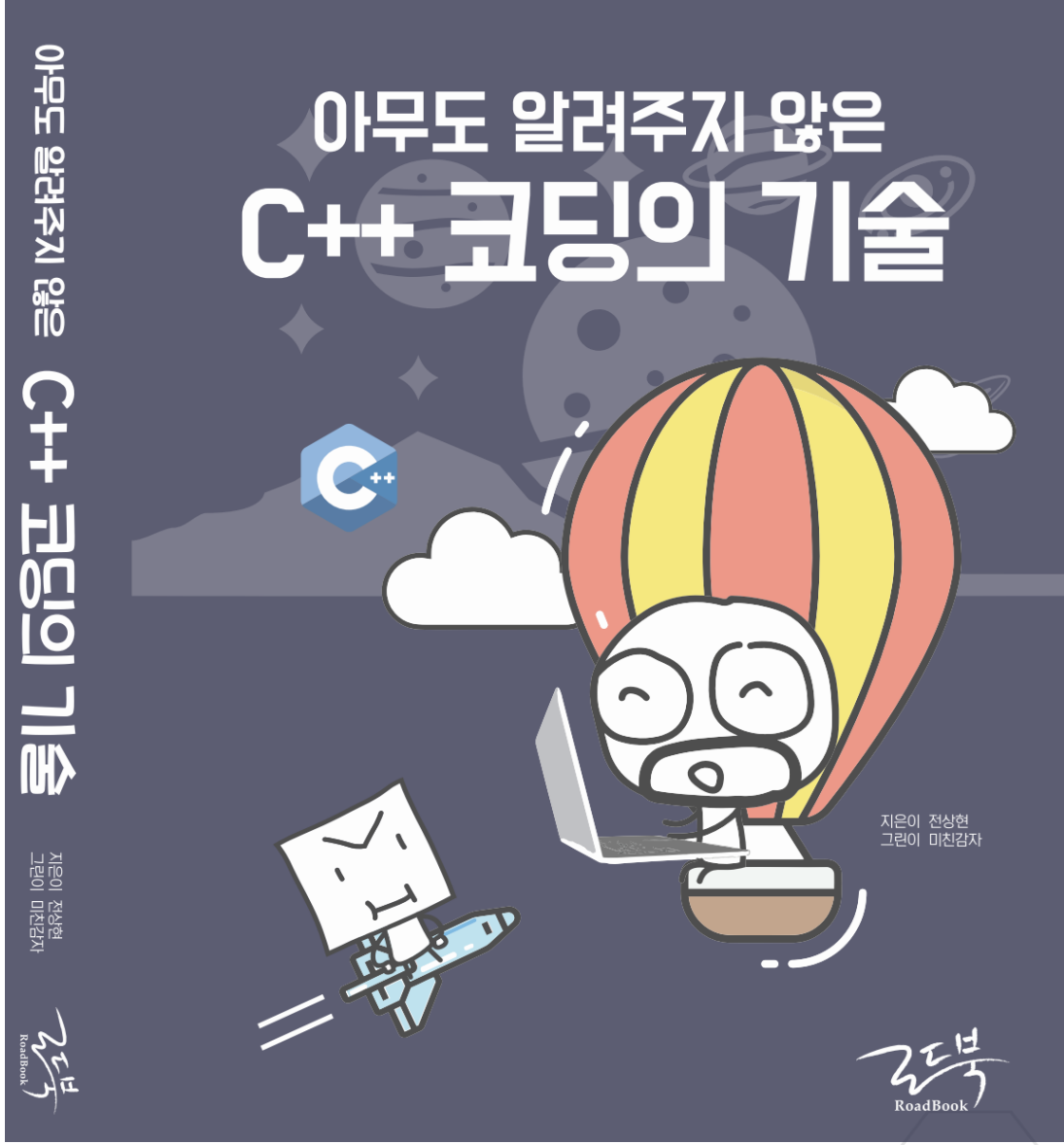
시스템 프로그래밍8

S-개발자 4기 2026-03-12(목)

서울 송파구 동남로 130, 2층 제 4강의실

악성코드검거단 MALWARE ARREST TEAM 대표이사 전상현

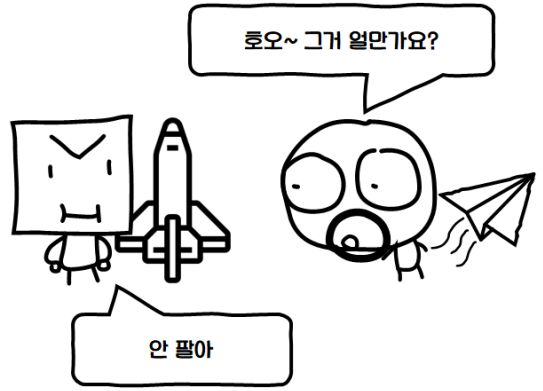


들어가기 전에 수업 개요

과정명	시스템 프로그래밍	강사명	전상현
강의시간	24시간 (4일 x 6시간)		
강의목표	컴퓨터 시스템을 이해한다. 나아가 다양한 시스템을 제어하는 C++ 프로그래밍 기술을 익힌다.		
평가방식	시험 50%, 실습 50%		

강의내용

시간(H)	제목	내용
1H	오리엔테이션	강사 소개 후 간단한 퀴즈와 함께 실행파일의 구조를 이해한다.
2H	변수형과 유니코드	시스템에 존재하는 모든 종류의 변수형을 알아본다. 다국어를 표현하는 문자체계인 유니코드를 이해한다.
3H	개발환경 구축하기	윈도우/리눅스(wsl)에 빌드할 수 있는 환경을 구성한다. 기본적인 개발툴 사용법을 익힌다.
2H	문자열 정복하기	C++과 친해지기 위한 코드를 작성해본다. VisualStudio 디버깅 기술을 배운다.
2H	C++ 컴파일러/링커/디버거 정복하기	유니코드 상호 변환하는 함수를 이해한다. 문자열을 자유자재로 다루는 방법을 배운다.
2H	파일시스템 정복하기	플랫폼별 파일/디렉토리 접근 함수를 이해한다. 파일 시스템을 자유자재로 다루는 방법을 배운다.
2H	데이터 정복하기	STL 자료구조를 이해하고, 실무 응용 기술을 배운다. XML/JSON/INI 등을 자유자재로 다루는 방법을 배운다.
2H	메모리 정복하기	C++의 꽃, 스택 메모리의 장단점을 알아본다. 4가지 영역의 메모리를 자유자재로 활용한다.
2H	소켓 정복하기	UDP/TCP 통신, DATAGRAM/STREAM의 차이를 이해한다. 소켓 통신 시스템을 자유자재로 다루는 방법을 배운다.
6H	**최종실습**	**앞에서 배운 지식과 기술을 활용하여 미니 프로젝트를 진행한다.**



코딩 실력이 곧 우주선이다.

우주같이 광활한 컴퓨터 세계로 여행을 떠나자.

<참고서적>

크로스플랫폼 핵심모듈 설계의 기술(2018) – 전상현, 로드북

아무도 알려주지 않은 C++ 코딩의 기술 (2023) – 전상현, 로드북

소켓 정복하기

소켓이란?

원격지의 컴퓨터와 데이터를 주고받는 것은 환상적인 일입니다. 요즘은 다양한 언어에서 이미 잘 만들어진 모듈로 누구나 손쉽게 구현할 수 있지만 소켓 수준에서 구현할 줄 아는 사람들은 드물지도 모릅니다.

소켓 프로그래밍은 **다양하게 발생할 수 있는 예외에 대비할 수 있어야 합니다**. 이미 만들어진 모듈을 쓰더라도 기본적인 원리를 알지 못하면 예상치 못한 문제로 시스템 장애를 유발할 수 있습니다.

소켓은 네트워크 패킷을 주고받을 수 있게 하는 커널 객체입니다. 모든 시스템에서 네트워크와 관련된 작업은 소켓을 통해서 이뤄집니다.

<윈도우>	<리눅스 or 맥OSX>
SOCKET WSAAPI socket(	int socket(
[in] int af,	int domain,
[in] int type,	int type,
[in] int protocol	int protocol
);	);

AddressFamily	Type	Protocol
AF_UNSPEC	SOCK_STREAM	IPPROTO_ICMP
AF_INET	SOCK_DGRAM	IPPROTO_IGMP
AF_IPX	SOCK_RAW	BTHPROTO_RFCOMM
AF_APPLETALK	SOCK_RDM	IPPROTO_TCP
AF_NETBIOS	SOCK_SEQPACKET	IPPROTO_UDP
AF_INET6		IPPROTO_ICMPV6
AF_IRDA		IPPROTO_RM
AF_BTH		

네트워크 시스템을 깊게 다루지 않는 이상 여러분들이 관심있는 소켓 타입은 아마 UDP와 TCP 정도일 것 입니다 두 경우의 소켓 생성 옵션은 다음과 같습니다.

소켓 형식	AddressFamily	Type	Protocol
TCP	AF_INET	SOCKET_STREAM	IPPROTO_TCP
UDP	AF_INET	SOCKET_DGRAM	IPPROTO_UDP

이렇게 생성된 소켓은 다음 API들을 이용해서 패킷을 송수신할 수 있게 됩니다.

(POSIX 함수라서 모든 플랫폼 공통)

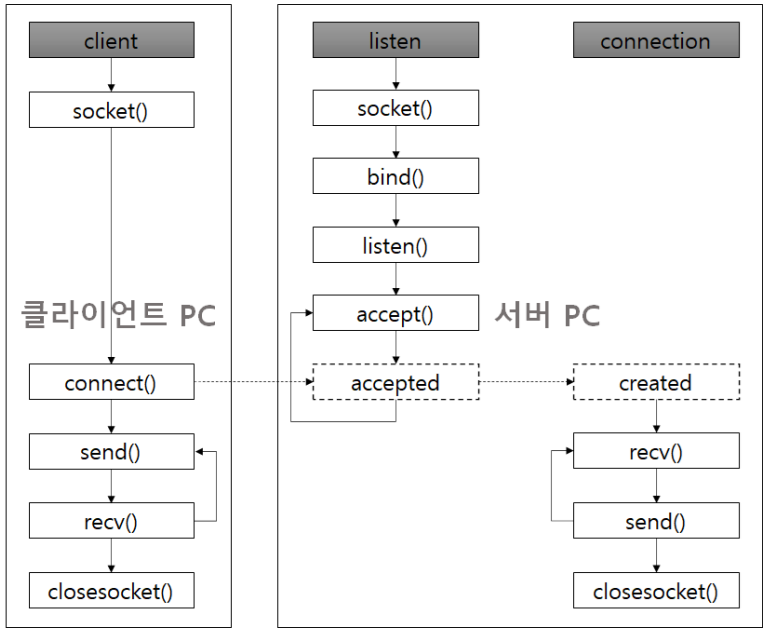
socket	connect	send	recv
bind	listen	accept	setsockopt
shutdown	closesocket	sendto	recvfrom

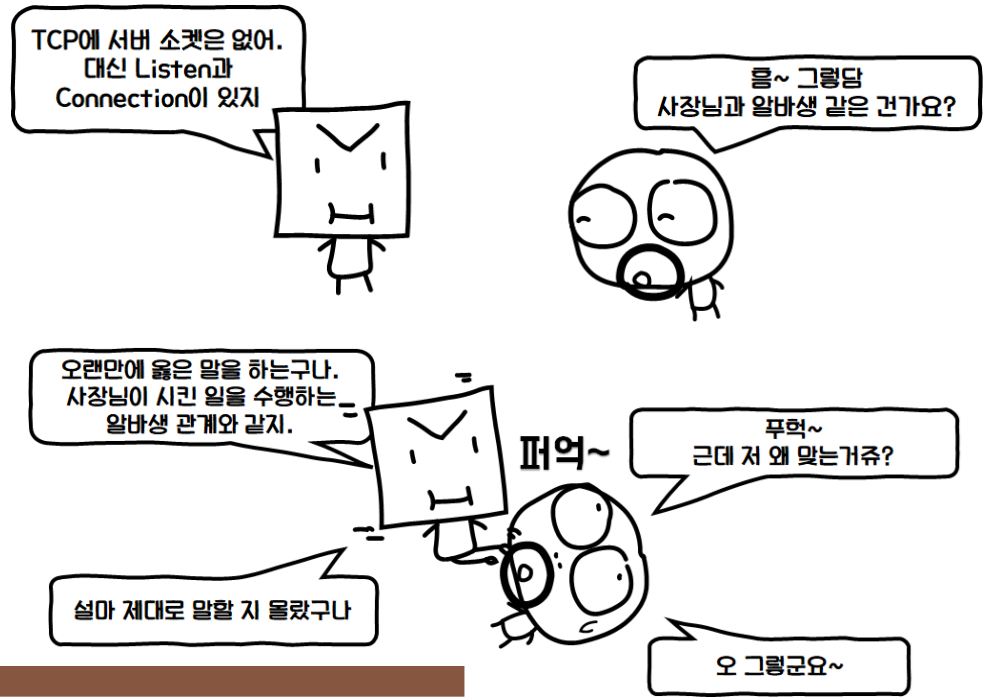
TCP 소켓은 개념적으로 세 가지로 분류합니다.

5508이라는 포트에 접속한다고 가정해봅시다. 하나의 클라이언트가 접속하면 5508 포트는 사용 중이 돼버려서 이후에 접속하는 클라이언트가 접속하지 못하게 될 겁니다. 하지만 우리가 익히 아는 것처럼 서버는 수백 수천 커넥션도 동시에 처리할 수 있죠. 분명히 하나의 포트는 하나의 소켓만 처리할 수 있을텐데? 이상함을 느끼셨어야 합니다.

TCP 소켓 이해하기

Client 소켓	Listen 소켓	Connection 소켓
원격지에서 서버로 접속할 때 사용하는 소켓	특정 포트로 원격 소켓의 접속을 기다리는 소켓	Listen 소켓으로부터 생성됨
서버측의 IP와 Port 정보를 이용해서 접근을 시도함	원격 소켓을 연결만 해주고 다시 다른 접속을 기다림	원격 소켓과 패킷 송수신을 수행하는 서버 소켓



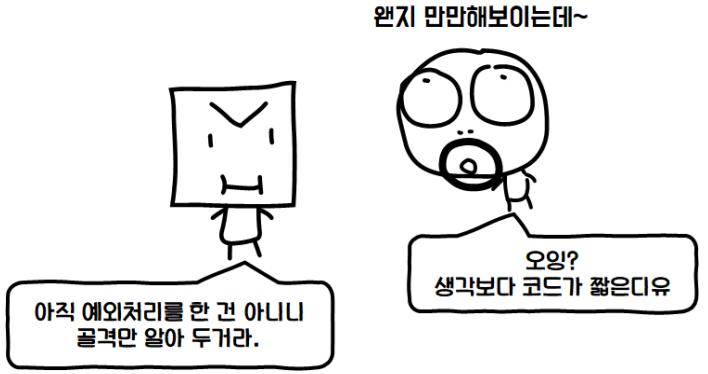


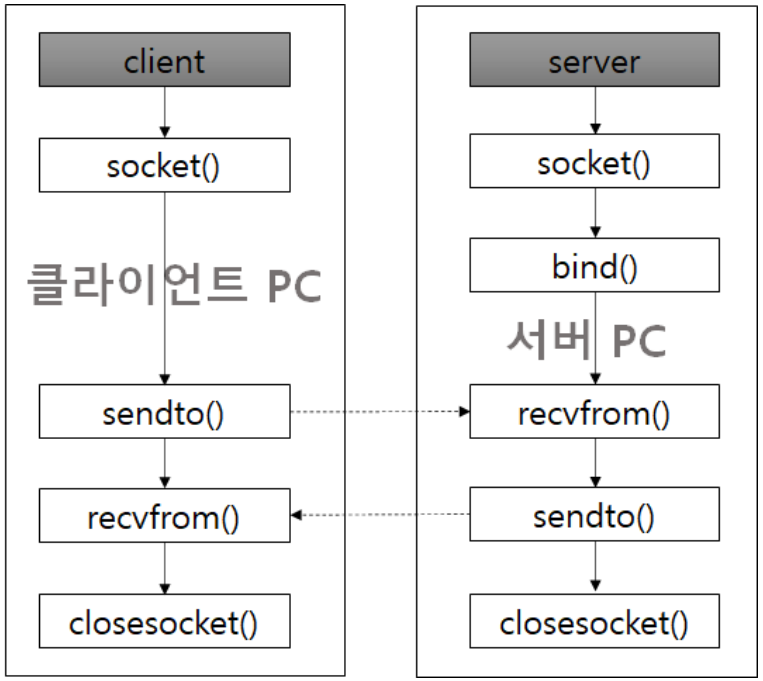
API 이름	Block여부	Listen 소켓에서의 역할
bind	None	어떤 포트를 사용할지 결정함
listen	None	TCP 연결을 위한3 way-handshake 수행 후 대기 최대 backlog 개수만큼 백그라운드에서 처리함
accept	Block	대기중인 접속자를 전담할 connection 소켓 생성 새로운 접속자가 나타날 때까지 함수가 block되고 반환하지 않음 Block을 해제하려면 listen 소켓을 close시켜야 함

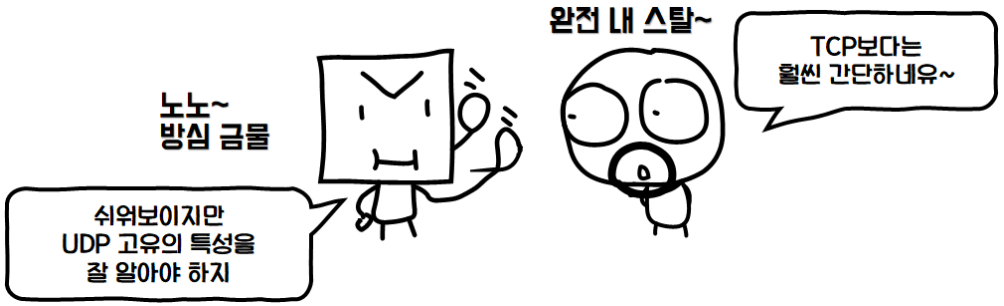
간단한 TCP 서버와 클라이언트 구현

```
const WORD g_wPort = 33481;
SOCKET g_listenSock;
int main(void)
{
    g_listenSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = INADDR_ANY;
    service.sin_port = htons(g_wPort);
    bind(g_listenSock, (sockaddr*)&service, sizeof(service));
    listen(g_listenSock, 10);

    sockaddr_in remoteInfo = { 0, };
    int nInfoLen = (int)sizeof(remoteInfo);
    SOCKET connectionSock;
    while(INVALID_SOCKET != (connectionSock = accept(g_listenSock,
                            (sockaddr*)&remoteInfo, &nInfoLen)))
    {
        char szBuffer[1500 + 1];
        int nReadSize = recv(connectionSock, szBuffer, 1500, 0);
        szBuffer[nReadSize] = 0;

        printf("server recved: %s₩n", szBuffer);

        send(connectionSock, szBuffer, nReadSize, 0);
        shutdown(connectionSock, SD_BOTH);
        closesocket(connectionSock);
    }
    return 0;
}
```

```
const WORD g_wPort = 33481;

int main(void)
{
    SOCKET clientSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    sockaddr_in remote;
    remote.sin_family = AF_INET;
    remote.sin_addr.S_un.S_un_b = { 127, 0, 0, 1 };
    remote.sin_port = htons(g_wPort);
    if (0 != connect(clientSock, (sockaddr*)&remote, sizeof(remote)))
        return -1;

    const char* pszMessage = "Hello world!";
    send(clientSock, pszMessage, strlen(pszMessage), 0);

    char szBuffer[1500 + 1];
    int nReadSize = recv(clientSock, szBuffer, 1500, 0);
    szBuffer[nReadSize] = 0;

    printf("client recved: %s₩n", szBuffer);
    closesocket(clientSock);
    return 0;
}
```







TCP에 비하면 UDP는 정말 쉽습니다. 일단 사용하는 API 수가 적거든요. 특징은 다음 두 가지에요.

① Client와 Server로 분류할 수 있다.

② 연결할 필요가 없으므로 사용되는 API 수가 적다.

하나의 서버에 둘 이상의 클라이언트가 데이터를 보내면 **각각의 클라이언트를 어떻게 구분할 수 있을까** 이해되지 않을 수 있어요. 하지만 다 방법이 있습니다. 그 비밀은 sendto와 recvfrom API에게 있습니다. TCP에서처럼 send 및 recv가 아닌 다른 API라는 것에 주목해야 합니다.

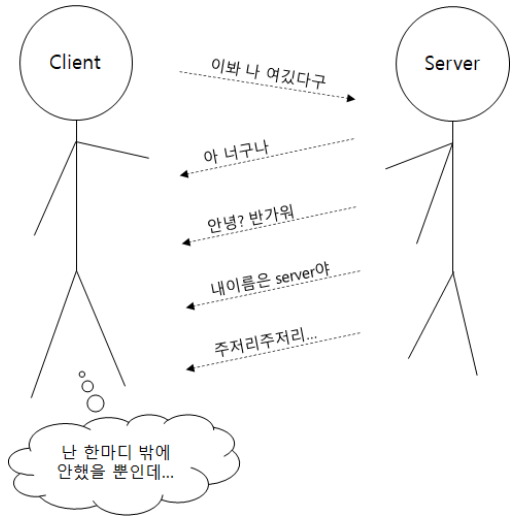
간단한 UDP 서버와 클라이언트 구현

```
int main(void)
{
    SOCKET connectionSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = INADDR_ANY;
    service.sin_port = htons(g_wPort);

    int nRet;
    nRet = bind(connectionSock, (sockaddr*)&service, sizeof(service));

    sockaddr_in remoteInfo = { 0, };
    int nInfoLen = (int)&remoteInfo;

    char szBuffer[1500 + 1];
    int nReadSize = recvfrom(connectionSock, szBuffer, 1500, 0, (sockaddr*)&remoteInfo, &nInfoLen);
    szBuffer[nReadSize] = 0;

    printf("server recved: %s₩n", szBuffer);

    sendto(connectionSock, szBuffer, nReadSize, 0, (sockaddr*)&remoteInfo, sizeof(remoteInfo));
    shutdown(connectionSock, SD_BOTH);
    closesocket(connectionSock);
    return 0;
}
```

```
int main(void)
{
    SOCKET clientSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    sockaddr_in remote;
    remote.sin_family = AF_INET;
    remote.sin_addr.S_un.S_un_b = { 127, 0, 0, 1 };
    remote.sin_port = htons(g_wPort);

    const char* pszMessage = "Hello world!";
    sendto(clientSock, pszMessage, strlen(pszMessage), 0, (sockaddr*)&remote,
(int)sizeof(remote));

    char szBuffer[1500 + 1];
    int nReadSize = recvfrom(clientSock, szBuffer, 1500, 0, nullptr, nullptr);
    szBuffer[nReadSize] = 0;

    printf("client recved: %s₩n", szBuffer);
    closesocket(clientSock);
    return 0;
}
```

TCP는 처음 접속할 때만 그 정보를 알려주고 접속된 이후에는 패킷만 주고받으면 돼서 편했는데 **UDP는 매번 받고 보낼 때마다 원격지 정보를 이용**해야 하는 겁니다.

다만 패킷을 먼저 받았던 곳으로 **되돌릴 때는 받은 그대로의 정보를 이용**하면 됩니다.



기본적으로 UDP는 P2P 통신으로 많이 쓰입니다. 문제는 **요즘의 피어(Peer)는 고정 IP가 아니라는 것이죠. 대부분의 가정에서는 공유기를 사용하고 있어 개인 사용자의 PC는 공유기 안의 사설 IP입니다**. 외부에서 그 IP를 알아도 접근할 수가 없다는 거죠.

하지만 우리가 좋아하는 **많은 게임들은 여러분들이 눈치채지 못하게 UDP로 P2P 통신을 하고 있습니다**. 심지어 우리가 그 게임을 **위해 포트 포워딩도 설정하지 않았는데** 말이죠. 그게 어떻게 가능할까요?

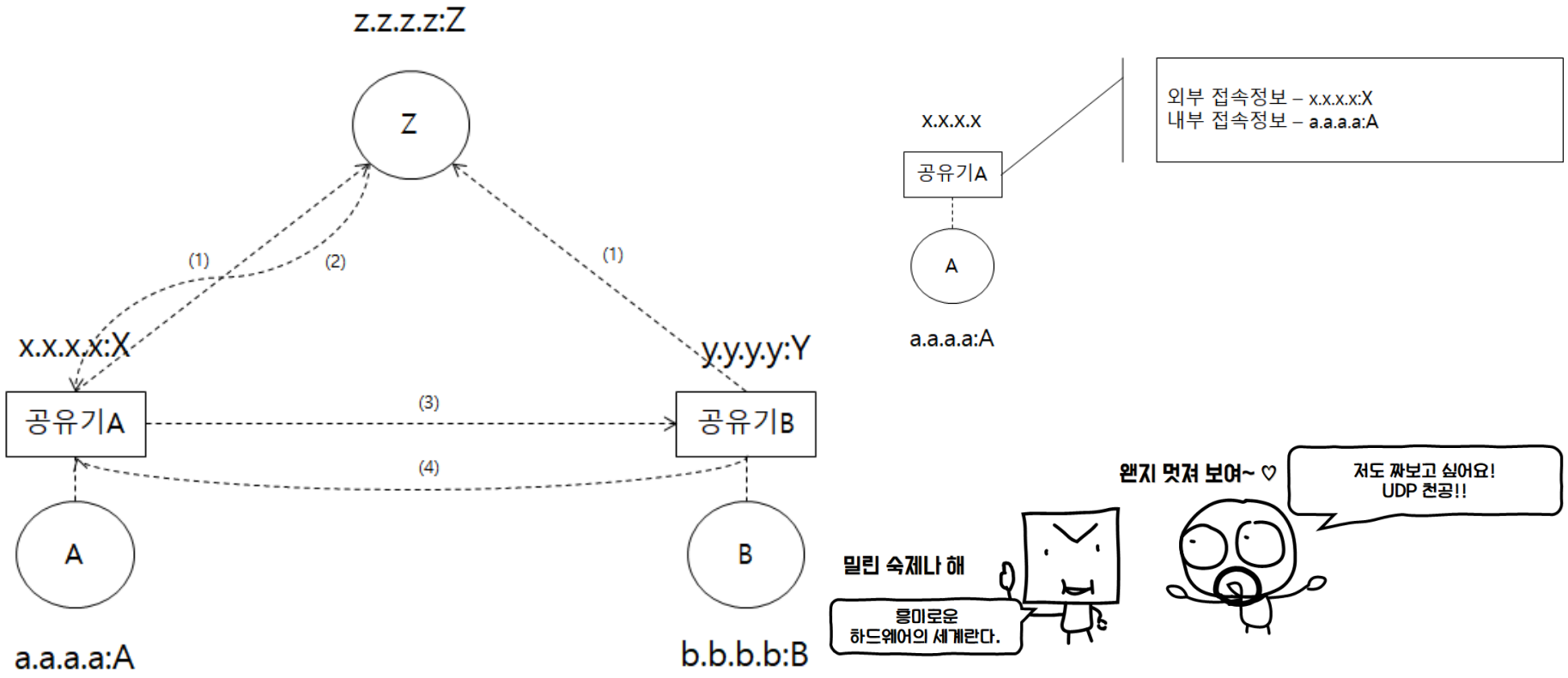
Z라는 중개자는 공용 IP를 갖고 있습니다. 서로 다른 영역의 사설 IP인 A와 B는 Z의 IP와 PORT를 이미 알고 있고요. 이제 다음과 같은 순서를 거치면 A와 B는 서로 UDP 통신을 할 수 있게 됩니다.

(1) A와 B는 각각 Z에게 임의의 UDP패킷을 전달(C는 A, B의 패킷을 받는 순간 둘의 접속 정보를 확보)

(2) Z가 A에게 B의 접속 정보를 전달함

(3) A는 B의 접속 정보를 이용해 임의의 UDP패킷을 전달(B도 이제 A의 접속 정보를 확보)

(4) B가 A에게 UDP 패킷을 전달함

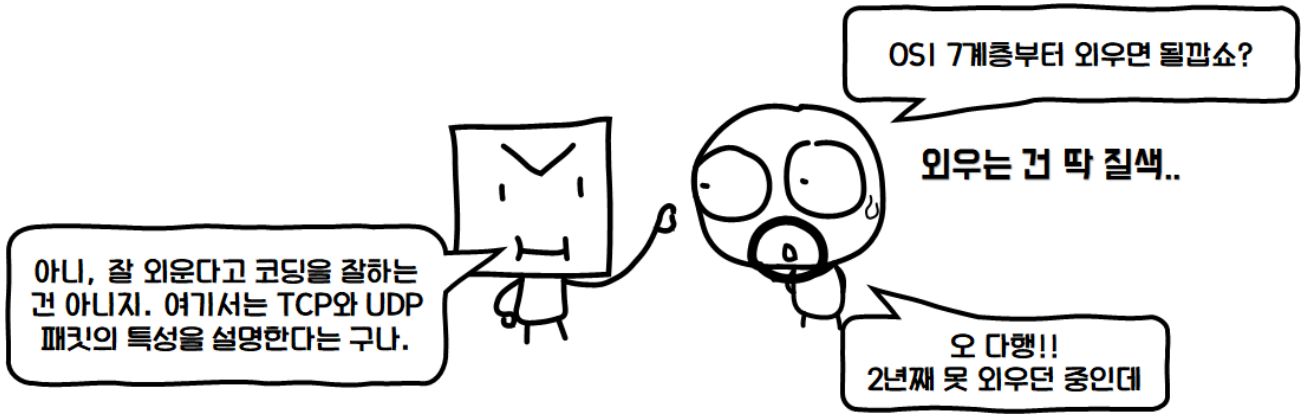


네트워크 정복하기

앞 장에서 소켓 프로그래밍을 해봤습니다만 사실 완전한 것은 아닙니다. 서비스 가능한 수준의 코드가 되려면 몇 가지 예외처리를 해야 합니다. 이번 장에서는 그와 관련된 **네트워크 기본 지식**을 익혀볼 예정입니다.

OSI 7계층으로 대표되는 네트워크 구성은 패킷 프로그래밍에 반드시 선행돼야 할 지식으로 여겨집니다. 하지만 알아야 할 것들이 워낙 많아서 정작 프로그래밍을 시작하기도 전에 지쳐버리는 상황을 겪고는 합니다. 코딩에 지속적인 재미를 느끼기 위해서는 코딩과 이론을 어느정도는 병행해야 하는데 그런 의미에서 예외 처리하지 않은 코드를 먼저 작성해보는 것이 도움이 됐으리라 생각합니다.

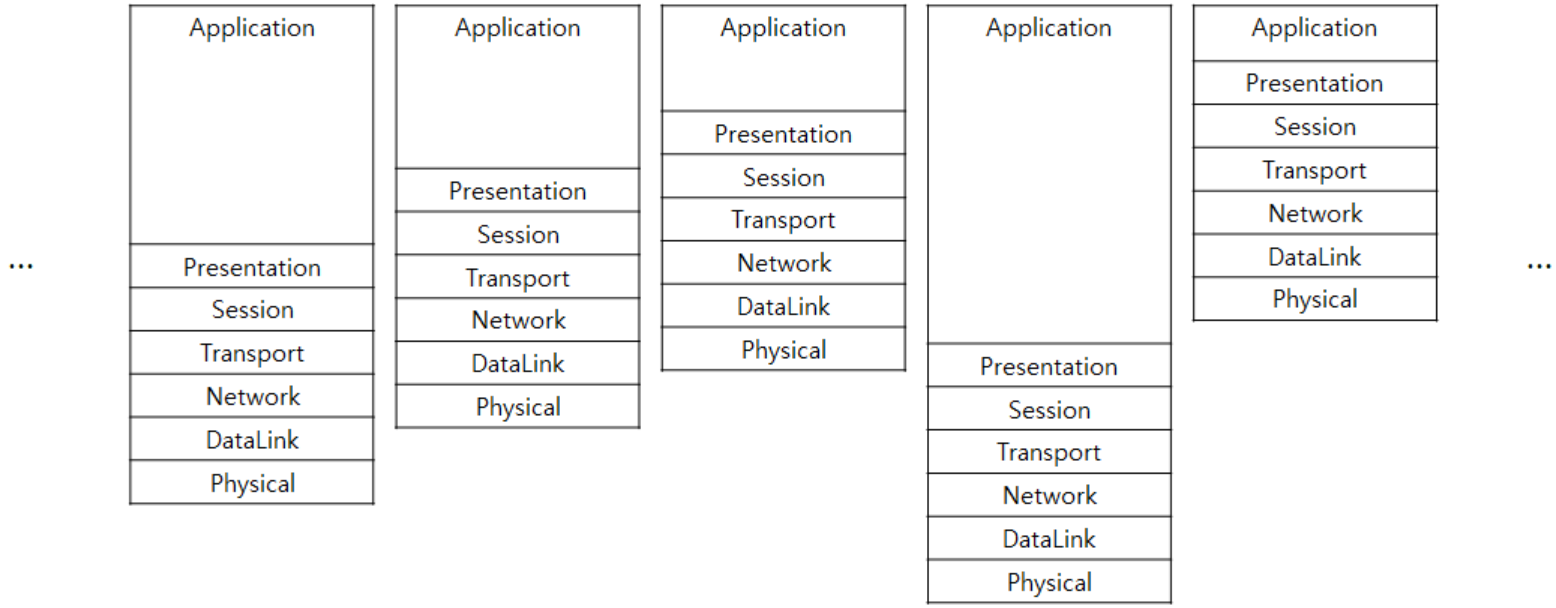
그럼 우리가 추가로 고려해야할 사항들은 무엇인지 차근차근 알아가 보도록 하겠습니다.



흔히들 네트워크 트래픽, 스트림이라는 표현을 많이 씁니다. 마치 **교통이나 물살처럼 흐른다**고 느껴지기 때문이죠. 그 흐름은 패킷들로 이루어졌습니다. **줄줄이 비엔나처럼 주욱 연결된 패킷들**은 각기 고유한 정보를 담고 있죠.

재밌는 것은 **모든 패킷들이 각각 OSI7계층 정보를 담고 있다는 겁니다**. 네트워크 이론을 공부하면 제일 먼저 배우는 이 개념을 과연 우리가 언제 쓸 일이 있을까 의문을 가졌을 텐데요, 생각보다 가까운 곳에 있습니다. 적어도 패킷 프로그래밍을 하는 우리들은 매번 그 일곱개의 계층을 하나씩 열어보는 거라고 볼 수 있습니다.

recv 함수로 수신한 데이터는 Application 영역에 해당합니다. 물론 소켓 타입에 다르지만 우리가 주로 다루는 TCP와 UDP만 두고 보면 요. 그 아래의 계층은 커널과 API에서 대신 해석해 줍니다. 참 편리하죠.



패킷은 원자단위 데이터

패킷의 크기는 다양합니다. 앞서 봤던 그림처럼 Application 영역의 크기가 영향을 주는 겁니다. 그렇다면 패킷의 크기가 무한정 커질 수 있는 것인가? 그렇지 않습니다. 여기에 **MTU**라는 개념이 있습니다.

- MTU: 최대 전송단위(Maximum Transmission Unit)라는 뜻으로 패킷의 최대 크기를 나타냄. 최대 1500바이트까지 지정할 수 있고, 보통 1500바이트를 사용함.

수많은 네트워크 장비들이 모두 알고 있어야 하는 것이 바로 MTU입니다. 그렇다면 패킷의 최대 크기가 왜 중요할까요? 그 이유는 다음과 같습니다.

- 패킷은 네트워크 데이터의 최소 단위이다.

즉, **일상 생활의 소포처럼 네트워크 세계에서도 데이터를 이루는 최소 단위**가 존재합니다. 만약 운송되는 소포를 배송업체가 임의로 자르거나 부셔서 이송하지 않는 것처럼 패킷도 반드시 온전히 전달될 수 있어야 합니다. 그러려면 최대 크기를 제한해야 하는 것이죠. 우리도 소포를 보낼 때 암묵적으로 너무 크면 안 된다는 알고 있으니까요.

반대로 말하면 다음과 같습니다.

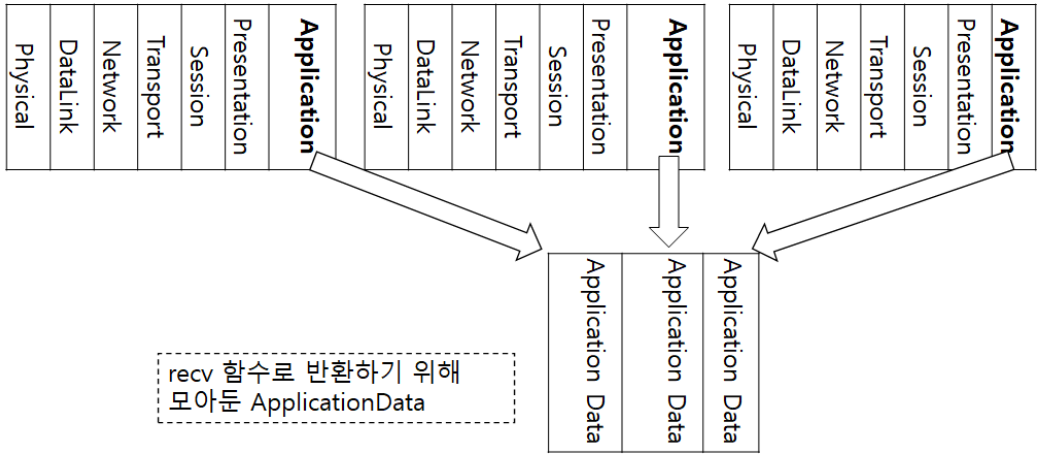
- 하나의 패킷에 담긴 데이터는 바뀌지 않는다.

우리가 알고 있는 상식 중 하나가 **패킷은 순서가 바뀌거나 유실될 수도 있다는 것입니다. 다만 그 최소 단위가 패킷이므로 하나의 패킷 안에서는 그런 일은 없습니다**. 전달 과정에 하드웨어나 전파 간섭으로 비트단위로 데이터가 손상되면 전송 계층에서 다시 수신하여 온전한 것만을 올려주게 되어있습니다.

다음과 같은 코드로 얻어오는 패킷은 꼭 요청한 크기만큼 반환하지 않을 수도 있습니다.

```
char buffer[3000];

int size = recv(sock, buffer, 3000, 0);
```

일단 네트워크 패킷이 도착하면 내부에서는 다음과 같이 Application Data를 따로 모아둡니다.



우리가 사용하는 recv함수는 이렇게 모아둔 데이터를 사용자 버퍼로 옮겨주는 것입니다. 이때 다음과 같은 네 가지 상황이 존재할 수 있습니다. 각각의 경우에 대해 이해하고 있어야 해요.

- 받아 둔 데이터가 없는 경우
- 받아 둔 데이터보다 요청하는 버퍼의 크기가 작은 경우
- 받아 둔 데이터보다 요청하는 버퍼의 크기가 큰 경우
- TCP 연결이 해제되거나 소켓이 닫힌 경우

- 받아 둔 데이터가 없는 경우

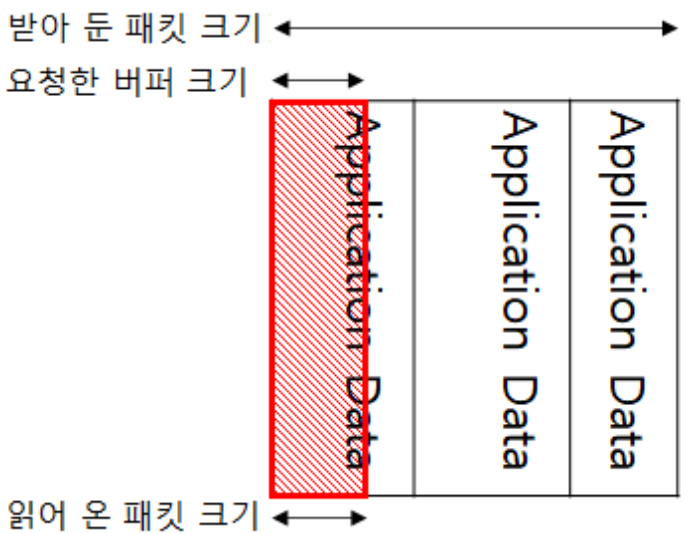
이 경우는 패킷 받을 때까지 recv 함수가 블로킹됩니다. 최대 대기 시간 즉, 타임아웃은 소켓 설정에 따라 다른데 아무 설정도 하지 않았다면 기본값이 적용됩니다. 경험상 5초~60초 사이였던 것 같아요.

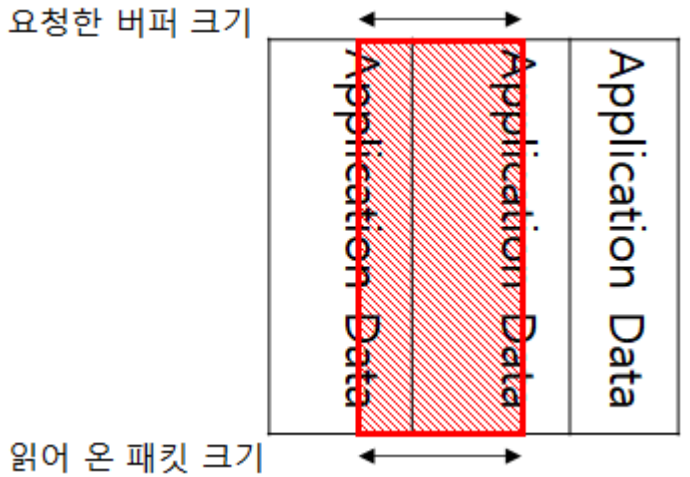
setsockopt 함수로 대기 시간을 변경할 수도 있습니다.

```
int value = 1000;

setsockopt(connectionSock, SOL_SOCKET, SO_RCVTIMEO, (const char*)&value, sizeof(nTimeOut));
```

- 받아 둔 데이터보다 요청하는 버퍼의 크기가 작은 경우

앞서 설명한대로 도착한 패킷은 시스템이 Application Data만 추려서 보관하고 있습니다. 그 크기보다 작은 버퍼로 recv 함수를 호출하면 읽어오는 크기는 다음과 같습니다.



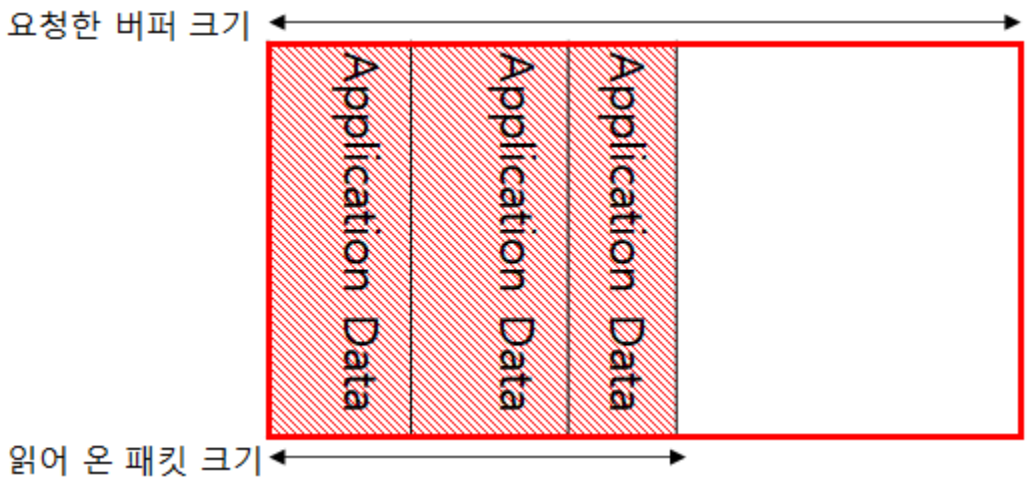


- 받아 둔 데이터보다 요청하는 버퍼의 크기가 큰 경우

이 때는 꽉 채워서가 아니라 되는대로 넘겨줍니다.

- 호출자: 가진 패킷 다 내놔
- 시스템: 이게 다 에요
- 호출자: 이게 뭐야, 더 없어?
- 시스템: 네 더 없어요, 패킷 생길 때까지 기다릴래요?
- 호출자: 일 없다. 나간다.

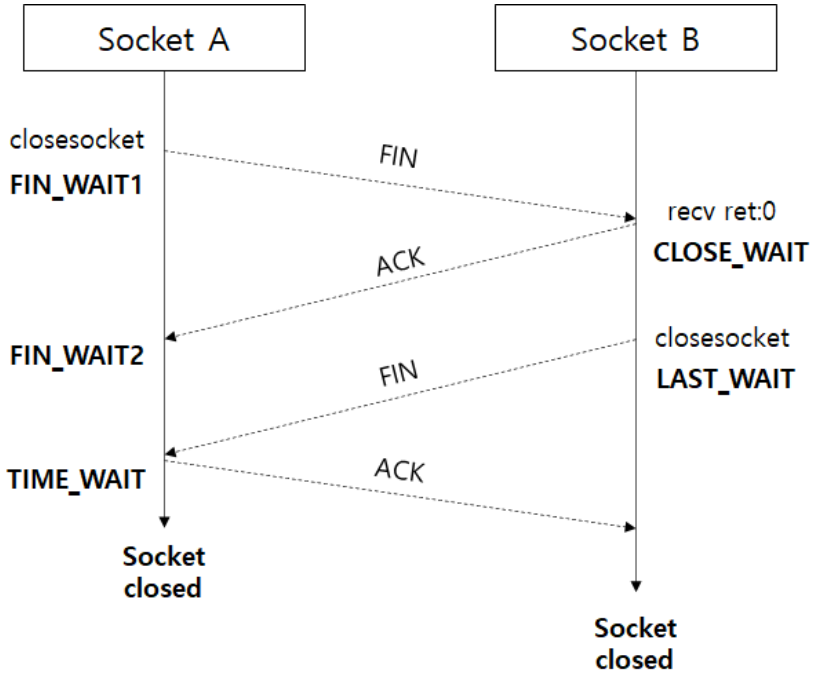
꽉 찰 때까지 기다리지 않습니다. 이것을 **패킷 찢어짐 현상**이라고 합니다. 요청한 만큼 다 읽어오지 못해서 여분이 남는 것이죠. 이 문제를 풀기 위한 방법을 잠시 후 알아보겠습니다.



- TCP 연결이 해제되거나 소켓이 닫힌 경우

만일 소켓에 에러가 발생하거나 더 이상 패킷을 받을 수 없는 상태이면 recv 함수는 -1을 반환합니다. 이 때는 더 이상 해당 소켓으로 작업을 지속해서는 안 됩니다. 소켓을 닫고 마무리합니다.

한편 0이 반환되는 경우도 있습니다. 0도 결과적으로 보면 에러처럼 보이지만 정확히 말하면 에러 상황은 아닙니다. 원격지 소켓이 먼저 닫힌 경우입니다. 이 때는 우리도 같이 닫아줘야 합니다. 마치 핸드폰 통화 도중 상대방이 먼저 끊으면 우리가 아무리 말해도 소용없는 것과 같습니다.



그런데 FIN 패킷을 받으면 왜 0이 반환될까?

UDP도 마찬가지로 수신된 패킷들은 시스템에서 미리 받아 모아둡니다. 하지만 그 처리 방식이 TCP와는 다릅니다. 마찬가지로 네 가지 경우로 구분해보겠습니다.

- 받아 둔 데이터가 없는 경우
- 단일 패킷보다 요청하는 버퍼의 크기가 작은 경우
- 요청하는 버퍼의 크기가 여러 패킷을 담을 만큼 큰 경우
- 소켓이 닫혀있거나 오류가 발생한 경우

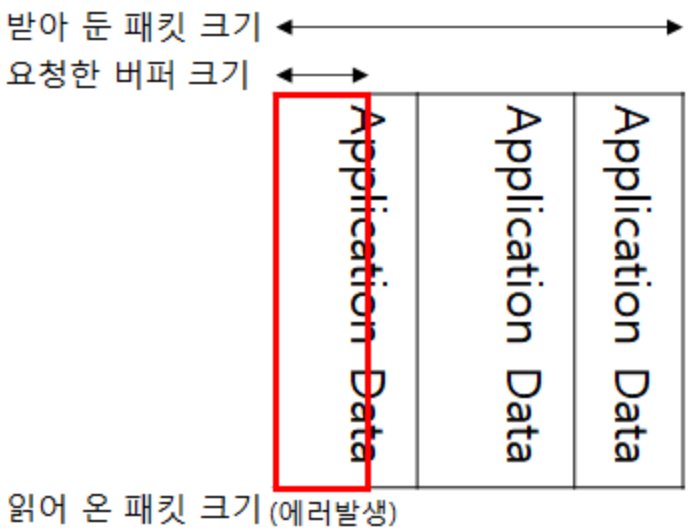
- 받아 둔 데이터가 없는 경우

TCP의 recv와 동일하게 동작합니다. 새로운 패킷을 받을 때까지 대기하게 됩니다. 역시 대기 시간을 조정하는 방법도 동일합니다.

- 단일 패킷보다 요청하는 버퍼의 크기가 작은 경우

기본적으로 UDP는 한 번에 하나의 ApplicationData만 얻어올 수 있습니다. TCP처럼 STREAM이 아니라 DGRAM으로 동작하기 때문에 패킷이 찢어지는 것을 애초에 허용하지 않습니다.

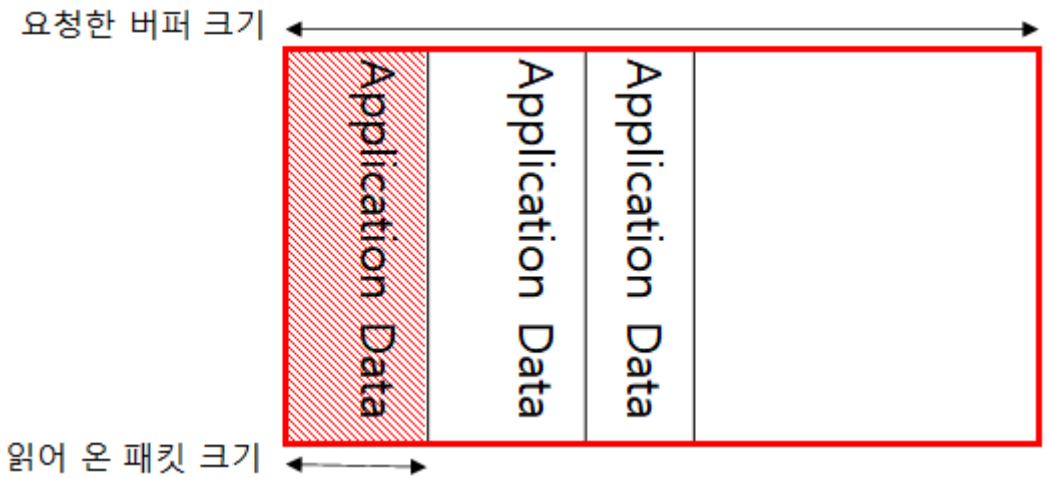
그런데 문제는 미리 받아 둔 단일 패킷보다 작은 크기의 버퍼로 요청할 때입니다.



TCP에서는 패킷이 찢어지더라도 받아놓은 만큼 버퍼에 최대한 담아 읽었던 반면, UDP에서는 아예 실패 처리됩니다. **반드시 완전체만 되돌려주겠다는 의미가 바로 DGRAM이라고 생각하면 됩니다**.

- 요청하는 버퍼의 크기가 여러 패킷을 담을 만큼 큰 경우

그럼 여러 패킷을 받을 수 있을 크기로 요청하면 어떻게 될까? 결과는 다음과 같습니다.



마찬가지로 TCP에서는 있는 대로 다 읽어 들였다면 UDP는 정직하게 하나만 돌려줍니다. 두 번째, 세 번째 패킷을 얻어오는 방법은 여러 번 부르는 방법 밖에 없습니다. 따라서 recvfrom 함수에 쓰일 버퍼의 크기는 1500바이트면 충분합니다. 단일 패킷보다 작으면 에러가 발생하고, MTU크기는 넘어설 수 없기 때문이죠.

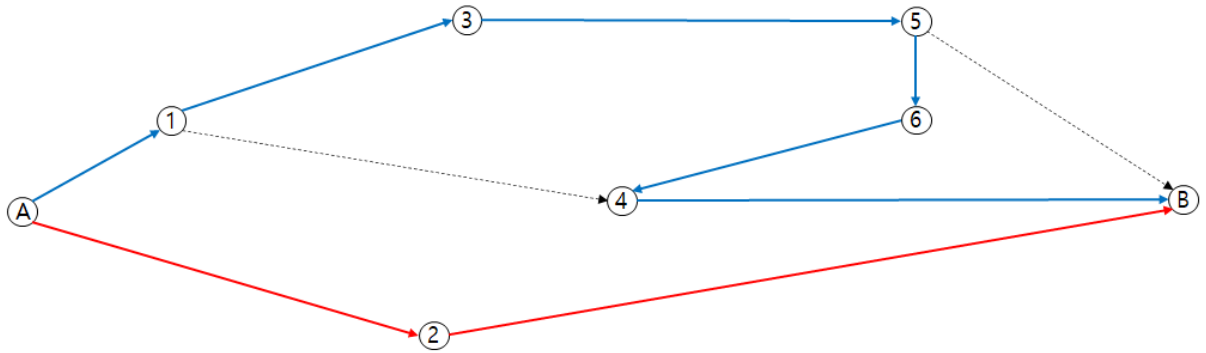
- 소켓이 닫혀있거나 오류가 발생한 경우

이것도 TCP와 동일합니다. -1을 반환하며 에러가 발생하므로 소켓을 닫고 루프를 탈출시켜야 합니다.

A와 B 사이에 1부터 6까지의 라우터가 있다면 도달할 수 있는 방법이 다양합니다. 대표적으로 다음 두 가지가 있겠네요.

1) 가장 느린 경로: A -> 1 -> 3 -> 5 -> 6 -> 4 -> B

2) 가장 빠른 경로: A -> 2 -> B

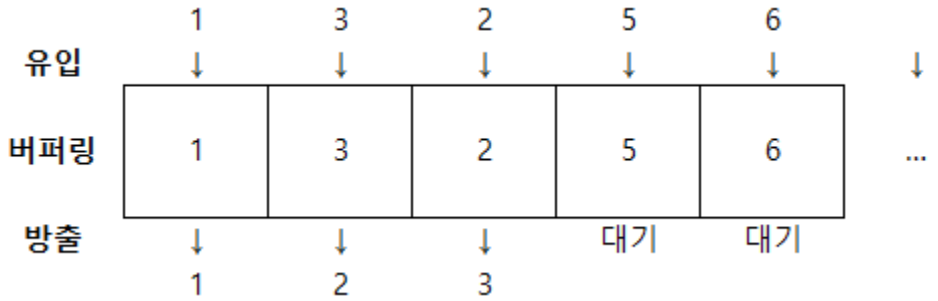


처음엔 가장 빠른 경로를 찾기 위해 여러 경로로 시도해 볼 겁니다. 그러다 어떤 패킷은 1)의 경로로, 어떤 패킷은 2)의 경로로 전달될 수도 있겠죠. 이런 경우에 뒤늦게 출발한 패킷이 최단 경로로 오면서 먼저 도착하는 상황이 발생합니다.

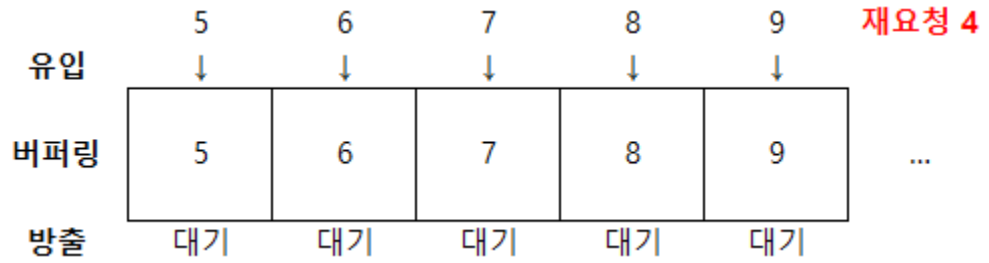
패킷의 순서를 교정하기 위한 TCP 정책

그럼 이렇게 순서가 바뀌거나 유실되는 패킷은 TCP에서 어떻게 처리해야 할까요? 다행히 우리가 직접 구현할 필요가 없습니다. 이미 다 해주니까요. 대신 원리는 알아둡시다.

TCP 계층은 수신되는 패킷이 순서대로 들어오면 그대로 내보내고, 순서에 어긋나면 맞는 패킷이 올 때까지 잠시 대기합니다.

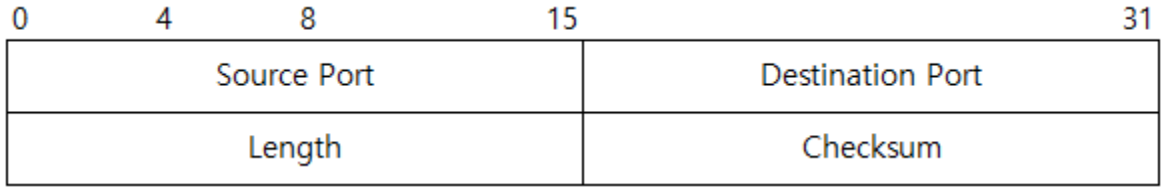


위의 그림을 보면 1 다음에 3이 왔지만 2가 올 때까지 기다렸다가 2를 먼저 내보내고 이후에 3을 내보내는 상황입니다. 그 다음 도착한 5와 6은 4번이 오지 않았으므로 대기하는 중이고요.



패킷의 순서를 교정하기 위한 UDP 정책

UDP는 TCP처럼 중간에 버퍼링하거나 순서를 보장해주는 과정이 존재하지 않습니다. 구조도 단순해요.



흔히들 UDP 패킷은 신뢰할 수 없고 속도가 빠르다고 합니다. 하지만 왜 신뢰하면 안 되고 속도는 얼마나 빠른지 정확히 알아야 할 필요가 있습니다.

먼저 신뢰할 수 없다는 것은 아래와 같이 다시 정의하는 것이 옳습니다.

1) MTU 미만의 UDP 패킷은 신뢰할 만하다. 받은 그대로 쓰면 된다.

2) MTU보다 크기가 큰 데이터는 UDP로 쓰면 안 된다. 패킷이 분할되는 과정에 순서가 바뀔 수 있다.

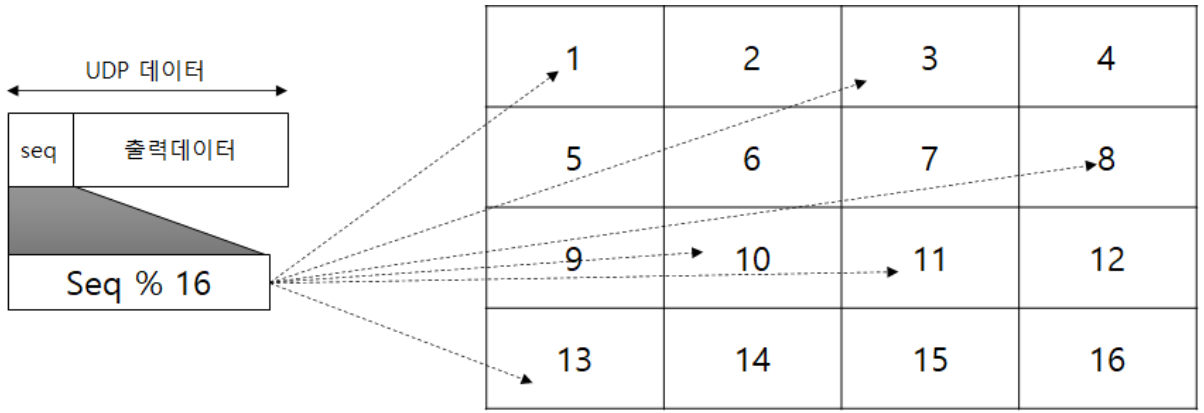
UDP 패킷은 순서가 뒤바뀌면 되돌릴 방법이 없습니다. Application 데이터에 Sequence Number를 적어 복구하는 방법도 있지만 그럴 바엔 TCP를 쓰는 것이 낫습니다.

정리하자면 UDP 패킷은 다음과 같은 상황에 적합합니다.

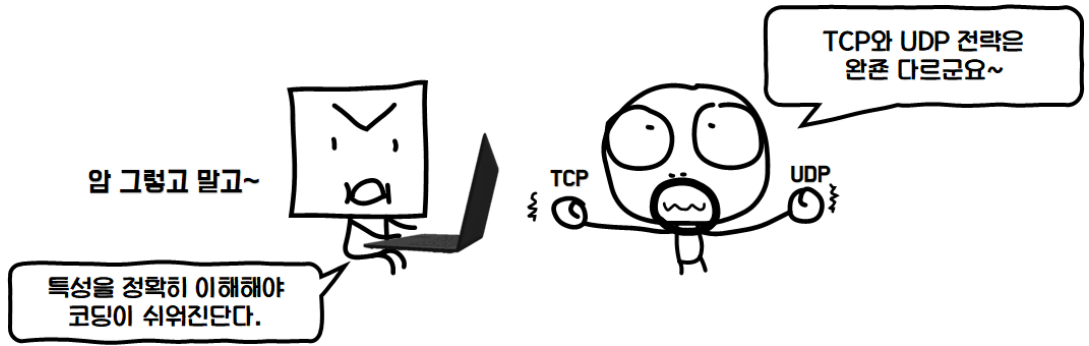
- MTU보다 작은 크기의 Application데이터만 사용할 때
- 패킷을 보낸 순서가 크게 중요하지 않을 때
- 네트워크 상황이 좋지 않을 때는 데이터를 적당히 버려도 상관없을 때

종합해보면 UDP는 화면에 출력하는 데이터를 전달할 때 유리합니다. 순서가 뒤바뀌거나 유실되어 일시적으로 보이지 않더라도 그걸 보는 사용자는 네트워크 상황이 좋지 않겠거니 하며 감안할 수 있기 때문이죠.

만일 출력 데이터를 다음과 같이 꾸밀 수만 있다면 패킷 한 두개쯤 유실되거나 순서가 어긋나도 크게 문제되지 않을 것입니다.



대략 16개로 분할한다면 16개 안에서 순서가 어긋나는 것은 영향이 없고, 유실된다 해도 일시적으로 이전에 그려졌던 그림이 남아있는 상태가 될테니까요.



Cppcore의 소켓함수

동기방식으로 동작하는 TCP 소켓 클래스

```
class CSyncTCPSocket
{
friend class CSyncServer;
friend class CSyncConnection;

protected:
SOCKET m_hSocket;

public:
CSyncTCPSocket(void);
virtual ~CSyncTCPSocket();

virtual ECODE Assign(SOCKET hAcceptedSocket);
virtual ECODE Connect(const char* pszIP, WORD wPort, DWORD dwTimeOut);
virtual void Close(void);

virtual ECODE Send(const void* pBuff, size_t tBufSize, DWORD dwTimeOut);
virtual ECODE Recv(void* pBuff, size_t tBufSize, DWORD dwTimeOut);
virtual ECODE Recv(void* pBuff, size_t tBufSize, size_t* ptRead, DWORD dwTimeOut);
virtual ECODE Peek(void* pBuff, size_t tBufSize, DWORD dwTimeOut, size_t* ptRead = NULL);

protected:
virtual ECODE SendWorker(SOCKET hSocket, const void* pBuff, size_t tBufSize, DWORD dwTimeOut, size_t* ptSent);
virtual ECODE RecvWorker(SOCKET hSocket, void* pBuff, size_t tBufSize, DWORD dwTimeOut, size_t* ptRead);
virtual ECODE PeekWorker(SOCKET hSocket, void* pBuff, size_t tBufSize, DWORD dwTimeOut, size_t* ptRead);
};
```

동기방식으로 동작하는 UDP 소켓 클래스

```
class CSyncUDPSocket
{
SOCKET m_hSocket;

public:
CSyncUDPSocket(void);
virtual ~CSyncUDPSocket();

virtual ECODE Listen(WORD wPort);
virtual ECODE Create(void);
virtual void Destroy(void);

virtual ECODE Broadcast(DWORD dwIP, WORD wPort, const void* pData, size_t
tDataSize);
virtual ECODE SendTo(const ST_SOURCE_INFO& pSourceInfo, const void* pBuff,
size_t tBufSize, DWORD dwTimeOut, size_t* ptSent = NULL);
virtual ECODE RecvFrom(void* pBuff, size_t tBufSize, DWORD dwTimeOut, size_t*
ptRead = NULL, ST_SOURCE_INFO* pSourceInfo = NULL);
};
```

Cppcore 클래스 서버 개발을 위한 동기 서버 클래스 (TCP 전용이다 당연히..)

```
class CSyncServer
{
protected:
...
public:
CSyncServer();
virtual ~CSyncServer();

virtual ECODE StartUp(const ST_SYNCSERVER_INIT& stInit);
virtual void ShutDown(void);

typedef bool (*FP_BROADCAST_CALLBACK)(CSyncConnection* pConnection, void* pContext);// check send packet or not
voidBroadcast(std::string strContext, FP_BROADCAST_CALLBACK fpCallback = NULL, void* pContext = NULL);
voidBroadcast(LPCVOID pData, size_t tSize, FP_BROADCAST_CALLBACK fpCallback = NULL, void* pContext = NULL);

size_tMaxConnectionCount(void) { return m_stInit.Connections.size(); }
size_tConnectionCount(void) { return m_setConnected.size(); }

private:
intListenThread(void * pContext);
intDisconnectThread(void * pContext);

friend intConnectionThreadCaller(void* pContext);
voidConnectionThread(CSyncConnection* pConnection);
};
```

```
enum E_DISCONNECT_TYPE
{
DISCONNECT_TYPE_UNDEFINED= 0,
DISCONNECT_TYPE_NO_RESPONSE_ON_CONNECT,// pending count=0,
DISCONNECT_TYPE_CONNECTED_AND_NO_RESPONSE,// pending count=100, no accept
DISCONNECT_TYPE_CONNECTED_AND_CLOSE,// pending count=100, accept and close
};

struct ST_SYNCSERVER_INIT
{
WORDwPort;
WORDwReserved;
DWORDdwRecvTimeOut;
E_DISCONNECT_TYPE nDisconnectType;
std::vector<CSyncConnection*>Connections;

ST_SYNCSERVER_INIT(void)
: wPort(0), wReserved(0), dwRecvTimeOut(10000)
, nDisconnectType(DISCONNECT_TYPE_NO_RESPONSE_ON_CONNECT)
, Connections()
{}
};
```

서버 개발을 위한 동기 Connection 소켓 클래스(상속받아 사용)

```
class CSyncConnection
{
friend class CSyncServer;

protected:
CSyncTCPSocket*m_pSocket;
std::stringm_strClientIP;

public:
CSyncConnection(CSyncTCPSocket* pSocket);
virtual~CSyncConnection();

virtual ECODESetAcceptedSocket(SOCKET hNewSocket){return
m_pSocket->Assign(hNewSocket);}
virtual CSyncTCPSocket* Raw(void){return m_pSocket;}

virtual voidOnConnect(void) = 0;
virtual voidOnClose(void) = 0;
virtual voidOnRecv(void) = 0;
};
```

서버 코드의 사용 예시

```
TEST(ServerTest, UBJsonProtocolTest)
{
    ST_SYNCSERVER_INIT stInit;
    stInit.wPort = 61503;
    stInit.Connections.push_back(new CUBJProtocolConenction());
    stInit.Connections.push_back(new CUBJProtocolConenction());
    stInit.Connections.push_back(new CUBJProtocolConenction());

    CSyncServer server;
    ASSERT_EQ(EC_SUCCESS, server.StartUp(stInit));
    Sleep(100);
    {
        ST_TEST_PACKET original;
        original.a = 1;
        original.b = 2;
        original.strContext = TEXT("hello?");
        original.vecBinary.resize(100, 5);

        CSyncTCPSocket client;
        CUBJsonProtocol protocol(&client);
        EXPECT_EQ(EC_SUCCESS, protocol.Connect("127.0.0.1", stInit.wPort, 1000));
        EXPECT_EQ(EC_SUCCESS, protocol.SendPacket(&original));

        ST_TEST_PACKET restored;
        EXPECT_EQ(EC_SUCCESS, protocol.RecvPacket(&restored));

        EXPECT_EQ(original.a, restored.a);
        EXPECT_EQ(original.b, restored.b);
        EXPECT_EQ(original.strContext, restored.strContext);
        ASSERT_EQ(original.vecBinary.size(), restored.vecBinary.size());
        EXPECT_EQ(0, memcmp(original.vecBinary.data(), restored.vecBinary.data(),
original.vecBinary.size()));

        protocol.Close();
    }
    server.ShutDown();
}
```

```
class CUBJProtocolConenction : public CSyncConnection
{
    CSyncTCPSocket m_Socket;
    CUBJsonProtocol m_Protocol;

public:
    CUBJProtocolConenction(void)
    : CSyncConnection(&m_Socket)
    , m_Socket()
    , m_Protocol(&m_Socket)
    {}
    ~CUBJProtocolConenction(void) {}

    void OnConnect(void)
    {
    }

    void OnClose(void)
    {
    }

    void OnRecv(void)
    {
        ST_TEST_PACKET packet;
        EXPECT_EQ(EC_SUCCESS, m_Protocol.RecvPacket(&packet));
        EXPECT_EQ(EC_SUCCESS, m_Protocol.SendPacket(&packet));
    }
};
```

고생 많으셨습니다!