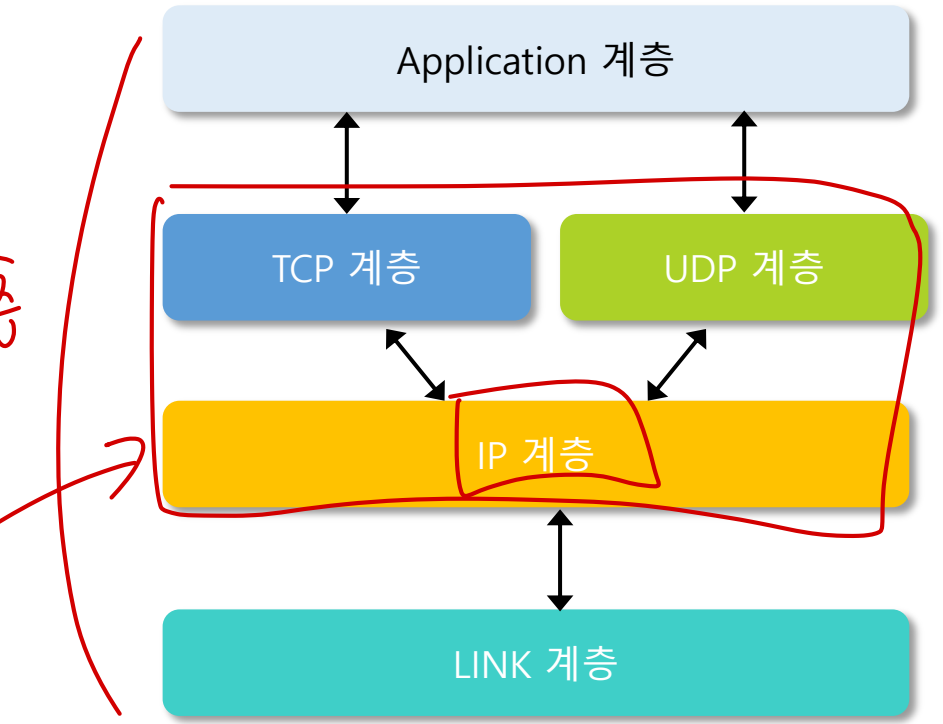
Chapter 04

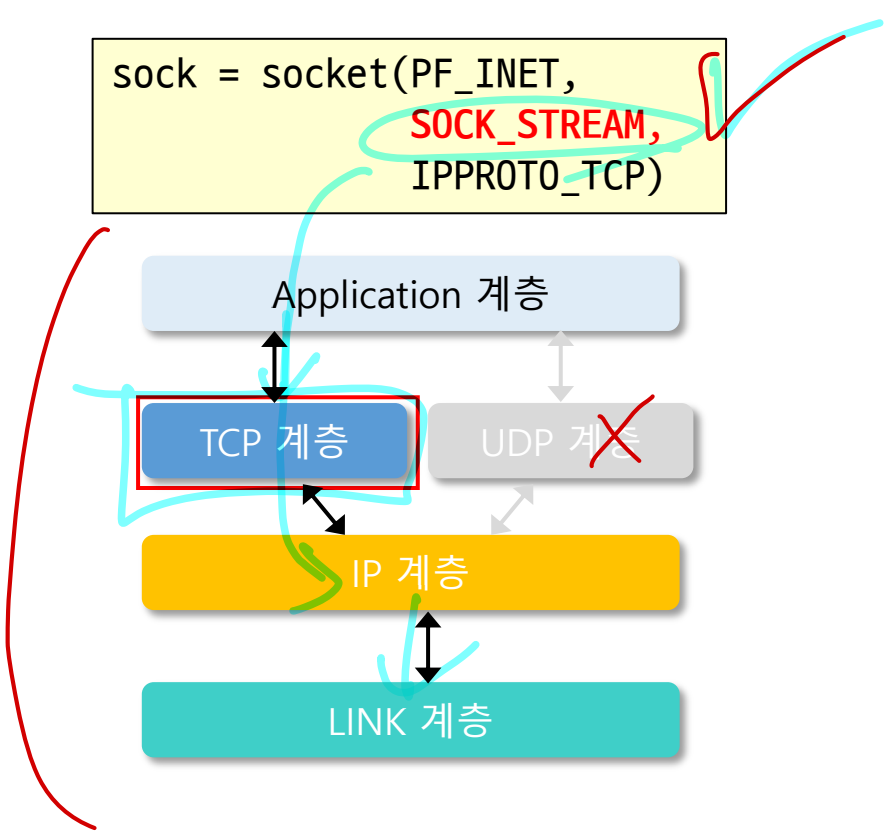
TCP 기반 서버 / 클라이언트 1

TCP/IP 프로토콜 스택

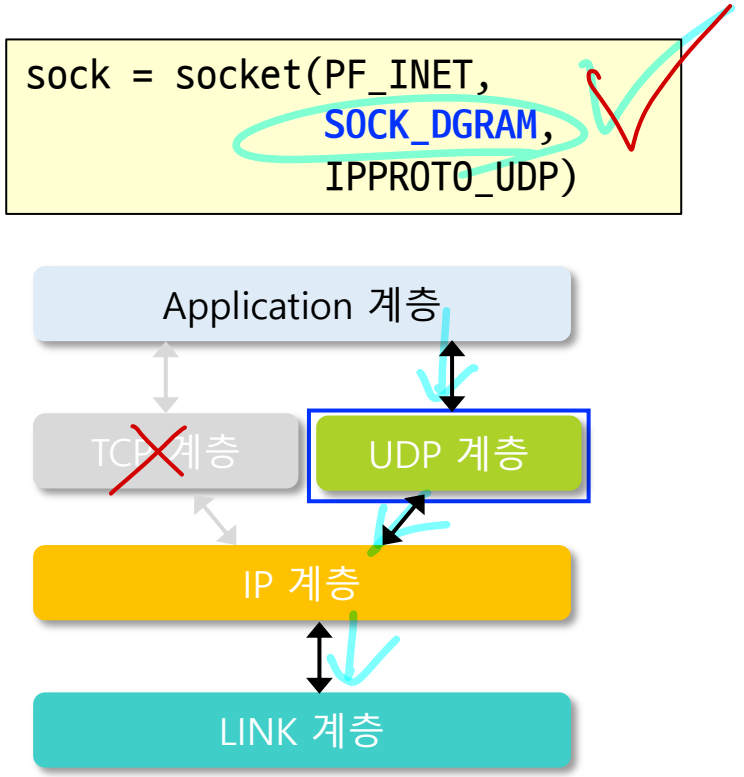
- TCP / IP 프로토콜 스택이란?
  - 인터넷 기반의 데이터 송수신을 목적으로 설계된 스택
  - 데이터 송수신의 과정을 네 개의 영역으로 계층화
  - 각 스택 별 영역을 전문화하고 표준화 함
  - 7계층으로 세분화가 되며, 4계층으로도 표현함



TCP 소켓과 UDP 소켓의 스택 FLOW



TCP 소켓의 스택 FLOW

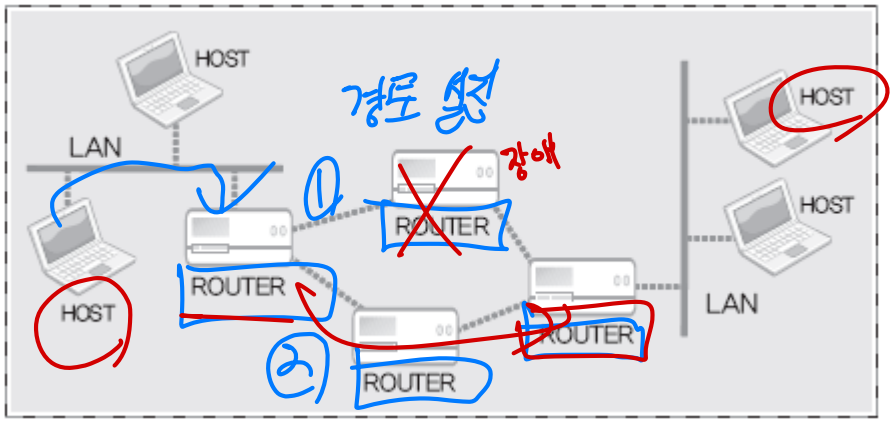


UDP 소켓의 스택 FLOW

LINK & IP계층

- LINK 계층의 기능 및 역할 (TCP/IP 프로토콜)
  - 물리적인 영역의 표준화 결과
  - LAN, WAN, MAN과 같은 물리적인 네트워크 표준 관련 프로토콜이 정의된 영역
  - 아래의 그림과 같은 물리적인 연결의 표준이 된다.

- IP 계층의 기능 및 역할
  - IP는 Internet Protocol을 의미
  - 경로 설정과 관련이 있는 프로토콜
    - 목적지로 데이터를 전송하기 위해 어떤 경로를 거쳐갈 것인가?
  - 오류 발생에 대한 해결 방법이 없음

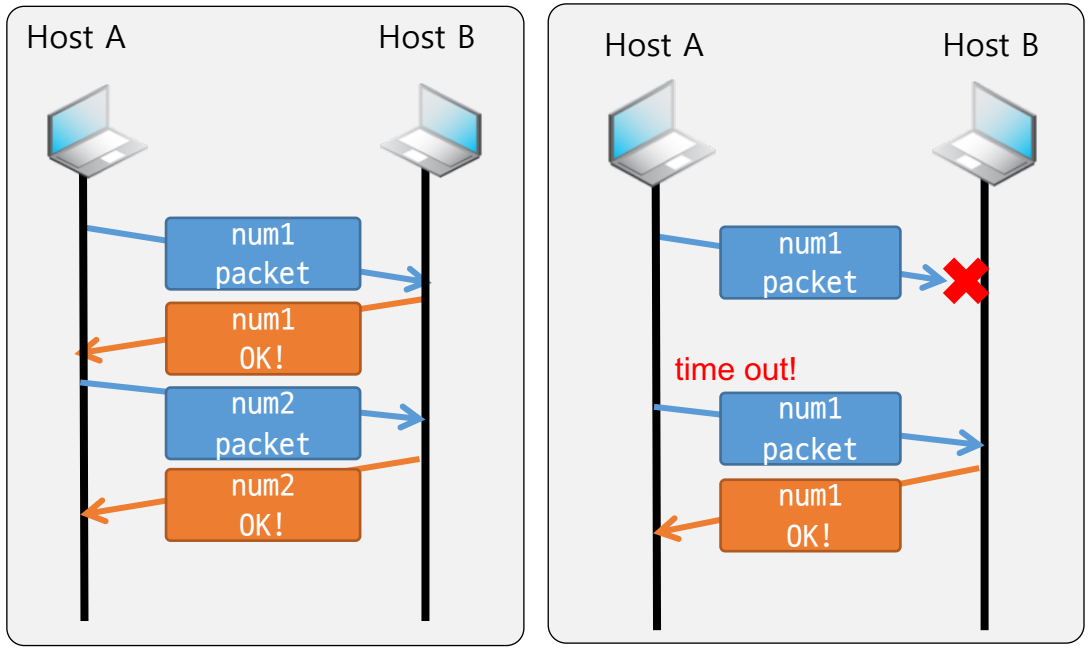


TCP/UDP 계층 : 전송 계층

- TCP/UDP 계층의 기능 및 역할
  - 실제 데이터의 송수신과 관련 있는 계층: 전송(Transport) 계층이라고도 함 포트 번호
  - TCP는 데이터의 전송을 보장하는 프로토콜(신뢰성 있는 프로토콜)
    - TCP는 UDP에 비해 복잡
    - TCP는 데이터 전송 중 확인 과정을 거침 (신뢰성 보장)
    - 데이터가 전달되지 못하면, 일정 시간 이후 재전송함
  - UDP는 신뢰성을 보장하지 않는 프로토콜

고속의 데이터 전송이 UDP를 많이 쓴다.

TCP 데이터 전송과정



Application 계층

- Application 계층
  - 프로그래머에 의해서 완성되는 계층
  - 응용프로그램의 프로토콜을 구성하는 계층
  - 소켓을 기반으로 프로토콜의 설계 및 구현
  - 소켓을 생성하면, 앞서 보인 LINK, IP, TCP/UDP 계층에 대한 내용은 감춰진다.
  - 응용 프로그래머는 Application 계층의 완성도에 집중하게 된다.

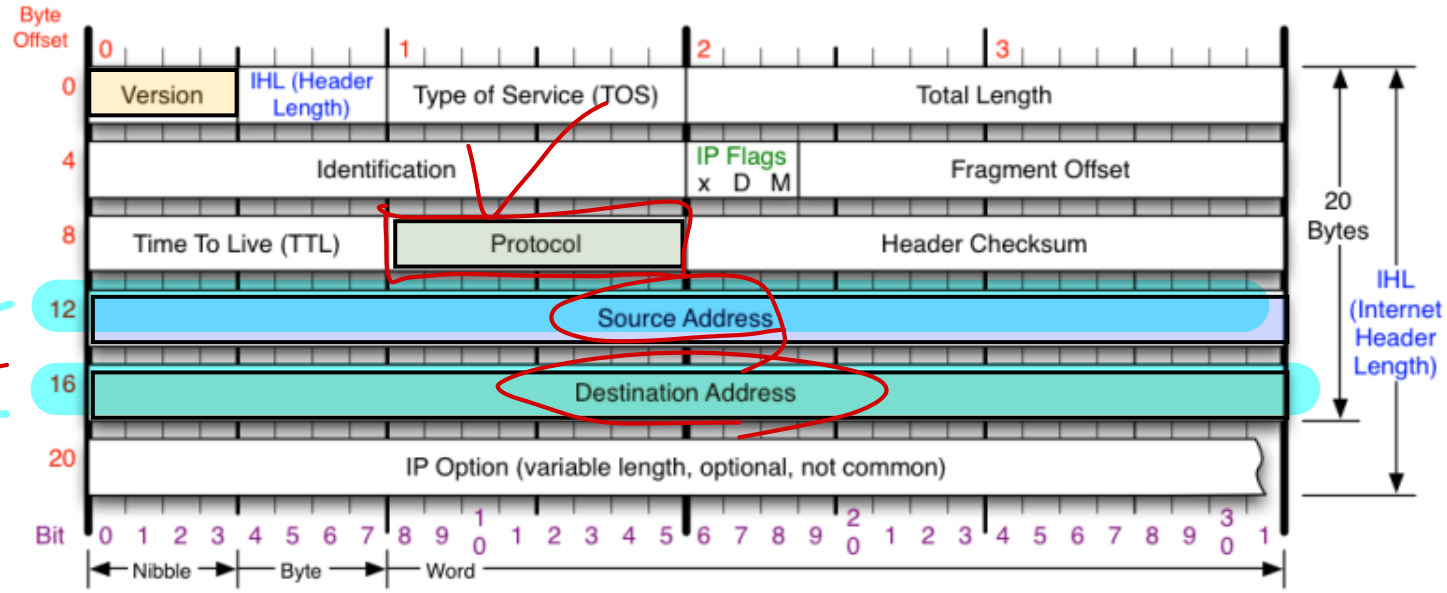
TCP Header

TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port														
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)														
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

- **Source port** (송신측 포트번호)
  - 설정하지 않을 경우, 랜덤 번호로 자동 할당됨 (hello_client.c)
- **Destination port** (수신측 포트번호)
- Sequence number
  - 전송하는 데이터의 순서를 표시
  - 수신측은 데이터의 순서를 파악하고 재조립함
- Acknowledgement number
  - 데이터를 받은 수신자가 다음 시퀀스 번호를 할당해서 전송
- Flags (NS ~ FIN)
  - SYN: 연결 시작을 위해 사용
  - FIN: 상대방과의 연결 종료를 요청 : close()
  - PSH: 송수신 데이터가 버퍼에 다 찰 때까지 기다리지 않고 즉시 전송(Push)

IP Header



Version

Version of IP Protocol. 4 and 6 are valid. This diagram represents version 4 structure only.

Header Length

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

Protocol

IP Protocol ID. Including (but not limited to):

1 ICMP	17 UDP	57 SKIP
2 IGMP	47 GRE	88 EIGRP
6 TCP	50 ESP	89 OSPF
9 IGRP	51 AH	115 L2TP

Total Length

Total length of IP datagram, or IP fragment if fragmented. Measured in Bytes.

Fragment Offset

Fragment offset from start of IP datagram. Measured in 8 byte (2 words, 64 bits) increments. If IP datagram is fragmented, fragment size (Total Length) must be a multiple of 8 bytes.

Header Checksum

Checksum of entire IP header

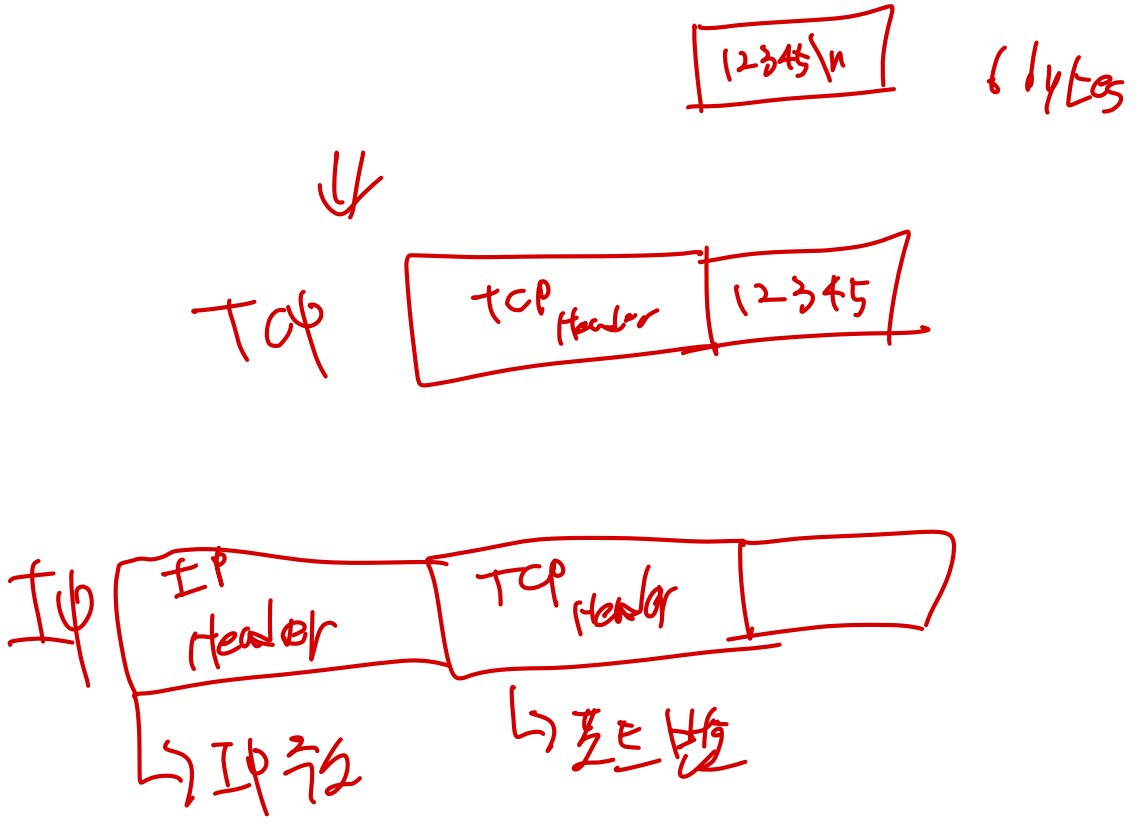
IP Flags

x D M

x 0x80 reserved (evil bit)
D 0x40 Do Not Fragment
M 0x20 More Fragments follow

RFC 791

Please refer to RFC 791 for the complete Internet Protocol (IP) Specification.


12345\n
6 bytes
TCP
TCP Header
12345
IP
IP Header
TCP Header
IP 头
TCP 头

TCP 서버의 기본적인 함수호출 순서

bind() 함수까지 호출이 되면 주소가 할당된 소켓을 얻음

listen() 함수의 호출을 통해서 연결요청이 가능한 상태가 됨

이번 단원에서는 listen 함수 호출이 의미하는 바에 대해서 주로 학습함.

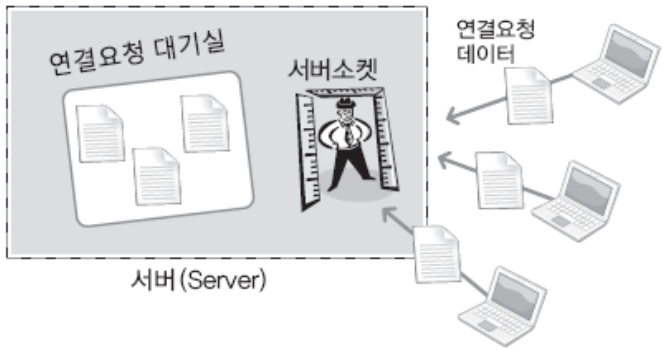
연결요청 대기 상태로의 진입

- listen() 함수

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

int listen(int sock, int backlog);
-> 성공 시 0, 실패 시 -1 반환
```

➢ sock
  - 클라이언트의 연결 요청을 처리하기 위해 대기상태에 두는 소켓 디스크립터
  - 서버 소켓(리스닝 소켓)

➢ backlog
  - 연결 요청 대기 큐(queue)의 크기
  - 큐의 크기가 5로 설정을 하면, 클라이언트의 연결 요청을 5개까지 대기시킬 수 있음



- 연결 요청도 일종의 데이터 전송임
- 연결 요청을 받아 들이기 위해 하나의 소켓이 필요함
- 이 소켓을 서버 소켓 또는 리스닝 소켓이라고 함
- listen() 함수 호출은 소켓을 리스닝 소켓이 되게 함

socket.h 파일 확인

- socket.h 파일 찾기 (Ubuntu 기준)

```
$ find /usr/include -name socket.h
/usr/include/linux/socket.h
/usr/include/asm-generic/socket.h
/usr/include/x86_64-linux-gnu/bits/socket.h
/usr/include/x86_64-linux-gnu/asm/socket.h
/usr/include/x86_64-linux-gnu/sys/socket.h
```

- listen()의 backlog값 확인
  - /usr/include/x86_64-linux-gnu/bits/socket.h에 정의되어 있음
    - #define SOMAXCONN = 4096 (리눅스 시스템에 따라 다름)

```
171 /* Maximum queue length specifiable by listen.  */
172 #define SOMAXCONN   4096
```

  - /sbin/sysctl –a ¦ grep somaxconn 명령어로 확인 가능
    - Ubuntu 20.04 버전 기준

```
$ sudo /sbin/sysctl -a ¦ grep somaxconn
[sudo] password for xxx:
net.core.somaxconn = 4096
```

PF_INET vs. AF_INET 비교

- PF (Protocol Family)
  - 프로토콜 체계 중 하나
- AF (Address Family)
  - 주소 체계 중 하나
  - PF_INET과 AF_INET 은 같은 상수값을 가짐

IP v4

<socket.h>

```
40 /* Protocol families.  */
41 #define PF_UNSPEC   0   /* Unspecified.  */
42 #define PF_LOCAL    1   /* Local to host (pipes and file-domain).  */
43 #define PF_UNIX     PF_LOCAL /* POSIX name for PF_LOCAL.  */
44 #define PF_FILE     PF_LOCAL /* Another non-standard name for PF_LOCAL.  */
45 #define PF_INET     2   /* IP protocol family.  */
46 #define PF_AX25     3   /* Amateur Radio AX.25.  */
47 #define PF_IPX      4   /* Novell Internet Protocol.  */
48 #define PF_APPLETALK    5   /* Appletalk DDP.  */
```

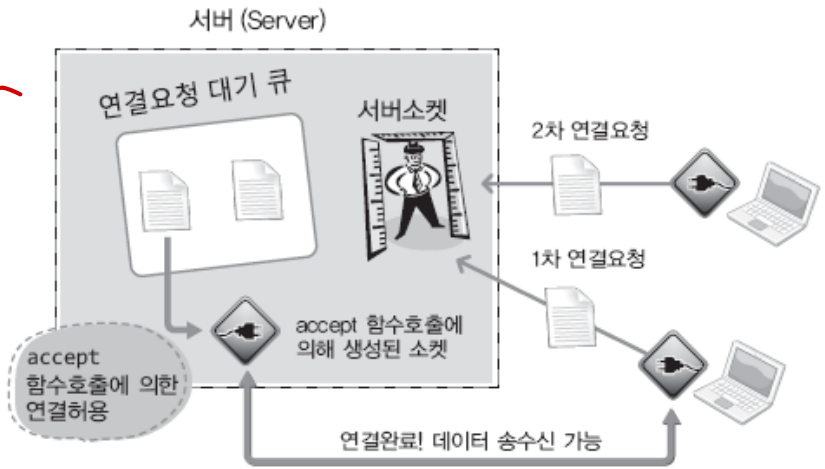
```
91 /* Address families.  */
92 #define AF_UNSPEC   PF_UNSPEC
93 #define AF_LOCAL    PF_LOCAL
94 #define AF_UNIX     PF_UNIX
95 #define AF_FILE     PF_FILE
96 #define AF_INET     PF_INET 2
97 #define AF_AX25     PF_AX25
98 #define AF_IPX      PF_IPX
99 #define AF_APPLETALK    PF_APPLETALK
```

서버: 클라이언트의 연결요청 수락(accept)

- accept() 함수

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

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

- sock: 서버 소켓의 파일 디스크립터 전달
- addr
  - 연결 요청을 한 클라이언트의 주소 정보를 담을 변수의 주소
  - accept() 함수 호출 후 addr 변수에 클라이언트의 주소 정보가 저장됨
- addrlen
  - 두 번째 매개변수 addr의 크기(바이트 단위)
  - 크기 정보를 addrlen에 저장한 다음, 변수의 주소 값을 전달



- accept() 함수는 연결 요청 정보를 참조하여 클라이언트와 통신을 위한 별도의 소켓을 생성함
- 이렇게 생성된 소켓을 이용하여 데이터 송수신이 진행됨

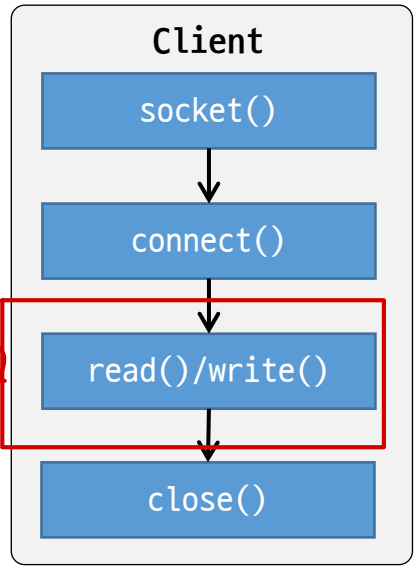
TCP 클라이언트의 기본적인 함수호출 순서

- connect() 함수

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

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

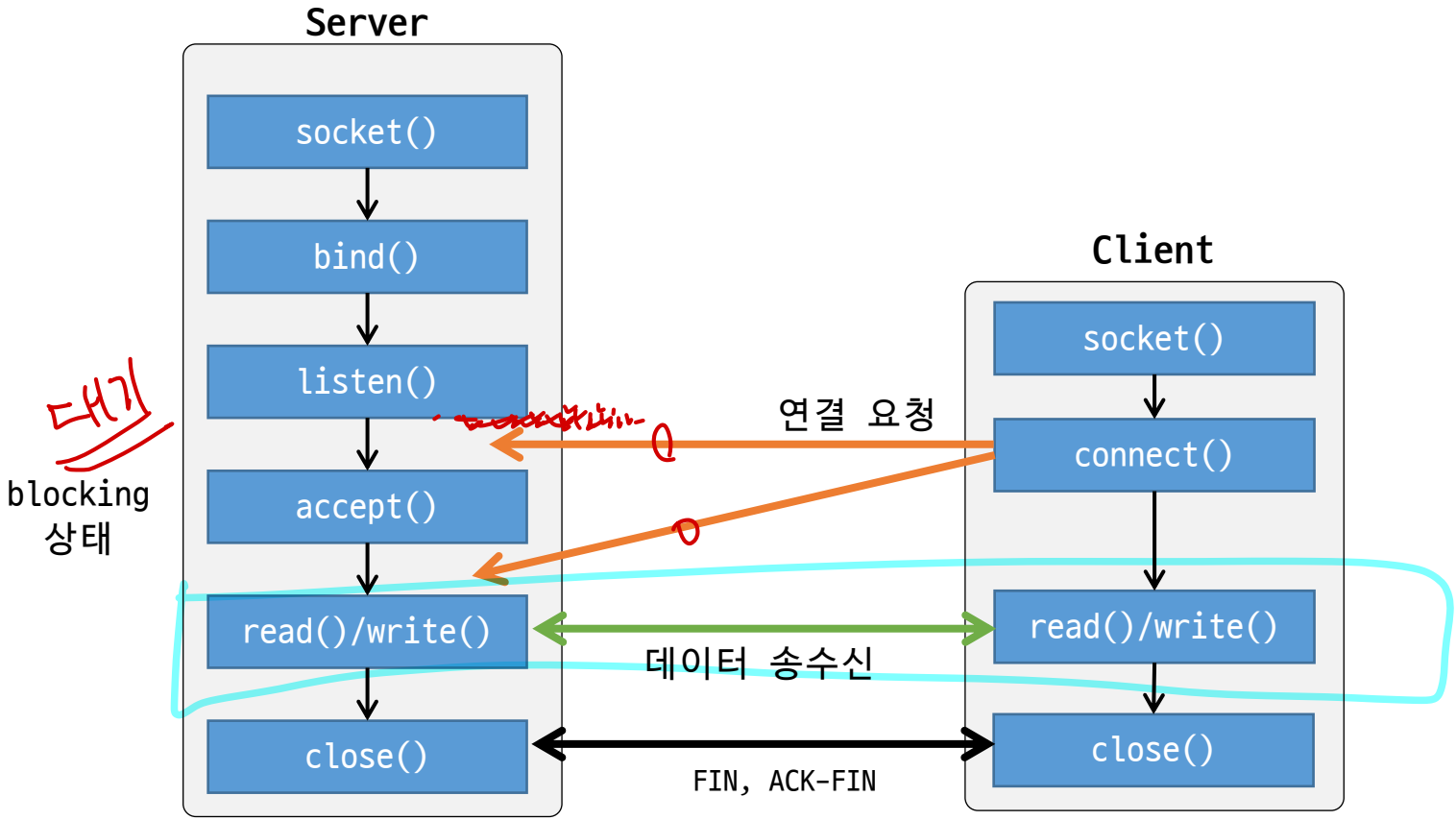
서버주소

- sock: 클라이언트 소켓의 파일 디스크립터
- serv_addr: 연결 요청할 서버의 주소 정보를 담을 변수의 주소 전달
- addrlen: sev_addr의 크기



- 클라이언트의 경우 소켓을 생성한 다음, 서버와 연결을 위해 connect() 함수 호출
- connect() 함수를 호출할 때, 서버의 주소 정보 전달

TCP 기반 서버, 클라이언트의 함수호출 관계



- 서버의 listen() 함수호출 이후에 클라이언트의 connect() 함수 호출이 유효함

hello_server.c

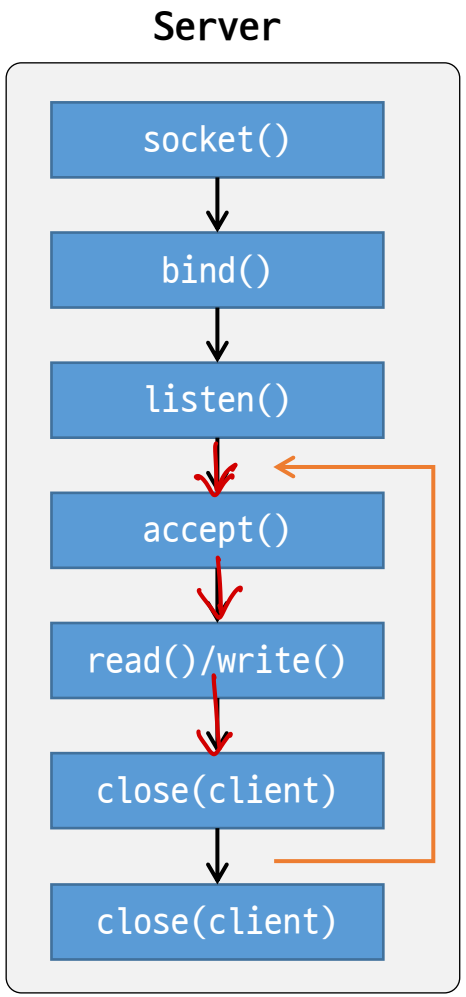
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

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

    /* 주소 정보 초기화 */
    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() 주소 정보 할당
    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");

    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

```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);
    }

    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]));

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

    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);
}
```

Iterative 서버의 구현

- Iterative 서버
  - 여러 클라이언트의 연결 요청 수락을 위해 어떤 방식으로 서버를 구현해야 될 것인가?



- 서버가 반복적으로 accept() 함수 호출
  - ✓ 계속된 클라이언트의 연결 요청을 수락할 수 있음
- 동시에 둘 이상의 클라이언트에게 서비스를 제공할 수 없음
- 기존 클라이언트 소켓을 close()하고 다른 클라이언트를 accept()
  - ✓ 한 순간에 하나의 클라이언트에게만 서비스 제공

Iterative 서버와 클라이언트의 일부

echo_server.c
코드 일부

```
for(i=0; i<5; i++)
{
    clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
    if(clnt_sock==-1)
        error_handling("accept() error");
    else
        printf("Connected client %d, client_sock: %d \n", i+1, clnt_sock);

    while((str_len=read(clnt_sock, message, BUF_SIZE))!=0)
        write(clnt_sock, message, str_len);

    close(clnt_sock);
    printf("close client socket: %d\n", clnt_sock);
}
```

str_len != 0

str_len == 0 이면 종료

echo_client.c
코드 일부

```
while(1)
{
    fputs("Input message(Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
        break;

    write(sock, message, strlen(message));
    str_len=read(sock, message, BUF_SIZE-1);
    message[str_len]=0;
    printf("Message from server: %s", message);
}
```

NULL 추가

Iterative 서버: echo_server.c #1

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

#define BUF_SIZE 1024
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    char message[BUF_SIZE];
    int str_len, i;
    struct sockaddr_in serv_adr;
    struct sockaddr_in clnt_adr;
    socklen_t clnt_adr_sz;
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    if(serv_sock==-1)
        error_handling("socket() error");

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

Iterative 서버: echo_server.c #2

```
    if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");

    if(listen(serv_sock, 5)==-1)
        error_handling("listen() error");

    clnt_adr_sz=sizeof(clnt_adr);
    for(i=0; i<5; i++)
    {
        clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);
        if(clnt_sock == -1)
            error_handling("accept() error");
        else
            printf("Connected client %d, client_sock: %d \n", i+1, clnt_sock);

        while((str_len = read(clnt_sock, message, BUF_SIZE))!=0)
            write(clnt_sock, message, str_len);

        close(clnt_sock);
        printf("close client socket: %d\n", clnt_sock);
    }
    close(serv_sock);
    return 0;
}

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

backlog=5
- 5개 클라이언트의 연결 요청을 대기

5개 클라이언트의 연결을 허용

Server는 수신한 데이터를 그대로 클라이언트에게 전달(echo)

Client가 접속을 종료하면 read()함수에서 0을 리턴함

echo_client.c #1

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

#define BUF_SIZE 1024
void error_handling(char *message);
int main(int argc, char *argv[])
{
    int sock;
    char message[BUF_SIZE];
    int str_len;
    struct sockaddr_in serv_adr;

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

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

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

echo_client.c #2

```
    if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
        error_handling("connect() error!");
    else
        puts("Connected.....");

    while(1)
    {
        fputs("Input message(Q to quit): ", stdout);
        fgets(message, BUF_SIZE, stdin);
        if(!strcmp(message,"q\n") || !strcmp(message,"Q\n"))
            break;

        write(sock, message, strlen(message));
        str_len=read(sock, message, BUF_SIZE-1);
        message[str_len]=0;
        printf("Message from server: %s", message);
    }
    close(sock);
    return 0;
}

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

Client가 입력한 값을 그대로 수신함 (echo)

echo_server, echo_client 실행

<echo_server>: 하나의 클라이언트에게만 서비스 수행

```
$ gcc echo_server.c –o eserver
$ ./eserver 9190

Connected client 1 , client_sock: 4
close client socket: 4
Connected client 2, client_sock: 4
```

Client #1이 연결 종료된 후
Cleint #2가 Server에 접속이 허용됨
- 동일한 socket_fd를 가짐

② 클라이언트 #2 접속 허용 (accept)

<echo_client #1>

```
$ gcc echo_client.c –o eclient
$ ./echo_client 127.0.0.1 9190
Connected..........
Input message(Q to quit): Hello
Message from server: Hello
Input message(Q to quit): Go
Message from server: Go
Input message(Q to quit): Q
```

① 클라이언트 #1 종료('Q' 입력)

<echo_client #2>

```
$ ./echo_client 127.0.0.1 9190
Connected..........
Input message(Q to quit): Hello
Message from server: Hello
Input message(Q to quit): hi
Message from server: hi
Input message(Q to quit):
```

- client #1이 "q"를 입력하면 연결이 종료되고
- client #2가 server로 부터 메시지를 수신함

에코 클라이언트의 문제점

- 문제점
  - 정상 동작은 하지만 문제의 발생 소지가 있는 코드

```
write(sock, message, strlen(message));

str_len=read(sock, message, BUF_SIZE-1);
message[str_len]=0;
printf("Message from server: %s", message);
```

  - TCP 데이터 송수신에는 경계가 존재하지 않음
    - 위의 코드는 한 번의 read() 함수 호출로 앞서 전송된 문자열 전체를 읽을 수 있다고 가정
    - 서버가 전송한 문자열의 일부분만 읽혀질 수도 있음
  - 전송할 데이터의 크기가 큰 경우
    - 운영체제(커널)는 내부적으로 여러 조각으로 나누어서 클라이언트에게 전송(MTU)

Questions?

LMS Q&A 게시판에 올려주세요.