컴퓨터네트워크 과제

<DNS packet and Client-server application using netlink socket>

2017320122 김정규

DNS란

도메인 네임 시스템(Domain Name System, DNS)은 호스트의 도메인 이름을 호스트의 네트워크 주소로 바꾸거나 그 반대의 변환을 수행할 수 있도록 하기 위해 개발되었다. 특정 컴퓨터(또는 네트워크로 연결된 임의의 장치)의 주소를 찾기 위해, 사람이 이해하기 쉬운 도메인 이름을 숫자로 된 식별 번호(IP 주소)로 변환해 준다. 도메인 네임 시스템은 흔히 "전화번호부"에 비유된다. 인터넷 도메인 주소 체계로서 TCP/IP의 응용에서, www.example.com과 같은 주 컴퓨터의 도메인 이름을 192.168.1.0과 같은 IP 주소로 변환하고 라우팅 정보를 제공하는 분산형 데이터베이스 시스템이다.

인터넷은 2개의 주요 이름공간을 관리하는데, 하나는 도메인 네임 계층, 다른 하나는 인터넷 프로토콜(IP) 주소 공간이다. 도메인 네임 시스템은 도메인 네임 계층을 관리하며 해당 네임 계층과 주소 공간 간의 변환 서비스를 제공한다. 인터넷 네임 서버와 통신 프로토콜은 도메인 네임 시스템을 구현한다. DNS 네임 서버는 도메인을 위한 DNS 레코드를 저장하는 서버이다. DNS 네임 서버는 데이터베이스에 대한 쿼리의 응답 정보와 함께 응답한다.

DNS는 크게 반복적 질의와 재귀적 질의로 나뉜다. 이들의 수행 과정은 다음과 같다.

- 반복적 질의

1) 로컬 DNS에게 요청을 함 -> 로컬은 해당 정보를 가지고 있지 않다면 루트 DNS 서버에게 요청하는 IP 알고 있냐고 물어봄

2) 루트 DNS는 자기는 모르지만 아마 최상위 DNS 서버(TLD)는 알고 있을 거라고 로컬 DNS에게 알려줌

3) 로컬 DNS는 다시 최상위 DNS 서버에게 질의함 알고 있냐고. -> 자기도 모르는데 책임 DNS 서버는 알고 있을 것이라 알려줌

4) 로컬 DNS는 책임 DNS 서버에게 알고 있냐고 물어봄 -> 책임 DNS는 알고 있기 때문에 해당 IP 주소를 알려줌

5) 로컬 DNS는 받아온 정보를 사용자에게 최종적으로 알려주고 자신의 DNS 레코드에 나중에 똑같은 요청에 대한 신속한 처리를 위해 저장함

- 재귀적 질의

1) 로컬 DNS에게 요청을 함 -> 로컬은 해당 정보를 가지고 있지 않다면 루트 DNS 서버에게 요청하는 IP 알고 있냐고 물어봄
2) 루트 DNS 서버는 모르기 때문에 최상위 DNS 서버에게 물어봄 알고 있냐고
3) 최상위 DNS 서버도 모르기 때문에 책임 DNS 서버에게 물어봄 알고 있냐고
4) 책임 DNS 서버는 알고 있기 때문에 알려줌. (최상위 DNS에게 다시)
5) 최상위 DNS는 받은 정보를 다시 루트 DNS에게 알려줌
6) 루트 DNS는 로컬 DNS에게 받은 정보를 알려줌
7) 로컬 DNS는 최종적으로 사용자에게 받은 정보를 전달하고 자신의 DNS 레코드에 해당 정보를 추가함.

이렇게 재귀적 질의는 재귀적인 방법을 이용하여 질의를 하는 것이고 반복적 질의는 DNS 서버마다 로컬 DNS가 반복적으로 질문을 하는 방법을 말한다.

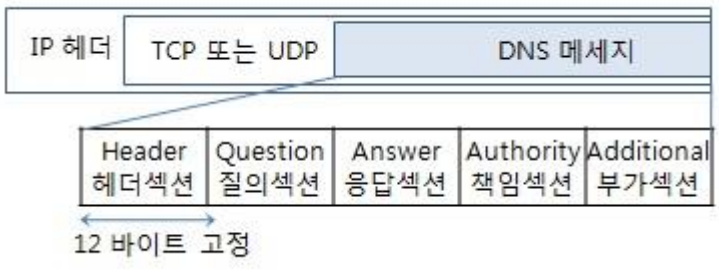
DNS 메시지

DNS 메시지란 DNS 질의 및 DNS 응답에 관련되어 전달되는 메시지로, DNS 클라이언트 및 서버 간에 또는 DNS 서버들(네임서버) 간에 전달된다. 이 DNS 메세지는 전체 DNS의 원활한 동작을 위한 주고받기, 복사 등에 이용된다.

- 질의 : 도메인네임에 대한 상세 정보를 요청
- 응답 : 질의에 대한 상세 정보의 응답
- 영역 전달 : 영역과 관련된 자원레코드 집합을 다른 네임서버로 복사하는 것
- 동적 갱신

DNS 메시지는 크게 질의메시지(Query)와 응답메시지(Response)로 구분된다. 자세한 구조는 다음과 같다.

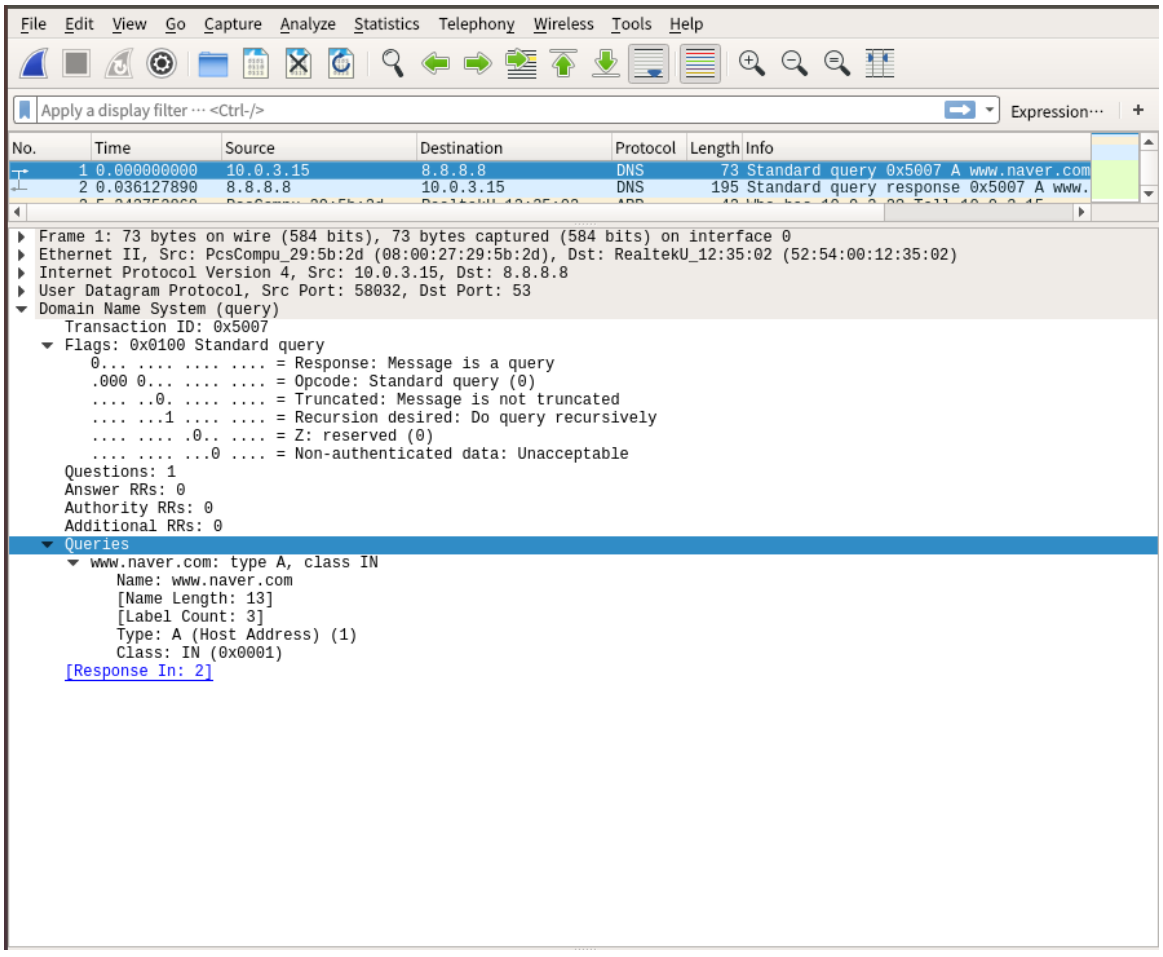
- 질의 메세지(2개 영역만 있음) = Header + 질의
- 응답 메세지(5개 영역 가능) = Header + (질의 + 응답 + 책임 + 부가정보)



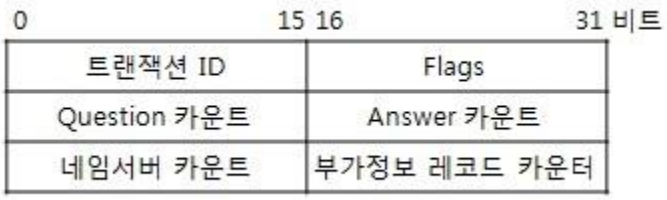
이러한 포멧은 몇가지 특징을 가지고 있다. 먼저 이것은 단일 기본 포멧이란 점이다. 위 그림처럼 모든 DNS 패킷 포멧은 단일 기본 구조를 사용한다. 두번째는 크기가 제한되어 있다는 것이다. UDP 512 바이트로 크기를 제한한다. (512 바이트 이상이면 TCP로 전환) 마지막으론 조회 실패 시 재전송 간격이 약 2 ~ 5초라는 것이다.

DNS 패킷 분석

DNS 클라이언트 코드를 통해 와이어샤크로 DNS 패킷을 캡쳐하였다. DNS의 요청과 응답이 각각 하나씩, 총 두개의 패킷을 얻을 수 있었다.



위의 캡쳐는 DNS 요청 패킷이다. 이를 분석하기 위해선 DNS 메시지 헤더 영역과 DNS 메시지 질의 영역에 대한 이해가 수반되어야 한다. 먼저 DNS 메시지 헤더부터 살펴보자



위의 그림은 메시지 헤더의 전반적인 구조를 설명한다. 각 필드의 세부적인 설명은 다음과 같다.

- 트랜잭션 식별자 (16비트)
  - 매 질의 마다 고유한 식별 ID 생성, 바로 이 식별자로 질의-응답 연결

- 플래그 (16비트)
  - QR (Query/Response)
    . DNS 메시지가 `DNS 질의(0)`,`DNS 응답(1)` 인지를 구분
  - Op code (Operation code) : 4 비트
    . 0 : 표준 질의 (또는 표준 질의에 대한 응답)
    . 1 : 역 질의 (Inverse Query)
    . 2 : 서버의 상태 요구
    . 4 : 통지
    . 5 : 갱신
  - AA (Authoritative Answer, 책임 플래그)
    . DNS 응답 메시지에만 사용됨
    . 1 : 네임서버 권한이 인정된 서버일 때
  - TC (Truncated)
    . 512 바이트 초과 여부
    . 1 : 응답메세지가 512 바이트 이상이라 512로 나뉘어짐
      . 이 경우에 클라이언트는 TCP 포트를 통해 질의를 재전송하게 됨
  - RD (Recursion Desired, 재귀 요구 플래그)
    . 1 : 클라이언트가 재귀 질의를 원함
      . 질의메세지에서 지정되며, 응답메세지에서 이를 반복함
      . 재귀 질의가 요청되지 않으면, 반복 질의로 간주하게 됨

- RA (Recursion Available, 재귀 유효 플래그)
  . 네임서버가 재귀 질의가 이용 가능한지를 나타냄
- 예약 (Reserved) : 통상 000으로 세팅
- rCode (response code, 응답/오류 코드)
  . 0 => No Error, 1 => Format Error(질의를 이해할 수 없음),
  2 => ServFail(Server Failure), 3 => 도메인 네임 존재하지 않음 등

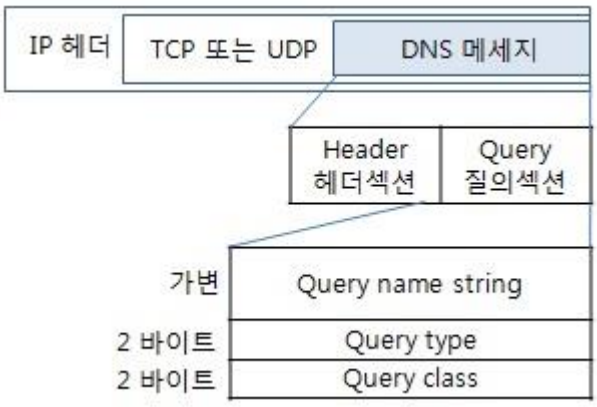
ㅇ 질의 카운트
- 질의의 수

ㅇ 응답 카운트
- 응답 RR(Resource Record)의 수

ㅇ 네임서버 카운트
- 책임 RR(Resource Record)의 수

ㅇ 추가정보 카운트
- 추가 RR(Resource Record)의 수

다음 그림은 DNS 메시지 필드 중 하나인 질의 필드를 설명한다.



ㅇ 질의명 스트링(Query name string) : 가변 길이, 최대길이는 63
- 질의의 대상(즉, DNS 갱신 대상)인 영역 이름
  . 연이은 다수의 라벨로 구성
  . 각 라벨은 길이를 나타내는 1 바이트 필드로 시작
  . `0` 나타나면, 이름의 끝을 의미

ㅇ 질의 타입(Query type) : 16 비트

- 클라이언트가 요청하는 질의 유형

. 1 (A) : IPv4 Address

. 2 (NS) : 네임서버

. 5 (CNAME) : Canonical Name (정식 이름)

. 6 (SOA) : Start of Authority

. 13 (HINFO) : 호스트 정보

. 15 (MX) : Mail exchange. 이메일을 메일 서버로 리다이렉트하도록 요청

. 28 (AAAA) : IPv6 Address

. 33 (SRV) : 특정 프로토콜이나 서비스에 대한 정보 요청

. 252 (AXFR) : 전체 DNS 영역 전달에 대한 요청 (질의에서만 사용됨)

. 255 (ANY) : 모든 레코드에 대한 요청 (질의에서만 사용됨)

ㅇ 질의 등급(Query class) : 16 비트

- DNS가 사용하는 특정 프로토콜을 나타냄

. 1 : IN (Internet) => 인터넷인 경우

. 3 : CS => COAS network

. 4 : HS => Hesoid server (MIT 개발)

위 내용을 토대로 캡쳐한 요청 패킷을 분석해보자.

1. IPv4를 사용하며 클라이언트 (10.0.3.15)이 DNS(8.8.8.8)으로 쿼리를 보낸 것을 볼 수 있다.

2. UDP를 사용한 것을 알 수 있다. 이 때 클라이언트는 임의의 포트 (58032)를 사용하고, 목적지엔 DNS의 포트인 53을 사용한 것을 확인할 수 있다.

3. DNS (query)

3-1 Transaction ID : DNS 쿼리와 응답에 연관된다. 사용자는 이 필드에서 DNS에 관련된 모든 것을 보기 위한 값을 필터링할 수 있습니다.

3-2 Flags : DNS 패킷의 성격을 제어하는 많은 필드로 구성되어 있다.

- Response(Query)

패킷이 요청하는 패킷(0)인지, 응답하는 패킷(1)인지 표시하는 비트이다. 위의 패

킷은 쿼리 패킷이므로 response값에 0이 있는 것을 확인할 수 있다. 이 비트에 대한 필터링은 아래와 같다.

dns.flags.response == 1(응답)

dns.flags.response == 0(요청)

- Opcode

Opcode는 쿼리의 유형을 지정햔다. 보통은 0000을 포함하고 있다.

- Authoritative Answer

도메인 이름에 대해 믿을 수 있는 서버로부터의 응답임을 표시한다.

- Truncation

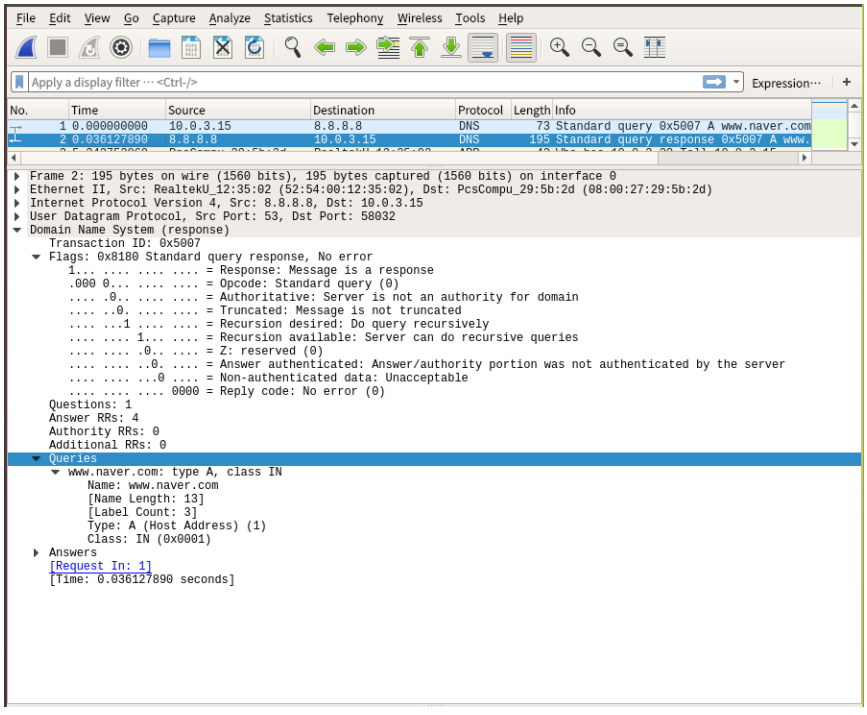
응답이 길어서 잘렸는지에 대해 알려주는 비트이다. 보통은 잘리는 경우가 없어 거의 0비트로 표시된다.

- Recurison Desired

재귀를 사용하는지 안 사용하는지를 알려주는 비트다. 대부분의 DNS는 재귀 쿼리를 사용한다.

- Reseved

예약된 비트이기 때문에 비워 놓는다. 그러므로 0으로 설정되어 있다.



위의 DNS 응답 패킷은 다른 부분은 요청 패킷과 같으나 flags를 비롯한 몇몇 부분은 다른 점을 확인할 수 있다.

1. 요청 패킷과는 반대로 이번에는 목적지였던 서버가(8.8.8.8) 클라이언트를 목적지로(10.0.3.15) 요청에 대한 응답을 보내준다.

2. UDP를 사용한다는 것을 알 수 있으며 DNS의 포트 넘버인 53번이 출발지 포트인 것을 확인할 수 있다.

3. DNS(response)

3-1 응답패킷의 flags 구조(요청 패킷에서 추가된 부분만 다룬다.)

(1) Authoritative : 도메인 이름에 대해 믿을 수 있는 서버로부터의 응답인지를 표시.(공식 DNS서버로부터의 응답인지)

(2) Recurision Available : 응답에서 정의된 재귀가 사용가능한지를 표시.

(3) Reply Code or Response Code : 응답에서 오류가 존재하는지 표시. 이 비트에는 0에서 5까지의 숫자가 오는데, 각 숫자에 대한 의미는 아래와 같다.

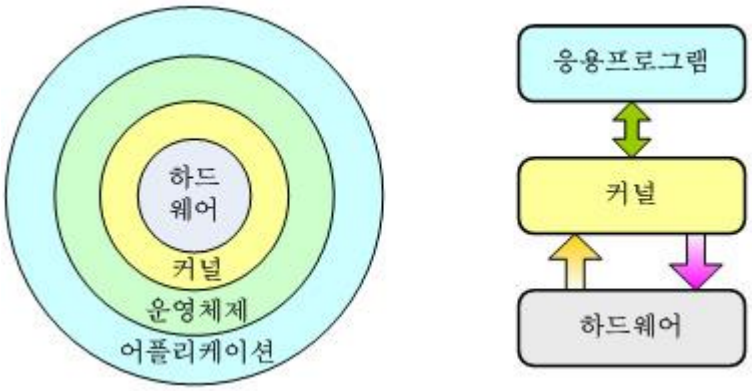
숫자(코드)	의미
0	오류 없음
1	형식 오류
2	서버 실패 (네임서버 문제)
3	네임 오류 (존재하지 않는 이름)
4	실행되지 않음
5	거부 (네임서버 규정으로 수행하지 못함)

3-2 Count

요청 패킷에서는 Question이 1, Answer RRs는 0이었던 것에 비해, 응답 패킷에서는 Answer RRs이 4가 되었다. 응답을 해주었기 때문이다. 만약 공식 DNS서버로부터 응답을 주었다면 Authority RRs도 다른 수가 될 것이다.

커널 모듈

과제에서 사용한 우분투를 비롯한 모든 운영체제의 핵심은 바로 커널(Kernel)이다. 커널은 운영체제와 하드웨어 사이에서 시스템의 자원을 관리하며 모든 시스템이 원활하게 동작할 수 있도록 제어하는 일종의 소프트웨어이다. 또한 DRAM에 상주하여 시스템의 구동에 필요한 환경설정과 수행되는 프로그램을 스케줄링한다. 아래의 그림을 참고하자.



커널은 CPU, 메모리, 하드웨어와 어플리케이션 사이에서 동작하면서 프로세서 관리, 메모리 관리, 파일시스템 관리, 네트워크 관리, 각종 디바이스 관리 기능을 수행한다. 운영체제에서 일어나는 모든 부분을 관리하고, 신규 디바이스 드라이버나 파일시스템 드라이버 등을 위해 업데이트가 필요할 때도 있다. 간단한 방법으로 필요한 커널 모듈을 로드(load) 및 언로드(Unload)할 수 있다.

위 이론을 기반으로 여기선 netlink를 이용해 커널과 사용자 사이의 정보를 전달할 수 있는 모듈을 사용했다. Netlink는 커널 모듈과 사용자 프로세스 사이에 정보를 전달하는 데 사용된다. 이것은 커널과 유저 사이의 양방향 링크가 가능하다. Netlink는 사용자 프로세스와 커널 모듈의 내부 커널 API와의 인터페이스에 기초를 두고 만들어 졌다.

Netlink 소켓은 다음과 같은 형식으로 사용된다.

sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

domain은 AF_NETLINK, 소켓 타입은 SOCK_RAW이다. 하지만 Netlink는 datagram에 기초한 서비스 이다. 소켓 타입의 값이 SOCK_RAW, SOCK_DGRAM 는 사용자가 인식하기 위한 것이지 Netlink 프로토콜은 둘을 구별할 수 없다. 모두 datagram으로 동작하게 된다.

Netlink_family selects the kernel module or netlink group to communicate with. The currently assigned netlink families are:

NETLINK_ROUTE : Receives routing updates and may be used to modify the IPv4 routing table, network routes, ip addresses, link parameters, neighbour setups, queueing disciplines, traffic classes and packet classifiers may all be controlled through NETLINK_ROUTE sockets

NETLINK_FIREWALL : Receives packets sent by the IPv4 firewall code.

NETLINK_ARPD : For managing the arp table from user space.

NETLINK_ROUTE6 : Receives and sends IPv6 routing table updates.

NETLINK_IP6_FW : To receive packets that failed the IPv6 firewall checks (currently not implemented).

NETLINK_TAPBASE...NETLINK_TAPBASE+15 : Are the instances of the ethertap device. Ethertap is a pseudo network tunnel device that allows an ethernet driver to be simulated from user space.

NETLINK_SKIP : Reserved for ENskip.

Server

UDP 서버 프로그램은 TCP 서버 프로그램과 달리 소켓에 주소를 할당하여 대기 큐의 크기를 지정한 뒤에 클라이언트의 접속을 기다리는 함수를 사용하지 않고 바로 데이터를 수신하는 함수를 사용하여 클라이언트에서 전송하는 데이터를 수신한다. 서버 프로그램의 대략적인 순서도는 다음과 같다.

1. 소켓을 초기화 (socket)

2. 소켓에 IP 주소 및 포트 번호 부여 (bind)

3. 클라이언트의 데이터를 수신(recvfrom)

4. 클라이언트에게 수신된 데이터를 전송(sendto)

5. 소켓 연결 종료(close)

UDP 서버 프로그램은 TCP 프로그램과 비교했을 때 대기 큐의 크기를 지정하는 listen() 함수와 클라이언트의 연결을 받아들이는 accept() 함수 없이 바로 recvfrom()함수를 사용하여 클라이언트의 메시지를 수신한다. TCP 서버 프로그램은 accept() 함수를 통해서 클라이언트의 접속을 받아들이면서 클라이언트의 접속 정보를 구조체 변수에 저장한다. 하지만, UDP프로그램은 접속이라는 개념 없이 바로 데이터를 송수신하기 때문에 recvfrom() 함수를 통해서 바로 데이터를 수신하면서 클라이언트의 접속 정보 또한 함께 구조체 변수에 저장하고 수신된 데이터는 배열 buff_rcv에 저장한다. 클라이언트로 수신 받은 데이터는 다시 한 번 클라이언트로 전송되며, recvfrom() 함수를 이용하여 데이터를 수신할 때 구조체 변수에 클라이언트의 주소 정보가 저장되어 있기 때문에 이것을 이용하여 클라이언트로 데이터를 전송할 수 있다.

```
/* Packet processing */
// Receive request from client, and process(get result from kernel and send back to client) received request.

while (1) {

    printf("Waiting for further request.\n\n");

    // Receive packet
    if (recvfrom(service_sock_fd, buff_rcv, MAX_PACKET_LENGTH, 0, (struct sockaddr*) & client_addr, &client_addr_len) < 0) {
        printf("Client packet receiving failure.\n");
        sendto(service_sock_fd, &net_error, 1, 0, (struct sockaddr*) & client_addr, (socklen_t)client_addr_len);
        printf("Waiting for further request.\n\n");
        continue;
    }

    printf("-----------------------------------------------------------------\n");
    printf("Data received.\n");

    // Parse packet
    name_length = (unsigned short*)(buff_rcv + 1);
    memcpy(name, buff_rcv + 3, ntohs(*name_length));
    *(name + ntohs(*name_length)) = 0;
    ipv4_addr = buff_rcv + 3 + ntohs(*name_length);

    // Verify Checksum
    if (!checksum_verify(buff_rcv, ntohs(*name_length))) {
        printf("%x\n", buff_rcv[ntohs(*name_length) + 4 - 1]);
        printf("Checksum error.\n");
        printf("-----------------------------------------------------------------\n\n");
        sendto(service_sock_fd, &net_error, 1, 0, (struct sockaddr*) & client_addr, (socklen_t)client_addr_len);
        printf("Waiting for further request.\n\n");
        continue;
    }
```

추가로 아래의 코드를 살펴보면 register, deregister, get 등이 이루어질 때 오류가 발생할 시 작업을 멈추고 오류 메시지를 출력하도록 하여 사용을 더 편리하게 하였다. 캡쳐는 register에만 해당하지만 나머지 부분도 이와 같은 식으로 작성하였다.

```
switch (*type) {

    // add
case MSG_REGISTER: {
    result = send_add_message(name, ipv4_addr[0] << 24 | ipv4_addr[1] << 16 | ipv4_addr[2] << 8 | ipv4_addr[3], buff_send);

    // Kernel failure
    if (result == MSG_FAILED) {
        kernel_failure[0] = MSG_REGISTER_RESPONSE;
        if (sendto(service_sock_fd, kernel_failure, 2, 0, (struct sockaddr*) & client_addr, (socklen_t)client_addr_len) < 0) {
            printf("Failed sending error to client.₩n");
        }
    }

    // Kernel network error
    else if (result > 0) {
        net_error[0] = MSG_REGISTER_RESPONSE;
        if (sendto(service_sock_fd, net_error, 2, 0, (struct sockaddr*) & client_addr, (socklen_t)client_addr_len) < 0) {
            printf("Failed sending error to client.₩n");
        }
    }

    // Success
    else {
        length = 9 + ntohs(*(unsigned short*)(buff_send + 2));
        if (sendto(service_sock_fd, buff_send, length, 0, (struct sockaddr*) & client_addr, (socklen_t)client_addr_len) < 0) {
            printf("Failed sending answer to client.₩n");
        }
    }
    break;
}
```

Client

UDP 클라이언트 프로그램은 서버 프로그램과 비슷하지만, 소켓에 주소 정보를 할당하는 bind() 함수를 사용하지 않기 때문에 UDP 서버 프로그램보다 간단하다. 클라이언트 프로그램의 대략적인 순서도는 다음과 같다.

1. 소켓을 초기화한다(socket)

2. 서버로 데이터 전송(sendto)

3. 서버로부터 데이터 수신(recvfrom)

4. 소켓 연결 종료(close)

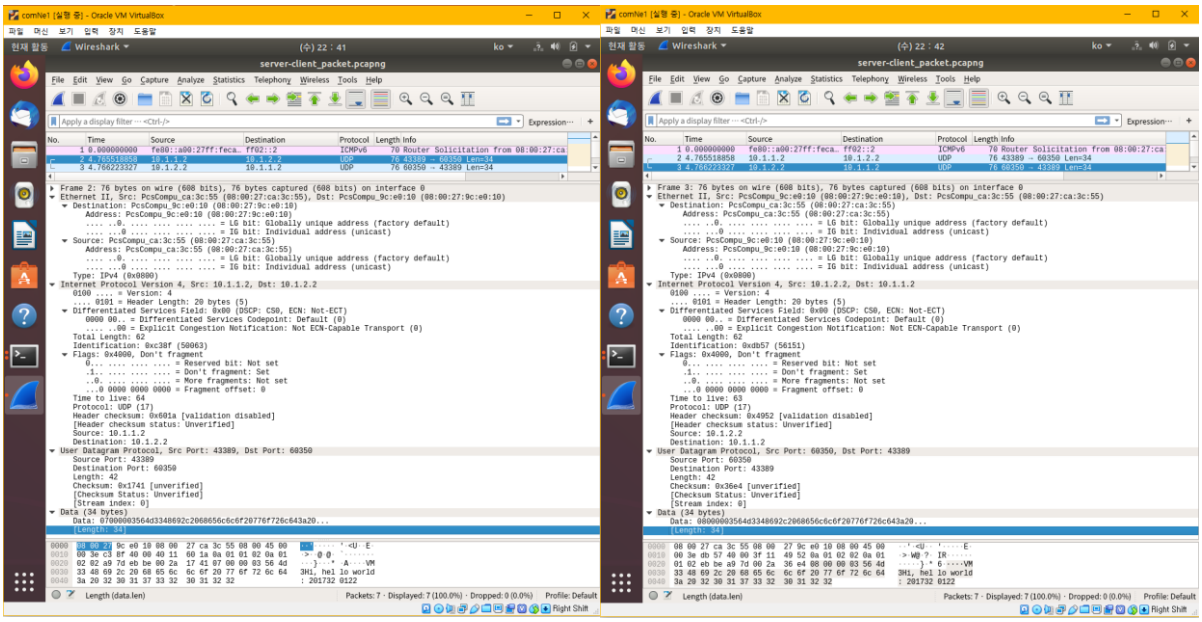
클라이언트에서 먼저 서버로 데이터를 전송하기 때문에 sendto() 함수를 먼저 사용한다. 서버는 클라이언트에서 전송한 데이터를 수신하고 다시 클라이언트로 데이터를 전송하므로 recvfrom() 함수를 사용하여 서버로부터의 응답 데이터를 수신한다.

UDP 클라이언트 프로그램은 주소를 add, del, get, verify하는 등의 기능을 가지며 이들을 각각 하나의 함수로 만들었다.

```
void do_verify(char* name) {

    /* Declaration */

    int sock_verify;
    struct sockaddr_in dest;
    char output[MAX_PACKET_LENGTH], result[MAX_PACKET_LENGTH];
    char ipv4_addr_string[MAX_IPV4_STRING_LENGTH];
    int name_len = strlen(name);
    int message_len = strlen("Hi, hello world: ") + strlen(MY_ID);
    int dest_len;
    struct get_result* ptr = head;
    char buffer_name[MAX_NAME_LENGTH];
    char buffer_verify_test[MAX_STRING_LENGTH] = "Hi, hello world: ";

    /* Connecting to destination */

    // Socket creation
    if ((sock_verify = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        printf("Socket creation error.₩n");
        return;
    }

    // Socket configuration
    dest_len = sizeof(dest);
    memset(&dest, 0, dest_len);
    dest.sin_family = AF_INET;
    dest.sin_port = htons(PORT_PEER_VERIFICATION);

    // Get IPv4 address string
    if (head == NULL) {
        printf("There should be get request before requesting verification.₩n₩n");
        return;
    }
    else {
        while(ptr != NULL) {
            if (strcmp(ptr->name, name) == 0) break;
            if (ptr->next == NULL) {
                printf("There should be get request before requesting verification.₩n₩n");
                return;
            }
        }
    }
```

위의 코드는 프로그램이 get한 디바이스의 address를 verify하는 함수다. 수신한 주소가 verification 과정을 거쳐 가능한 오류들을 모두 피하고 저장되면 성공했다는 메시지를 출력한다. 나머지 기능들도 위와 비슷한 포멧으로 작성하였으며 main 함수에서 이들을 취합한다. 이 설명들을 제외한 부분은 사용자의 인풋을 제어한다.

Captured packet on VM1



VM2를 서버로 사용하며 이를 거쳐 클라이언트 VM1, VM3이 서로 통신한 패킷의 캡쳐본이다. UDP 패킷이며 각 클라이언트에 지정한 주소끼리 서로 데이터를 주고 받는 것을 확인할 수 있다. 주고 받는 데이터에 'Hi, hello world'와 학번 '2017320122'가 포함되어있다.