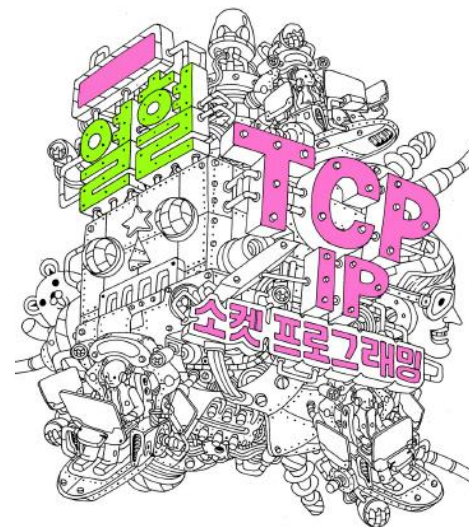




# 윤성우의 열혈 TCP/IP 소켓 프로그래밍

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

## Chapter 04. TCP 기반 서버 / 클라이언트 1



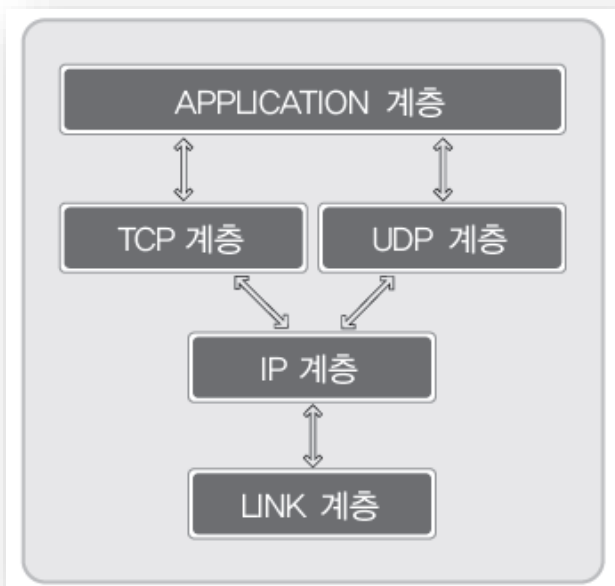
## Chapter 04-1. TCP와 UDP에 대한 이해

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

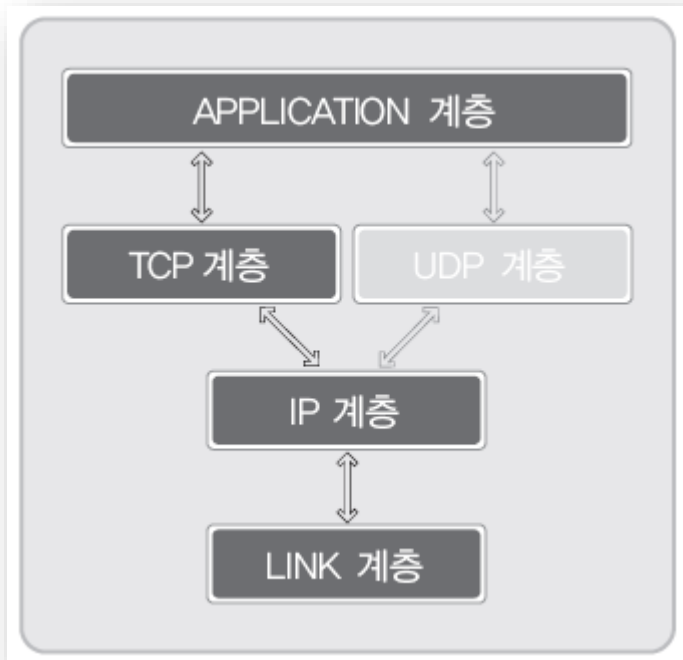
# TCP/IP 프로토콜 스택

## ▶ TCP / IP 프로토콜 스택이란?

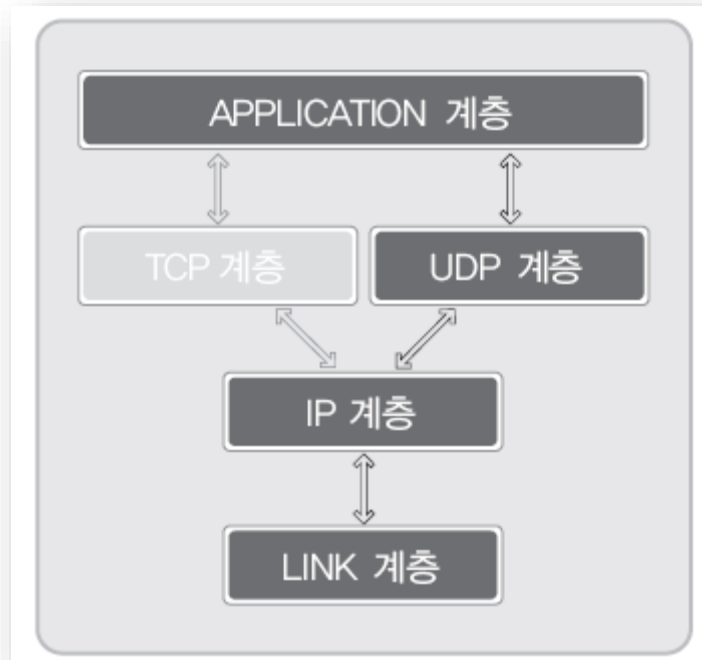
- ▶ 인터넷 기반의 데이터 송수신을 목적으로 설계된 개별 통신 계층의 묶음
- ▶ 큰 문제를 작게 나눠서 각 계층별로 특화된 작업 실행
- ▶ 데이터 송수신의 과정을 네 개의 영역으로 계층화 한 결과
- ▶ 각 스택 별 영역을 전문화하고 표준화 함
- ▶ OSI/IOS는 7 계층으로 세분화가 되며,  
TCP/IP는 4 계층으로 표현함



# TCP 소켓과 UDP 소켓의 스택 FLOW



TCP 소켓의 스택 FLOW



UDP 소켓의 스택 FLOW

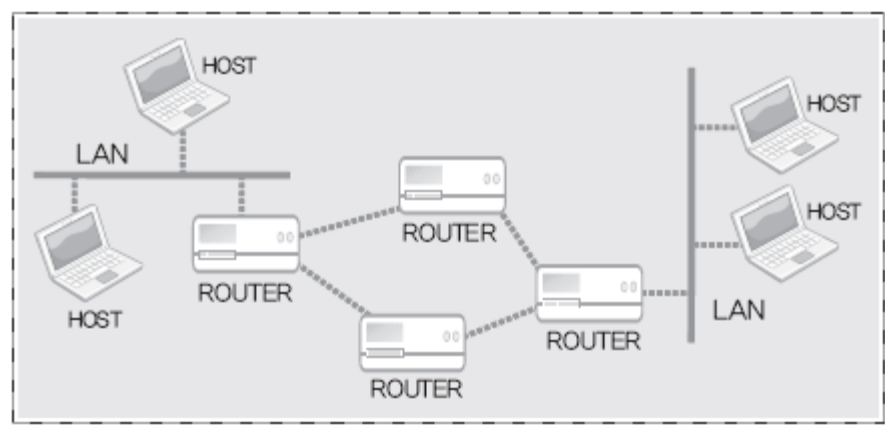
# IP & LINK 계층

## ▶ IP 계층의 기능 및 역할

- ▶ IP는 Internet Protocol을 의미하며, IP 주소 설정 및 경로 설정과 관련이 있는 프로토콜

## ▶ LINK 계층의 기능 및 역할

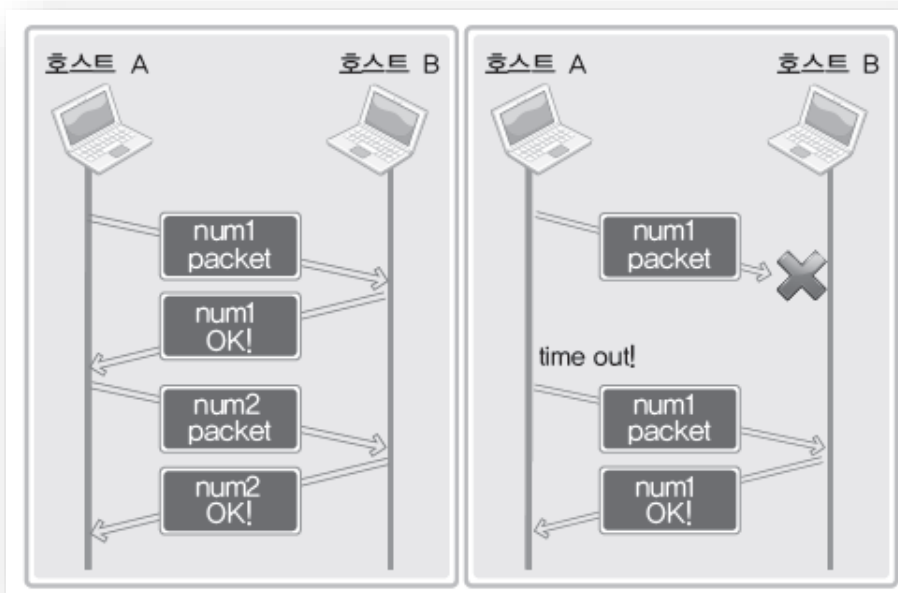
- ▶ 물리적인 영역의 표준화 결과
- ▶ LAN, WAN, MAN과 같은 물리적인 네트워크 표준 관련 프로토콜이 정의된 영역
- ▶ 아래의 그림과 같은 물리적인 연결의 표준이 된다.



# TCP/UDP 계층

## ▶ TCP/UDP 계층의 기능 및 역할

- ▶ 실제 데이터의 송수신과 관련 있는 계층으로 전송(Transport) 계층이라고도 함
- ▶ TCP는 데이터전송의 신뢰성을 보장하는 프로토콜(신뢰성 있는 프로토콜),  
UDP는 데이터 전송의 신뢰성을 보장하지 않는 프로토콜
- ▶ TCP는 신뢰성을 보장하기 때문에 UDP에 비해 복잡한 프로토콜



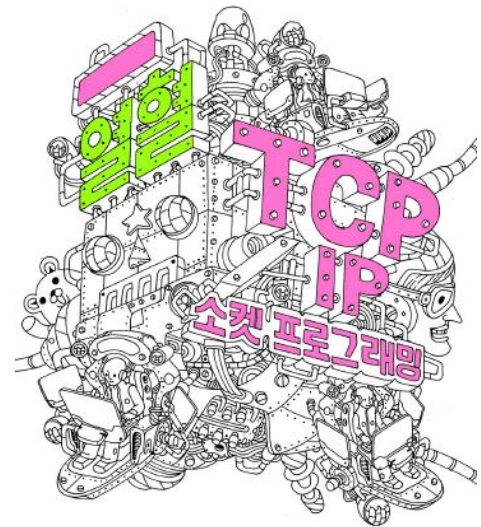
TCP는 왼쪽의 그림에서 보듯이 상호 확인 과정을 거친다. 때문에 신뢰성을 보장하지만, 그만큼 복잡한 과정을 거쳐서 데이터 전송이 이뤄진다.

# APPLICATION 계층

---



- ▶ 프로그래머에 의해서 완성되는 APPLICATION 계층
  - ▶ 응용프로그램의 프로토콜을 구성하는 계층
  - ▶ 소켓을 기반으로 완성하는 프로토콜을 의미함
  - ▶ 소켓을 생성하면, 앞서 보인 TCP/UDP, IP, LINK 계층에 대한 상세 내용은 감춰진다.
    - ▶ Hiding 또는 Encapsulation이라 함
  - ▶ 그래서 응용 프로그래머는 APPLICATION 계층의 완성에 집중할 수 있게 됨



## Chapter 04-2. TCP기반 서버, 클라이언트 의 구현

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판



# TCP 서버의 기본적인 함수호출 순서



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.

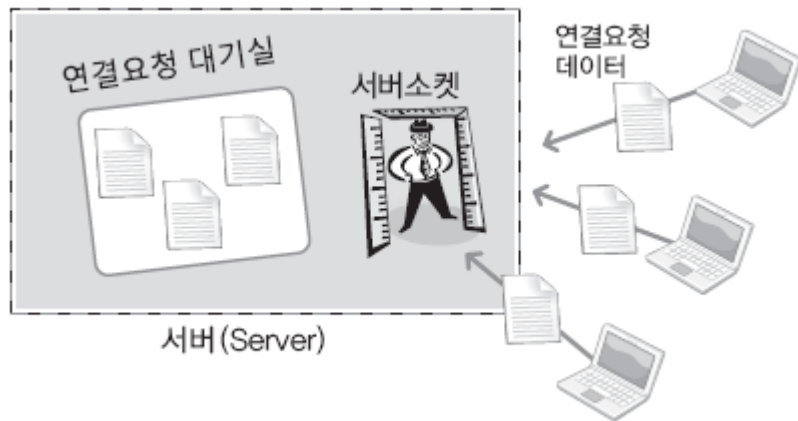
# 서버에서 연결요청 대기 상태로의 진입

```
#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, 5가 전달되면 큐의 크기가 5가 되어 클라이언트의 연결요청을 5개까지 대기시킬 수 있다.



연결요청도 일종의 데이터 전송이다. 따라서 연결요청을 받아들이기 위해서 하나의 소켓이 필요하다. 그리고 이 소켓을 가리켜 **서버소켓** 또는 **리스닝 소켓**이라 한다. listen 함수의 호출은 이미 생성된 소켓을 리스닝 소켓으로 설정 변경한다.

# 서버에서 클라이언트의 연결요청 수락

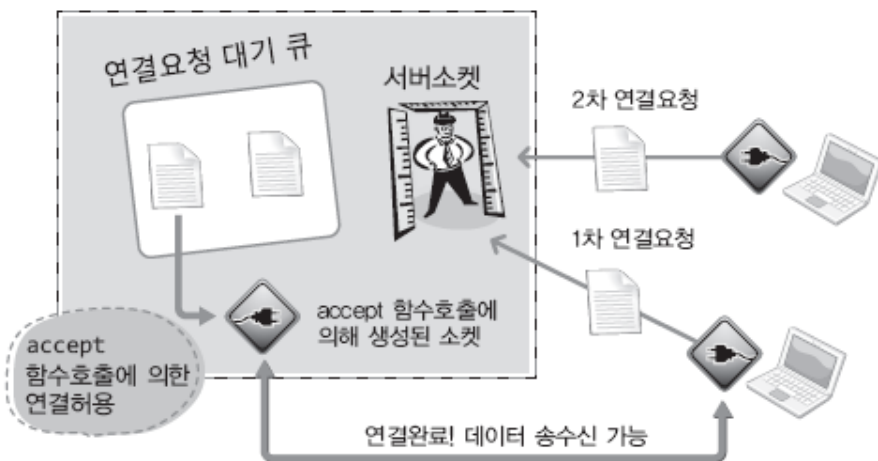
```
#include <sys/socket.h>
```

```
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.

서버 (Server)



accept() 과정에서 연결요청 정보를 참조하여 클라이언트 소켓과 통신하기 위한 별도의 소켓을 추가로 하나 더 생성한다. 그리고 이렇게 생성된 소켓을 대상으로 데이터의 송수신이 진행되며, 생성된 소켓의 파일 디스크립터가 반환된다.

# 클라이언트에서 기본적인 함수호출 순서

```
#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   연결요청 한 클라이언트의 주소정보를 담을 변수의 주소 값 전달, 함수호출이 완료되면 인자로 전달된 주소의 변수에는 클라이언트의 주소정보가 채워진다.
- addrlen   두 번째 매개변수 servaddr에 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.

socket( )

소켓생성



connect( )

연결요청



read( )/write( )

데이터 송수신

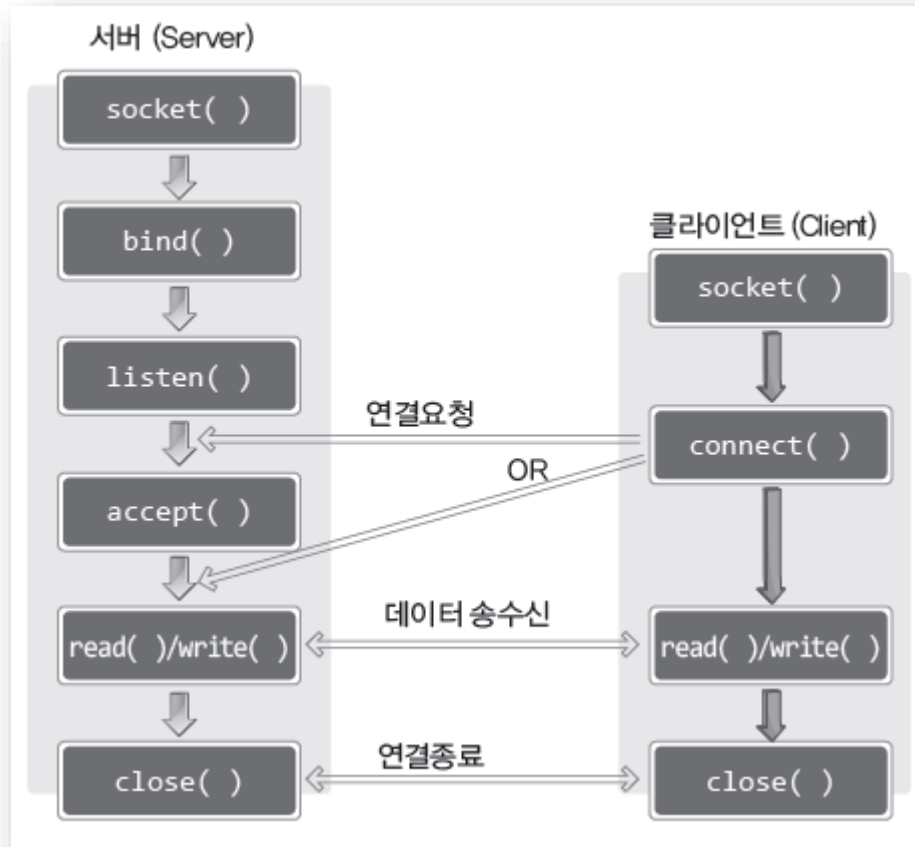


close( )

연결종료

클라이언트의 경우 소켓을 생성하고, 이 소켓을 대상으로 연결 요청을 위해서 connect 함수를 호출하는 것이 전부이다. 그리고 connect 함수를 호출할 때 연결할 서버의 주소 정보도 함께 전달한다.

# TCP 기반 서버, 클라이언트의 함수 호출 관계

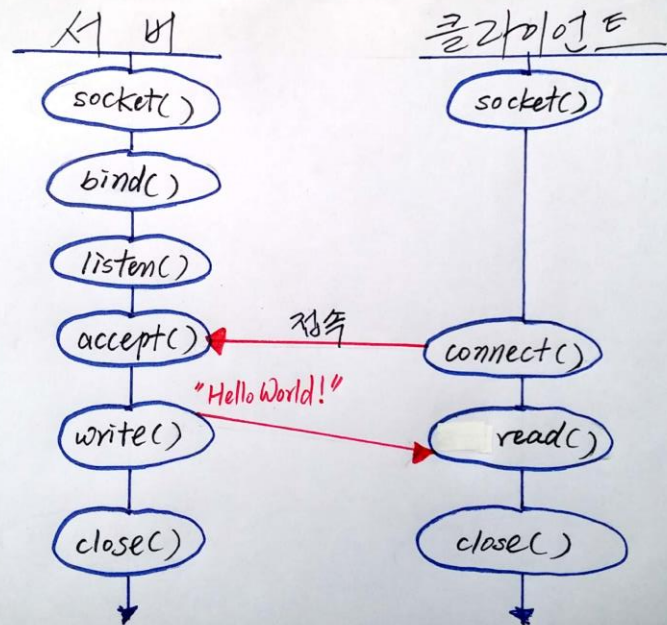


확인할 사항은, 서버의 listen 함수 호출 이후에 클라이언트 connect 함수 호출이 유효하다는 점이다. 더불어 그 이유까지도 설명할 수 있어야 한다.

# 1장 HelloWorld 전송 프로그램 리뷰

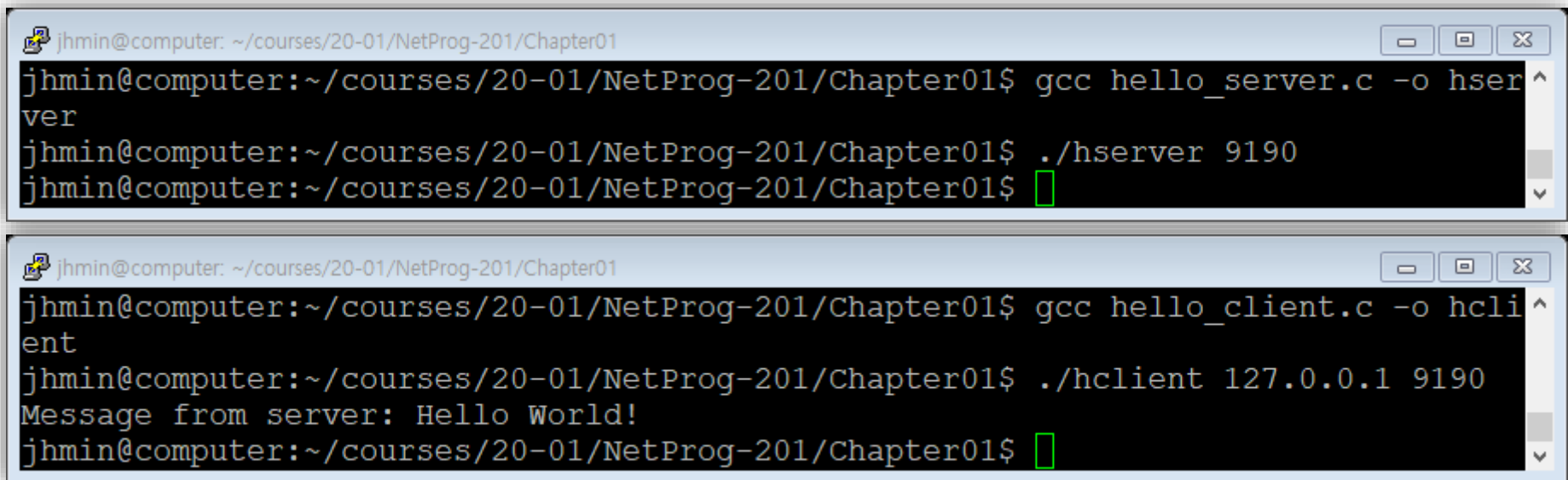
- ▶ `hello_server.c`, `hello_client.c` 소스 코드 참조
  - ▶ 2개의 네트워크 코드 분석할 때는 서버 소스 코드 먼저

/장 ( `hello_server.c` / `hello_client.c` )



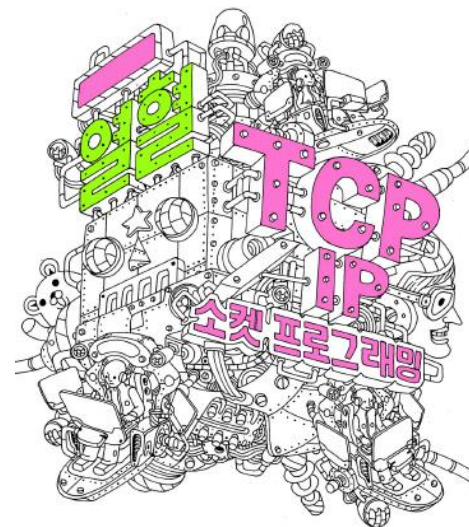
# 1장 HelloWorld 전송 프로그램 리뷰

- ▶ hello\_server.c, hello\_client.c 소스 코드 컴파일 후 실행
  - ▶ 2개의 네트워크 코드 실행할 때도 서버 먼저 실행



```
jhmin@computer: ~/courses/20-01/NetProg-201/Chapter01
jhmin@computer:~/courses/20-01/NetProg-201/Chapter01$ gcc hello_server.c -o hserver
jhmin@computer:~/courses/20-01/NetProg-201/Chapter01$ ./hserver 9190
jhmin@computer:~/courses/20-01/NetProg-201/Chapter01$

jhmin@computer: ~/courses/20-01/NetProg-201/Chapter01
jhmin@computer:~/courses/20-01/NetProg-201/Chapter01$ gcc hello_client.c -o hclient
jhmin@computer:~/courses/20-01/NetProg-201/Chapter01$ ./hclient 127.0.0.1 9190
Message from server: Hello World!
jhmin@computer:~/courses/20-01/NetProg-201/Chapter01$
```

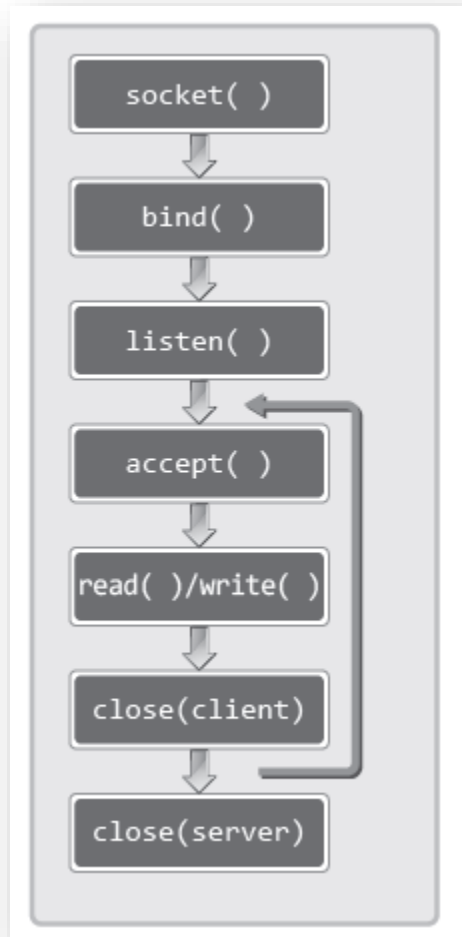


## Chapter 04-3. Iterative 기반의 서버, 클라이언트의 구현

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판



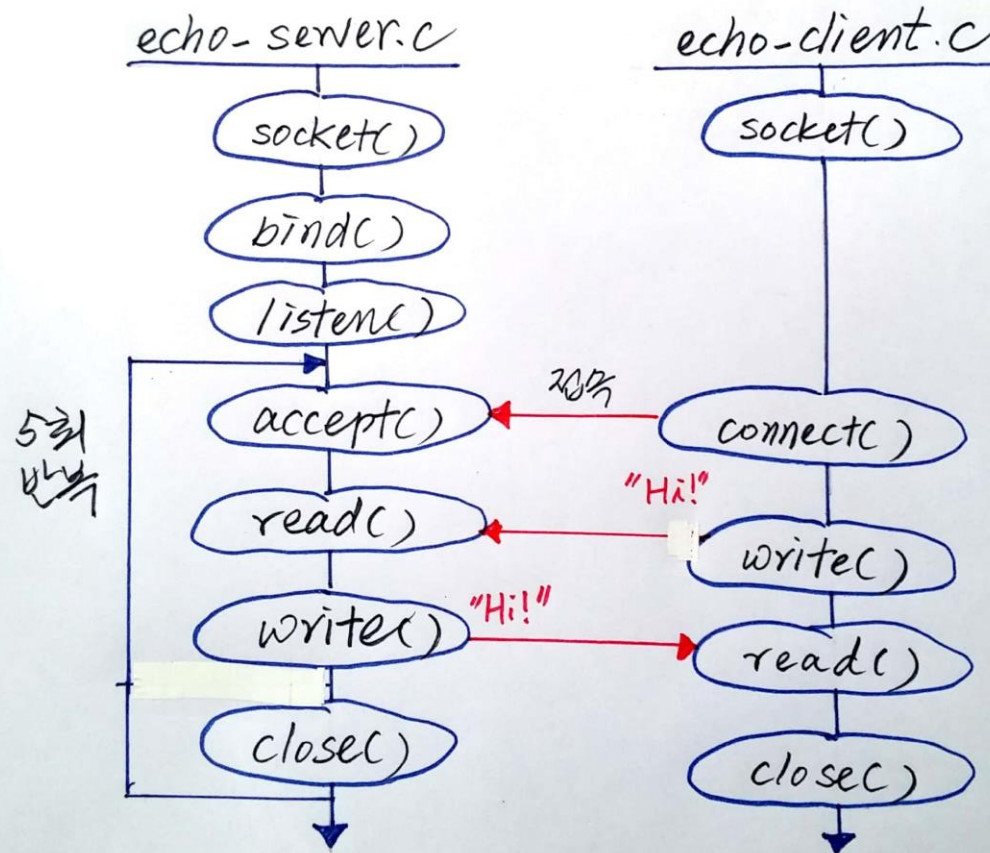
# 서버에서 Iterative 서버 구현



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

# Iterative 기반 서버 / 클라이언트 동작

## 4장 Iterative 기반 서버 / 클라이언트



# Iterative 기반 동일 서버 접속 실행

```
$ cd chap04
$ make
gcc echo_server.c -o eserver
gcc echo_client.c -o eclient
$ ./eserver 9100 &
[1] 27134
$ ./eclient 127.0.0.1 9100
Connected.....
Input message(Q to quit): Connected client 1
Good morning!
Message from server: Good morning!
Input message(Q to quit): Hi!
Message from server: Hi!
Input message(Q to quit): q
$ jobs
[1]+  실행중                  ./eserver 9100 &
$ kill -9 %1
$
[1]+  죽었음                  ./eserver 9100
$
```

← 4장 디렉토리로 이동  
← 2개 프로그램 컴파일

← 서버 프로그램 실행(포트 번호 주의)

← 클라이언트 프로그램 실행(포트 번호 주의)

← 클라이언트에서 첫번째 문장 입력

← 클라이언트에서 두번째 문장 입력

← 클라이언트에서 종료 문자 입력

← 백그라운드에서 실행 중인 서버 프로세스 확인

← 백그라운드에서 실행 중인 서버 프로세스 강제 종료

# Iterative 기반 **원격** 서버 접속 실행

원격서버 : 210.93.57.72

```
210.93.57.72 - PuTTY
[jhmin@computer ~/courses/2019-1/yoona_tcp/Chapter04]$ make
gcc echo_server.c -o eserver -lsocket -lnsl
gcc echo_client.c -o eclient -lsocket -lnsl
[jhmin@computer ~/courses/2019-1/yoona_tcp/Chapter04]$ ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index
1
    inet 127.0.0.1 netmask ff000000
eri0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
    inet 210.93.57.72 netmask fffffff0 broadcast 210.93.57.255
[jhmin@computer ~/courses/2019-1/yoona_tcp/Chapter04]$ eserver 9190
Connected client 1
```

로컬서버 : 210.93.57.50

```
jhmin@computer: ~/courses/NetPro-191/NetProg/Chapter04
jhmin@computer:~/courses/NetPro-191/NetProg/Chapter04$ eclient 210.93.57.72 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): q
jhmin@computer:~/courses/NetPro-191/NetProg/Chapter04$
```

# Iterative 기반 동일 서버 5회 순차 접속 실행

```
~$ ./eserver 9100
Connected client 1
Connected client 2
Connected client 3
Connected client 4
Connected client 5
~$
```

서버에 클라이언트 5회  
접속 후 프로그램 종료

```
computer.kpu.ac.kr - PuTTY
~/courses/NetProg-201/Chapter04$ ./eclient 127.0.0.1 9100
Connected.....
Input message(Q to quit): q
~/courses/NetProg-201/Chapter04$
```

```
Input message(Q to quit): q
~/courses/NetProg-201/Chapter04$
```

```
Input message(Q to quit): q
~/courses/NetProg-201/Chapter04$
```

```
Input message(Q to quit): q
~/courses/NetProg-201/Chapter04$
```

```
Input message(Q to quit): q
~/courses/NetProg-201/Chapter04$
```

# Iterative 기반 서버 / 클라이언트 코드 리뷰

## ▶ echo\_server.c

서버 코드의 일부  
(5회 반복)

```
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);
}
```

## ▶ echo\_client.c

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

정상 동작하지만 오류 발생 가능성이 포함된 TCP 에코 클라이언트의 코드 부분

```
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! 그런데 위의 코드는 다음 사항을 가정하고 있다.

**“한 번의 read 함수호출로 앞서 전송된 문자열 전체를 읽어 들일 수 있다.”**

그러나 이는 잘못된 가정이다. TCP에는 데이터의 경계가 존재하지 않기 때문에 서버가 전송한 문자열의 일부만 읽혀질 수도 있다.

# ※ 통신에 사용할 특정 소켓(9100번) 상태 확인하기

\$ netstat -an | grep 9100 ← 9100번 포트 사용 상태 확인 명령어

```
tcp      0      0 0.0.0.0:9100          0.0.0.0:*        LISTEN
```

← 9100번 포트 현재 사용 중임

\$ netstat -anp | grep 9100 ← 9100번 포트 사용 프로세스 확인 명령어

(Not all processes could be identified, non-owned process info  
will not be shown, you would have to be root to see it all.)

```
tcp      0      0 0.0.0.0:9100          0.0.0.0:*        LISTEN -
```

← 9100번 포트 사용 중인 프로세스  
필드에는 자기 계정것만 표시되고  
다른 계정것은 root만 볼 수 있음

\$ ps aux | grep 9100

← 9100번 포트 사용 프로세스 사용자  
확인 명령어

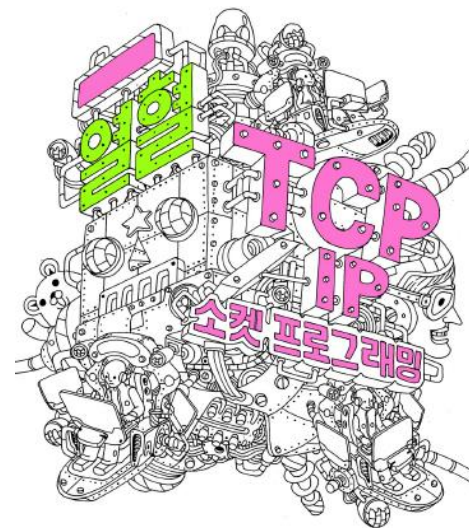
```
jhmin   18273  0.0  0.0  22824  1060 pts/58  S+   11:44   0:00 grep --  
color=auto 9100
```

```
ce00c000 22653  0.0  0.0   4508   756 ?    S    3월20   0:00 ./eserver 9100
```

← eserver 프로세스에서 사용 중임을 표시

\$





## Chapter 04-4. 윈도우 기반으로 구현하기

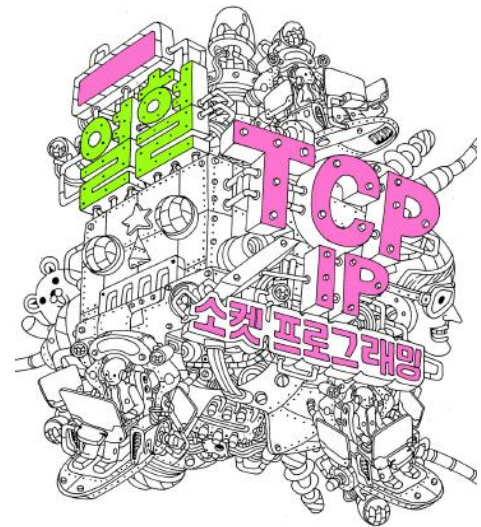
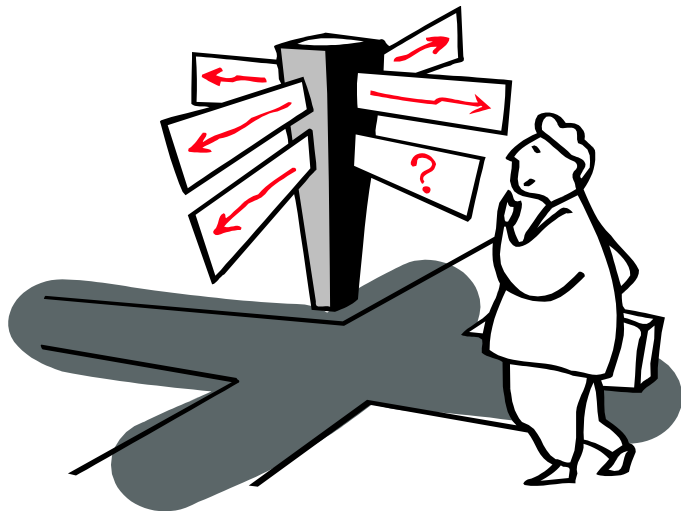
윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

# 윈도우 기반으로 변경할 때 필요한 것

---

1. WSASStartup, WSACleanup 함수호출을 통한 소켓 라이브러리의 초기화와 해제
2. 자료형과 변수의 이름을 윈도우 스타일로 변경하기
3. 데이터 송수신을 위해서 read, write 함수 대신 recv, send 함수 호출하기
4. 소켓의 종료를 위해서 close 대신 closesocket 함수 호출하기

마치 공식을 적용하듯이(소스의 내용을 잘 모르는 상태에서) 윈도우 기반으로 예제를 변경할 수도 있다. 그만큼 리눅스 기반 예제와 윈도우 기반 예제는 동일하다!



Chapter 4가 끝났습니다. 질문 있으신지요?