# Computer Network Project Final Report

**이선호, 이현민, 노순현**

# 1. Introduction

현재의 네트워크 아키텍쳐는 레이어 계층 구조이며 각 레이어의 서비스들이 모아져 최종적으로 사용자가 사용하는 서비스를 가능하게 하여준다.. 각 레이어를 지나면서 패킷에는 새로운 헤더가 붙게되는 것이 특징이다. 하지만 Wifi의 사용과 같은 무선 통신의 경우 실질적인 데이터보다 헤더로 인하여 발생하는 대역폭의 소모가 클 수 있다.

주로 사용되는 네트워크 환경 구성을 생각하여 본다면, 서버와 이더넷으로 연결된 라우터, 그리고 라우터와 Wifi를 사용하여 통신하는 엔드포인트 단말이 될 것이다. 본 프로젝트에서는 라우터와 엔드포인트 간에서 주고받는 패킷 중 TCP ACK 패킷으로 인하여 발생하는 대역폭의 소모를 줄이는 것을 목표로 한다. 이를 위해 구현하고자 하는 시스템은 다음과 같다.

**TCP ACK**

Time

**AP**

**Client**

**TCP Data**

**L2 ACK dsfddACKACK**

**Server**

**TCP Data**

Time

Time

**TCP ACK**

**System Design**

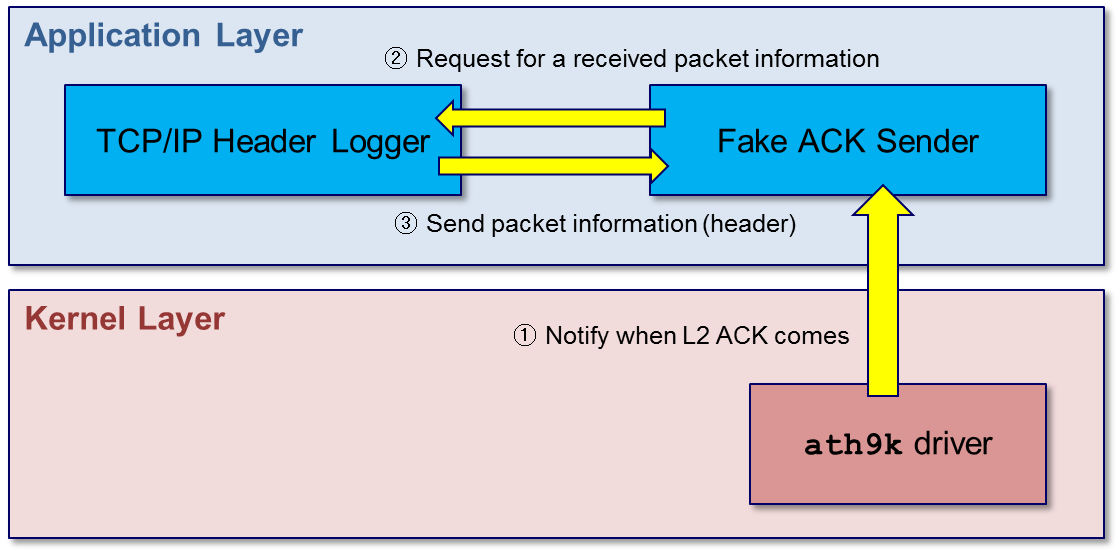
위 그림과 같이 TCP 클라이언트가 TCP Data를 받은 경우 레이어 2의 네트워크 어댑터에서 L2 ACK을 보내준다. 이후 TCP 레이어(L4) 에서도 TCP ACK을 보내게 된다. L2 ACK은 레이어 2의 성공적인 데이터 수신에 대한 것이며, TCP ACK(L4 ACK)은 레이어 4의 성공적인 데이터 수신에 관한 것이다. 그런데 이때 L2 레이어까지의 전송 성공은 L4 레이어의 전송 성공을 의미한다고 봐도 무방하다. 따라서 AP가 클라이언트로부터 L2 ACK을 받은 경우 AP에서 TCP ACK을 생성하여 보내주고 Client에서는 TCP ACK을 따로 generate하지 않도록 한다. 따라서 실제 구현해야 할 사항은 크게 다음과 같이 나눌 수 있다.

1. AP에서 들어오는 TCP Data의 IP header와 TCP header를 저장한다.
2. AP가 Client로 부터 L2 ACK을 받은 경우 앞서 1)에서 저장한 header 정보를 바탕으로 Fake TCP ACK을 생성하여 Server에게 전송한다.
3. Client에서는 따로 TCP ACK을 보내지 않도록 한다.

실험환경은 아마존 서버, 이더넷으로 인터넷에 연결된 라우터, 라우터를 통해 서버와 통신하는 노트북으로 구성하였다.

# 2. Solution Overview

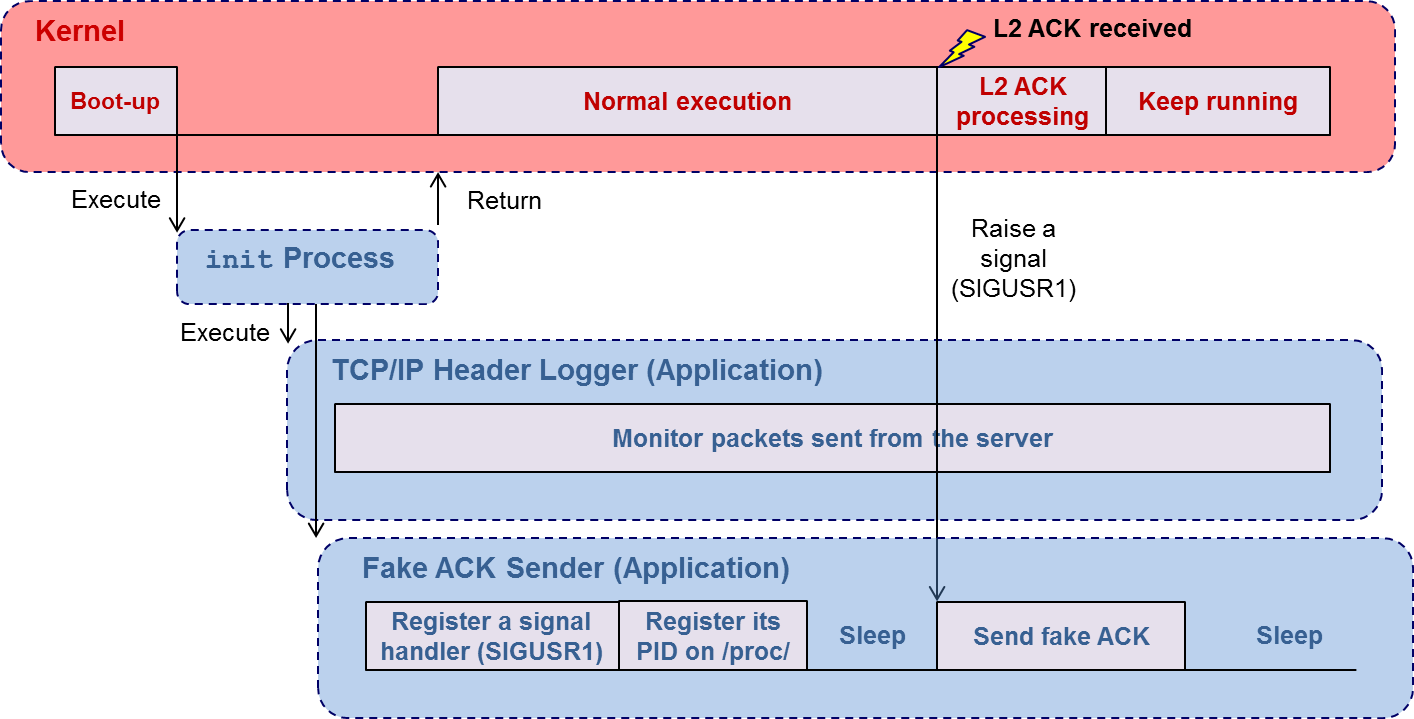
본 장에서는 우리가 설계한 시스템을 소개한다. 1장에서 언급했듯이 우리가 제안하는 시스템은 AP에서 (1) 서버로부터 받는 packet의 TCP/IP header를 저장해야 하고 (2) 노트북으로부터 L2 not duplicated L2 ACK을 받는 시점에 Fake ACK을 만들어서 서버에게 전송해야 한다. 우리는 이 동작을 하는 시스템을 구현하기 위해 두 개의 응용을 구현하고, ath9k 드라이버를 수정하였다. 전체적 구조는 아래 그림과 같다.



**Proposed Solution Architecture (AP)**

Kernel layer에는 L2 단의 무선 네트워크 통신을 담당하는 ath9k 드라이버가 있고 application layer에는 두 개의 응용, Fake ACK sender와 TCP/IP header logger가 존재한다. ath9k 드라이버는 노트북에서 L2 ACK이 도착할 때마다 Fake ACK sender에게 signal을 보낸다. Fake ACK sender는 잠들어 있다가 signal을 받으면 깨어나서 미리 등록해둔 signal handler를 통해 TCP/IP header logger에게 다음으로 L4 ACK을 보내야 하는 packet 정보를 IPC(inter-process communication)을 통해 요청한다. TCP/IP header logger는 요청을 받을 때마다 큐에서 저장되어 있는 다음 packet 정보를 fake ACK sender에게 보낸다. 물론 TCP/IP header logger는 이런 동작을 수행하기 위해 항상 server에서 전송되는 packet에 대한 정보를 기록하고 있어야 한다.

아래의 그림은 보다 더 구체적으로 두 응용의 동작 메커니즘을 설명하는 그림이다.



**Fake ACK sending mechanism**

앞서 언급한 두 응용은 커널의 초기화 과정에서 수행을 시작하고, TCP/IP header logger는 이 때부터 계속 server에서 오는 packet 정보를 기록한다. Fake ACK sender는 signal handler를 등록한 후 sleep 상태에 빠진다. 무사히 packet 전송이 되고 L2 ACK을 받을 경우 kernel 단에서 던진 signal에 깨어나서 fake ACK을 보내기 위한 일련의 과정을 수행한다.

마지막으로, not duplicated L2 ACK이 오는 시점은 커널 코드 상에서 아래와 같다.

|  |
| --- |
| static void ath\_tx\_complete(struct ath\_softc \*sc, struct sk\_buff \*skb,  int tx\_flags, struct ath\_txq \*txq) {  …  if (!(tx\_flags & ATH\_TX\_ERROR)) {  …  **send\_signal\_to\_header\_logger();**  }  …  } |

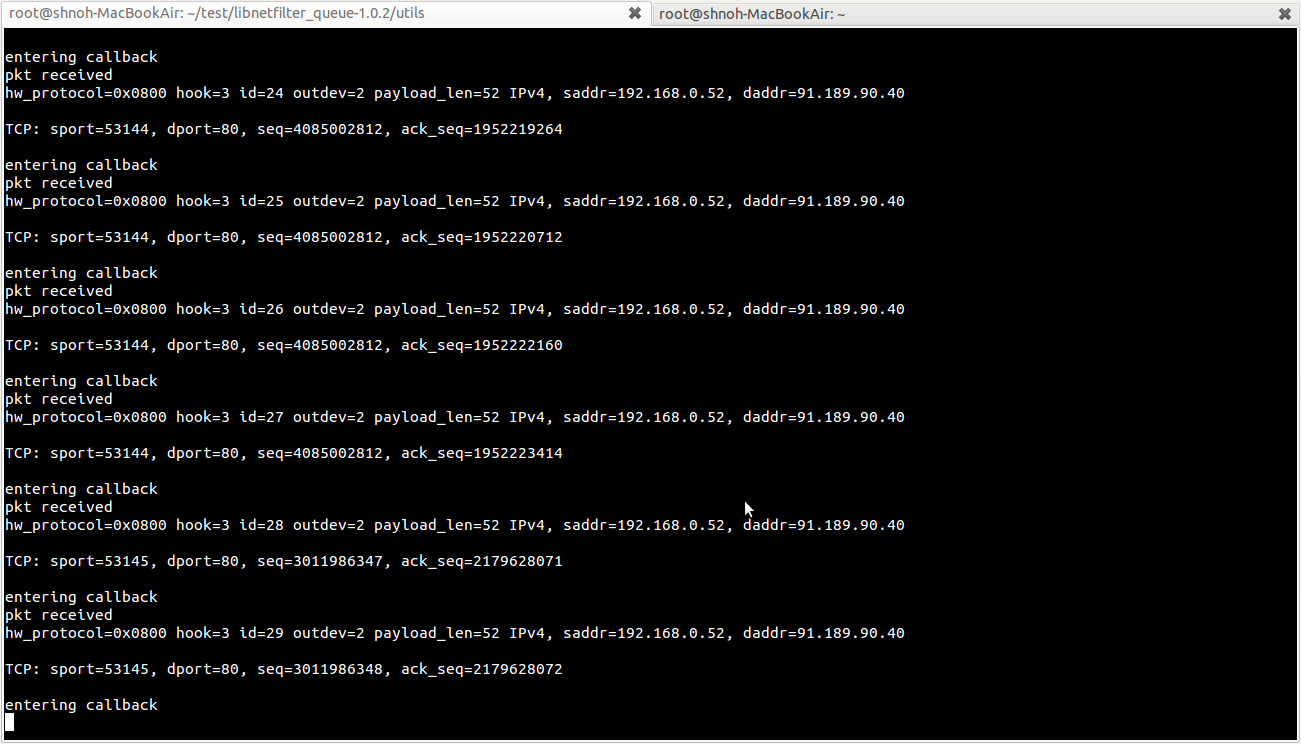
**drivers/net/wireless/ath/ath9k/xmit.c**

각 응용에 대한 동작은 3장과 4장에서 더 구체적으로 설명하도록 한다.

아쉽게도 해당 메커니즘을 아직 모두 구현하지 못했고, 구현한 일부 코드들만 git에 첨부한다.

# 3. Capturing TCP/IP headers (AP)

본 장에서는 2장에서 소개한 두 응용 중 TCP/IP header를 기록하는 응용의 구체적인 개발 내용과 검증 결과를 소개한다. 우리는 교수님과 조교님께서 주신 hint를 기반으로 libnetfilter\_queue 라이브러리를 사용하여 서버에서 AP로 전송되는 TCP/IP header를 캡쳐하였다. 구현 검증을 위해 저장하는 TCP/IP header의 내용을 출력한 결과는 아래 그림과 같다. 여기서 수행한 응용은 packet이 들어올 때마다 검증을 위해 해당 packet의 IP header와 TCP header 정보의 일부를 출력해준다.



**TCP/IP header logger 검증 실험 결과**

우리가 구현한 코드를 아래에 표시한다. 기본적으로 libnetfilter\_queue의 공식 홈페이지에 공개된 기본 예제인 nfqnl\_test.c를 사용하고, 아래와 같은 코드를 추가한다.

|  |
| --- |
| static u\_int32\_t print\_pkt (struct nfq\_data \*tb){  …  ret = nfq\_get\_payload(tb, &data)  …  **struct iphdr \*iph = ((struct iphdr \*) data);**  **/\* ipheader 로깅 및 검증을 위한 출력 작업 \*/**  **if (iph->protocol == 6) { // 만약 TCP라면**  **struct tcphdr \*tcp = ((struct tcphdr \*) (data + (iph->ihl << 2)));**  **/\* tcpheader 로깅 및 검증을 위한 출력 작업 \*/**  **}**  …  } |

**TCP/IP header logger 관련 주요 코드**

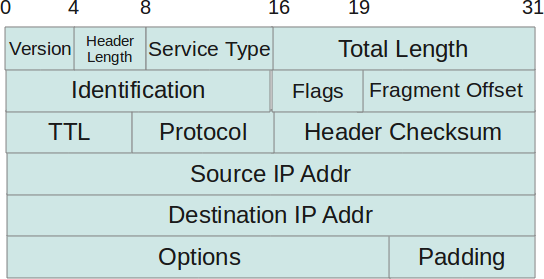
수행 방법은 컴파일 후 두 콘솔 창에서 아래의 두 명령어를 순차적으로 수행시키면 된다.

|  |
| --- |
| # gcc -Wall -o test nfqnl\_test.c -lnfnetlink -lnetfilter\_queue  # sudo ./test  # sudo iptables -A OUTPUT -p tcp -j NFQUEUE |

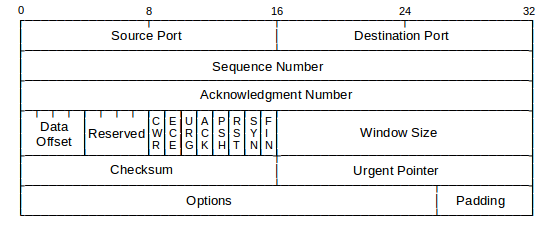
작성한 코드의 크로스컴파일에는 openwrt 디렉토리에 있는 mips-openwrt-linux-uclibc-gcc 파일을 사용한다.

# 4. Sending Fake TCP ACKs (AP)

Raw socket을 이용해서 Fake TCP ACKs을 만들었다. TCP ACK 이라서 data부분은 필요없기 때문에 ip와 tcp header만 만들어 주면 된다. 그 header의 구성 목록은 아래와 같다.



IP Header



TCP Header

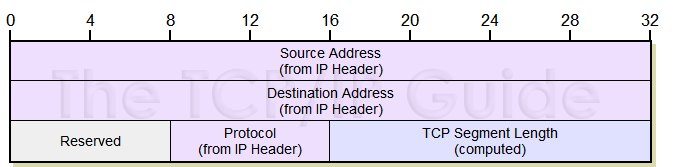
Version이나 Header Length등 Header의 대부분이 항상 같은 값을 가지므로 Ack\_packet이라는 header 틀을 만들었다. Wireshark로 client에서 TCP ACKs을 보내는 것을 캡쳐해서 캡쳐한 값으로 ACK\_packet에 채워 넣었다. 그리고 보내는 ACK마다 달라지는 값이 있는데 IP Header의 Id값, TCP Header의 Sequence Number, Acknowledgment Number, 그리고 각 Header의 checksum이다. 먼저 IP Header의 Id값은 rand함수를 이용하여 랜덤값으로 지정해 두었다. Flags의 More Fragment Follow값을 unset하고 Fragment Offset도 0이므로 Id가 같아도 문제가 없기 떄문에 Id값을 랜덤으로 생성하였다. 그리고 TCP Header의 Sequence Number와 Acknowledgment Number는 libnetfilter\_queue를 통해 걸러낸 TCP DATA packet에서 해당값에 맞에 설정하였다.

IP Header checksum은 다음과 같은 과정을 통해서 구한다.

1. IP Header의 checksum 부분을 0으로 채운다.
2. IP Header를 2 bytes씩 나누고 각각을 더한다.
3. 더하는 과정에서 overflow가 나면 overflow난 부분을 다시 더해준다.
4. 다 더하면 1의 보수를 취해준다.

구현은 Header를 unsigned short pointer로 casting했다. 그 이유는 unsigned short가 2 bytes 데이터 type이고 unsigned이므로 그냥 덧셈을 할 수 있기 때문이다. 그리고 그 pointer를 ++하면 그 다음 2 bytes를 읽으르모 계속 ++를 해주면서 4 bytes 데이터 type인 int에 더해 주었다. 다 더한 뒤에는 overflow가 생겼을 것이므로 앞 2bytes 부분과 뒤 2 bytes 부분을 더해 주었다. 그리고 마지막으로 endian이 달라지기 때문에 앞부분과 뒤부분의 위치를 바꿔 주었다.

TCP Header checksum은 IP Header checksum과 거의 같지만 IP Header를 통해서 Pseudo Header를 만드는 과정이 필요하다. Psuedo Header는 다음과 같이 구성되어 있다.

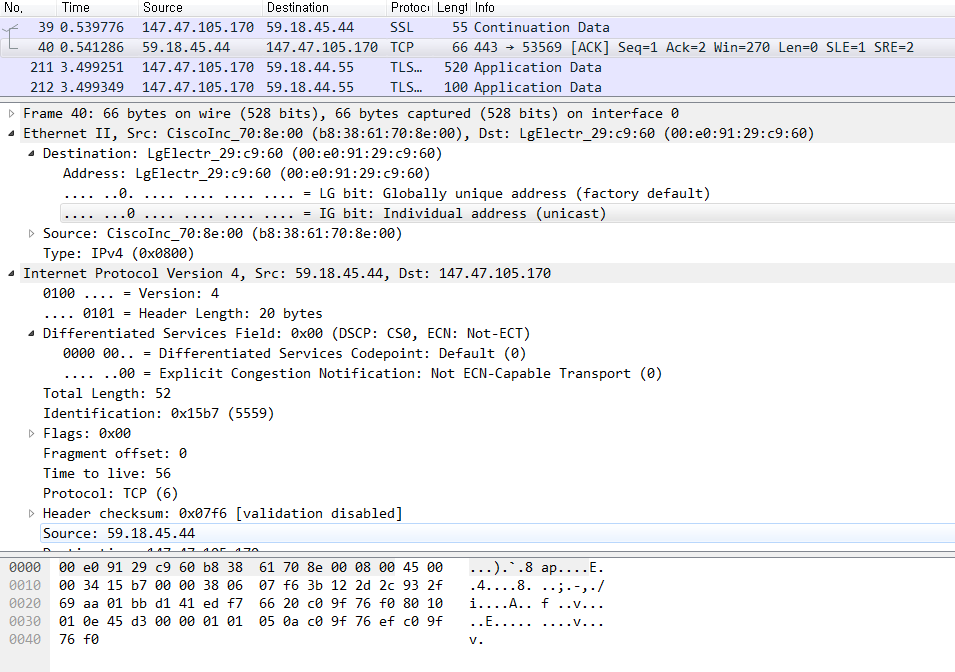


Pseudo Header

Source와 Destination의 Address는 IP Header를 통해서 알아내고 Reserved은 0 Protocol은 TCP이므로 06으로 채워넣고 마지막 TCP Sement Length는 ACK을 보내므로 정해져있으므로 정해진 값으로 채워 넣었다.

TCP Header checksum은 위에 구한 Pseudo checksum과 TCP Header부분을 합치고 그 합치 부분에 해서 IP Header checksum을 구한 방법으로 TCP Header checksum을 계산하였다. 마찬가지로 checksum을 구할 때 TCP Header의 checksum부분을 0으로 채우고 계산했다.

잘 구현했는지 테스트 해보기 위해서 wireshark로 한 TCP ACK을 캡쳐하여 아래와 같은 패킷을 얻었다. 그리고 그 패킷의 값을 char 배열에 넣은 뒤 IP Header와 TCP Header의 checksum을 계산하였다.



TCP ACK 캡쳐

그 결과는 아래과 같았고 캡쳐한 패킷의 checksum과 계산한 checksum의 결과가 일치함을 확인할 수 있었다.



checksum 계산 결과

# 5. Disabling TCP ACKs (Client)

L4 ACK의 전송은 리눅스 커널에서 처리하고 있다. 따라서 리눅스 내부의 L4 ACK 전송을 담당하는 부분을 찾아 그 기능을 끄면 된다. 리눅스 커널에서는 tcp\_output.c 파일에서 TCP ACK 전송을 위한 처리를 해준다. 간략히 과정을 살펴보면 다음과 같다.

tcp\_send\_ack()

tcp\_transmit\_skb()

to Layer 3

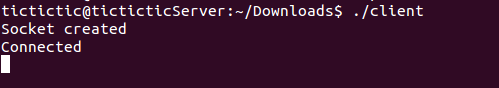
**TCP ACK 전송 과정**

tcp\_send\_ack 함수는 tcp layer에서 ACK을 보내줘야 할 경우 ACK 패킷을 생성하고 tcp\_transmit\_skb 함수를 호출하여 패킷을 전달하여 준다. tcp\_transmit\_skb 함수는 최종적으로 데이터를 L3 layer로 전달하여 주는 역할을 한다. 따라서 tcp\_send\_ack 함수에서 tcp\_transmit\_skb 함수로 ACK 패킷을 전달하지 않도록 함으로써 client 단에서의 TCP ACK 전송을 막을 수 있다.

|  |
| --- |
| void tcp\_send\_ack(struct sock \*sk)  {  struct sk\_buff \*buff;  …  buff = alloc\_skb(MAX\_TCP\_HEADER, sk\_gfp\_atomic(sk, GFP\_ATOMIC));  …  /\* Send it off, this clears delayed acks for us. \*/  TCP\_SKB\_CB(buff)->when = tcp\_time\_stamp;  **// tcp\_transmit\_skb(sk, buff, 0, sk\_gfp\_atomic(sk, GFP\_ATOMIC));**  } |

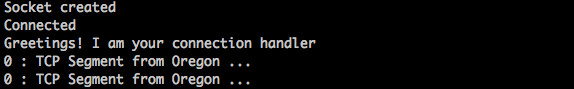
**tcp\_send\_ack 함수**

tcp\_send\_ack의 구현은 위와 같다. 가장 마지막 라인이 tcp\_transmit\_skb를 호출하여 ACK 패킷을 전달하는 부분이므로 이를 주석처리 함으로써 TCP ACK 전송이 일어나지 않도록 하였다. 실제로 클라이언트에서 실험하여 본 결과, TCP ACK의 전송이 발생하지 않는 것을 확인 할 수 있었다.



**TCP ACK blocking**

원래라면 Connectd 이후에 아래 사진과 같이 전송 메시지가 떠야하지만 서버측에서 ACK를 받지 못하였으므로 정상적인 통신이 이루어지지 않아 위의 사진과 같은 결과가 나오는 것이다.



**TCP ACK non-blocking**

# 6. Roles

* 이선호 – Sending Fake L4 ACKs 설계, TCP/IP header checksum 구현 및 테스트, 보고서 작성(4장)
* 이현민 – Sending Fake L4 ACKs 설계, Disabling L4 ACKs 설계 및 테스트, 보고서 작성(1장, 5장)
* 노순현 – Solution architecture(2장) 설계, TCP/IP header capture 응용 개발(3장), 보고서 작성(2장, 3장)

# 7. Conclusions

우리는 본 과제에서 서버와 라우터, 엔드포인트 단말 간의 통신 과정에서 TCP ACK 패킷 전송으로 인해 발생하는 대역폭의 소모를 줄이는 시스템을 개발하였다. 이 목적을 달성하기 위해서는 첫째, 서버에서 AP로 들어오는 TCP Data의 IP header와 TCP header를 저장해야 하고, 둘째, AP가 client로부터 L2 ACK을 받는 시점에 Fake TCP ACK을 생성하여 Server에게 전송해야 한다. 마지막으로, client의 커널에서 packet 전송을 받았을 때 TCP ACK을 추가적으로 생성하지 않도록 수정이 필요하다. 우리는 이 세 가지 기법을 각각 독자적으로 구현하고 검증하였다. 안타깝게도, 세 가지 기법을 합친 최종 솔루션을 구현하지는 못 하였다.

하지만 서버와 엔드포인트간 데이터가 전달되는 방식과 각 네트워크 레이어가 이 과정에서 실제 어떻게 동작하는지, 또한 그 구현이 어떠한지에 대하여 이해하였다. 또한 제시한 방식들을 조금 더 다듬고 활용한다면 최종적으로 만들고자 하는 시스템을 완성할 수 있을 것이다.