# Chapter 01

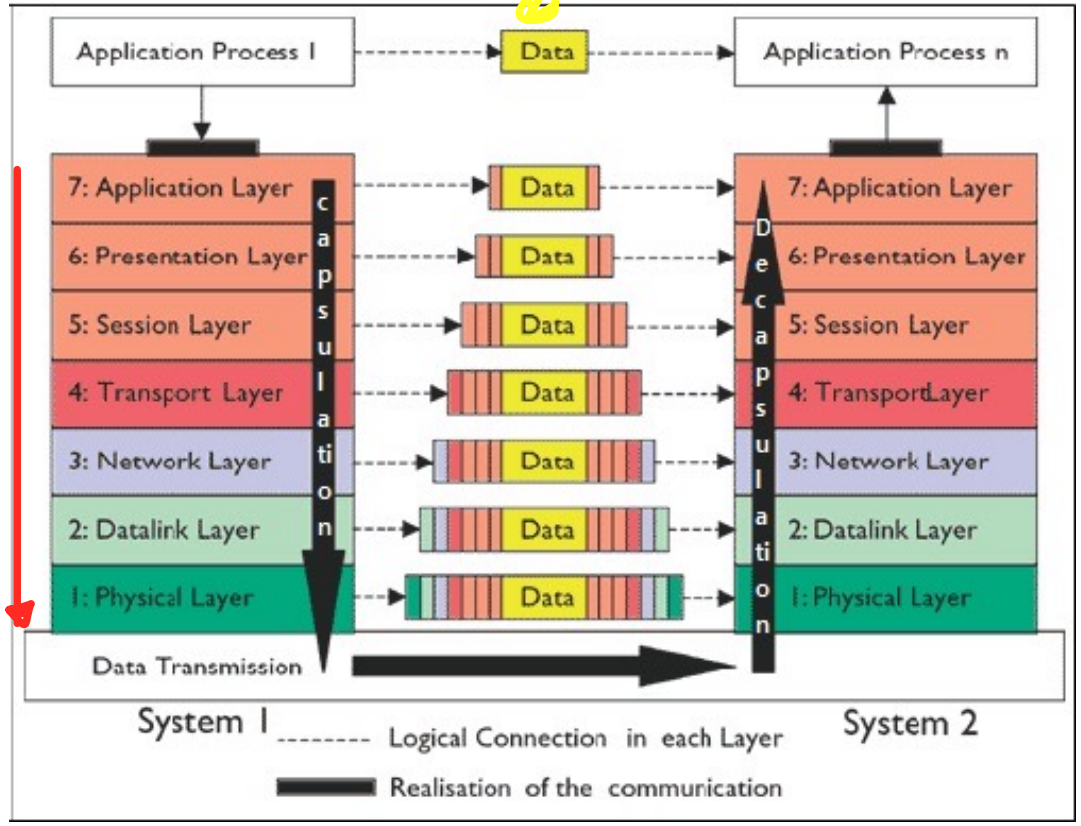
네트워크 프로그래밍과 소켓의 이해

# OSI 7 Layers

참조 모델

- 국제표준화기구(ISO)에서 개발한 모델
  - 컴퓨터 네트워크 프로토콜 디자인과 통신을 7계층으로 나누어 설명
  - 각 계층은 하위 계층의 기능만을 이용, 상위 계층에게 기능을 제공

두 어플리케이션이 주고 받는 data




https://blog.naver.com/PostView.nhn?blogId=pst8627&logNo=221670903384

# OSI 7 Layers

- OSI 7 layer
  - 레퍼런스 모델: 프로토콜 개발에 참조 용도
- TCP/IP protocol
  - Transmission Control Protocol / Internet Protocol
  - 실제로 구현된 프로토콜 모델

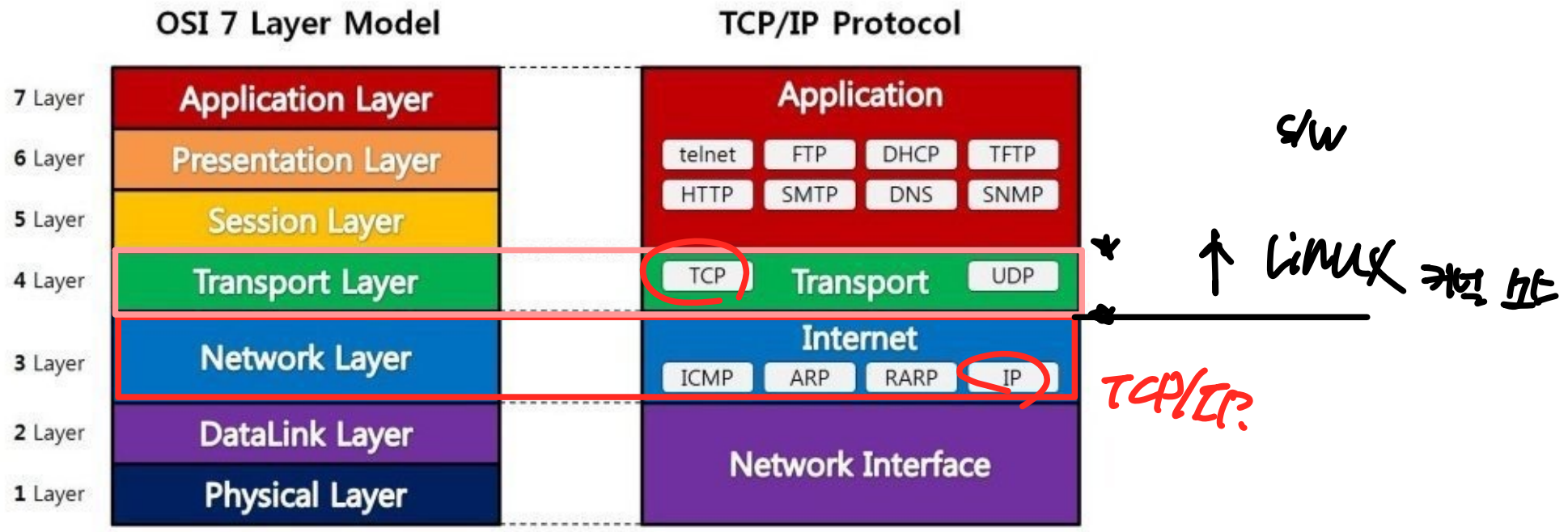
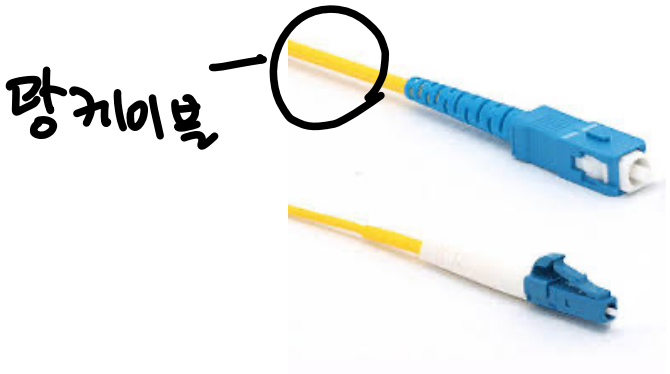



그림 참조: https://shlee0882.tistory.com/110

# 1계층: Physical Layer

- 역할
  - 전기적, 기계적 특성을 이용해서 통신 케이블로 데이터 전송
  - 통신 단위는 비트(0, 1)
  - 데이터를 전달하는 기능 (어떤 데이터 인지 관여하지 않음)
- 장비
  - 통신 케이블, 리피터, 허브 등

물리계층




리피터


허브

# 2계층: Data Link Layer

- 역할
  - 물리 계층을 통해 송수신되는 정보의 오류 검출 및 흐름 관리
  - 안전한 정보의 전달을 수행할 수 있도록 도움
  - 통신 오류 검출, 재전송
  - MAC 주소를 이용하여 통신
- 예
  - 브리지, 스위치 (스위칭 브리지) 등

스위치

# 3계층: Network Layer

- 역할
  - 데이터를 목적지까지 전달하는 기능(라우팅)
  - 경로 선택하고, 경로에 따라 패킷을 전송
  - IP 주소 사용
  - 다양한 프로토콜 및 라우팅 기술 사용
- 장비
  - 라우터

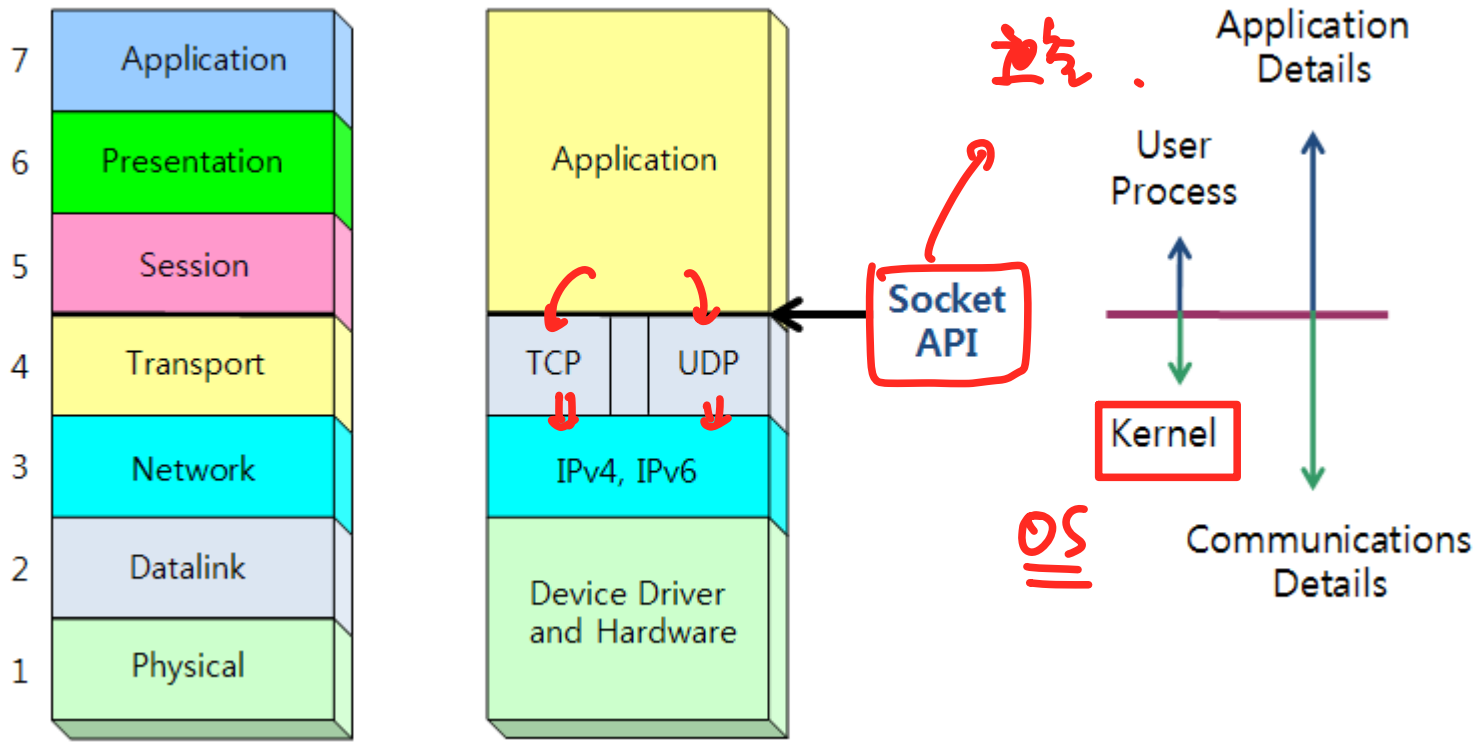
# 그 외 계층들

- 4계층: Transport layer (전송 계층)
  - 최종 시스템 및 호스트 간의 데이터 전송 조율을 담당
  - 흐름 제어, 중복 검사
  - 보낼 데이터의 용량, 목적지 등을 처리
  - TCP/UDP 프로토콜 사용
- 5계층: Session layer (세션 계층)
  - 통신 세션을 구성하는 계층으로 포트(port)연결
  - 사용자간의 포트 연결(session)이 유효한지 확인하고 설정
- 6계층: Presentation layer (표현 계층)
  - 운영체제의 일부분
  - 네트워크 형식 ⇔ 응용프로그램 형식
  - 부호화, 변화, 데이터 압축 및 암호화 등
- 7계층: Application layer (응용 계층)
  - 응용 프로세스간 정보 교환
  - 각종 응용 프로그램

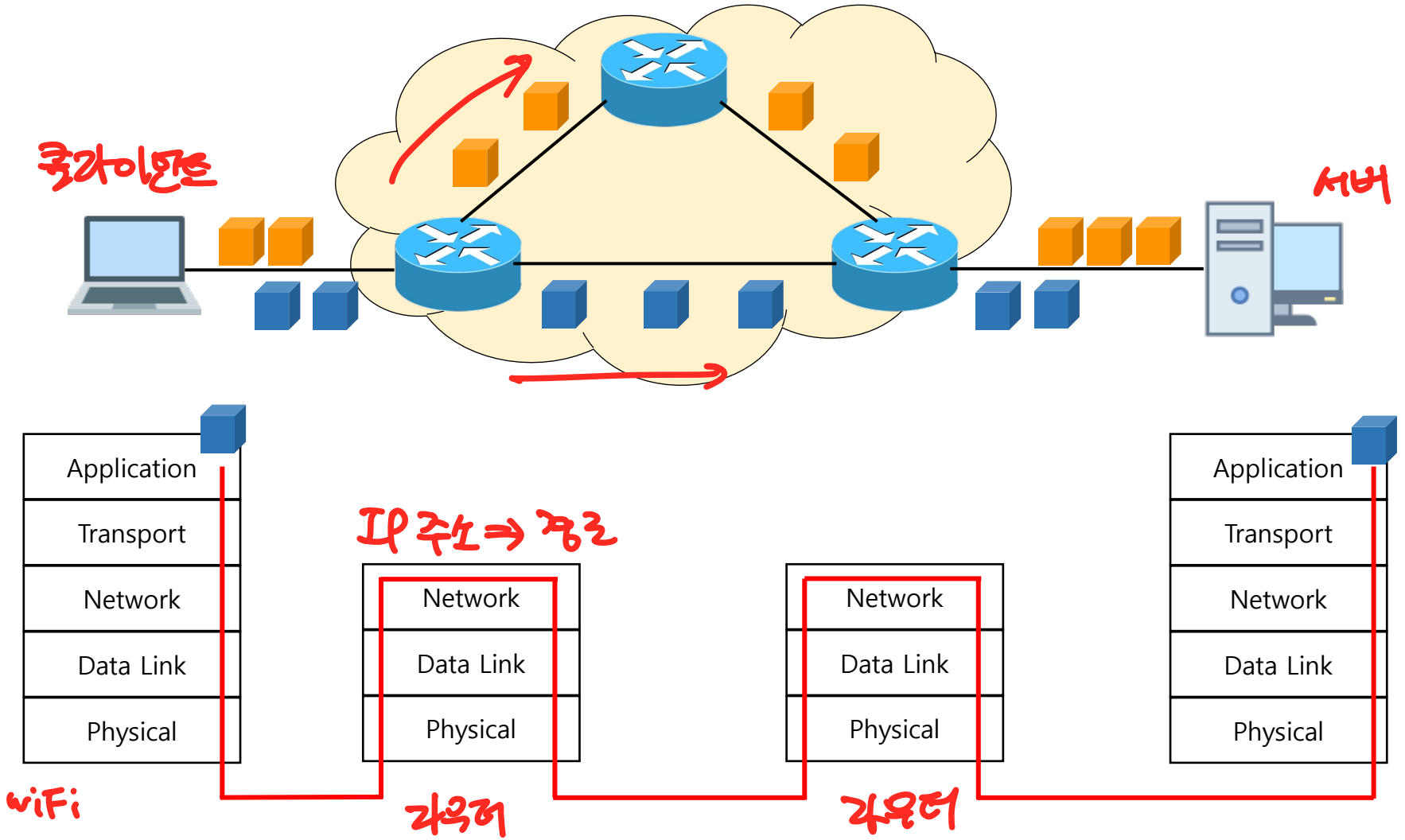
TCP/UDP

TCP/IP

# Socket API


7 Application
6 Presentation
5 Session
4 Transport
3 Network
2 Datalink
1 Physical
Application
TCP
UDP
IPv4, IPv6
Device Driver and Hardware
Socket API
Application Details
User Process
Kernel
OS
Communications Details

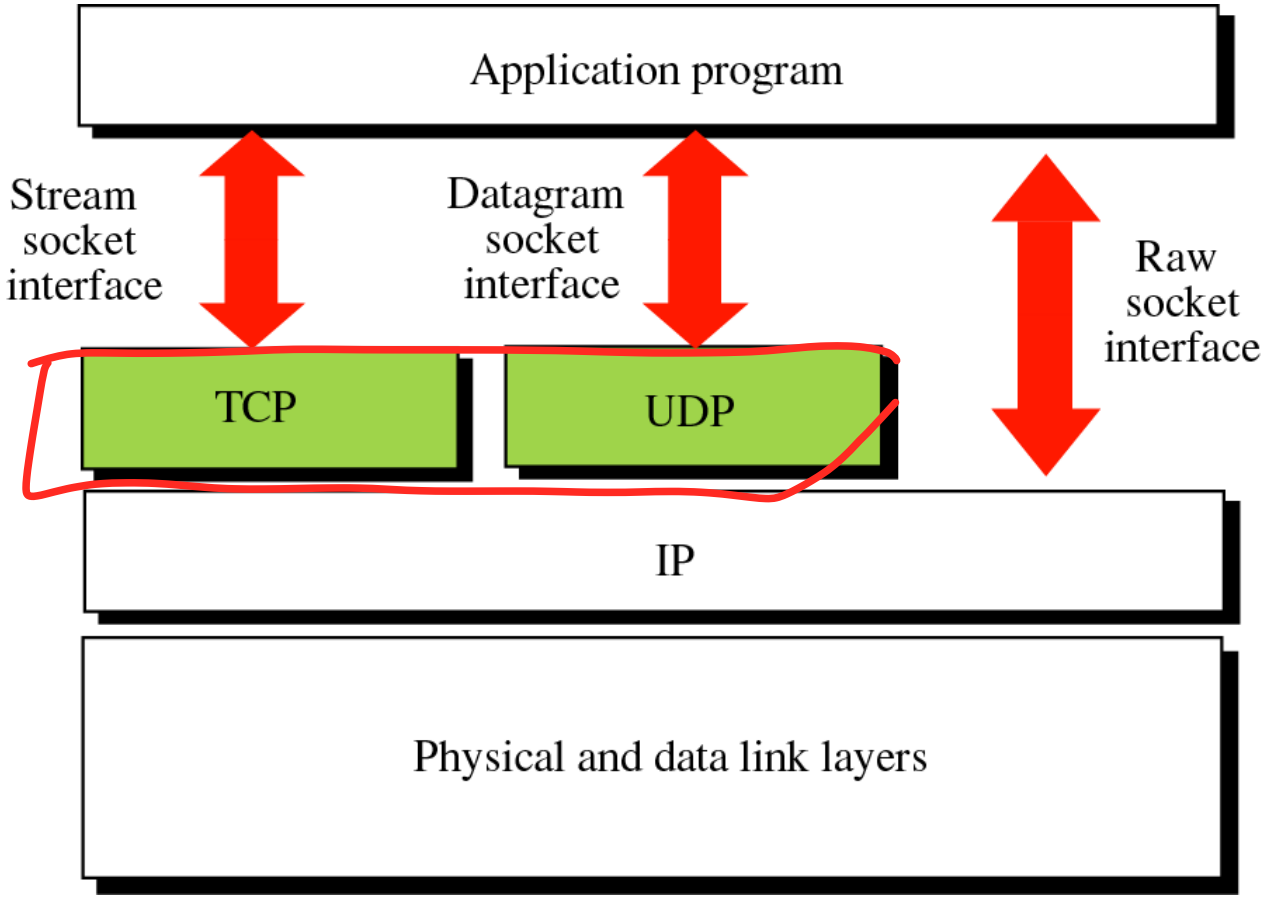
# 데이터 전송


클라이언트
서버
Application
Transport
Network
Data Link
Physical
IP 주소 ⇒ 경로
Network
Data Link
Physical
Network
Data Link
Physical
Application
Transport
Network
Data Link
Physical
WiFi
라우터
라우터

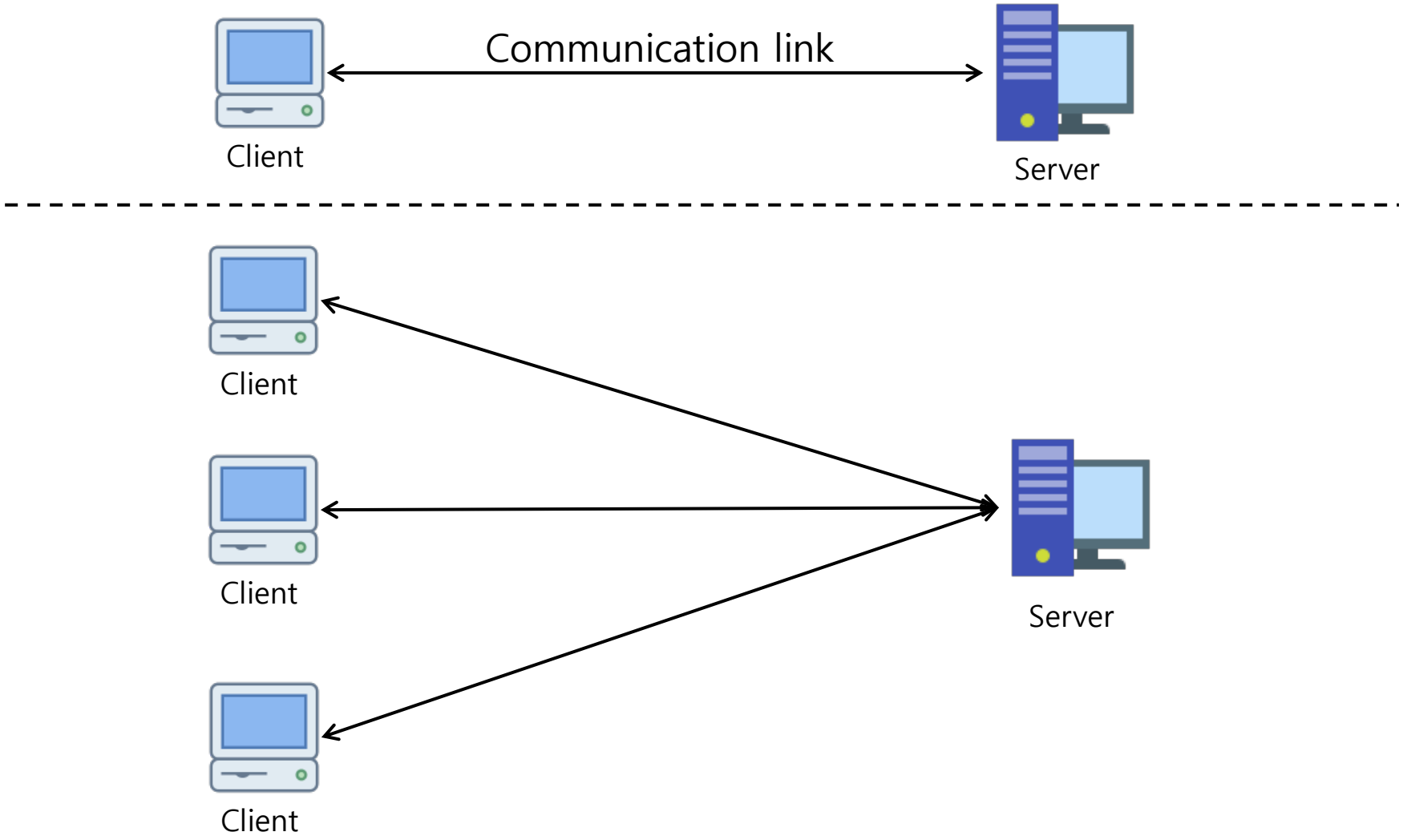
# Network (Socket) Programming

- 소켓 프로그래밍
  - 네트워킹을 위한 응용프로그램 개발
    - HTTP, SMTP, FTP, …
  - Socket API를 어떻게 사용할 것인가?
    - Application ⇔ API ⇔ TCP/IP
  - Client-Server 통신 모델
  - User-level 프로그래밍

# Socket Programming



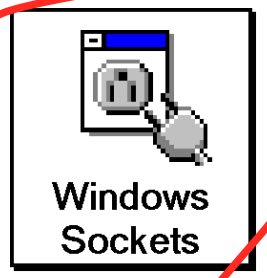
# Client-Server Model



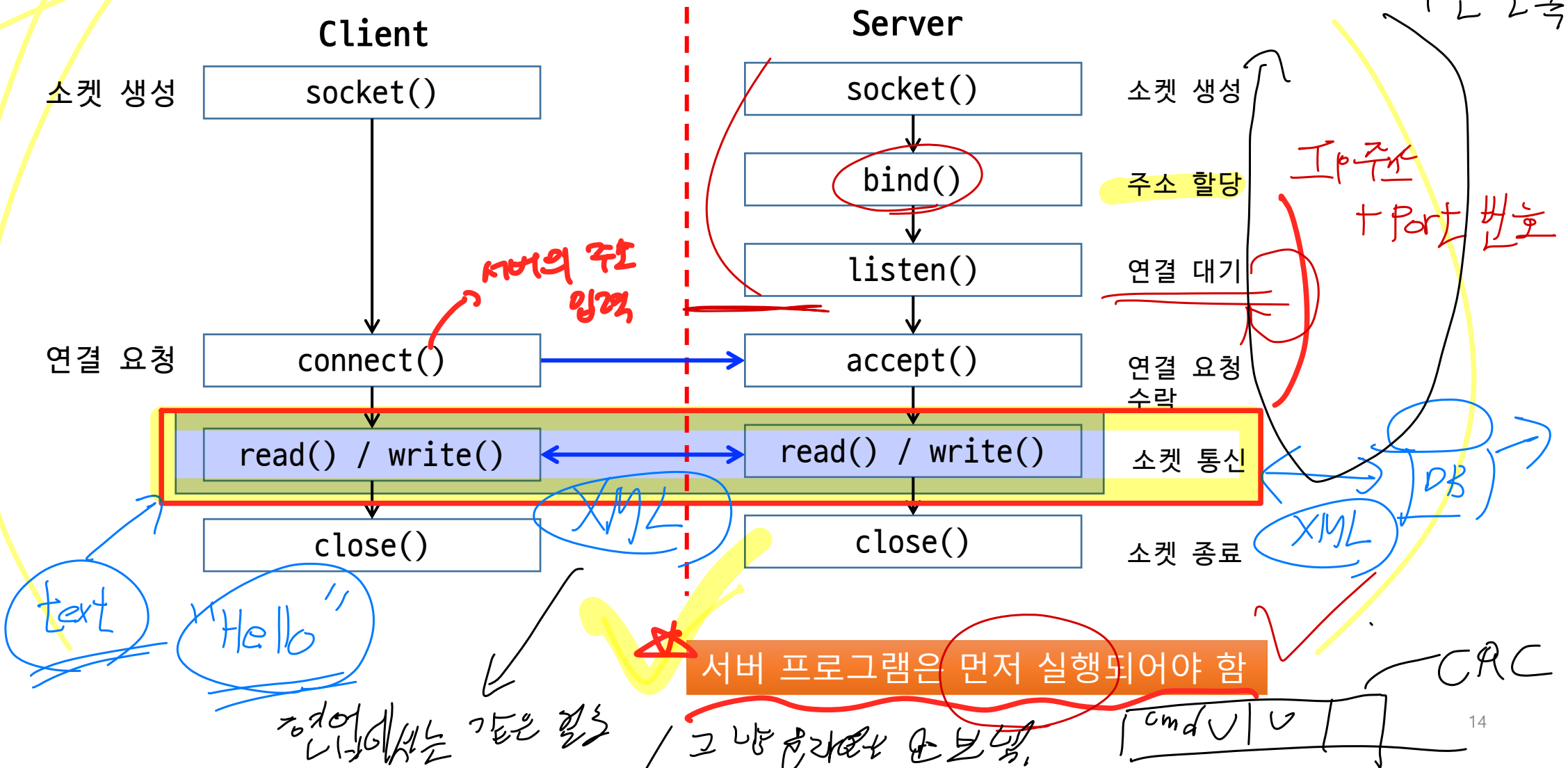
# 네트워크 프로그래밍과 소켓에 대한 이해

- 네트워크 프로그래밍이란?
  - 소켓(socket)을 기반으로 프로그래밍을 함
  - 소켓 프로그래밍이라고도 함.
  - 네트워크로 연결된 둘 이상의 컴퓨터 사이에서의 데이터 송수신 프로그램의 작성을 의미함.

- 소켓에 대한 간단한 이해
  - 네트워크(인터넷)의 연결 도구
  - 운영체제에 의해 제공이 되는 소프트웨어적인 장치
  - 소켓은 프로그래머에게 데이터 송수신에 대해 물리적, 소프트웨어적인 세부내용을 신경 쓰지 않게 한다.



# 소켓 통신 함수 순서 (TCP) : HTTP



# 소켓의 생성 (서버, 클라이언트): socket()

- 소켓의 비유와 분류
  - TCP 소켓은 전화기에 비유될 수 있음
    - 연결 설정 후 데이터 통신이 이루어짐
  - 소켓은 **socket()** 함수의 호출을 통해서 생성
    - 전화를 거는 용도의 소켓과 전화를 수신하는 용도의 소켓 생성 방법에는 차이가 있다.
  - 소켓의 생성: socket() 함수 호출

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
-> 성공 시 파일 디스크립터, 실패 시 -1 반환
```

    - domain: IPv4, IPv6, IPX, low level socket 등 설정
    - type: TCP, UDP 설정
    - protocol: 특정 프로토콜 사용을 지정 (보통 0)

# 소켓의 주소 할당 (서버): bind()

## bind() 함수

- 소켓의 주소 할당 및 연결
- 전화기에 전화번호가 부여되듯이 소켓에도 주소정보가 할당된다.
- 소켓의 주소 정보는 IP주소와 PORT번호로 구성이 된다.

```
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, socklent_t addrlen);
-> 성공 시 0, 실패 시 -1 반환
```

## struct sockaddr_in 설정

- serv_addr.sin_family = AF_INET;  // IPv4 사용
- serv_addr.sin_addr.s_addr=htonl(INADDR_ANY); // 주소 할당
  - INADDR_ANY: 소켓이 동작하는 컴퓨터의 IP 주소가 자동으로 할당
- serv_addr.sin_port = htons(atoi(argv[1])); // 포트번호 할당
  - htons: 호스트 바이트 순서를 네트워크 바이트 순서(Big endian)로 변경

# 연결 요청 대기 상태: listen()

- listen() 함수
  - 클라이언트의 연결 요청을 기다리는 상태의 소켓으로 변경
    - 걸려오는 전화를 받을 수 있는 상태
    - 클라이언트의 연결 요청을 기다림
    - 연결 요청을 받는 소켓(서버)에서만 필요한 상태

서버만 수행..

  - 전화를 거는 용도의 소켓 (클라이언트)
    - 연결요청이 가능한 상태의 소켓이 될 필요가 없음

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
-> 성공 시 0, 실패 시 -1 반환
```

int socket(); 과

대기할 큐(queue)의 개수

  - backlog: 연결 요청 대기 큐의 수
  - 소켓에 할당된 IP와 PORT번호로 연결 요청이 가능한 상태가 됨

# 연결 요청 수락 (서버): accept()

- accept() 함수
  - 연결요청을 수락하는 기능 수행
  - 걸려오는 전화에 대해서 수락의 의미로 수화기를 드는 것에 비유
  - 연결요청이 수락되어야 데이터의 송수신이 가능하다.
  - 수락된 이후에 데이터의 송수신은 양방향으로 가능
    - accept() 함수 호출 이후에 데이터의 송수신이 가능
    - 단, 연결요청이 있을 때에만 accept()함수가 반환

malloc() 동적 메모리 할당 → X → free()

적어도 이 수업에서는 안쓴다. thread에서 딱 한번 씀.

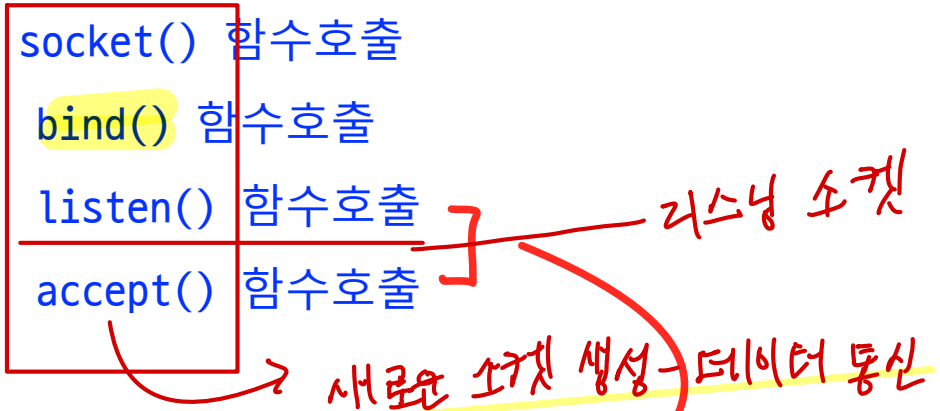
```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
-> 성공 시 파일 디스크립터, 실패 시 -1 반환
```

클라이언트의 주소 정보가 저장

왜 포인터로 쓰는지 정확히 이해하기

# 서버의 소켓 통신 정리

- 연결요청을 허용하는 소켓의 생성 과정
  - 1단계: 소켓의 생성
  - 2단계: IP주소와 Port번호의 할당
  - 3단계: 연결요청 가능상태로 변경
  - 4단계: 연결요청에 대한 수락




- 예제 hello_server.c를 통해서 함수의 호출과정 확인하기
  - 연결요청을 허용하는 프로그램을 가리켜 일반적으로 서버(Server)라 한다.
  - 서버는 연결을 요청하는 클라이언트보다 먼저 실행되어야 한다.
  - 클라이언트보다 복잡한 실행의 과정을 거친다.
  - 이렇게 생성된 소켓을 가리켜 서버 소켓 또는 리스닝 소켓이라 한다.

clnt_sock = accept(serv_sock, ...)
↑ 리스닝 소켓
↳ 데이터 통신용 소켓

# 클라이언트 소켓의 구현

- 클라이언트: 연결을 요청하는 소켓의 구현
  - 전화를 거는 상황에 비유할 수 있다.
  - 리스닝 소켓과 달리 구현의 과정이 매우 간단하다.
  - '소켓의 생성'과 '연결의 요청'으로 구분된다.
  - 연결 요청: connect() 함수

```
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, socklent_t addrlen);
-> 성공 시 0, 실패 시 -1 반환
```

int sockfd = socket(...)

서버의 IP주소 + 포트번호

서버의 IP, Port 번호

- 예제 hello_client.c를 통해서 함수의 호출과정 확인하기
  - 함수의 호출과 데이터가 실제 송수신 됨을 확인
  - 소스코드의 이해는 점진적으로...

# sockaddr, sockaddr_in 구조체 비교

- **sockaddr** 구조체: 소켓의 주소를 담는 기본 구조체 (16 bytes)

```
struct sockaddr
{
    sa_family_t sa_family;   // address family (2 bytes)
    char sa_data[14];        // IP address + Port number (14 bytes)
};
```

- **sockaddr_in** 구조체: 16 bytes

```
struct sockaddr_in {
    sa_family_t sin_family;  // 2 bytes
    uint16_t sin_port;       // 2 bytes (0~65535): Port number
    struct in_addr sin_addr; // 4 bytes: IP address
    char sin_zero[8];        // 8 bytes: 전체 크기를 16바이트로 맞추기 위함
};
struct in_addr {
    in_addr_t s_addr;  // 4 bytes
};
```

  - sockaddr 구조체의 sa_family가 AF_INET (IPv4)인 경우에 사용
  - sockaddr을 사용할 경우, sa_data에 IP주소와 Port 번호가 조합되어 각각 입력이 불편함
    - IPv4 주소 체계를 사용하기 쉽게 하기 위해 sockaddr_in 구조체를 사용

# hello_server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

void error_handling(char *message);

int main(int argc, char *argv[])
{
  int serv_sock;
  int clnt_sock;

  struct sockaddr_in serv_addr;
  struct sockaddr_in clnt_addr;
  socklen_t clnt_addr_size;

  char message[]="Hello World!";

  if(argc!=2){
    printf("Usage : %s <port>\n", argv[0]);
    exit(1);
  }

  // 1단계: 소켓 생성
  serv_sock=socket(PF_INET, SOCK_STREAM, 0);
  if(serv_sock == -1)
    error_handling("socket() error");

  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family=AF_INET;
  serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
  serv_addr.sin_port=htons(atoi(argv[1]));

  // 2단계: bind
  if(bind(serv_sock, (struct sockaddr*) &serv_addr,
          sizeof(serv_addr))==-1 )
    error_handling("bind() error");

  // 3단계: listen
  if(listen(serv_sock, 5)==-1)
    error_handling("listen() error");

  clnt_addr_size=sizeof(clnt_addr);
  // 4단계: accept
  clnt_sock=accept(serv_sock,(struct
            sockaddr*)&clnt_addr,&clnt_addr_size);
  if(clnt_sock==-1)
    error_handling("accept() error");

  // 클라이언트로 메시지 전송
  write(clnt_sock, message, sizeof(message));
  close(clnt_sock);
  close(serv_sock);
  return 0;
}

void error_handling(char *message)
{
  fputs(message, stderr);
  fputc('\n', stderr);
  exit(1);
}
```

# hello_client.c (클라이언트)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

void error_handling(char *message);

int main(int argc, char* argv[])
{
  int sock;
  struct sockaddr_in serv_addr;
  char message[30];
  int str_len;

  if(argc!=3){
    printf("Usage : %s <IP> <port>\n", argv[0]);
    exit(1);
  }

  // 1. 소켓 생성
  sock=socket(PF_INET, SOCK_STREAM, 0);
  if(sock == -1)
    error_handling("socket() error");

  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family=AF_INET;
  serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
  serv_addr.sin_port=htons(atoi(argv[2]));

  // 2. 연결 요청
  if(connect(sock, (struct sockaddr*)&serv_addr,
      sizeof(serv_addr))==-1)
    error_handling("connect() error!");

  // 3. 서버가 전송한 메시지 수신
  str_len=read(sock, message, sizeof(message)-1);
  if(str_len==-1)
    error_handling("read() error!");

  printf("Message from server: %s \n", message);
  close(sock);
  return 0;
}

void error_handling(char *message)
{
  fputs(message, stderr);
  fputc('\n', stderr);
  exit(1);
}
```

서버의 IP 주소 및 서버의 포트번호 입력 여부 확인

접속할 서버의 주소 설정

$ ./server 9190

포트번호 만

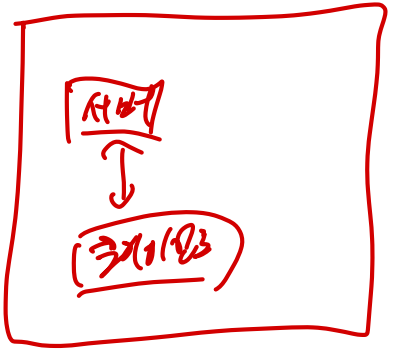
why? IP주소는 "INADDR_ANY로 자동설정"

$ ./client 서버주소 포트번호

$ ./client 127.0.0.1 9190

[0] [1] [2]

Computer




argc : 3

argv[0] : 프로그램 이름

argv[1] : 서버 IP주소

argv[2] : port 번호

# 리눅스 환경에서 실행 방법: Server

- Server 컴파일 및 실행방법
  - 항상 서버를 먼저 실행 시킴
  - 서버가 먼저 실행되어서 클라이언트의 접속을 기다림

  - Server 소스 컴파일: gcc 소스파일 –o 실행파일

```
$ gcc hello_server.c –o hserver
```

  - Server 프로그램 실행

```
$ ./hserver 9190
```

9190: 포트 번호
클라이언트의 접속 요청을 대기

➢ ./hserver: 현재 디렉토리에 있는 hserver 파일을 실행
➢ 9190: 서버가 사용하는 포트 번호

# 리눅스 환경에서 실행 방법: Client

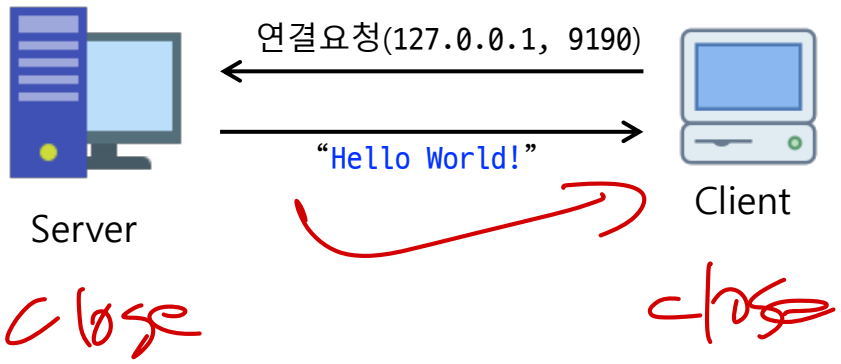
- Client 컴파일 및 실행방법
  - Client 컴파일

```
$ gcc hello_client.c –o hclient
```

  - Client 실행

```
$ ./hclient 127.0.0.1 9190
Message from server: Hello World!
```

    - 127.0.0.1: 서버의 IP 주소
      - 127.0.0.1: loopback 주소 (자기 자신을 의미)
      - 서버와 클라이언트가 동일 컴퓨터에서 동작하기 때문
    - 9190: 서버의 포트 번호



# 저 수준 파일 입출력과 파일 디스크립터

- 저 수준 파일 입출력
  - ANSI의 표준함수가 아닌, 운영체제가 제공하는 함수 기반의 파일 입출력
  - 표준이 아니기 때문에 운영체제에 대한 호환성이 없다.
  - 리눅스는 소켓도 파일로 간주
    - 저 수준 파일 입출력 함수를 기반으로 소켓 기반의 데이터 송수신이 가능함

fopen()
fread()
fwrite()

→ open(), read(), write()

(소켓)

- 파일 디스크립터
  - 운영체제가 만든 파일(그리고 소켓)을 구분하기 위한 일종의 숫자
  - 저 수준 파일 입출력 함수는 입출력을 목적으로 파일 디스크립터를 요구함
  - 저 수준 파일 입출력 함수에게 소켓의 파일 디스크립터를 전달하면,
    - 소켓을 대상으로 입출력을 진행

| 파일 디스크립터 | 대 상 |
|---|---|
| 0 | 표준입력: Standard Input |
| 1 | 표준출력: Standard Output |
| 2 | 표준에러: Standard Error |

3
4
5 ...

# 파일 열기와 닫기

- 파일 열기: open() 함수

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flag);
-> 성공 시 파일 디스크립터, 실패 시 -1 반환
```

| 열기 모드 | 의미 |
|---|---|
| O_CREAT | 필요하면 파일을 생성 |
| O_TRUNC | 기존 데이터 전부 삭제 |
| O_APPEND | 기존 데이터를 보존하고, 뒤에 이어서 저장 |
| O_RDONLY | 읽기 전용으로 파일 열기 |
| O_WRONLY | 쓰기 전용으로 파일 열기 |
| O_RDWR | 일기, 쓰기 겸용으로 파일 열기 |

- path: 파일 이름을 나타내는 문자열의 주소값 전달
- flag: 파일 열기 모드 정보 전달

- 파일 닫기: close() 함수

```
#include <unistd.h>

int close(int fd);
-> 성공 시 0, 실패 시 -1 반환
```

- fd: 닫는 파일 또는 소켓의 파일 디스크립터 (open 함수 호출 시 반환된 파일 디스크립터 이용)

# 파일에 데이터 쓰기

- 파일 쓰기: write() 함수

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);
-> 성공 시 0, 실패 시 -1 반환
```

- ssize_t : signed int
- size_t : unsigned int

- fd: 데이터를 저장할 파일 디스크립터
- buf: 전송할 데이터가 저장된 버퍼의 주소값
- nbytes: 전송할 데이터의 바이트 수

```
char buf[]="Let's go!\n";

fd = open("data.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);
if(fd==-1)
    error_handling("open() error!");

printf("file descriptor: %d \n", fd);

if(write(fd, buf, sizeof(buf)) == -1)
    error_handling("write() error!");
```

r 4
w 2
x 1

# 파일에 데이터 쓰기: low_open.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
void error_handling(char* message);

int main(void)
{
    int fd;
    int size;
    char buf[]="Let's go!\n";

    fd = open("data.txt", O_CREAT|O_WRONLY|O_TRUNC, 0644);
    if(fd == -1)
        error_handling("open() error!");

    printf("file descriptor: %d \n", fd);
    size = write(fd, buf, sizeof(buf));
    printf("write size: %d\n", size);
    if(size == -1)
        error_handling("write() error!");

    close(fd);
    return 0;
}
```

0644: 파일 권한 설정
- 6: rw
- 4: r
- 파일 소유자는 read/wirte
- 그 외: read만 가능

저수준 파일 출력 (write 함수)

```c
void error_handling(char* message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

컴파일 및 실행

```
$ gcc low_open.c -o lopen
$ ./lopen
file descriptor: 3
write size: 11
```

파일(data.txt) 내용 확인

```
$ cat data.txt
Let's go!
```

# 파일에 저장된 데이터 읽기

1. 읽은 바이트 수만큼 저장한다

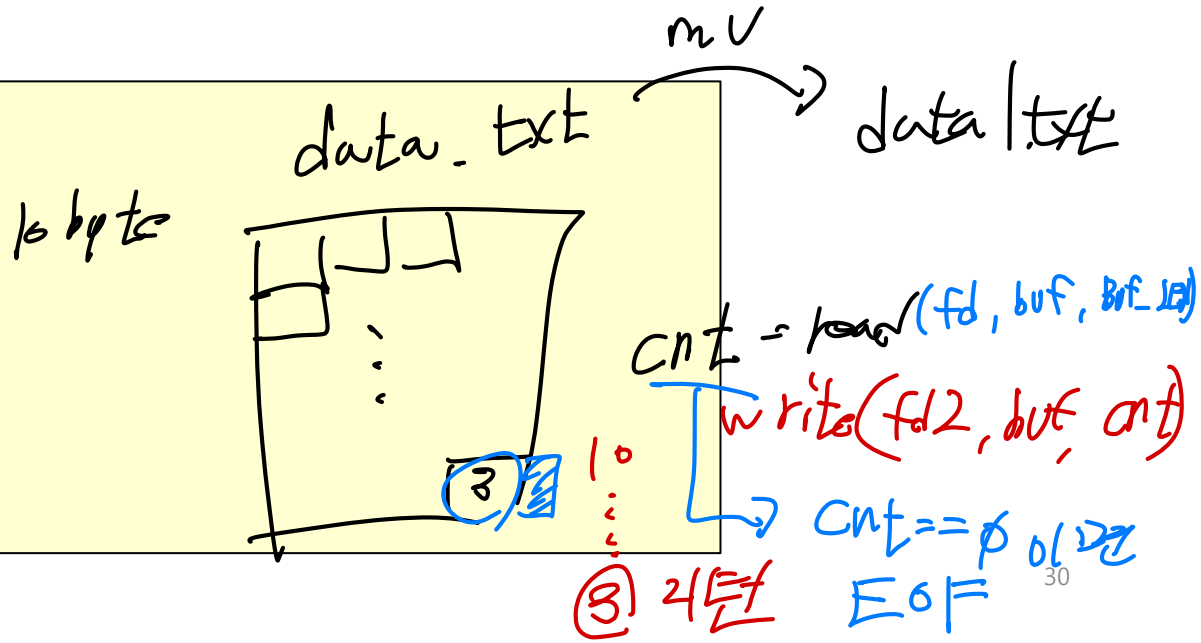
- 파일 읽기: read() 함수

```
#include <unistd.h>
ssize_t read(int fd, const void *buf, size_t nbytes);
-> 성공 시 수신한 바이트 수(단 파일의 끝을 만나면 0 리턴), 실패 시 -1 반환
```

read() 함수
- 실제 읽은 바이트 수 리턴

- fd: 데이터를 수신할 파일 디스크립터 (소켓 디스크립터)
- buf: 수신한 데이터를 저장할 버퍼의 주소
- nbytes: 수신할 최대 바이트 수

```
fd = open("data.txt", O_RDONLY);

if(fd == -1)
    error_handling("open() error!");

size = read(fd, buf, sizeof(buf));
if(size == -1)
    error_handling("read() error!");

printf("file data: %s", buf);
```



# 파일에 저장된 데이터 읽기 : low_read.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define BUF_SIZE 100
void error_handling(char* message);
int main(void)
{
    int fd;
    int size;
    char buf[BUF_SIZE];

    fd=open("data.txt", O_RDONLY);
    if(fd==-1)
        error_handling("open() error!");

    printf("file descriptor: %d \n" , fd);
    size = read(fd, buf, sizeof(buf));
    printf("read size: %d\n", size);
    //if(read(fd, buf, sizeof(buf))==-1)
    if(size == -1)
        error_handling("read() error!");
    printf("file data: %s", buf);
    close(fd);
    return 0;
}
```

저수준 파일 입력
(read 함수)

```c
void error_handling(char* message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

실행 결과

```
$ gcc low_read.c -o lread

$ ./lread
file descriptor: 3
read size: 11
file data: Let's go!
```

# 파일 디스크립터와 소켓

- 파일 디스크립터
  - 파일 생성 및 소켓 생성에 파일 디스크립터 정수 값이 할당됨
  - 순서대로 정수 값을 할당함
  - 3번부터 할당됨:0, 1, 2는 이미 할당되어 있음
  - 리눅스는 파일과 소켓을 동일하게 간주함을 확인할 수 있음

| 파일 디스크립터 | 대상 |
|---|---|
| 0 | 표준입력: stdin |
| 1 | 표준출력: stdout |
| 2 | 표준에러: stderr |

write(1, ··· )
↳ printf( )

# fd_seri.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/socket.h>

int main(void)
{
    int fd1, fd2, fd3;

    fd1=socket(PF_INET, SOCK_STREAM, 0);
    fd2=open("test.dat", O_CREAT|O_WRONLY|O_TRUNC);
    fd3=socket(PF_INET, SOCK_DGRAM, 0);

    printf("file descriptor 1: %d\n", fd1);
    printf("file descriptor 2: %d\n", fd2);
    printf("file descriptor 3: %d\n", fd3);

    close(fd1);
    close(fd2);
    close(fd3);
    return 0;
}
```

TCP 소켓

파일 open

UDP 소켓

파일 디스크립터는 순차적으로 증가

Linux 특성

파일, 소켓, 드라이버를 모두 파일로 취급.

실행 결과

```
$ gcc fd_seri.c -o fd_seri
$ ./fd_seri
file descriptor 1: 3
file descriptor 2: 4
file descriptor 3: 5
```

# htons(), ntohs()

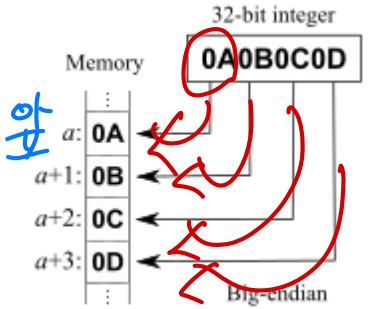
- **htons(): host to network (short type)**
  - 호스트 바이트 순서를 네트워크 바이트 순서로 변환
  - 네트워크 바이트 순서: Big endian방식 사용
  - #include <arpa/inet.h>에 정의
- **ntohs(): network to host (short type)**
  - 네트워크 바이트 순서를 호스트 바이트 순서로 변환

- 0x12345678 을 메모리에 저장하는 방식
  - Big-Endian: ARM, IBM, SPARC, Motorola

| 메모리 주소 | 0x100 | 0x101 | 0x102 | 0x103 |
|---|---|---|---|---|
| 변수값 | 0x12 | 0x34 | 0x56 | 0x78 |

  - Little Endian: Intel, ARM

| 메모리 주소 | 0x100 | 0x101 | 0x102 | 0x103 |
|---|---|---|---|---|
| 변수값 | 0x78 | 0x56 | 0x34 | 0x12 |



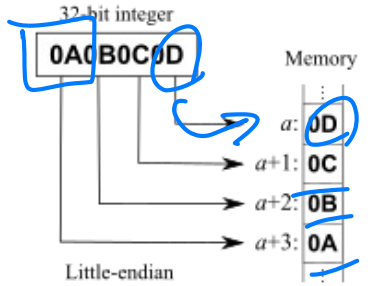
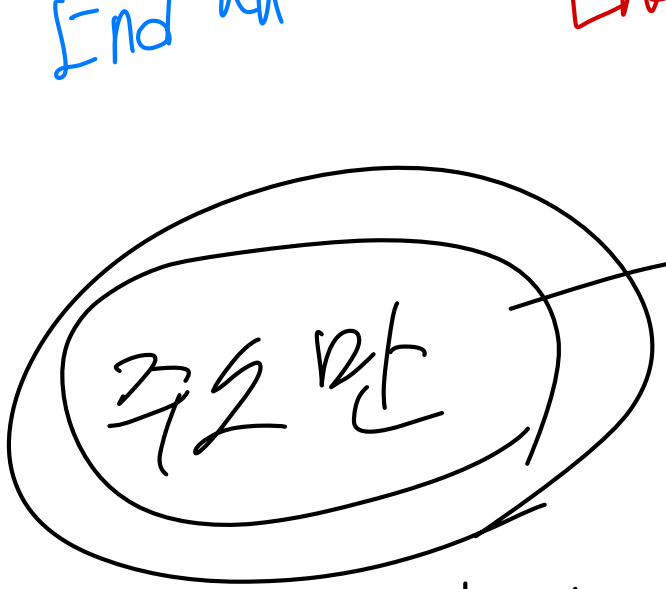




그림 참고: https://en.wikipedia.org/wiki/Endianness

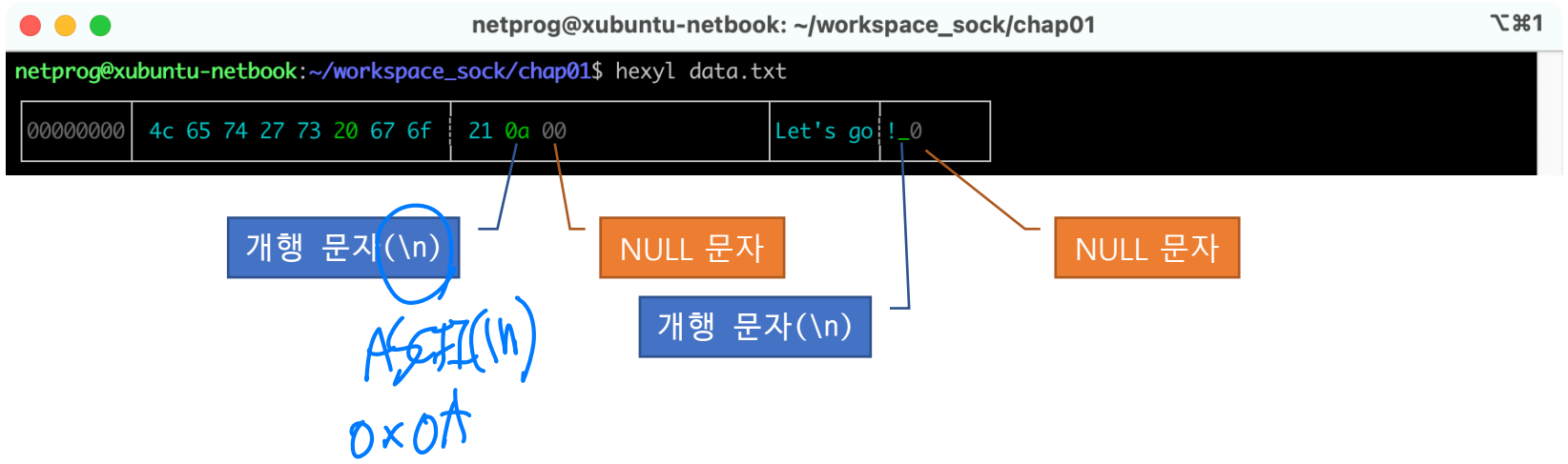

# 참고 자료




프리미엄 베이직 보내서 신경 안 써도 된다.

# 파일 내용을 16진수(hex)로 보기 #1

- hexyl 프로그램 설치 및 실행
  - $ sudo apt install hexyl
  - $ hexyl 파일이름



# 파일 내용을 16진수(hex)로 보기 #2

- hexdump 명령어

```
$ hexdump -C data.txt
00000000  4c 65 74 27 73 20 67 6f  21 0a 00         |Let's go!..|
```

- xxd 명령어

```
$ xxd data.txt
00000000: 4c65 7427 7320 676f 210a 00              Let's go!..
```

# Questions?

LMS Q&A 게시판에 올려주세요.