

윈도우 네트워크 프로그래밍

TCP/IP

▶ 종단 시스템(end-system)/호스트()

- 최종 사용자(end-user)를 위한 **어플리케이션을 수행하는 주체**(인터넷이 연결된 PC, 스마트폰 등등)

▶ 라우터(router)

- 종단 시스템에 속한 네트워크와 다른 네트워크를 연결, **서로 다른 네트워크에 속한 종단 시스템끼리 상호 데이터를 교환** 할 수 있도록 하는 장비

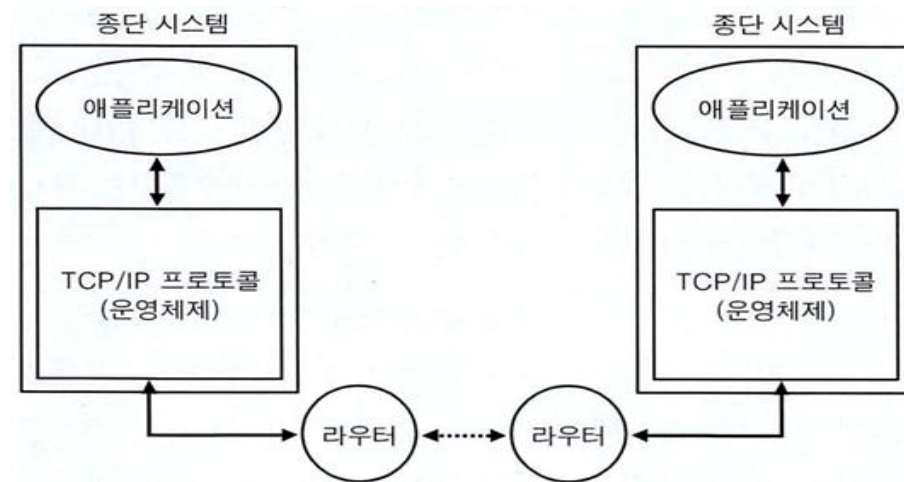
▶ 프로토콜

- 종단 시스템간 통신 수행을 위한 **정해진 절차와 방법**

TCP/IP

▶ TCP/IP 프로토콜

- 인터넷의 핵심 프로토콜이 **TCP와 IP**이며 이를 **포함한 각종 프로토콜**을 TCP/IP 프로토콜이라 한다



TCP/IP 프로토콜을 이용한 통신

TCP/IP

- ▶ 일반적으로 프로토콜은 기능별로 나누어 **계층적(layer)으로 구현한다**

- ▶ **네트워크 액세스 계층(network access layer)**

- 물리적 네트워크를 통한 실질적인 데이터 전송 담당

물리적 주소 : ~~10-10-00-10-00-00~~

- ▶ **인터넷 계층(internet layer)**

- 네트워크 액세스 계층의 도움을 받아, 전송 계층이 내려 보낸 데이터를 종단 시스템(호스트)까지 전달하는 역할
- 물리적 주소를 사용하지 않고 논리 주소인 IP 주소(internet Protocol address)를 사용

애플리케이션 계층	TELNET, FTP, HTTP, SMTP, MIME, SNMP, ...
전송 계층	TCP, UDP
인터넷 계층	IP
네트워크 액세스 계층	디바이스 드라이버 네트워크 하드웨어

TCP/IP

▶ 인터넷 계층(internet layer)

- 데이터 전송을 위해서는 전송 경로가 필요하기 때문에 **전송 경로를 알아오기 위한 라우팅(routing) 작업이 필요**
- 라우팅 : 목적지까지 데이터를 보내기 위한 일련의 작업, 전용 컴퓨터를 라우터(router)라 부른다

▶ 전송 계층(transport layer)

- 최종적인 통신 목적지를 정하고, **오류 없이 데이터를 전송하는 역할**
- 해당 프로세스를 지정하는 일종의 주소인 **포트 번호(port number)를 사용**
- TCP(Transmission Control Protocol)과 UDP(User Datagram Protocol)를 사용

▶ 애플리케이션 계층(application layer)

- ▶ 전송 계층을 기반으로 하는 다수의 프로토콜과 이 프로토콜을 이용하는 응용 프로그램(application)을 포괄한다(Telnet, FTP, HTTP, SMTP...)
- ▶ **소켓을 이용한 네트워크 애플리케이션도 여기에 속한다**

TCP/IP

▶ TCP와 UDP의 특징 비교

TCP	UDP
연결형(connection-oriented) 프로토콜 - 연결에 성공해야 통신이 가능	비연결형(connectionless) 프로토콜 - 연결 없이 통신 가능
데이터 경계를 구분하지 않음 - 바이트 스트림(byte-stream) 서비스	데이터 경계를 구분 - 데이터그램(datagram) 서비스
신뢰성 있는 데이터 전송 - 데이터를 재전송함	비신뢰적인 데이터 전송 - 데이터를 재전송하지 않음
1 대 1 통신(unicast)	1 대 1 통신(unicast) 1 대 다 통신(broadcast) 다 대 다 통신(multicast)

TCP/IP

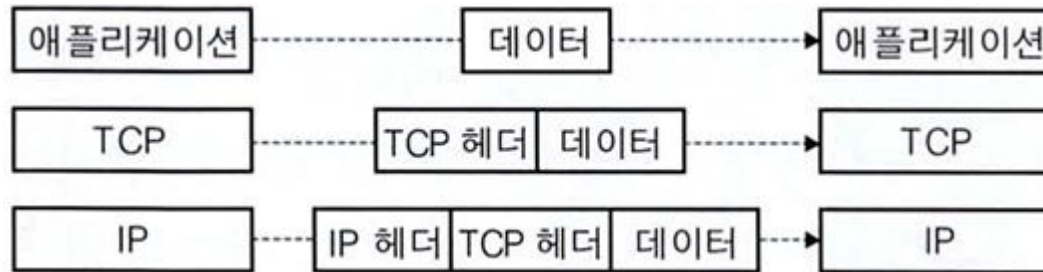
▶ 패킷 전송 원리

- ▶ 애플리케이션에서 보내는 데이터를 목적지까지 전송하기 위해서는 각각의 프로토콜에서 정의한 제어 정보(IP주소, 포트 번호, ...)가 필요
- ▶ 제어 정보는 위치에 따라 앞에 붙는 헤더(header)와 뒤에 붙는 트레일러(trailer)로 나누며, 이러한 제어 정보가 결합된 형태의 실제 전송하는 데이터를 패킷(packet)이라 부른다
- ▶ 패킷(packet) = 제어 정보 + 데이터

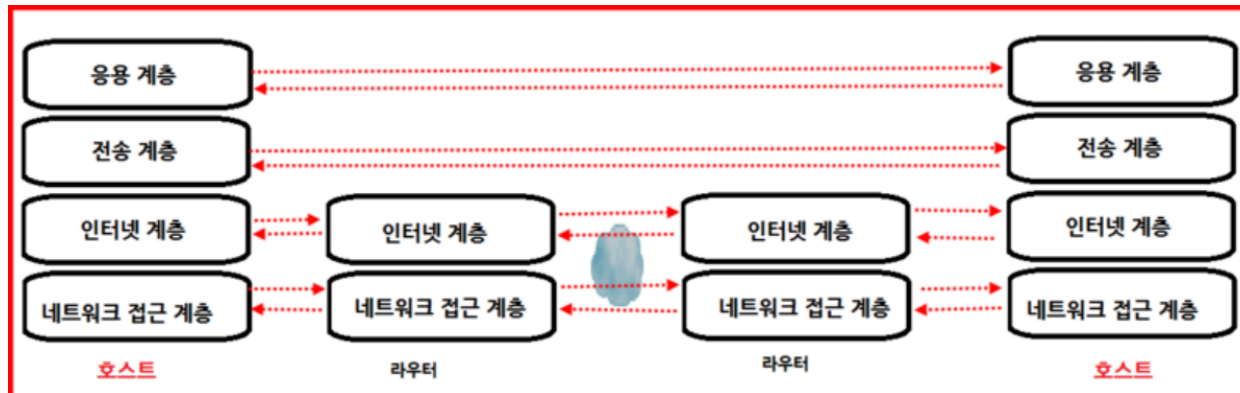


TCP/IP

▶ 계층 관점에서 본 패킷 전송 형태



▶ 계층간의 통신



TCP/IP

▶ IP 주소

- IPv4: 32비트, 8비트 단위 4부분으로 구분, 10진수로 표기(예) 255.255.255.255)
- IPv6: 128비트, 16비트 단위 8부분으로 구분, 16진수로 표기
(2020:230:abcd:ffff:0000:0000:ffff:1111)
- 폐쇄된 네트워크이거나 IP를 공유하는 경우가 아니라면 **IP 주소는 전세계적으로 유일한 값**을 가진다
- IPv4형식의 주소가 고갈될 것으로 보여 IPv6형식의 주소가 만들어 졌다

▶ 포트 번호

- 전송된 데이터가 **어느 프로세스(process)에서 사용되는지 알기 위한 식별자**
- 16비트의 정수, 0 ~ 65535(2^{16} 개)의 범위를 사용 가능
- 0 ~ 1023은 용도가 정해져 있어 함부로 사용하면 안된다
- 일반적으로 **1024 ~ 49151 범위의 값을 사용**한다

포트 번호	분류
0 ~ 1023	Well-known port(유명한, 알려진 서버)
1024 ~ 49151	Registered port(기업 등에서 관리를 위한 포트)
49152 ~ 65535	Dynamic and/or private port(일반 사용자들이 자유롭게 사용 가능한 포트)

TCP/IP

▶ 클라이언트/서버 모델

- 두 개의 애플리케이션이 상호 작용하는 방식을 나타내는 용어
- 프로세스간 통신(IPC, Inter-Process Communication) 기법을 이용하여 상호 정보를 교환
- 서로 다른 종단 시스템에서 실행된 프로세스가 있고 서로 접속을 하려고 할 때 접속이 성공하려면 반드시 상대의 프로세스가 실행 중이어야 하는데 **타이밍의 문제로 접속 실패할 확률이 높아** 이를 **보안**하기 위한 방법
- **먼저 실행되어 대기하는 서버(server)와 나중에 실행되는 클라이언트로(client) 나누어** 처리하도록 한다
- **클라이언트** : 서버에 접속하기 위한 **IP 주소(or 도메인 이름)와 포트 번호를 알고있어야 한다**
- **서버**:클라이언트에서 보내는 **패킷에 정보가 있어 주소 등을 미리 알 필요가 없다**

소켓(socket)

▶ 소켓(Socket)

- 소프트웨어로 작성된 추상적인 개념의 **통신 접속점**
- 네트워크 애플리케이션은 **소켓을 통하여** 통신망의 **데이터를 송수신** 한다

▶ 소켓의 개념을 바라보는 관점

- 데이터 타입
- 통신 종단점(communication end-point)
- 네트워크 프로그래밍 인터페이스

● 데이터 타입 관점의 소켓

- 파일 디스크립터(file descriptor) 혹은 핸들(handle)과 유사한 개념(**통신을 위해 관리하는 데이터를 간접적으로 참조할 수 있게 한다**)
- **파일 입출력과 유사한 형태**를 지녔다
- 통신과 관련된 다양한 작업을 할 수 있는 간편한 데이터 타입

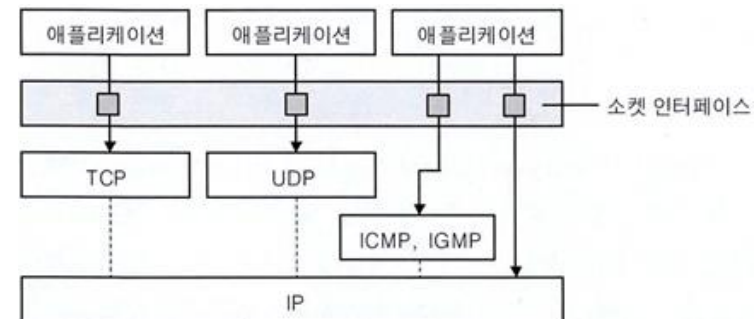
소켓(socket)

- 통신 종단점 관점의 소켓

- 통신을 하기 위한 다섯 가지 정보(프로토콜, 송신측 IP 주소, 포트 번호, 수신측 IP 주소, 포트번호)의 집합체
- 클라이언트 소켓이 서버 소켓으로 send()를 호출하여 **데이터를 보내고**, 서버의 소켓이 클라이언트 소켓에서 보낸 데이터를 recv() 함수를 호출하여 **받는다**

- 네트워크 프로그래밍 인터페이스 관점의 소켓

- 하나의 네트워크 프로그래밍 인터페이스
- **양쪽 모두 소켓을 사용할 필요는 없다**
- 양쪽 모두 **동일한 프로토콜**을 사용, **정해진 형태와 절차**에 따라 데이터를 주고 받아야 한다
- 일반적으로 애플리케이션 계층과 전송 계층 사이에 위치하는 것으로 간주하여 전송 계층을 건너뛰고 인터넷 계층과 연결하는 것도 가능



윈도우 소켓(windows socket)

▶ 윈도우 소켓

- 버클리 유닉스(Berkeley Software Distribution UNIX)에서 사용하던 네트워크 인터페이스인(소켓)을 **윈도우 환경에서 사용할 수 있게 만든 것**
- Windows Socket을 줄여 **원속(winsock)**이라 부른다

▶ 특징(유닉스 소켓 프로그래밍과의 차이)

- DLL(Dynamic-Link Library)을 통하여 대부분의 기능을 제고하기 때문에 DLL 초기화와 종료 작업을 위한 함수가 필요
- 윈도우 애플리케이션은 보통 GUI(Graphical User Interface)를 기반으로 하며, 메시지 구동 방식으로 동작하기에 이를 위한 확장 함수가 존재 한다
- **운영체제 차원에서 멀티쓰레드(multithread)를 지원**하여, 이런 환경에서 안정적으로 동작하기 위한 구조와 이를 위한 함수가 필요

윈도우 소켓(windows socket)

▶ 장점

- 유닉스 소켓과 소스 코드 수준에서 호환성이 높다
- 가장 널리 쓰이는 네트워크 프로그래밍 인터페이스, 여러 환경에서 사용할 수 있다
- TCP/IP 외에도 다양한 종류의 프로토콜을 지원하여 프로토콜 변경에 용이하다
- 세부적인 제어가 가능하여 고성능의 네트워크 애플리케이션 개발이 가능하다

▶ 단점

- 애플리케이션 수준의 프로토콜을 프로그래머가 직접 설계하여야 한다
- 서로 다른 바이트 정렬(byte ordering) 방식을 사용하거나 데이터 처리 단위가 서로 다른 종단 시스템간 통신을 할 경우, 애플리케이션 수준에서 데이터 변환 처리가 필요하다

원속 초기화/종료

▶ WSAStartup()

- 모든 원속 프로그램에서 소켓 API를 호출하기 전에 반드시 해당 초기화 함수를 호출해야 한다

```
WSADATA wsa;
```

```
if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) return -1;
```

▶ WSACleanup()

- 원속 사용 중지함을 운영체제에 알리고 관련 리소스를 반환하는 역할을 한다
- 함수 호출 실패할 경우 WSAGetLastError() 함수로 알 수 있다

원속 초기화/종료

▶ 새 프로젝트 => 콘솔 응용 프로그램

- 프로젝트 속성 => 구성 속성 => 고급 => 문자 집합: 멀티바이트 문자 집합 사용

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#pragma comment(lib, "ws2_32.lib") // 원속 라이브러리 사용을 알린다.
#include <WinSock2.h> // 원속 사용을 위하여 헤더 파일 추가.

int main()
{
    // 원속을 이용하여 작업을 하겠다. 원속 초기화.
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) return -1;
    MessageBox(NULL, "원속을 사용할 준비가 되었다.", "원속 사용 준비 완료.", MB_OK);
    // 원속 사용이 끝났다.
    WSACleanup();
}
```


오류 처리

- ▶ **오류 처리를 할 필요가 없는 경우**

- 리턴 값이 없거나 호출 시 항상 성공하는 일부 소켓 함수

- ▶ **리턴 값만으로 오류를 처리하는 경우**

- WSAStartup() 함수

- ▶ **리턴 값으로 오류 발생을 확인, 구체적인 내용은 오류 코드를 이용하여 확인하는 경우**

- 대부분의 소켓 함수

오류 처리

▶ WSAGetLastError()

- 소켓 함수 호출 결과, 오류가 발생할 경우 해당 함수를 이용하여 **오류 코드**를 얻을 수 있다

사용 예

```
if (소켓 함수() == 오류)
{
    int error_code = WSAGetLastError();
    std::cout << error_code에 해당하는 오류 메시지;
}
```

오류 처리

▶ FormatMessage()

- WSAGetLastError()에서 얻은 **오류 코드를 오류 메시지로** 자동으로 **변경**시켜 준다

```
FormatMessage(DWORD dwFlags, LPCVOID lpSource, DWORD dwMessageId, DWORD dwLanguageId, LPSTR lpBuffer, DWORD nSize, va_list *Arguments);
```

dwFlags : **FORMAT_MESSAGE_ALLOCATE_BUFFER** (오류 메시지 저장 공간을 함수가 알아서 할당한 다) | **FORMAT_MESSAGE_FROM_SYSTEM**(운영체제로부터 오류 메시지를 가져온다)

lpSource : **NULL**

dwMessageId : **WSAGetLastError()**

dwLanguageId : 오류 메시지를 표시할 언어, **MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT)** <= 사용자가 제어판에 설정한 기본 언어를 사용

lpBuffer : 오류 메시지의 시작 주소가 저장된다. 오류 메시지 사용이 끝나면 LocalFree() 함수 로 할당된 메모리를 반환 하여야 한다

nSize : **0**

Arguments : **NULL**

오류 처리

```
void err_quit(const char* msg)
{
    LPVOID lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL, WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf, 0, NULL);

    // msg : 메시지 박스의 타이틀(Caption)
    MessageBox(NULL, (LPCSTR)lpMsgBuf, msg, MB_ICONERROR);

    // 메모리 해제, 핸들 무효화.
    LocalFree(lpMsgBuf);

    exit(-1);
}
```

- 오류 메시지를 메시지 박스에 출력 후 애플리케이션을 종료 시킨다

오류 처리

```
void err_display(const char* msg)
{
    LPVOID lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL, WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf, 0, NULL);

    printf("[%s] %s\n", msg, (LPCSTR)lpMsgBuf);

    LocalFree(lpMsgBuf);
}
```

- 오류 메시지를 출력하고 애플리케이션을 종료하지 않는다
- 사소한 오류가 발생할 경우 사용

소켓 생성/닫기

- ▶ `SOCKET` `socket(int af, int type, int protocol)`
 - `af` : 주소 체계를 지정, `AF_INET`(internetwork : TCP, UDP, etc를 사용)
 - `type` : 소켓 타입 지정, `SOCK_STREAM` 또는 `SOCK_DGRAM` 사용
 - `protocol` : 사용할 프로토콜 지정, 프로토콜 결정에 모호함이 없다면 0을 사용

소켓 타입		특성
SOCK_STREAM		신뢰성 있는 데이터 전송 제공, 연결형 프로토콜
SOCK_DGRAM		비신뢰적인 데이터 전송 기능 제공, 비연결형 프로토콜
사용할 프로토콜	주소 체계	소켓 타입
TCP	AF_INET	SOCK_STREAM
UDP	AF_INET	SOCK_DGRAM

- ▶ `int` `closesocket(SOCKET s)`
 - 소켓을 닫고 관련 리소스를 반환

소켓 주소 구조체(socket address structures)

- ▶ 네트워크 프로그램에서 필요로 하는 주소 정보를 담고 있으며, 다양한 소켓 함수의 인자를 사용
- ▶ 여러 소켓 주소 구조체 중 가장 기본이 되는 구조체

```
struct SOCKADDR {  
    unsigned short sa_family; // 2 바이트.  
    char sa_data[14];         // 14 바이트.  
} // 16 바이트.
```

sa_family : 주소 체계, 부호 없는 16비트 정수 값 사용

sa_data[14] : 해당 주소 체계에서 사용하는 주소의 정보

소켓 주소 구조체(socket address structures)

```
int main()
{
    // 윈속 초기화.
    ...

    // socket() listen_socket 생성.
    SOCKET listen_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (INVALID_SOCKET == listen_socket) err_quit("socket");

    // 계속 작업 진행...

    // closesocket(listen_socket)
    closesocket(listen_socket);

    // 윈속 사용이 끝났다.
    ...
}
```


소켓 주소 구조체(socket address structures)

- ▶ 실제 프로그래밍에서는 애플리케이션이 사용할 프로토콜에 맞는 소켓 주소 구조체를 사용한다
- ▶ 주로 사용되는 TCP/IP 프로토콜(IPv4)

```
struct SOCKADDR_IN {  
    unsigned short sin_family; // 2 바이트.  
    unsigned short sin_port;   // 2 바이트.  
    IN_ADDR sin_addr;          // 4 바이트. IPv4 Internet address 구조체.  
    char sin_zero[8];          // 8 바이트.  
} // 16 바이트.
```

sin_family : 주소 체계

sin_port : 포트 번호

sin_addr : IP 주소, in_addr 구조체

sin_zero : 사용되지 않는다. 0.

바이트 정렬 함수

- ▶ 바이트 정렬 : 메모리 데이터를 저장할 때의 바이트 순서
 - 빅 엔디안(big-endian) : 최상위 바이트(Most Significant Byte)부터 차례로 정렬
 - 리틀 엔디안(little-endian) : 최하위 바이트(Least Significant Byte)부터 차례로 정렬
 - **의도하지 않은 프로세스로 데이터 전달될 위험이 있으므로 IP 주소, 포트 번호는 빅 엔디안**을 사용하기로 약속되어 있다
 - 네트워크 용어 : **빅 엔디안 = 네트워크 바이트 정렬**(network byte ordering)
 - 시스템 내의 고유한 바이트 정렬을 **호스트 바이트 정렬**(host byte ordering)

```
u_short htons( u_short hostshort); // host to network short
```

```
u_long htonl(u_long hostlong); // host to network long
```

```
u_short ntohs(u_short hostshort);
```

```
u_long ntohl(u_long hostlong);
```

일반적으로 값을 **넘겨주기** 위하여 **hton*()**를 사용하고 결과로 받아 사용할 때 **ntoh*()** 형식을 사용한다

바이트 정렬

```
int main()
{
    // 원속 초기화.
    ...

    // socket() listen_socket 생성.
    ...

    // 서버 정보 객체 설정.
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(9000);
    serveraddr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);

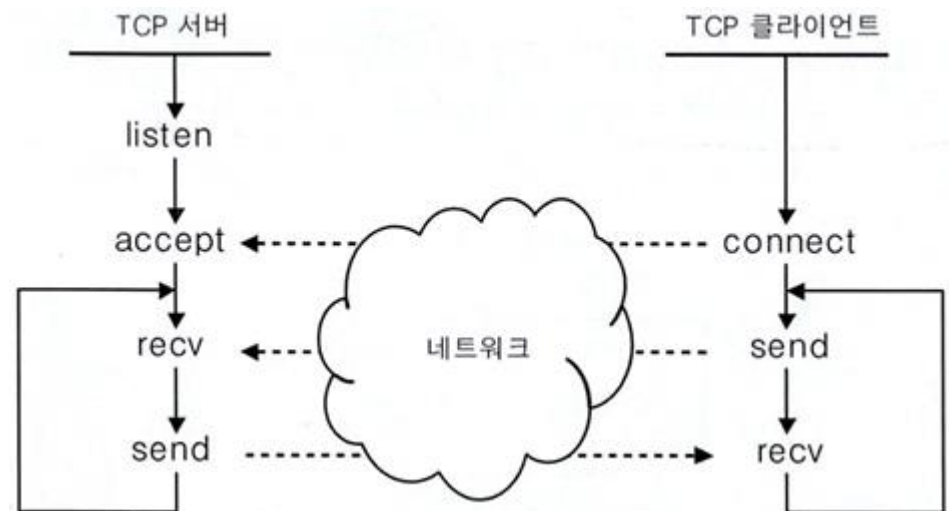
    // closesocket(listen_socket)
    ...

    // 원속 사용이 끝났다.
    ...
}
```

TCP 서버/클라이언트

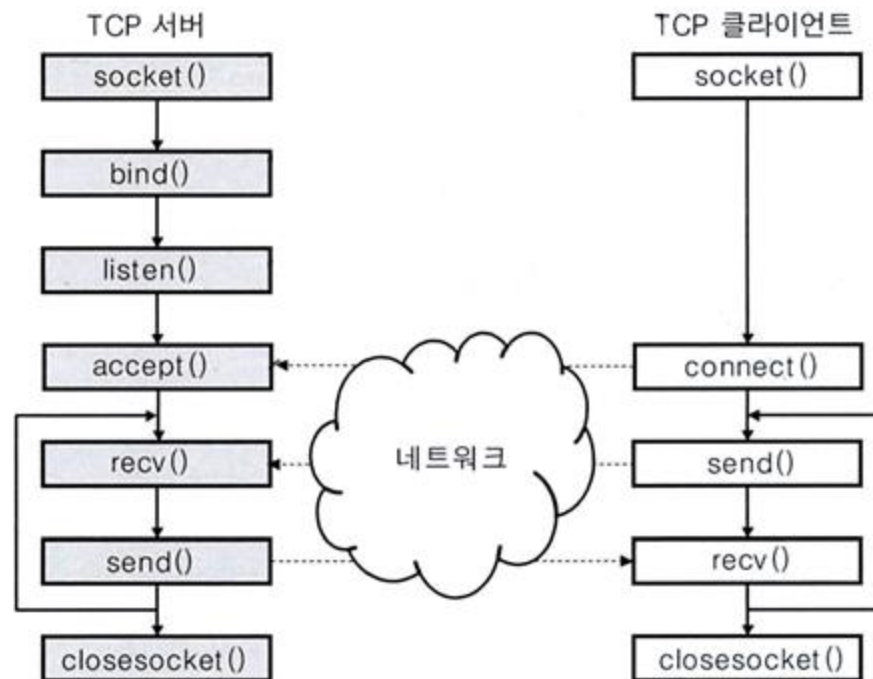
▶ TCP 서버/클라이언트의 동작 방식

- 서버는 먼저 실행되어 클라이언트 **접속을 기다린다(listen)**
- 클라이언트가 **서버에 접속(connect)**하여 데이터를 보낸다(send)
- 서버는 **클라이언트 접속을 수용(accept)**하고 클라이언트가 보낸 데이터를 받아(recv) 처리한다
- 서버는 처리한 데이터를 클라이언트에 보낸다(send)
- 클라이언트는 서버가 보낸 데이터를 받아(recv) 자신의 목적에 맞게 사용한다



TCP 서버/클라이언트

▶ 사용되는 함수와 흐름



TCP 서버 / 클라이언트

▶ 서버 함수

- `socket()` : 소켓을 생성
- `bind()` : 지역 IP 주소와 지역 포트 번호를 결정
- `listen()` : TCP 상태를 LISTENING으로 변경, 접속 받을 준비
- `accept()` : 자신에게 접속한 클라이언트와 통신할 수 있는 새로운 소켓 생성, 원격 IP 주소와 원격 포트 번호가 결정
- `send()`, `recv()` : 클라이언트와 통신을 수행
- `closesocket()` : 통신이 끝나면 사용이 끝난 소켓을 닫는다
- 새로운 클라이언트가 접속될 때마다 `accept()` ~ `send()`, `recv()`가 반복

지역 IP 주소, 지역 포트 번호 : 서버 또는 클라이언트 자신의 주소

원격 IP 주소, 원격 포트 번호 : 서버 또는 클라이언트가 통신하는 상대의 주소

TCP 서버

▶ TCPServer 새 콘솔 프로젝트 생성

```
int main()
{
    ...

    // bind() 소켓 설정.
    if (bind(listen_socket, (SOCKADDR*)&serveraddr, sizeof(serveraddr)) == SOCKET_ERROR)
    {
        closesocket(listen_socket);
        WSACleanup();
        err_quit("bind");
    }

    // listen() 수신 대기열 생성.
    if (listen(listen_socket, SOMAXCONN) == SOCKET_ERROR)
    {
        closesocket(listen_socket);
        WSACleanup();
        err_quit("listen");
    }

    ...
}
```

TCP 서버

```
#define MAX_BUFFER_SIZE 256
int main()
{
    ...
    // 데이터 통신에 사용할 변수.
    SOCKADDR_IN clientaddr;
    int addrlen = sizeof(SOCKADDR_IN);
    ZeroMemory(&clientaddr, addrlen);

    SOCKET client_socket;
    int retval;
    char buf[MAX_BUFFER_SIZE + 1];
    while (1)
    {
        // accept() 연결 대기.
        client_socket = accept(listen_socket, (SOCKADDR*)&clientaddr, &addrlen);
        if (INVALID_SOCKET == client_socket) continue;

        printf("\n[TCP 서버] 클라이언트 접속 : IP 주소=%s, 포트 번호=%d\n", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

        // send()
        // recv()

        // 클라이언트 소켓 종료.
        closesocket(client_socket);
        printf("\n[TCP 서버] 클라이언트 종료 : IP 주소=%s, 포트 번호=%d\n", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));
    }
    ...
}
```


TCP 서버

```
int main()
{
    ...
    while (1)
    {
        ...
        // 데이터 통신.
        while (1)
        {
            ZeroMemory(buf, sizeof(buf));
            // 데이터 받기.
            retval = recv(client_socket, buf, sizeof(buf), 0);
            if (SOCKET_ERROR == retval) break;
            else if (0 == retval) break;

            // 받은 데이터 출력.
            buf[retval - 1] = '\0';
            printf("Wn[TCP/%s:%d] %sWn ", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port), buf);

            // 데이터 보내기.
            retval = send(client_socket, buf, retval, 0);
            if (SOCKET_ERROR == retval) break;
        }
        ...
    }
    ...
}
```

TCP 서버 / 클라이언트

▶ 클라이언트 함수

- `socket()` : 소켓을 생성
- `connect()` : 서버에 접속
- `send()`, `recv()` : 서버와 통신을 수행
- `closesocket()` : 서버와의 통신이 끝나면 소켓을 닫는다

▶ `send()`, `recv()` : 송수신 버퍼를 이용하여 데이터를 주고 받는다

- 송신 버퍼(send buffer) : 데이터를 전송하기 전에 임시로 저장해두는 영역
- 수신 버퍼(receive buffer) : 받은 데이터를 애플리케이션이 처리하기 전까지 임시로 저장해두는 영역
- 소켓 버퍼(socket buffer) : 송신 버퍼, 수신 버퍼를 통틀어 이르는 용어

TCP 서버 / 클라이언트

▶ 블로킹(blocking) 소켓

- send() 함수를 호출할 때, 송신 버퍼의 여유 공간이 보낼 데이터의 크기보다 작을 경우 해당 프로세스를 대기 상태(wait state)로 만들고 송신 버퍼의 여유 공간이 생기면 깨어나 크기만큼 데이터 복사가 이루어 진다

▶ 논블로킹(nonblocking) 소켓

- ioctlsocket() 함수를 이용하여 블로킹 소켓을 논블로킹 소켓으로 변경할 수 있다
- send() 함수가 호출되면 송신 버퍼의 여유 공간 만큼 데이터 복사 후 실제 복사된 바이트 수를 리턴한다
- 프로그램이 복잡해 지며, CPU 사용량이 증가한다

TCP 클라이언트

▶ TCPClient 새 콘솔 프로젝트 생성

```
int main()
{
    // 윈속 초기화.
    ...

    // socket() 소켓 생성.
    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    if (INVALID_SOCKET == sock) return -1;

    // 작업 진행...

    closesocket(sock);

    // 윈속 사용이 끝났다.
    WSACleanup();
}
```

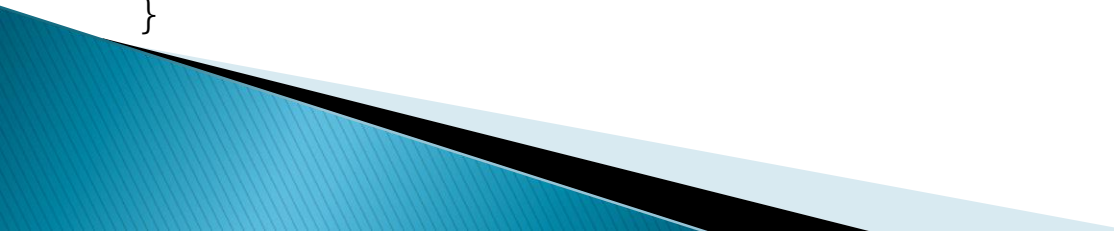
TCP 클라이언트

```
int main()
{
    ...

    // 서버 정보 객체 설정.
    SOCKADDR_IN serveraddr;
    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(9000);
    serveraddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");

    // connect()
    if (SOCKET_ERROR == connect(sock, (SOCKADDR*)&serveraddr,
    sizeof(serveraddr))) err_quit("connect");

    ...
}
```



TCP 클라이언트

```
#define MAX_BUFFER_SIZE 256
int main()
{
    ...
    int len, retval;
    char buf[MAX_BUFFER_SIZE + 1];
    while (1)
    {
        ZeroMemory(buf, sizeof(buf));
        printf("\n[보낼 데이터]");
        if (fgets(buf, MAX_BUFFER_SIZE, stdin) == NULL) break;

        len = strlen(buf);
        if (buf[len - 1] == '\n') buf[len - 1] = '\0';
        if (strlen(buf) == 0) break;

        // 데이터 보내기.
        retval = send(sock, buf, sizeof(buf), 0);
        if (SOCKET_ERROR == retval) break;

        printf("[TCP 클라이언트] %d바이트를 보냈습니다.\n", retval);
    }
}
```

TCP 클라이언트

```
// 데이터 받기.
```

```
ZeroMemory(buf, sizeof(buf));
```

```
retval = recv(sock, buf, sizeof(buf), 0);
```

```
if (SOCKET_ERROR == retval) break;
```

```
else if (0 == retval) break;
```

```
// 받은 데이터 출력.
```

```
buf[retval - 1] = '\0';
```

```
printf("[TCP 클라이언트] %d바이트를 받았습니다.\n", retval);
```

```
printf("[받은 데이터]%s\n", buf);
```

```
}
```

```
...
```

```
}
```



멀티스레드(multi thread)

▶ 스레드

- 실제 CPU 시간을 할당 받아 수행되는 실행단위

▶ 주 스레드(primary thread)

- `main()`, `WinMain()`에서 시작되는 스레드
- 프로세스가 실행될 때 생성

▶ 컨텍스트 전환(context switch)

- CPU와 운영체제의 협동으로 이루어지는 스레드 실행 상태의 저장과 복원 작업
- 각 스레드는 다른 스레드의 존재와 무관하게 자신의 상태를 유지하며 실행가능

▶ 스레드 함수(thread function)

- 스레드 실행 시작점이 되는 함수
- 예) `main()`

멀티스레드(multi thread)

▶ 스레드 생성

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    SIZE_T dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId);
```

lpThreadAttributes : 구조체 변수의 주소값, **NULL**을 사용

dwStackSize : 새로 생성할 스레드에 할당될 스택의 크기, **0** 사용 시 기본 **1MB**가 할당

lpStartAddress : 함수 스레드의 시작 주소, **DWORD WINAPI ThreadProc(LPVOID lpParameter) {}**

lpParameter : 스레드 함수에 전달될 인자, 전달할 인자가 없다면 **NULL**

dwCreationFlags : 스레드 생성을 제어하는 값, **0** 또는 **NULL**, **CREATE_SUSPENDED**(ResumThread() 함수가 호출되기 전까지 실행되지 않는다)

lpThreadId : **스레드의 ID**가 저장, 필요 없다면 **NULL**(Dos에서는 사용 하면 안됨)

```
HANDLE hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);  
if (NULL == hThread) std::cout << "스레드 생성 실패!!" << std::endl;  
CloseHandle(hThread);
```

멀티스레드(multi thread)

▶ 스레드 종료

- 정상적인 종료 방법

: 스레드 함수에서 리턴한다

: 스레드 함수 내에서 **ExitThread()** 함수를 호출

- 강제적인 종료 방법

: **TerminateThread()** 함수를 호출

: 주 스레드가 종료되면서 모든 스레드가 종료된다

멀티스레드 TCP 서버

- ▶ 소켓만을 지닌 경우 해당 소켓에 연관된 주소 정보를 알려주는 함수

`int getpeername(SOCKET s, struct sockaddr* name, int* namelen)`
- 소켓 데이터 구조체에 저장된 원격 IP 주소와 원격 포트 번호를 알려준다

`int getsockname(SOCKET s, struct sockaddr* name, int* namelen)`
- 소켓 데이터 구조체에 저장된 지역 IP 주소와 지역 포트 번호를 알려준다

```
SOCKADDR_IN clientaddr;  
int addrlen = sizeof(clientaddr);  
getpeername(client_socket, (SOCKADDR*)&clientaddr, &addrlen);
```

멀티스레드 TCP 서버

```
SOCKET client_socket;
HANDLE hThread;
DWORD threadID;
while (1)
{
    // accept() 연결 대기.
    client_socket = accept(listen_socket, (SOCKADDR*)&clientaddr, &addrlen);
    if (INVALID_SOCKET == client_socket) continue;

    printf("\n[TCP 서버] 클라이언트 접속 : IP 주소=%s, 포트 번호=%d\n",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));

    // 스레드 생성.
    hThread = CreateThread(NULL, 0, ProcessClient, (LPVOID)client_socket, 0, &threadID);
    if (NULL == hThread) std::cout << "[오류] 스레드 생성 실패!" << std::endl;
    else CloseHandle(hThread);
}
```

멀티스레드 TCP 서버

```
DWORD WINAPI ProcessClient(LPVOID arg)
{
    SOCKET client_sock = (SOCKET)arg;
    SOCKADDR_IN clientaddr;
    int addrlen = sizeof(clientaddr);
    getpeername(client_sock, (SOCKADDR*)&clientaddr, &addrlen); // 클라이언트 정보 얻기.
    int retval;
    while (1)
    {
        // send(), recv()
        ...
    }

    // closesocket()
    closesocket(client_sock);
    printf("Wn[TCP 서버] 클라이언트 종료 : IP 주소=%s, 포트 번호=%dWn",
        inet_ntoa(clientaddr.sin_addr),
        ntohs(clientaddr.sin_port));

    return 0;
}
```

스레드 동기화

- ▶ 멀티스레드 환경에서 발생하는 문제를 해결하기 위한 일련의 작업을 스레드 동기화(thread synchronization)라 부른다
- ▶ **필요한 경우**
 - 두 개 이상의 스레드가 **공유 리소스를 접근할 때**, 오직 한 스레드만 접근을 허용해야 하는 경우
 - 특정 사건 발생을 다른 스레드에 알리는 경우(한 스레드가 작업을 완료 후, 대기 중인 다른 스레드를 깨우는 경우)

스레드 동기화

▶ 동기화 객체(synchronization object)

- 두 스레드가 동시에 진행되서는 안되는 상황이 있을 때, 두 스레드 모두 매개체를 통해 진행 가능 여부를 판단하고, 이에 근거하여 진행을 계속하거나 대기 하도록 하는데, 이러한 **매개체 역할을 할 수 있는 것들**을 통틀어 동기화 객체라 부른다

종류	주요 용도
임계 영역(critical section)	공유 리소스에 대해 오직 하나의 스레드만 접근 허용 (한 프로세스에 속한 스레드에만 사용 가능)
뮤텍스(mutex)	공유 리소스에 대해 오직 하나의 스레드만 접근 허용 (서로 다른 프로세스에 속한 스레드에도 사용 가능)
이벤트(event)	특정 사건 발생을 다른 스레드에 알린다
세마포어(semaphore)	한정된 개수의 자원을 여러 스레드가 사용하려 할 때 접근 제어
대기 기능 타이머(waitable timer)	특정 시간이 되면 대기 중인 스레드를 깨운다

스레드 동기화

▶ 임계 영역

- 스레드 동기화를 위하여 사용하지만, 동기화 객체로 분류되지 않으며, 특징 또한 다르다
- 유저(user) 영역의 메모리에 존재하는 구조체 이기에 다른 프로세스가 접근 할 수 없어 한 프로세스에 속한 스레드 동기화에만 사용 가능
- **일반적인 동기화 객체보다 빠르고 효율적이다**

`CRITICAL_SECTION cs; // 전역 변수로 선언.`

`InitializeCriticalSection(&cs); // 임계영역 사용하기 전 초기화.`

`EnterCriticalSection(&cs); // 공용 리소스를 사용하기 전에 호출.`

`LeaveCriticalSection(&cs); // 공용 리소스 사용이 끝나면 호출.`

`DeleteCriticalSection(&cs); // 임계 영역 사용이 끝나면 호출.`

스레드 동기화

▶ 뮤텝스

- 동기화 객체이다
- 스레드에서 어떠한 데이터를 사용하고 있는 동안 다른 스레드는 이 데이터를 건드리지 못하게 한다
- 데이터를 먼저 사용하고 있는 스레드에서 사용이 끝날 때 까지 기다리게 한다
- 하나의 프로세스에서 여러 스레드를 사용할 때 사용한다

```
HANDLE g_hMutex; // 전역 변수로 선언.  
// 뮤텝스를 생성.  
// NULL:하위 프로세스에 상속 불가.  
// false:해당 뮤텝스의 권한을 호출한 스레드가 가지지 못하게 한다.  
// NULL:해당 뮤텝스의 이름. 이름 없이 작성.  
g_hMutex = CreateMutex(NULL, false, NULL);  
// 공용 리소스를 사용, 임의의 시간(INFINITE:작업이 끝날 때 까지)동안 대기.  
WaitForSingleObject(g_hMutex, INFINITE);  
ReleaseMutex(g_hMutex); // 공용 리소스 사용이 끝났다.  
CloseHandle(g_hMutex); // 뮤텝스 사용을 끝내고 제거.
```

스레드 동기화

```
HANDLE g_hMutex; // 전역 변수.
int main()
{
    g_hMutex = CreateMutex(NULL, false, NULL);
    if (NULL == g_hMutex) return -1; // 뮤텁스 생성 실패.
    if (GetLastError() == ERROR_ALREADY_EXISTS) // 이미 생성된 뮤텁스가 있습니다!!
    {
        CloseHandle(g_hMutex);
        return -1;
    }

    ...

    WaitForSingleObject(g_hMutex, INFINITE);
    // 다른 스레드에서도 사용하는 공용 리소스를 사용한다.
    ...
    ReleaseMutex(g_hMutex);

    ...

    CloseHandle(g_hMutex);
}
```

구조체 메모리 정렬

```
typedef struct
{
    char cData; // 4byte
    int iData;  // 4byte
    short sData; // 4byte
}TEST;
```

cData (1byte)	empty (3byte)	sData (2byte)	empty (2byte)	iData (4byte)
------------------	------------------	------------------	------------------	------------------

```
#pragma pack(1)
typedef struct
{
    char cData; // 1byte
    int iData;  // 4byte
    short sData; // 2byte
}TEST;
#pragma pack()
```

Packet을 이용한 데이터 주고 받기

// 데이터 받기.

```
char buf[sizeof(Packet)];
```

```
ZeroMemory(buf, sizeof(buf));
```

```
recv(client_sock, buf, sizeof(buf), 0);
```

```
Packet* recv_packet = (Packet*)buf;
```

// 데이터 보내기.

```
Packet send_packet;
```

// 발송할 데이터를 패킷에 넣는다...

```
send(sock, (char*)&send_packet, sizeof(Packet), 0);
```