시스템 프로그래밍

2차 과제

Network Packet Filtering

group 15

컴퓨터학과 2013210107 김정수

컴퓨터학과 2014210075 이한얼

제출 : 2016.12.22

사용 freeday – 0일

환경

가상머신 : Oracle VM VirtualBox

운영체제 : Ubuntu 16.04 LTS

커널 : linux-4.4.1

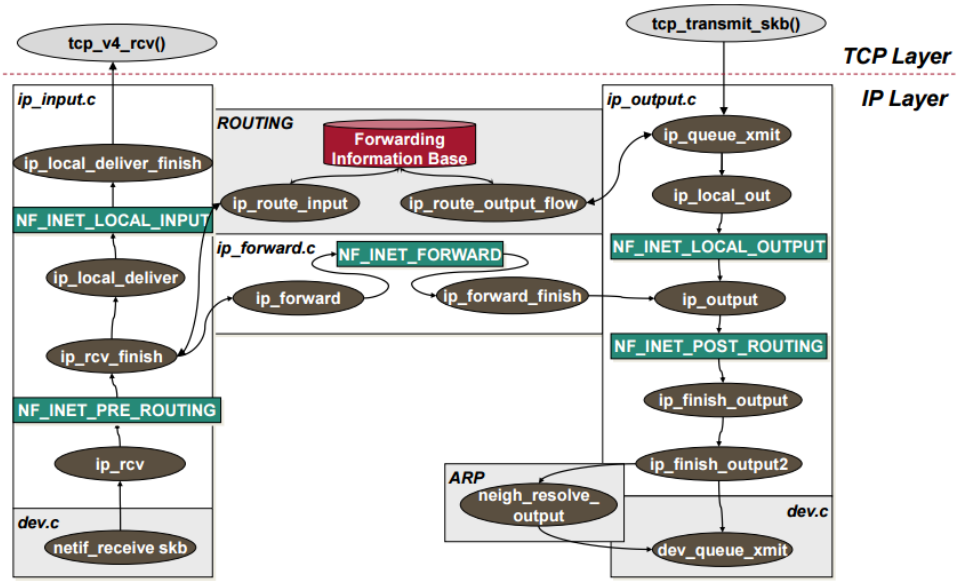
Netfilter 및 Hooking

Packet의 전송 및 수신은 크게 다음과 같은 4가지 Layer로 이루어진다.

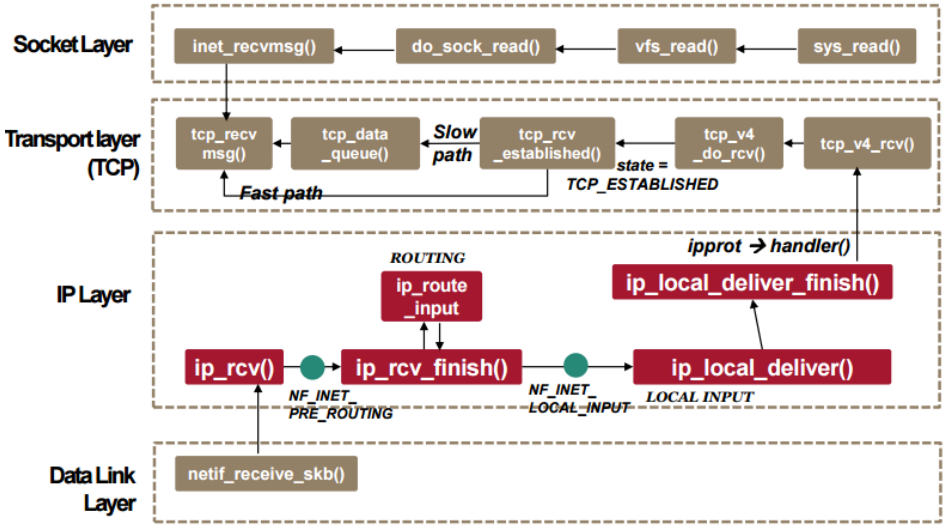
[Socket Layer] – [Transport Layer] – [IP Layer] – [Data Link Layer]

물론, 네트워크 통신 방법에 따라 그 명칭이나 사용법은 달라질 수 있으나, 가장 기본적으로 TCP/IP 소켓 프로그래밍을 기준으로 설명하도록 하겠다.
Netfilter는 리눅스 커널 내부의 네트워크 관련 프레임워크이다. 다양한 네트워크 관련 연산을 핸들러 형태로 구현할 수 있도록 제공한 것이 바로 hook이다.
위에서 설명한 Layer 중 IP Layer에서 hook함수를 등록하여 packet이 이 hook함수를 거치게 만들어준다.

IP 단에서 hook함수를 등록할 수 있는 지점은 총 다섯 곳이다.
- NF_INET_LOCAL_INPUT / NF_INET_PRE_ROUTING / NF_INET_FORWARD
/ NF_INET_LOCAL_OUTPUT / NF_INET_POST_ROUTING



그림의 왼쪽 부분은 수신을 하는 과정이고, 오른쪽 부분은 전송하는 과정이다.
이번 과제는 packet을 수신하는 과정에서 filtering을 하기 때문에 왼쪽 부분을 좀 더 자세히 보겠다.



IP Layer에서 ip_rcv()를 통하여 받아온 packet은 TCP Layer까지 가는데 두 번의 hook을 지날 수 있다. 이번 과제에서는 단순히 연결된 IP 및 PORT만 확인하여 총 데이터량에 따라 packet을 DROP할 것이기 때문에 굳이 NF_INET_LOCAL_INPUT이 있는 곳까지 불필요한 처리를 할 필요없이 NF_INET_PRE_ROUTING 에서 hook 함수를 등록하여 filtering 하여 주는 것이 가장 적절해 보인다.

커널 레벨 네트워킹 코드 분석 – lxr 활용

- **전송 과정**

각 과정은 전체를 가져오지 않고 중요한 부분만 가져와서 설명하도록 하겠다.

1) 패킷의 첫 전송은 /fs/read_write.c에 위치한
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count) 부터 시작한다.

```
605         if (f.file) {
606                 loff_t pos = file_pos_read(f.file);
607                 ret = vfs_write(f.file, buf, count, &pos);
```

여기서 f는 받아온 file descriptor인데, 이 fd의 file과 buf, count 등을 vfs_write 함수의 호출과 함께 그 인자로 넘겨준다.

2) /fs/read_write.c 에 위치한
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)

vfs_write를 읽어 내려가다 보면, 예외처리 후 return값을 위해 다시 __vfs_write를 불러냄을 알 수 있다.

```
560                 ret = __vfs_write(file, buf, count, pos);
```

따라서 이번에는 __vfs_write를 따라가 본다. 이 역시 같은 /fs/read_write.c에 위치해 있다.

```
509         if (file->f_op->write)
510                 return file->f_op->write(file, p, count, pos);
511         else if (file->f_op->write_iter)
512                 return new_sync_write(file, p, count, pos);
```

위 그림과 같이 우리는 file자체의 operation에 write가 없는 경우, write_iter의 존재를 확인하게 되고 다시 new_sync_write 함수를 호출함을 볼 수 있다. 따라서 이번에는 new_sync_write 함수를 따라가보자.

이 함수 안에는 kiocb라는 새로운 구조체가 등장하게 된다. 이는 애초에 write_iter가 kiocb 구조체를 인자로 사용하기 때문이다.

```
1702         ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
```

Kiocb 구조체를 살펴보면 이 또한 file 구조체를 ki_filp라는 이름으로 가지고 있음을 알 수 있다.

```
326 struct kiocb {
327         struct file             *ki_filp;
328         loff_t                  ki_pos;
```

다시, 원래 함수였던 new_sync_write로 돌아와보면,

```
488 static ssize_t new_sync_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos)
```

...

```
495         init_sync_kiocb(&kiocb, filp);
```

받아온 file을 kiocb에 적용 시키고

```
499         ret = filp->f_op->write_iter(&kiocb, &iter);
```

write_iter의 인자로 넘겨준다.

3) 위 2번에서는 vfs_write를 통하여 받아온 파일을 kiocb 형태로 바꾸어 주고, 이 구조체를 file의 operation의 write_iter로 넘겨줌을 알 수 있었다.
만약 이 file operation이 socket_file_ops라면 (우리는 socket을 쓰기 때문)
다음을 확인할 수 있다. (/net/socket.c 에 위치)

```
140 static const struct file_operations socket_file_ops = {
141         .owner =        THIS_MODULE,
142         .llseek =       no_llseek,
143         .read_iter =    sock_read_iter,
144         .write_iter =   sock_write_iter,
```

write_iter 가 sock_write_iter임을 보아 kiocb로 변환된 파일이 이 함수에서 쓰임을 알 수 있다.
같은 /net/socket.c에 위치한 sock_write_iter 함수를 살펴보자.

```
812 static ssize_t sock_write_iter(struct kiocb *iocb, struct iov_iter *from)
813 {
814         struct file *file = iocb->ki_filp;
815         struct socket *sock = file->private_data;
```

받아온 파일의 포인터를 가져오고, 파일 구조체의 private_data 포인터로 소켓 구조체에 접근한다.
이하 메시지 처리 이후 sock_sendmsg를 호출하게 된다.

```
829         res = sock_sendmsg(sock, &msg);
```

이는 다시, sock_sendmsg_nosec을 호출하게 되고,

```
626 int sock_sendmsg(struct socket *sock, struct msghdr *msg)
627 {
        ...
631         return err ?: sock_sendmsg_nosec(sock, msg);
```

결론적으로 socket의 proto_ops 내의 sendmsg를 호출하게 된다.

```
619 static inline int sock_sendmsg_nosec(struct socket *sock, struct msghdr *msg)
620 {
621         int ret = sock->ops->sendmsg(sock, msg, msg_data_left(msg));
```

우리는 이제, socket을 살펴볼 필요가 있다. 결론적으로는 socket의 sendmsg가 호출되었기 때문이다.

socket 구조체의 proto_ops는 tcp이기 때문에 다음과 같다. (/net/ipv4/af_inet.c에 위치)

```
898 const struct proto_ops inet_stream_ops = {
899         .family            = PF_INET,
900         .owner             = THIS_MODULE,
...
913         .sendmsg           = inet_sendmsg,
```

결론적으로 inet_sendmsg 가 호출되게 된다.

socket과 메시지를 넘겨받은 inet_sendmsg는 socket내의 proto에 속한 sendmsg를 호출하게된다.

```
723 int inet_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
724 {
725         struct sock *sk = sock->sk;
...
734         return sk->sk_prot->sendmsg(sk, msg, size);
```

4) 이 전 단계에서 마지막으로 호출한 함수는 socket 내의 proto의 sendmsg이다.
tcp를 사용하기 때문에 proto는 tcp_port가 될 것이고, 이를 직접 확인해보면 다음과 같다. (/net/ipv4/tcp_ipv4.c 에 위치)

```
2362 struct proto tcp_prot = {
2363         .name                   = "TCP",
2364         .owner                  = THIS_MODULE,
...
2376         .sendmsg                = tcp_sendmsg,
...
```

위에서 알 수 있듯, tcp_sendmsg를 호출하게 된다.

/net/ipv4/tcp.c 에 위치한 tcp_sendmsg는 다음과 같다.

```
1097 int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
```

이로써 우리는 Transport Layer로 넘어오게 된다.

5) Tcp_sendmsg에서는 user space에서 kernel space로의 payload의 복사가 일어난다.

```
1097 int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
1098 {
1099         struct tcp_sock *tp = tcp_sk(sk);
1100         struct sk_buff *skb;
```

이 sk_buff에 최대한 패킷을 채워 보내는 시도를 하며, 새로운 skb가 필요한 경우 새로 할당하게 된다. 이 때, 최대한의 사이즈를 구하기 위하여 tcp_send_mss 함수를 호출한다.

```
1163          mss_now = tcp_send_mss(sk, &size_goal, flags);
```

While 문을 통하여,

```
1171         while (msg_data_left(msg)) {
1172                 int copy = 0;
1173                 int max = size_goal;
```

메시지의 data가 남아 있지 않을 때까지 sk_buff를 생성 및 복사하여 준다.

```
1197                         skb = sk_stream_alloc_skb(sk,
1198                                                   select_size(sk, sg, first_skb),
1199                                                   sk->sk_allocation,
1200                                                   first_skb);
```

다음을 통해 copy하게 된다.

```
1231                         err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
```

마지막으로 copy가 성공했을 경우 해당 sock을 push하게 된다.

```
1314         if (copied)
1315                 tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
```

- **수신 과정**

1) 수신 과정 역시 전송과 비슷하게 /fs/read_write.c에 위치한 system call 으로부터 시작한다.

```
584 SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
585 {
586         struct fd f = fdget_pos(fd);
```

이 곳에서 file과 buf 및 count를 vfs_read로 넘겨준다.

```
589         if (f.file) {
590                 loff_t pos = file_pos_read(f.file);
591                 ret = vfs_read(f.file, buf, count, &pos);
```

2) /fs/read_write.c에 위치한 vfs_read에서는 다시 한번 직접 수행하는 __vfs_read를 호출한다.

```
460 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
461 {
        ...
475                 ret = __vfs_read(file, buf, count, pos);
```

여기도 wrtie와 마찬가지로 new_sync_read에서 전달 받은 file을 kiocb 구조체로 변환해준 다음 file operation socket_file_ops의 sock_read_iter의 인자로 넘겨준다.

```
448 ssize_t __vfs_read(struct file *file, char __user *buf, size_t count,
449                    loff_t *pos)
450 {
451         if (file->f_op->read)
452                 return file->f_op->read(file, buf, count, pos);
453         else if (file->f_op->read_iter)
454                 return new_sync_read(file, buf, count, pos);
```

...

```
431 static ssize_t new_sync_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos)
432 {
442         ret = filp->f_op->read_iter(&kiocb, &iter);
```

3) 이제 넘겨 받은 kiocb를 사용할 sock_read_iter 함수를 살펴보자. (/net/socket.c 에 위치)

```
790 static ssize_t sock_read_iter(struct kiocb *iocb, struct iov_iter *to)
791 {
792         struct file *file = iocb->ki_filp;
793         struct socket *sock = file->private_data;
794         struct msghdr msg = {.msg_iter = *to,
795                              .msg_iocb = iocb};
```

이 함수는 socket과 메시지를 sock_recvmsg로 넘겨주고, (/net/socket.c 에 위치)

```
807         res = sock_recvmsg(sock, &msg, msg.msg_flags);
```

sock_recvmsg 함수를 따라가보면,

```
726 int sock_recvmsg(struct socket *sock, struct msghdr *msg, int flags)
727 {
        ...
730         return err ?: sock_recvmsg_nosec(sock, msg, flags);
```

sock_recvmsg_nosec 함수로 그대로 데이터들을 넘겨줌을 알 수 있다.

여기서 socket의 proto_ops 내의 recvmsg 함수를 호출하게 된다.

```
720 static inline int sock_recvmsg_nosec(struct socket *sock, struct msghdr *msg,
721                                       int flags)
722 {
723         return sock->ops->recvmsg(sock, msg, msg_data_left(msg), flags);
```

4) 전송 때와 마찬가지로 이제 socket 내의 proto_ops의 recvmsg 함수를 찾아가 보도록 하자. socket 구조체의 proto_ops는 tcp이기 때문에 다음과 같다. (/net/ipv4/af_inet.c에 위치)

```
898 const struct proto_ops inet_stream_ops = {
899         .family            = PF_INET,
900         .owner             = THIS_MODULE,
...
914         .recvmsg           = inet_recvmsg,
```

inet_recvmsg를 호출함을 알 수 있고,

```
756 int inet_recvmsg(struct socket *sock, struct msghdr *msg, size_t size,
757                   int flags)
758 {
759         struct sock *sk = sock->sk;
```

이는 다시 socket내의 proto에 속한 recvmsg를 호출함을 알 수 있다.

```
765         err = sk->sk_prot->recvmsg(sk, msg, size, flags & MSG_DONTWAIT,
766                                    flags & ~MSG_DONTWAIT, &addr_len);
```

tcp를 사용하기 때문에 proto는 tcp_port가 될 것이고, 이를 직접 확인해보면 다음과 같다. (/net/ipv4/tcp_ipv4.c 에 위치)

```
2362 struct proto tcp_prot = {
2363         .name                   = "TCP",
2364         .owner                  = THIS_MODULE,
...
2375         .recvmsg                = tcp_recvmsg,
```

최종적으로 tcp_recvmsg를 호출함으로써 Transport Layer로 넘어가게 된다.

5) tcp_recvmsg에서는 우선 sock을 ucopy 구조체를 가지고 있는 tcp_sock으로 확장하여 처리한다.

```
1609 int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
1610                 int flags, int *addr_len)
1611 {
1612         struct tcp_sock *tp = tcp_sk(sk);
1613         int copied = 0;
```

만일 busy_loop을 사용할 수 있을 때이고(non-block), skb_queue가 비어있다면 읽을 데이터가 없는 것이므로 busy_loop을 통해 wait하게 된다.

```
1627         if (sk_can_busy_loop(sk) && skb_queue_empty(&sk->sk_receive_queue) &&
1628             (sk->sk_state == TCP_ESTABLISHED))
1629                 sk_busy_loop(sk, nonblock);
```

이후, lock_sock()을 통해 해당 sock을 lock해주고 자신이 사용함을 표시한다.

```
1631            lock_sock(sk);
```

만일, TCP_LISTEN을 받게 되는 경우 바로 sock을 해제하면서 함수가 끝난다.

```
1634            if (sk->sk_state == TCP_LISTEN)
1635                    goto out;
```

그렇지 않으면, flag를 확인하여 긴급한 urgent data면 특별한 처리를 하게된다.

이는 바로 recv_urg로 이동하여 tcp_recv_urg함수 처리 후, 해당 함수에서 나가게된다.

```
1639            /* Urgent data needs to be handled specially. */
1640            if (flags & MSG_OOB)
1641                    goto recv_urg;
```

...

```
1937 recv_urg:
1938            err = tcp_recv_urg(sk, msg, len, flags);
1939            goto out;
```

이후 MSG_PEEK에 대한 처리도 있는데, 이는 입력버퍼를 검사하는 옵션이므로 여기서는 패스하도록 하겠다.

이후 읽어야 할 데이터의 양을 target에 저장한다.

```
1618            int target;             /* Read at least this many bytes */
```

이 값은 sock_rcvlowat함수를 통해 가져오게 된다.

```
1664            target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
```

//

```
2078 static inline int sock_rcvlowat(const struct sock *sk, int waitall, int len)
2079 {
2080         return (waitall ? len : min_t(int, sk->sk_rcvlowat, len)) ? : 1;
2081 }
```

이제 전체 데이터의 길이 len이 0이 될 때까지 While문을 돌리게 된다.

```
1666            do {
                    ...
1902            } while (len > 0);
```

그 사이에는 urgent data에 대한 처리나

```
1669                    /* Are we at urgent data? Stop if we have
1670                    if (tp->urg_data && tp->urg_seq == *seq) {
1671                            if (copied)
```

각 queue들에 대한 처리를 볼 수 있다.

receive queue

```
1681                    last = skb_peek_tail(&sk->sk_receive_queue);
1682                    skb_queue_walk(&sk->sk_receive_queue, skb) {
1683                            last = skb;
```

Prequeue

```
1794                            if (!skb_queue_empty(&tp->ucopy.prequeue))
1795                                    goto do_prequeue;
```

...

```
1822 do_prequeue:
1823                                    tcp_prequeue_process(sk);
1824
```

또한 읽어들인 data에 대해서 읽은 copy는 증가시키고 읽을 len은 감소시키는 처리도한다.

```
1876                    *seq += used;
1877                    copied += used;
1878                    len -= used;
```

목표량이 수행된 양보다 적을 경우 sleep하지 않고 backlog queue를 처리하기도 한다.

```
1800                    if (copied >= target) {
1801                            /* Do not sleep, just process backlog. */
1802                            release_sock(sk);
1803                            lock_sock(sk);
1804                    } else {
1805                            sk_wait_data(sk, &timeo, last);
1806                    }
```

작성한 소스코드에 대한 설명

- **server.c – 제공 / client.c – 기존 warm_up 과제 그대로 사용**

- **aa.c – packet_limit 변수를 변경할 proc 및 hook 함수 등록 관련 LKM 소스**

- **<header>**

  proc을 사용하기 위한 헤더 + netfilter를 사용하기 위한 헤더 + tcp/ip 관련 헤더

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/proc_fs.h>
5 #include <linux/netfilter.h>
6 #include <linux/netfilter_ipv4.h>
7
8 #include <linux/ip.h>   // ip header
9 #include <linux/tcp.h>  // tcp header
```

- **<define>**

  proc에 생성할 폴더명/파일명

  미리 정해둔 연결하게될 서버 PORT 들 (client 파일 역시 동일한 PORT로 코딩되었다.)

```
11 #define PROC_DIRNAME "myproc"
12 #define PROC_FILENAME "myproc"
13
14 #define PORT1 1111
15 #define PORT2 2222
16 #define PORT3 3333
17 #define PORT4 4444
18 #define PORT5 5555
```

- **<미리 선언된 전역변수들>**

  packet_limit : netfilter hook함수에 의해 제한할 총 데이터량.

  sport/dport : 해당 패킷에 들어있는 source/destination port를 받아올 변수

  server_port[5] : 미리 정해둔 연결하게될 각각의 서버 port를 저장할 변수

  data_len[5] : 각각의 서버 port에 대한 현재까지 들어온 총 데이터량

  proc_dir/proc_file : proc 관련 directory/file 변수

```
20 unsigned int packet_limit;
21 unsigned int sport;     // source port
22 unsigned int dport;     // destination port
23
24 unsigned int server_port[5];    // each server port.
25 unsigned int data_len[5];       // data lenth for each port
26
27
28 static struct proc_dir_entry *proc_dir;
29 static struct proc_dir_entry *proc_file;
```

- **<my_hook_fn>**

  등록되는 hook함수 및 sk_buff의 tcp/ip 헤더를 받아올 변수

```
31 static unsigned int my_hook_fn(void *priv,
32         struct sk_buff *skb, const struct nf_hook_state *state) {
33
34
35     struct iphdr  *ih; // ip header
36     struct tcphdr *th; // tcp header
```

  sk_buff가 존재하면 ip_hdr() 함수를 이용하여 skb의 ip헤더를 받아오고,

  받아온 ip헤더에 문제가 없으면 계속해서 진행한다.

```
38     if(!skb)
39         return NF_DROP;
40
41     ih = ip_hdr(skb);
42
43     if(!ih)
44         return NF_DROP;
```

ip헤더를 통하여 프로토콜이 TCP이면 계속해서 진행한다.
tcp_hdr() 함수를 통하여 skb의 tcp 헤더를 받아온다.
tcp헤더의 source와 dest를 htons를 이용하여 변환함으로써
sport/dport에 각각 출발/목적지 port 변호를 받아온다.
우리가 확인할 값은 서버쪽 포트로 확인할 것이기 때문에 sport이다.

```
46      // if TCP, go on
47      if(ih->protocol == IPPROTO_TCP) {
48
49          int i;
50
51          th = tcp_hdr(skb);
52          // get source port and destination port
53          sport = htons((unsigned short int) th->source);
54          dport = htons((unsigned short int) th->dest);
```

for문을 통하여 skb가 server의 어떤 PORT에서 왔는지 확인 후,
해당 PORT의 데이터 총량이 packet_limit을 초과한다면 패당 패킷을 drop시킨다.
만약 패킷을 받게될 경우 받은 skb의 데이터량 만큼 해당 PORT 데이터 총량 변수를 증가시킨다.

```
56          for(i=0; i<5; i++) {
57              if(sport == server_port[i]) {
58                  // print previous information.
59                  printk(KERN_INFO "##SPORT : %u, ##Length : %u\n",
60                          server_port[i], data_len[i]);
61
62                  // DROP packet, if over limit
63                  if(data_len[i] > packet_limit)
64                      return NF_DROP;
65
66                  // skb->len = skb->tail - skb->data
67                  data_len[i] += skb->len;
68              }
69          }
```

위 과정에서 DROP되지 않았다면 NF_ACCEPT를 통하여 패킷을 받는다.

```
72      return NF_ACCEPT;
```

- **<my_nf_ops>**

등록할 hook에 대한 설정 정보이다.
.hook : 처리할 hook의 내용은 my_hook_fn 으로 정함.

.hooknum : 해당 hook이 들어갈 지점. 위 이론 설명에서 하였듯,
NF_INET_PRE_ROUTING과 NF_INET_LOCAL_IN 둘 다 가능하지만, 전자를 사용하였다.

.pf : IPv4를 사용한다 명시할 PF_INET
.priority : 해당 훅에 대한 우선순위. NF_IP_PRI_FIRST를 통하여 최우선으로 설정.

```
81 static struct nf_hook_ops my_nf_ops = {
82     .hook = my_hook_fn, // call my hook function
83     .hooknum = NF_INET_PRE_ROUTING, // also, 'NF_INET_LOCAL_IN' is ok.
84     .pf = PF_INET,      // IPv4
85     .priority = NF_IP_PRI_FIRST, // set highest priority
86 };
```

- **<my_write>**

  proc/myproc/myproc 파일에 packet_limit 값을 쓸 때, 처리될 함수.

  simple_strtol을 통하여 버퍼에 들어온 string값을 숫자형태로 바꾸어 packet_limit에 저장한다. 완료 후, 커널 메시지 출력.

```
89 static ssize_t my_write(struct file *file, const char __user *user_buffer,
90         size_t count, loff_t *ppos) {
91
92     printk(KERN_INFO "SImple module write!\n");
93
94     // update packet_limit by writing.
95     packet_limit = simple_strtol(user_buffer, NULL, 10);
96
97     printk(KERN_INFO "##packet limit : %u\n", packet_limit);
98
99     return count;
00 }
```

- **<myproc_fops>**

  proc에 대한 설정 정보.

  .write : write 처리 함수로 my_write를 등록.

```
102 static const struct file_operations myproc_fops = {
103     .owner = THIS_MODULE,
104     .write = my_write,
105 };
```

- **<simple_init>**

  모듈이 처음에 등록될 때 실행된다.

  각 포트의 받은 데이터 총량 및 server_port 값 초기화.

```
107 static int __init simple_init(void) {
108
109     // initiate port and each port data lenght.
110     int i;
111     for (i = 0; i < 5; i++) {
112         data_len[i] = 0;
113     }
114     server_port[0] = PORT1;
115     server_port[1] = PORT2;
116     server_port[2] = PORT3;
117     server_port[3] = PORT4;
118     server_port[4] = PORT5;
119
120     printk(KERN_INFO "SIMple module init!\n");
```

이어서 처음 packet_limit은 10000으로 설정.

proc을 만들어 준다.

```
122     // initiate packet_limit.
123     packet_limit = 10000;
124     printk(KERN_INFO "##packet limit = %u\n", packet_limit);
125
126     // make proc dir/file
127     proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
128     proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &myproc_fops);
129     printk(KERN_INFO "##Proc file make success\n");
```

마지막으로 nf_register_hook을 통하여 hook함수를 netfilter에 등록.

```
131     // register hook function
132     nf_register_hook(&my_nf_ops);
133     printk(KERN_INFO "##nf_hook registered\n");
```

- **<simple_exit>**

  모듈을 내릴 때 실행된다.

  remove_proc_entry를 통하여 proc폴더/파일을 삭제

  nf_unregister_hook을 통하여 등록했던 hook함수 제거

```
138 static void __exit simple_exit(void) {
139     printk(KERN_INFO "simple module exit!\n");
140     remove_proc_entry(PROC_FILENAME, proc_dir); // remove file
141     remove_proc_entry(PROC_DIRNAME, NULL);      // remove directory
142     nf_unregister_hook(&my_nf_ops);
143 }
```

실험 방법에 대한 설명 및 로그파일 결과 분석, 그래프

서버, 클라이언트 ip주소를 설정.

서버 :

```
$ sudo ifconfig enp0s3 192.168.56.103
```

클라이언트 :

```
sudo ifconfig enp0s3 192.168.56.104
```

LKM 등록 :

```
SP_02main/net$ sudo insmod aa.ko
```

커널 메시지 확인

```
406704] SIMple module init!
406709] ##packet limit = 10000
406718] ##Proc file make success
406993] ##nf_hook registered
```

packet limit : 500000으로 설정.

```
alBox:/proc# echo 500000 > myproc/myproc
```

커널 메시지 확인

```
4024] SImple module write!
4032] ##packet limit : 500000
```

서버 쪽 포트 설정 및 실행:

```
syspro@oslab:~$ ./server
Type Server Ports(format:<Port1> <Port2> <Port3> <Port4> <Port5> :
1111 2222 3333 4444 5555
```

클라이언트 실행

```
osta@osta-VirtualBox:~/SP_02$ ./client
port :: 5555
port :: 3333
port :: 2222
port :: 1111
port :: 4444
```

패킷이 500000에서 멈췄는지 커널을 통해 확인 후

```
446] ##SPORT : 2222, ##Length : 504846
294] ##SPORT : 3333, ##Length : 505839
344] ##SPORT : 5555, ##Length : 500837
452] ##SPORT : 1111, ##Length : 503759
256] ##SPORT : 4444, ##Length : 500467
500] ##SPORT : 2222, ##Length : 504846
323] ##SPORT : 3333, ##Length : 505339
```

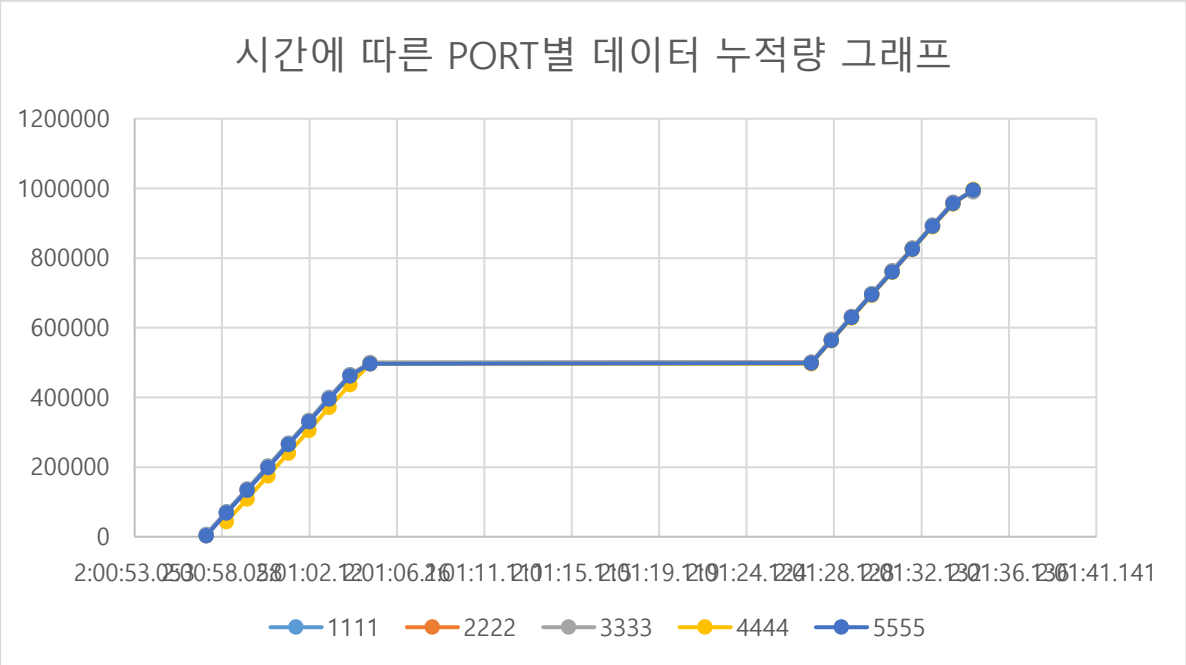
packet limit : 1000000으로 재설정

```
alBox:/proc# echo 1000000 > myproc/myproc
```

더 받았는지 확인

```
##SPORT : 4444, ##Length : 1005203
##SPORT : 1111, ##Length : 1002331
##SPORT : 2222, ##Length : 1002390
##SPORT : 3333, ##Length : 1000559
##SPORT : 4444, ##Length : 1005203
##SPORT : 5555, ##Length : 1001455
```

로그파일 결과 시간에 따른 데이터 누적 그래프



로그파일의 결과는 확연하게 잘 나타나 있다. 다만 port별 누적 경향이 똑같아서 그래프상으로는 심하게 겹치게 나온 것이 아쉽긴 했다.

packet_limit을 처음에 500,000으로 맞춰놓았을 때, 모든 port들이 더 이상 packet을 받지 않음을 볼 수 있다. 또한, 그 후에 다시 packet_limit을 1,000,000으로 바꾸자 다시 packet 수신이 시작됨을 볼 수 있다. 이는 누적량이 1,000,000을 넘는 순간 다시 멈추게 된다.

애초에 그래프에 사용한 데이터는 시간 별로 완전한 데이터를 받을 때마다의 기록이지만,

만약 LKM에서 커널 메시지로 기록한 쪼개진 packet이 들어올 때마다의 데이터를 이용하면 더 자세히 그래프를 그릴 수 있을 것 같지만, 우선은 안내에 따라 수신 프로그램의 로그파일을 이용하였다.

모든 port의 누적 경향이 같은 이유는 로그파일을 확인해보니, 모든 포트가 계속해서 일정하게 최대의 크기인 65535의 크기로 자료를 받아오기 때문임을 알 수 있었다. (아마도 서버에서 그렇게 전송하므로)

결과적으로는 과제의 목표는 확실히 이룬 것으로 보인다.

과제 수행 시 어려웠던 부분과 해결 방법

생각보다 마지막 과제는 어렵지 않았던 것 같다. 기말고사의 내용과 겹쳤기 때문에 이론상으로 이해하고 과제를 수행하니, 따라가기가 쉬웠다. 다만, 커널 레벨 네트워크 코드를 분석하는 과정에서 너무나 자료가 방대하기 때문에 하나하나 찾아보고 공부하는 것이 복잡하지만 재미있었던 것 같다.
대부분의 문제는 구글 검색과 수업 ppt자료를 참고하여 해결하였기에 수월하였기에 문제가 되었던 점은 대표적으로 다음의 2가지였던 것 같다.

1) 완성된 코드 확인을 위해 LKM을 내렸다 올렸다 반복하는 과정에서
myproc에 오류가 생기는 문제

이 문제는 LKM을 제거했을 경우에도 계속해서 myproc이 남아있었기 때문이었다.
찾아본 결과 module exit 과 함께 remove_proc_entry 함수를 이용하여 myproc 폴더 및 파일을 제거해주는 과정을 추가해주니, 다시 새로 해당 폴더 및 파일을 생성할 때 문제가 생기지 않았다.

2) 서버 전송이 끊기는 문제

packet_limit의 값을 업데이트 하기 전에 확연한 비교를 위하여 약간의 시간을 두었는데, 다시 packet_limit 값을 올려도 애초에 서버에서 데이터가 전송되지 않는 문제를 발견하였다. 그 시간을 너무 길게하지 않고 약 30초 정도로 잡자 문제없이 다시 서버에서 데이터를 전송함을 알 수 있었다. 따라서 결과를 구하는 데에는 크게 문제는 없었다.