

삼성 청년 SW 아카데미

네트워크 프로그래밍

<알림>

본 강의는 삼성 청년 SW아카데미의 콘텐츠로
보안서약서에 의거하여
강의 내용을 어떠한 사유로도 임의로 복사,
촬영, 녹음, 복제, 보관, 전송하거나
허가 받지 않은 저장매체를
이용한 보관, 제3자에게 누설, 공개,
또는 사용하는 등의 행위를 금합니다.

1장. Network 기본

챕터의 포인트

- Protocol
- IP
- TCP/UDP
- PORT

Protocol

통신의 약속 = 프로토콜

- 통신의 두 주체가 같은 프로토콜 쓴다면, 약속된 규칙으로 통신이 가능하다.



적의 상태를 표현하기 위한 신호 약속



상상 못 했을때와 했을 때 신호 약속

통신 구현하기 전, 프로토콜을 완벽히 이해 해야 하는 이유

- 완벽한 통신 Application 제작 가능
- 통신 App을 위한 Library 제작 가능
- 프로토콜 내용대로 수행하는 Server 구축 가능

통신 규칙을 정확히 알아야,
프로그램을 정확히 만든다.

→ 네트워크 프로그래밍에서 프로토콜 내용의 대한 이해는 필수

IP

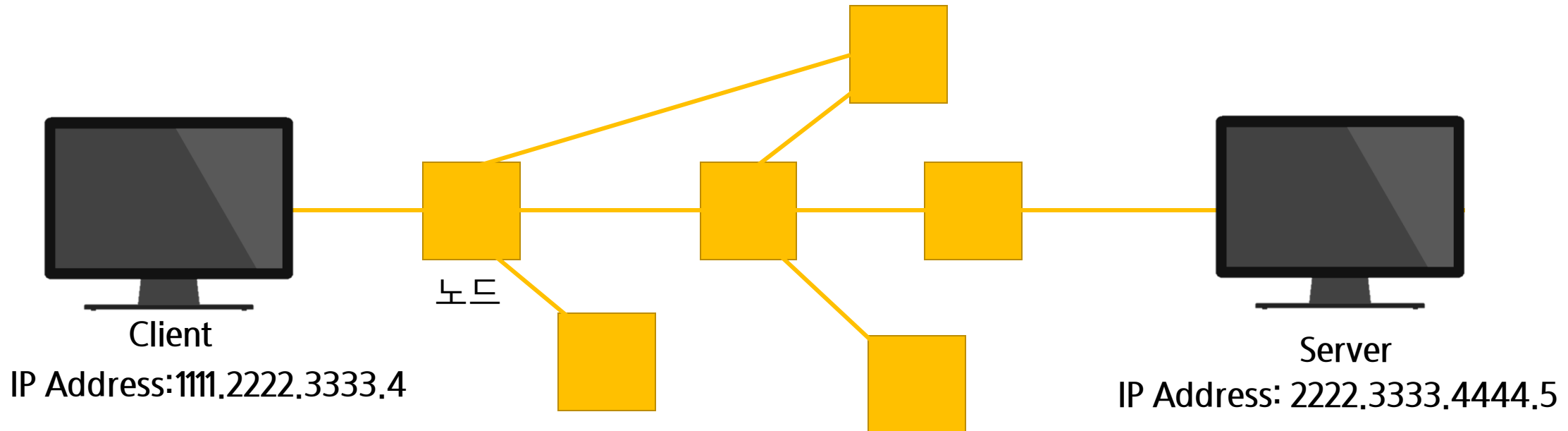
IP(Internet Protocol)

- 송신 호스트와 수신 호스트가 정보를 주고받는데 사용하는 규약
- 패킷 혹은 데이터그램이라고 하는 덩어리로 나뉘어 전송
- 비 신뢰성, 비연결성



IP Address(Internet Protocol)

- 인터넷상에 있는 컴퓨터의 고유한 주소



- 비 연결성

- 패킷을 받을 대상이 없거나 해당 도착지의 주소가 불능 상태여도 패킷을 전송

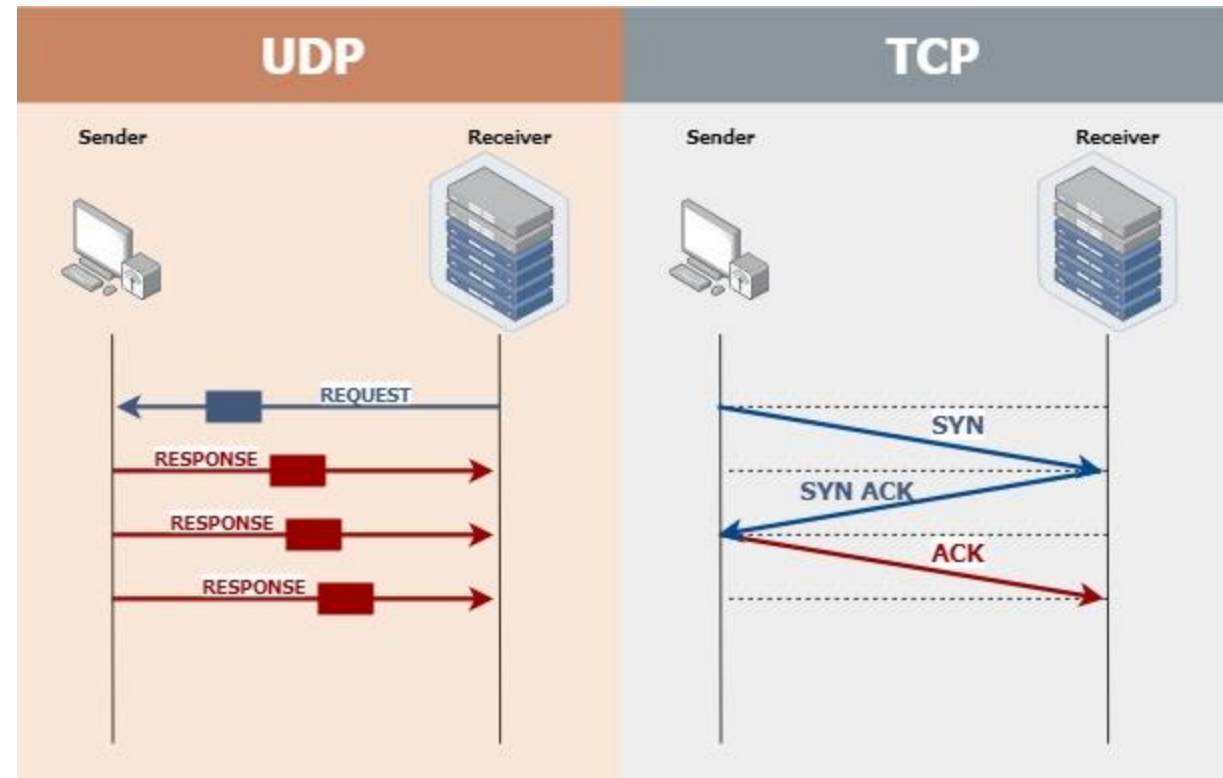
- 비 신뢰성

- 통신 도중 패킷이 사라질 수 있음
- 패킷이 순서대로 도착하지 않을 수 있음
Ex) A, B, C, D (전송) -> D, C, A, B (도착)

TCP/UDP

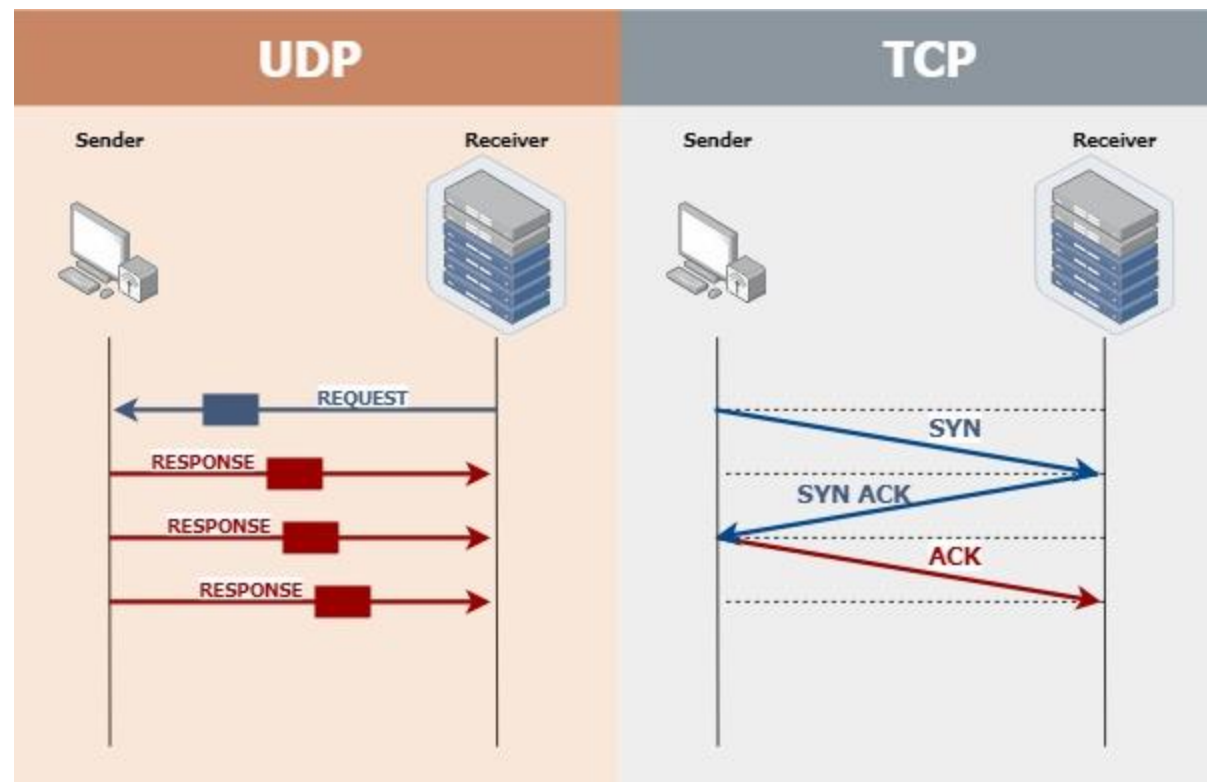
TCP (transmission Control Protocol)

- 전송 제어 프로토콜
- IP의 핵심 프로토콜 중 하나
- 연결 지향 (3 way handshake)
- 데이터 전달 보증
- 순서 보증
- 신뢰할 수 있는 프로토콜



UDP(User Datagram Protocol)

- 하얀 도화지에 비유(기능이 거의 없음)
- 데이터 전달, 순서 보장 X
- 속도가 빠르다.(제약이 없기때문에)
- 최신 HTTP에서는 UDP를 채택



Port

동시 다발적 통신을 위한 가상의 문을 뜻함

Port 사용예시 1

- IP 를 통해, 어떤 컴퓨팅 장치가 통신을 할 지 결정이 된다.
- 각 Node의 어떤 App에서 처리를 할 지 결정을 짓는데 Port가 사용된다.

Port 사용예시 2

- 웹서비스에서,
어떤 서비스를 선택할 지 포트번호에 따라 서버를 선택할 수 있다.

Well-known Port Number (0 ~ 1023)

- Document에서 Well-known 은 예약된 이라는 뜻으로 쓰인다.
 - 80 / 443 : HTTP / HTTPS
 - 22 : FTP / SSH

Registered Port (1024 ~ 49151)

- 여러 Tool이 쓰는 포트들
 - 3306 : MySQL
 - 8080 : HTTP 대체용

Dynamic Port (49152 ~ 65535)

- 필요할 때 마다 임의의 번호를 사용하는 곳

2장. Socket 사용하기

챕터의 포인트

- 소켓이란?
- AWE EC2 인스턴스 생성
- Echo 프로그램

소켓이란?

소켓이란 ?

- 컴퓨터 네트워크를 경유하는 프로세스 간 통신의 종착점
- 소켓 = Interface



- 전기 소켓 = 집 안에서 사용하는 Interface
- 내부 구조, 전기 원리, 전기 종류 등을 몰라도 **콘센트(Interface)** 만 꽂으면 사용 가능
- 마찬가지로, Socket 이라는 S/W Interface 가 존재하여 내부 프로토콜 원리를 몰라도 **Socket Interface**만 알고 있으면 즉시 통신 가능
- 소켓의 원리 보다는 구현 사용법 위주로 수업 진행

AWS EC2 인스턴스 생성

실습

- **AWS 를 활용한 서버 환경 구성**

- 불가피하게 AWS 를 사용할 수 없다면, Virtual Machine 에 클라이언트 & 서버 코드를 모두 작성
-> “내 컴퓨터“ 에 “내 컴퓨터“ 를 연결하는 방식 (127.0.0.1 에 연결)

- **소켓 프로그래밍 (echo 서버)**

AWS EC2 인스턴스 생성 (1/2)

- 서울 리전, “20.04” 검색하여 Ubuntu Server 20.04 LTS 선택

최근 사용

Quick Start

Amazon Linux

aws

macOS

Mac

Ubuntu

ubuntu®

Windows

Microsoft

Red Hat

Red Hat

S

더 많은 AMI 찾아보기

AWS, Marketplace 및 커뮤니티의 AMI 포함

Amazon Machine Image(AMI)

Ubuntu Server 20.04 LTS (HVM), SSD Volume Type

ami-0ea5eb4b05645aa8a (64비트(x86)) / ami-0678638ec320e38b6 (64비트(Arm))

가상화: hvm ENA 활성화됨: true 루트 디바이스 유형: ebs

프리 티어 사용 가능 ▼

AWS EC2 인스턴스 생성 (2/2) - 보안 그룹 구성

- 보안 그룹 이름, 설명 : 영어로 작성
- **ICMP 허용**
 - 모든 ICMP - IPv4
 - 소스 : Anywhere - IPv4
- **사용자 지정 TCP**
 - 소스 : Anywhere - IPv4
 - 포트 : 실습 때 사용할 포트(12345)

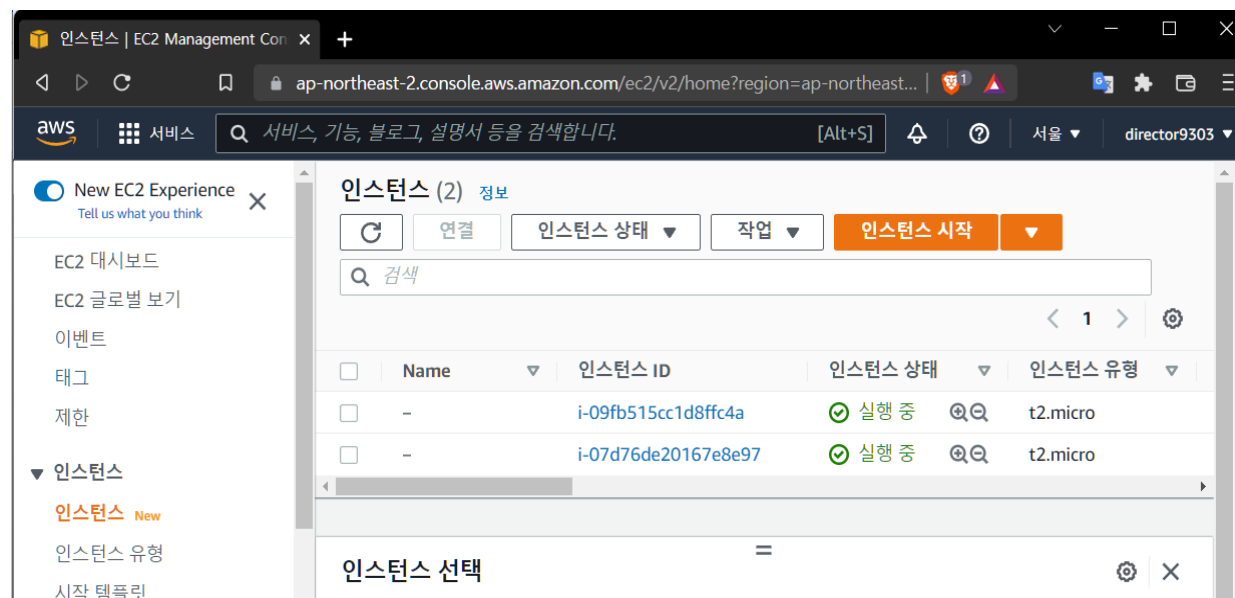
인바운드 규칙 정보

보안 그룹 규칙 ID	유형 정보	프로토콜 정보	포트 범위 정보	소스 정보	설명 - 선택 사항	정보
sgr-023e8e1b258ef5cbd	SSH ▼	TCP	22	사용자 ... ▼ Q 0.0.0.0/0 X		삭제
-	모든 ICMP - IPv4 ▼	ICMP	전체	Anywh... ▼ Q 0.0.0.0/0 X		삭제
-	사용자 지정 TCP ▼	TCP	12345	Anywh... ▼ Q 0.0.0.0/0 X		삭제

규칙 추가

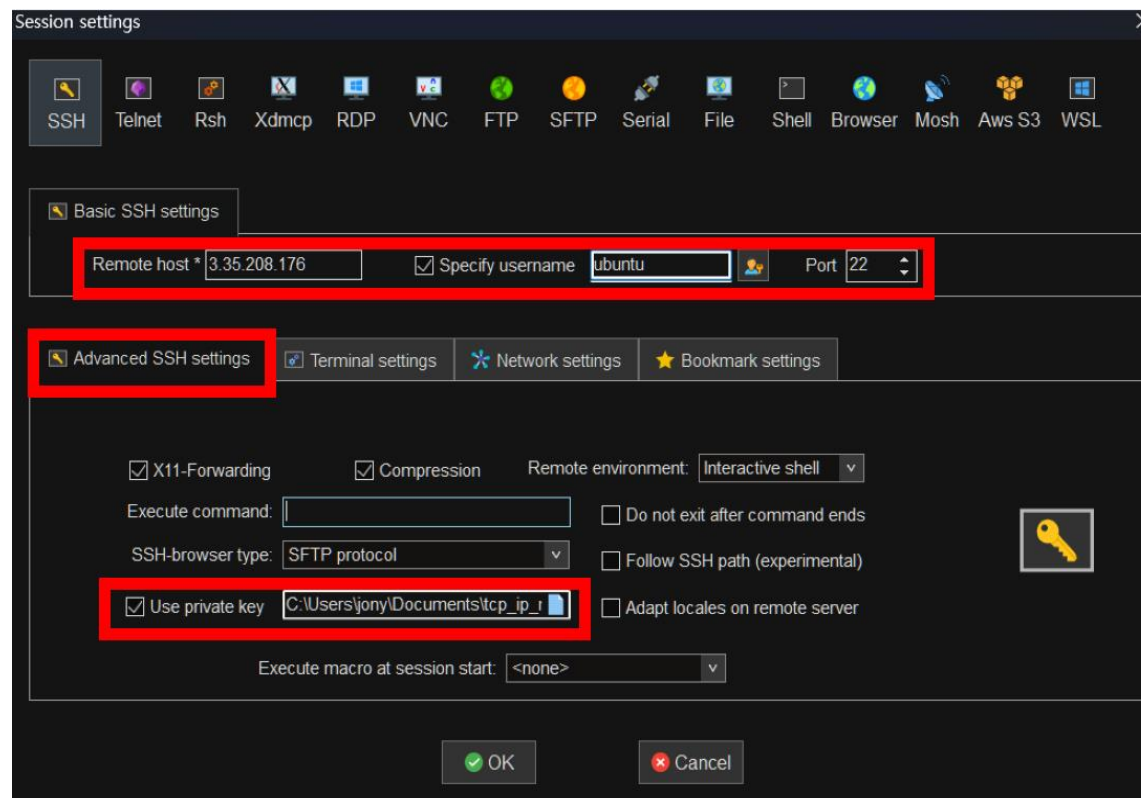
AWS EC2 인스턴스 접속 (1/3)

- 생성한 인스턴스의 IP 주소를 알아내어 SSH 접속할 예정
- 우측의 페이지로 가서 IP 주소를 알아냄



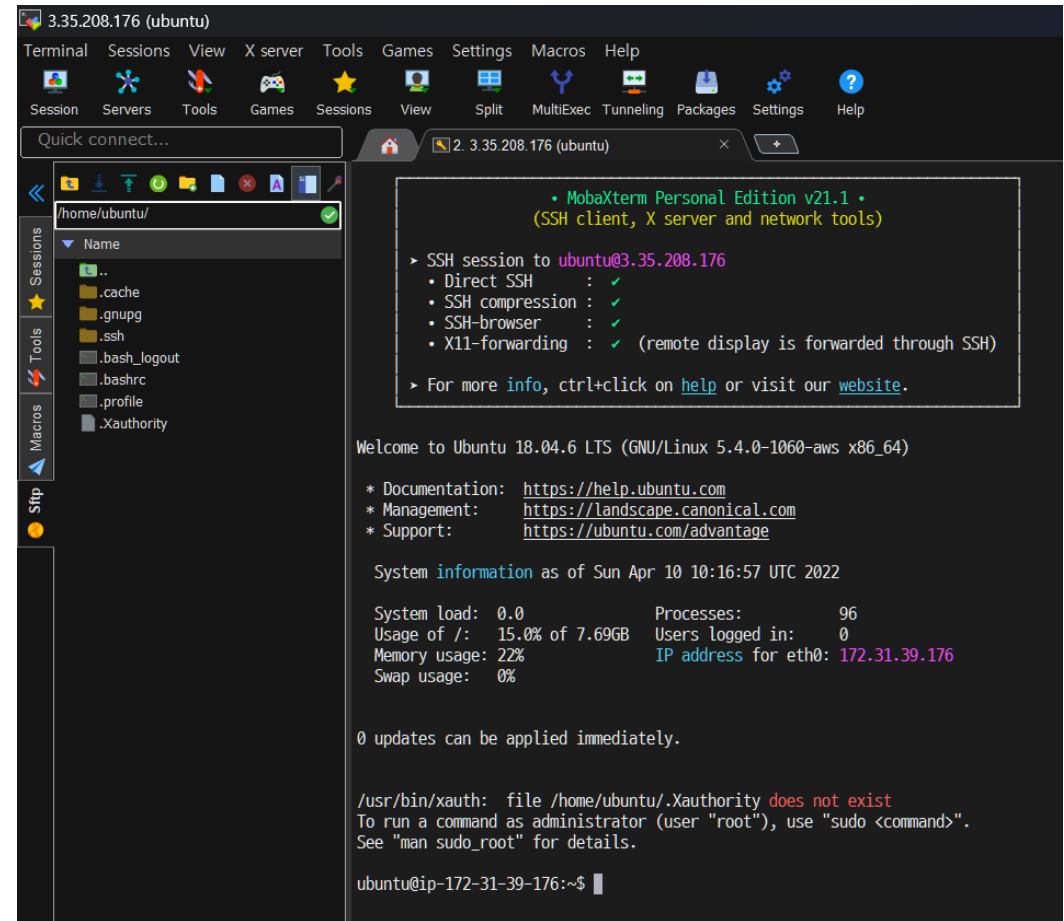
AWS EC2 인스턴스 접속 (2/3)

- MobaXterm 을 이용하여 접속
- Remote host : 인스턴스 IP
- username 입력 (aws 는 ubuntu)
- Port : 22 (SSH 접속)
- Advanced SSH Settings
 - 다운로드 받은 키 등록



AWS EC2 인스턴스 접속 (3/3)

- 성공적으로 접속한 화면
- 현재는 기본적인 Ubuntu 20.04 LTS 만 설치됨
- 추가적으로 간단한 몇 가지 세팅이 필요함



The screenshot shows a MobaXterm window titled '3.35.208.176 (ubuntu)'. The terminal displays the following output:

```
• MobaXterm Personal Edition v21.1 •
(SSH client, X server and network tools)

> SSH session to ubuntu@3.35.208.176
• Direct SSH : ✓
• SSH compression : ✓
• SSH-browser : ✓
• X11-forwarding : ✓ (remote display is forwarded through SSH)

> For more info, ctrl+click on help or visit our website.

Welcome to Ubuntu 18.04.6 LTS (GNU/Linux 5.4.0-1060-aws x86_64)

* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage

System information as of Sun Apr 10 10:16:57 UTC 2022

System load: 0.0          Processes: 96
Usage of /: 15.0% of 7.69GB Users logged in: 0
Memory usage: 22%        IP address for eth0: 172.31.39.176
Swap usage: 0%

0 updates can be applied immediately.

/usr/bin/xauth: file /home/ubuntu/.Xauthority does not exist
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-39-176:~$
```

서버 환경 설정 (1/5) - 미러 변경

- `sudo vi etc/apt/sources.list`
- `:%s/ap-northeast-2.ec2.archive.ubuntu/mirror.kakao/g`

```
deb http://mirror.kakao.com/ubuntu/ bionic-updates multiverse
# deb-src http://mirror.kakao.com/ubuntu/ bionic-updates multiverse

## N.B. software from this repository may not have been tested as
## extensively as that contained in the main release, although it includes
## newer versions of some applications which may provide useful features.
## Also, please note that software in backports WILL NOT receive any review
## or updates from the Ubuntu security team.
deb http://mirror.kakao.com/ubuntu/ bionic-backports main restricted universe multiverse
# deb-src http://mirror.kakao.com/ubuntu/ bionic-backports main restricted universe multivers
e

## Uncomment the following two lines to add software from Canonical's
## 'partner' repository.
## This software is not part of Ubuntu, but is offered by Canonical and the
## respective vendors as a service to Ubuntu users.
# deb http://archive.canonical.com/ubuntu bionic partner
# deb-src http://archive.canonical.com/ubuntu bionic partner

deb http://security.ubuntu.com/ubuntu bionic-security main restricted
# deb-src http://security.ubuntu.com/ubuntu bionic-security main restricted
"/etc/apt/sources.list" 57L, 3128C written
```

43,1 89%

서버 환경 설정 (2/5) - 패키지 리스트 업데이트

- 패키지 리스트 업데이트

```
$ sudo apt update
```

- 업그레이드는 따로 하지 않음

서버 환경 설정 (3/5) - 서버 시간 세팅

- 서버 시간 세팅

```
ubuntu@ip-172-31-39-176:~$ sudo timedatectl set-timezone Asia/Seoul
ubuntu@ip-172-31-39-176:~$ date
Sun Apr 10 19:25:03 KST 2022
ubuntu@ip-172-31-39-176:~$
```

- UTC 이므로 한국 시간인 KST 로 변경

```
$ sudo timedatectl set-timezone Asia/Seoul
```

```
$ date
```

서버 환경 설정 (4/5) - vi 세팅

- 다음을 .vimrc 에 작성하자

```
set nu  
set ts=4  
set sw=4  
set ls=2
```


서버 환경 설정 (5/5) - 빌드 환경 구성

- 빌드를 위하여 gcc 와 make 설치

```
$ sudo apt install gcc make
```

- 작업을 위한 디렉토리 생성

```
$ mkdir server-socket
```

```
$ cd server-socket
```

Echo 프로그램

Echo 서버 구성

- Echo 서버 - 사용자가 무엇을 입력하든 서버가 입력한 데이터를 그대로 되돌려주는 프로그램
- 파일이름 - echo_server.c

Echo 서버 실행

```
$ gcc echo_server.c -o echo_server
```

- 이렇게 서버 쪽에 생성한 소켓을 가리켜 “서버 소켓” 또는 “리스닝(Listening) 소켓” 이라고 한다.
- 이제 테스트를 해보자
 - 네트워크는 테스트가 제일 중요하다 !
- 클라이언트 코드는 아직 작성도 하지 않았는데, 어떻게 테스트가 가능한가요 ?
 - 우분투 내부적으로 NetCat 이라는 프로그램이 존재한다.

NetCat(넛캣)

- TCP, UDP 네트워크 연결을 통해 데이터를 읽거나 쓸 수 있도록 만든 유틸리티 프로그램
- NetCat 을 사용하는 이유
 - 서버 개발자 : 서버 코드를 책임지고 “테스트“ 해야함
 - 클라이언트 개발자 : 클라이언트 코드를 책임지고 “테스트“ 해야함
 - 서버와 클라이언트 코드를 전부 작성하고 연결하여 테스트하면 난이도가 높고 시간이 오래 걸린다
 - 각각을 테스트 한 후, 테스트 성공 시 “이어서 붙여“ 최종 테스트를 진행한다
 - cat : 단순 출력하는 Unix 명령어
 - net cat : network 상에서 사용하는 cat 이며 nc 라고 한다

NetCat 실습 (1/2) - AWS 에서 가짜 클라이언트 만들기

- 가짜 클라이언트 생성 명령어

```
$ nc [IP주소] [port]
```

1. 작성한 서버 실행

```
$ ./echo_server
```

2. 가짜 클라이언트 작동 - 터미널을 하나 더 열고 aws 접속 후 실행

```
$ nc 127.0.0.1 12345
```

- 127.0.0.1: “내 컴퓨터“ 를 의미. 즉, aws 서버를 말한다. 그 서버의 “12345 포트“ 에 접근하는 것이다.

NetCat 실습 (2/2) - 테스트 결과

- 에코 서버가 정상적으로 동작하는 것을 확인 할 수 있다

```
ubuntu@ip-172-31-40-15:~$ nc 127.0.0.1 12345  
즐거운 네트워크 수업시간!  
즐거운 네트워크 수업시간!
```

클라이언트 구성

- 사용자가 입력한 데이터를 Echo 서버에게 전송해주는 클라이언트 코드를 구성해보자
- 파일이름 - echo_client.c
- `$ gcc echo_client.c -o echo_client`

NetCat 실습 (1/2) - 가짜 서버 만들기

- 이번엔 클라이언트 테스트 이므로, 가짜 서버가 필요하다
- 가짜 서버 생성 명령어

```
$ nc -l [port]
```

1. 가짜 서버 실행

```
$ nc -l 12345
```

2. 클라이언트 코드 실행 - 터미널을 하나 더 켜 후 실행

```
$ ./echo_client
```

NetCat 실습 (2/2) - 테스트 결과

- 테스트 방법 : 파일명

```
jony@jony-ubuntu1804-server:~/client-socket$ ./echo_client
```

- 그리고 서버를 확인하면 클라이언트에서 전송한 내용을 확인할 수 있다.

```
jony@jony-ubuntu1804-server:~/socket-client-practice$ nc -l 12345  
you are awesome
```

- 서버 터미널에서 텍스트 입력 후 엔터 치면, 클라이언트 터미널에도 찍힌다.

서버와 클라이언트 모두 사용해보기 (1/2)

- NetCat 을 사용하지 않고 우리가 구성한 코드로 서버와 클라이언트를 테스트 해보자

1. 서버 실행

```
$ ./echo_server
```

- 정상적으로 실행되면 터미널이 멈출 것이다.

2. 클라이언트 실행

```
$ ./echo_client
```

서버와 클라이언트 모두 사용해보기 (2/2)

- 에코 서버와 클라이언트가 모두 정상적으로 동작한다.
- 동작 과정을 자세히 알아보자

안녕 싸피
안녕 싸피

3장. Socket 이해하기

챕터의 포인트

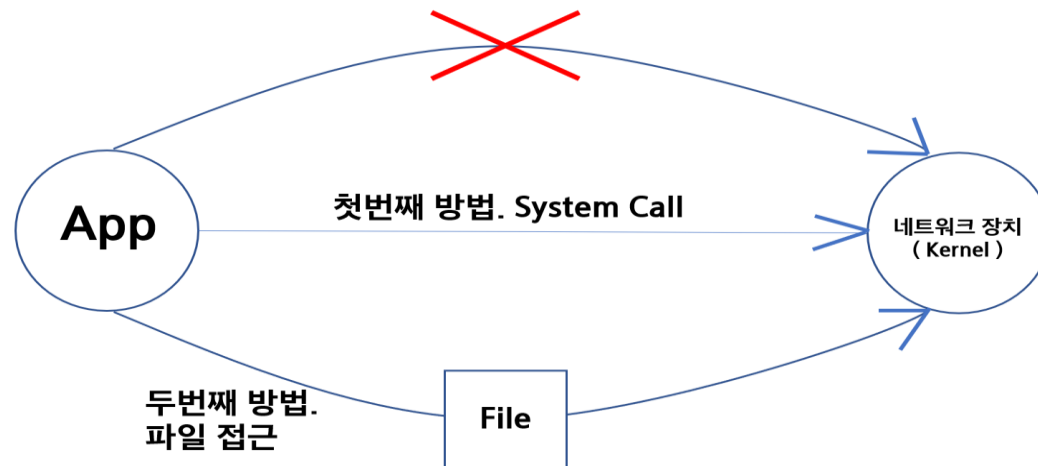
- 파일 입출력
- 소켓 주소 할당
- bind / listen / accept
- echo 클라이언트 분석

파일 입출력

리눅스의 특징

- 리눅스는 모든 것을 파일로 간주한다
 - 디렉토리, 장치 etc...
- 소켓 또한 파일의 일종으로 취급한다
 - 즉, **소켓 조작은 파일 조작과 동일하게 간주된다.**
 - **파일 입출력 함수를 소켓 입출력 함수(= 네트워크 송수신)에 사용할 수 있다는 뜻**
- 네트워크 송수신 함수를 다시 해석하면 다음과 같다
 - read() 사용 시 파일을 읽는다 -> “수신” 한다. 즉, 데이터를 받는다.
 - write() 사용 시 파일에 쓴다 -> “송신” 한다. 즉, 데이터를 보낸다.
 - 소켓 파일에 무엇인가 적으면 그대로 “송신”되며, 무엇인가 읽으면 그대로 “수신” 된다는 뜻
- Application 개발자는 소켓 파일만 건드리면, 바로 소켓 통신이 가능하다.

리눅스의 특징 - 그림으로 설명



- 리눅스 세상에서 Application 이 커널의 함수를 직접 호출하는 것은 금지 되어있다.
- 커널에게 어떤 것을 부탁하고 싶을 때는
 - 방법1. System Call
 - 방법2. 커널이 관리하는 특수 파일을 통하여 부탁하는 방법
- 소켓 프로그래밍은 그 중,
 - 방법2. 특수한 파일에 접근하여 커널에게 요청을 하고 결과를 받는다.

파일 디스크립터(File Descriptor)

- 소켓은 굉장히 복잡한 “파일”이다
- 시스템으로부터 할당 받은 파일 또는 소켓에 부여된 **정수**를 파일 디스크립터라고 한다.
- 복잡한 파일을 단순히 파일 디스크립터라는 정수로 “치환” 하여 작업의 편의성을 높인 것이다

소켓 주소 할당

TCP 기반 서버/클라이언트

- TCP 기반 서버에서 기본적인 함수 호출 순서는 다음과 같다.

- | | |
|---------------------|----------------|
| 1. socket() | 소켓 생성 |
| 2. bind() | 소켓에 주소 할당 |
| 3. listen() | 클라이언트 연결 요청 대기 |
| 4. accept() | 클라이언트 연결 승인 |
| 5. read() / write() | 통신 |
| 6. close() | 소켓 닫기 |

- 소켓은 인터페이스, 다른 말로는 프레임워크이기 때문에,
반드시 위 과정을 외워야 한다.

```
server_sock = socket(PF_INET, SOCK_STREAM, 0);
```

- socket() 성공 시 파일 디스크립터, 실패 시 -1
 - PF_INET IPv4
 - SOCK_STREAM 연결지향형 소켓
 - 두 가지를 만족시키는 건 TCP/IP 밖에 없으므로,
 - 맨 마지막 매개변수는 0
-
- 우리 수업에선 TCP/IP 소켓만 다루기 때문에, **소켓 생성은 이 구문 이외 다른 걸 쓰지 않는다.**

```
server_sock = socket(PF_INET, SOCK_STREAM, 0);  
if (server_sock == -1)  
{  
    printf("ERROR :: 1_Socket Create Error\n");  
    exit(1);  
}
```

옵션값 지정

- 서버 종료 후, 3분 정도 동일 포트에 재할당 불가능
- 원활한 실습을 위해, 이 기능을 강제로 켜는 코드

```
int optval = 1;  
setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, (void *)&optval, sizeof(optval));
```

소켓 주소 할당

- `socket()` 을 통하여 소켓을 생성하면 FD(파일 디스크립트) 값과 소켓의 유형이 지정된다.
- 이제 이 서버 소켓을 이용하여 클라이언트와 통신하기 위해서는 주소(IP & PORT) 를 할당해야 한다.
- IP 주소로 컴퓨터를 특정, PORT 번호로 소켓을 특정할 수 있어야 하기 때문

소켓 주소 할당

- 에코 프로그램 코드에서는 아래 부분에 속한다.

```
// 주소 설정
struct sockaddr_in server_addr = {0};
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(atoi(PORT));
```

- 소켓 뿐 아니라 임베디드 리눅스 개발, 커널 개발 등에서 흔히 사용되는 옵션 지정 방식이 잘 나와있다
 1. 속성값을 저장할 구조체 하나 만들기 (server_addr)
 2. 만들어진 구조체에 값을 다 채우기
 3. 만든 구조체 넘기기

소켓 주소 할당

- 소켓 주소를 할당하기 위해 주소를 담을 구조체를 초기화한다.
- 본 강의에서는 소켓 주소를 담을 구조체로 대표적인 sockaddr 과 sockaddr_in 구조체를 다룬다.

```
struct sockaddr_in server_addr = {0};
```

- 현재 server_addr 의 타입은 sockaddr_in 구조체이다.
 - 개발자가 IP, PORT 를 입력하기 편한 구조체
- 이것은 나중에 sockaddr 구조체로 형변환될 예정
 - bind() 에서 실제로 원하는 구조체

소켓 주소 할당

- 주소체계 지정

```
server_addr.sin_family = AF_INET;
```

- `sin_family`
 - 주소체계 지정
 - `AF_INET` 은 IPv4 를 의미. 본 강의에서는 IPv4 체계만 다룬다.

소켓 주소 할당

- IP 주소 배정

- sockaddr_in 구조체의 sin_addr.s_addr 에 ip 주소를 배정한다
- INADDR_ANY : 호스트 자기 자신의 IP를 의미

- PORT 배정

- sockaddr_in 구조체의 sin_port 에 port 배정

```
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);  
server_addr.sin_port = htons(atoi(PORT));
```

소켓 주소 할당 - IP 주소 배정 시 주의사항

- IP 주소는 “네트워크 바이트 순서” 로 넣어야 한다. (= 빅 엔디안)
- 전 세계 네트워크 앱은 반드시 “빅 엔디안” 방식을 사용하자고 약속되어 있다.
- 빅 엔디안 변경 함수
 - htonl() - int 형 IP 를 빅 엔디안으로 변경하는 함수
 - h : host 바이트 순서. 즉, 원래 cpu 의 엔디안 방식
 - to : ~로 바뀌라
 - n : network 바이트 순서. 즉, 빅 엔디안
 - l : long int 형으로 변경 (ip 는 long int 로 저장되어 있다)
 - htons() - int 형 PORT 를 빅 엔디안으로 변경하는 함수
 - s : short int 형으로 변경 (port 는 short int 로 저장되어 있다)
 - atoi : 문자열을 int 로

```
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);  
server_addr.sin_port = htons(atoi(PORT));
```

bind / listen / accept

주소 정보를 실제 소켓에 등록 - bind()

- 이때까지 할당한 주소 정보를 실제 소켓에 등록해야 한다
- `server_sock`: 소켓 파일 디스크립터 (정수형)
- `(struct sockaddr *)&server_addr`
 - `sockaddr_in` 타입의 `server_addr` 을 `sockaddr` 타입으로 형변환
- `sizeof(server_addr)`: `server_addr` 의 사이즈
- 실패 시 -1 을 반환

```
// bind
if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1)
{
    printf("ERROR :: 2_bind Error\n");
    exit(1);
}
```

연결요청 대기상태 - listen()

- 클라이언트의 접속을 대기 - 접속이 이루어지지 않으면 다음 단계로 절대 넘어가지 않는다.
 - server_sock : bind() 를 거쳐서 주소가 할당된 소켓의 파일 디스크립터
 - 5 : 연결요청 대기 큐의 사이즈
- 연결요청 대기상태
 - 클라이언트의 연결요청 시, 연결이 수락될 때 까지 요청 자체를 대기시킬 수 있는 상태
- 연결요청 대기 큐
 - 연결요청이 들어가는 대기실
 - 크기를 지정해주어야 함 (개발자 판단에 의존)
 - 웹 서버 같이 연결 요청이 많을 경우 최소 15 이상
 - 대기 큐의 크기는 최대 동시접속 클라이언트 수가 아니다

```
if (listen(server_sock, 5) == -1)
{
    printf("ERROR :: 3_listen Error");
    exit(1);
}
```

클라이언트의 연결요청 수락 - accept()

- 통신을 위해서는 서버가 클라이언트의 요청을 수락해야 한다.
- **accept 함수의 역할**
 1. 함수 호출 성공 시 내부적으로 데이터 입출력에 사용할 소켓 생성
 2. 해당 소켓의 파일 디스크립터를 반환
 3. 연결 요청을 한 클라이언트 소켓을 연결
- **accept() 과정까지 끝나면, 실제 데이터를 주고받으면 된다.**

• 필요한 변수 준비

- 파일 디스크립터, 주소 구조체, 길이 선언
- `client_sock` 이 배열이라면 동시에 여러 클라이언트에게 서비스 가능

• 무한루프 존재 이유

- 무한루프가 아니라면, 한 명의 클라이언트의 연결이 끊겼을 때, 서버도 종료됨
- 즉, 지속적인 서비스 위해 무한루프 사용
 - 사용자 1 끝나면 사용자 2 ...
- 무한루프가 끝나는 조건
 - Ctrl + C 입력 시
 - `interrupt` 함수 동작
 - `accept` 실패 시

```
client_sock = 0;  
struct sockaddr_in client_addr = {0};  
socklen_t client_addr_len = sizeof(client_addr);
```

```
while (1)  
{
```

클라이언트의 연결요청 수락 - accept()

- 매번 새로운 client 의 주소 정보를 받기 위해, client_addr 초기화
- 연결요청 대기 큐 에서 대기중인 클라이언트의 연결 요청을 수락
 - client_sock : accept 의 리턴값, 즉 파일 디스크립터 (accept 된 클라이언트의 소켓이 배정됨)
 - server_sock : bind(), listen() 을 거쳐 만든 서버 소켓의 파일 디스크립터
 - (struct sockaddr *)&client_addr : 클라이언트의 주소 정보
 - &client_addr_len : client_addr 의 크기를 받을 변수의 주소값
- 즉, 대기실에 있는 클라이언트를 하나씩 받아들임

```
while (1)
{
    memset(&client_addr, 0, sizeof(client_addr));
    client_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_addr_len);
    if (client_sock == -1)
    {
        printf("ERROR :: 4_accept Error\n");
        break;
    }
}
```

• 무한루프 존재 이유

- 클라이언트가 하나의 echo 를 완료했다고 클라이언트의 접속을 끊어버리면 안된다.
- 즉, 이 프로그램은 이중 무한루프 사용
- 맨 처음, 메시지 초기화
- read: 수신
 - 성공 시 buf 길이
 - 실패 시 0 반환 - 연결 끊김
- 종료 조건
 1. 연결이 끊겼거나(len==0)
 2. client 에서 exit 를 입력했거나둘 중 하나라도 해당되면 루프 탈출 후 "클라이언트 소켓" 종료
- write: 송신
 - 받은 메시지를 그대로 클라이언트에게 전송

```
char buf[100];
while (1)
{
    memset(buf, 0, 100);
    // 클라이언트에서 보낸 메시지를 buf 에 담음
    int len = read(client_sock, buf, 99);
    // remove '\n'
    removeEnterChar(buf);
    // client 와 연결이 끊어졌을 때 클라이언트 종료
    if (len == 0)
    {
        printf("INFO :: Disconnect with client... BYE\n");
        break;
    }
    // client 에서 exit 입력했을 때 클라이언트 종료
    if (!strcmp("exit", buf))
    {
        printf("INFO :: Client want close... BYE\n");
        break;
    }
    // 받은 메시지를 그대로, 클라이언트에게 전송
    write(client_sock, buf, strlen(buf));
}
close(client_sock);
```

- 서버 소켓의 종료 시점
 - accept 실패, 또는 Ctrl + C 입력

```
// 서버 소켓 닫기  
close(server_sock);  
return 0;
```

```
}
```

echo 클라이언트 분석

TCP 기반 서버/클라이언트

- TCP 기반 클라이언트에서 기본적인 함수 호출 순서는 다음과 같다.

- | | |
|---------------------|---------|
| 1. socket() | 소켓 생성 |
| 2. connect() | 연결 요청 |
| 3. read() / write() | 데이터 송수신 |
| 4. close() | 연결 종료 |

- 클라이언트는 서버와 프로그램의 목적이 다르기 때문에 다른 함수를 사용한다.
- 서버와 마찬가지로 위의 흐름도 반드시 암기하여야 한다.

서버/클라이언트 소켓의 차이

- 서버와 클라이언트의 소켓 초기화 목적을 생각해보자

- server - IP 211.217.168.13, PORT 9190 으로 “들어오는 데이터는 전부 다 나에게 오라!”
- client - IP 211.217.168.13, PORT 9190 으로 “연결해라”

	서버	클라이언트
IP	서버 소켓이 동작하는 컴퓨터의 IP 주소	통신을 원하는 원격지 서버의 IP 주소
PORT	소켓이 “위치할” 포트 번호	소켓이 “위치한” 포트 번호

- 초기화 코드가 비슷해 보여도 전혀 다른 목적을 위해 존재한다.

inet_addr(IP)

- 현재 IP의 타입은 string
 - 56.32.123.88
- 이것을 빅 엔디안 long int로 바꿔야
- inet_addr()는, IP를 빅 엔디안 long int로 파싱해주는 함수다.

```
struct sockaddr_in addr = {0};  
addr.sin_family = AF_INET;  
addr.sin_addr.s_addr = inet_addr(IP);  
addr.sin_port = htons(atoi(PORT));
```


클라이언트의 연결 요청 - connect()

- bind() 와 거의 유사한 코드

```
// Connect
if (connect(sock, (struct sockaddr *)&addr, sizeof(addr)) == -1)
{
    printf("ERROR :: 2_Connect Error\n");
    exit(1);
}
```

- bind() : 서버 소켓이 될 예정인 소켓에 주소를 배정하여 서버 통신이 가능하도록 하는 함수
- connect() : 연결할 서버의 주소 정보를 받는 것. 즉, 원격 연결할 서버의 주소 정보를 소켓에 담는 함수
 - sock : connect 성공 시 소켓의 파일 디스크립터
 - serv_addr : 서버의 주소 정보

클라이언트의 종료 동작과정

- 클라이언트의 `close()` 조건
 - 사용자가 `exit` 입력
 - 서버가 중단되어 `read` 의 값이 0일 때
- 클라이언트에서 소켓을 `close()` 하면 서버에 EOF 가 전달된다.
 - 읽었을 때, `len` 이 0 이 되는 경우는,
 - 클라이언트 소켓에서 EOF 를 전달했을 때다.
 - 이 때 더 이상 클라이언트로 `write` 하지 않고 `while` 을 탈출하게 된다.

```
int len = read(client_sock, buf, 99);
```

```
// client 와 연결이 끊어졌을 때 클라이언트 종료
if (len == 0)
{
    printf("INFO :: Disconnect with client... BYE\n");
    break;
}
```

- 각각의 소켓은 언제 종료되는가?
 1. 서버 소켓
 2. 서버의 클라이언트 소켓
 3. 클라이언트의 클라이언트 소켓
- 정확한 종료 시점을 모두 이해해야 제대로 된 소켓 프로그램을 만들 수 있다.

서버와 클라이언트의 동작 (1/4)

1. 서버를 실행하면, 서버는 12345 포트를 서버 소켓으로 열어두게 된다.

```
$ ./echo_server
```

2. `listen()` 을 거치면서, 서버 소켓은 클라이언트가 접속할 때까지 기다린다.

```
// listen
if (listen(server_sock, 5) == -1)
{
    printf("ERROR :: 3_listen Error");
    exit(1);
}
```

서버와 클라이언트의 동작 (2/4)

3. 클라이언트를 실행시킨다

```
$ ./echo_client
```

4. 클라이언트는 connect() 를 거치며 연결 요청을 보냈다.

```
// Connect
if (connect(sock, (struct sockaddr *)&addr, sizeof(addr)) == -1)
{
    printf("ERROR :: 2_Connect Error\n");
    exit(1);
}
```

서버와 클라이언트의 동작 (3/4)

4.1 이 때, 서버는 대기중인 클라이언트를 accept() 한다.

```
// accpet
client_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_addr_len);
if (client_sock == -1)
{
    printf("ERROR :: 4_accept Error\n");
    break;
}
```

5. 연결이 성공한 클라이언트는 사용자의 입력을 기다린다.

- 문자열을 입력하고 엔터를 치면 서버로 전송한다.
- 이 과정을 종료 조건(exit, ctrl+c)을 만족시키지 전까지 반복한다.

서버와 클라이언트의 동작 (4/4)

6. 서버는 클라이언트가 보낸 메시지를 읽은 후, 받은 걸 그대로 돌려준다.

```
while (1)
{
    memset(buf, 0, 100);
    int len = read(client_sock, buf, 99);
```

```
write(client_sock, buf, strlen(buf));
```

7. 클라이언트는 서버에서 보낸 메시지를 read 한다.

```
memset(buf, 0, 100);
int len = read(sock, buf, 99);
if (len == 0)
{
    printf("INFO :: Server Disconnected\n");
    break;
}
printf("%s\n", buf);
```

- echo_client.c 와 echo_server.c 를 완벽히 이해하자.
 - TCP/IP 소켓은 프레임워크다.
 - 규칙만 지키면 어렵지 않다.