# Embedded Network Programming

윤찬식

chansigi@ioacademy.co.kr
(주)아임구루

# #1. Endian

- 정의: CPU가 데이터에 메모리를 저장하고 해석하는 방식
  - ✓ Intel CPU(x86/x64): Litten Endian
  - ✓ ARM: Big/Litten Endian
- 빅 엔디안: 시작 주소에 상위 바이트부터 기록한다.
- 리틀 엔디안: 시작 주소에 하위 바이트부터 기록한다.

```
void printByteOrder(void* value, int size)
{
    char* p = (char*)value;
    for (int i = 0 ; i < size; ++i) {
        printf("%x ", p[i]);
    }
    putchar('\n');
}

int main()
{
    int value = 0x12345678;

    printByteOrder(&value, sizeof(value));
}
```

# #1. Endian

- 현재 시스템에서 엔디안을 확인하는 방법

```
int main()
{
#if __BYTE_ORDER == __LITTLE_ENDIAN
    printf("Little\n");
#elif __BYTE_ORDER == __BIG_ENDIAN
    printf("Big\n");
#endif
}
```

# #1. Endian

- 컴퓨터의 CPU의 엔디안은 동일하지 않기 때문에, 네트워크에서 데이터를 전송할 때 약속이 필요하다.
  - ✓ 빅 엔디안을 사용합니다.
- 네트워크로 전송되는 데이터에 대해서 빅 엔디안으로 저장해야 하고, 전송받은 데이터를 해석할 때는 각자의 CPU에서 처리하는 방식으로 변환해야 한다.
- Host to Network: htonl, htons
- Network to Host: ntohl, ntohs

```
int int32ToBigEndian(unsigned int n) {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    return  ((n & 0xff000000) >> 24) |
            ((n & 0xff0000) >> 8) |
            ((n & 0xff00) << 8) |
            ((n & 0xff) << 24);
#else
    return n;
#endif
}

int main()
{
    int value = 0x12345678;
    printByteOrder(&value, sizeof(value));

    // value = int32ToBigEndian(value);
    value = htonl(value);
    printByteOrder(&value, sizeof(value));
}
```

# #1. Endian

- inet_addr: IP의 문자열을 32비트 빅 엔디안으로 변환하는 함수
- inet_network: IP의 문자열을 32비트 호스트 엔디안으로 변환하는 함수

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

int main()
{
    const char* address = "127.0.0.1";

    // in_addr_t value = inet_addr(address);
    in_addr_t value = inet_network(address);
    printf("%x\n", value);

    const char* bad = "1.2.3.300";
    value = inet_addr(bad);
    if (value == INADDR_NONE)
    {
        fprintf(stderr, "잘못된 IP 주소 입니다...\n");
    }
}
```

# #2. 표준 입출력

- 표준 라이브러리의 핵심: 고수준 IO
  - ✓ C의 표준 입출력함수는 버퍼링을 합니다.

```c
#include <stdio.h>

int main()
{
    // printf("test logging message...");
    fprintf(stderr, "test logging message...");
    while (1);
}
```

# #2. 표준 입출력

- fflush: FILE 구조체 내부의 버퍼를 비웁니다.
  - ✓ 주의사항: stdin를 통해 입력 버퍼를 비우는 것은 비표준 입니다. (Windows에서만 동작합니다.)

```c
#include <stdio.h>

int main()
{
    FILE* fp = fopen("test.txt", "w");
    if (fp == NULL) {
        fprintf(stderr, "File Open error..\n");
        return -1;
    }

    fprintf(fp, "Hello, Network Prgramming...\n");

    // FILE 구조체의 버퍼를 비운다.
    fflush(fp);

    getchar();
    fclose(fp);
}
```

# #3. 시스템 콜

- 소켓 프로그래밍은 저수준 IO를 사용합니다.
- 저수준 IO: 운영체제에서 제공하는 IO 관련 API
  - ✓ Windows: Windows API
  - ✓ Linux/Unix: System Call
- 시스템 콜은 버퍼링을 제공하지 않기 때문에, 프로그래머가 직접 버퍼를 관리해야 합니다.

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <stdio.h>

int main()
{
    int fd = open("test.txt", O_RDONLY);
    printf("fd : %d\n", fd);

    char buf[1024]; // = {0, };
    int n = read(fd, buf, sizeof buf);

    write(1, buf, n);
    close(fd);
}
```

# #3. 시스템 콜

- File의 내용을 출력하는 cat을 구현해봅시다.

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "argc != 2\n");
        return -1;
    }

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open()");
        return -1;
    }

    int n;
    char buf[1024];
    while ((n = read(fd, buf, sizeof buf)) > 0) {
        write(STDOUT_FILENO, buf, n);
    }

    close(fd);
}
```

# #4. DNS

- 쉘 명령 nslookup을 통해 도메인을 IP 주소로 변환할 수 있습니다.
  - ✓ nslookup naver.com
- gethostbyname: 도메인 주소 정보를 반환합니다.

```c
#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main()
{
    struct hostent *host = gethostbyname("test-server");
    if (host == NULL) {
        fprintf(stderr, "cannot found...\n");
        return -1;
    }

    printf("name: %s\n", host->h_name);

    for (int i = 0 ; host->h_aliases[i] ; i++)
        printf("Aliases: %s\n", host->h_aliases[i]);

    for (int i = 0 ; host->h_addr_list[i]; i++)
        printf("%s\n", inet_ntoa(*(struct in_addr*)host->h_addr_list[i]));
}
```

# #5. Web Client

```c
int main()
{
    int sock = socket(PF_INET,      // protocol
                      SOCK_STREAM,  // TCP / UDP
                      0);

    struct sockaddr_in addr = {0, };
    addr.sin_family = AF_INET;
    addr.sin_port = htons(80);
    addr.sin_addr.s_addr = inet_addr("120.50.131.112");

    int ret = connect(sock, (struct sockaddr*)&addr, sizeof addr);
    if (ret != 0)
    {
        printf("connect to server...\n");
        return -1;
    }

    char buf[] = "GET /\r\n";
    write(sock, buf, strlen(buf));

    int n;
    while ((n = read(sock, buf, sizeof buf)) > 0) {
        write(1, buf, n);
    }

    close(sock);
}
```

# #6. Echo Client 1

```c
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>

int main()
{
    // 1. 소켓 생성
    int csock = socket(PF_INET, SOCK_STREAM, 0);

    // 2. 소켓 구조체에 자신의 IP / PORT 지정 : bind()
    // => 생략 가능, 생략할 경우 비어있는 임의의 포트가 선택된다.

    // 3. 서버에 접속 요청
    struct sockaddr_in addr = {0, };
    addr.sin_family = AF_INET;
    addr.sin_port = htons(5001);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    int ret = connect(csock, (struct sockaddr*)&addr, sizeof addr);
    if (ret == -1) {
        perror("connect()");
        return -1;
    }

    getchar();
    close(csock);
}
```

# #6. Echo Server 1

- 서버 만드는 방법
  - ✓ Socket 생성
  - ✓ 연결을 수락할 IP / PORT 할당
  - ✓ 연결 수락 가능 상태로 변경
  - ✓ 연경 요청 수락
- Time Wait - Address already in use
  - ✓ TIME_WAIT: 특정 포트에 바인드 되어 있는 소켓을 바로 해지하는 것이 아니라, 지연되거나 재전송된 세그먼트들이 연결이 끊긴 이후에 네트워크에서 빠져나가도록 해준다.
  - ✓ TIME_WAIT이 없는 경우: 같은 포트의 새로운 연결을 손상시킬 수 있다.
- 서버에서 TIME_WAIT을 사용하지 않는 이유
  - ✓ 클라이언트의 소켓은 고정되어 있지 않다.
  - ✓ 최신 네트워크 커널 구현은 TCP 패킷의 SYN 번호가 TIME_WAIT 상태의 연결로부터 마지막 SYN 번호보다 클 경우에만 연결을 허용한다.
  - ✓ 일반적으로 서버는 종료되지 않을 뿐더러, 종료 되었을때 바로 시작할 수 있어야 한다.

# #6. Echo Server - TIME_WAIT

```
int main()
{
    int ssock = socket(PF_INET, SOCK_STREAM, 0);

    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(5001);
    saddr.sin_addr.s_addr = INADDR_ANY;

    int option = true;
    setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof option);

    int ret = bind(ssock, (struct sockaddr*)&saddr, sizeof saddr);
    ret = listen(ssock, 5);

    struct sockaddr_in caddr = {0, };
    socklen_t socklen = sizeof(caddr);
    int csock = accept(ssock, (struct sockaddr*)&caddr, &socklen);

    char* cip = inet_ntoa(caddr.sin_addr);
    printf("%s\n", cip);

    getchar();

    close(csock);
    close(ssock);
}
```

# #7. Echo Client

```
int main()
{
    int csock = socket(PF_INET, SOCK_STREAM, 0);

    struct sockaddr_in addr = {0, };
    addr.sin_family = AF_INET;
    addr.sin_port = htons(5001);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    int ret = connect(csock, (struct sockaddr*)&addr, sizeof addr);
    while (true)
    {
        char buf[1024];
        fgets(buf, sizeof buf, stdin);

        write(csock, buf, strlen(buf));
        int len = read(csock, buf, sizeof buf);
        if (len == 0)
        {
            printf("disconnected from server...\n");
            break;
        }
        else if (len == -1)
        {
            perror("read()");
            break;
        }

        write(STDOUT_FILENO, buf, len);
    }

    printf("program end...\n");
    close(csock);
}
```

# #7. Echo Server - 종료 처리

- read의 반환값을 통해 연결의 종료/에러 여부를 확인해야 한다.

```
int main()
{
    // ...
    struct sockaddr_in caddr = {0, };
    socklen_t socklen = sizeof(caddr);
    int csock = accept(ssock, (struct sockaddr*)&caddr, &socklen);

    char* cip = inet_ntoa(caddr.sin_addr);
    printf("client ip: %s\n", cip);

    while (true)
    {
        char buf[1024];
        int len = read(csock, buf, sizeof buf);
        if (len == 0)
        {
            printf("disconnected...\n");
            break;
        }
        else if (len == -1)
        {
            perror("read()");
            break;
        }

        write(csock, buf, len);
    }
    // ...
}
```

# #8. Remote Cat Client

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

    int ret = connect(csock, (struct sockaddr*)&addr, sizeof addr);
    if (ret == -1) {
        perror("connect()");
        return -1;
    }

    int fd = open("client1.cc", O_RDONLY);
    if (fd == -1)
    {
        perror("open()");
        return -1;
    }

    char ch;
    int len;
    while ((len = read(fd, &ch, sizeof ch)) > 0) {
        write(csock, &ch, len);
    }

    close(csock);
}
```

# #8. Remote Cat Server

```
int main()
{
    // ...
    int option = true;
    setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof option);

    int ret = bind(ssock, (struct sockaddr*)&saddr, sizeof saddr);
    ret = listen(ssock, 5);

    struct sockaddr_in caddr = {0, };
    socklen_t socklen = sizeof(caddr);
    int csock = accept(ssock, (struct sockaddr*)&caddr, &socklen);

    char* cip = inet_ntoa(caddr.sin_addr);
    printf("%s\n", cip);

    int len;
    char buf[1024];
    while ((len = read(csock, buf, sizeof buf)) > 0) {
        write(STDOUT_FILENO, buf, len);
    }

    close(csock);
    close(ssock);
}
```

# #8. Nagle Algorithm - Client

```c
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#include <stdio.h>
#include <termios.h>

static struct termios old, newSetting;
void initTermios(int echo)
{
    tcgetattr(0, &old);
    newSetting = old;
    newSetting.c_lflag &= ~ICANON;
    newSetting.c_lflag &= echo ? ECHO : ~ECHO;
    tcsetattr(0, TCSANOW, &newSetting);
}

void resetTermios(void)
{
    tcsetattr(0, TCSANOW, &old);
}
```

# #8. Nagle Algorithm - Client

```
char getch_(int echo)
{
    char ch;
    initTermios(echo);
    ch = getchar();
    resetTermios();
    return ch;
}

char getch(void)
{
    return getch_(0);
}

char getche(void)
{
    return getch_(1);
}

int main()
{
    int csock = socket(PF_INET, SOCK_STREAM, 0);

    struct sockaddr_in addr = {0, };
    addr.sin_family = AF_INET;
    addr.sin_port = htons(5001);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

# #8. Nagle Algorithm - Client

- Nagle 알고리즘: 앞서 전송한 데이터에 대한 ACK 메세지를 수신해야만, 다음 데이터를 전송하는 알고리즘

```
    int ret = connect(csock, (struct sockaddr*)&addr, sizeof addr);
    if (ret == -1) {
        perror("connect()");
        return -1;
    }

    int option = true;
    socklen_t socklen = sizeof(option);
    setsockopt(csock, IPPROTO_TCP, TCP_NODELAY, &option, socklen);

    char ch;
    while ((ch = getch()) != EOF)
    {
        write(csock, &ch, 1);
    }

    close(csock);
}
```

# #8. Nagle Algorithm - Server

```
int main()
{
    int ssock = socket(PF_INET, SOCK_STREAM, 0);

    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(5000);

    saddr.sin_addr.s_addr = INADDR_ANY;

    int option = true;
    setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof option);

    int ret = bind(ssock, (struct sockaddr*)&saddr, sizeof saddr);
    ret = listen(ssock, 5);

    struct sockaddr_in caddr = {0, };
    socklen_t socklen = sizeof(caddr);
    int csock = accept(ssock, (struct sockaddr*)&caddr, &socklen);

    char* cip = inet_ntoa(caddr.sin_addr);
    printf("%s\n", cip);

    int len;
    char buf[1024];
    while ((len = read(csock, buf, sizeof buf)) > 0) {
        printf("len : %d\n", len);
        write(STDOUT_FILENO, buf, len);
    }

    close(csock);
    close(ssock);
}
```

# #9. Calculator Client

```
int main()
{
    // ...
    int ret = connect(csock, (struct sockaddr*)&addr, sizeof addr);

    while (true)
    {
        int lhs, rhs;
        char op;
        int result;
        scanf("%d%c%d", &lhs, &op, &rhs);

        char buf[1024];
        char* p = buf;

        *p = op;
        p += sizeof(op);

        *(int*)p = lhs;
        p += sizeof(lhs);

        *(int*)p = rhs;
        p += sizeof(rhs);

        write(csock, buf, p - buf);

        int len = read(csock, &result, sizeof result);
        printf("result: %d\n", result);
    }

    close(csock);
}
```

# #9. Calculator Server

```
int main()
{
    // ...
    int len;
    char buf[1024];
    int lhs, rhs;
    char op;
    while (true)
    {
        len = read(csock, buf, sizeof buf);

        char* p = buf;
        op = *p;
        p += sizeof(op);

        lhs = *(int*)p;
        p += sizeof(lhs);

        rhs = *(int*)p;
        p += sizeof(rhs);

        int result = calculate(lhs, rhs, op);
        write(csock, &result, sizeof result);
    }
    close(csock);
    close(ssock);
}
```

# #10. Packet

- 패킷을 읽고 쓰는 방법은 항상 동일합니다.
  - ✓ 일반화된 라이브러리를 만들어 봅시다. - Packet.h

```
#ifndef PACKET_H
#define PACKET_H

struct Packet
{
    char *start;
    char *current;

    Packet(char *buf) : start(buf), current(buf) {}

    inline void writeByte(char value)
    {
        *current = value;
        current += sizeof(value);
    }

    inline void writeInt32(int value)
    {
        *(int *)current = value;
        current += sizeof(value);
    }

    inline char readByte()
    {
        char value = *current;
        current += sizeof(value);
        return value;
    }
```

# #10. Packet

```
    inline int readInt32()
    {
        int value = *(int *)current;
        current += sizeof(value);
        return value;
    }

    inline int size()
    {
        return current - start;
    }

    inline char *data()
    {
        return start;
    }
};

#endif
```

# #10. Calculator Client - Packet

```
int main()
{
    // ...
    int ret = connect(csock, (struct sockaddr*)&addr, sizeof addr);

    while (true)
    {
        int lhs, rhs;
        char op;
        int result;
        scanf("%d%c%d", &lhs, &op, &rhs);

        char buf[1024];
        Packet packet(buf);
        packet.writeByte(op);
        packet.writeInt32(lhs);
        packet.writeInt32(rhs);
        write(csock, packet.data(), packet.size());

        int len = read(csock, &result, sizeof result);
        if (len == 0)
            break;
        else if (len == -1)
        {
            perror("read()");
            break;
        }

        printf("result: %d\n", result);
    }

    close(csock);
}
```

# #10. Calculator Server - Packet

```
int main()
{
    // ...
    int len;
    char buf[1024];
    int lhs, rhs;
    char op;
    while (true)
    {
        len = read(csock, buf, sizeof buf);

        Packet packet(buf);
        op = packet.readByte();
        lhs = packet.readInt32();
        rhs = packet.readInt32();

        int result = calculate(lhs, rhs, op);
        write(csock, &result, sizeof result);
    }
    close(csock);
    close(ssock);
}
```

# #11. Calculator Client - Packet

- 문제점: 데이터를 수신하고, 전송만 하는 경우 문제가 발생할 수 있다.

```
int main()
{
    // ...
    int i;
    for (i = 0 ; i < 1000; ++i)
    {
        int lhs = 10, rhs = 32;
        char op = '+';
        int result = 0;

        char buf[1024];
        Packet packet(buf);
        packet.writeByte(op);
        packet.writeInt32(lhs);
        packet.writeInt32(rhs);

        write(csock, packet.data(), packet.size());
    }

    close(csock);
}
```

# #11. Calculator Server - Packet

```
int main()
{
    // ...
    int len;
    char buf[1024];
    int lhs, rhs;
    char op;
    int count = 0;
    while (true)
    {
        len = read(csock, buf, sizeof buf);

        Packet packet(buf);
        op = packet.readByte();
        lhs = packet.readInt32();
        rhs = packet.readInt32();

        int result = calculate(lhs, rhs, op);
        printf("%2d - result: %d\n", ++count, result);
    }

    close(csock);
    close(ssock);
}
```

# #12. Calculator Client - Packet

- 해결방법: TCP는 스트림 지향 프로토콜 입니다. 데이터의 경계가 존재하지 않습니다.
  - ✓ 경계 처리를 위해, 데이터를 보내기 전에 길이를 먼저 보냅니다.

```
int main()
{
    // ...
    int i;
    for (i = 0 ; i < 1000; ++i)
    {
        int lhs = 10, rhs = 32;
        char op = '+';
        int result = 0;

        char buf[1024];
        Packet packet(buf);
        packet.writeByte(op);
        packet.writeInt32(lhs);
        packet.writeInt32(rhs);

        int len = packet.size();
        write(csock, &len, sizeof len);
        write(csock, packet.data(), len);
    }

    close(csock);
}
```

# #12. Calculator Server - Packet

```
int main()
{
    // ...
    int count = 0;
    int packetlen;
    while (true)
    {
        len = read(csock, &packetlen, sizeof packetlen);
        len = read(csock, buf, packetlen);

        Packet packet(buf);
        op = packet.readByte();
        lhs = packet.readInt32();
        rhs = packet.readInt32();

        int result = calculate(lhs, rhs, op);
        printf("%2d - result: %d\n", ++count, result);
    }

    close(csock);
    close(ssock);
}
```

# #13. Calculator Server - readn

- 아직 온전한 패킷이 전달되지 않은 상태에서 read가 호출된다면, 문제가 발생합니다.
  - ✓ 해결 방법: readn을 통해서 온전한 패킷의 도착을 보장해주어야 합니다.

```
int readn(int fd, void* buf, int len)
{
     int n = len;
     int ret;

     while (n > 0)
     {
          ret = read(fd, buf, n);
          if (ret < 0) return -1;
          else if (ret == 0) return len - n;

          buf = (char*)buf + ret;
          n -= ret;
     }

     return len;
}
```

# #13. Calculator Server - readn

```
int main()
{
     // ...
     int len;
     int lhs, rhs;
     char op;
     int count = 0;
     int packetlen;
     while (true)
     {
          len = readn(csock, &packetlen, sizeof packetlen);
          len = readn(csock, buf, packetlen);

          Packet packet(buf);
          op = packet.readByte();
          lhs = packet.readInt32();
          rhs = packet.readInt32();

          int result = calculate(lhs, rhs, op);
          printf("%2d - result: %d\n", ++count, result);
     }

     close(csock);
     close(ssock);
}
```

# #14. Protocol Buffer

- 프로토콜은 언어에 독립적입니다.
  - ✓ 프로토콜이 변경될 때마다, 각 언어로 구현된 프로토콜 분석의 코드는 변경되어야 합니다.
    - 해결방법: Google Protocol Buffer
    - protoc calculator.proto --cpp_out=.

```
message Calculator {
     required int32 op = 1;
     required int32 lhs = 2;
     required int32 rhs = 3;
}
```

# #15. Protocol Buffer - Client

```
#include "calculator.pb.h"

int main()
{
    // ...

     int i;
     char buf[1024];
     for (i = 0 ; i < 50000; ++i)
     {
          int lhs = 10000, rhs = 32;
          char op = '+';
          int result = 0;

          Calculator* calc = new Calculator;
          calc->set_op(op);
          calc->set_lhs(lhs);
          calc->set_rhs(rhs);

          int len = calc->ByteSize();
          calc->SerializeToArray(buf, len);

          write(csock, &len, sizeof len);
          write(csock, buf, len);
     }

     close(csock);
}
```

# #15. Protocol Buffer - Server

```
#include "calculator.pb.h"

int main()
{
     // ...
     char buf[1024];
     int lhs, rhs;
     char op;
     int count = 0;
     int packetlen;
     while (true)
     {
          len = readn(csock, &packetlen, sizeof packetlen);
          len = readn(csock, buf, packetlen);

          Calculator calc;
          calc.ParseFromArray(buf, packetlen);

          op = calc.op();
          lhs = calc.lhs();
          rhs = calc.rhs();

          int result = calculate(lhs, rhs, op);
          printf("%2d - result: %d\n", ++count, result);
     }

     close(csock);
     close(ssock);
}
```

# #16. Multi Process Model

- 서버는 하나의 클라이언트가 아닌 동시에 여러 클라이언트에게 서비스를 제공할 수 있어야 합니다.
- fork()
  - ✓ fork()의 반환 값을 통해 부모 프로세스 / 자식 프로세스의 로직을 분리할 수 있습니다.
  - ✓ 연결이 종료된 클라이언트를 처리하는 자식 프로세스

```
#include <signal.h>
#include <sys/wait.h>

void onClose(int signum)
{
     while (waitpid(-1, 0, WNOHANG) > 0)
          printf("Child process exited...\n");
}
```

# #16. Multi Process Model

- 사용하지 않는 파일 디스크립터에 대해서는 반드시 close 해주어야 합니다.

```
int main()
{
	signal(SIGCHLD, &onClose);

	// …

	while (true)
	{
		struct sockaddr_in caddr = {0, };
		socklen_t socklen = sizeof(caddr);
		int csock = accept(ssock, (struct sockaddr*)&caddr, &socklen);

		if (fork() == 0)
		{
			char buf[1024];
			while (true)
			{
				int len = read(csock, buf, sizeof buf);
				if (len <= 0)
					break;

				write(1, buf, len);
			}

			close(csock);
			exit(0);
		}

		close(csock);
	}

	close(ssock);
}
```

# #16. Thread

- 병행성(Concurrency): 동시에 수행되는 것처럼 보이지만, 순차적으로 발생
- 병렬성(Parallelism): 동시에 수행되는 병렬적인 작업
- 리눅스에서는 pthread 라이브러리를 통해 멀티 스레드 프로그래밍이 가능합니다.

```
#include <pthread.h>
#include <stdio.h>

void* foo(void* p)
{
     printf("foo\n");
     return (void*)0xff;
}

int main()
{
     pthread_t thread;
     pthread_create(&thread, 0, &foo, 0);

     void* result;
     pthread_join(thread, &result);
     printf("result : %p\n", result);
}
```

# #16. Thread

- 스레드는 같은 프로세스 주소 공간 안에서 동작합니다.
  - ✓ 스레드가 접근하는 메모리가 유효한 영역인지에 대해서 항상 주의해야 합니다.
  - ✓ 하나의 스레드가 비정상적으로 종료되면, 프로세스가 종료합니다.
    - 프로세스 내부의 모든 스레드가 종료합니다.

```
void* goo(void* arg)
{
	sleep(1);

	int* p  = (int*)arg;
	for (int i = 0 ; i < 10; ++i)
	{
		printf("%d\n", p[i]);
	}

	return 0;
}

void* foo(void* arg)
{
	int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

	pthread_t thread;
	pthread_create(&thread, 0, &goo, arr);
	pthread_detach(thread);

	return 0;
}
```

# #16. Thread

- 스레드는 종료 처리가 중요합니다.
  - ✓ 스레드 함수가 반환한다.
  - ✓ pthread_exit()
  - ✓ pthread_cancel()
  - ✓ 프로세스 종료

```
class Resource
{
public:
     Resource() { printf("자원 생성...\n"); }
     ~Resource() { printf("자원 해지...\n"); }
};

void* foo(void* arg)
{
     Resource resource;  // 지역 객체
     Resource* p = new Resource;

     sleep(100);

     delete p;
}

int main()
{
     pthread_t thread;
     pthread_create(&thread, 0, &foo, 0);
     // pthread_join(thread, 0);
     sleep(3);
     pthread_cancel(thread);
     printf("cancel....\n");

     getchar();
}
```

# #16. Thread

- 데이터 경쟁 상태(Race condition)를 해결하기 위해서는 동기화가 필요합니다.
  - ✓ mutex : pthread_mutex_lock / pthread_mutex_unlock

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* foo(void* p)
{
     static int x;

     for (int i = 0 ; i < 1000 ; ++i)
     {
          pthread_mutex_lock(&mutex);
          x = 100;
          x += 1;
          printf("%s: %d\n", (const char*)p, x);
          pthread_mutex_unlock(&mutex);
     }

     return 0;
}

int main()
{
     pthread_t threads[3];
     pthread_create(&threads[0], 0, &foo, (void*)"A");
     pthread_create(&threads[1], 0, &foo, (void*)"  B");
     pthread_create(&threads[2], 0, &foo, (void*)"    C");

     for (int i = 0 ; i < 3 ; ++i)
          pthread_join(threads[i], 0);

     printf("program end...\n");
}
```

# #16. Thread

- RAII를 이용하면 동기화 객체를 안전하게 사용할 수 있습니다.
  - ✓ 예외로 인한 데드락 문제를 해결할 수 있습니다.

```
class Mutex
{
	pthread_mutex_t mutex;
public:
	Mutex()
	{
		pthread_mutex_init(&mutex, 0);
	}

	void lock()  { pthread_mutex_lock(&mutex);  printf("lock\n"); }
	void unlock() { pthread_mutex_unlock(&mutex);  printf("unlock\n"); }

	class AutoLock
	{
		Mutex& mutex;
	public:
		inline AutoLock(Mutex& m): mutex(m)  { mutex.lock(); }
		inline ~AutoLock()                   { mutex.unlock(); }
	};
};

static Mutex mutex;
void* foo(void* p)
{
	static int x;

	for (int i = 0 ; i < 1000 ; ++i)
	{
		Mutex::AutoLock lock(mutex);
		x = 100;
		x += 1;
		printf("%s: %d\n", (const char*)p, x);
	}

	return 0;
}
```

# #16. Thread - TLS

- 함수가 정적 메모리를 사용하는 변수를 사용하면 재진입이 불가능합니다.
  - ✓ TLS를 이용하면 됩니다.
    - 정적 TLS: 컴파일러 확장 명령 __thread
    - 동적 TLS: pthread_setspecific / pthread_getspecific

```
int next3times()
{
     static __thread int n = 0;
     n += 3;
     return n;
}

void* foo(void* p)
{
     printf("%s: %d\n", (char*)p, next3times());
     printf("%s: %d\n", (char*)p, next3times());
     printf("%s: %d\n", (char*)p, next3times());

     return 0;
}

int main()
{
     pthread_t threads[2];
     pthread_create(&threads[0], 0, &foo, (void*)"A");
     pthread_create(&threads[1], 0, &foo, (void*)"B");

     for (int i = 0 ; i < 2; ++i)
          pthread_join(threads[i], 0);
}
```

# #16. Thread - Atomic Operations

- 특정 변수에 값을 대입하거나 연산하는 작업을 뮤텍스를 이용해서 처리하면 비효율적입니다.
  - ✓ CPU에서 제공하는 원자적 연산을 이용하면 됩니다.

```
int x = 0;
void* foo(void* p)
{
     for (int i = 0 ; i < 100000; ++i)
     {
          __sync_fetch_and_add(&x, 1);

          #if 0
          asm (
               "movl     x, %eax \n"
               "addl     $1, %eax \n"
               "movl     %eax, x \n"
          );
          #endif

     }

     return 0;
}

int main()
{
     pthread_t thread[3];
     for (int i = 0 ; i < 3; ++i)
          pthread_create(&thread[i], 0, &foo, 0);

     for (int i = 0 ; i < 3; ++i)
          pthread_join(thread[i], 0);

     printf("result : %d\n", x);
}
```

# #16. Thread - volatile

- 다른 스레드에게 변경된 값을 관찰할 수 없는 문제가 발생합니다.
  - ✓ 메모리 가시성 문제

```
void* foo(void* p)
{
     volatile int* pn = (int*)p;
     while (*pn) {}
     printf("foo finish\n");

     return 0;
}

int main()
{
     int n = 1;
     pthread_t thread;
     pthread_create(&thread, 0, &foo, &n);
     getchar();

     n = 0;
     pthread_join(thread, 0);
     printf("main finish\n");
}
```

# #16. Thread - volatile

- 다른 스레드에게 변경된 값을 관찰할 수 없는 문제가 발생합니다.
  - ✓ 메모리 가시성 문제

```
void* foo(void* p)
{
     volatile int* pn = (int*)p;
     while (*pn) {}
     printf("foo finish\n");

     return 0;
}

int main()
{
     int n = 1;
     pthread_t thread;
     pthread_create(&thread, 0, &foo, &n);
     getchar();

     n = 0;
     pthread_join(thread, 0);
     printf("main finish\n");
}
```

# #16. Thread - false sharing

- 아래의 프로그램은 멀티 코어에서 성능 문제가 발생합니다.

```
struct data
{
     long a;
     long b;
};

struct data data;
void* foo(void* p) {
     for (int i = 0; i < 500000000; ++i)
          data.a += 1;
     return 0;
}

void* goo(void* p) {
     for (int i = 0; i < 500000000; ++i)
          data.b += 1;
     return 0;
}

int main()
{
     pthread_t thread[2];
     pthread_create(&thread[0], 0, &foo, 0);
     pthread_create(&thread[1], 0, &goo, 0);

     for (int i = 0 ; i < 2; ++i)
          pthread_join(thread[i], 0);
     printf("%ld %ld\n", data.a, data.b);
}
```

# #16. Thread - false sharing

- 각 코어에서 동작하는 스레드에서 접근하는 데이터가 같은 캐시 라인을 공유하는 경우, 캐시가 지속적으로 무효화되는 문제가 발생합니다.

```
struct data
{
     long a __attribute__ ((aligned (64)));
     long b __attribute__ ((aligned (64)));
};

struct data data;
void* foo(void* p) {
     for (int i = 0; i < 500000000; ++i)
          data.a += 1;
     return 0;
}

void* goo(void* p) {
     for (int i = 0; i < 500000000; ++i)
          data.b += 1;
     return 0;
}

int main()
{
     pthread_t thread[2];
     pthread_create(&thread[0], 0, &foo, 0);
     pthread_create(&thread[1], 0, &goo, 0);

     for (int i = 0 ; i < 2; ++i)
          pthread_join(thread[i], 0);
     printf("%ld %ld\n", data.a, data.b);
}
```

# #17. Thread - Scalability

- 동시에 수행할 수 있는 스레드의 개수가 많을 수록, 프로그램의 성능은 상승해야 합니다.

```
int sum = 0;
int num_of_threads = 1;

void* foo(void* arg)
{
     for (int i = 0; i < 50000000 / num_of_threads; i++)
          sum += 2;
     return 0;
}

int current_ms()
{
     struct timeval val;
     unsigned int ms;
     gettimeofday(&val, 0);

     ms = val.tv_sec * 1000 + val.tv_usec / 1000;
     return ms;
}

int main()
{
     pthread_t thread[32];

     for (num_of_threads = 1;
               num_of_threads <= 16;
               num_of_threads *= 2) {

          sum = 0;
          int start = current_ms();
          for (int i = 0; i < num_of_threads; ++i)
               pthread_create(&thread[i], 0, &foo, 0);

          for (int i = 0; i < num_of_threads; ++i)
               pthread_join(thread[i], 0);

          int ms = current_ms() - start;
          printf("%d threads, Result is %d, %dms\n",
                    num_of_threads, sum, ms);
     }
}
```

# #17. Thread - Scalability

- Mutex
  - ✓ 1 threads, Result is 100000000, 1010ms
  - ✓ 2 threads, Result is 100000000, 5259ms
  - ✓ 4 threads, Result is 100000000, 5003ms
  - ✓ 8 threads, Result is 100000000, 4025ms
  - ✓ 16 threads, Result is 100000000, 5093ms

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* foo(void* arg)
{
     for (int i = 0; i < 50000000 / num_of_threads; i++)
     {
          pthread_mutex_lock(&mutex);
          sum += 2;
          pthread_mutex_unlock(&mutex);
     }
     return 0;
}
```

# #17. Thread - Scalability

- Spinlock
  - ✓ 1 threads, Result is 100000000, 411ms
  - ✓ 2 threads, Result is 100000000, 2250ms
  - ✓ 4 threads, Result is 100000000, 3353ms
  - ✓ 8 threads, Result is 100000000, 6582ms
  - ✓ 16 threads, Result is 100000000, 11546ms

```
static pthread_spinlock_t spinlock;
void* foo(void* arg)
{
     for (int i = 0; i < 50000000 / num_of_threads; i++)
     {
          pthread_spin_lock(&spinlock);
          sum += 2;
          pthread_spin_unlock(&spinlock);
     }
     return 0;
}
```

# #17. Thread - Scalability

- Atomic Operations
  - ✓ 1 threads, Result is 100000000, 303ms
  - ✓ 2 threads, Result is 100000000, 1443ms
  - ✓ 4 threads, Result is 100000000, 1433ms
  - ✓ 8 threads, Result is 100000000, 1660ms
  - ✓ 16 threads, Result is 100000000, 1691ms

```
void* foo(void* arg)
{
     for (int i = 0; i < 50000000 / num_of_threads; i++)
     {
          __sync_fetch_and_add(&sum, 2);
     }
     return 0;
}
```

# #17. Thread - Scalability

- 동기화는 프로그램의 확장성에 악영향을 미칩니다.
  - ✓ 최대한 병렬성을 살릴 수 있도록 코드를 작성해야 합니다.
  - ✓ Map-Reduce Model(Fork-Join Model)

```
void* foo(void* arg)
{
     int localsum = 0;
     for (int i = 0; i < 50000000 / num_of_threads; i++)
     {
          localsum += 2;
     }

     __sync_fetch_and_add(&sum, localsum);
     return 0;
}
```

# #17. Thread - 병렬 라이브러리

- HW마다 최적화된 스레드의 개수는 다릅니다.
  - ✓ Core 2: 4 ~ 6개 / Core 8: 16 ~ 24개
- 스레드를 직접 생성해서 사용하면, 프로그램의 코드는 HW가 달라질 때마다 변경되어야 합니다.
  - ✓ 병렬 라이브러리를 이용하는 것이 좋습니다.

```
#include <tbb/tbb.h>
using namespace tbb;

bool is_prime(int n) {
     if (n < 2)
          return false;

     for (int i = 2; i < n; ++i) {
          if ((n % i) == 0)
               return false;
     }

     return true;
}

int main()
{
     array<int, 200000> arr;

     iota(arr.begin(), arr.end(), 0);
     int start = current_ms();

     combinable<long> sum;
     parallel_for_each(arr.begin(), arr.end(), [&sum](int i) {
          sum.local() += is_prime(i) ? i : 0;
     });

     printf("sum: %ld\n", sum.combine(plus<int>()));
     printf("ms: %d\n", current_ms() - start);
}
```

# #17. Chat Client

```
void* readHandler(void* arg)
{
	int csock = *(int*)arg;
	free(arg);

	char buf[1024];
	while (true)
	{
		int len = read(csock, buf, sizeof buf);
		if (len == 0)
		{
			printf("end connection...\n");
			break;
		}
		else if (len == -1)
		{
			perror("read()");
			break;
		}

		write(1, buf, len);
	}

}

int main()
{
	// ...

	int* arg = (int*)malloc(sizeof csock);
	*arg = csock;

	pthread_t thread;
	pthread_create(&thread, 0, &readHandler, arg);
	pthread_detach(thread);

	char buf[1024];
	while (fgets(buf, sizeof buf, stdin))
	{
		if (write(csock, buf, strlen(buf)) < 0)
		{
			perror("write()");
			break;
		}
	}

	close(csock);
}
```

# #17. Chat Server

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <pthread.h>

static void* connectionHandler(void* arg)
{
	int csock = *(int*)arg;
	free(arg);

	char buf[1024];
	while (true)
	{
		int len = read(csock, buf, sizeof buf);
		if (len == 0)
		{
			printf("disconnect from client...
\n");
			break;
		}
		else if (len == -1)
		{
			perror("read()");
			break;
		}

		write(csock, buf, len);
	}

	close(csock);
	printf("thread exit...\n");
}
```

```
int main()
{
	int ssock = socket(PF_INET, SOCK_STREAM, 0);

	struct sockaddr_in saddr;
	saddr.sin_family = AF_INET;
	saddr.sin_port = htons(5001);
	saddr.sin_addr.s_addr = INADDR_ANY;

	int option = true;
	setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR,
&option, sizeof option);
	int ret = bind(ssock, (struct sockaddr*)&saddr,
sizeof saddr);
	if (ret == -1)
	{
		perror("bind()");
		return -1;
	}

	ret = listen(ssock, 5);
	if (ret == -1)
	{
		perror("listen()");
		return -1;
	}

	while (true)
	{
		struct sockaddr_in caddr = {0, };
		socklen_t socklen = sizeof(caddr);
		int csock = accept(ssock, (struct
sockaddr*)&caddr, &socklen);
		printf("connect from client...\n");

		pthread_t thread;
		int* arg = (int*)malloc(sizeof csock);
		*arg = csock;

		pthread_create(&thread, 0,
&connectionHandler, arg);
		pthread_detach(thread);
	}

	close(ssock);
}
```

# #17. Chat Server2

```
static vector<int> clients;

void addClient(int sock) {
     clients.push_back(sock);
}

void removeClient(int sock) {
     clients.erase(
          remove(clients.begin(), clients.end(), sock),
          clients.end());
}

void broadcast(char* buf, int len)
{
     for (int sock : clients)
          write(sock, buf, len);
}

static void* connectionHandler(void* arg)
{
     int csock = *(int*)arg;
     free(arg);

     addClient(csock);

     char buf[1024];
     while (true)
     {
          int len = read(csock, buf, sizeof buf);
          if (len == 0)
          {
               break;
          }
          else if (len == -1)
          {
               perror("read()");
               break;
          }

          broadcast(buf, len);
     }

     close(csock);
     removeClient(csock);
     printf("disconnect from client...\n");
}
```

# #17. Chat Server2

- 클라이언트를 추가하거나, 제거하는 모든 작업은 동시에 일어날 수 있다.
  - ✓ 동기화가 필요합니다.

```
class ScopedLock
{
	pthread_mutex_t& mutex;
public:
	ScopedLock(pthread_mutex_t& m) : mutex(m)
	{ pthread_mutex_lock(&mutex); }

	~ScopedLock()
	{ pthread_mutex_unlock(&mutex); }
};

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void addClient(int sock)
{
	ScopedLock lock(mutex);
	clients.push_back(sock);
}

void removeClient(int sock)
{
	ScopedLock lock(mutex);
	clients.erase(
		remove(clients.begin(), clients.end(), sock),
		clients.end());
}

void broadcast(char* buf, int len)
{
	ScopedLock lock(mutex);
	for (int sock : clients)
		write(sock, buf, len);
}
```

# #17. Chat Server2

- Thread per Connection 모델의 문제점
  - ✓ 엄청나게 많은 스레드가 생성된다.
  - ✓ 프로세서 보다 훨씬 많은 스레드가 만들어져 동작 하면, 대부분의 스레드는 대기 상태에 있다.
    - 컨텍스트 스위칭 비용이 크다.
  - ✓ OS 마다 만들 수 있는 스레드의 개수는 한계가 있다.
- 일정한 수준 까지는 스레드를 추가적으로 만들어 사용하면, 성능상의 이점을 얻을 수 있지만, 점점 성능이 떨어진다.

# #18. select

```
int main() {
     // ...
      fd_set socks, readsocks;
      int maxfd = ssock;

      FD_ZERO(&socks);
      FD_SET(ssock, &socks);

      int maxfds = ssock;
      while (1) {
            readsocks = socks;
            ret = select(maxfds + 1, &readsocks, 0, 0, 0);
            if (ret == 0) continue;
            for (int i = 0; i < maxfds + 1; ++i) {
                  if (FD_ISSET(i, &readsocks)) {
                        if (i == ssock) {
                              struct sockaddr_in caddr;

                              socklen_t size = sizeof(caddr);
                              int csock =
                                    accept(ssock, (struct sockaddr*)&caddr, &size);

                              FD_SET(csock, &socks);
                              if (maxfds < csock) maxfds = csock;

                              char* client_ip = inet_ntoa(caddr.sin_addr);
                              printf("클라이언트 %s 가 접속되었습니다.\n", client_ip);
                        } else {
                              int n;
                              char buf[1024];

                              int sock = i;
                              n = read(sock, buf, sizeof(buf));

                              if (n <= 0) {
                                    printf("연결 종료!\n");
                                    close(sock);
                                    FD_CLR(sock, &socks);
                              } else
                                    n = write(sock, buf, n);
                        }
                  }
            }
      }
      close(ssock);
}
```

# #19. poll

```
int main() {
  // ...
  listen(ssock, 5);
  struct pollfd fds[1024];

  int nfds = 1;
  fds[0].fd = ssock;
  fds[0].events = POLLIN;

  bool compress = false;
  while (1) {
    poll(fds, nfds, -1);

    if (fds[0].revents & POLLIN) {
      struct sockaddr_in caddr;
      socklen_t addrlen = sizeof caddr;
      int csock = accept(ssock, (struct
sockaddr*)&caddr, &addrlen);
      printf("new connection\n");

      fds[nfds].fd = csock;
      fds[nfds].events = POLLIN;
      nfds++;
    }

    for (int i = 1; i < nfds; ++i) {
      if (!fds[i].revents & POLLIN) continue;

      int csock = fds[i].fd;
      char buf[1024];
      int len = read(csock, buf, sizeof buf);
      if (len <= 0) {
        printf("disconnect from client...\n");
        close(csock);

        fds[i].fd = -1;
        compress = true;
        continue;
      }

      write(csock, buf, len);
    }
```

```
    if (compress) {
      compress = false;
      for (int i = 0; i < nfds; ++i) {
        if (fds[i].fd == -1) {
          for (int j = i; j < nfds; ++j) fds[j].fd =
fds[j + 1].fd;
          --i;
          --nfds;
        }
      }
    }
  }

  close(ssock);
}
```

# #20. select vs poll

- poll 시스템 콜은 시스템 V 에서 제공하는 다중 입출력 방식
- poll은 select의 결점을 보완하지만, 이식성은 select가 더 높다.
- poll은 파일 디스크립터에 더할 필요가 없다.
- poll은 파일 디스크립터 숫자가 큰 경우 좀 더 효율적으로 동작한다.
  - ✓ select는 매번 해당 비트까지 검사해야 한다.
- select의 파일 디스크립터가 연속적이지 않고 드문드문 흩어져 있는 경우 심각하다.  poll은 딱 맞는 크기의 배열 하나만 사용하면 된다.
- select를 사용하면 파일 디스크립터 집합을 반환하는 시점에 재구성되므로 매번 초기화를 해주어야 한다.

# #21. epoll

```
int main() {
  int ssock = socket(PF_INET, SOCK_STREAM, 0);

  struct sockaddr_in saddr = {
      0,
  };
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(5000);
  saddr.sin_addr.s_addr = INADDR_ANY;

  int option = 1;
  setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR, &option,
sizeof option);
  if (bind(ssock, (struct sockaddr*)&saddr, sizeof
saddr) == -1) {
    perror("bind()");
    return -1;
  }

  if (listen(ssock, SOMAXCONN) == -1) {
    perror("listen()");
    return -1;
  }

  int efd = epoll_create1(0);
  if (efd == -1) {
    perror("epoll_create1()");
    return -1;
  }

  // server socket
  struct epoll_event event;
  event.events = EPOLLIN;
  event.data.fd = ssock;
  epoll_ctl(efd, EPOLL_CTL_ADD, ssock, &event);

  struct epoll_event events[1024];
```

```
while (true) {
    int count = epoll_wait(efd, events, 1024, -1);

    for (int i = 0; i < count; ++i) {
      if (events[i].data.fd == ssock) {
        struct sockaddr_in caddr = {
            0,
        };
        socklen_t addrlen = sizeof caddr;
        int csock = accept(ssock, (struct
sockaddr*)&caddr, &addrlen);

        event.events = EPOLLIN;
        event.data.fd = csock;
        epoll_ctl(efd, EPOLL_CTL_ADD, csock, &event);
        printf("connected from client...\n");
      } else {
        int csock = events[i].data.fd;

        char buf[1024];
        int len = read(csock, buf, sizeof buf);
        if (len == 0 || len == -1) {
          printf("disconnected from client...\n");
          close(csock);

          epoll_ctl(efd, EPOLL_CTL_DEL, csock, NULL);
          continue;
        }

        write(csock, buf, len);
      }
    }
  }

  close(ssock);
}
```

# #21. epoll

- epoll 장점
  - ✓ 상태 변화 확인을 위한 전체 파일 디스크립터를 대상으로 한 반복문이 필요없다.
  - ✓ select 처럼 매번 관찰하고자 하는 디스크립터의 정보를 전달할 필요없다.

- epoll_create: 저장소 생성
- epoll_ctl:  저장소에 디스크립터를 등록 / 삭제
- epoll_wait: poll, select

# #22. epoll

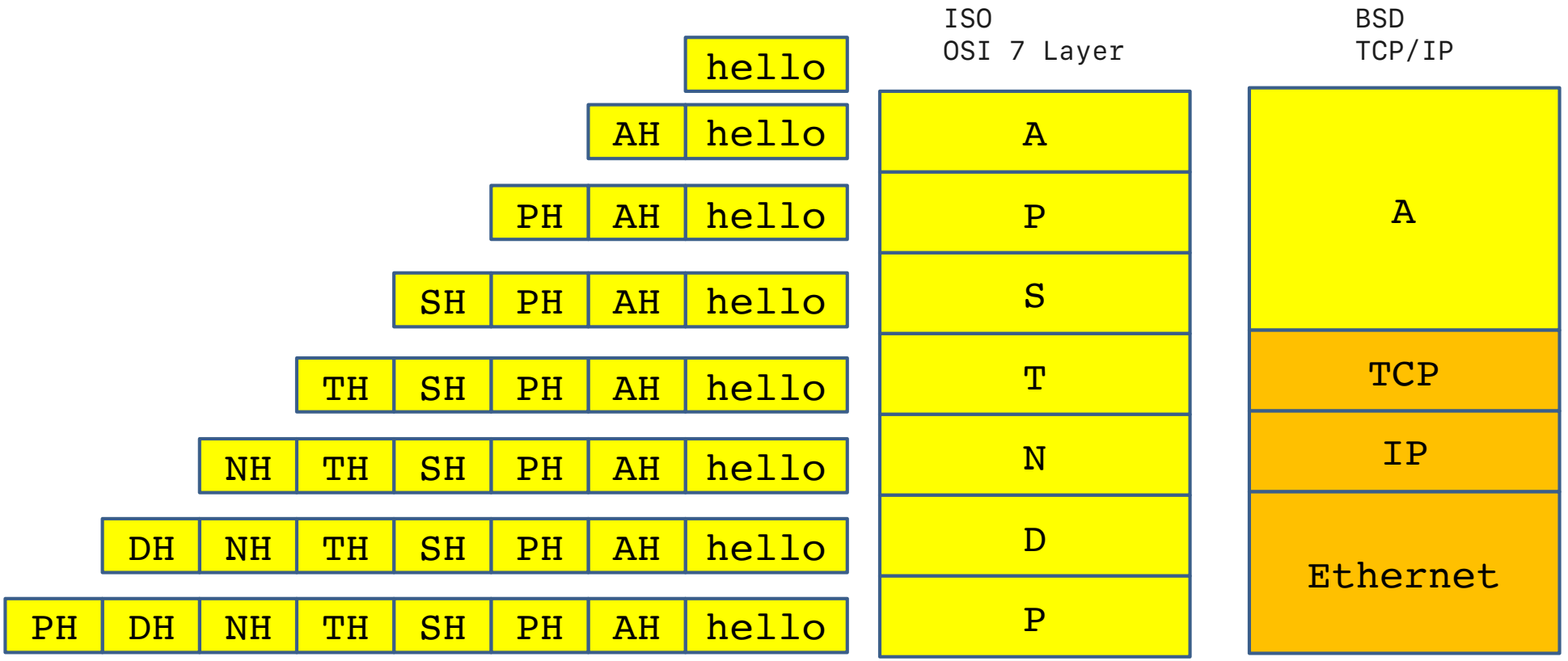
```
void setNonBlocking(int fd) {
  int flags = fcntl(fd, F_GETFL, 0);
  fcntl(fd, F_SETLF, flags | O_NONBLOCK);
}

int main() {
  int ssock = socket(PF_INET, SOCK_STREAM, 0);

  struct sockaddr_in saddr = {
      0,
  };
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(5000);
  saddr.sin_addr.s_addr = INADDR_ANY;

  int option = 1;
  setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR, &option,
sizeof option);
  if (bind(ssock, (struct sockaddr*)&saddr, sizeof
saddr) == -1) {
    perror("bind()");
    return -1;
  }

  if (listen(ssock, SOMAXCONN) == -1) {
    perror("listen()");
    return -1;
  }

  int efd = epoll_create1(0);
  if (efd == -1) {
    perror("epoll_create1()");
    return -1;
  }

  // server socket
  struct epoll_event event;
  event.events = EPOLLIN;
  event.data.fd = ssock;
  epoll_ctl(efd, EPOLL_CTL_ADD, ssock, &event);
```

```
struct epoll_event events[1024];
  int call_count = 0;
  while (true) {
    int count = epoll_wait(efd, events, 1024, -1);
    printf("epoll_wait: %d\n", ++call_count);

    for (int i = 0; i < count; ++i) {
      if (events[i].data.fd == ssock) {
        struct sockaddr_in caddr = {
            0,
        };
        socklen_t addrlen = sizeof caddr;
        int csock = accept(ssock, (struct
sockaddr*)&caddr, &addrlen);

        setNonBlocking(csock);
        event.events = EPOLLIN;
        event.data.fd = csock;
        epoll_ctl(efd, EPOLL_CTL_ADD, csock, &event);
        printf("connected from client...\n");
      } else {
        int csock = events[i].data.fd;

        char buf[1];
        int len = read(csock, buf, sizeof buf);
        if (len == -1 && errno == EAGAIN) {
          continue;
        } else if (len == 0 || len == -1) {
          printf("disconnected from client...\n");
          close(csock);

          epoll_ctl(efd, EPOLL_CTL_DEL, csock, NULL);
          continue;
        }

        // write(csock, buf, len);
      }
    }
  }

  close(ssock);
}
```

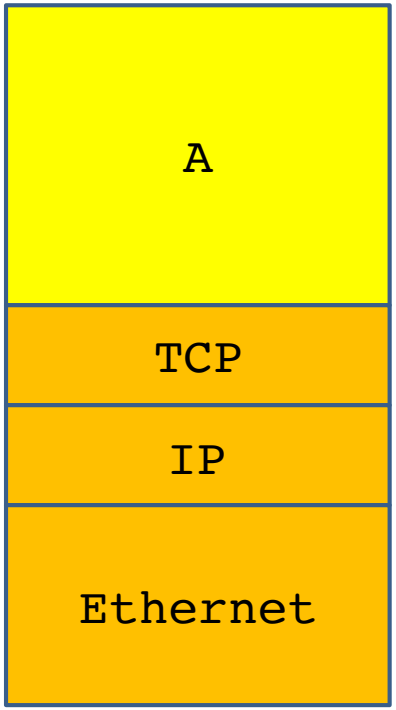
# #22. epoll

- 엣지 트리거는 결국 데이터를 수신하는 시점과 처리하는 시점을 분리하고자 할 때 사용한다.  하지만, 데이터의 수신이 완료되지 않았을 경우 read 함수에서 멈추는 경우가 발생할 수 있다.
  - ✓ read() 동작을 non-blocking 으로 변경해주어야 한다.

# 네트워크 개론


hello
AH hello
PH AH hello
SH PH AH hello
TH SH PH AH hello
NH TH SH PH AH hello
DH NH TH SH PH AH hello
PH DH NH TH SH PH AH hello
ISO
OSI 7 Layer
A
P
S
T
N
D
P
BSD
TCP/IP
A
TCP
IP
Ethernet

# BSD - TCP/IP



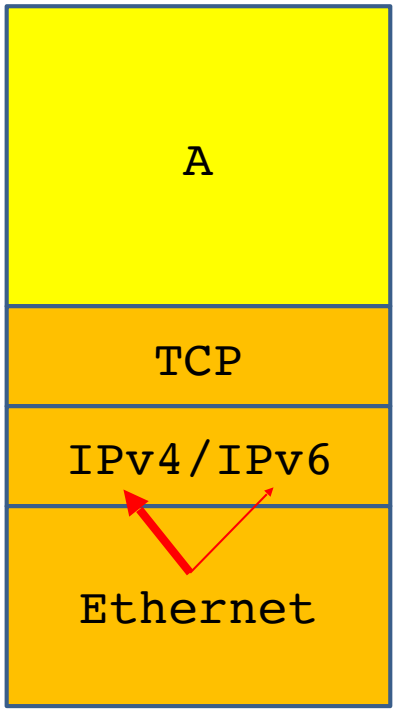

Ethernet

```
#define ETH_ALEN    6
#define ETH_HLEN    14

struct ethhdr {
    unsigned char   h_dest[ETH_ALEN];   /* destination eth addr */
    unsigned char   h_source[ETH_ALEN]; /* source ether addr    */
    __be16      h_proto;        /* packet type ID field */
} __attribute__((packed));
```

```
# ifconfig
HWaddr 08:00:27:1f:2d:6c
```

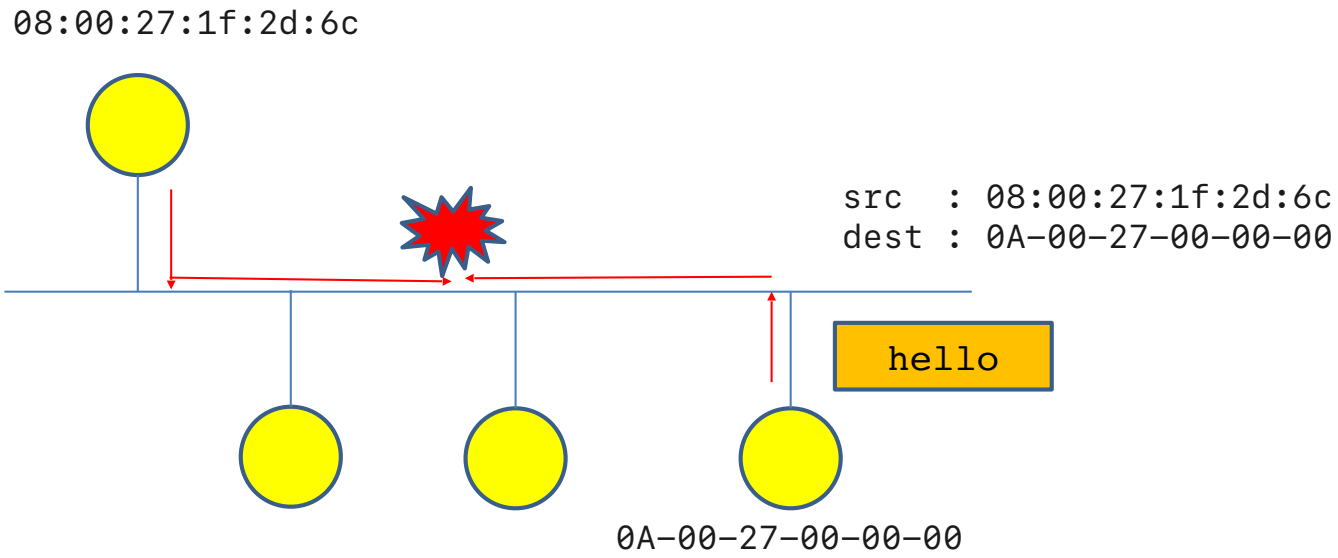
```
c:\>ipconfig /all

        0A-00-27   00-00-00
         회사명      고유번호
```

BSD
TCP/IP

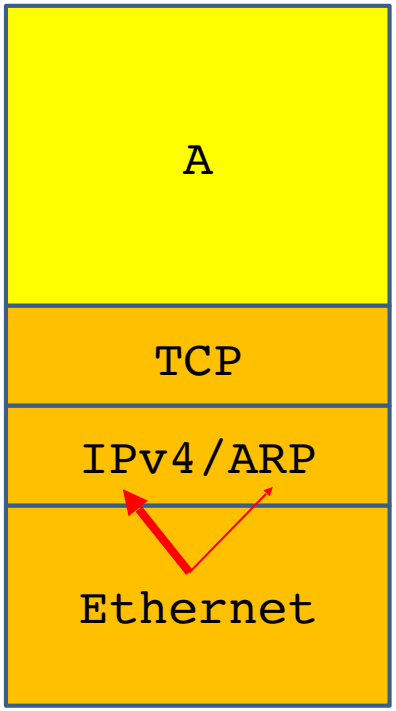



CSMA/CD
HWaddr 08:00:27:1f:2d:6c
c:\>ipconfig /all

0A-00-27 00-00-00
회사명 고유번호



BSD
TCP/IP



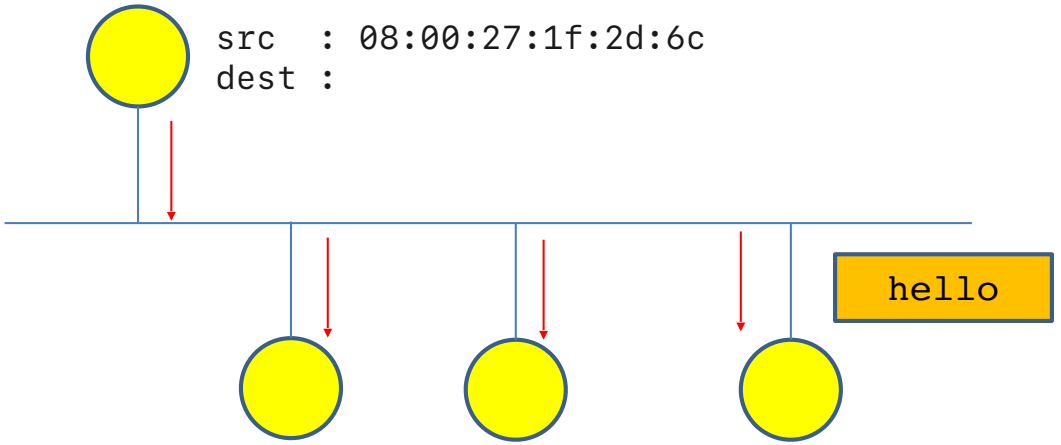

ARP
src_ip : 192.168.60.56
src : 08:00:27:1f:2d:6c
dst_ip : 192.168.60.1
dest :

Ethernet
dest : ff:ff:ff:ff:ff:ff
src : 08:00:27:1f:2d:6c

ARP
dst_ip : 192.168.60.56
dest : 08:00:27:1f:2d:6c
src_ip : 192.168.60.1
src : 0A-00-27-00-00-00

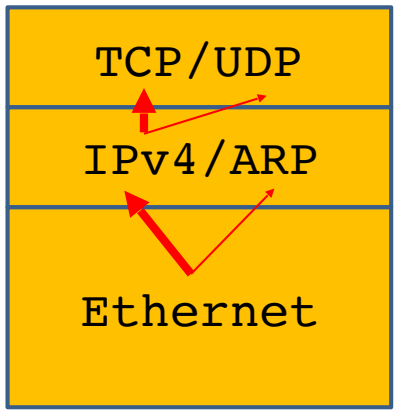
Ethernet
src : 0A-00-27-00-00-00
dest : 08:00:27:1f:2d:6c

08:00:27:1f:2d:6c
192.168.60.56

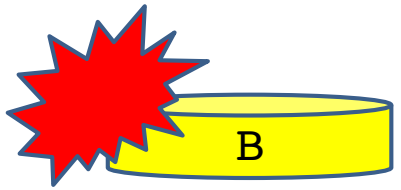



0A-00-27-00-00-00
192.168.60.1

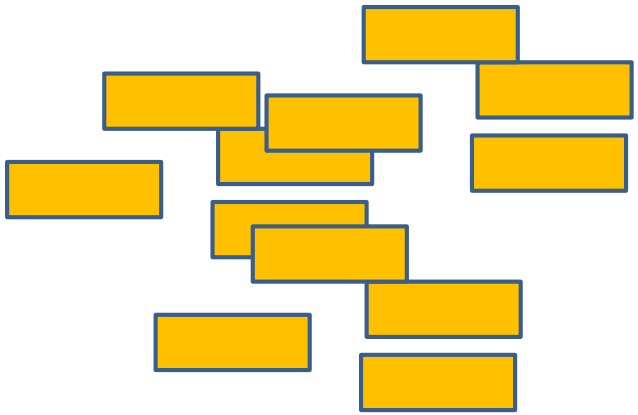
```
struct iphdr {
    __u8    ihl:4,          // 헤더의 길이(옵션 체크 때문에 존재) : 4byte정렬
        version:4;          // version 정보 : ipv4 => 4 , ipv6 => 6
    __u8    tos;            // 현재는 사용하지 않는다.
    __be16  tot_len;        // 패킷전체길이( 유효 데이터 길이 때문에 존재 ) : 60
    __be16  id;
    __be16  frag_off;
    __u8    ttl;            // 잘못된 경로의 패킷의 자동 파괴 기능 때문에 존재
    __u8    protocol;       // 6 : TCP,   17 : UDP
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
};
```








TTL=0 => ICMP

```
4500
04c0
c1cf
4000
3706
0000
0e00
4b89
c0a8
3c38
```

IP 헤더의 모든 값을 2byte 단위로 더한다.

```
     2D8FE
```

carry 발생시 carry를 한번 더 더한다.

```
D8FE
   2
D900
```

결과에 1의 보수를 취한다.

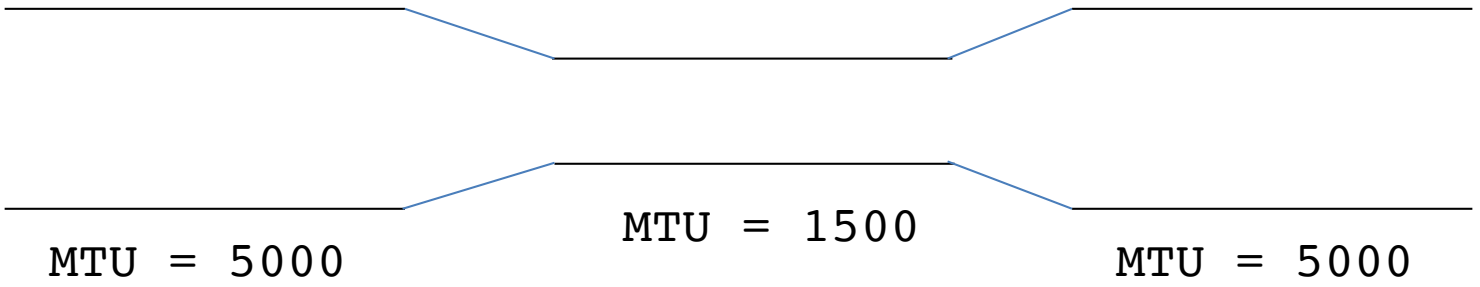
```
1101100100000000 ~
0010 0110 1111 1111
26ff
```

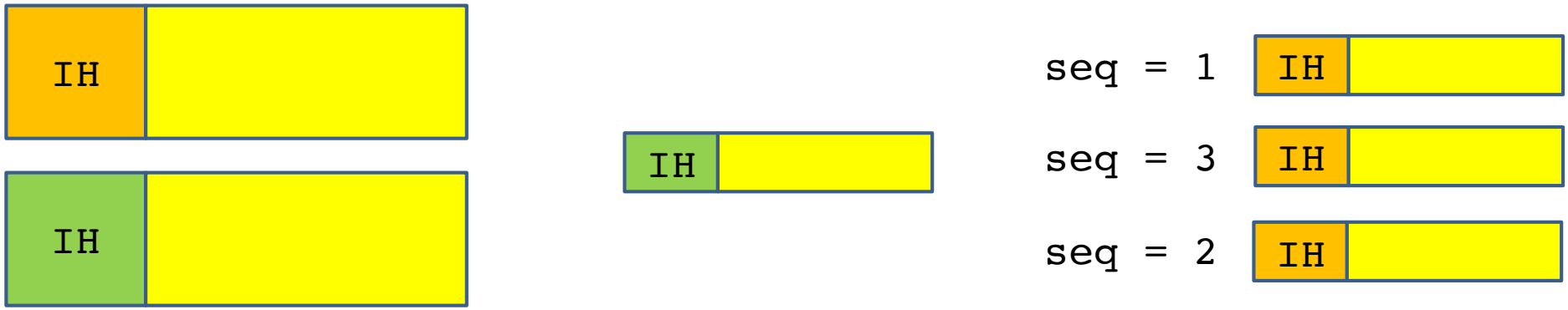
```
4500
04c0
c1cf
4000
3706
26ff
0e00
4b89
c0a8
3c38
```

IP 헤더의 모든 값을 2byte 단위로 더한다.

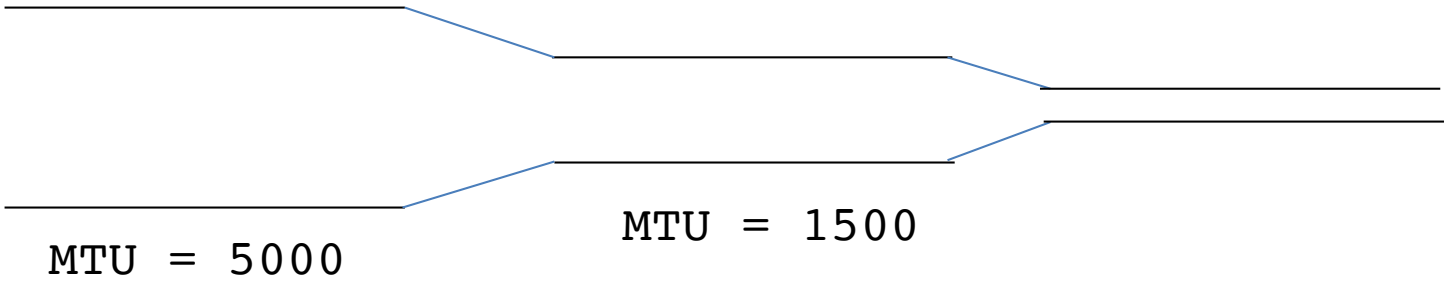
```
     2FFFD

carry 발생시 carry를 한번 더 더한다.
FFFD
   2
FFFF

결과에 1의 보수를 취한다.
0
```

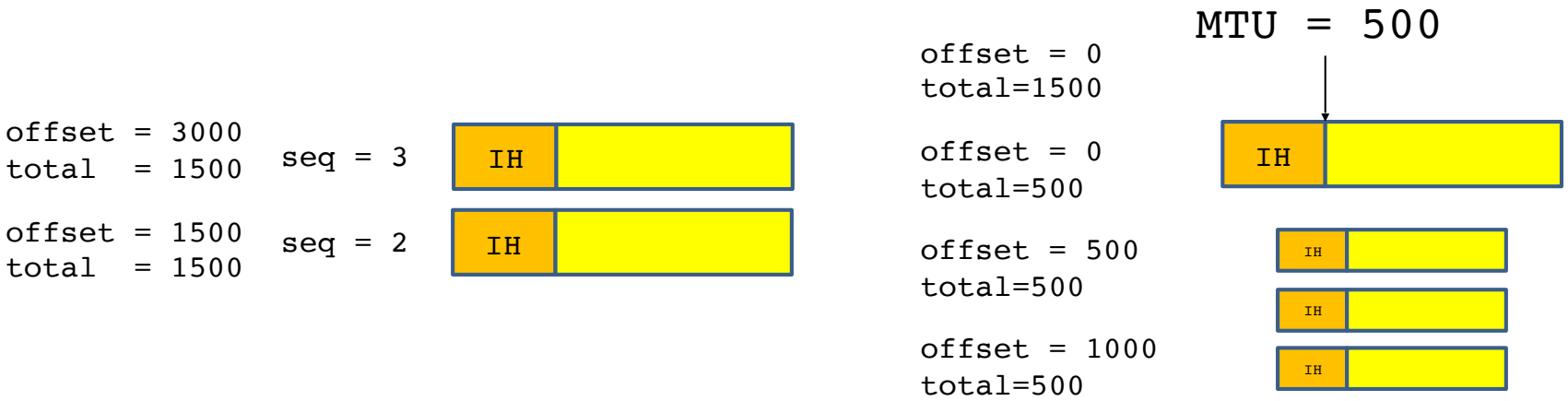
```
struct iphdr {
    __u8    ihl:4,          // 헤더의 길이( 옵션 체크 때문에 존재 ) : 4byte정렬
        version:4;          // version 정보 : ipv4 => 4 , ipv6 => 6
    __u8    tos;            // 현재는 사용하지 않는다.
    __be16  tot_len;        // 패킷전체길이( 유효 데이터 길이 때문에 존재 ) : 60
    __be16  id;             // 패킷의 고유번호 ( 조각난 패킷을 조립 할때 필요 )
    __be16  frag_off;
    __u8    ttl;            // 잘못된 경로의 패킷의 자동 파괴 기능 때문에 존재
    __u8    protocol;       // 6 : TCP,   17 : UDP
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
};
```



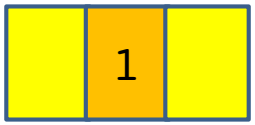



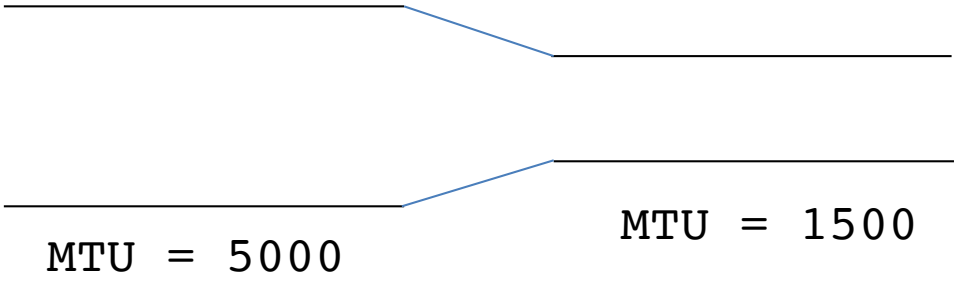
```
struct iphdr {
    __u8    ihl:4,          // 헤더의 길이( 옵션 체크 때문에 존재 ) : 4byte정렬
       version:4;           // version 정보 : ipv4 => 4 , ipv6 => 6
    __u8    tos;            // 현재는 사용하지 않는다.
    __be16  tot_len;        // 패킷전체길이( 유효 데이터 길이 때문에 존재 ) : 60
    __be16  id;             // 패킷의 고유번호 ( 조각난 패킷을 조립 할때 필요 )
    __be16  frag_off;       // 데이터의 떨어진 거리 ( 원본 데이터로 부터의 거리 )
                            // MF : 1 중간 패킷 MF=0 마지막 패킷
    __u8    ttl;            // 잘못된 경로의 패킷의 자동 파괴 기능 때문에 존재
    __u8    protocol;       // 6 : TCP,   17 : UDP
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
};
```



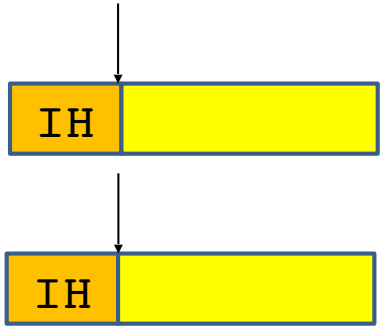



```
struct iphdr {
    __u8    ihl:4,          // 헤더의 길이( 옵션 체크 때문에 존재 ) : 4byte정렬
        version:4;          // version 정보 : ipv4 => 4 , ipv6 => 6
    __u8    tos;            // 현재는 사용하지 않는다.
    __be16  tot_len;        // 패킷전체길이( 유효 데이터 길이 때문에 존재 ) : 60
    __be16  id;             // 패킷의 고유번호 ( 조각난 패킷을 조립 할때 필요 )
    __be16  frag_off;       // 데이터의 떨어진 거리 ( 원본 데이터로 부터의 거리 )
                            // 단위가 : 8byte
                            // MF : 1 중간 패킷 MF=0 마지막 패킷
    __u8    ttl;            // 잘못된 경로의 패킷의 자동 파괴 기능 때문에 존재
    __u8    protocol;       // 6 : TCP,   17 : UDP
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
};
```

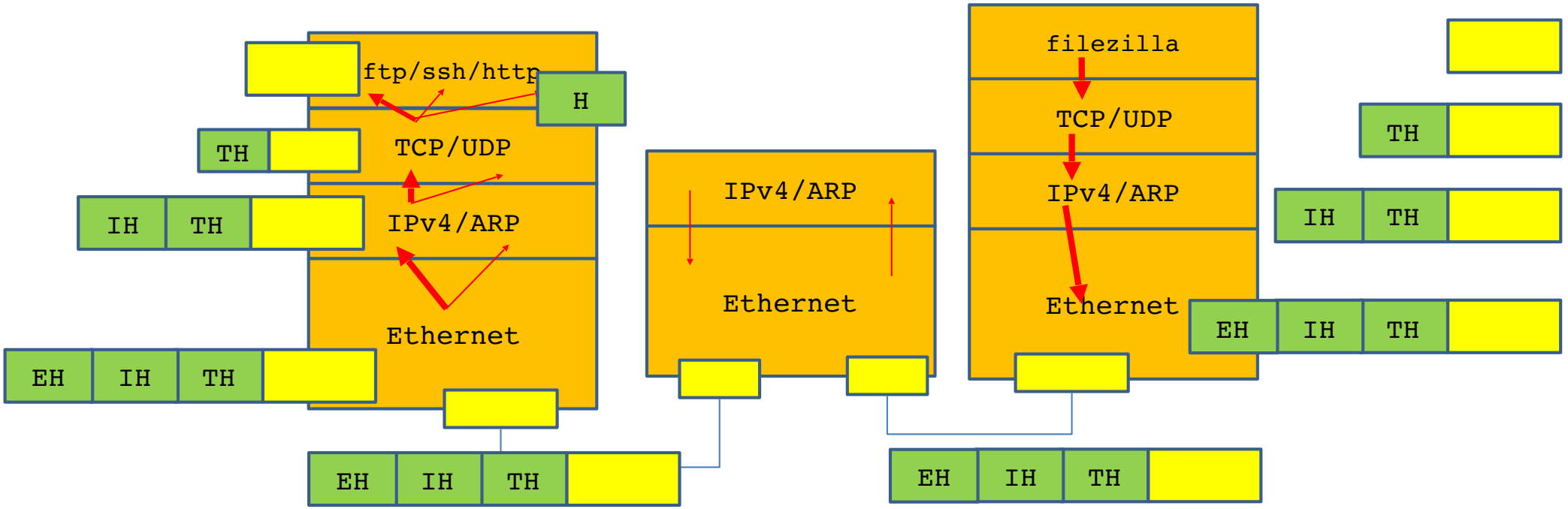


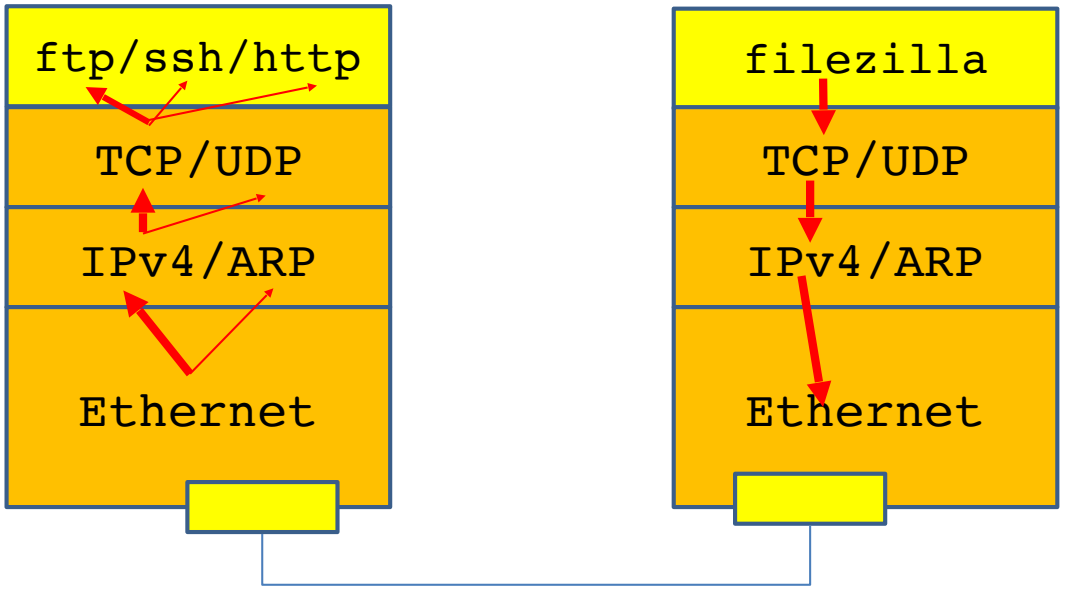

id = 0xf6b3
offset = 0
total = 1500
MF = 1



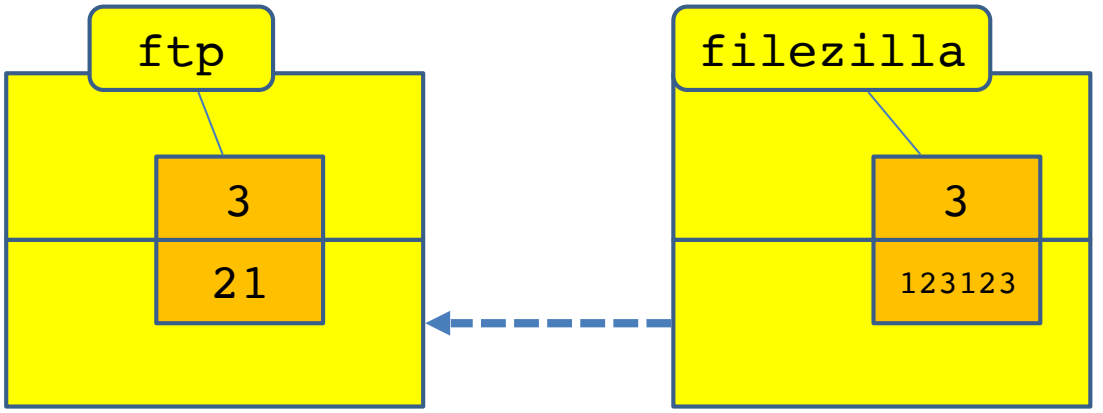

id = 0xf6b3
offset = 185
total = 548
MF = 0


ftp/ssh/http
H
TH
TCP/UDP
IH
TH
IPv4/ARP
Ethernet
EH
IH
TH
EH
IH
TH
IPv4/ARP
Ethernet
filezilla
TCP/UDP
IPv4/ARP
Ethernet
EH
IH
TH
TH
IH
TH
EH
IH
TH

```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```



```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```




```
int sd;
sd = socket();
read( sd, buff, size );
```

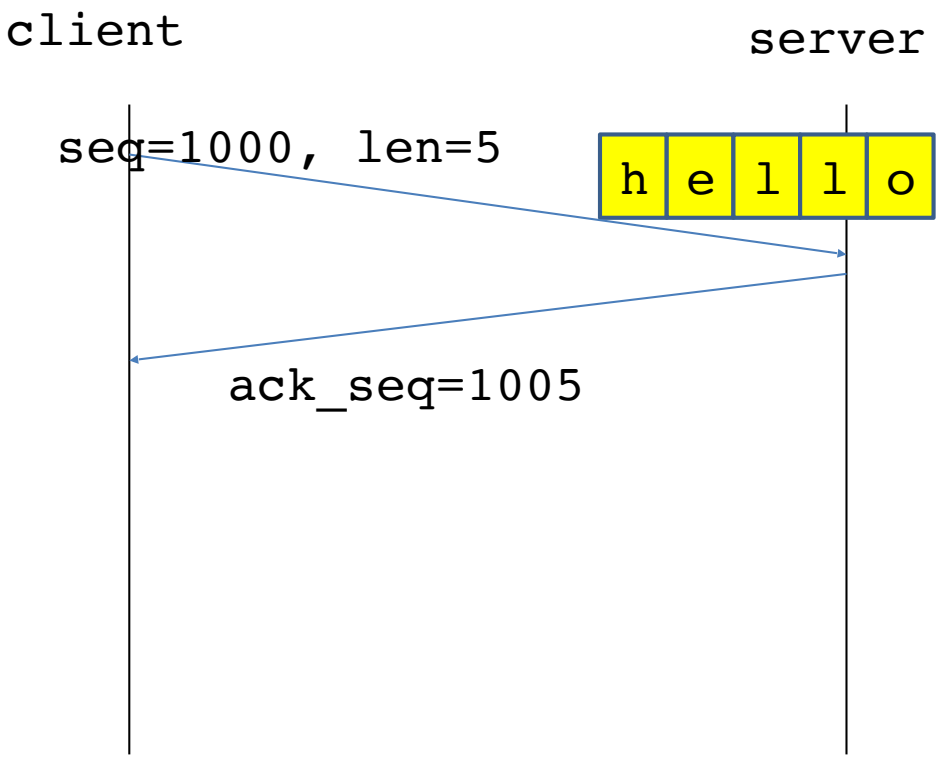
```
int sd;
sd = socket();
write(fd, buff, size );
```

```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,   // 버퍼링 금지
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```

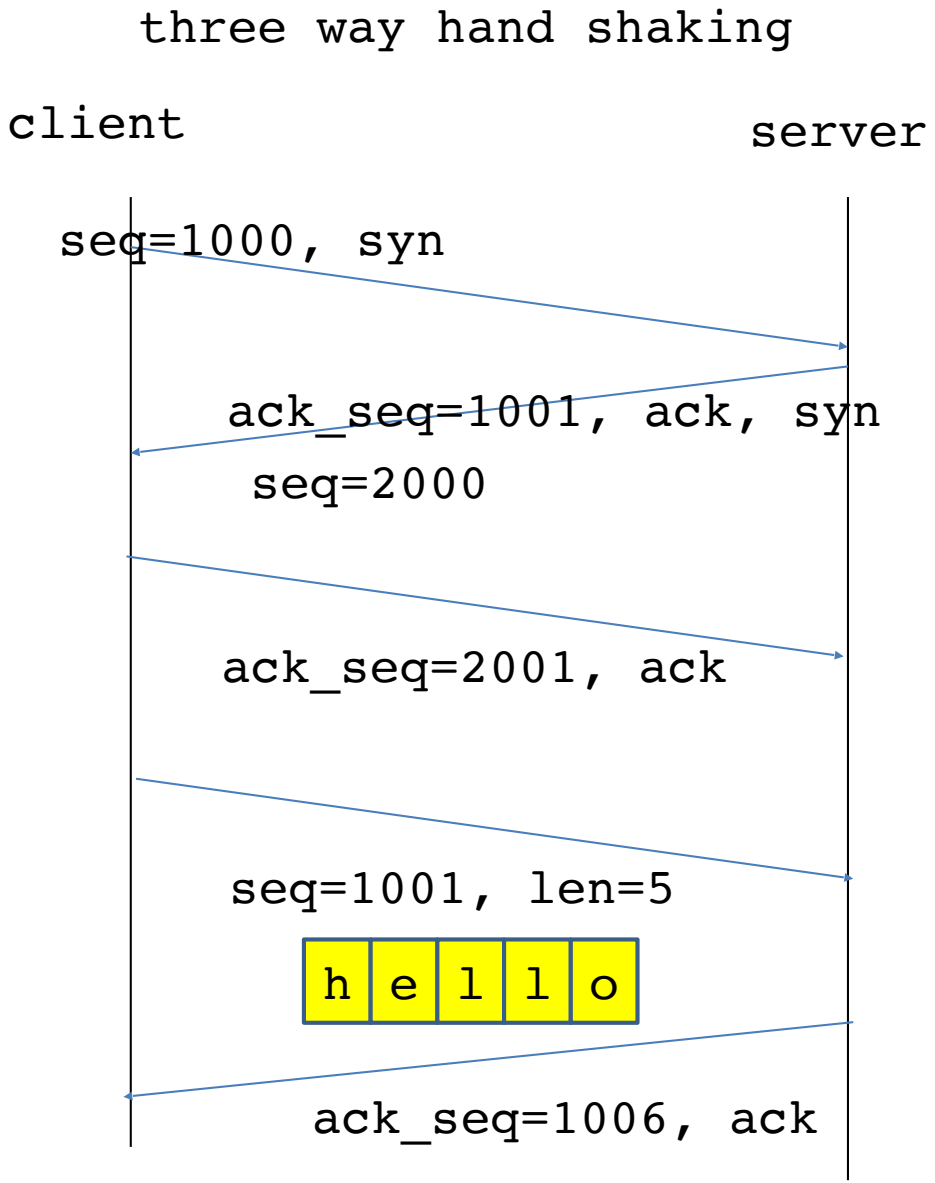
TCP 역할

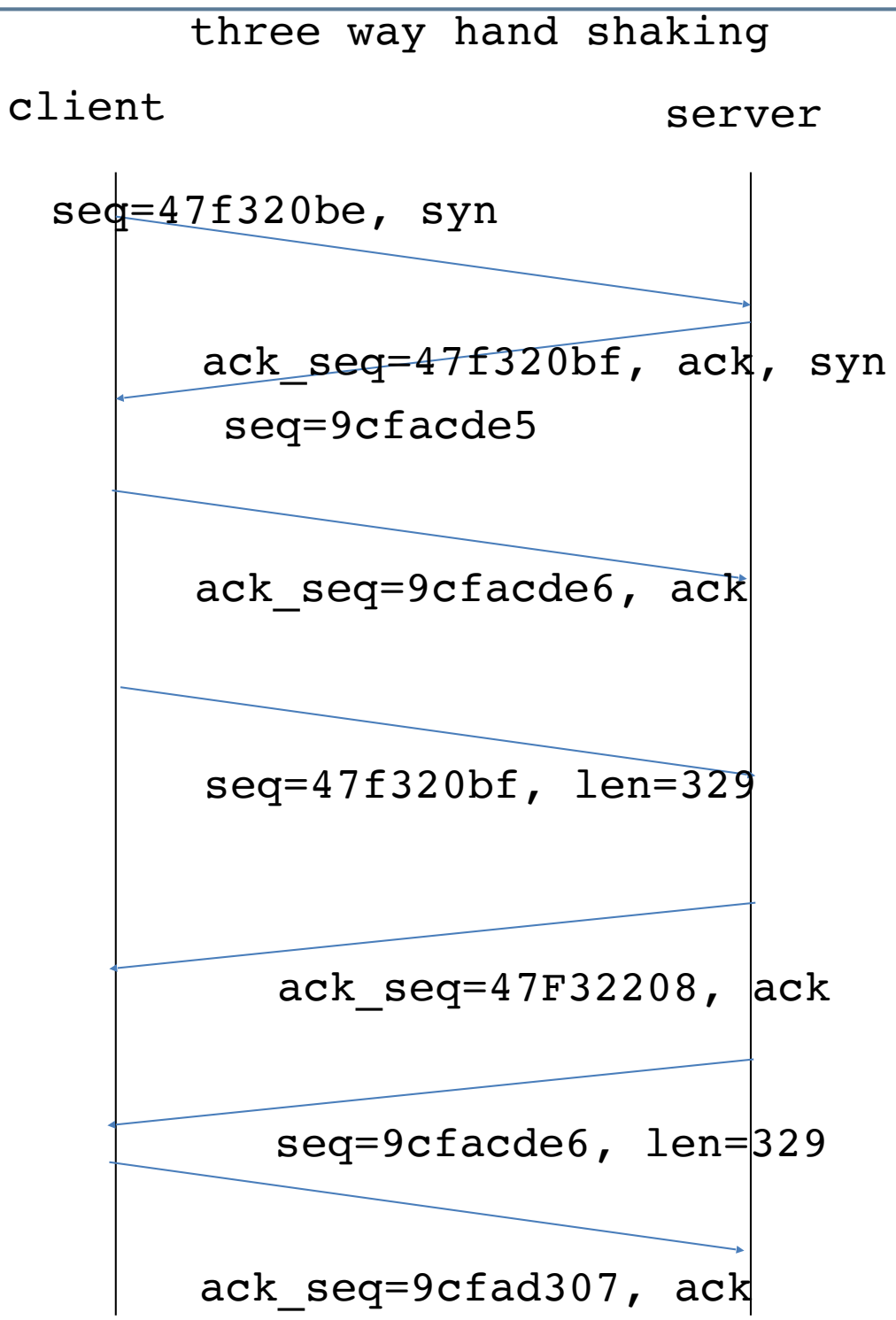
- 신뢰성 있는 전송
- 데이터 에러 처리
- 버퍼 overflow 방지
- 순서 제어

```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```



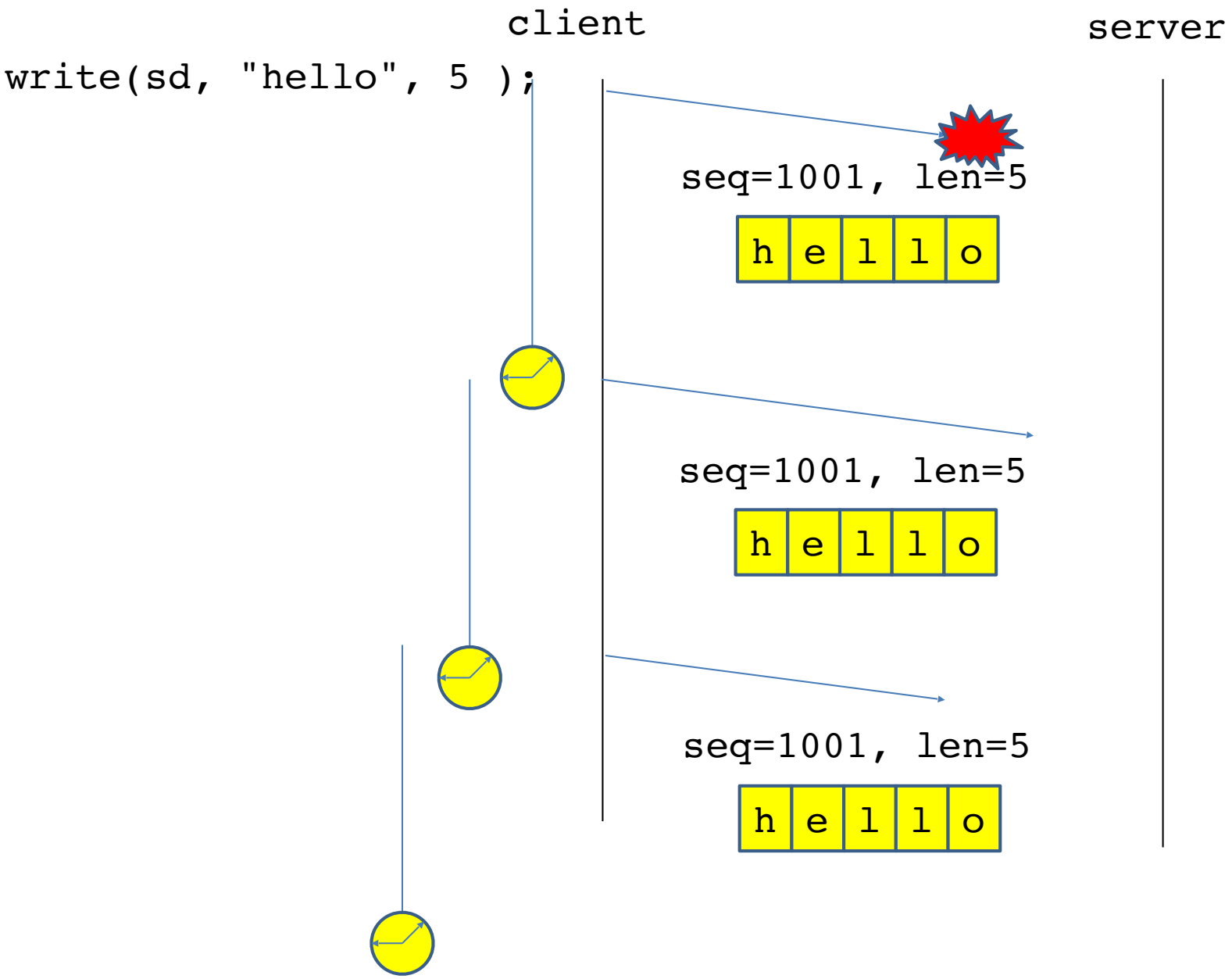
```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
        __u16   res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```



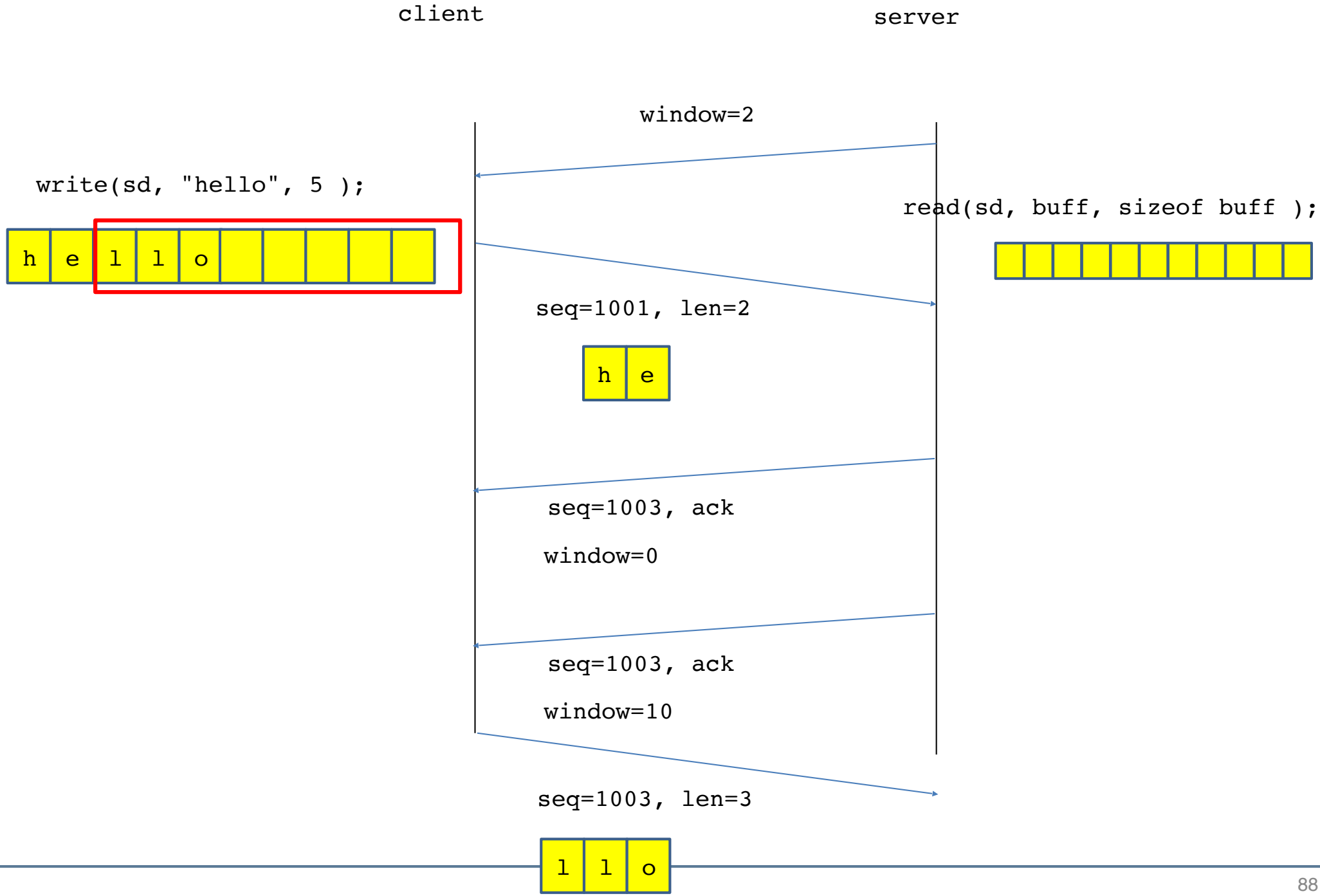

three way hand shaking
client
server
seq=47f320be, syn
ack_seq=47f320bf, ack, syn
seq=9cfacde5
ack_seq=9cfacde6, ack
seq=47f320bf, len=329
ack_seq=47F32208, ack
seq=9cfacde6, len=329
ack_seq=9cfad307, ack

TCP는 확인 응답 시스템이다.

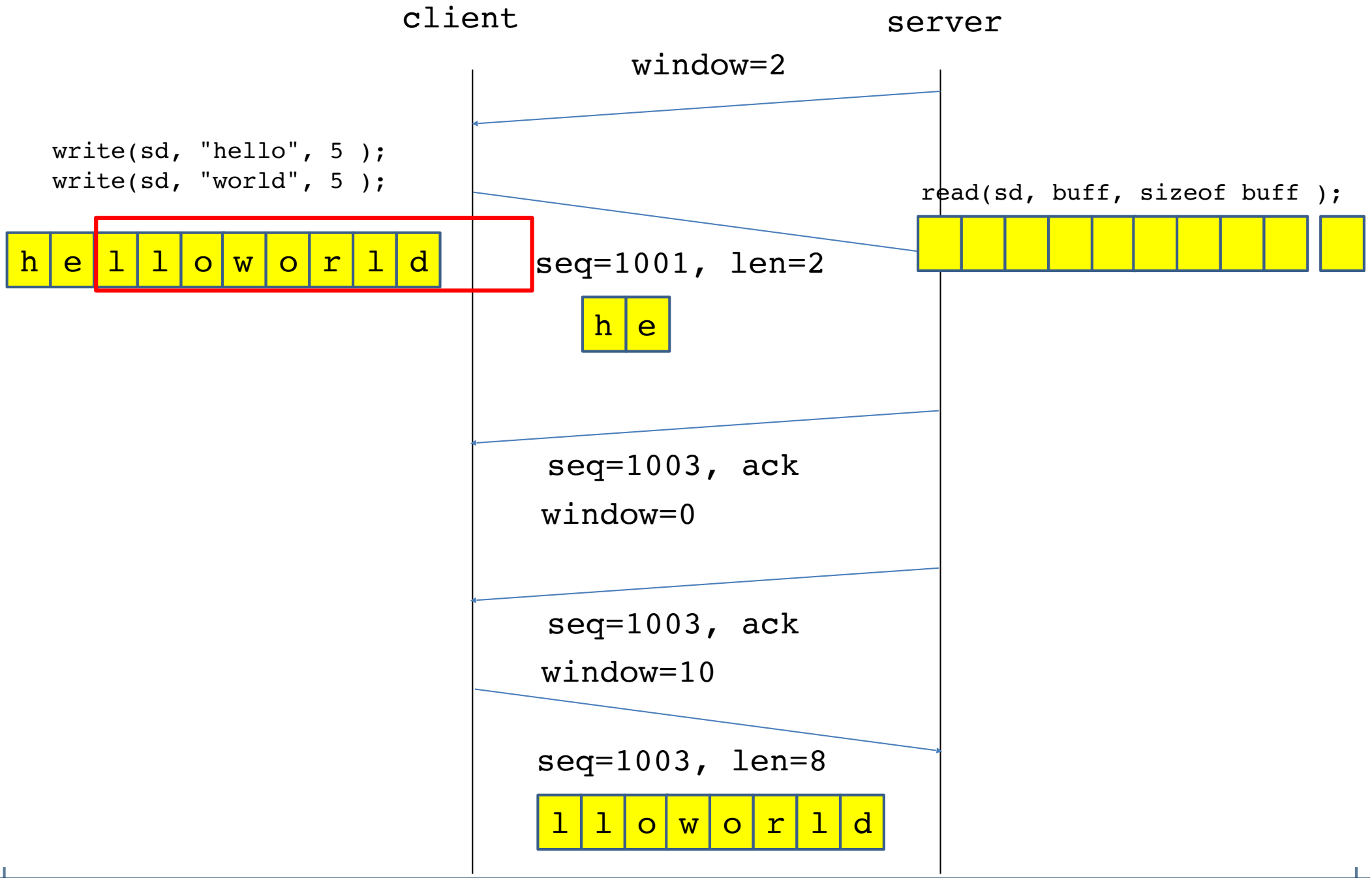
TCP는 재전송 시스템 이다. : 재전송 시간 = RTT * 2;



TCP는 확인 응답 시스템이다.
TCP는 재전송 시스템 이다. : 재전송 시간 = RTT * 2;

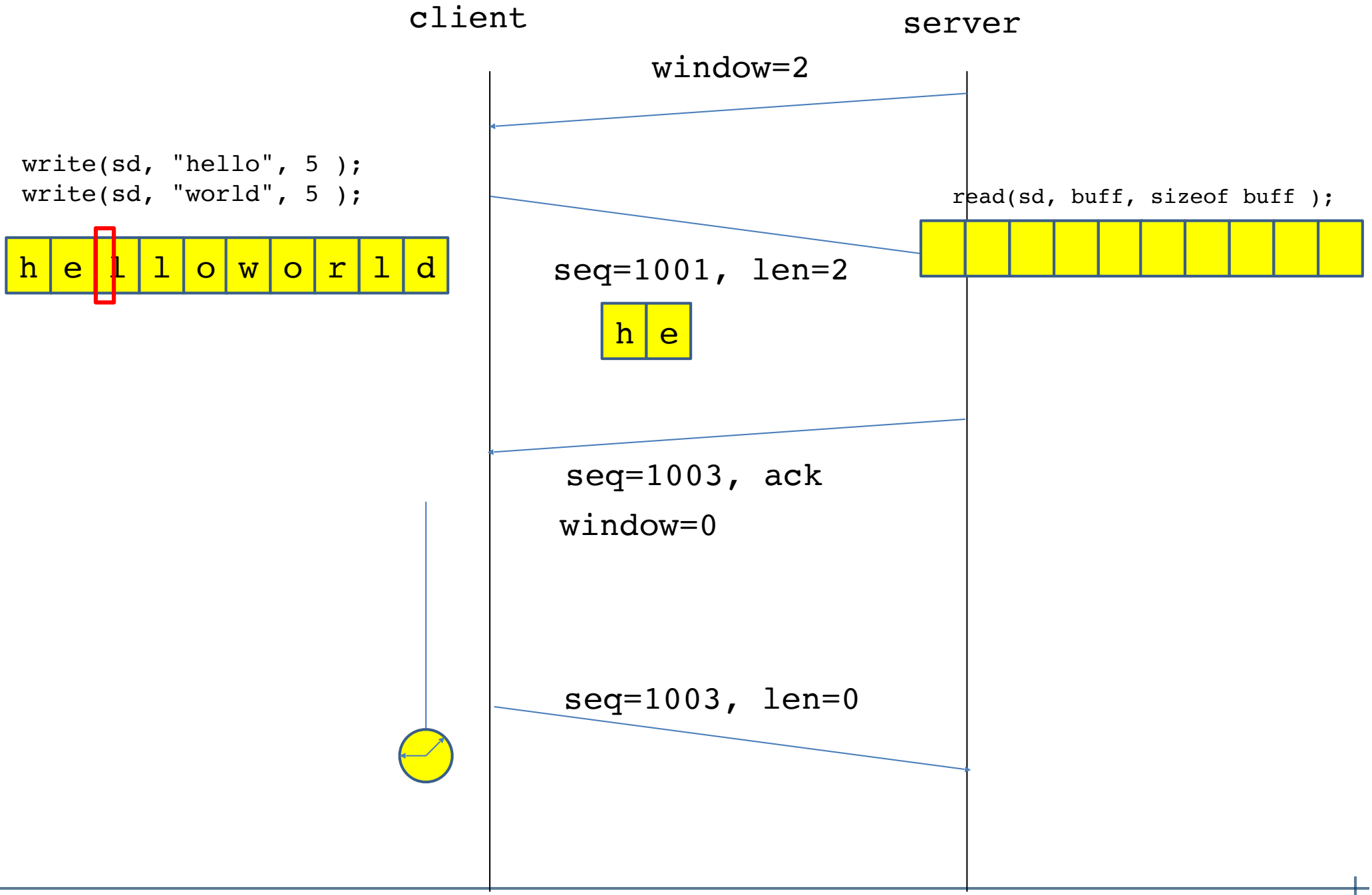


TCP는 확인 응답 시스템이다.
TCP는 재전송 시스템 이다. : 재전송 시간 = RTT * 2;

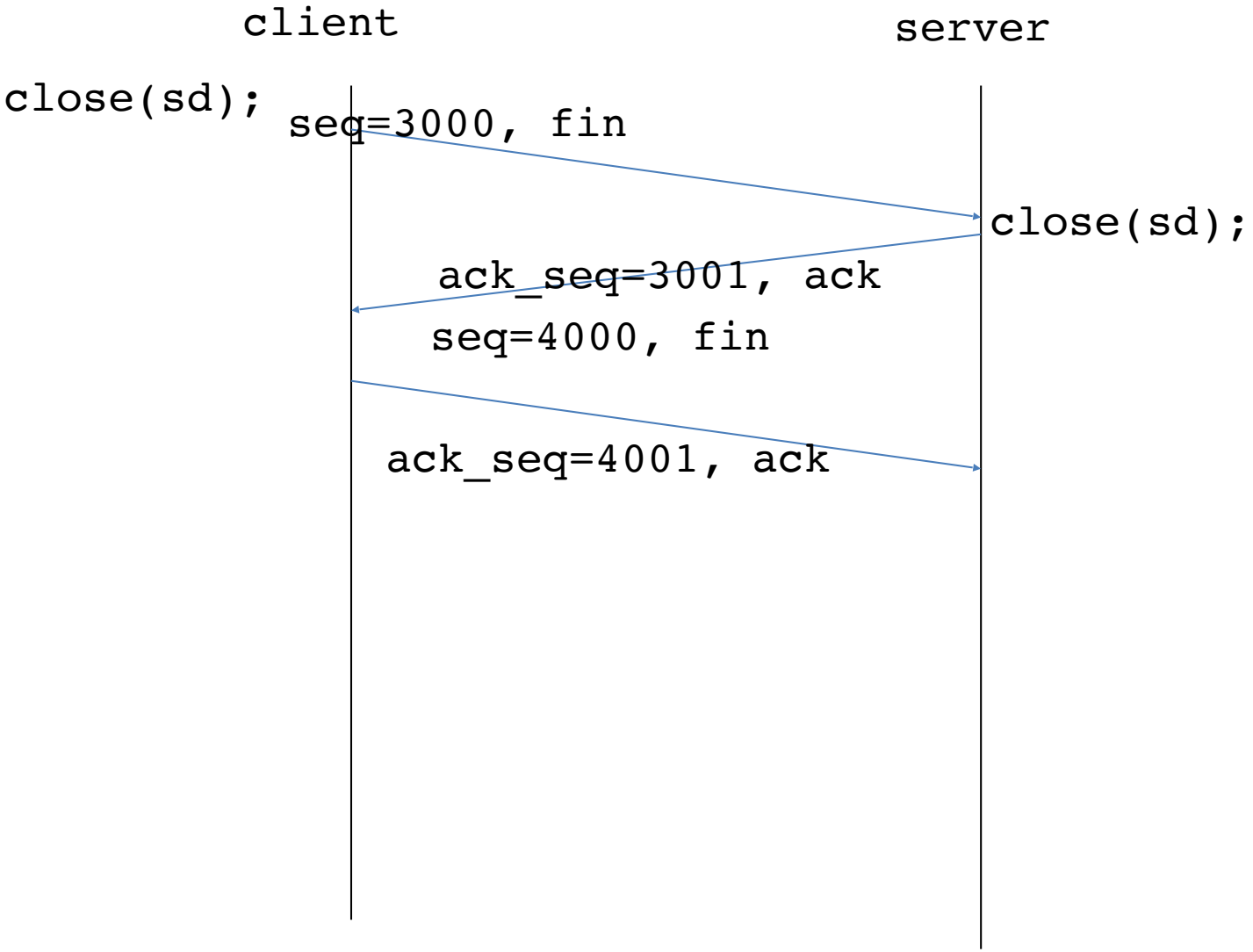


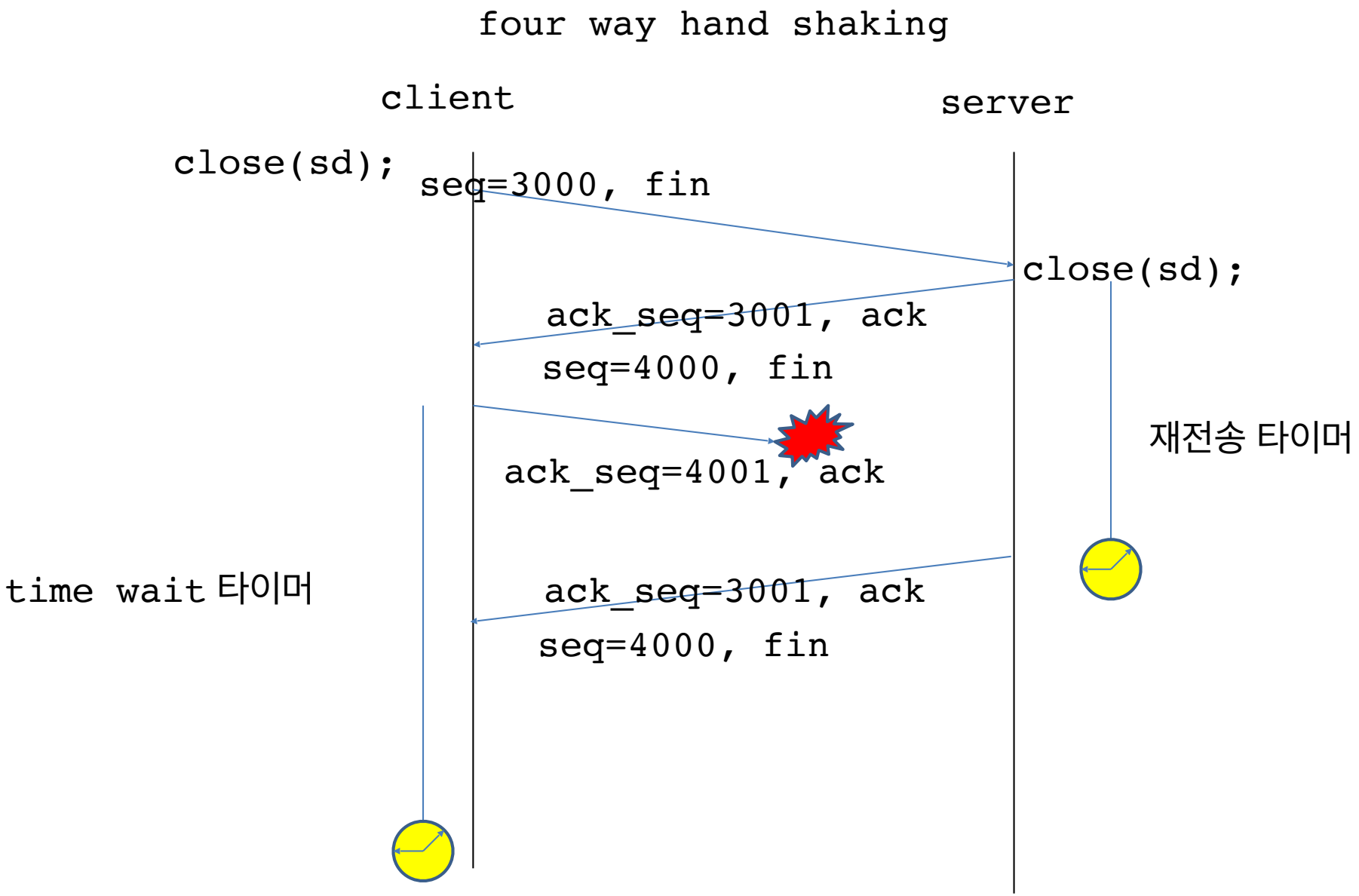
TCP는 확인 응답 시스템이다.

TCP는 재전송 시스템 이다. : 재전송 시간 = RTT * 2;

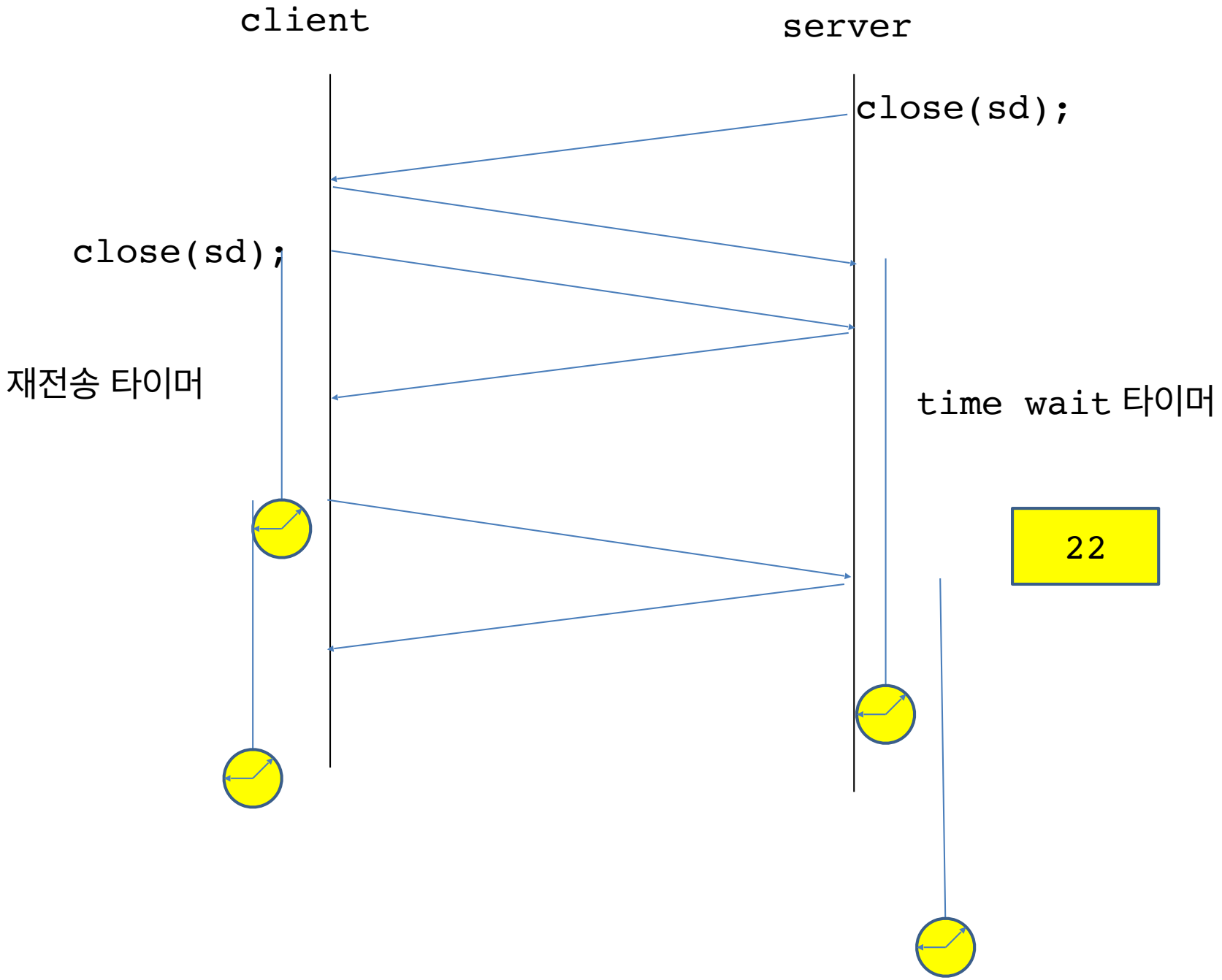


four way hand shaking




four way hand shaking
client
server
close(sd);
seq=3000, fin
close(sd);
ack_seq=3001, ack
seq=4000, fin
재전송 타이머
ack_seq=4001, ack
time wait 타이머
ack_seq=3001, ack
seq=4000, fin

# four way hand shaking



# Notes

# Notes

# Notes

# Notes

# Notes

# Notes

# Notes