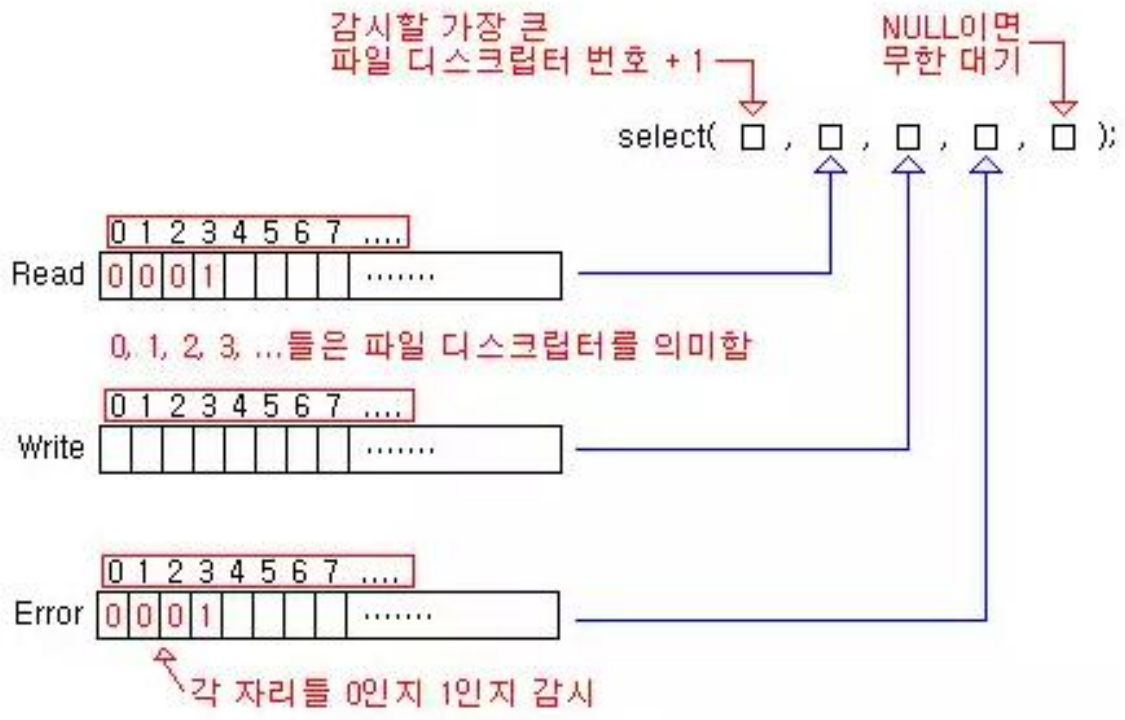
epoll을 사용한 비동기 프로그래밍

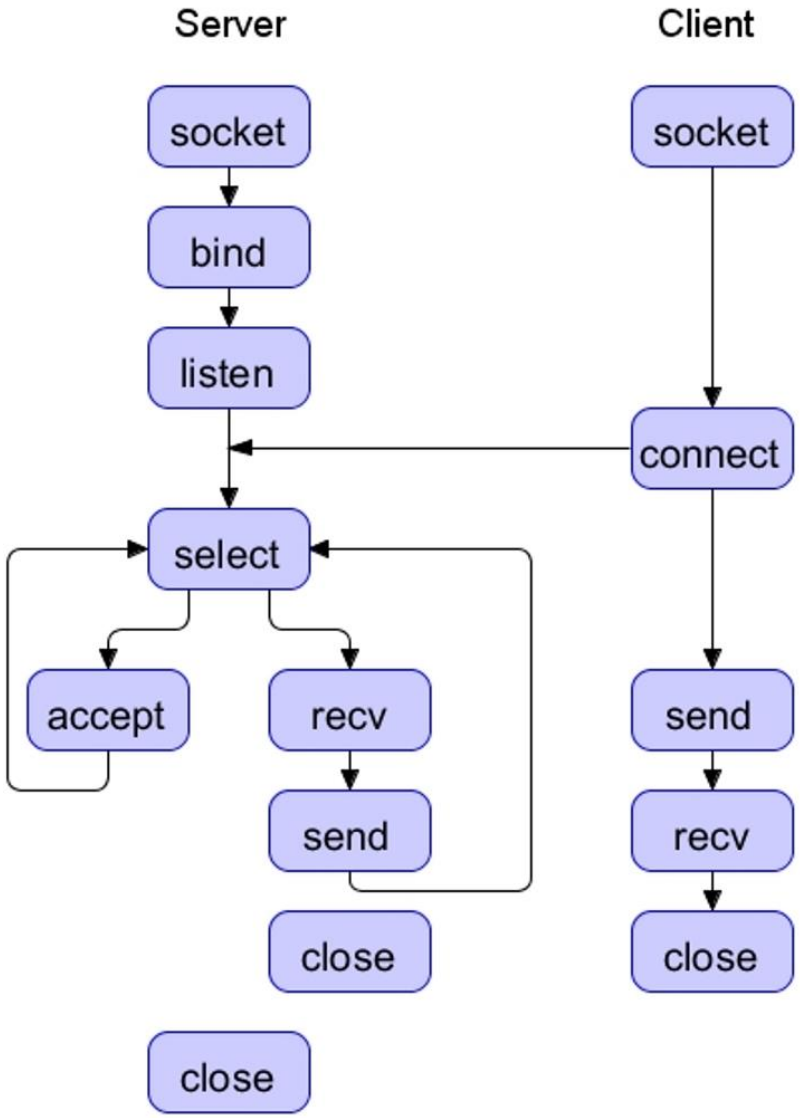
최흥배
https://github.com/jacking75/choiHeungbae

처음에는 직접 만들려고 했는데.....
구글링 해보니 이미 좋은 글들이 꽤 있더군요...
굿이 제가 또 같은 것을 만드는 것은 시간 낭비 같더군요..
그래서 그 자료들을 잘 정리해서 전달하기로 했어요..
요즘 말로 좋게 말하면 큐레이션(?)^^;;

select - 가장 기본적인...



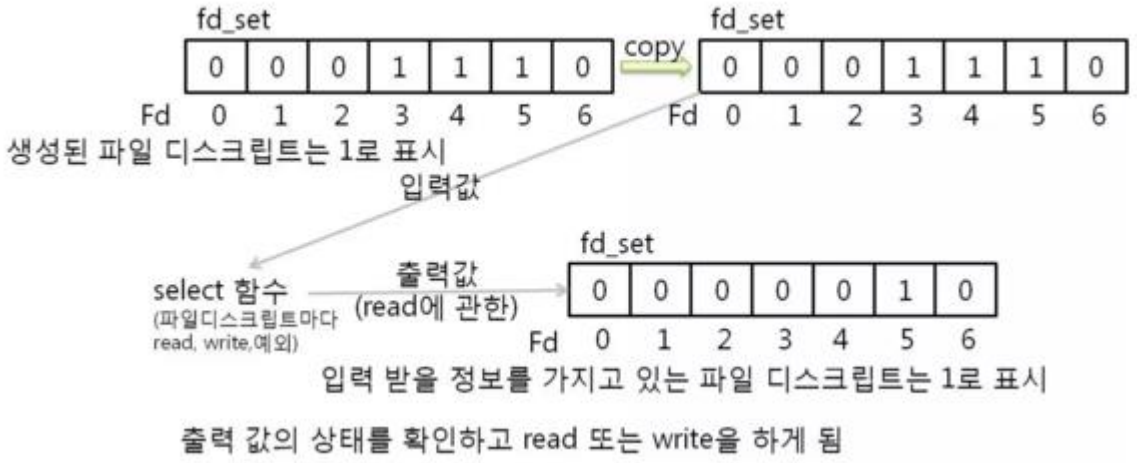
http://cfile10.uf.tistory.com/image/2737CE3653AB79E002307A



http://liuj.fcu.edu.tw/net_pg/figures/concurrent_1p.jpg

그러나 select는....

- 고정 비트 테이블인 fd_set을 사용하는데 크기가 고정(bitmask. 1024)
  poll로 해결할 수는 있지만...
- 이벤트가 발생을 감지하기 위해서는 순차검색
  fd를 처음부터 끝까지 하나씩 조사한다. O(n)
  poll도 순차검색
- 데이터가 오면 기존 fd_set을 모두 변경(복사 비용)
- 대용량 네트워크에서는 사용하기 힘들다



http://comfun.tistory.com/entry/12%EC%9E%A5-IO%EB%A9%80%ED%8B%B0%ED%94%8C%EB%A0%89%EC%8B%B1


c10k
"10 thousand clients" problem

epoll

- 파일 디스크립트(fd) 수가 무제한
- 파일 디스크립트를 커널에서 관리하므로 상태가 바뀐 것만을 직접 통지
  O(1)
  fd_set 복사가 필요 없다
- 성능은 select < poll < epoll

epoll 3대장

- epoll_create
  만들자

- epoll_ctl
  어떤 것의 어떤 상태를 알고 싶다

- epoll_wait
  상태에 변화가 있을 때까지 기다리자

epoll : IOCP

epoll	IOCP
epoll_create	CreateIoCompletionPort
epoll_ctl	CreateIoCompletionPort
epoll_wait	GetQueuedCompletionStatus

epoll_create

int epoll_create(int size);

epoll_create는 size 만큼의 커널 폴링 공간을 만드는 함수이다. 리턴 값은 그냥 정수 값인데 fd_epoll이라고 부르기로 하자. 이 fd_epoll를 이용해서 앞으로 다른 조작들을 하게 된다.

```
int fd_epoll;
bool is_epoll_init = false;

int EpollInit(int size)
{
	if((fd_epoll = epoll_create(size)) > 0) is_epoll_init = true;
	return fd_epoll;
}
```

- size 값은 정수인데 무작정 큰 수를 쓸 수는 없다.
- 예상되는 최대 동시접속 수로 한다.
- 운영체제가 이 숫자를 허용하는지 먼저 확인해야 한다.
- 서버의 한계(ServerLimits)를 숙지하고, 적당한 값을 써 주거나 서버한계를 늘려야 한다.

출처: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

epoll_ctl

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

epoll_ctl은 epoll이 관심을 가져주길 바라는 fd와 그 fd에서 발생하는 관심 있는 사건의 종류를 등록하는 인터페이스.

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

struct epoll_event {
    __uint32_t events;  /* Epoll events */
    epoll_data_t data;  /* User data variable */
};
```

공용체 하나와 32비트 정수를 가지는 평범한 구조체.
사건은 (epoll_event).events를 이용한다

```
int EpollAdd(const int fd)
{
        struct epoll_event ev;

        ev.events  = EPOLLIN | EPOLLOUT | EPOLLERR;
        ev.data.fd = fd;

        return epoll_ctl(fd_epoll, EPOLL_CTL_ADD, fd, &ev);
}
```

보통 서버에서는 입출력 시점에 관심이 많기 때문에 위 코드에서는 IN/OUT/ERR 세가지 이벤트를 설정하도록 했다.

출처: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

op 인자에 추가되는 flag

EPOLL_CTL_ADD	fd 를 epfd의 관심 목록에 추가, 이미 목록에 존재한다면 EEXIST 에러를 발생 시킨다. event 집합은 *ev에 저장 된다.
EPOLL_CTL_MOD	*ev에 지정된 정보를 이용해 fd 설정 변경. 관심 목록에 없는 fd라면 ENOENT 에러를 발생 시킨다.
EPOLL_CTL_DEL	epfd 에서 fd를 제공 한다. epfd 관심 목록에 없는 fd를 제거하려면 ENOENT 에러를 발생 한다. fd를 닫으면 epoll 관심 목록에서 자동 제거 된다.

출처: http://z-man.tistory.com/158

epoll_wait

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

- epoll_wait 함수는 관심 있는 fd들에 무슨 일이 일어났는지 조사한다. 사건들의 리스트를 (epoll_event).events[] 의 배열로 전달한다. 또, 실제 동시 접속수와는 상관없이 maxevents 파라미터로 최대 몇 개까지의 event만 처리할 것임을 지정해 주도록 하고 있다.

- 만약 현재 접속수가 1만이라면 최악의 경우 1만개의 연결에서 사건이 발생할 가능성도 있기 때문에 1만개의 events[] 배열을 위해 메모리를 확보해 놓아야 하지만, 이 maxevents 파라미터를 통해 한번에 처리하길 희망하는 숫자를 제한할 수 있다.

- timeout은 epoll_wait의 동작특성을 지정해주는 중요한 요소인데 밀리세컨드 단위로 지정해주도록 되어 있다. 이 시간만큼 사건발생을 기다리라는 의미인데, 기다리는 도중에 사건이 발생하면 즉시 리턴 된다. 이 값에 (-1)을 지정해주면 영원히 사건을 기다리고(blocking), 0을 지정해주면, 사건이 있건 없건 조사만 하고 즉시 리턴 한다(즉 기다리지 않는다).

출처: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

- 간단한 채팅서버의 경우를 살펴보자. 서버가 어떠한 일을 해야 하는 시점은 이용자 누군가가 데이터를 보내왔을 때인데, 아무도 아무 말도 하지 않는다면 서버는 굳이 프로세싱을 할 이유가 없다. 이럴 때 timeout을 (-1)로 지정해두고 이용자들의 입력이 없는 동안 운영체제에 프로세싱 타임을 넘기도록 한다.

- 온라인게임(특히 MMORPG)의 경우에는 이용자의 입력이 전혀 없는 도중이라도, 몬스터에 관련된 처리, 적절한 저장, 다른 서버와의 통신들을 해야 하므로 적절한 timeout을 지정해 주도록 한다.

- 뭔가의 프로세싱을 주로 하면서 잠깐 잠깐 통신 이벤트를 처리하고자 하는 경우, 즉 프로세스의 CPU 점유를 높게 해서 무언가를 하고 싶은 경우에는 timeout 0을 설정하여 CPU를 독점하도록 설계할 수도 있다.

- 별도 thread를 구성하여 이 thread 가 입출력을 전담하도록 프로그램을 작성하고자 하는 경우에는 당연히 timeout을 (-1)로 설정하여 남는 시간을 다른 thread, 혹은 운영체제에 돌려 주도록 한다.

출처: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

```
#define MAX_EVENTS  100 // 최대 100개를 한번에 처리할 것이다.

struct epoll_event events[MAX_EVENTS];
int nfds, n;

for(;;){
    // 발생한 사건의 갯수를 얻어낸다. 0인 경우는 아무 일도 발생하지 않은 것
    nfds = epoll_wait(fd_epoll, events, MAX_EVENTS, 10);

    if(nfds < 0) {
        // critical error
        fprintf(stderr, "epoll_wait() error : %s\n", strerror(errno));
        exit(-1);
    }

    // 아무 일도 일어나지 않았다.
    if(nfds == 0){
        // idle
        continue;
    }

    for(n=0; n < nfds; ++n) OnEvent(&events[n]);
}
```

epoll_wait를 호출하고, 그 결과로 이벤트가 발생한 fd의 갯수가 돌아오며 어떤 fd들인지와 어떤 이벤트들인지는 epoll_event 구조체 배열에 담겨진다. 그 갯수만큼의 이벤트 처리를 해 주도록 했다.

출처: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

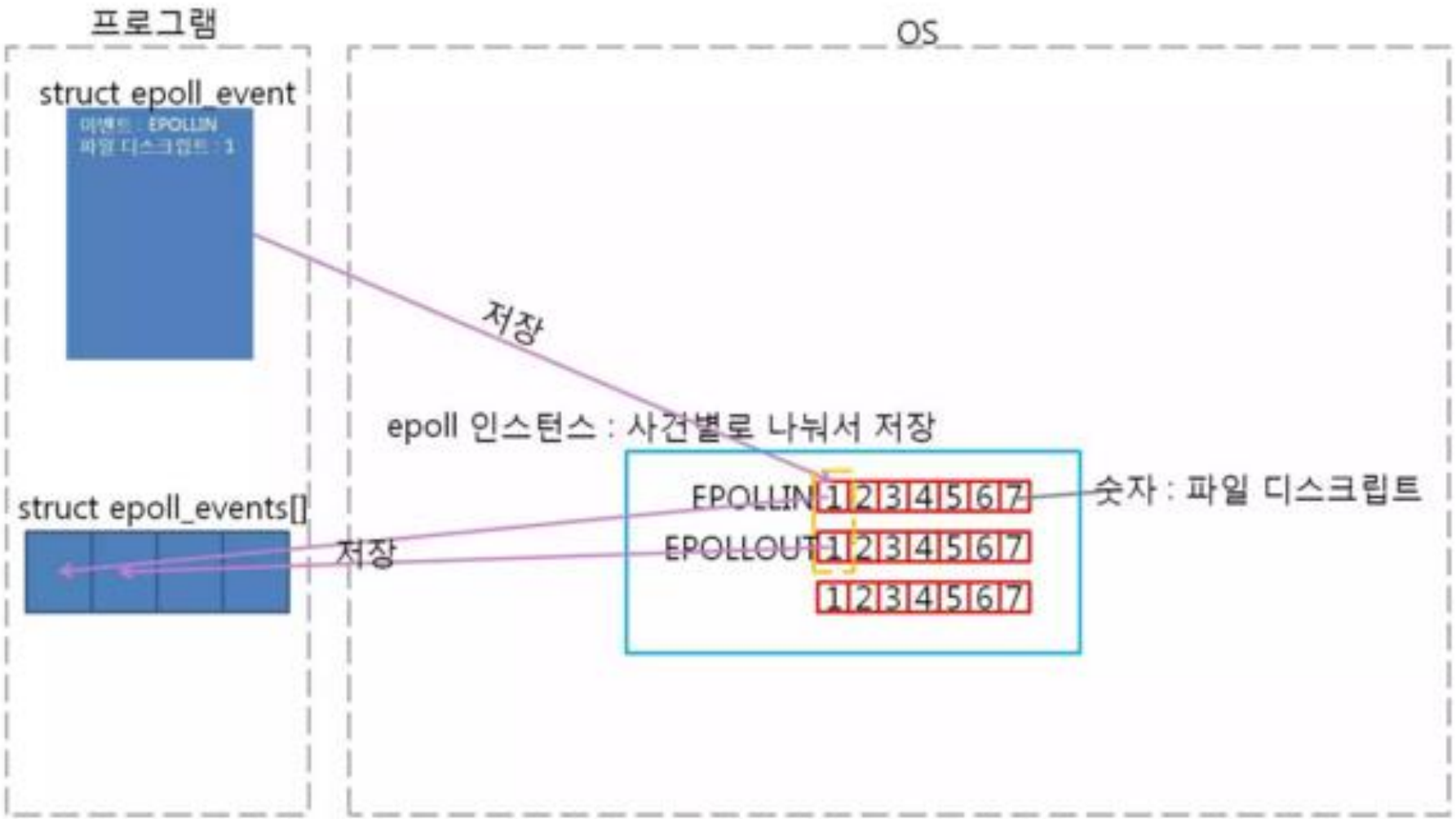
```
int OnEvent(const struct epoll_event *event)
{
    int nread;
    char buf[1024];

    if( event->events & EPOLLIN ){
        nread = read(event->data.fd, buf, 1024);

        if( nread < 1){
            fprintf(stdout,  " nread returns : %d\n", nread);
         } else {
            fprintf(stdout, "data : %s\n", buf);
            buf[0] = 0;
         }
    }
    if( event->events & EPOLLOUT){

    }
    if( event->events & EPOLLERR){

    }

    return 1;
}
```

구현은 특별한 내용은 없고,
다만 각 event들에 대해서 해당 비트가 셋 되어 있는지 확인하도록 하고 있고,
EPOLLIN 이벤트 즉 데이터가 수신 했을 때 읽어서 화면에 표시하도록 했다.

출처: http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

- EPOLLIN : 수신할 데이터가 존재하는 상황 (EPOLL_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- EPOLLOUT : 출력버퍼가 비워져서 당장 데이터를 전송할 수 있는 상황 (EPOLL_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- EPOLLPRI : OOB 데이터가 수신된 상황 (EPOLL_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- EPOLLRDHUP : 연결이 종료되거나 Half-close 가 진행된 상황, 이는 엣지 트리거 방식에서 유용하게 사용될 수 있다. 상대편 소켓 셧다운 (EPOLL_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- EPOLLERR : 에러가 발생한 상황 (EPOLL_WAIT의 출력으로만 사용됨)
- EPOLLHUP : 장애발생 (hangup) (EPOLL_WAIT의 출력으로만 사용됨)
- EPOLLET : 이벤트의 감지를 엣지 트리거 방식으로 동작 (EPOLL_CTL의 입력에만 사용됨)
- EPOLLONESHOT : 이벤트가 한번 감지되면, 해당 파일 디스크립터에서는 더 이상 이벤트를 발생시키지 않는다. 따라서 epoll_ctl 함수의 두번째 인자로 EPOLL_CTL_MOD을 전달해서 이벤트를 재설정해야 한다. (EPOLL_CTL의 입력에만 사용됨)

출처: http://stardreamsw.tistory.com/entry/epoll-%ED%95%A8%EC%88%98%EC%99%80-%EA%B5%AC%EC%A1%B0%EC%B2%B4-%ED%8C%8C%EB%9D%BC%EB%AF%B8%ED%84%B0



출처: http://comfun.tistory.com/entry/17%EC%9E%A5-select%EB%B3%B4%EB%8B%A4-%EB%82%98%EC%9D%80-epoll

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

int epoll_create(int size);
성공시 epoll 파일 디스크립터, 실패시 -1 반환

size : epoll 인스턴스의 크기 정보
```

출처: http://comfun.tistory.com/entry/17%EC%9E%A5-select%EB%B3%B4%EB%8B%A4-%EB%82%98%EC%9D%80-epoll

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

int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);
성공시 0, 실패시 -1 반환

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

출처: http://comfun.tistory.com/entry/17%EC%9E%A5-select%EB%B3%B4%EB%8B%A4-%EB%82%98%EC%9D%80-epoll

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

int epoll_wait(int epfd, struct epoll_event* events, int maxevents, int timeout);
성공시 이벤트가 발생한 파일 디스크립터의 수, 실패시 -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 발생 할 때까지 무한 대기
```

출처: http://comfun.tistory.com/entry/17%EC%9E%A5-select%EB%B3%B4%EB%8B%A4-%EB%82%98%EC%9D%80-epoll

Sample: Echo Server

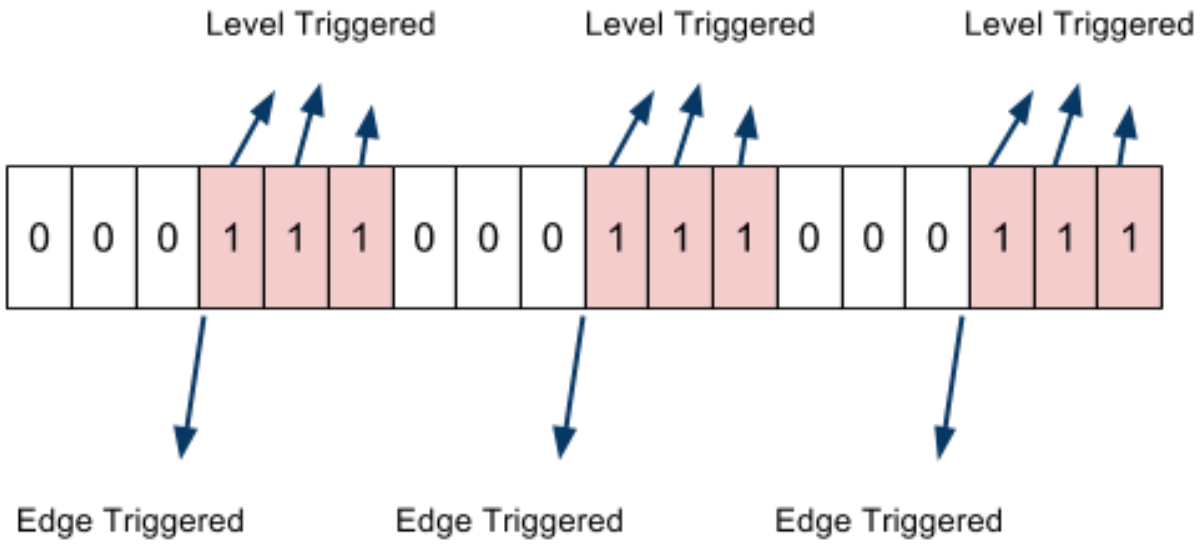
https://gist.github.com/jacking75/6e8cef4141e82811c806

Edge Trigger 과 Level Trigger

- epoll에서는 Edge Trigger과 level Trigger 둘 중 하나를 선택해야 한다.
- 기본은 level Trigger

- Level-triggered는 **특정 준위 (상태)가 유지되는 동안 감지**.
- Edge-triggered는 **특정 준위가 변화하는 시점에서만 감지**.

- 예를 들어 디지털 신호 00001110000111000111 에서 1에 대한 Trigger이라면 LT는 1이 유지되는 시간 동안 횟수에 상관없이 발생하고, Edge-triggered는 0에서 1로 변하는 시점에서만 발생한다. 즉 이 경우 ET는 3회 발생한다.



출처: http://www.joinc.co.kr/modules/moniwiki/wiki.php/Site/Network_Programing/AdvancedComm/epoll24

- LT 방식은 select나 poll처럼 작동을 하기 때문에 쉽게 이해할 수 있다. select(:2), poll은 LT로 작동하는 데, 소켓 버퍼가 비기 전까지 1로 설정하기 때문에, 소켓에 버퍼가 있는 동안에는 계속 반환한다.

- 반면 ET는 처음 이벤트가 발생해서 1 이 되었을 때만 발생한다. 만약 이벤트가 발생해서 데이터를 읽었는데, 버퍼의 데이터를 모두 읽지 않고 다음 wait 함수를 호출할 경우, wait 함수에서 봉쇄되는 문제가 생길 수 있다. 자칫하면 영원히 봉쇄될 수도 있다.

- 다음의 시나리오를 가정해자.
  read sid of pipe(RFD)에 있는 파일 지정자가 epoll 장치에 ET 상태로 추가된다.
  Pipe write가 2Kb의 데이터를 쓴다.
  epoll_wait(2)가 호출되고 RFD는 이벤트가 발생한 파일 지정자를 리턴 한다.
  Pipe reader은 RFD로 부터 1Kb데이터를 읽는다.
  epoll_wait(2)가 호출된다.
  ET 방식이기 때문에 epoll_wait에서 봉쇄된다.

- ET로 작동하게 하려면 non-blocking 소켓에 사용해야 한다.

- **read(2) 나 write(2) 가 errno로 EGAIN을 반환 할 때만 wait한다(epoll_wait)**.

출처: http://www.joinc.co.kr/modules/moniwiki/wiki.php/Site/Network_Programing/AdvancedComm/epoll24

Edge Trigger가 좋은 점

- 성능보다 다음 사항 때문에 좋다.
- 여러 스레드에서[epoll_wait] 했을 때 하나의 스레드만 통보를 받는다.
- EPOLLOUT로 쓰기 가능 플래그를 일일이 add/del 하지 않는다.
  읽기와 달리 소켓은 대부분의 경우 쓰기 가능하므로 레벨 트리거는 플래그를 일일이 붙일 필요가 있다.
  엣지 트리거는 플래그를 켜지 않아도 된다.

Edge Trigger가 나쁜 점

- epoll(2)의 엣지트리거에서는 디스크립터 정보를 통한 수신/송신으로 EAGAIN이 왔을 때 모든 데이터를 수신/송신 했는지 확신 할 수 없다.
- 적은 데이터 양으로 한번의 수신/송신에서는 문제 없지만 데이터 양이 많을 때는 모든 데이터를 처리했는지를 확인해야 한다. 이 조사는 애플리케이션 측에서 책임을 져야한다.
- 즉, 각 디스립터에 대한 수신/송신을 하는 데이터를 개인 데이터 영역에 일시 저장하는 장치를 마련해야 한다.

Edge Trigger 사용

```
mev.events = EPOLLIN | EPOLLET ....;
epoll_ctl(mepollfd, EPOLL_CTL_ADD, afd, &mev);
```

- 대상 디스크립터를 epoll(7)로 관리할 때 EPOLLET 플래그를 붙인다.
- 대상 디스크립터를 논 블로킹으로 설정 한다.
- 대상 디스크립터를 통해서 모든 데이터를 read(2)/write(2) 할 때 EAGAIN가 반환되면 처리를 마친다.

Level Trigger를 Edge Trigger처럼 사용

- 대상 디스크립터를 epoll(7)로 관리할 때 EPOLLONESHOT 플래그를 붙인다.
- 대상 디스크립터를 논 블로킹 설정으로 한다.
- 대상 디스크립터를 통해서 모든 데이터를 read(2)/write(2)때 EAGAIN가 반환되면 처리를 마친다.
- 처리를 마친 때 epoll_ctl(2)을 실행하고, 이때의 플래그로서 EPOLL_CTL_MOD 플래그를 사용한다.
- EPOLL_CTL_MOD 플래그를 사용해서 epoll_ctl(2)을 실행하는 것은 유저(프로그래머)에게 맡긴다.

EPOLLONESHOT

- 기본적으론 epoll_ctl에서 관심목록에 추가하면 제거할 때까지 활성 상태로 남게 된다.
- 특정 fd로부터 통지를 받으려면 ev.ents값에 EPOLLONESHOT를 지정하면 된다.
- 지정하고 wait를 호출하며 해당 fd가 ready 상태인지 알려준 뒤 해당 fd를 비활성화 되고, 이후에 다시 wait를 호출하면 비활성화된 fd는 안 알려준다.(필요할 경우 EPOLL_CTL_MOD를 사용해 재활성화 해야 한다.)

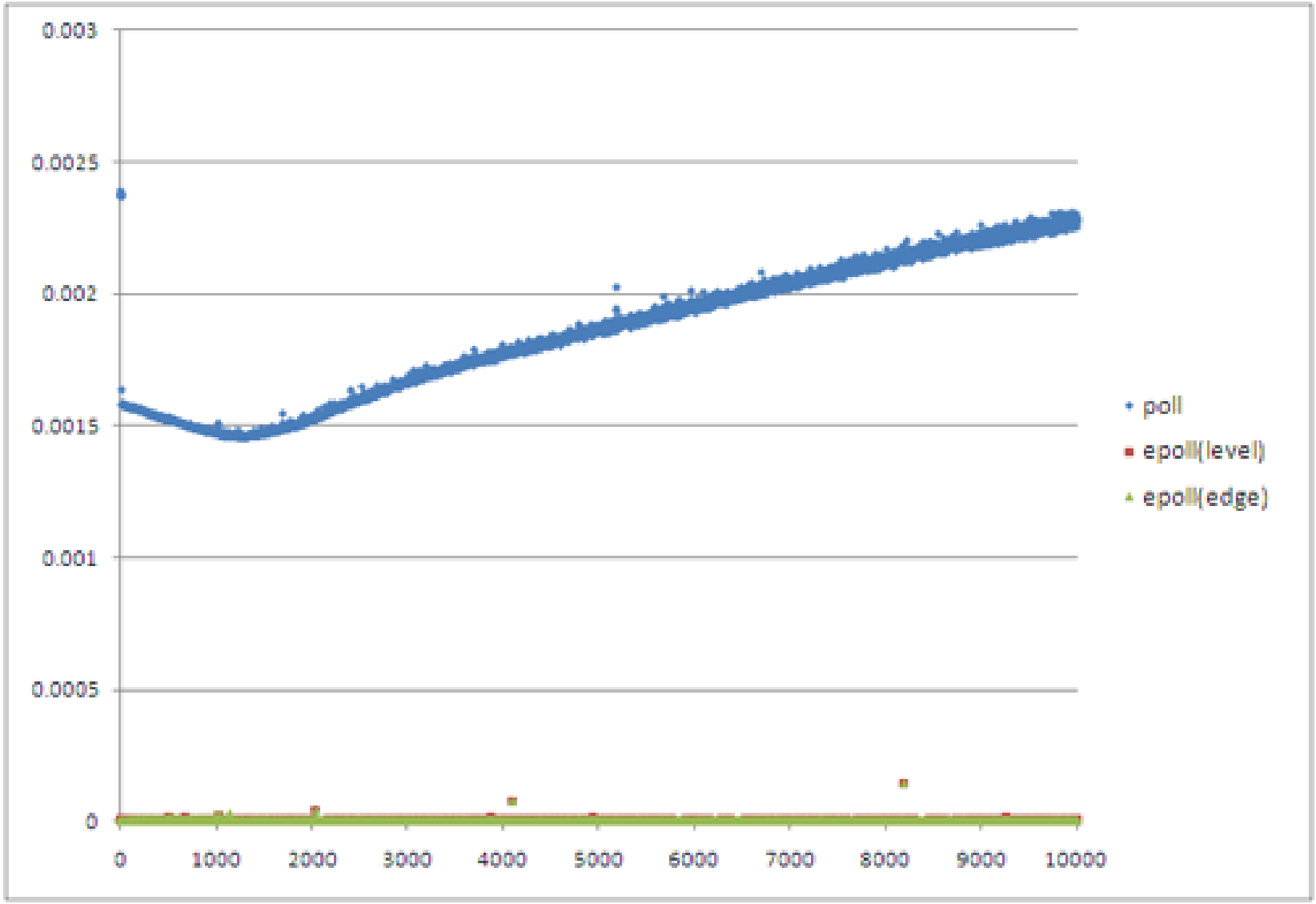
출처: http://z-man.tistory.com/158

poll vs epoll

```
# cat /proc/cpuinfo | grep "model name"
model name : Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz
model name : Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz
# cat /proc/meminfo | grep MemTotal
MemTotal: 2015528 kB
# cat /etc/redhat-release
CentOS release 5.3 (Final)
# uname -a
Linux host 2.6.18-128.el5 #1 SMP Wed Dec 17 11:41:38 EST 2008 x86_64 x86_64 x86_64 GNU/Linux
```

출처: http://codezine.jp/article/detail/4830

연결 성능 조사




```
# ./poll_conn 10000
# ./epoll_level_conn 10000
# ./epoll_edge_conn 10000
```

가로 축은 커넥션 수, 세로축은 초.

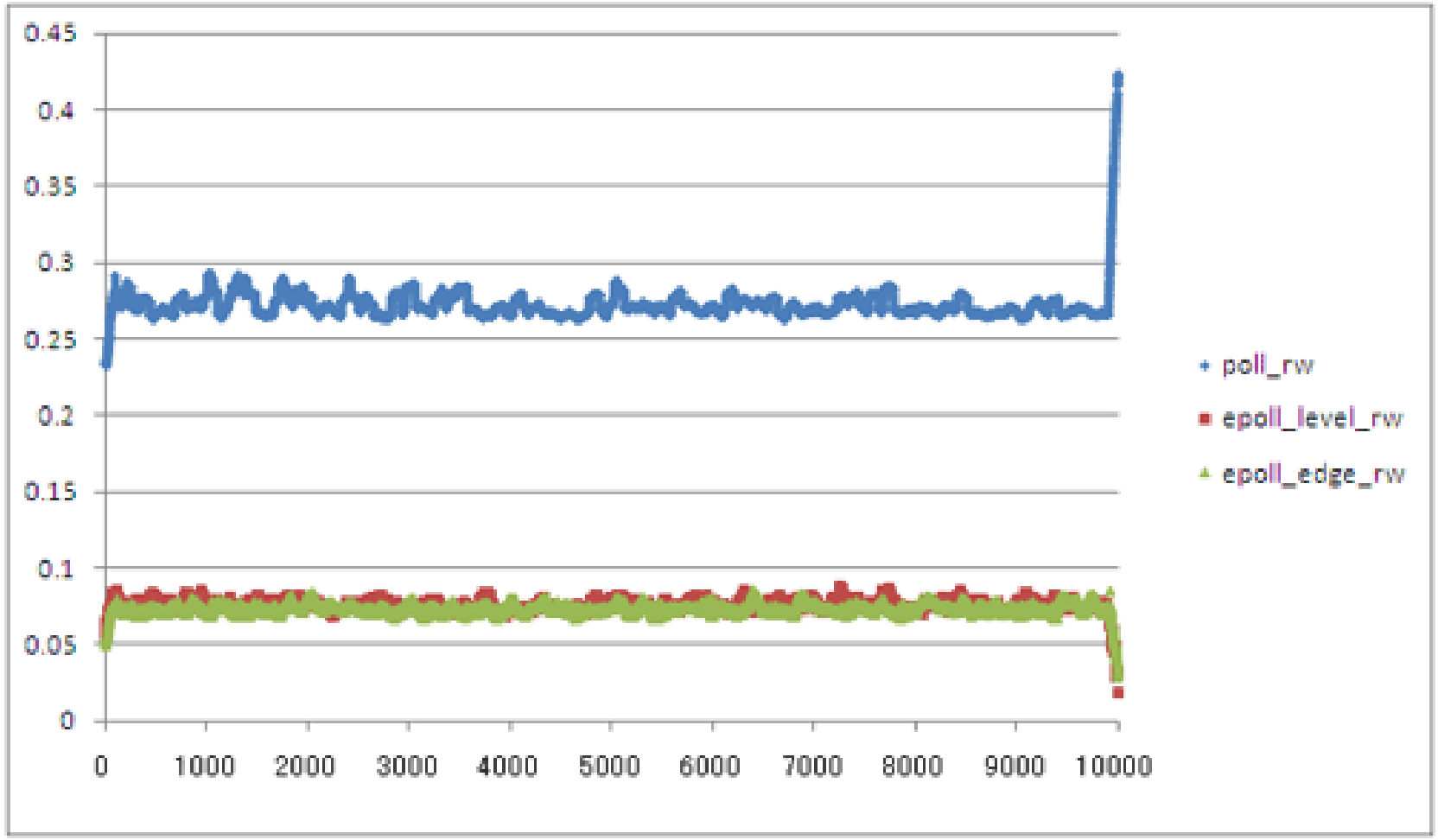
poll은 커넥션 수에 따라서 완만한 각도로 늘었고, epoll은 커넥션의 수에 따른 응답치의 변화가 전혀 없고, 0부근에서 추이하고 있다.

이것만 보면 poll은 매우 늦고 epoll의 성능이 좋음을 알 수 있다.

출처: http://codezine.jp/article/detail/4830

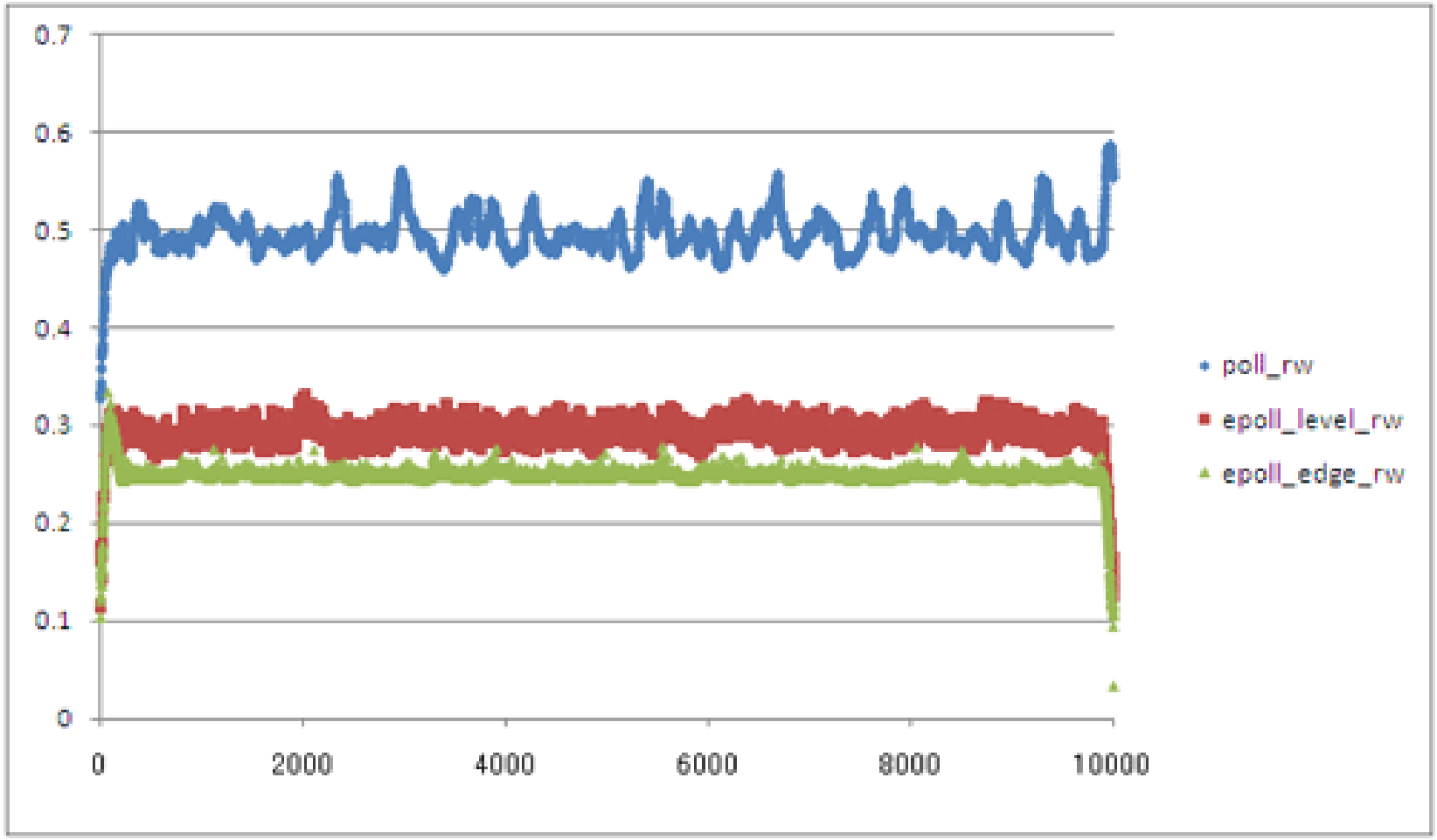
poll vs epoll: 송수신

연결 수는 10,000, 처리 회수는 100회와 1000 회, 데이터 크기는 1,024 바이트와 10,240 바이트.



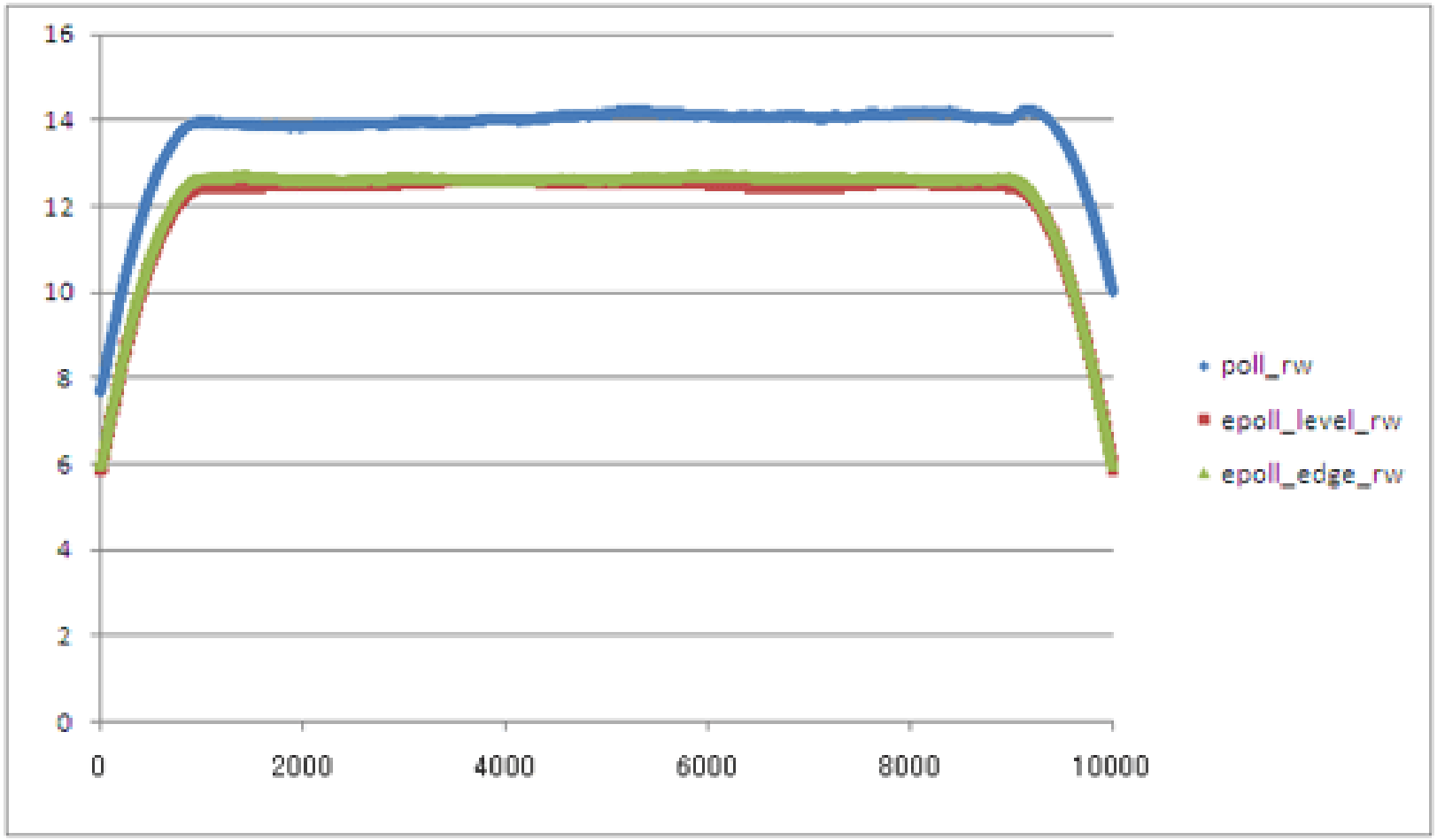

```
# ./poll_performance 10000 100 1024
# ./epoll_level_performance 10000 100 1024
# ./epoll_edge_performance 10000 100 1024
```

출처: http://codezine.jp/article/detail/4830




```
# ./poll_performance 10000 100 10240
# ./epoll_level_performance 10000 100 10240
# ./epoll_edge_performance 10000 100 10240
```

출처: http://codezine.jp/article/detail/4830



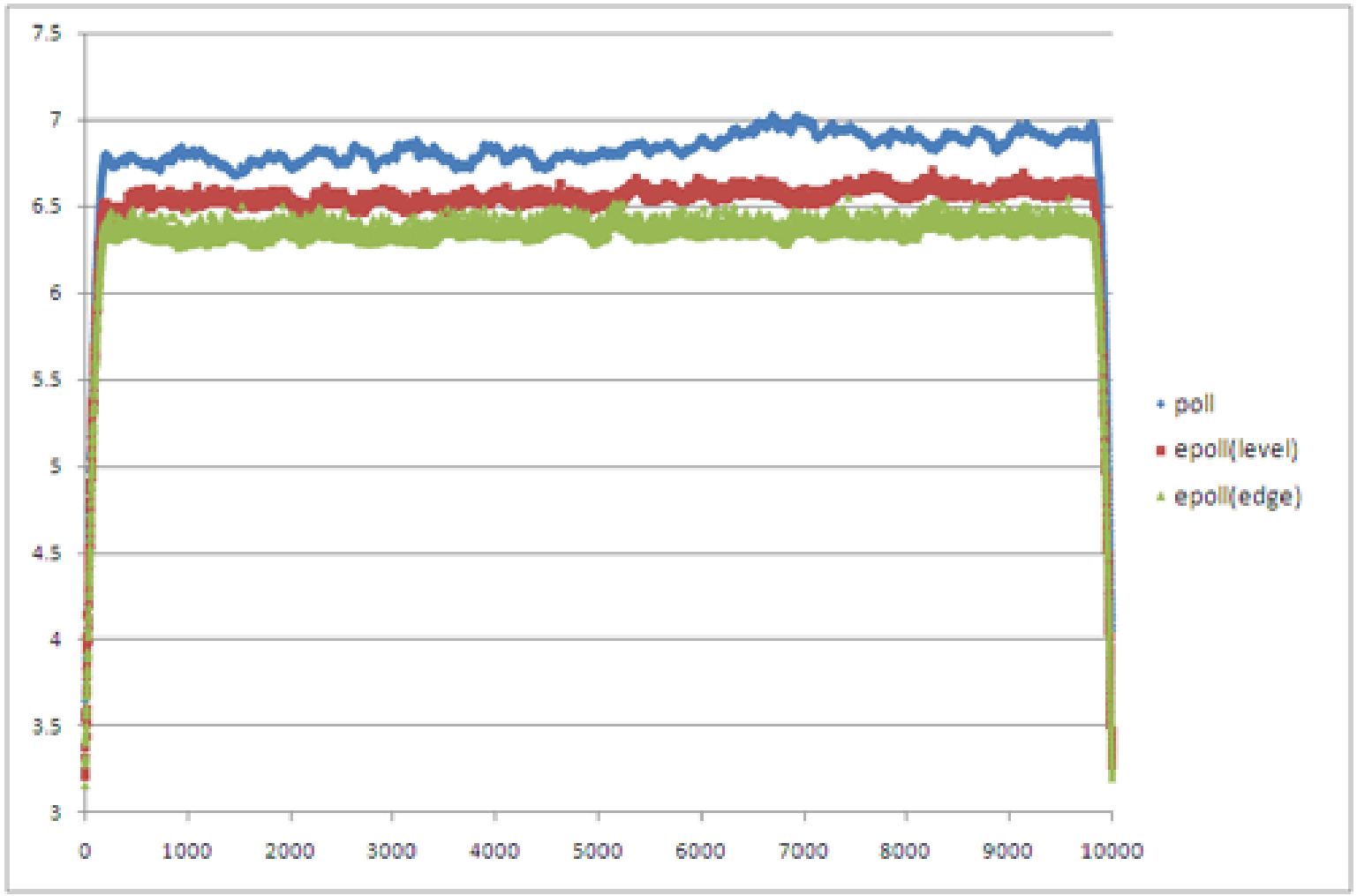

```
# ./poll_performance 10000 1000 1024
# ./epoll_level_performance 10000 1000 1024
# ./epoll_edge_performance 10000 1000 1024
```

출처: http://codezine.jp/article/detail/4830

연결 성능 테스트와 비슷하게 epoll의 성능이 poll에 비해서 뛰어난 것을 알 수 있다.

epoll의 엣지트리거가 레벨트리거에 비해 좀 더 좋다는 것도 알 수 있다.

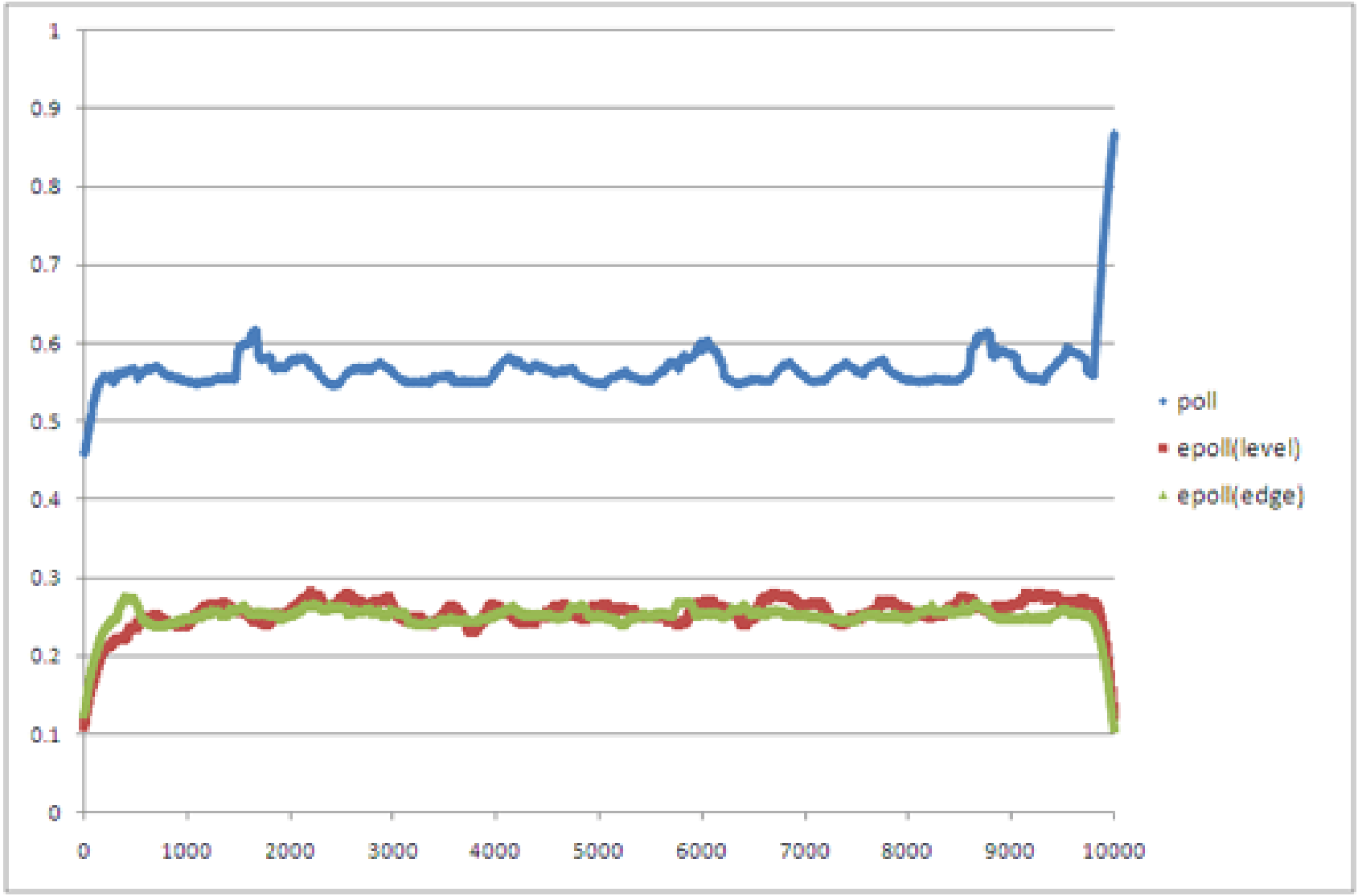
poll vs epoll: 송수신(큰 데이터)




```
# ./poll_performance_2 10000 100 102400
# ./epoll_level_performance_2 10000 100 102400
# ./epoll_edge_performance_2 10000 100 102400
```

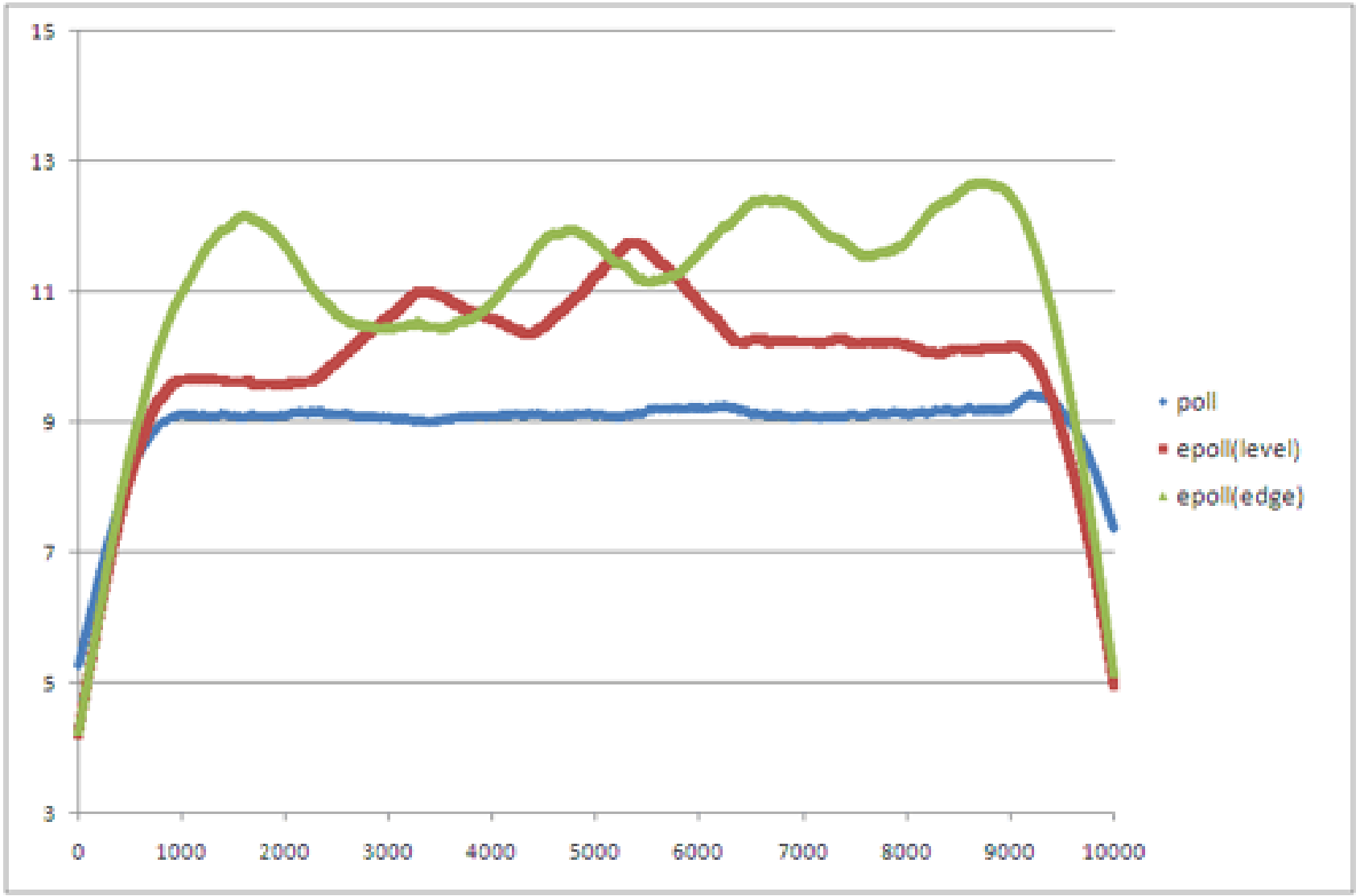
출처: http://codezine.jp/article/detail/4830

이번에는 데이터 크기는 같고 많은 연결 수에서 회수를 변경하여 성능 조사.
연결 수는 10000, 처리 회수를 100회, 500회, 1000회로하고 데이터 량은 1024 바이트로 한다.



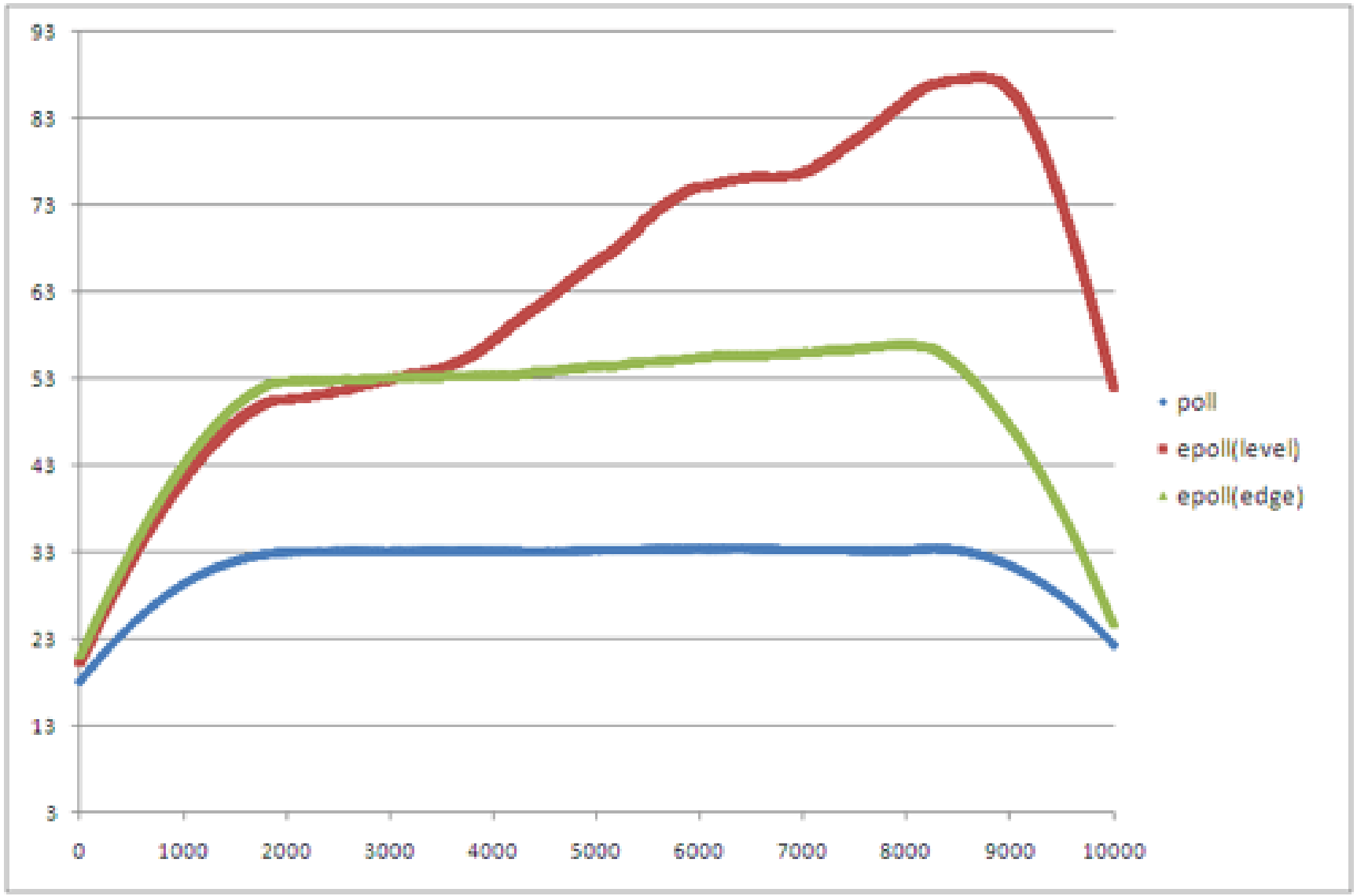

```
# ./poll_performance_2 10000 100 1024
# ./epoll_level_performance_2 10000 100 1024
# ./epoll_edge_performance_2 10000 100 1024
```

출처: http://codezine.jp/article/detail/4830




```
# ./poll_performance_2 10000 500 1024
# ./epoll_level_performance_2 10000 500 1024
# ./epoll_edge_performance_2 10000 500 1024
```

출처: http://codezine.jp/article/detail/4830




```
# ./poll_performance_2 10000 1000 1024
# ./epoll_level_performance_2 10000 1000 1024
# ./epoll_edge_performance_2 10000 1000 1024
```

출처: http://codezine.jp/article/detail/4830

epoll이 느린 이유...

디스크립터의 송신/수신 대상 데이터의 관리 방법으로 이 테스트에서는 간결하게 구조체 배열을 준비하고 멤버로서 디스크립터 및 송수신 데이터 양을 관리하고 있다.

epoll의 경우 ready가 된 디스크립터만 알려주기 때문에 그 디스크립터가 관리 하고 있는 데이터가 어떤 것인지 디스크립터 관리용 구조체에서 검색할 필요가 있으며, 그 로직을 이번에는 단순히 선두에서 찾는 방법으로 하고 있다.

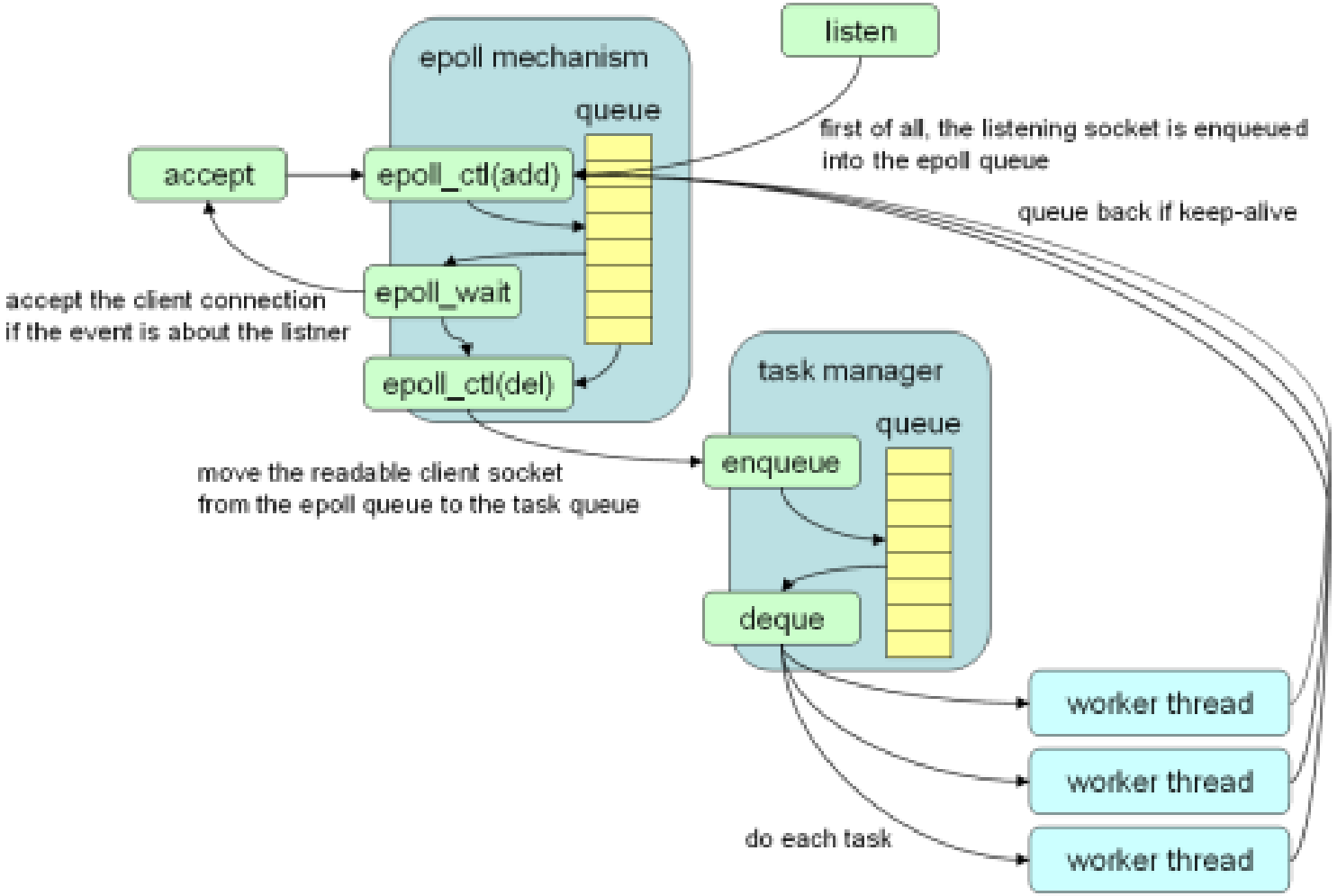
```
for(j=0;process_info[j]. sock!=evs[i]. data.fd;j++);
```

poll의 경우 반환된 값은 ready가 된 기술자 수이고, 어느 디스크립터가 ready로 됐는지 확인하기 위해 poll용 배열을 선두에서 찾지만, poll용 배열의 인덱스 값이 그대로 디스크립터 관리용 구조체의 인덱스 값과 동일하므로 관리 정보를 취득할 때 선두에서 쭉 찾을 필요가 없다

```
if(pfds[i]. revents&(POLLIN| POLLOUT)==0){
    continue;
}
```

즉 각 디스크립터에 대해서 데이터를 관리할 경우 어떻게 epoll(2)에서는 어떻게 관리하느냐에 따라서 성능치가 poll(2)을 사용했을 때보다 밑돌 수 있다. 디스크립터 정보 관리 방식이 그대로 성능에 직결되는 것을 제대로 인식하고 epoll(2)에 의존하지 않는 알고리즘이나 설계가 필요하다.

epoll과 멀티스레드



보스/워커 스레드 모델.

보스 스레드는 epoll 큐를 감시하다가 반환 된 파일 디스크립터를 epoll 큐에서 태스크 큐로 이동시킨다.
태스크 큐 내에서 파일 디스크립터를 발견한 워커 스레드는 꺼낸 파일 디스크립터를 통해 세션을 실행하고, 끝나면 파일 디스크립터를 epoll 큐에 돌려준다.

이렇게 하면 항상 정해진 수의 워커 스레드가 병렬로 동작하므로, 각각의 세션의 성능이 안정된다. 리퀘스트 수가 너무 많은 경우에는 일시적으로 태스크 큐가 넘치지만 그래도 서버에 hang이 걸리는 경우는 없다.

워커 스레드 수는 CPU의 코어 개수보다 조금 많은 정도로 하면 좋다.

출처: http://alpha.mixi.co.jp/entry/2008/10658/

epoll-example
(멀티스레드 버전, 엣지트리거 사용)

https://github.com/RajivKurian/epoll-example

참고

Joinc - epoll
http://www.joinc.co.kr/modules/moniwiki/wiki.php/Site/Network_Programing/AdvancedComm/epoll24

[C언어] epoll 설명
http://blueheartscabin.blogspot.kr/2013/08/c-epoll.html

17장-select보다-나은-epoll
http://comfun.tistory.com/entry/17%EC%9E%A5-select%EB%B3%B4%EB%8B%A4-%EB%82%98%EC%9D%80-epoll

I/O - epoll
http://z-man.tistory.com/158

epoll 함수와 구조체 파라미터
http://stardreamsw.tistory.com/entry/epoll-%ED%95%A8%EC%88%98%EC%99%80-%EA%B5%AC%EC%A1%B0%EC%B2%B4-%ED%8C%8C%EB%9D%BC%EB%AF%B8%ED%84%B0

epoll 코드

linux epoll을 이용한 echo server(엣지트리거 사용)
http://goldenretriever.tistory.com/entry/linux-epoll%EC%9D%84-%EC%9D%B4%EC%9A%A9%ED%95%9C-echo-server

epoll sample (엣지트리거 사용)
https://github.com/araijn/samples/tree/master/C/epoll

epoll echo server
http://blog.ilovelinux.org/2009/10/epoll-echo-server.html

echo chat server
http://victor8481.tistory.com/98

채팅 서버 - epoll 버전
http://firstboos.tistory.com/entry/%EC%B1%84%ED%8C%85-%EC%84%9C%EB%B2%84-epoll

TCP Echo Server Example in C++ Using Epoll
http://blog.varunajayasiri.com/tcp-echo-server-example-in-c-using-epoll

간단한 epoll thread-pooling server
http://alleysark.tistory.com/228

epoll-example (멀티스레드 버전, 엣지트리거 사용)
https://github.com/RajivKurian/epoll-example

a server application implemented through nonblock IO and thread pool (레벨트리거 사용)
https://github.com/honeyligo/EpollServer

eznetpp: Asynchronous network library for c++ programmers
https://github.com/kangic/eznetpp

zsummer is a cross-platform C++ high performance lightweight network library. via IOCP/EPOLL
https://github.com/zsummer/zsummer

EasyGameServer
https://github.com/zeliard/EasyGameServer/tree/linux