

Chapter 05

TCP 기반 서버 / 클라이언트 2

에코 클라이언트의 문제점 확인하기

■ 에코 서버의 코드

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

- 서버는 데이터의 경계를 구분하지 않고, 수신한 데이터를 그대로 전송 ✓
o ➤ write() 함수 호출 회수와 무관하게 수신한 데이터를 전송하면 됨 ⇒ 역할
➤ 두 번의 write() 함수 호출을 통해서 데이터를 전송하건, 세 번의 write() 함수 호출을 통해서 데이터를 전송하건 문제 되지 않음

■ 에코 클라이언트 코드

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

- read() 함수 호출을 통해 자신이 전송한 문자열을 한 번에 수신하기를 원함
➤ 데이터의 경계를 구분해야 됨 -> 이런 데이터 송수신 방식은 문제가 됨
➤ TCP의 read(), write() 함수 호출은 데이터의 경계를 구분하지 않기 때문

에코 클라이언트의 해결책!: echo_client2.c

과제 2 분 때

4시간 0분

```
while(1)
{
    fputs("Input message(Q to quit): ", stdout);
    fgets(message, BUF_SIZE, stdin);
    if(!strcmp(message, "q\n") || !strcmp(message, "Q\n"))
        break;
```

```
    str_len = write(sock, message, strlen(message));
```

```
    printf("str_len: %d\n", str_len);
```

```
    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;
```

```
        printf("recv_cnt: %d, recv_len: %d\n", recv_cnt, recv_len);
```

```
    }
```

```
    message[str_len]=0;
```

```
    printf("Message from server: %s", message);
```

```
}
```

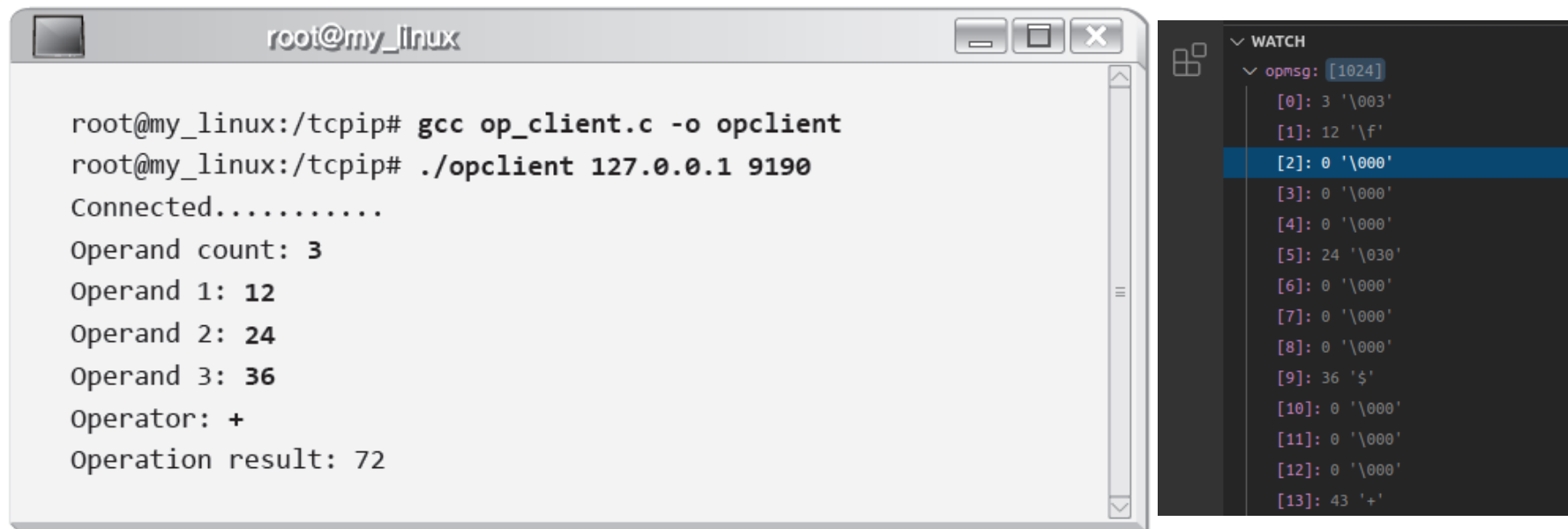
str_len:
전송한 바이트 수

정확히 전송한 바이트
수만큼 데이터 수신

- 전송한 바이트 수만큼 데이터를 수신할 때까지 반복해야 됨
- write() 함수 호출을 통해서 전송한 데이터의 길이만큼 읽어 들이기 위해 반복문 필요
- TCP 기반 데이터 송수신시 부가적으로 필요한 부분임

계산기 프로그램

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 요청되면 클라이언트에는 $4-3-2$ 의 연산결과가 전달되어야 한다. 즉, 뺄셈의 경우에는 첫 번째 정수를 대상으로 뺄셈이 진행되어야 한다.



The image shows a terminal window on the left and a debugger's watch window on the right. The terminal window, titled 'root@my_linux', shows the compilation and execution of a program named 'opclient'. The program receives three operands (12, 24, 36) and an addition operator, resulting in 72. The watch window on the right shows the memory addresses and values of variables, with the 'opmsg' array highlighted. The array contains the sequence of data received from the server: 3, 12, 0, 24, 0, 36, 0, 0, 0, 0, 0, 0, 0, 43.

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

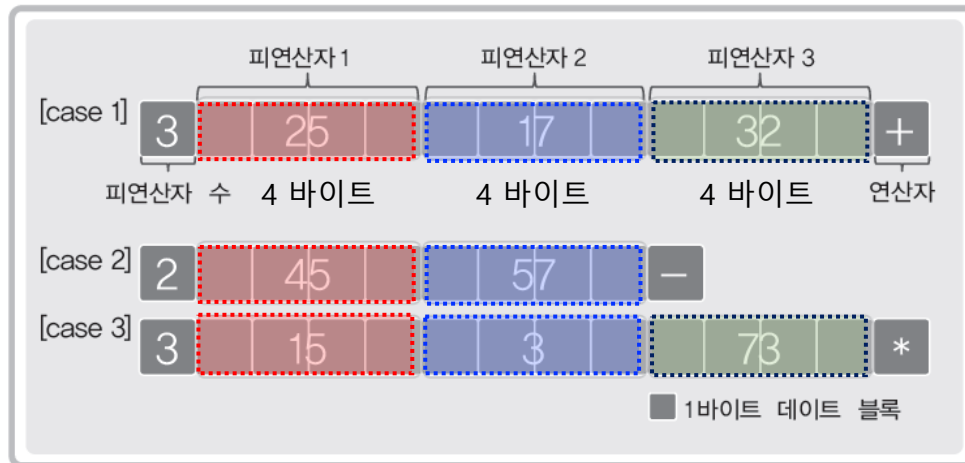
WATCH

Address	Value
[0]: 3	'\003'
[1]: 12	'\f'
[2]: 0	'\000'
[3]: 0	'\000'
[4]: 0	'\000'
[5]: 24	'\030'
[6]: 0	'\000'
[7]: 0	'\000'
[8]: 0	'\000'
[9]: 36	'\$'
[10]: 0	'\000'
[11]: 0	'\000'
[12]: 0	'\000'
[13]: 43	'+'

위의 명세를 기반으로 하는 클라이언트 프로그램의 실행의 예

서버, 클라이언트의 구현

- 클라이언트는 서버에 접속하자마자 피연산자의 개수 정보를 1바이트 정수 형태로 전달함
- 클라이언트가 서버에 전달하는 정수 하나는 4 바이트로 표현
- 정수를 전달한 다음에는 연산의 종류를 전달함. 연산 정보는 1바이트로 전달
- 문자 +, -, * 중 하나를 선택해서 전달함
- 서버는 연산 결과를 4바이트 정수의 형태로 클라이언트에게 전달
- 연산 결과를 얻은 클라이언트는 서버와의 연결을 종료함



프로토콜은 위와 같이(그 이상으로)
명확히 정의해야 한다.

op_server.c op_client.c 참조

▶ 그림 05-1: 클라이언트 op_client.c의 데이터 전송 포맷

op_client.c

— 궁금한 사항 보아
문제는 안 남?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 1024
#define RLT_SIZE 4
#define OPSZ 4
void error_handling(char *message);

int main(int argc, char *argv[])
{
    int sock;
    char opmsg[BUF_SIZE];
    int result, opnd_cnt, i;
    struct sockaddr_in serv_adr;
    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_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=inet_addr(argv[1]);
    serv_adr.sin_port=htons(atoi(argv[2]));
```

```
    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);
    fgetc(stdin);
    - 버퍼에 남아있는 '\n'을 없앴

    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;
}

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

op_server.c #1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUF_SIZE 1024
#define OPSZ 4
void error_handling(char *message);
int calculate(int opnum, int opnds[], char oprator);

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    char opinfo[BUF_SIZE];
    int result, opnd_cnt, i;
    int recv_cnt, recv_len;
    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t clnt_adr_sz;
    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");
```

op_server.c #2

```
memset(&serv_adr, 0, sizeof(serv_adr));
serv_adr.sin_family=AF_INET;
serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
serv_adr.sin_port=htons(atoi(argv[1]));

if(bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))== -1)
    error_handling("bind() error");
if(listen(serv_sock, 5)== -1)
    error_handling("listen() error");
clnt_adr_sz=sizeof(clnt_adr);

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


op_server.c #3

```
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 '*':
            for(i=1; i<opnum; i++) result*=opnds[i];
            break;
    }
    return result;
}

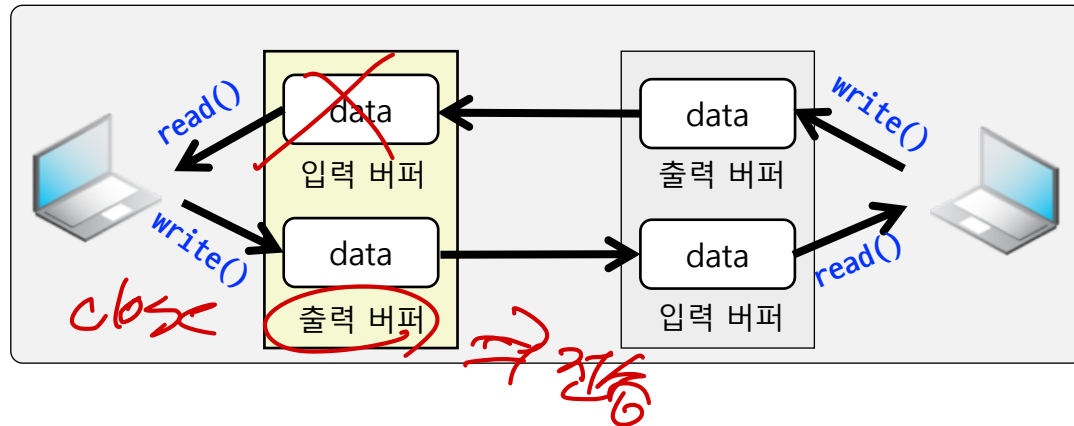
void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

TCP 소켓에 존재하는 입출력 버퍼

- 입출력 버퍼는 TCP 소켓 각각에 대해 별도로 존재한다.
- 입출력 버퍼는 소켓생성시 자동으로 생성된다.
- 소켓을 닫아도 출력 버퍼에 남아있는 데이터는 계속해서 전송이 이뤄진다.
- 소켓을 닫으면 입력 버퍼에 남아있는 데이터는 소멸되어버린다.

이와 같은 버퍼가 존재하기 때문에
데이터의 슬라이딩 윈도우 프로토콜의
적용이 가능

이로 인해서 버퍼가 차고 넘치는 상황은
발생하지 않는다.



소켓 A 야 50바이트까지는 보내도 괜찮아!
소켓 B OK!

소켓 A 내가 20바이트 비웠으니까 70바이트까지 괜찮아
소켓 B OK!

슬라이딩 윈도우 프로토콜의
데이터 송수신 유형

흐름 제어 기법 (Flow control) ✓

■ 흐름 제어 기법

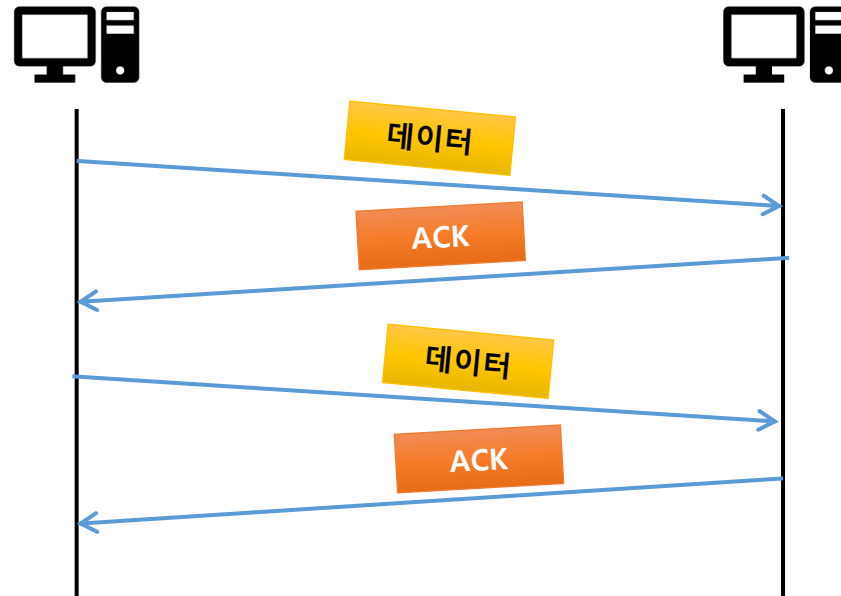
- 송신측과 수신측의 데이터 처리 속도 차이를 해결하기 위한 방법

- 정지-대기(Stop-and-wait) 기법
- 슬라이딩 윈도우 (Sliding Window) 기법

1) ■ Stop-and wait 흐름 제어 기법

- 전송 측이 프레임을 전송한 다음, 각 데이터 프레임에 대한 ACK를 기다림
- ACK 프레임이 도착하면 다음 프레임을 전송함

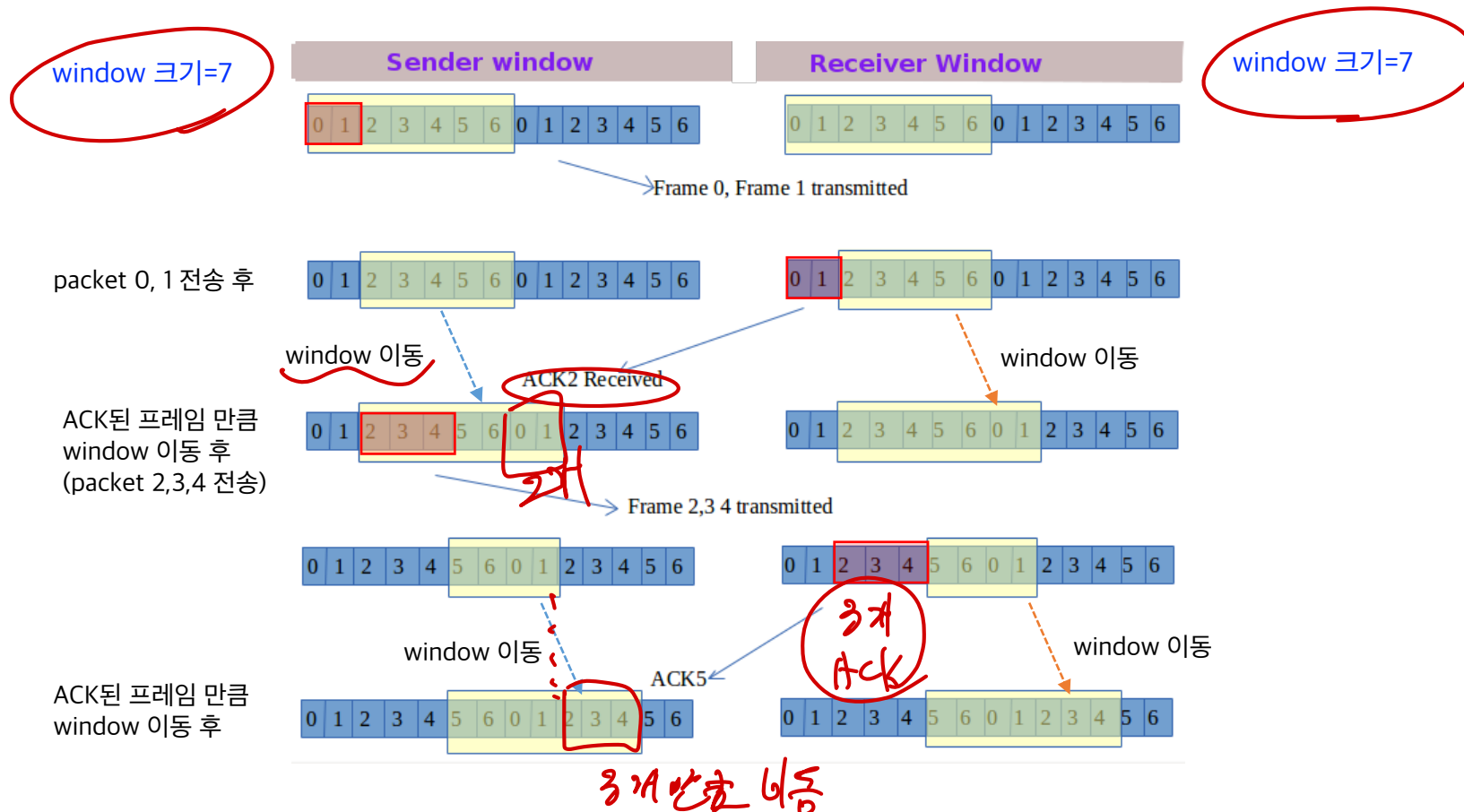
Ack 가 올 때까지 대기!!



슬라이딩 윈도우 (Sliding Window)

■ 슬라이딩 윈도우 동작 방법

- 수신측에서 설정한 윈도우 크기만큼 송신측에서 확인 응답(ACK) 없이 패킷을 전송할 수 있음
 - 데이터 흐름을 동적으로 조절하는 기법
 - Window: 전송 및 수신 측에서 만들어진 **버퍼의 크기**
- ACK 프레임이 도착하면, 전송측 윈도우는 ACK 프레임수에 따라 오른쪽 경계가 이동하여 윈도우 크기가 늘어남



비전서무
서공의
성동과 불

TCP의 동작 원리1: 연결 설정 단계

이거 다 구현이
작업 중이지

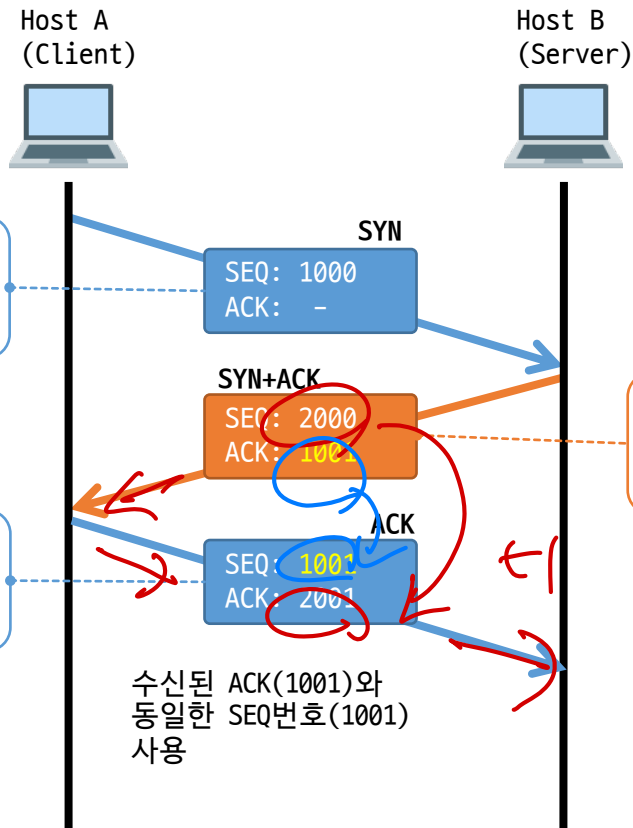
■ 연결 설정 단계: Three-way handshaking

- [Shake 1] 소켓 A: Hi! 소켓 B, 내가 전달할 데이터가 있으니 우리 연결 좀 하자
- [Shake 2] 소켓 B: Okay! 지금 나도 준비가 되었으니 언제든지 시작해도 좋다.
- [Shake 3] 소켓 A: Thank you! 내 요청을 들어줘서 고맙다.

Three-way handshaking (SYN - SYN+ACK - ACK)

“내가 지금 보내는 이 패킷에 1000이라는 번호를 부여하니, 잘 받았다면 다음에는 1001번 패킷을 전달하라고 내게 말해달라!”

“좀 전에 전송한 SEQ가 2000인 패킷은 잘 받았으니, 다음 번에는 SEQ가 2001인 패킷을 전송하기 바란다!”

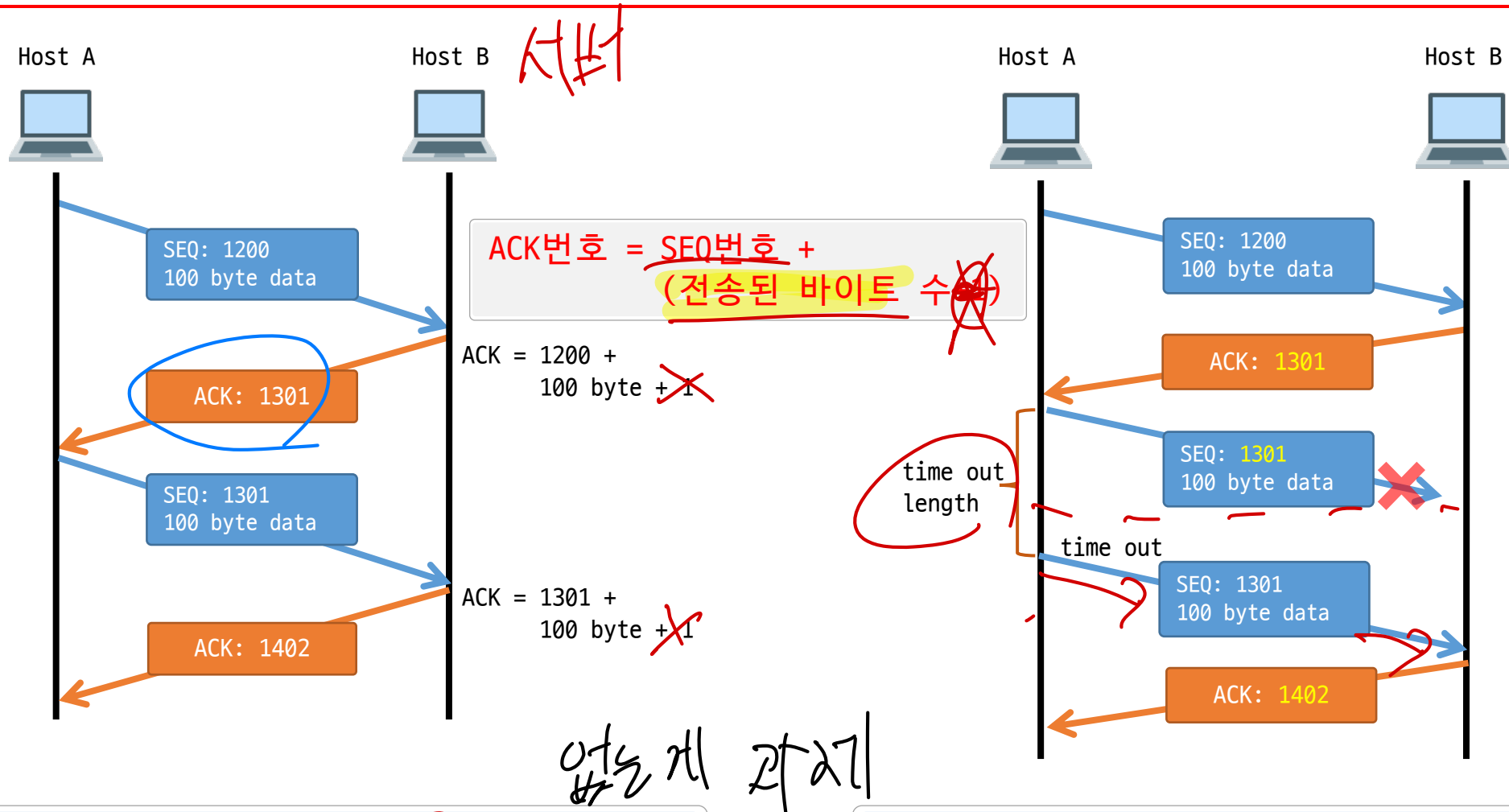


처음 시작 SEQ번호는 랜덤하게 생성됨

“내가 지금 보내는 이 패킷에 2000이라는 번호를 부여하니, 잘 받았다면 다음에는 2001번 패킷을 전달하라고 내게 말해달라!”

ACK번호 = 수신된 SEQ 번호 + 1

TCP의 동작 원리2: 데이터 송수신



ACK의 값을 전송된 바이트 크기만큼 증가시키는 이유

- 패킷의 전송 유무 뿐만 아니라 데이터의 손실 유무까지 확인하기 위함

SEQ 전송 시 타이머 작동

- SEQ에 대한 ACK가 전송되지 않을 경우 데이터 재전송

TCP의 동작 원리3: 상대 소켓과의 연결종료

■ 연결 종료 단계: Four-way handshake

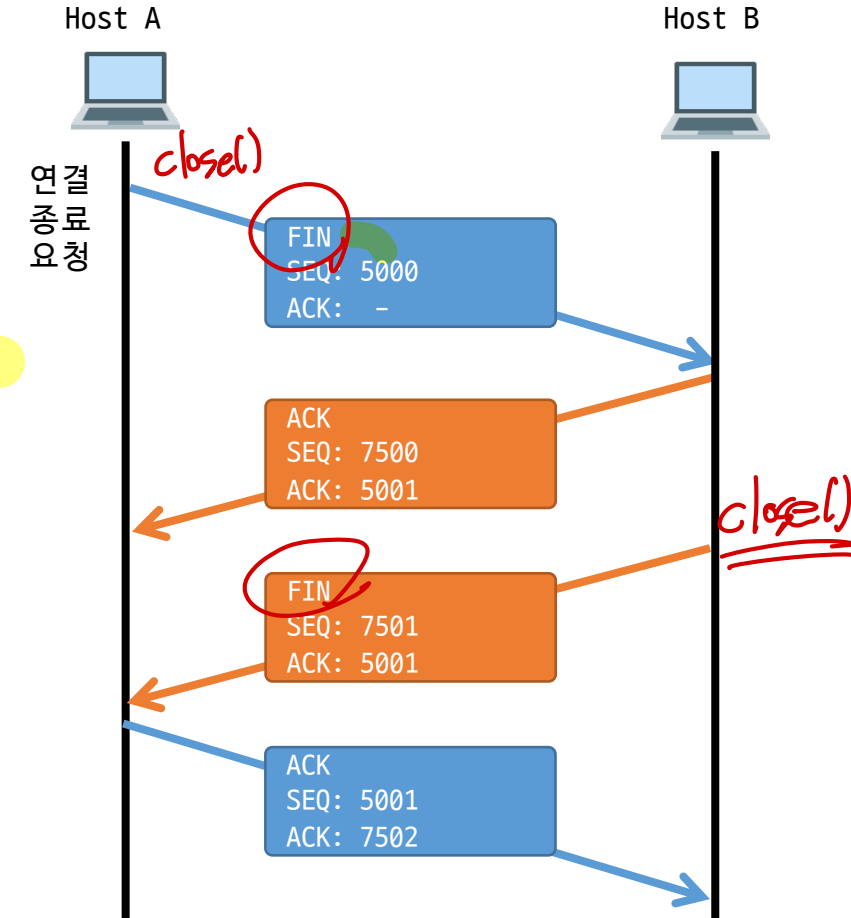
- [Shake 1] 소켓 A: 전 연결을 끊고자 합니다.
- [Shake 2] 소켓 B: 아! 그러세요? 잠시만 기다리세요.
- [Shake 3] 소켓 B: 네 저도 준비가 끝났습니다. 그럼 연결을 끊으시지요.
- [Shake 4] 소켓 A: 네! 그 동안 즐거웠습니다.

Four-way handshake

FIN - ACK - FIN - ACK

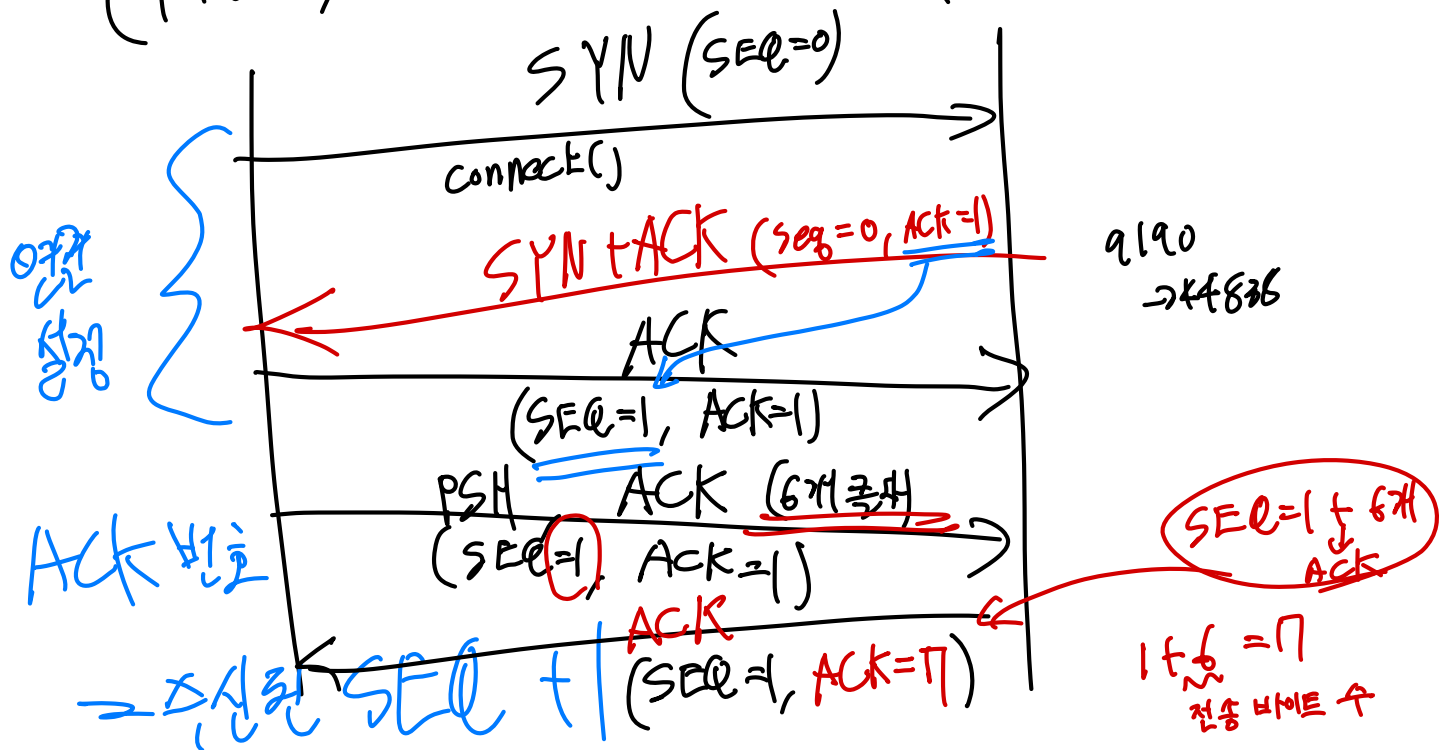
Four-way handshake 과정을 거쳐서 연결을 종료하는 이유

- 일방적 종료로 인한 데이터의 손실을 막기 위함



Client
(44838)

Server
(9190)



Questions?

LMS Q&A 게시판에 올려주세요.