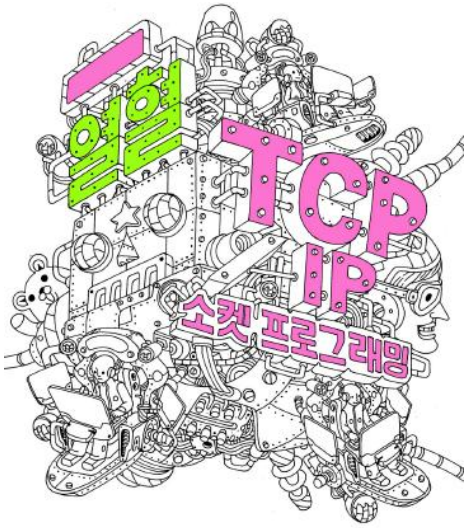
윤성우의 열혈 TCP/IP 소켓 프로그래밍

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

Chapter 17. select보다 낳은 epoll



Chapter 17-1. epoll의 이해와 활용

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

select 기반의 IO 멀티플렉싱이 느린 이유

select 함수의 단점

- select 함수호출 이후에 항상 등장하는, 모든 파일 디스크립터를 대상으로 하는 반복문
- select 함수를 호출할 때마다 인자로 매번 전달해야 하는 관찰대상에 대한 정보들

select 함수는 운영체제의 커널에 의해서 완성되는 기능이 아닌, 순수하게 함수에 의해 완성되는 기능이다. 따라서 select 함수의 호출을 통해서 전달된 정보는 운영체제에 등록되지 않는 것이며, 그래서 select 함수를 호출할 때마다 매번 관련 정보를 전달해야 한다. 그리고 이것이 select 함수가 지니는 단점의 가장 큰 원인이다.

단점의 해결책!

"운영체제에게 관찰대상에 대한 정보를 딱 한번만 알려주고서, 관찰대상의 범위, 또는 내용에 변경이 있을 때 변경 사항만 알려주도록 하자."

쉽게 말해서 select 함수의 단점 극복을 위해서는 운영체제 레벨에서 멀티플렉싱 기능을 지원해야 한다는 뜻이다. 그리고 리눅스의 epoll, 윈도우의 IOCP가 그러한 예에 해당한다.

select 이거 필요 없는 것인가?

select 방식이 사용되기 위한 조건!

- 서버의 접속자 수가 많지 않다.
- 다양한 운영체제에서 운영이 가능해야 한다.

운영체제 레벨이 아닌, 함수 레벨에서 완성되는 기능이다 보니, 호환성이 상대적으로 좋다. 즉, 리눅스의 epoll 기반 서버를 윈도우의 IOCP 기반으로 변경하는 것은 일이 될 수 있는데, 리눅스의 select 기반 서버를 윈도우의 select 기반 서버로 변경하는 것은 매우 간단하다.

절대 우위를 점하는 서버의 모델은 없다! 따라서 상황에 맞게 적절한 모델을 선택할 수 있어야 한다.

epoll 방식의 장점

select와 비교한 epoll의 장점

- 상태변화의 확인을 위한, 전체 파일 디스크립터를 대상으로 하는 반복문이 필요 없다.
- select 함수에 대응하는 epoll_wait 함수호출 시, 관찰대상의 정보를 매번 전달할 필요가 없다.

상태변화 관찰에 더 낳은 방법을 제공한다. 그리고 커널에서 상태정보를 유지하기 때문에 관찰대상의 정보를 매번 전달하지 않아도 된다.

위의 장점은 다른 운영체제에서 제공하는 모든 멀티플렉싱 서버의 장점이기도 하다.

epoll의 구현에 필요한 함수와 구조체

- epoll_create epoll 파일 디스크립터 저장소 생성
- epoll_ctl 저장소에 파일 디스크립터 등록 및 삭제
- epoll_wait select 함수와 마찬가지로 파일 디스크립터의 변화를 대기한다.

```
struct epoll_event
{
    __uint32_t events;
    epoll_data_t data;
}

    typedef union epoll_data
    {
        void *ptr;
        int fd;
        __uint32_t u32;
        __uint64_t u64;
    } epoll_data_t;
```

위의 세 함수 호출을 통해서 epoll의 기능이 완성된다.

위의 세 함수와 왼쪽의 구조체가 어떻게 사용되는지 이해하면, epoll을 이해하는 셈이 된다.

왼쪽의 구조체는 소켓 디스크립터의 등록 및 이벤트 발생의 확인에 사용되는 구조체이다.

epoll_create

```
#include <sys/epoll.h>

int epoll_create(int size);
```

➡ 성공 시 epoll 파일 디스크립터, 실패 시 -1 반환

- size : epoll 인스턴스의 크기정보.

운영체제가 관리하는, **epoll 인스턴스**라 불리는 파일 디스크립터의 저장소를 생성!
소멸 시 close 함수호출을 통한 종료의 과정이 필요하다.

위의 함수호출을 통해서 생성된 epll 인스턴스에 관찰대상을 저장 및 삭제하는 함수가 **epoll_ctl**이고, epoll 인스턴스에 등록된 파일 디스크립터를 대상으로 이벤트의 발생 유무를 확인하는 함수가 **epoll_wait**이다.

epoll_ctl

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

➜ 성공 시 0, 실패 시 -1 반환

- epfd 관찰대상을 등록할 epoll 인스턴스의 파일 디스크립터.
- op 관찰대상의 추가, 삭제 또는 변경여부 지정.
- fd 등록할 관찰대상의 파일 디스크립터.
- event 관찰대상의 관찰 이벤트 유형.

두 번째 전달인자에 따라 등록, 삭제 및 변경이 이뤄진다.

두 번째 전달인자

- EPOLL_CTL_ADD 파일 디스크립터를 epoll 인스턴스에 등록한다.
- EPOLL_CTL_DEL 파일 디스크립터를 epoll 인스턴스에서 삭제한다.
- EPOLL_CTL_MOD 등록된 파일 디스크립터의 이벤트 발생상황을 변경한다.

```
epoll_ctl(A, EPOLL_CTL_ADD, B, C);
```

epoll 인스턴스 A에, 파일 디스크립터 B를 등록하되, C를 통해 전달된 이벤트의 관찰을 목적으로 등록을 진행한다.”

```
epoll_ctl(A, EPOLL_CTL_DEL, B, NULL);
```

epoll 인스턴스 A에서 파일 디스크립터 B를 삭제한다.

epoll_ctl 함수 기반의 디스크립터 등록

epoll_event 구조체는 이벤트의 유형 등록에 사용된다.
그리고 이벤트 발생시 발생된 이벤트의 정보로도 사용된다.

```
struct epoll_event event;
. . . . .
event.events=EPOLLIN;
event.data.fd=sockfd;
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &event);
. . . . .
```

- EPOLLIN — 수신할 데이터가 존재하는 상황
- EPOLLOUT — 출력버퍼가 비워져서 당장 데이터를 전송할 수 있는 상황
- EPOLLPRI — OOB 데이터가 수신된 상황
- EPOLLRDHUP — 연결이 종료되거나 Half-close가 진행된 상황, 이는 엣지 트리거 방식에서 유용하게 사용될 수 있다.
- EPOLLERR — 에러가 발생한 상황
- EPOLLET — 이벤트의 감지를 엣지 트리거 방식으로 동작시킨다.
- EPOLLONESHOT — 이벤트가 한번 감지되면, 해당 파일 디스크립터에서는 더 이상 이벤트를 발생시키지 않는다. 따라서 epoll_ctl 함수의 두 번째 인자로 EPOLL_CTL_MOD을 전달해서 이벤트를 재설정해야 한다.

비트 OR 연산을 통해 둘 이상을 함께 등록할 수 있다.

epoll_wait

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

➜ 성공 시 이벤트가 발생한 파일 디스크립터의 수, 실패 시 -1 반환

- epfd — 이벤트 발생의 관찰영역인 epoll 인스턴스의 파일 디스크립터.
- events — 이벤트가 발생한 파일 디스크립터가 채워질 버퍼의 주소 값.
- maxevents — 두 번째 인자로 전달된 주소 값의 버퍼에 등록 가능한 최대 이벤트 수.
- timeout — 1/1000초 단위의 대기시간, -1 전달 시, 이벤트가 발생할 때까지 무한 대기.

```
int event_cnt;
struct epoll_event *ep_events;
. . . . .
ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);
. . . . .
event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
. . . . .
```

epoll_wait 함수 반환 후, 이벤트 발생한 파일 디스크립터의 수가 반환되고, 두 번째 인자로 전달된 주소의 메모리 공간에 이벤트 발생한 파일 디스크립터 별도로 묶인다.

epoll_wait 함수의 두 번째 인자를 통해서 이벤트 발생한 디스크립터가 별도로 묶이므로, 전체 파일 디스크립터 대상의 반복문은 불필요 하다.

epoll 기반의 에코 서버1

리스닝 소켓의 등록과 연결요청의 대기.

```
epfd=epoll_create(EPOLL_SIZE);
ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

event.events=EPOLLIN;
event.data.fd=serv_sock;
epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);

while(1)
{
    event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    if(event_cnt==-1)
    {
        puts("epoll_wait() error");
        break;
    }

    for(i=0; i<event_cnt; i++)
    {
       . . . .
    }
}
```

epoll 서버의 기본 모델이다. 리스닝 소켓으로 전달되는 연결요청도 수신된 데이터의 일종이므로 리스닝 소켓을 epoll 인스턴스에 등록하되, EPOLLIN을 대상으로 등록한다.

epoll 기반의 에코 서버2

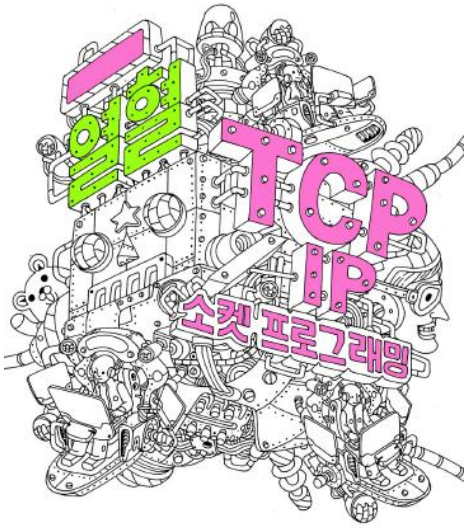
epll_wait 함수호출 이후의 반복문

```
for(i=0; i<event_cnt; i++)
{
    if(ep_events[i].data.fd==serv_sock)
    {
        adr_sz=sizeof(clnt_adr);
        clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
        event.events=EPOLLIN;
        event.data.fd=clnt_sock;
        epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
        printf("connected client: %d \n", clnt_sock);
    }
    else
    {
        str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
        if(str_len==0)  // close request!
        {
            epoll_ctl(
                epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
            close(ep_events[i].data.fd);
            printf("closed client: %d \n", ep_events[i].data.fd);
        }
        else
        {
            write(ep_events[i].data.fd, buf, str_len);  // echo!
        }
    }
}
```

연결요청의 경우, 수락 및 디스크립터 등록의 과정을 거친다.

종료요청의 경우, 디스크립터 해제의 과정을 거친다.

메시지 수신의 경우 에코 처리한다.



Chapter 17-2. 레벨 트리거와 엣지 트리거

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

레벨 트리거와 엣지 트리거의 차이

- 아들 엄마 세뱃돈으로 5,000원 받았어요.
- 엄마 아주 훌륭하구나!
- 아들 엄마 옆집 숙희가 떡볶이 사달래서 사줬더니 2,000원 남았어요.
- 엄마 장하다 우리아들~
- 아들 엄마 변신가면 샀더니 500원 남았어요.
- 엄마 그래 용돈 다 쓰면 굶으면 된다!
- 아들 엄마 여전히 500원 갖고 있어요. 굶을 순 없잖아요.
- 엄마 그래 매우 현명하구나!
- 아들 엄마 여전히 500원 갖고 있어요. 끝까지 지켜야지요.
- 엄마 그래 힘내거라!

레벨 트리거 방식은 입력 버퍼에 데이터가 남아있는 동안에 계속해서 이벤트를 발생시킨다. 데이터 양의 변화에 상관없이!

레벨 트리거와 엣지 트리거의 차이는 이벤트의 발생 방식에 있다.

- 아들 엄마 세뱃돈으로 5,000원 받았어요.
- 엄마 음 다음엔 더 노력하거라.
- 아들 . . . . . .
- 엄마 말 좀 해라! 그 돈 어쨌냐? 계속 말 안 할거냐?

엣지 트리거 방식은 입력 버퍼에 데이터가 들어오는 순간 딱 한번만 이벤트를 발생시킨다.

레벨 트리거의 이벤트 특성 파악하기

예제 echo_EPLTserv.c의 일부

```
#define BUF_SIZE 4
#define EPOLL_SIZE 50
```

버퍼의 크기를 4바이트로 줄여서 수신된 메시지를 한번에 읽어 들이지 못하도록 하였다.

소켓은 기본적으로 레벨 트리거로 동작한다.

```
while(1)
{
    event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    if(event_cnt==-1)
    {
        puts("epoll_wait() error");
        break;
    }

    puts("return epoll_wait");
    for(i=0; i<event_cnt; i++)
    {
        if(ep_events[i].data.fd==serv_sock)
        {
```

이벤트가 발생해서 epoll_wait 함수가 반환할 때마다 문자열이 출력되도록 하였다.

```
root@my_linux:/tcpip# gcc echo_EPLTserv.c -o serv
root@my_linux:/tcpip# ./serv 9190
return epoll_wait
connected client: 5
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
connected client: 6
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
return epoll_wait
closed client: 5
return epoll_wait
closed client: 6
```

실행결과는 버퍼에 입력데이터가 남아있는 상황에서 이벤트가 발생함을 보이고 있다.

엣지 트리거 기반의 서버 구현을 위해 필요한 것

1. 넌-블로킹 IO로 소켓 속성 변경

```
int flag=fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flag|O_NONBLOCK);
```

fcntl 함수호출을 통해서 소켓의 기본 설정정보를 얻은 다음, 거기에 O_NONBLOCK 속성을 더해서 소켓의 특성을 재설정한다.

엣지 트리거는 데이터 수신 시 딱 한번만 이벤트가 발생하기 때문에 이벤트가 발생했을 때 충분한 양의 버퍼를 마련한 다음에 모든 데이터를 다 읽어 들여야 한다. 즉, 데이터의 분량에 따라서 IO로 인한 DELAY가 생길 수 있다. 그래서 엣지 트리거에서는 넌-블로킹 IO를 이용한다. 입력 함수의 호출과 다른 작업을 병행할 수 있기 때문이다.

2. 입력버퍼의 상태확인

```
int errno;
```

넌-블로킹 IO 기반으로 데이터 입력 시 데이터 수신이 완료되었는지 별도로 확인해야 한다. 헤더파일 <error.h>를 포함하고 변수 errno을 참조한다. errno에 EAGAIN이 저장되면 버퍼가 빈 상태이다.

엣지 트리거 기반의 echo 서버1

예제 echo_EPETserv.c의 일부

```
#define BUF_SIZE 4
#define EPOLL_SIZE 50
```

여전히 버퍼의 크기를 4바이트로 줄여서 수신된 메시지를 한번에 읽어 들이지 못하도록 하였다.

```
epfd=epoll_create(EPOLL_SIZE);
ep_events=malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

setnonblockingmode(serv_sock);
event.events=EPOLLIN;
event.data.fd=serv_sock;
epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);

while(1)
{
    event_cnt=epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
```

리스닝 소켓도 비동기 IO를 진행하도록 옵션을 설정하고 있다.

```
void setnonblockingmode(int fd)
{
    int flag=fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flag|O_NONBLOCK);
}
```

연결요청을 수락해서 생성된 소켓에 대해서도 비동기 IO의 옵션을 설정해야 한다.

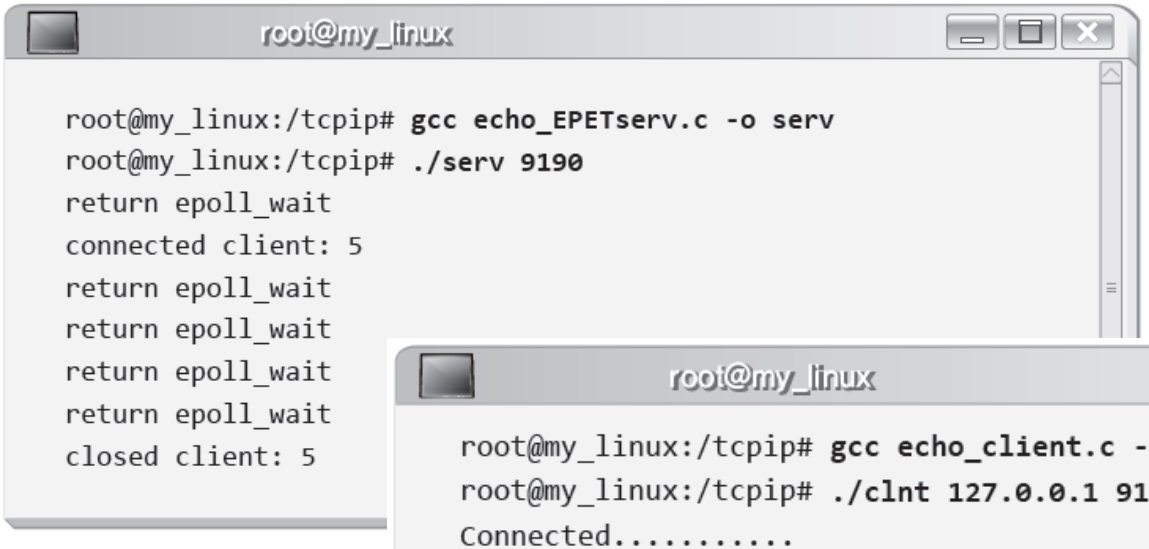
엣지 트리거 기반의 echo 서버2

예제 echo_EPETserv.c의 일부(else... 수신할 데이터가 있는 경우)

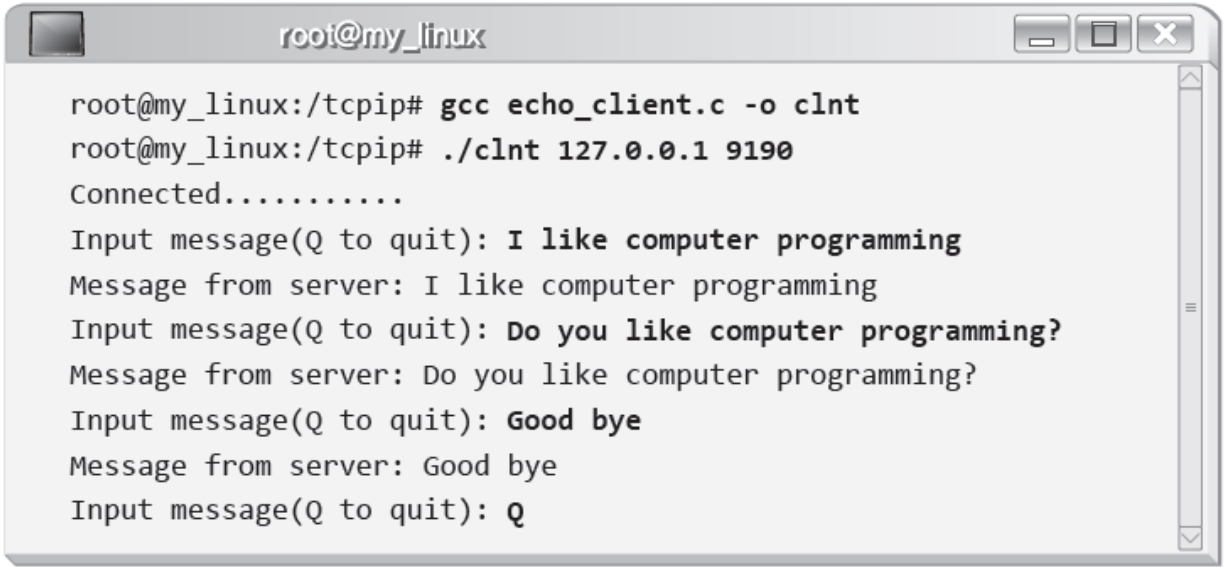
```
else
{
    while(1)
    {
        str_len=read(ep_events[i].data.fd, buf, BUF_SIZE);
        if(str_len==0) {    // close request!
            epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
            close(ep_events[i].data.fd);
            printf("closed client: %d \n", ep_events[i].data.fd);
            break;
        }
        else if(str_len<0) {
            if(errno==EAGAIN)
                break;
        }
        else {
            write(ep_events[i].data.fd, buf, str_len); // echo!
        }
    }
}
```

errno가 EAGAIN일 때까지, 즉 버퍼가 완전히 비워질 때까지 반복문을 계속 돌며 데이터를 수신하고 있다.

엣지 트리거 기반의 echo 서버의 실행결과



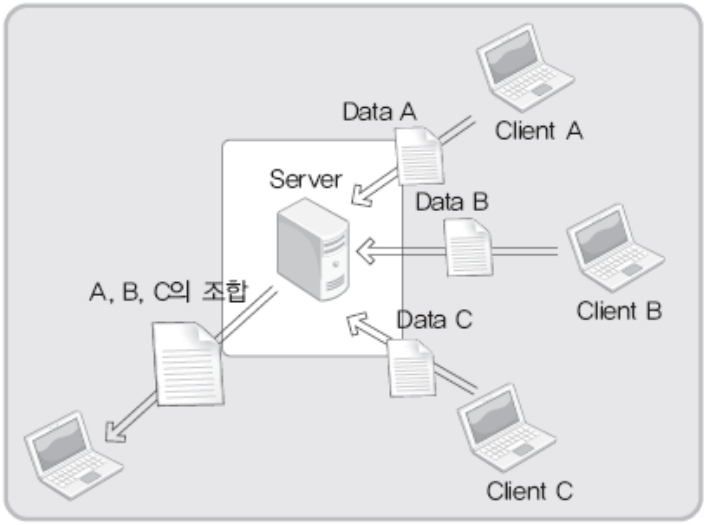
왼쪽은 서버의 실행결과이다. 클라이언트가 종료될 때까지 총 4회의 이벤트가 발생했음을 알 수 있다.



오른쪽은 클라이언트의 실행결과이다. 총 4회의 메시지를 전달했음을 알 수 있다.

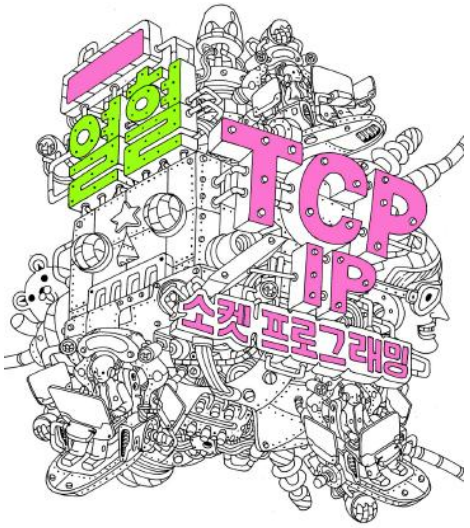
이렇듯, 엣지 트리거 기반에서는 데이터의 송수신횟수와 이벤트의 발생수가 일치한다.

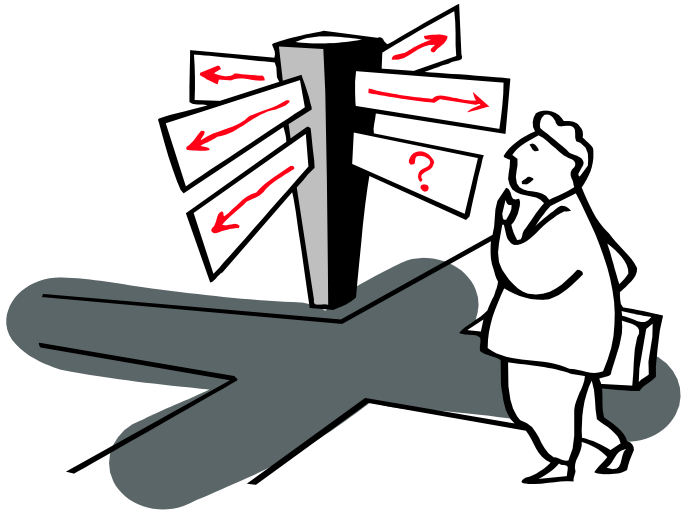
엣지 트리거와 레벨 트리거의 비교



- 서버는 클라이언트 A, B, C로부터 각각 데이터를 수신한다.
- 서버는 수신한 데이터를 A, B, C의 순으로 조합한다.
- 조합한 데이터는 임의의 호스트에게 전달한다.

위와 같은 시나리오 상에서 클라이언트가 서버에 접속 및 데이터를 전송하는 순서는 서버의 기대와 상관이 없다. 이처럼 서버측에서의 컨트롤 요소가 많은 경우에는 엣지 트리거가 유리하다. 반면, 서버의 역할이 상대적으로 단순하고 또 데이터 송수신의 상황이 다양하지 않다면, 레벨 트리거 방식을 선택할만하다.





Chapter 17이 끝났습니다. 질문 있으신지요?