, `STOR`, `QUIT` 등

📨 SMTP (Simple Mail Transfer Protocol)

📌 목적

- **이메일 발송 전용**
- 클라이언트(MUA) → 서버(MTA) → 대상 MTA로 전달

📋 메시지 예시

```
HELO mail.example.com
MAIL FROM:<alice@example.com>
RCPT TO:<bob@example.net>
DATA
Subject: Hello
<내용>
.
QUIT
```

💡 특징

항목	설명
포트	TCP 25 (기본), 587 (STARTTLS), 465 (SSL)
한계	수신 기능 없음 (→ POP3, IMAP 필요)
보안	TLS 또는 STARTTLS 기반 암호화 필요

🧭 OSI 계층 정리

계층	사용자 인식 수준	주요 역할
7 응용 계층	사용자 ↔ 네트워크	사용자 요청 처리
6 표현 계층	데이터 구조, 암호화	인코딩, 압축, 암호화
5 세션 계층	연결 유지	세션 상태 관리
4 전송 계층	TCP/UDP	포트 관리, 전송 보장
3 네트워크 계층	IP	라우팅, 주소 지정
2 데이터 링크 계층	MAC	프레임 전달
1 물리 계층	전기 신호	케이블, NIC 등 하드웨어 전송

✅ 요약 정리

항목	HTTP	DNS	FTP	SMTP
용도	웹 문서	이름 ↔ IP	파일 전송	메일 발송
전송 계층	TCP	UDP/TCP	TCP	TCP
포트	80/443	53	20/21	25/587
특이점	요청/응답 기반	캐싱 존재	제어+데이터 채널	수신은 별도 프로토콜 필요

• HTTP, DNS, FTP, SMTP 등 프로토콜

✅ 1. HTTP (HyperText Transfer Protocol)

📌 목적

- 웹 문서, 이미지, JSON 등 **웹 자원의 요청/응답 처리**

📦 계층 구조

- **OSI 계층**: 응용 계층 (7계층)
- **전송 계층**: TCP
- **포트 번호**: 80 (HTTP), 443 (HTTPS)

🔄 메시지 구조

✅ 요청(Request)

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: curl/7.81.0
Accept: */*
```

✅ 응답(Response)

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1024

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

🔐 보안

- **HTTPS = HTTP + TLS**
- TLS로 암호화된 채널 위에서 HTTP가 동작함 (계층 분리)

🧪 실전 명령어

```
curl -v http://example.com
curl -X POST -d '{"id":1}' -H "Content-Type: application/json"
https://example.com/api
```

✅ 2. DNS (Domain Name System)

📌 목적

- **도메인 이름을 IP 주소로 변환**

📦 계층 구조

- **OSI 계층**: 응용 계층 (7계층)
- **전송 계층**: UDP (기본), TCP (Zone Transfer)
- **포트 번호**: 53

🔄 메시지 구조

필드	설명
Header	Transaction ID, Flags (Query/Response), QDCount, ANCount
Question	질의 도메인, 타입(A, AAAA, MX 등), 클래스(IN)
Answer	IP 주소 응답 등
Authority	권한 서버 정보
Additional	부가정보 (예: 네임서버 IP)

💡 질의 타입

타입	설명
A	IPv4 주소
AAAA	IPv6 주소
MX	메일 서버
CNAME	별칭 (Canonical Name)
NS	네임서버 정보

🧪 실전 명령어

```
dig google.com A
dig +short www.naver.com
nslookup example.com
```

✅ 3. FTP (File Transfer Protocol)

📌 목적

- **파일 업로드/다운로드 전송 프로토콜**

📦 계층 구조

- **OSI 계층**: 응용 계층 (7계층)
- **전송 계층**: TCP
- **포트 번호**: 21 (Control), 20 (Data)

🔗 연결 방식

채널	포트	설명
Control Channel	21	명령 전달용
Data Channel	20	파일 전송용 (Active mode)

📋 FTP 명령어 흐름

```
USER alice
PASS secret
CWD /public
LIST
RETR file.txt
STOR newfile.txt
QUIT
```

🔐 보안

- FTP는 기본적으로 평문 전송
- FTPS (SSL 기반), SFTP (SSH 기반)로 보안 대체 필요

🧪 실전 명령어

```
ftp ftp.gnu.org
# 또는
lftp -u username,password ftp.example.com
```

✅ 4. SMTP (Simple Mail Transfer Protocol)

📌 목적

- **이메일을 발송**하는 전용 프로토콜

📦 계층 구조

- **OSI 계층**: 응용 계층 (7계층)
- **전송 계층**: TCP
- **포트 번호**: 25 (기본), 587 (STARTTLS), 465 (SSL)

📋 SMTP 명령어 흐름

```
HELO mail.example.com
MAIL FROM:<alice@example.com>
RCPT TO:<bob@example.com>
DATA
Subject: Hello
This is a test message.
.
QUIT
```

📩 이메일 구조

필드	설명
From, To	송수신자 이메일 주소
Subject	제목
Body	본문 (텍스트 또는 MIME 멀티파트)

🔐 보안

- STARTTLS 또는 SSL 기반 암호화 필요
- 인증 방식: PLAIN, LOGIN, CRAM-MD5 등

🧪 실전 명령어 (telnet 직접 테스트)

```
telnet smtp.example.com 25
EHLO myclient
AUTH LOGIN
MAIL FROM:<alice@example.com>
RCPT TO:<bob@example.com>
DATA
Hello world!
.
QUIT
```

✅ 전체 요약 비교표

프로토콜	목적	포트	전송 계층	보안 확장
HTTP	웹 요청/응답	80 / 443	TCP	HTTPS (TLS)
DNS	이름 → IP 매핑	53	UDP/TCP	DNSSEC
FTP	파일 전송	21 / 20	TCP	FTPS, SFTP
SMTP	메일 발송	25 / 587 / 465	TCP	STARTTLS, SMTPS

🧭 OSI 계층별 위치

```
응용 계층 —— HTTP, FTP, DNS, SMTP
표현 계층 —— TLS/SSL, 인코딩, 압축
세션 계층 —— 연결 유지, TLS Handshake
전송 계층 —— TCP/UDP
네트워크 계층 —— IP
데이터 링크 계층 —— Ethernet, ARP
물리 계층 —— 케이블, 신호
```

- # 사용자 요청/응답 흐름

✅ 1. 전체 시나리오 요약

> 사용자가 웹 브라우저에 `www.example.com` 을 입력하고 Enter를 누른 순간부터,
> 화면에 페이지가 로딩되기까지 벌어지는 모든 네트워크 흐름을 하나씩 추적해보자.

📶 2. 계층별 요청/응답 흐름 (OSI 7계층 기준)

```
[사용자 입력]
↓
7. Application Layer: HTTP 요청 생성 (브라우저)
↓
6. Presentation Layer: JSON 인코딩, GZIP 압축, TLS 암호화
↓
5. Session Layer: TLS Handshake, Session ID 협상
↓
4. Transport Layer: TCP 연결 및 데이터 세그먼트화
↓
3. Network Layer: IP 주소를 목적지로 라우팅
↓
2. Data Link Layer: MAC 주소 설정, 프레임 생성
↓
1. Physical Layer: 신호 전송 (Ethernet/Wi-Fi)

< 응답은 위 순서 반대로 위로 올라옴 >
```

📜 3. 요청 단계별 상세 흐름

단계	설명
① 주소 입력	브라우저가 `www.example.com` 입력
② DNS 질의	클라이언트가 DNS 서버에 A/AAAA 레코드 요청 → IP 획득
③ TCP 연결	3-way handshake (SYN → SYN+ACK → ACK)
④ TLS Handshake (HTTPS인 경우)	인증서 교환, 암호화 알고리즘 협상, 대칭키 생성
⑤ HTTP 요청 전송	`GET /index.html HTTP/1.1` 등
⑥ 서버 처리	요청 URI 해석, 백엔드 호출, DB 조회
⑦ HTTP 응답 반환	`HTTP/1.1 200 OK + Content-Type + Body`
⑧ 렌더링	브라우저가 HTML, CSS, JS 해석 및 표시

🔁 4. Request/Response 구조 예시 (HTTP 기준)

📤 요청 (Request)

```
GET /api/user?id=5 HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: application/json
```

📥 응답 (Response)

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 42

{"id":5,"name":"Alice","email":"a@x.com"}
```

🔐 5. 암호화와 상태 유지 (HTTPS + 세션)

기술	역할	OSI 계층
TLS	암호화, 인증, 세션 설정	표현 + 세션 계층
Cookie	사용자 세션 식별	응용 계층
JWT	토큰 기반 인증 정보 전달	응용 계층

🧪 6. 실전 디버깅 예시

🔍 `curl`로 요청 확인

```
curl -v https://jsonplaceholder.typicode.com/posts/1
```

🔍 Wireshark로 트래픽 캡처

- 필터: `http`, `tcp.port == 443`
- SYN, ACK, TLS, HTTP 메시지를 눈으로 확인 가능

🔄 7. 기타 프로토콜에서의 요청/응답 구조

프로토콜	요청 구조	응답 구조
HTTP	`GET /path`	`200 OK` + Body
DNS	A 질의	A 레코드 응답
FTP	`RETR file.txt`	`150 Data Connection`
SMTP	`MAIL FROM:`	`250 OK`
gRPC	바이너리 메시지	Stream 또는 단일 응답

🧭 8. 요청 흐름 요약 다이어그램

```
사용자 → 브라우저 → DNS → TCP 연결 → TLS → HTTP 요청
                                  ↓
                               서버 처리
                                  ↓
사용자 ← 브라우저 ← HTTP 응답 ← TLS ← TCP
```

✅ 요약 정리

항목	내용
구조	Request → Process → Response
관여 계층	OSI 7, 6, 5, 4, 3
특징	비동기 가능 (AJAX), 암호화 가능 (TLS), 무상태 모델 (HTTP)
테스트 도구	curl, Postman, Wireshark, telnet, browser DevTools

1.10 포트, 소켓, 바인딩 개념

✅ 1. 포트 (Port)

📌 개념

- 포트는 **운영체제에서 실행 중인 네트워크 애플리케이션을 식별하기 위한 논리적 숫자 식별자**
- 하나의 IP 주소에는 수천 개의 애플리케이션이 접속할 수 있는데, **각 애플리케이션을 구분하기 위해 포트 번호를 사용**

📦 범위 및 종류

범위	종류	예시
0 ~ 1023	**Well-known 포트**	HTTP: 80, HTTPS: 443, FTP: 21
1024 ~ 49151	**Registered 포트**	MySQL: 3306, Docker: 2375
49152 ~ 65535	**Dynamic/Private 포트**	클라이언트용 임시 포트

📌 동작 방식

- 서버: **지정된 포트를 listen()** 하며 요청 대기
- 클라이언트: OS가 **자동으로 임시 포트**를 할당하여 통신

✅ 2. 소켓 (Socket)

📌 개념

- 소켓은 **네트워크를 통한 통신을 위한 소프트웨어 구조체**
- IP 주소 + 포트 번호 + 프로토콜(TCP/UDP) 을 결합한 **종단점(endpoint)** 을 의미

> 즉, 소켓은 **"통신용 파일 디스크립터"** 이자, **"IP/Port의 추상화 객체"**

📋 구조

필드	설명
IP 주소	어느 호스트와 통신할지
포트	어느 애플리케이션과 통신할지
프로토콜	TCP or UDP
소켓 타입	`SOCK_STREAM`, `SOCK_DGRAM`

📌 종류

소켓 타입	설명
`SOCK_STREAM`	TCP 연결형 소켓
`SOCK_DGRAM`	UDP 비연결형 소켓
`SOCK_RAW`	사용자 정의 패킷 생성용

✅ 3. 바인딩 (Binding)

📌 개념

소켓에 IP 주소와 포트 번호를 할당하는 작업을 **"바인딩(binding)"** 이라고 부른다.

- 서버 측에서는 `bind()` 함수를 통해 **소켓 ↔ IP:Port 매핑**을 수행해야 함
- 바인딩하지 않으면 서버는 클라이언트의 요청을 받을 수 없음

🧪 C 코드 예시 (TCP 서버 바인딩)

```
int server_fd = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;

bind(server_fd, (struct sockaddr*)&addr, sizeof(addr));  // 포트 8080에 바인딩
listen(server_fd, 5);
```

🧭 계층적 연결 구조

```
[ 응용 계층 ]          ← 웹 서버, DB, 메일 서비스
[ 전송 계층 ]          ← TCP/UDP 포트 관리
[ 네트워크 계층 ]      ← IP 주소
```

- 소켓 = (IP 주소, 포트, 프로토콜)
- 이 조합은 **5-튜플** 또는 **소켓 페어**로 불리며, 양방향 연결을 구성함

🔗 종단 간 소켓 쌍 (TCP 5-Tuple)

항목	설명
Source IP	클라이언트 IP
Source Port	클라이언트 포트

항목	설명
Dest IP	서버 IP
Dest Port	서버 포트
Protocol	TCP 또는 UDP

🧪 실전 확인 명령어

▶️ 소켓 열려 있는지 확인

```
netstat -tulnp | grep 8080
ss -tulwn
```

▶️ 바인딩 에러 원인 예시

- "Address already in use" → 해당 포트가 이미 다른 프로세스에 바인딩됨
- 해결: `SO_REUSEADDR` 옵션 사용

✅ 요약 정리

개념	설명
포트	한 IP에서 여러 애플리케이션을 구분하기 위한 숫자
소켓	IP, 포트, 프로토콜을 조합한 통신 단위
바인딩	소켓에 IP/Port를 할당하여 통신 수신 가능하게 만듦
연결	클라이언트가 서버의 IP:Port에 연결을 시도하여 소켓 연결 형성

1.11 IP 주소 체계 (IPv4, IPv6)

✅ 1. 개요

IP 주소는 네트워크 상에서 **각 장치(호스트)를 식별**하기 위한 고유한 숫자이다.
현재 두 가지 체계가 존재한다:

종류	크기	주소 개수	표기법
IPv4	32비트	약 42억 개 (2^{32})	10.0.0.1
IPv6	128비트	사실상 무한 (2^{128})	2001:0db8::1

🧱 2. IPv4 주소 체계

📌 구조

IPv4 주소는 **32비트** 정수이며, 일반적으로 **8비트씩 끊어** 4개의 10진수로 표현함.

```
11000000.10101000.00000001.00000001
→ 192.168.1.1
```

🧩 주소 구분

종류	예시	설명
공인 IP	8.8.8.8	인터넷 상에서 유일하게 식별되는 주소
사설 IP	192.168.0.1, 10.0.0.1	로컬 네트워크 전용
루프백	127.0.0.1	자기 자신을 가리킴
브로드캐스트	255.255.255.255	전체 네트워크에 전송
네트워크 주소	192.168.0.0	특정 서브넷 식별용

📐 서브넷팅 (Subnetting)

- **IP 주소 + 서브넷 마스크** = 네트워크 구분
- 마스크 예: `255.255.255.0` → `/24`

예시

```
IP: 192.168.1.42
Subnet Mask: 255.255.255.0
→ 네트워크: 192.168.1.0/24
→ 호스트 범위: 192.168.1.1 ~ 192.168.1.254
```

🌐 3. IPv6 주소 체계

📌 구조

IPv6는 **128비트** 주소 공간을 사용하며, **16비트 단위**로 8개의 **16진수 블록**으로 표현함.

```
2001:0db8:85a3:0000:0000:8a2e:0370:7334
→ 축약: 2001:db8:85a3::8a2e:370:7334
```

📋 주요 주소 유형

주소	설명
Unicast	단일 대상
Multicast	그룹 대상
Anycast	가장 가까운 노드
Loopback	`::1` (IPv4의 127.0.0.1에 해당)
Link-local	`fe80::/10`, 같은 링크에서만 유효 (라우팅 안 됨)
Global Unicast	인터넷에서 사용되는 고유한 주소 (`2000::/3`)

🧱 IPv6의 구조적 특징

- **ARP 없음** → 대신 **Neighbor Discovery Protocol (NDP)** 사용
- **브로드캐스트 없음** → 대신 **멀티캐스트 사용**
- **헤더 간소화** (고정 크기 40바이트)
- **IPSec 내장 지원**

🧪 4. 소켓 프로그래밍에서의 IP 체계

▶️ IPv4

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
inet_pton(AF_INET, "192.168.0.10", &addr.sin_addr);
```

▶️ IPv6

```
struct sockaddr_in6 addr6;
addr6.sin6_family = AF_INET6;
addr6.sin6_port = htons(8080);
inet_pton(AF_INET6, "2001:db8::1", &addr6.sin6_addr);
```

📊 5. IPv4 vs IPv6 비교 정리

항목	IPv4	IPv6
주소 길이	32비트	128비트
표현 방식	10진수 4개	16진수 8블록

항목	IPv4	IPv6
주소 수	약 42억	사실상 무한
헤더 크기	가변 (20~60B)	고정 (40B)
브로드캐스트	지원함	없음 (멀티캐스트만)
NAT	일반적	사용 지양
보안	선택적 IPSec	내장 IPSec
호환성	전통적 시스템	최신 환경 중심

📜 실전 명령어

```
# IP 확인
ip a          # IPv4/IPv6 모두 확인
ifconfig      # (구형 명령)

# 특정 도메인의 IPv6 주소 질의
dig AAAA google.com

# ping
ping 8.8.8.8        # IPv4
ping6 2001:4860::8888  # IPv6
```

🧭 계층 관계 요약

```
[ 응용 계층 ]   ← HTTP, FTP, DNS
[ 전송 계층 ]   ← TCP/UDP + 포트 번호
[ 네트워크 계층 ] ← IP 주소 (IPv4 or IPv6)
```

- IPv4/IPv6는 전송 계층(TCP/UDP)의 아래에서 **라우팅, 주소 지정**을 담당
- 소켓 생성 시, `AF_INET` vs `AF_INET6` 로 체계를 구분

✅ 요약 정리

구분	IPv4	IPv6
주소 크기	32비트	128비트
표현 예	192.168.1.1	2001:db8::1
라우팅 방식	브로드캐스트, NAT	멀티캐스트, Anycast
시스템 호출	`sockaddr_in`	`sockaddr_in6`

구분	IPv4	IPv6
전환 방식	NAT, Dual Stack, Tunneling	점진적 확산 중

1.12 DNS, DHCP, NAT, 방화벽

✅ 개요 요약

구성요소	역할	OSI 계층
DNS	도메인 이름 ↔ IP 주소 변환	응용 계층 (7계층)
DHCP	클라이언트에 IP 주소 자동 할당	응용 계층 (7계층)
NAT	사설 ↔ 공인 IP 주소 변환	네트워크 계층 (3계층)
방화벽	트래픽 필터링 및 보안 정책 적용	전 계층 관여, 주로 3~4계층

🌐 1. DNS (Domain Name System)

📌 개요

DNS는 **사람이 읽기 쉬운 도메인 이름**을
네트워크가 처리할 수 있는 IP 주소로 변환해주는 분산형 데이터베이스 시스템이다.

> 예: `www.google.com` → `142.250.196.4`

🔍 작동 흐름

1. 사용자가 도메인 입력
2. 클라이언트 OS가 **로컬 DNS 캐시** 확인
3. 없으면 **재귀 DNS 서버**로 요청
4. 루트 → TLD → Authoritative 서버 순으로 질의
5. IP 주소 반환 → 응용 계층(HTTP 등)이 사용

📜 질의 방식

방식	설명
Recursive	클라이언트 대신 전체 질의 수행
Iterative	클라이언트가 각 DNS 서버에 직접 단계별 질의

🧪 실전 명령어

```
dig example.com
nslookup google.com
host naver.com
```

📡 2. DHCP (Dynamic Host Configuration Protocol)

📌 개요

DHCP는 **IP 주소, 서브넷 마스크, 게이트웨이, DNS 등 네트워크 설정을 자동으로 할당**해주는 프로토콜이다.

🔁 4단계 작동 흐름 (DORA)

```
1. DHCP Discover    ← 브로드캐스트: "IP 주소 줄 사람?"
2. DHCP Offer       ← 서버가 제안: "192.168.0.100 써봐"
3. DHCP Request     ← 클라이언트가 수락
4. DHCP Acknowledgement ← 서버가 최종 확정
```

- 포트: UDP 67 (서버), UDP 68 (클라이언트)

🔐 임대(Lease) 개념

항목	설명
임대 시간	IP를 사용할 수 있는 시간
갱신 요청	시간이 지나면 재갱신
만료 시	IP 반납 또는 재할당 시도

🧪 확인 방법

```
ip a              # IP 주소 확인
nmcli device show # DHCP로 받은 세부 정보
```

🔁 3. NAT (Network Address Translation)

📌 개요

NAT는 내부 사설 IP 주소를 **하나의 공인 IP로 변환하여 외부와 통신 가능하게 해주는 기술**이다.
IPv4 주소 부족 문제를 해결하기 위해 필수적으로 사용됨.

🧩 NAT 종류

종류	설명	예시
Static NAT	내부 IP ↔ 고정 외부 IP 1:1 매핑	서버 공개 시
Dynamic NAT	풀 내 IP에서 임의 할당	ISP 환경
PAT (NAPT)	다수의 내부 IP ↔ 하나의 공인 IP + 포트	홈 라우터에서 일반적 사용

📍 예시: PAT

```
내부: 192.168.1.10:34567 → 공인: 203.0.113.5:49100
내부: 192.168.1.11:34568 → 공인: 203.0.113.5:49101
```

- 외부 응답은 포트 번호 기반으로 다시 내부로 전달됨

🧪 리눅스에서 NAT 설정 (iptables)

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

🔥 4. 방화벽 (Firewall)

📌 개요

방화벽은 **네트워크 패킷을 감시, 차단, 허용, 수정하는 보안 장치 또는 소프트웨어**이다.
내부 네트워크를 보호하고, **정책 기반 트래픽 제어**를 수행한다.

🔐 방화벽 기준

기준	예시
IP 주소	특정 출발지 차단
포트 번호	`DROP all except 22, 80, 443`
프로토콜	UDP 차단, ICMP 제한
패킷 내용	DPI (Deep Packet Inspection) 기반 제어도 가능

🧰 방화벽 유형

유형	설명
패킷 필터링	IP/Port 단위 제어

유형	설명
상태 추적 (Stateful)	세션 기반 판단
애플리케이션 레벨	HTTP 요청 헤더 기반 판단 가능
네트워크 IDS/IPS	침입 탐지 및 차단 통합 기능

🧪 리눅스 방화벽 예시 (ufw, iptables)

```
ufw enable
ufw allow 22    # SSH 허용
ufw deny 23     # Telnet 차단

iptables -A INPUT -p tcp --dport 80 -j ACCEPT
iptables -A INPUT -p tcp --dport 23 -j DROP
```

🧭 종합 구조도

```
[사용자]
 ↓          ← DNS 질의
[DHCP로 IP 할당]
 ↓
[NAT 변환] ← 내부 사설 IP ↔ 공인 IP
 ↓
[방화벽 검사] ← 정책에 따라 허용/차단
 ↓
[인터넷 서버]
```

✅ 요약 정리

항목	기능	주요 포트	계층
DNS	도메인 → IP 변환	UDP/TCP 53	응용
DHCP	IP 자동 할당	UDP 67/68	응용
NAT	사설 ↔ 공인 IP 변환	-	네트워크
방화벽	트래픽 필터링/보안	-	3~7 계층까지 가능