Hw2. Packet filtering

1. 과제 전반의 진행
2. 리눅스 커널 뮤듈
   1. Handler 를 이용하여 sk\_buff로 부터 ip주소와 mac 주소 얻기
   2. Procfs 를 이용하여 app에서 특정 패킷 filtering on / off
   3. 최근 100개의 packet 저장 그리고 출력
   4. 마지막까지 나를 괴롭힌 ktime\_t tstamp
3. APP 개발
   1. Server app
   2. Client app
4. 결과출력
   1. 준비작업
   2. 필터링 시작전
   3. 필터링

컴퓨터 통신 공학부

2010210107 홍보의

2014.11.25.

시스템 프로그래밍

유 혁 교수님

1. 과제 전반의 진행

개발 전반은, 조교님이 제공해 주신 예제코드를 수정하여 시작하였다. Notifier 등록에 대한 과정이 세팅이 되어 있었으며, 여기에 procfs 사용을 위한 read write 함수의 설정이 추가되었다. Procfs 중에서 write 함수를 사용하였으며, read는 사용하지 않았다.

숙제2에 있어서 가장 중요한 함수는 check\_packet 함수로서 이 함수는 notifier를 등록할때 handler 의 모습으로서 등록이 된다. Notifier 는 커널의 상태가 변할 때 ( 패킷이 들어올 때) handler 가 등록되어 있다면 이를 수행시킨다. 이 때 이 핸들러를 내가 원하는 대로 변형하여 원하는 동작을 수행 할 수 있다.

Check\_packet 함수 내에서 sk\_buff 구조체를 이용하여 ip\_hdr 함수와 mac\_hdr 함수를 통해 ip헤더와 mac헤더 구조체를 얻을 수 있었고 이 두 구조체에는 송신측 ip주소, mac 주소가 존재했다. 이를 printk를 통해서 콘솔창에 출력을 한다. 출력하는 것은 odroid로 들어오는 모든 패킷에 대하여 진행한다. 커널 내부에는 최근 들어온 100개의 packet 정보(송신측 mac, ip, device 정보, 시간정보,pass/consume 여부)를 저장 할 수 있게 해 놓았는데, 이곳에 저장되는 packet 은 과제에서 요구하는 2개의 device(데스크탑과 휴대폰)에서 오는 것 만을 확인하여 저장한다.

본 과제의 목적은 패킷 필터링이다. 안드로이드 어플리케이션 과의 상호작용 (proc\_fs write 를 사용하였다) 을 통해서 특정 mac 주소로부터 송신되는 packet 을 CONSUME 하는 작업을 한다. 2개의 client 로부터 어느 client 를 필터링 할 것인지를 버튼을 통하여 선택한다. 어플리케이션에서는 desktop , otherdev 두 버튼을 만들어 놓았다. 각각 다른 mac 주소에 대햐여 필터링을 진행한다.

1. 리눅스 커널 모듈
   1. Handler 를 이용하여 sk\_buff로 부터 ip주소와 mac 주소 얻기

Check\_packet 함수는 이번 과제에서 가장 핵심적인 함수이다. 일단 모듈이 등록되면 기기로 들어오는 모든 패킷은 본 함수를 거치게 되어있다. 이곳에서 패킷의 소켓버퍼 구조체에 접근을 할 수 있으며, 소켓 버퍼 구조체는 패킷의 데이터 뿐만 아니라 헤더정보도 모두 가지고 있다.

|  |
| --- |
| struct sk\_buff{  …..  sk\_buff\_data\_t transport\_header;  sk\_buff\_data\_t network\_header;  sk\_buff\_data\_t mac\_header  …..  }; |

아래의 구조체가 odroid3.4.y의 sk\_buff 구조체 이다. Sk\_buff\_data\_t 형의 자료형으로 nerwork\_headr 와 mac\_headr 가 선언되어 있다. Sk\_buff\_data\_t는 소켓버퍼와 같은 헤더파일인 skbuff.h에 선언되어 있으며, typedef unsigned int sk\_buff\_data\_t 로 선언되어 있다. 32비트 정수값인 것으로 보아 소켓 버퍼 구조체의 network\_header 란 네트워크 헤더의 주소값이다. 이를 통해서 network header 에 접근, 송신 ip 주소를 얻어내는 것이 목표이다.

|  |
| --- |
| struct iphdr {  ……  \_\_u8 protocol; //프로토콜은 여기  \_\_sum16 check;  \_\_be32 saddr; //여기가 송신측 주소.  \_\_be32 daddr;  /\*The options start here. \*/  }; |

이를 위해서 ip\_hdr 라는 함수를 사용하였다. 이 함수는 ip.h 에 선언되어 있는 함수로서, 소켓버퍼 포인터를 넘겨주면 간편하게 ip header 를 전해준다. 이 ip header 에는 u32로 선언되어 있는 saddr 값이 있다. 이는 send address 의 약자로서 이것이 바로 송신측 ip 주소이다, 디버깅을 위하여 출력을 하였다.



다음과 같은 주소값들을 얻어 낼 수 있었다. 순서가 바뀌어 있는 이유는 네트워크 byte order 로 되어있기 때문이다. 이는 ntohl(network to host long)함수를 이용하여 호스트 주소로 변환 할 수 있다. 하지만 구글에 접속을 해 보아도 접속이 되지 않고 송신측 ip가 구글 ip와 상관이 없다.

* 첫번째 문제 : 첫번째는 예제코드의 check\_packet 함수의 리턴값인 EXACT\_PACKET을 수정하지 않고 사용했기 때문이다. 이 때문에 모든 패킷이 PASS되지 않았고, 구글에 접속을 할 수 없었다. Netdevice.h 에 선언되어 있는 rx\_handler 의 리턴값은 다음과 같이 정의되어 있다.

|  |
| --- |
| enum [rx\_handler\_result](http://lxr.free-electrons.com/ident?i=rx_handler_result) {  RX\_HANDLER\_CONSUMED,  RX\_HANDLER\_ANOTHER,  RX\_HANDLER\_EXACT,  RX\_HANDLER\_PASS,  };  ***@RX\_HANDLER\_EXACT: Force exact delivery, no wildcard.***  ***@RX\_HANDLER\_PASS: Do nothing, passe the skb as if no rx\_handler was called.*** |

Exact는 정확히 어떠한 역할을 수행하는지 모르지만, 패킷이 마치 consumed 로 설정것 처럼 들어오지 않는 현상을 보였다. 리턴값을 pass로 바꾸니 정상적으로 동작하였다. 하지만 이후에도 웹 서비스는 정상적으로 수신송신이 되나, sk\_buff로부터 가져온 헤더 데이터는 엉뚱한 ip만 나왔다. 그래서 protocol을 찍어 보았더니. Protocol 0 이 찍혔다.



Protocol 0 은 IPv6 Hop-by-Hop Option 로 정의되어 있는데, 이해가 되지 않았고, 모든 패킷이 0인 것도 이해가 되지 않았고, 내가 사용하고 있는 것은 ipv4 즉 4번 프로토콜이 아닌가 싶었다. 그래서 데스크탑과의 socket 통신 어플리케이션을 작성한 이후에 다시 실행해 보았으나 같은 결과를 보였다.

* 두번째 문제 : 두번째는 check\_pakcet 즉 , rx\_handler 로 등록한 함수의 원형이 sk\_buff의 더블 포인터형을 인자로 받는다는 것이었다. 혼자서 알아내지는 못하였지만 다음부터 문문제 봉착했을 때 문제의 근본을 더 폭넓게 생각해야 겠다.

이번에는 ip header 구조체 내부에서 실수를 한 것이라 가정하고 그곳에 매여서 한단계 내부의 문제를 생각하지 못했다.

|  |
| --- |
| typedef [rx\_handler\_result\_t](http://lxr.free-electrons.com/ident?i=rx_handler_result_t) [rx\_handler\_func\_t](http://lxr.free-electrons.com/ident?i=rx_handler_func_t)(struct [sk\_buff](http://lxr.free-electrons.com/ident?i=sk_buff) \*\*pskb); |

다음과 같이 nerdevice.h 에 선언되어 있는rx\_handler\_func\_t 는 sk\_buff의 더블 포인터를 받는다. Rx\_handler\_retsult\_t의 바로 아래에 선언되어 있다. 결굴 이를 수정하고 나서 올바른 주소값이 나오는 것을 확인 할 수 있었다.

결과적으로 check\_packet 함수는 다음과 같은 형태가 되었다.

|  |
| --- |
| rx\_handler\_result\_t check\_packet(struct sk\_buff \*\*skb){    struct ethhdr \*hw2\_machdr;// inline function is just translated to  unsigned int ip\_sdr; // the code so i have to declare machdr here.  unsigned char\* hw2\_smac;  long long hw2\_time;  hw2\_iphdr = ip\_hdr(\*skb);  hw2\_machdr = eth\_hdr(\*skb);  hw2\_time = ktime\_to\_ns((\*skb)->tstamp);  ip\_sdr = ntohl(hw2\_iphdr->saddr);  hw2\_smac = hw2\_machdr->h\_source;    //here we get pointer of mac address uns char h\_source[6]  printk(KERN\_INFO "time : %lld ip\_sdr :%u.%u.%u.%u mac\_sdr =%x:%x:%x:%x:%x:%x \n",hw2\_time,(ip\_sdr & 0xFF000000) >>24,(ip\_sdr & 0x00FF0000) >>16,(ip\_sdr & 0x0000FF00) >>8 ,(ip\_sdr & 0x000000FF),hw2\_smac[0],hw2\_smac[1],hw2\_smac[2],hw2\_smac[3],hw2\_smac[4],hw2\_smac[5]);  ……….  } |

위를 보면 hw2\_iphdr = ip\_hdr(\*skb); 를 통해서 ip\_header 를 받고, 32bit unsinged int 값에 ip\_sdr = ntohl(hw2\_iphdr->saddr); 와 같이 저장을 해서,(ip\_sdr & 0xFF000000) >>24,(ip\_sdr & 0x00FF0000) >>16,(ip\_sdr & 0x0000FF00) >>8 ,(ip\_sdr & 0x000000FF) 와 같이 2byte 씩 잘라서 .을 포함하여 출력하고 있다.

mac\_header 의 경우에는 if\_ehter.h 에 있는 api를 사용하여 mac주소를 가져왔다.

|  |
| --- |
| 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)); |

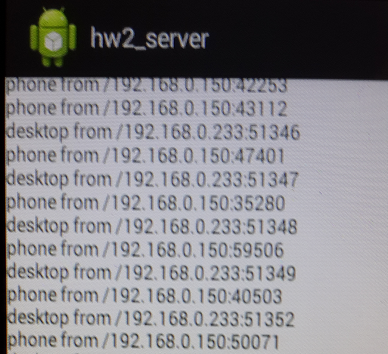
다음과 같이 ethhdr가 선언되어 있는데, 이 구조체를 이용하였고,

|  |
| --- |
| static inline struct ethhdr \*eth\_hdr(const struct sk\_buff \*skb)  {  return (struct ethhdr \*)skb\_mac\_header(skb);  } |

위의 eth\_hdr 함수를 사용하여,헤더를 얻어서 그 속의 h\_source배열을 획득했다. 여기에 맥 주소 8자리가 unsigned char 형 으로 선언되어 있다. 즉, 먼저hw2\_machdr = eth\_hdr(\*skb); 를 통해 헤더를 얻고 hw2\_smac = hw2\_machdr->h\_source; 송신 맥 주소를 얻고 ),hw2\_smac[0],hw2\_smac[1],hw2\_smac[2],hw2\_smac[3],hw2\_smac[4],hw2\_smac[5] 와 같이 출력을 행한다. 위의 eth\_hdr 구조체를 보면 \_\_attribute\_\_((packed));로 선언되어 있는 것을 볼 수 있는데 , 이는 데이터를 padding 하기 위한 옵션이다. Ip\_hdr 의 경우에는 그렇지 않은데 mac\_hdr의 경우에는 헤더의 크기를 일정하게 유지 한다는 것을 알 수 있었다.

* 1. Procfs 를 이용하여 app에서 특정 패킷 filtering on / off

과제 시나리오는 Desktop과 휴대폰이 odroid 로 패킷을 보내면, 이 패킷이 통과된다면, odroid는 데이터를 app의 화면에 찍어준다. Desktop은 “ desktop from ip주소 “ 의 형태로 데이터를 보내서 이 데이터가 어느 ip에서 온 것인지 확인 할 수 있게 한다. 휴대폰의 경우는 “ phone from ip주소 “ 형식을 사용한다.



앱이 실행되고 두개의 client 가 패킷을 보내면 다음과 같은 화면이 보여진다.

|  |
| --- |
| out.writeUTF("desktop from "+ client.getLocalSocketAddress()); |

Odroid 에 설치되는 server app 에는 버튼이 총 3개 존재한다. 첫번째 버튼은 desktop 두번째 버튼은 otherdev 이다. 각각을 클릭하면, 해당 디바이스가 filtering 된다. 만약 desktop 버튼을 누른다면 server app은 다음과 같은 코드를 수행한다.

|  |
| --- |
| proc\_f = **new** File("/proc/hw2\_proc");  fw = **new** FileWriter(proc\_f);  bfw = **new** BufferedWriter(fw);  bfw.write(str2);  bfw.close(); |

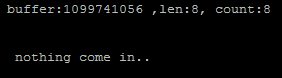
이미 모듈에는 hw2\_proc 라는 모듈이 생성되어 있다. 이를 open 하고 여기에 write작업을 한다. Str2 로 명명된 write 를 할 스트링은 desktop의 경우에는 hw2\_desktop 이고 otherdev를 클릭했을 시에는 hw2\_otherdev 이다. 이렇게 쓰여진 것은 procfs 의 write 함수 내부에서 각 버튼에 해당되는 작업으로 분기한다.

|  |
| --- |
| char hw2\_buff[20];  if(copy\_from\_user(hw2\_buff,buffer,len))  return -EFAULT; |

위의 코드는 모듈의 write 함수 내부에 있는 코드이다. 만약 app에서 write 를 수행하면, copy\_from\_user를 이용하여, 이를 커널 내부로 옮겨온다. Write 함수 api의 파라미터인 buffer 에는 유저가 write 한 데이터가 존재하고, 또다른 파라미터 count 에는 들어온 데이터의 문자수가 기록되어 있으니, 이를 이용하면 된다.

* 여기서 한가지 문제점이 발생했었는데, copy\_from\_user 는 올바르게 수행 되었을 때 0을 리턴한다는 것이다. 처음에 0이면 오류로 생각하고 코딩을 했다가 오래 고생을 했다. 처음에 if(copy\_from\_user( …) ==0) printk(“nothing come in”); else printk(“ok”);

와 같이 리턴값이 0 이면 오류로 처리했는데, 계속하여 오류값만 나왔다 그래서 write 함수로부터 오는 user data가 저장되어 있는 주소, 그 크기를 가지고 있는 len 값 을 출력해 보았다.



출력결과 주소값으로 보이는 데이터와 내가 보낸 문자열의 문자 개수에 해당하는 len 값이 잘 들어오는 것을 볼 수 있었다. 그래서 다른 부분에 문제가 있음을 확인 했고, copy\_from\_user의 kernel api와 사용예를 확인하여, return 값이 성공시 0 임을 확인 하였다. 문제를 해결해 가는 좋은 경험을 하였다.

이렇게 받아온 데이터를 다음과 같이 파싱한다.

|  |
| --- |
| static int my\_write(struct file \*file, const char \_\_user \*buffer,unsigned long count, void \*data)  {  …….  char \*hw2\_desktop\_str = "hw2\_desktop";  char \*hw2\_otherdev\_str = "hw2\_otherdev";  …….  if(strncmp(hw2\_buff,hw2\_desktop\_str,11) == 0)  { … }  if(strncmp(hw2\_buff,hw2\_otherdev\_str,12) == 0)  { …}  } |

위와 같이 strncmp 함수를 이용하여, 들어온 문자열이 무엇인지 판단하여 분기, 이후에 해당 작업을 수행한다.

이때 수행하는 작업은, 전역변수로 지정되어 있는 hw\_desktop\_on 과 hw\_otherdev\_on를 1-> 0 혹은 0 ->1 로 변경하여 켜고 꺼주는 일이다. 초기에 두 변수는 모두 0으로 설정되어 있으며, 아무런 필터링이 일어나지 않는다. 하지만 버튼을 누르면 해당 디바이스의 filter 가 on 되고, 그 데이터는 handler 에 의해서 RX\_HANDLER\_CONSUME 을 리턴하여 패킷을 통과시키지 않게 된다. 이를 위해서는 handler ,가 맥 주소를 판단하여 desktop 혹은 phone 이 보낸 패킷이 맞는지를 판단하여야 하며, 이에 대한 check\_packet함수의 코드는 다음과 같다.

|  |
| --- |
| if((hw2\_smac[0] == 0x20) && (hw2\_smac[1] == 0x16) && (hw2\_smac[2]==0xd8) && (hw2\_smac[3] == 0x87) && (hw2\_smac[4] == 0x9d) && (hw2\_smac[5] == 0x89))  hw2\_desktop\_filter = 1;  else  hw2\_desktop\_filter = 0;  ……..  if((hw2\_desktop\_filter == 1) && (hw2\_desktop\_on ==1))  {  printk(KERN\_INFO "\n\n\n desktop packet is consumed !!! \n\n\n");  return RX\_HANDLER\_CONSUMED;  } |

위의 code는 desktop의 경우만을 따로 본 것이다. Hw2\_desktop\_filter가 1이 되었으면 이는 desktop 에서 온 패킷이라는 뜻이다. 마지막 return 을 결정하는 것은 위의 write 함수에 의해서 변경되는 hw\_desktop\_on 변수와 check\_packet 에서 mac주소의 확인을 통해서 변경되는 hw2\_desktop\_filter 변수이다. 두 값이 모두 1 이라면, 사용자가 해당 디바이스 필터링을 원하고(버튼을 홀수번 눌렀고) 그리고 지금 들어온 패킷이 desktop 에서 들어온 패킷 이므로, RX\_HANDLER\_CONSUMED 를 호출한다.

여기서 불편한 점은, mac 주소를 동적으로 할당하는 것이 아니기 때문에, mac 주소를 바꾸고 싶다면, 이를 바꾸고 모듈을 다시 컴파일 해 주어야 한다는 것이다.

* 1. 최근 100개의 패킷 저장. 그리고 출력.

배열을 한 개 만들어서 패킷의 상태정보를 저장하도록 하였다. 배열에는 패킷이 보내진 기기가 desktop 인지 phone 인지. ip주소 mac주소는 무엇인지. pass되었는지 consume 되었는지. 처리된 시간값은 언제인지 등이 나타나 있다. 이 정보는 app의 store 버튼을 누르면 console 화면에 출력이 되게 되어있다.

* 1. 마지막까지 나를 괴롭힌 ktime\_t tstamp

Packet 이 들어온 시간을 기록하기 위해서 sk\_buff에 있는 tstamp 를 이용하려 하였다. 이름을 보면 딱 timestamp. 패킷이 들어온 시간을 기록하고자 만들어 놓은 것임에 틀림 없다. 자료형도 ktime\_t 로 되어있는데, 이는 include/linux/ktime.h 를 보면 정의되어 있다.

|  |
| --- |
| union ktime {  s64 tv64;  #if BITS\_PER\_LONG != 64 && !defined(CONFIG\_KTIME\_SCALAR)  struct {  # ifdef \_\_BIG\_ENDIAN  s32 sec, nsec;  # else  s32 nsec, sec;  # endif  } tv;  #endif  }; |

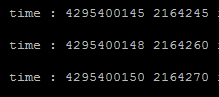
위와 같은 형태로 정의된 union이다. 헌데 check\_packet 함수에서 sk\_buff 구조체로 접근하여 tstamp 변수 tv64를 printk 해 보았다. (\*skb)->tstamp.tv64 하지만 0이외에 다른 값이 출력되지 않았다. 그래서 ktime.h에 존재하는 여러가지 함수를 사용해서 출력을 시도해 보았다.

|  |
| --- |
| static inline s64 ktime\_to\_us(const ktime\_t kt)  {  struct timeval tv = ktime\_to\_timeval(kt);  return (s64) tv.tv\_sec \* USEC\_PER\_SEC + tv.tv\_usec;  }  static inline s64 ktime\_to\_ms(const ktime\_t kt)  {  struct timeval tv = ktime\_to\_timeval(kt);  return (s64) tv.tv\_sec \* MSEC\_PER\_SEC + tv.tv\_usec / USEC\_PER\_MSEC;  } |

다음과 같은 ktime\_to\_us 나 ktime\_to\_ms 함수를 사용해 보았는데, 이들또한 작동하지 않았다. 아마 tstamp값이 아예 0으로 세팅되어 있는 것 같았다.



위에 보이는 것이 0으로 출력되는 여러 함수를 활용한 tstamp값 이다. 어쩔수 없이 jiffie 값을 이용하여 시간값을 저장 하기로 하였다. 하지만 엄밀히 이야기 한다면, 이 jiffie 값은 패킷을 받은 시간과는 상관이 없다. 왜냐하면, check\_packet 함수에 접근하는 시간은 rx\_handler 가 호출되는 시간인데, 이 rx\_hanlder는 netif\_receive\_skb 함수 내에서 존재 여부가 확인되고, 호출된다. 즉, 패킷이 cpu에서 스케줄링 되고 bottom\_half 처리가 시작되고 나서의 시간 이다. 이는 패킷이 스케줄링이 되어 처리될 때의 시간이기 때문에, 패킷이 들어온 순서대로 처리되는 것이 아니라 스케줄링시 처리순서가 바뀐다는 사실을 확인하지 못한다. 아쉬운 대로 jiffie 값으로 rx\_handler로 패킷이 진입한 시간을 micro second 단위 기록을 하였다.

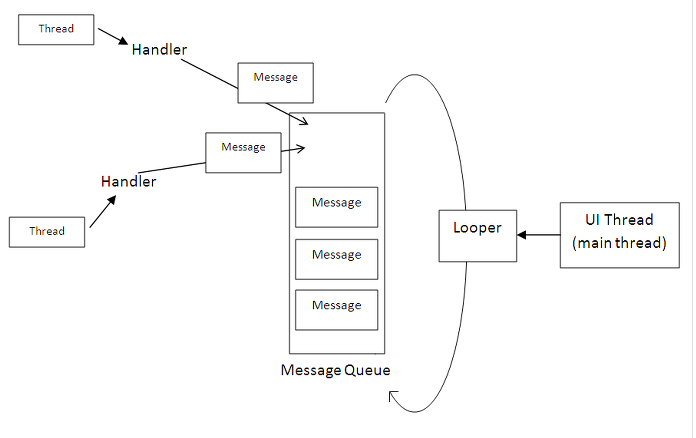


왼쪽은 jiffies 값 오른쪽은 jiffies\_to\_ms로 변환을 한 값.

1. APP개발
   1. Server app

서버 app의 목표는 1. 버튼을 누르면 Procfs 를 통한 filter on / off 2. Server thread를 만들어서 서버의 ip로 들어오는 패킷을 계속 기다림. 3. 서버로 들어온(RX\_HANDLER\_PASS 된) 패킷이 보낸 데이터를 화면에 출력.

간단한 개발이라 생각하여, 도전하였으나, 나의 짧은 android지식이 금방 바닥을 들어냈다. 안드로이드는 UI의 갱신을 여러 쓰레드에서 하는 것을 막는다.



다음과 같이 main thread는 하나 뿐이다. 이를 위해서 handler 라는 class를 사용한다. 정확한 작동 기전을 모두 알지는 못하였다. 안드로이드를 더 자세히 알아야 하겠다는 동기부여가 되었다. Handler 의 post 메소드를 통하여 UI thread 가 해야 할 작업을 등록해 주는데, 나는 이를 통해서 clinet 가 보낸 정보를 지속적으로 업데이트 하였다.

* 1. Client app

Client app 의 경우 기본적으로 desktop에서 java를 이용하여 작업할 때에는 문제가 없었으나, android의 경우에는 또 mainthread가 문제였다.

이번에는 network thread의 경우 mainthread (UI thread) 가 작업을 못하여 화면이 멈춘 것 처럼 보이는 현상을 막고 원활한 app의 구동을 위하여, 별도의 thread 에서 network 작업을 해야한다. 이런때에는 asynctask class를 상속하여 , 작업할 것을 지정해 주니 잘 동작 하였다. Asynctask에는 runInBackground 라는 메소드가 존재하는데, 이는 메인 쓰레드는 계속 동작하고 그 배경에서 또다른 작업을 진행 할 수 있게 하는 메소드 이다.

|  |
| --- |
| **protected** Long doInBackground(String... params) { //run in background method.  **while**(**true**){  **try**  {  System.*out*.println("Connecting to " + serverName  + " on port " + port); //debugging message.  Socket client = **new** Socket(serverName, port); //socket connect address.  System.*out*.println("Just connected to "  + client.getRemoteSocketAddress());  OutputStream outToServer = client.getOutputStream();  DataOutputStream out =  **new** DataOutputStream(outToServer); // make stream.  out.writeUTF("phone from "  + client.getLocalSocketAddress()); //send data phone from 'my ip'  client.close();  Thread.*sleep*(800);  }**catch**(Exception e)  {  e.printStackTrace();  }  }  }  } |

다음과 같이 메소드를 작성하고 새로만든 asynctask 클라스를 execute() 해주면 client 로서 패킷을 서버로 계속 보낸다.

1. 결과 출력
   1. 준비작업.

mac주소를 동적으로 받아서 할 생각을 하지 못한 것은 아니지만, 처음에는 어떻게 해야 하는지 감이 오지 않았다. 하지만 하다보니, procfs write 기능으로 내가 보내고 싶은 문자열을 보낼 수 있다는 것을 알았다. 하지만 수정을 하려면 서버앱에서 필터링을 원하는 ip혹은 mac 주소를 문자열로 보내고, LKM에서는 그를 파싱해서 해당 ip 혹은 mac 주소를 가진 패킷을 filtering 해 주어야 한다. 전반적으로 바꾸어야 하기 때문에 기존의 내 desktop pc와 내 휴대폰의 mac 주소를 filtering 하는 식으로 과제를 진행하였다.

먼저 app의 두 버튼 desktop과 otherdev는 각각 LKM에서 한 개의 mac 주소를 의미한다.

LKM을 보면. 다음과 같은 부분이 있다.

|  |
| --- |
| if((hw2\_smac[0] == 0x20) && (hw2\_smac[1] == 0x16) && (hw2\_smac[2]==0xd8) && (hw2\_smac[3] == 0x87) && (hw2\_smac[4] == 0x9d) && (hw2\_smac[5] == 0x89))  {  …….  hw2\_desktop\_filter = 1;  ….  }  else  hw2\_desktop\_filter = 0;  if((hw2\_smac[0] == 0xbc) && (hw2\_smac[1] == 0x72) && (hw2\_smac[2] == 0xb1) && (hw2\_smac[3] == 0xd0) && (hw2\_smac[4] == 0x02) && (hw2\_smac[5] == 0x71))  {  …….  hw2\_otherdev\_filter = 1;  ….  }  else  hw2\_otherdev\_filter = 0; |

위의 부분에서 if문 안에 들어있는 mac 주소가 2가지 있다 위의것은 desktop에 해당하고 아래것은 otherdev에 해당한다. 그곳에 mac 주소를 내가 filtering 하기를 원하는 mac 주소로 세팅을 하여야 한다.

또한,2개의 client에 대하여 server ip 주소를 바꾸어야 한다. Android app와 java 앱에

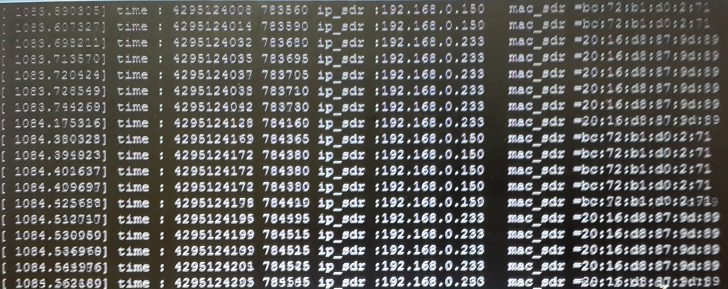
|  |
| --- |
| serverName = new String("192.168.0.4"); |

다음과 같이 serverName 이라 명명된 부분에 ip 값을 odroid 의 wifi ip 주소로 세팅을 해 주어야 한다.

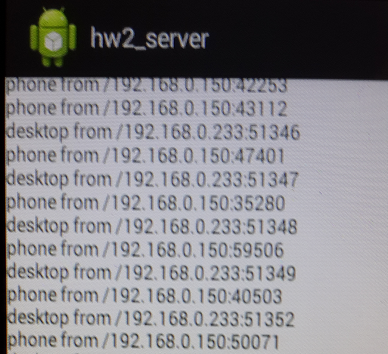
이러한 번거로운 작업들은 사실 client 에서는 연결할 server ip를 동적으로 받고,

서버 에서는 fitering 할 ip 혹은 mac 주소를 proc write 를 이용해서 동적으로 LKM으로 전달한다면, 더 유용한 프로그램이 될 것 같다.

* 1. Filtering 시작 전.



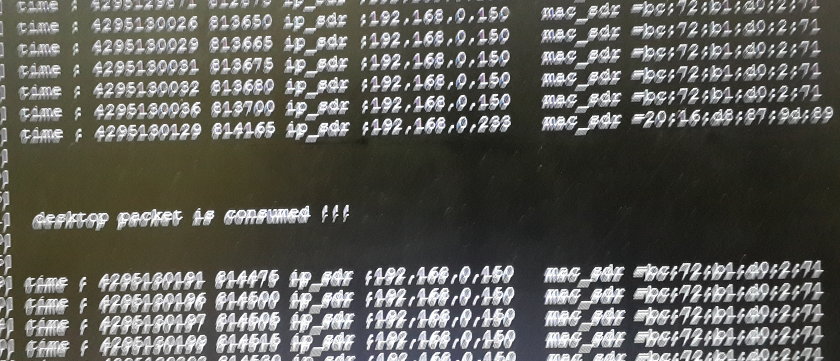
필터링을 하기 이전에는 다음과 같이 커널에는 print 가 이루어지고 있고,



Server(odroid)에서는 패킷이 잘 들어오고 있다.

* 1. 필터링.

필터링을 시작하기 위하여 odroid app의 desktop 버튼을 누른다. 그러면 해당 mac 주소릴 지닌 기기는 패킷을 송신 할 수 없다.



bc:72:b1:d0:2:71은 개 휴대폰의 mac 주소이고 20:16:d8:87:9d:89의 경우에는 내 desktop의 mac 주소이다. 위와 같이 20:16:d8:87:9d:89가 접근하려 할 때. Check\_Packet은 desktop packet is consumed !!! 이라는 문자열을 프린트 하고 RX\_CONSUMED 을 리턴하게 된다.

그러면 app의 경우에는 다음과 같이 phone 에서 보내는 packet 은 들어오지 않는다. 물론 desktop에 해당하는 packet 은 지속적으로 출력이 된다.

