Comparison of Netfilter and eBPF packet processing

2021320097 이현서

제출일: 6/20(금)

프리데이 사용일 수: 7일

1. 배경지식

1) Netfilter

Netfilter는 Linux 커널 수준에서 패킷 필터링, 네트워크 주소 변환(NAT), 패킷 로깅 등을 수행할 수 있도록 해주는 프레임워크이다. 이는 사용자 공간에서 iptables, nftables 등의 명령어를 통해 접근되며, 커널 내부의 여러 훅(hook) 지점을 통해 수신/전송 중인 패킷을 검사하거나 수정하는 기능을 제공한다.

Netfilter는 NF_INET_PRE_ROUTING, NF_INET_LOCAL_IN, NF_INET_FORWARD, NF_INET_LOCAL_OUT, NF_INET_POST_ROUTING 총 다섯 개 훅을 기반으로 작동한다. 이러한 훅 지점에서 nf_register_net_hook() 함수를 통해 사용자 정의 콜백 함수를 등록할 수 있도록 하며, 이를 통해 패킷 필터링, 변경, 드롭(drop) 또는 수락(accept) 같은 정책을 적용할 수 있다.

Netfilter의 비교적 높은 수준의 추상화와 유연한 제어가 가능하지만 단점 또한 존재한다. Netfilter는 리눅스 커널 내부의 패킷 처리 루틴 중 비교적 상위 계층에서 동작한다. 네트워크 인터페이스에서 수신된 패킷은 먼저 드라이버를 통해 softirq 컨텍스트로 올라오고, 커널이 패킷을 파싱하여 skb (socket buffer)를 생성한 이후에야 Netfilter 훅 포인트에 도달한다. Netfilter는 이 지점에서 필터링, 포워딩, NAT, 로깅 등의 작업을 수행할 수 있기 때문에, 패킷 하나를 처리하는 데에 이미 상당한 커널 리소스가 소모된 이후여서, 고속 패킷 처리 상황에서는 병목이 발생할 수 있다.

2) eBPF

eBPF는 커널 내부에 안전하게 코드를 주입하고 실행할 수 있게 해주는 고성능의 가상 머신 기반 기술이다. 기존 BPF(Berkeley Packet Filter)는 단순히 패킷 필터링만 수행했지만, eBPF는 더 확장되어 커널 내부의 다양한 이벤트에 대응할 수 있도록 발전했다.

eBPF는 C 언어로 작성된 코드를 변환하여 로딩되며, bpftool, libbpf, iproute2 등 다양한 도구로 관리할 수 있다. 네트워크 영역에서는 eBPF를 활용한 패킷 필터링, 트래픽 계측, 리디렉션, 로드 밸런싱 등의 고성능 처리가 가능하다.

3) XDP

XDP는 eBPF 기술을 기반으로 한 고성능 네트워크 패킷 처리 기술로, 네트워크 인터페이스 카드(NIC)에서 패킷이 수신된 직후, 즉 커널 네트워크 스택에 진입하기 이전 단계에서 직접 eBPF 프로그램을 실행할 수 있도록 한다. 이로 인해 최소한의 오버헤드로 패킷을 드롭하거나 수정, 리디렉션할 수 있어 초고속 처리가 가능하다.

XDP의 동작 위치는 리눅스 네트워크 스택 중 가장 빠른 수준인 driver layer이며, 일반적인 Netfilter보다 훨씬 앞단에서 동작한다. 이 때문에 고성능 방화벽, DDoS 대응, 패킷 필터링 등에 매우 유리하다.

2. 코드 설명

1) my_nefilter.c

```
#define TARGET_PORT 8080
#define REDIRECT_IP 0x0A000102    // 10.0.1.2
#define REDIRECT_PORT 8083
```

리다이렉션 기준 포트와 목적지 IP주소 및 포트를 설정한다.

```
static unsigned int my_nf_hookfn(void *priv,
                                 struct sk_buff *skb,
                                 const struct nf_hook_state *state) {
    struct iphdr *iph;
    struct udphdr *udph;

    if (!skb)
        return NF_ACCEPT;

    iph = ip_hdr(skb);
    if (iph->protocol != IPPROTO_UDP)
        return NF_ACCEPT;

    udph = udp_hdr(skb);

    if (ntohs(udph->dest) == TARGET_PORT) {

        printk(KERN_INFO "FORWARDING: UDP:%pI4:%u;%pI4:%u\n",
            &iph->saddr, ntohs(udph->source),
            &iph->daddr, ntohs(udph->dest));


        iph->daddr = htonl(REDIRECT_IP);
        udph->dest = htons(REDIRECT_PORT);

        udph->check = 0;
        skb->csum = 0;
        skb->ip_summed = CHECKSUM_NONE;
        ip_send_check(iph);


        printk(KERN_INFO "MODIFIED:     UDP:%pI4:%u;%pI4:%u\n",
            &iph->saddr, ntohs(udph->source),
```

```
                &iph->daddr, ntohs(udph->dest));
    }

    return NF_ACCEPT;
}
```

my_nf_hookfn은 NF_INET_PRE_ROUTING 지점에 등록되어, 패킷이 라우팅 결정 전에 커널 네트워크 스택을 통과하는 시점에서 실행된다.

함수는 먼저 skb(socket buffer)가 존재하지 않으면 아무 작업 없이 NF_ACCEPT를 반환하여 패킷을 그대로 통과시킨다. 이후 ip_hdr(skb)를 통해 IP 헤더를 추출하고, 해당 패킷이 UDP 프로토콜인지 확인한다. 만약 UDP가 아니라면 역시 NF_ACCEPT로 통과시킨다.

다음으로 UDP 헤더를 가져와서, 목적지 포트 번호가 TARGET_PORT로 정의된 포트(예: 8080)인지 확인한다. 만약 일치한다면, 해당 패킷은 리다이렉션 대상이므로 처리에 들어간다.

리다이렉션 처리 과정에서는 먼저 printk()를 통해 기존의 출발지와 목적지 IP 및 포트를 커널 로그에 출력하여 디버깅 정보를 남긴다. 이후 목적지 IP를 REDIRECT_IP, 포트를 REDIRECT_PORT로 변경한다.

패킷 헤더가 변경되었으므로, 체크섬을 다시 계산해주어야 하기 때문에UDP 체크섬과 skb->csum을 0으로 초기화하고, 커널에게 수동 체크섬 계산을 요청하기 위해 skb->ip_summed를 CHECKSUM_NONE으로 설정한다. 마지막으로 ip_send_check() 함수를 호출하여 IP 헤더의 체크섬을 다시 계산한다.

```
static struct nf_hook_ops nfho = {
    .hook        = my_nf_hookfn,
    .hooknum    = NF_INET_PRE_ROUTING,
    .pf          = PF_INET,
    .priority = NF_IP_PRI_FIRST
};
```

my_nf_hookfn을 등록하고 NF_INET_PRE_ROUTNIG 에 훅을 걸도록 한다. 또한 IPv4 패킷에 대해서만 동작하도록 하고 가장 먼저 실행되도록 우선순위를 설정한다.

```
static int __init my_init(void) {
    return nf_register_net_hook(&init_net, &nfho);
}

static void __exit my_exit(void) {
    nf_unregister_net_hook(&init_net, &nfho);
}
```

```
module_init(my_init);
module_exit(my_exit);
```

모듈이 로드되면 my_init()이 실행되어 훅이 등록되게 하고 모듈이 언로드되면 my_exit()이 실행되어 훅이 해제되도록 한다.

2) my_xdp.c

```
#define ETH_P_IP 0x0800
#define TARGET_PORT 8080
#define REDIRECT_PORT 8083
#define REDIRECT_IP 0x0a000102
```

IPv4 프로토콜 사용을 명시하고, 리다이렉션 기준 포트와 목적지 IP주소 및 포트를 설정한다.

```
SEC("xdp")
```

함수가 XDP hook point에 붙을 것임을 명시한다.

```
static __always_inline __u16 csum_fold_helper(__u32 csum) {
    #pragma unroll
    for (int i = 0; i < 4; i++) {
        if ((csum >> 16) == 0)
            break;
        csum = (csum & 0xFFFF) + (csum >> 16);
    }
    return ~csum;
}
```

추후 최종 체크섬 계산에 사용하기 위하여 누적값을 16비트로 flod하고 보수를 취하는 함수이다. 이 때 #pragma unroll을 사용하여 루프를 안정적으로 수행하도록 하였다.

```
static __always_inline __u32 sum16(const void *start, __u32 size, const void *data_end) {
    __u32 sum = 0;
    const __u16 *ptr = start;

    #pragma unroll
    for (int i = 0; i < 256; i++) {
        if ((void *)(ptr + 1) > data_end || size < 2)
            break;
        sum += *ptr++;
        size -= 2;
    }
```

```
    if (size == 1) {
        if ((void *)ptr >= data_end) return 0;
        sum += *(__u8 *)ptr;
    }

    return sum;
}
```

이 또한 추후 체크섬 계산에 사용하기 위해 16비트 단위 합산을 구하는 함수이다. 여기서도 #pragm unroll을 통해 루프가 안정적으로 동작하도록 하였다.

```
int redirect_udp(struct xdp_md *ctx) {
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;

    struct ethhdr *eth = data;
    if ((void *)(eth + 1) > data_end) return XDP_PASS;

    if (bpf_ntohs(eth->h_proto) != ETH_P_IP) return XDP_PASS;

    struct iphdr *iph = (void *)(eth + 1);
    if ((void *)(iph + 1) > data_end) return XDP_PASS;
    if (iph->protocol != IPPROTO_UDP) return XDP_PASS;

    struct udphdr *udph = (void *)iph + iph->ihl * 4;
    if ((void *)(udph + 1) > data_end) return XDP_PASS;

    if (bpf_ntohs(udph->dest) != TARGET_PORT) return XDP_PASS;

    //bpf_printk("Before: %pI4:%d -> %pI4:%d\n",
       //         &iph->saddr, bpf_ntohs(udph->source),
          //      &iph->daddr, bpf_ntohs(udph->dest));

    iph->daddr = bpf_htonl(REDIRECT_IP);
    udph->dest = bpf_htons(REDIRECT_PORT);

    iph->check = 0;
    __u32 ip_csum = sum16(iph, iph->ihl * 4, data_end);
    iph->check = csum_fold_helper(ip_csum);
```

```
    __u16 udp_len = bpf_ntohs(udph->len);
    udph->check = 0;

    __u32 pseudo = 0;
    pseudo += (iph->saddr >> 16) + (iph->saddr & 0xffff);
    pseudo += (iph->daddr >> 16) + (iph->daddr & 0xffff);
    pseudo += bpf_htons(IPPROTO_UDP);
    pseudo += udph->len;

    __u32 udp_csum = pseudo + sum16(udph, udp_len, data_end);
    udph->check = csum_fold_helper(udp_csum);

    //bpf_printk("After: Redirected to %pI4:%d\n", &iph->daddr, REDIRECT_PORT);

    return XDP_PASS;
}
```

ctx-> data 와 ctx->data_end를 통해 패킷의 시작과 끝 주소를 얻는다. 이후 ethernet 헤더를 읽고 data_end를 넘어서거나 ethernet 프로토콜이 IPv4가 아니면 XDP_PASS 처리한다. 다음으로 IP 헤더를 읽어와 똑같이 data_end를 넘는지 확인하고 UDP인지 확인한다. 그 다음으로는 UDP 헤더를 가져와서 data_end를 넘는지 확인하고 TARGET_PORT가 맞는지 확인한다.

확인을 마친 목적지 IP와 포트를 새로운 값으로 변경해준다. 다음 IP헤더의 체크섬을 0으로 초기화 시킨 뒤, 새로 계산한다. 마지막으로 csum_fold_helper()를 통해 최종 체크섬으로 fold하여 마무리한다. UDP도 마찬가지로 길이를 가져오고 체크섬을 0으로 초기화시킨다. IP 체크섬과는 다르게 송신 및 수신 IP, 프로토콜 번호, UDP길이를 포함하여 가상 헤더를 구성하고, csum_fold_helper()를 통해 최종 체크섬을 저장한다. 마지막으로 변경된 패킷을 XDP_PASS를 사용하여 넘긴다.

3. 실험 설명

1) 클라이언트부터 서버까지의 패킷 경로

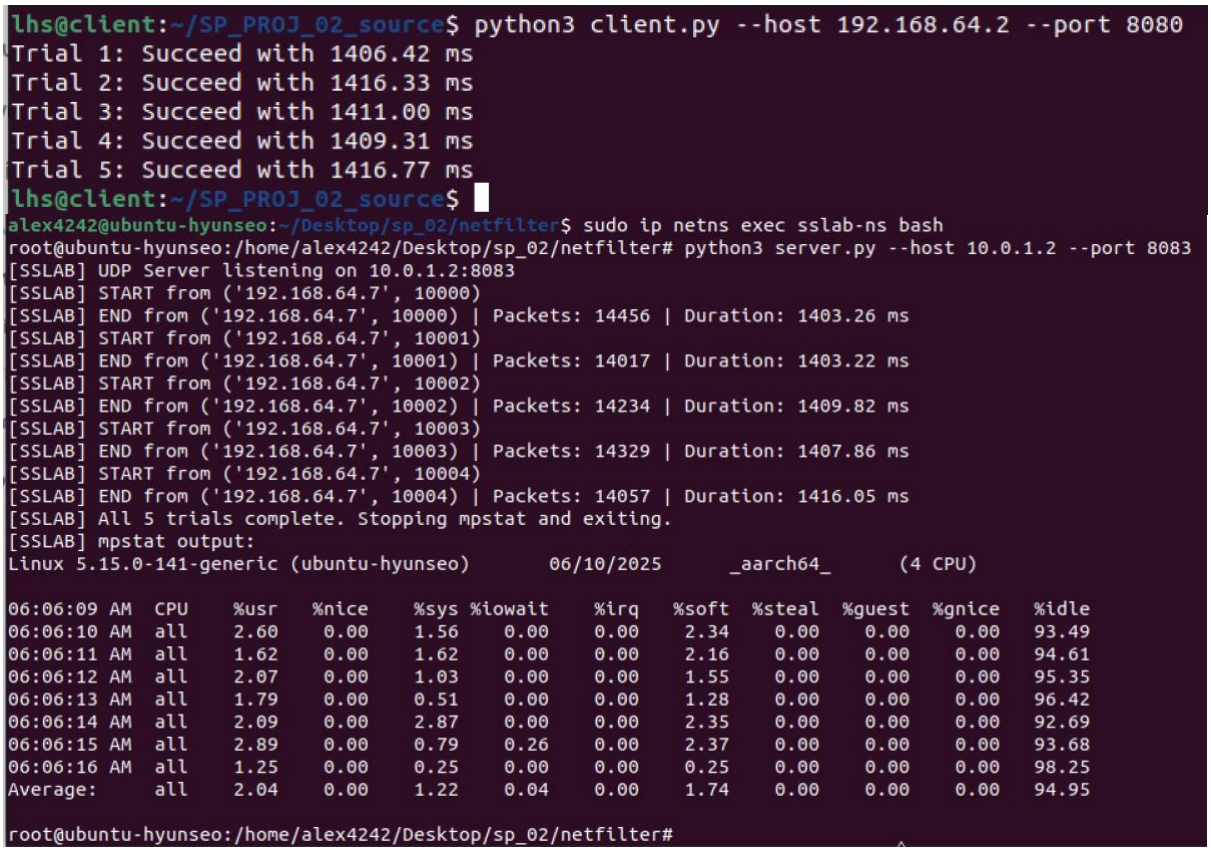
본 실험에서는 VM1(클라이언트)이 UDP 패킷을 VM2(서버)의 루트 네임스페이스 IP (192.168.64.2)의 8080 포트로 전송한다. VM2는 이 패킷을 수신한 뒤, Netfilter 또는 eBPF/XDP 프로그램을 통해 패킷의 목적지 정보를 수정하고, 네트워크 네임스페이스 내부에 위치한 서버(10.0.1.2:8083)로 리다이렉션한다.

2) 측정 요소

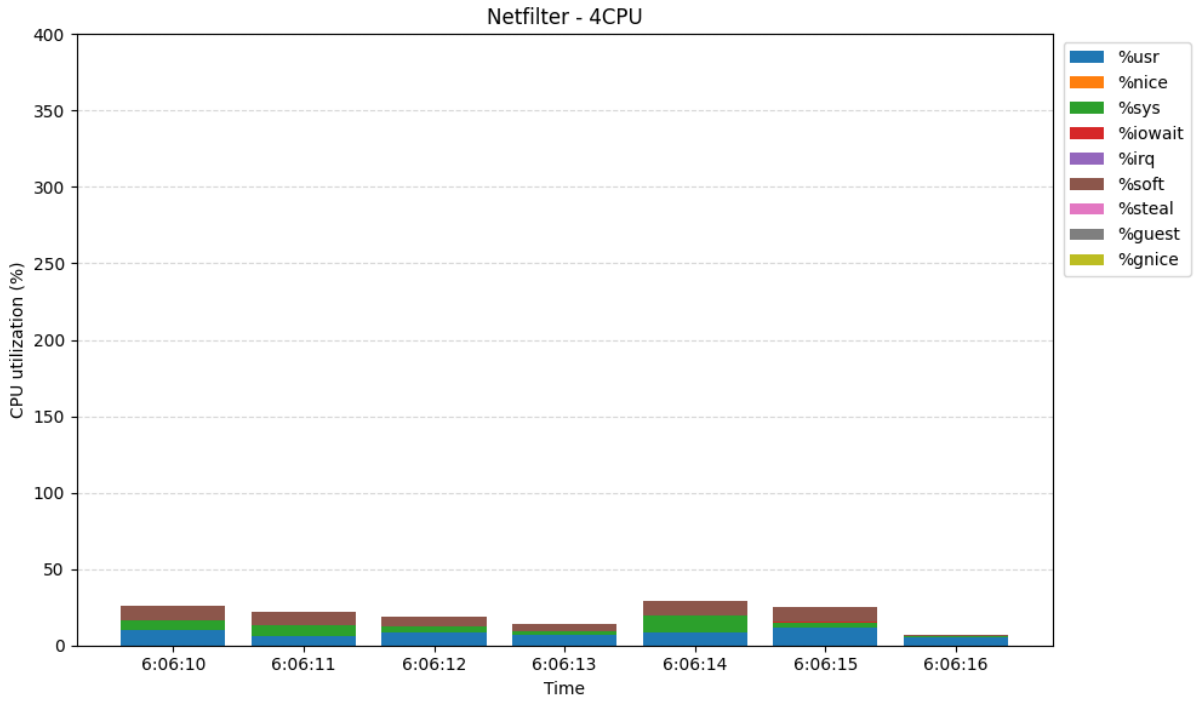
총 5번의 실험을 반복하면서 로그를 통해 패킷 리다이렉션을 확인하고, mpstat을 통해 전체 CPU 사용률을 측정한다.

4. 실험 결과 및 분석

1) Nefilter



```
alex4242@ubuntu-hyunseo:~/Desktop/sp_02/netfilter$ sudo dmesg | grep UDP
[  101.651835] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651835] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651836] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651836] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651837] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651853] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651854] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651854] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651855] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651855] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651856] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651856] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651857] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651857] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651858] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651858] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651858] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.651859] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.651859] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.653079] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.653120] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.653136] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.653164] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.653185] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.653253] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.653282] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.653305] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
[  101.653403] FORWARDING: UDP:192.168.64.7:10004;192.168.64.2:8080
[  101.653416] MODIFIED:   UDP:192.168.64.7:10004;10.0.1.2:8083
```



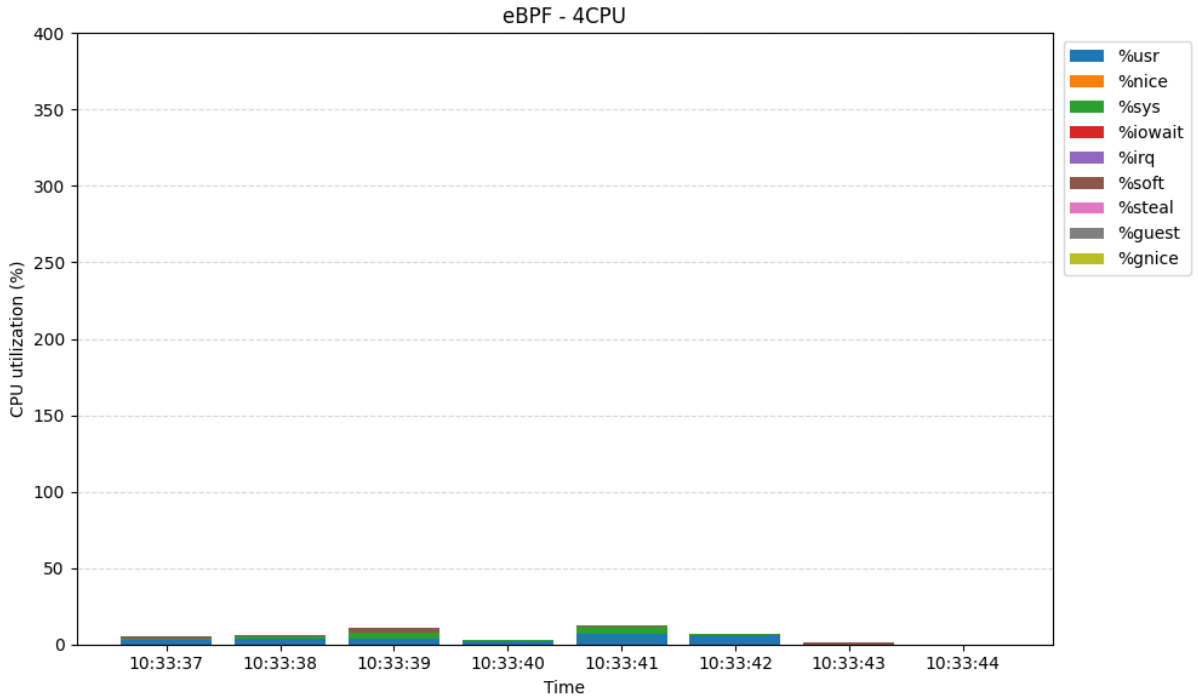
%usr, $sys, $soft 모두 합쳐 약 20%(전체 400%)의 높은 사용률을 보여주고 있다. 이는 XDP 특성상 커널 네트워크 스택 처리가 거의 생략되었기 때문으로 보인다.

2) XDP(eBPF)

```
lhs@client:~/SP_PROJ_02_source$ python3 client.py --host 192.168.64.2 --port 8080
Trial 1: Succeed with 1395.76 ms
Trial 2: Succeed with 1421.98 ms
Trial 3: Succeed with 1416.27 ms
Trial 4: Succeed with 1414.98 ms
Trial 5: Succeed with 1414.15 ms
```

```
root@ubuntu-hyunseo:/home/alex4242/Desktop/sp_02/eBPF# python3 server.py --host 10.0.1.2 --port 8083
[SSLAB] UDP Server listening on 10.0.1.2:8083
[SSLAB] START from ('192.168.64.7', 10000)
[SSLAB] END from ('192.168.64.7', 10000) | Packets: 1 | Duration: 1394.35 ms
[SSLAB] START from ('192.168.64.7', 10001)
[SSLAB] END from ('192.168.64.7', 10001) | Packets: 1 | Duration: 1420.31 ms
[SSLAB] START from ('192.168.64.7', 10002)
[SSLAB] END from ('192.168.64.7', 10002) | Packets: 1 | Duration: 1414.06 ms
[SSLAB] START from ('192.168.64.7', 10003)
[SSLAB] END from ('192.168.64.7', 10003) | Packets: 1 | Duration: 1412.18 ms
[SSLAB] START from ('192.168.64.7', 10004)
[SSLAB] END from ('192.168.64.7', 10004) | Packets: 1 | Duration: 1412.45 ms
[SSLAB] All 5 trials complete. Stopping mpstat and exiting.
[SSLAB] mpstat output:
Linux 5.15.0-141-generic (ubuntu-hyunseo)       06/19/2025      _aarch64_      (4 CPU)

10:33:37 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
10:33:38 AM  all    0.76    0.00    0.25    0.00    0.00    0.25    0.00    0.00    0.00   98.73
10:33:39 AM  all    1.03    0.00    0.26    0.00    0.00    0.26    0.00    0.00    0.00   98.45
10:33:40 AM  all    1.01    0.00    1.01    0.00    0.00    0.76    0.00    0.00    0.00   97.22
10:33:41 AM  all    0.51    0.00    0.25    0.00    0.00    0.00    0.00    0.00    0.00   99.24
10:33:42 AM  all    1.81    0.00    1.04    0.00    0.00    0.26    0.00    0.00    0.00   96.89
10:33:43 AM  all    1.55    0.00    0.26    0.00    0.00    0.00    0.00    0.00    0.00   98.19
10:33:44 AM  all    0.00    0.00    0.00    0.00    0.00    0.25    0.00    0.00    0.00   99.75
Average:     all    0.95    0.00    0.44    0.00    0.00    0.26    0.00    0.00    0.00   98.36
```



%usr, $sys, $soft 모두 합쳐 약 6%(전체 400%)의 낮은 사용률을 보여주고 있다. 이는 Netfilter가 네트워크 스택 상위에서 동작하기 때문에 그런 것으로 보인다.

3) 비교 분석

Netfilter는 커널의 네트워크 스택 상위 계층 (IP 레이어 이후)에서 동작하기 때문에, 수신된 패킷이 NIC → 드라이버 → L2 → L3 → Netfilter → L4를 거친 후에야 처리된다. 이 과정에서 여러 커널 시스템에 의한 오버헤드가 발생하게 되고 패킷을 처리하는 과정에서 CPU 리소스를 더 많이 사용하게 된다.

반면, XDP(eBPF)는 NIC 드라이버 수준에서 동작하는 초고속 패킷 처리 경로로, 패킷이 네트워크 스택에 진입하기도 전에 필터링이나 리다이렉션을 수행한다. 이로 인해 불필요한 경로를 생략할 수 있어, CPU 사용률이 획기적으로 낮아지고 처리 지연도 크게 줄어든다.

두 그래프를 비교해보면 알 수 있듯이 실제 실험 결과 또한 Netfilter의 CPU 사용률이 XDP보다 3배정도 더 높게 측정되었다. 이는 XDP(eBPF)를 사용하여 리다이렉션 할 시 Netfilter 대비 3배 이상의 효율성을 보인다고 해석할 수 있다.

5. 어려웠던 점

1) 맥북 UTM 네트워크 문제

맥북 UTM 기본 네트워크 카드로 작동 시 패킷이 합쳐지는 문제가 발생하였다. 이 후 네트워크 카드를 intel e1000으로 변경하여 문제를 해결하였다.

2) mpstat 미출력

mpstat이 출력되기도 전에 프로그램이 종료되는 문제가 발생하였다. 이후 client.py에 sleep 문을 추가하여 문제를 해결하였다.

2) vmlinux.h

vmlinux 관련 config 파일을 수정하지 않고 빌드한 커스텀 커널을 사용하여 vmlinux 사용에 어려움이 있었다. 이후 커스텀 커널이 아닌 기본 커널을 사용하여 문제를 해결하였다.