

네트워크 프로그래밍

06. TCP 기반 서버/클라이언트



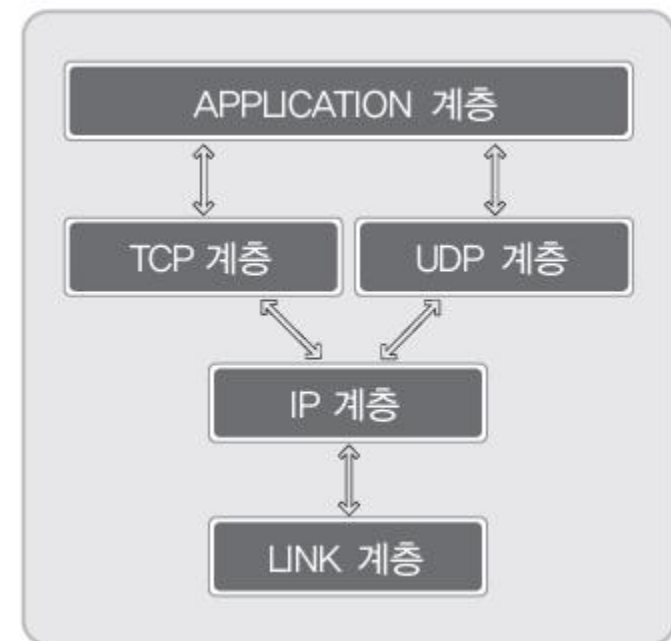
# TCP와 UDP에 대한 이해

# TCP/IP 프로토콜 스택

3

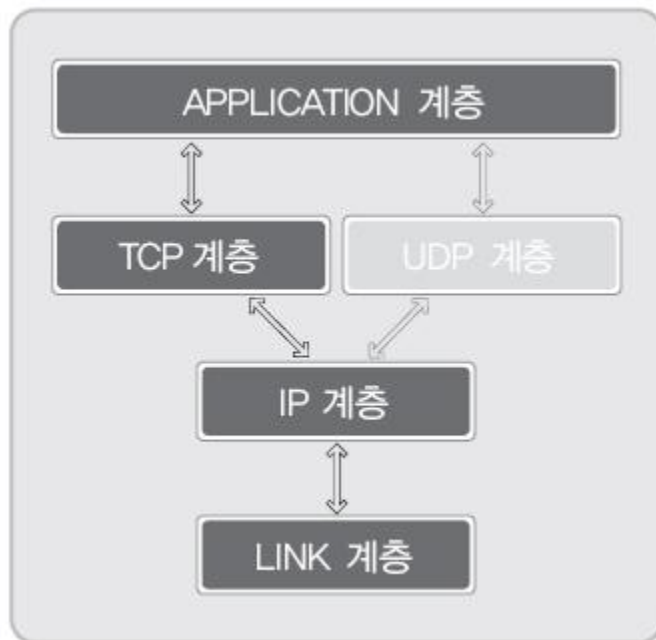
## □ TCP / IP 프로토콜 스택이란?

- 인터넷 기반의 데이터 송수신을 목적으로 설계된 스택
- 큰 문제를 작게 나눠서 계층화 한 결과
- 데이터 송수신의 과정을 네 개의 영역으로 계층화 한 결과
- 각 스택 별 영역을 전문화하고 표준화 함
- 7계층으로 세분화가 되며, 4계층으로도 표현함

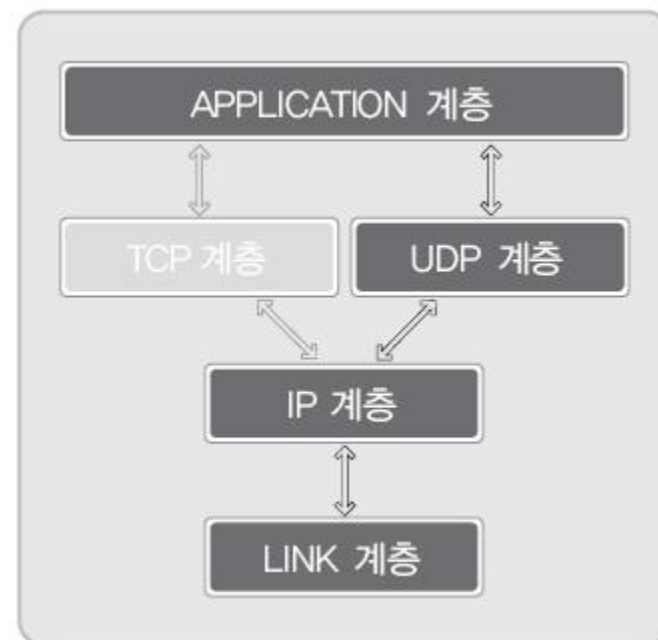


# TCP 소켓과 UDP 소켓의 스택 FLOW

4



TCP 소켓의 스택 FLOW

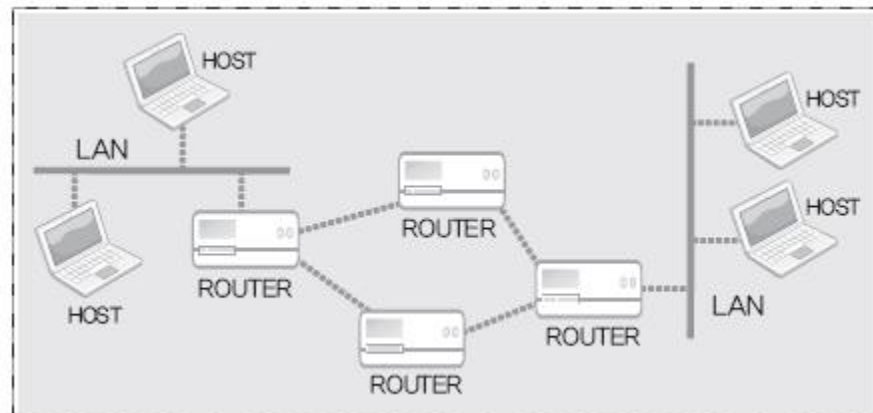


UDP 소켓의 스택 FLOW

# LINK & IP계층

5

- LINK 계층의 기능 및 역할
  - ▣ 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 의미함
  - ▣ 경로의 설정과 관련이 있는 프로토콜

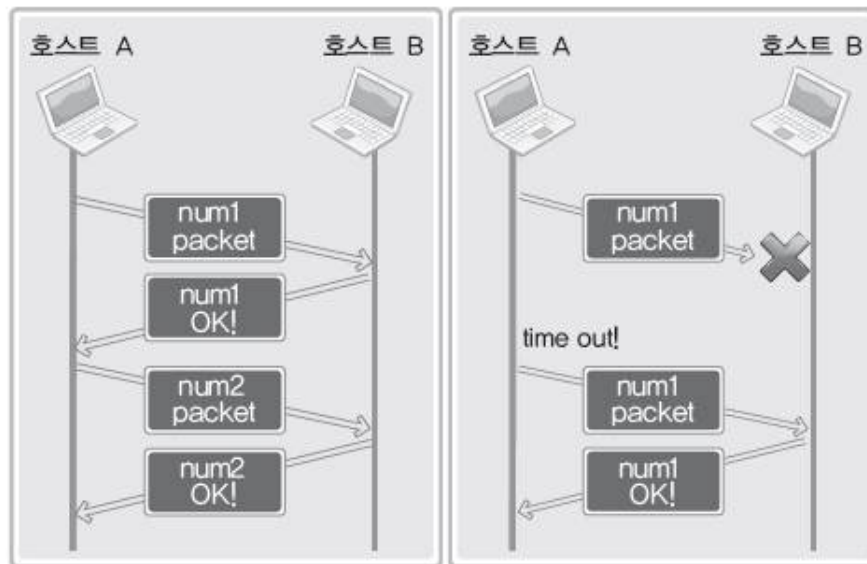


# TCP/UDP 계층

6

## □ TCP/UDP 계층의 기능 및 역할

- 실제 데이터의 송수신과 관련 있는 계층
- 그래서 전송(Transport) 계층이라고도 함
- TCP는 데이터의 전송을 보장하는 프로토콜(신뢰성 있는 프로토콜), UDP는 보장하지 않는 프로토콜
- TCP는 신뢰성을 보장하기 때문에 UDP에 비해 복잡한 프로토콜이다.



TCP는 왼쪽의 그림에서 보이듯이 확인의 과정을 거친다. 때문에 신뢰성을 보장하지만, 그만큼 복잡한 과정을 거쳐서 데이터의 전송이 이뤄진다.

# APPLICATION 계층

7

- 프로그래머에 의해서 완성되는 APPLICATION 계층
  - ▣ 응용프로그램의 프로토콜을 구성하는 계층
  - ▣ 소켓을 기반으로 완성하는 프로토콜을 의미함
  - ▣ 소켓을 생성하면, 앞서 보인 LINK, IP, TCP/UDP 계층에 대한 내용은 감춰진다.
  - ▣ 그러니 응용 프로그래머는 APPLICATION 계층의 완성에 집중하게 된다.



# TCP 클라이언트/서버의 구현



# TCP 서버의 기본적인 함수호출 순서

9



`bind` 함수까지 호출이 되면 주소가 할당된 소켓을 얻게 된다. 따라서 `listen` 함수의 호출을 통해서 연결요청이 가능한 상태가 되어야 한다. 이번 단원에서는 바로 이 `listen` 함수의 호출이 의미하는 바에 대해서 주로 학습한다.

# 연결요청 대기 상태로의 진입

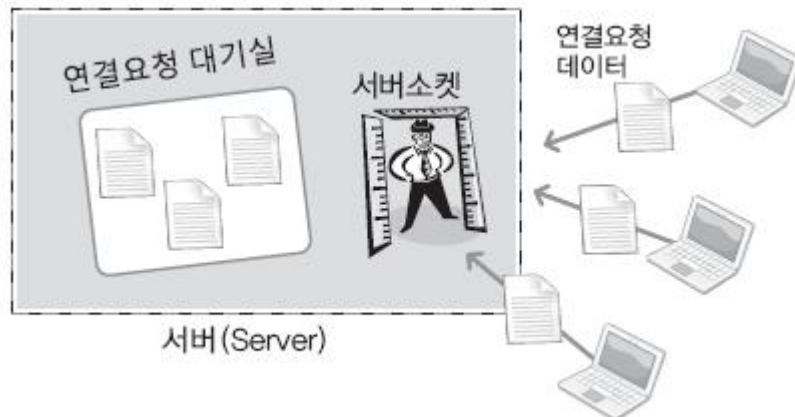
10

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

```
int listen(int sock, int backlog);
```

➔ 성공 시 0, 실패 시 -1 반환

- sock 연결요청 대기상태에 두고자 하는 소켓의 파일 디스크립터 전달, 이 함수의 인자로 전달된 디스크립터의 소켓이 서버 소켓(리스닝 소켓)이 된다.
- backlog 연결요청 대기 큐(Queue)의 크기정보 전달 **동시에 여러번 오는 연결요청을 처리해야하므로 상대적으로 늦게 온 요청들을 커널내에서 큐에 저장해둔다**



연결요청도 일종의 데이터 전송이다. 따라서 연결요청을 받아들이기 위해서도 하나의 소켓이 필요하다. 그리고 이 소켓을 가리켜 **서버소켓** 또는 **리스닝 소켓**이라 한다. listen 함수의 호출은 소켓을 리스닝 소켓이 되게 한다.

# actual number of queued connections for values of backlog

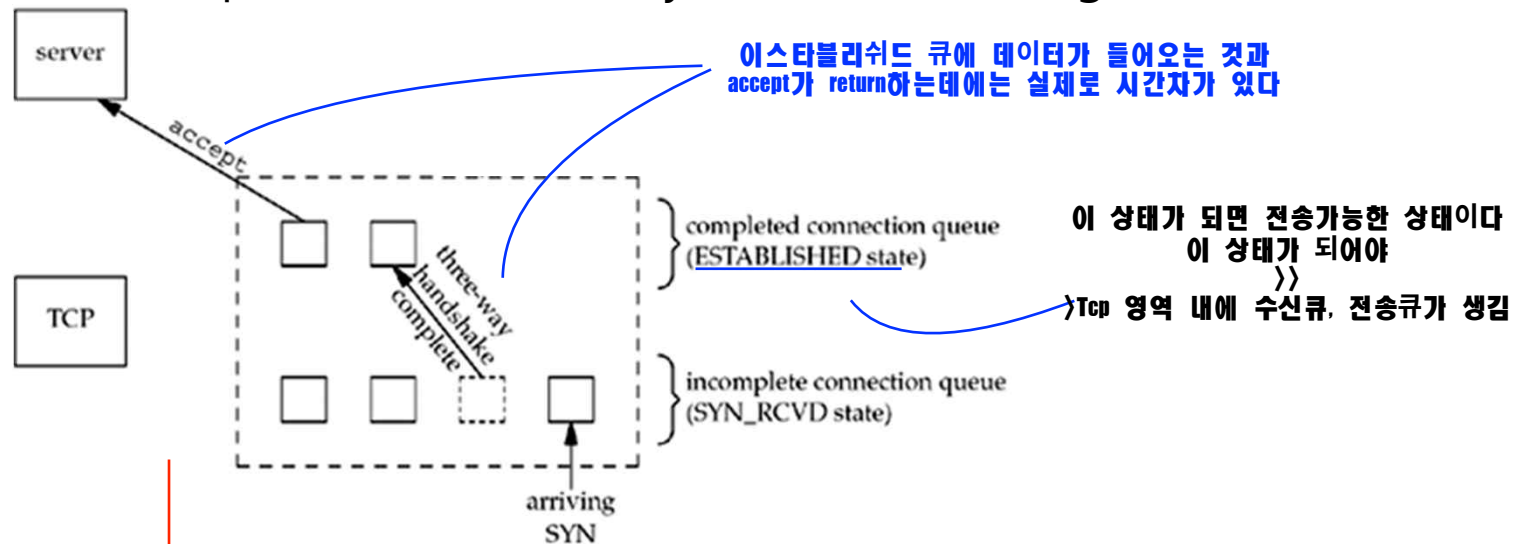
11

<i>backlog</i>	Maximum actual number of queued connections				
	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

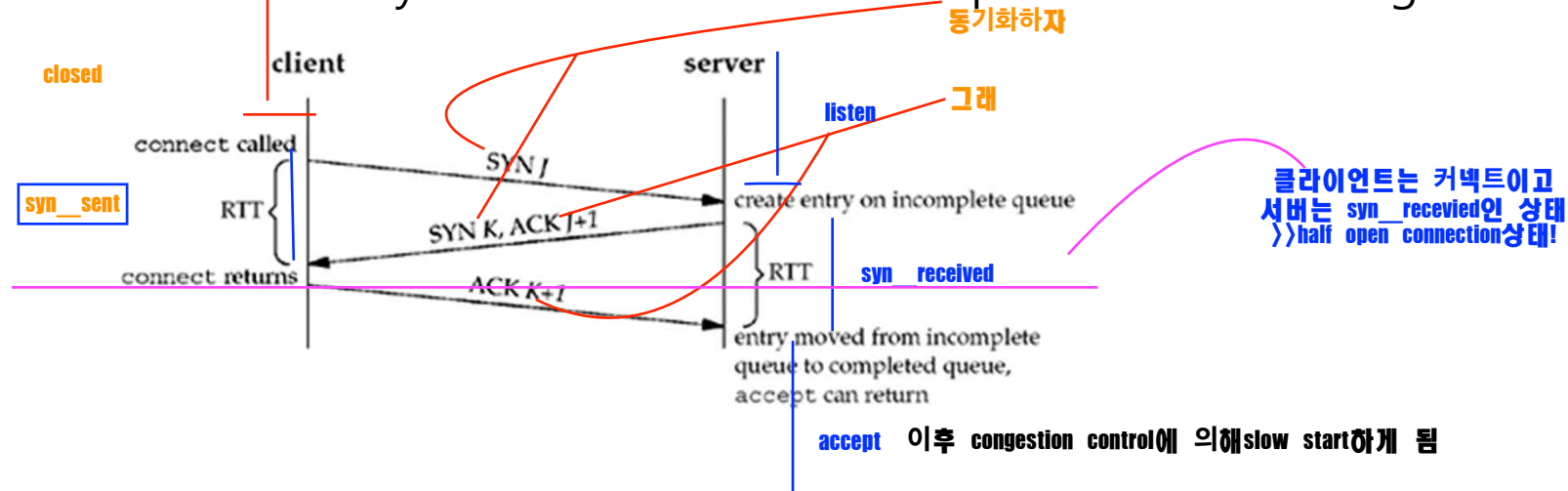
# TCP three-way handshake and the two queues

12

- The two queues maintained by TCP for a listening socket.



- TCP three-way handshake and the two queues for a listening socket.



## TCP three-way handshake and the two queues

13

- incomplete connection queue가 full이면,

버린다  
>> 큐가 다 찼으니까  
그럼 알아서 재전송하겠지

- completed connection queue가 full이면,

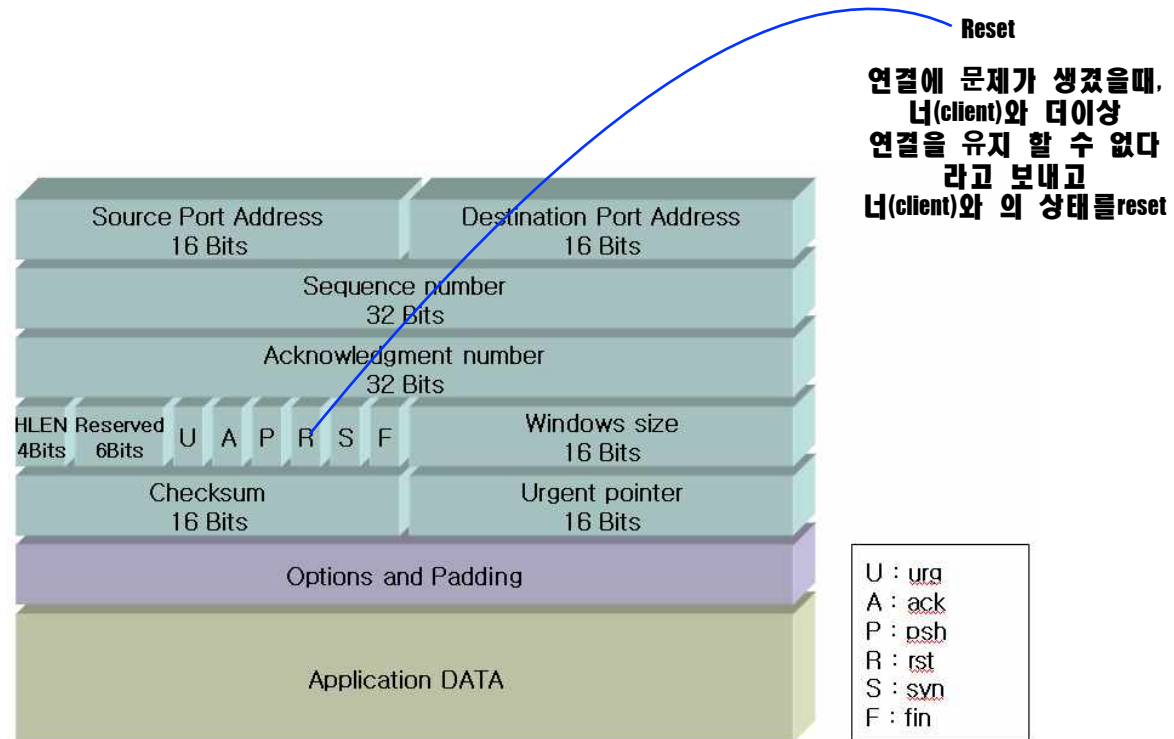
- 3방향 핸드셰이크는 완료되었으나, accept()이전인 상황에서 도착한 데이터는 어떻게 되는가

수신 큐에 저장되어 있다  
>> 만약 application계층에서 읽으려고 한다면  
커넥티드 소켓의 파일디스크립터를 가져와서 읽어야함

질문!

# TCP 세그먼트 구조

14



# 클라이언트의 연결요청 수락

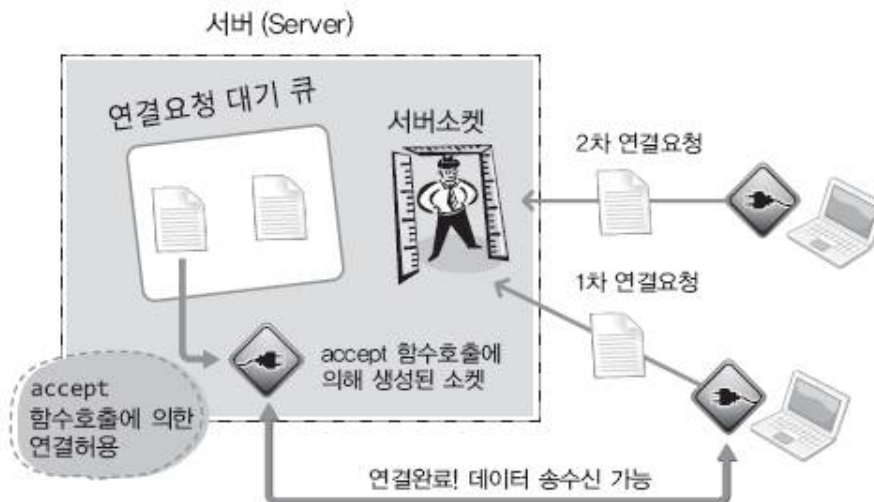
15

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

int accept(int sock, struct sockaddr * addr, socklen_t * addrlen);
```

→ 성공 시 생성된 소켓의 파일 디스크립터, 실패 시 -1 반환

- sock 서버 소켓의 파일 디스크립터 전달.
- addr 연결요청 한 클라이언트의 주소정보를 담을 변수의 주소 값 전달, 함수호출이 완료되면 인자로 전달된 주소의 변수에는 클라이언트의 주소정보가 채워진다.
- addrlen 두 번째 매개변수 addr에 전달된 주소의 변수 크기를 바이트 단위로 전달, 단 크기정보를 변수에 저장한 다음에 변수의 주소 값을 전달한다. 그리고 함수호출이 완료되면 크기정보로 채워져 있던 변수에는 클라이언트의 주소정보 길이가 바이트 단위로 계산되어 채워진다.



연결요청 정보를 참조하여 클라이언트 소켓과의 통신을 위한 **별도의 소켓을 추가로 하나 더 생성** 한다.

# TCP 클라이언트의 기본적인 함수호출 순서

16

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

```
int connect(int sock, const struct sockaddr * servaddr, socklen_t addrlen);
```

➔ 성공 시 생성된 소켓의 파일 디스크립터, 실패 시 -1 반환

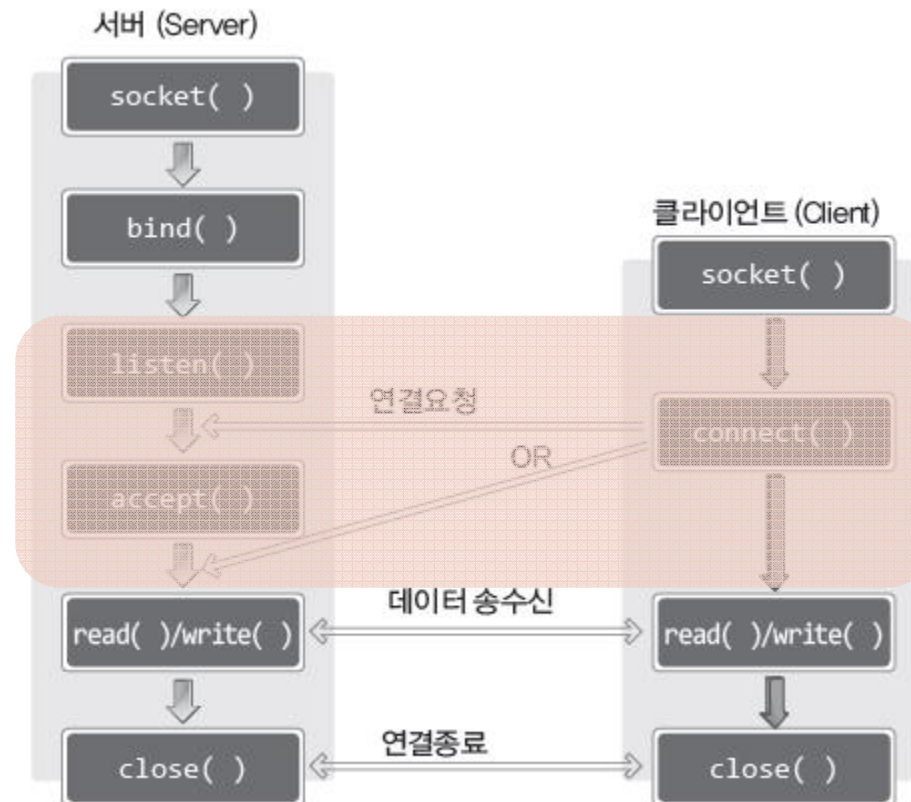
- sock     클라이언트 소켓의 파일 디스크립터 전달.
- servaddr 서버의 주소정보를 담을 변수의 주소 값
- addrlen   servaddr에 저장된 변수 크기(바이트 단위)





# TCP 기반 서버, 클라이언트의 함수호출 관계

17



## connect() 호출이 실패할 때

18

### □ -1을 반환

- ▣ 어느 정도 시간이 흐른 후에도 확인 응답을 받지 못하는 경우
    - errno 변수를 **ETIMEDOUT**로 설정
  - ▣ 거부 메시지를 받는 경우
    - errno 변수를 **ECONNREFUSED**로 설정
- 예) 목적지 대상 포트에 어떤 프로그램도 연결되지 않은 경우  
예) 서버가 listen()을 호출하기 전

## hello\_server.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)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}

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);
    }

    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]));

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

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

    clnt_addr_size=sizeof(clnt_addr);
    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;
}
```

connected sock

sockaddr\_in이 아니다  
> 즉 범용이 아니기 때문에  
바운더리를 정확히 잡기 위해서  
뒤에 sizeof(clnt\_addr);로 받은값을  
넣어줘야한다.

## hello\_client.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]));

    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;
}
```

20

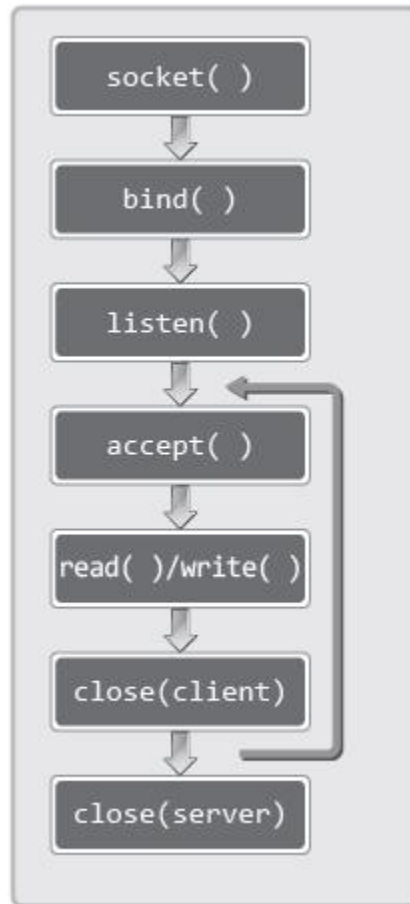
```
void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```



# Iterative 방식 TCP 서버 구현

# Iterative 서버의 구현

22



왼쪽의 그림과 같이 반복적으로 `accept` 함수를 호출하면, 계속해서 클라이언트의 연결요청을 수락할 수 있다. 그러나, 동시에 둘 이상의 클라이언트에게 서비스를 제공할 수 있는 모델은 아니다.

## Iterative 서버와 클라이언트의 일부

23

```
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 \n", i+1);

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

    close(clnt_sock);
}
```

서버 코드의 일부

클라이언트 코드의 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);
}
```

# 에코 클라이언트의 문제점

24

제대로 동작은 하나 문제의 발생 소지가 있는 TCP 에코 클라이언트의 코드

한번만 읽으니까  
데이터가 소실되었을때  
처리 할 수 가 없다  
>> 따라서 받은 사이즈가 보낸 사이즈와  
동일할때까지 반복문을 넣어주면된다

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

**TCP의 데이터 송수신에는 경계가 존재하지 않는다!**

☞ 서버가 전송한 문자열의 일부만 읽혀질 수도 있다.



# 에코 클라이언트의 문제점 확인하기

25

## 에코 서버의 코드

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

서버는 데이터의 경계를 구분하지 않고 수신된 데이터를 그대로 전송할 의무만 갖는다. TCP가 본디 데이터의 경계가 없는 프로토콜이므로, 두 번의 **write** 함수 호출을 통해서 데이터를 전송하건, 세 번의 **write** 함수호출을 통해서 데이터를 전송하건, 문제 되지 않는다.

## 에코 클라이언트의 코드

```
write(sock, message, strlen(message));
str_len=read(sock, message, BUF_SIZE-1);
```

반면, 클라이언트는 문장 단위로 데이터를 송수신하기 때문에, 데이터의 경계를 구분해야 한다. 때문에 이와 같은 데이터 송수신 방식은 문제가 된다.

## 에코 클라이언트의 해결책!

26

```
str_len=write(sock, message, strlen(message));  
recv_len=0;  
while(recv_len<str_len)  
{  
    recv_cnt=read(sock, &message[recv_len], BUF_SIZE-1);  
    if(recv_cnt==-1)  
        error_handling("read() error!");  
    recv_len+=recv_cnt;  
}  
message[recv_len]=0;  
printf("Message from server: %s", message);
```

**write** 함수호출을 통해서 전송한 데이터의 길이만큼 읽어 들이기 위한 반복문의 삽입이 필요하다. 이것이 TCP를 기반으로 데이터를 구분지어 읽어 들이는데 부가적으로 필요한 구분이다.

같은상황이지만  
위는 블록모드  
아래는 논블록모드

27

## read() 호출의 가능성

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

→ 성공 시 수신한 바이트 수(단 파일의 끝을 만나면 0), 실패 시 -1 반환

- fd 데이터 수신대상을 나타내는 파일 디스크립터 전달.
- buf 수신한 데이터를 저장할 버퍼의 주소 값 전달.
- nbytes 수신할 최대 바이트 수 전달.

- 호출이 nbytes와 같은 값을 반환한다.
  - nbytes 바이트 전체를 읽어서 buf에 저장했다. 결과는 의도한 바와 같다.
- 호출이 nbytes보다 작지만 0보다는 큰 값을 반환한다.
  - 읽은 바이트는 buf에 저장된다. 이런 상황은 시그널이 중간에 읽기를 중단시켰거나, 읽는 도중에 에러가 발생해서 1바이트 이상이지만 nbytes 길이만큼 데이터를 가져오지 못했거나, nbytes 바이트를 읽기 전에 EOF에 도달한 경우에 발생한다. 다시 read()를 호출하면 남은 바이트를 남은 버퍼 공간에 읽을 수 있다.
- 0을 반환한다.
  - 이는 EOF를 나타낸다. 더 이상 읽을 데이터가 없다.
- 현재 사용 가능한 데이터가 없기 때문에 호출이 블록된다.
  - 논블록 모드에서는 이런 상황이 발생하지 않는다.
- 호출이 -1을 반환하고 errno를 EINTR로 설정한다.
  - EINTR은 바이트를 읽기 전에 시그널이 도착했음을 알려준다. 이럴 때는 다시 호출하면 된다.
- 호출이 -1을 반환하고 errno를 EAGAIN으로 설정한다.
  - EAGAIN은 현재 읽을 데이터가 없기 때문에 블록된 상태이며 나중에 반드시 다시 읽기 요청을 해야 한다고 알려준다. 논블록 모드일 때만 일어나는 상황이다.
- 호출이 -1을 반환하고 errno를 EINTR이나 EAGAIN이 아닌 다른 값으로 설정한다.
  - 이는 심각한 에러가 발생했음을 알려준다.

에러가 발생했어도 읽은만큼은 리턴한다

eof의 경우

소켓을 닫을때 물어준다고 생각하면됨

질문: 논블록 모드가 필요한 이유?

논블록 모드에서는  
블록모드에서 블록이 걸리는  
리턴을 한다

부분에

## 전체 바이트 읽기

28

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("read");
        break;
    }

    len -= ret;
    buf += ret;
}
```

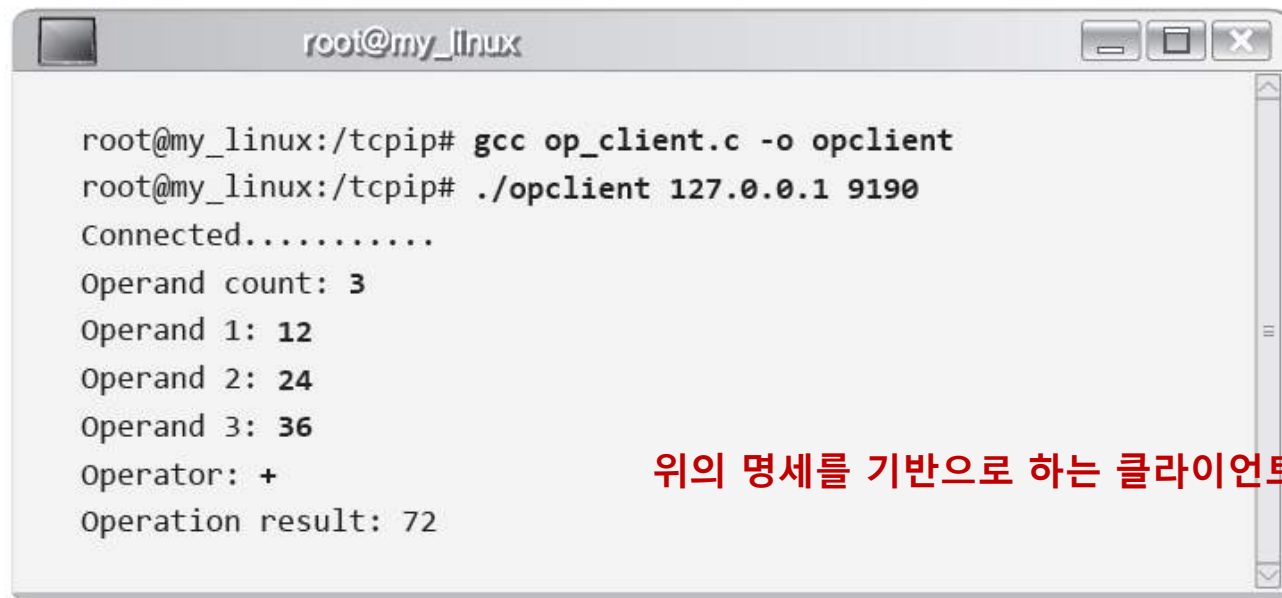


## 계산기 프로그램 구현하기 (애플리케이션 프로토콜)

## 계산기 프로그램 구현하기(애플리케이션 프로토콜)

서버는 클라이언트로부터 여러 개의 숫자와 연산자 정보를 전달받는다. 그러면 서버는 전달받은 숫자를 바탕으로 덧셈, 뺄셈 또는 곱셈을 계산해서 그 결과를 클라이언트에게 전달한다. 예를 들어서 서버로 3, 5, 9가 전달되고 덧셈연산이 요청된다면 클라이언트에는  $3+5+9$ 의 연산결과가 전달되어야 하고, 곱셈연산이 요청된다면 클라이언트에는  $3\times 5\times 9$ 의 연산결과가 전달되어야 한다. 단, 서버로 4, 3, 2가 전달되고 뺄셈연산이 요청된다면 클라이언트에는  $4-3-2$ 의 연산결과가 전달되어야 한다. 즉, 뺄셈의 경우에는 첫 번째 정수를 대상으로 뺄셈이 진행되어야 한다.

이와 같은 서버 클라이언트 사이에서의 데이터 송수신 명세가 바로 프로토콜이다!



```
root@my_linux

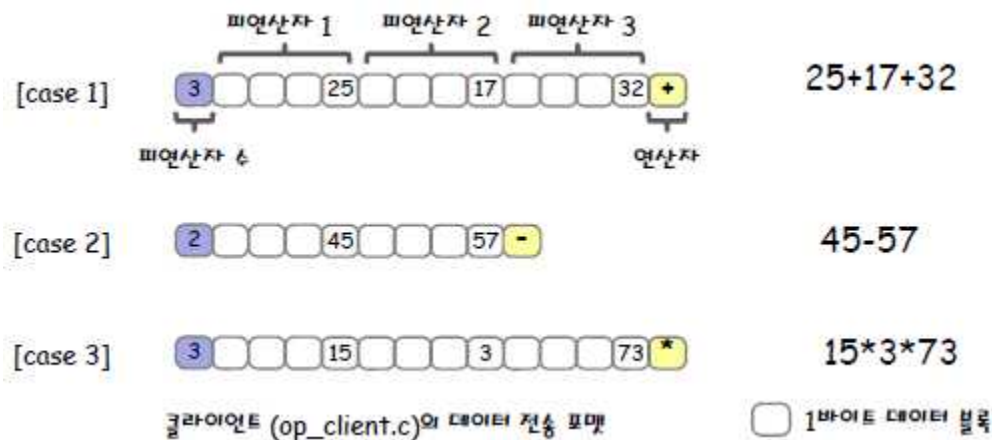
root@my_linux:/tcpip# gcc op_client.c -o opclient
root@my_linux:/tcpip# ./opclient 127.0.0.1 9190
Connected.....
Operand count: 3
Operand 1: 12
Operand 2: 24
Operand 3: 36
Operator: +
Operation result: 72
```

위의 명세를 기반으로 하는 클라이언트 프로그램의 실행의 예

# 서버, 클라이언트의 구현

31

- 클라이언트는 서버에 접속하자마자 피연산자의 개수정보를 1바이트 정수형태로 전달한다.
- 클라이언트가 서버에 전달하는 정수 하나는 4바이트로 표현한다.
- 정수를 전달한 다음에는 연산의 종류를 전달한다. 연산정보는 1바이트로 전달한다.
- 문자 +, -, \* 중 하나를 선택해서 전달한다.
- 서버는 연산결과를 4바이트 정수의 형태로 클라이언트에게 전달한다.
- 연산결과를 얻은 클라이언트는 서버와의 연결을 종료한다.



프로토콜은 위와 같이(그 이상으로)  
명확히 정의해야 한다.

op\_server.c op\_client.c를 통해  
구현하였으니, 참조!

# op\_client.c

32

```
...
#define BUF_SIZE 1024
#define RLT_SIZE 4
#define OPSZ 4
...
int main(int argc, char *argv[]) {
    int sock;
    char opmsg[BUF_SIZE];
    int result, opnd_cnt, i;
    struct sockaddr_in serv_adr;
    ...
    if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
        error_handling("connect() error!");
    else
        puts("Connected.....");

    fputs("Operand count: ", stdout);
    scanf("%d", &opnd_cnt);
    opmsg[0]=(char)opnd_cnt;

    for(i=0; i<opnd_cnt; i++) {
        printf("Operand %d: ", i+1);
        scanf("%d", (int*)&opmsg[i*OPSZ+1]);
    }
    fgetc(stdin);
    fputs("Operator: ", stdout);
    scanf("%c", &opmsg[opnd_cnt*OPSZ+1]);
    write(sock, opmsg, opnd_cnt*OPSZ+2);
    read(sock, &result, RLT_SIZE);

    printf("Operation result: %d\n", result);
    close(sock);
    return 0;
}
```

개행문자 회수



# op\_server.c

C의 경우 char 형 배열을 많이사용  
이를 받을때 포인터 캐스팅을 이용해서  
내가 원하는 타입으로 읽으라고 할 수 있음

33

```
...
#define BUF_SIZE 1024
#define OPSZ 4
...
int main(int argc, char *argv[]) {
    char opinfo[BUF_SIZE];
    int result, opnd_cnt, i;
    int rcv_cnt, rcv_len;
    ...
    for(i=0; i<5; i++) {
        opnd_cnt=0;
        clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);

        read(clnt_sock, &opnd_cnt, 1);

        rcv_len=0;
        while((opnd_cnt*OPSZ+1)>rcv_len) {
            rcv_cnt=read(clnt_sock, &opinfo[rcv_len], BUF_SIZE-1);
            rcv_len+=rcv_cnt;
        }
        result=calculate(opnd_cnt, (int*)opinfo, opinfo[rcv_len-1]);
        write(clnt_sock, (char*)&result, sizeof(result));
        close(clnt_sock);
    }
    close(serv_sock);
    return 0;
}
```

```
int calculate(int opnum, int opnds[], char op) {
    int result=opnds[0], i;

    switch(op) {
        case '+':
            for(i=1; i<opnum; i++) result+=opnds[i];
            break;
        case '-':
            for(i=1; i<opnum; i++) result-=opnds[i];
            break;
        case '*':
            ...
    }
    return result;
}
```