MOFI Implementation

using 6-to-4 Tunnel and Netfilter

June 2012

Ji-In Kim and Nak-Jung Choi

요 약

6-to-4 Tunneling 기법을 이용한 MOFI 구현에 대한 기술 문서로써 구현을 위해서 6-to-4 Tunneling 기법과 netfilter, iptables를 이용한 구현 설계와 구현 과정, 실제 구현에 대한 테스트 결과를 정리한 문서이다.

Table of Contents

1. 서론

6-to-4 터널링 기법 기반의 MOFI 구현은 Linux를 기반으로 MOFI를 real machine에 구현하여 MOFI 테스트 환경을 구축한다. MOFI 테스트 환경을 구축 함으로써 프로토콜 기능 검증, 설계 상의 여러 부족한 점을 보완이 가능하며, 특히, 6-to-4 터널링 기법을 사용함으로써 기존의 IPv6 응용 프로그램을 그대로 사용할 수 있을 뿐만 아니라 MOFI의 HID 기반의 end-to-end 통신 두 가지 요건을 모두 만족 시킬 수 있는 장점을 가진다. 그리고 netfilter를 사용함으로써, 직접적인 Linux kernel 코드에 대한 수정 없이 MOFI-stack 구현이 가능하다.

2. 6-to-4 Tunneling 기법을 이용한 MOFI 구현

MOFI에서의 통신의 기본이 되는 HID는 128bit로 정의되어 있다. 따라서, MOFI는 기존의 IPv6응용 프로그램을 재사용할 수 있는 장점을 가진다. 이는 Host 구현에서도 장점을 가지게 되는데 Linux kernel에서 별도의 MOFI stack에 대한 구현 없이 기존의 IPv6 stack을 MOFI stack으로 변환하여 사용할 수 있게 된다. 이 경우 Host의 수정을 최소화할 수 있는 장점을 가지며, 이러한 장점은 나중에 MOFI의 배포에 대한 오버헤드를 줄일 수 있는 하나의 방법이 될 수 있다.

본 6-to-4 tunneling 기법을 이용한 MOFI 구현에서는 아래 그림 1과 같은 네트워크 모델을 기본으로 하고 있다.

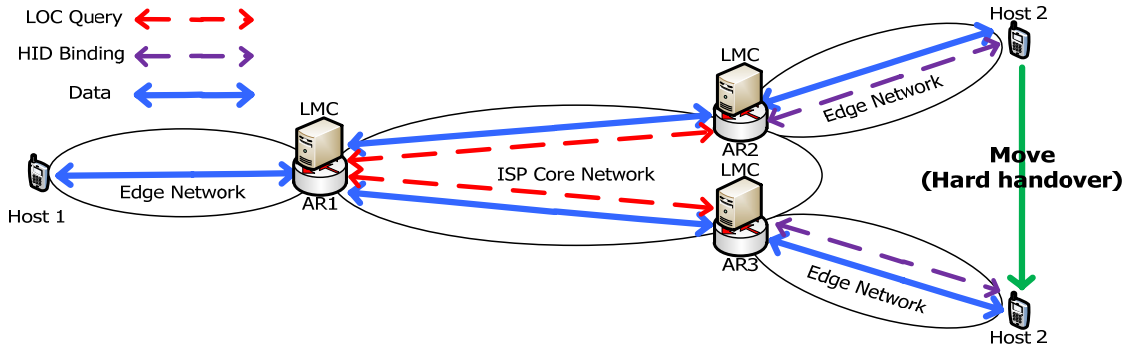

그림 1. 6-to-4 tunneling 기법을 이용한 MOFI 구현 네트워크 모델

2.1 Host 구현

MOFI Host는 Ubuntu Linux 9.10 운영체제를 기반으로 하여 6-to-4 tunneling 기법을 응용하여 개발하였다. 6-to-4 tunneling 기법은 tunnelling 기반의 IPv6/IPv4 전환 기법의 하나로 IPv4 망 내에서 독립된 IPv6망을 접속하는 것으로 IPv4 주소를 사용하지 않고 터널의 양 끝의 IPv6 prefix를 통해서 구분하는 기술이다. 이러한 6-to-4 tunneling을 이용한 MOFI Host의 구현은 아래 그림 2와 같다.

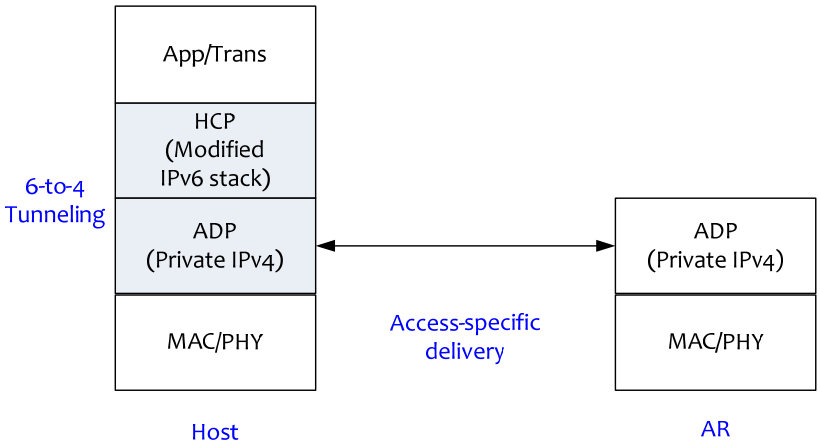

그림 2. MOFI Host 스택

MOFI Host 스택에는 기본적으로 MAC/PHY, ADP, HCP, App/Trans이 포함된다. 기존의 TCP/IP 프로토콜 스택의 IP 계층을 대신해서 ADP와 HCP 계층이 추가되었으며, 이들 프로토콜 스택을 구현하기 위해 6-to-4 tunneling 기법을 사용한다.

6-to-4 tunneling 기법을 사용하기 위해서 우선 가상의 논리적 인터페이스를 만들고, 이를 통해 주소할당 및 라우팅 처리를 하는 과정을 거치게 된다.

```
#/sbin/ip tunnel add mofiifc mode sit ttl <ttldefault> remote any local <local IPv4 address>
#/sbin/ip link set dev mofiifc up
#/sbin/ip addr add <HID address(AS number)>/16 dev mofiifc
```

위 명령은 6-to-4 tunneling 기법에서 mofiifc라는 MOFI를 위한 가상 인터페이스를 만들고, 이를 활성화하고, HID를 입력하는 과정이다. 원래의 6-to-4 tunneling 기법에서는 이 과정에서 IPv4 주소를 바탕으로

계산한 IPv6 주소를 삽입해야 하지만, 6-to-4 tunneling 기법을 이용한 MOFI 구현에서는 AS number를 포함하고 있는 MOFI HID를 입력하게 된다.

위와 같은 명령어를 사용해서 6-to-4 tunneling 기법을 이용한 MOFI Host를 구현한다.

2.2 AR 구현

MOFI AR은 MOFI에서 하나의 액세스 네트워크를 구성하는 게이트웨이의 역할을 하고 있기 때문에 두 개의 인터페이스를 통해 액세스 네트워크와 백본 네트워크와의 연결하는 역할을 수행한다. 이를 구현하기 위해 본 구현에서는 iptables와 netfilter를 이용한다. Iptables와 netfilter는 기본적인 Linux kernel에 포함되어 있으며, 사용을 위해서는 Linux kernel을 재 컴파일 해야 한다. 본 문서에서는 Iptables와 netfilter 설치와 관련된 내용은 생략하도록 한다. MOFI AR은 Ubuntu Linux 10.04의 Kernel version은 2.6.32.16을 기반으로 구현하였다.

그림 3은 AR의 프로토콜 스택을 나타낸다.

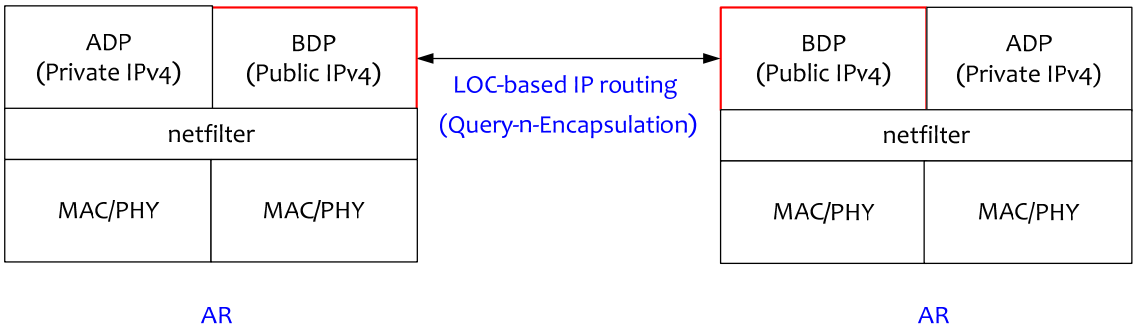

그림 3. MOFI AR 스택

MOFI AR 스택에는 MAC/PHY 계층과 ADP/BDP 계층 사이에 헤더 변환을 위해 netfilter가 존재를 하며, 이 Netfilter를 이용하여 패킷 헤더의 변경을 수행한다.

2.2.1 Netfilter란?

Netfilter는 네트워크 인터페이스를 지나가는 패킷을 hooking해서 정보 조작을 가능하게 해주는 Linux kernel에 포함된 기능으로, Netfilter의 hooking 포인트는 NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN, NF_IP_FORWARD, NF_IP_LOCAL_OUT, NF_IP_POST_ROUTING 총 5 부분으로 이루어져 있다.

그림 4는 Linux kernel에서의 netfilter hooking 포인트의 정확한 위치를 보여준다.

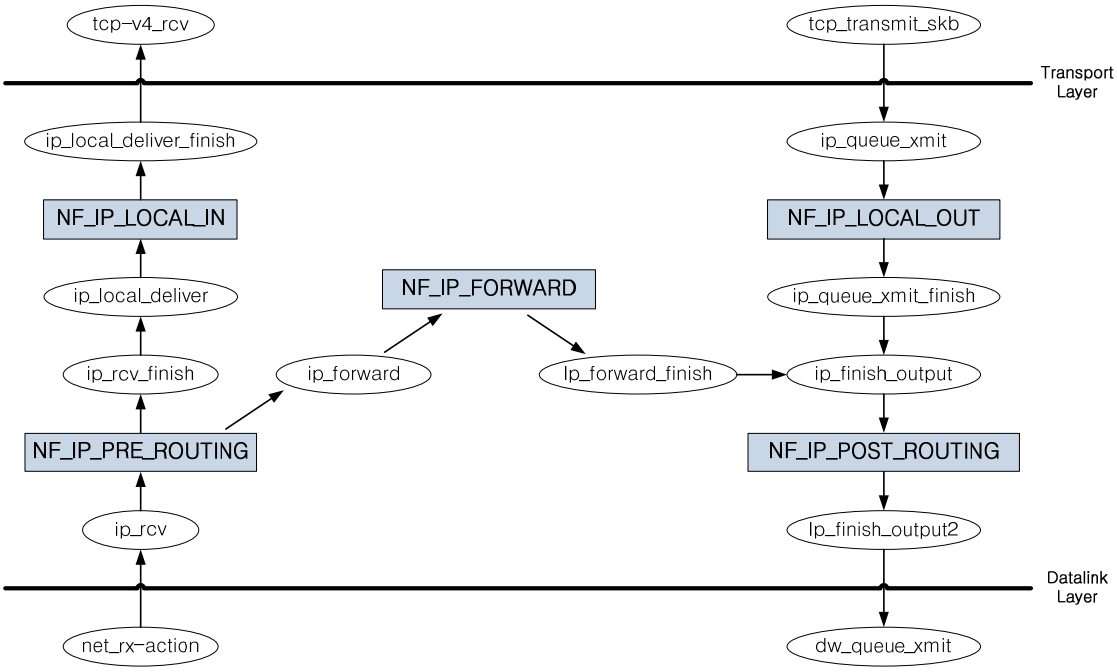

그림 4. Netfilter Hooking 포인트 위치

MOFI AR 구현을 위해서는 NF_IP_POST_ROUTING 부분을 hooking하여 6-to-4 tunneling 기법에서 이용하는 tunnelling mapping server를 거치지 않고 액세스 네트워크의 패킷이 백본 네트워크를 통해 도착지 AR에 도착할 수 있도록 패킷을 조작한다.

각각의 hooking 포인트에 대한 설명은 아래 표 1에 나와 있다.

표 1. Netfilter의 Hooking 포인트

Hooking 포인트	설명
NF_IP_PRE_ROUTING	네트워크 인터페이스로 들어오는 모든 패킷을 처리
NF_IP_LOCAL_IN	현재 노드로 전송된 패킷만 처리
NF_IP_FORWARD	도착지 주소가 다른 패킷만 처리
NF_IP_LOCAL_OUT	다른 노드로 전달되는 패킷만 처리
NF_IP_POST_ROUTING	네트워크 인터페이스에서 나가는 모든 패킷을 처리

2.2.2 Iptables와 Netfilter의 이용

MOFI AR을 구현하기 위해서는 우선 iptables를 이용하여 패킷 전달을 설정한다. 이는 기존의 iptables의 기능을 그대로 이용한 것으로 특별히 이번 구현을 위해 수정하거나 추가한 부분은 없다.

```
#echo “1” > /proc/sys/net/ipv4/ip_forward

#iptables –t nat –A POSTROUTING –s <HOST Private IPv4> -j SNAT –to <AR Public IPv4>

#iptables –t nat –A PREROUTING –d <AR Public IPv4> - j DNAT –to <HOST Private IPv4>
```

위 명령은 iptables의 ip_forward 기능을 이용하여 액세스 네트워크와 백본 네트워크를 연결하는 것을 나타낸다. Linux kernel 레벨에서 ip_forward 기능은 활성화하고, iptables 명령을 이용하여 Local IPv4 address 주소로 들어오는 모든 패킷을 netfilter의 기능을 이용하여 백본 네트워크로 통하는 인터페이스인 eth1의 NF_IP_POST_ROUTING단으로 전달한다.

2.2.3 netfilter를 이용한 패킷 헤더 변경 소스 코드

Netfilter를 이용한 패킷 헤더 변경을 위해서는 다음과 같은 소스코드가 필요하다.

```
//module header
#include <linux/errno.h>
#include <linux/file.h>
#include <linux/if_ether.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/module.h>
#include <linux/skbuff.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/netfilter_ipv6.h>
#include <linux/in.h>
#include <linux/in6.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/ip.h>
#include <linux/ipv6.h>
#include <net/tcp.h>
#include <asm/uaccess.h>
#include <asm/io.h>


static struct nf_hook_ops netfilter_ops1;


unsigned int main_hook(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in, const struct
net_device *out, int (*okfn)(struct sk_buff*))
{
        struct iphdr *iph = ip_hdr(skb);
```

```
        if(iph->saddr == in_aton("192.168.0.100"))
        {
                iph->daddr = in_aton("155.230.105.217");
                ip_send_check(iph);
        }

        return NF_ACCEPT;
}


//using two hooking method
unsigned int main_hook2(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in, const struct
net_device *out, int (*okfn)(struct sk_buff*))
{
        struct iphdr *iph = ip_hdr(skb);

        if(iph->saddr == in_aton("192.168.0.100"))
        {
                iph->daddr = in_aton("155.230.105.219");
                ip_send_check(iph);
        }

        return NF_ACCEPT;
}

int init_module()
{
        int ret = 0;
        char * argv[] = {"/run_app", NULL};
        char * envp[] = {"HOME=/","TERM=linux","PATH=/sbin:/usr/sbin:/bin:/usr/bin",NULL};

        ret = call_usermodehelper("/run_app", argv, envp, 1);

        printk("ret = %d\n",ret);
        if(ret == 0)
        {
```

```
                netfilter_ops1.hook = main_hook;
                netfilter_ops1.pf = PF_INET;
                netfilter_ops1.hooknum = NF_INET_FORWARD;
                netfilter_ops1.priority = 1;
                nf_register_hook(&netfilter_ops1);

                return 0;
        }

        else
        {
                netfilter_ops1.hook = main_hook2;
                netfilter_ops1.pf = PF_INET;
                netfilter_ops1.hooknum = NF_INET_FORWARD;
                netfilter_ops1.priority = 1;
                nf_register_hook(&netfilter_ops1);

                return 0;
        }

}
void cleanup_module()
{
        nf_unregister_hook(&netfilter_ops1);
}
```

그림 5. Netfilter를 이용한 헤더 변경 소스코드

그림 5는 Hooking Point인 NF_IP_POST_ROUTING을 통해서 패킷 헤더를 변경 시키기 위한 코드로 이를컴파일 하여 실행하면 헤더 변경을 수행한다. Hooking 부분을 2군데로 둔 경우는 현재 Test환경에서 AR2,3로 두었기 때문에 두군데로 가는 경우에 따라서 Hooking 포인트가 달라 지기 때문이다.

아래 그림 6은 위에서 작성한 프로그램을 통해 Netfilter hooking이 제대로 일어났는지 확인하기 위한 Wireshark를 통한 패킷 캡쳐 결과이다.

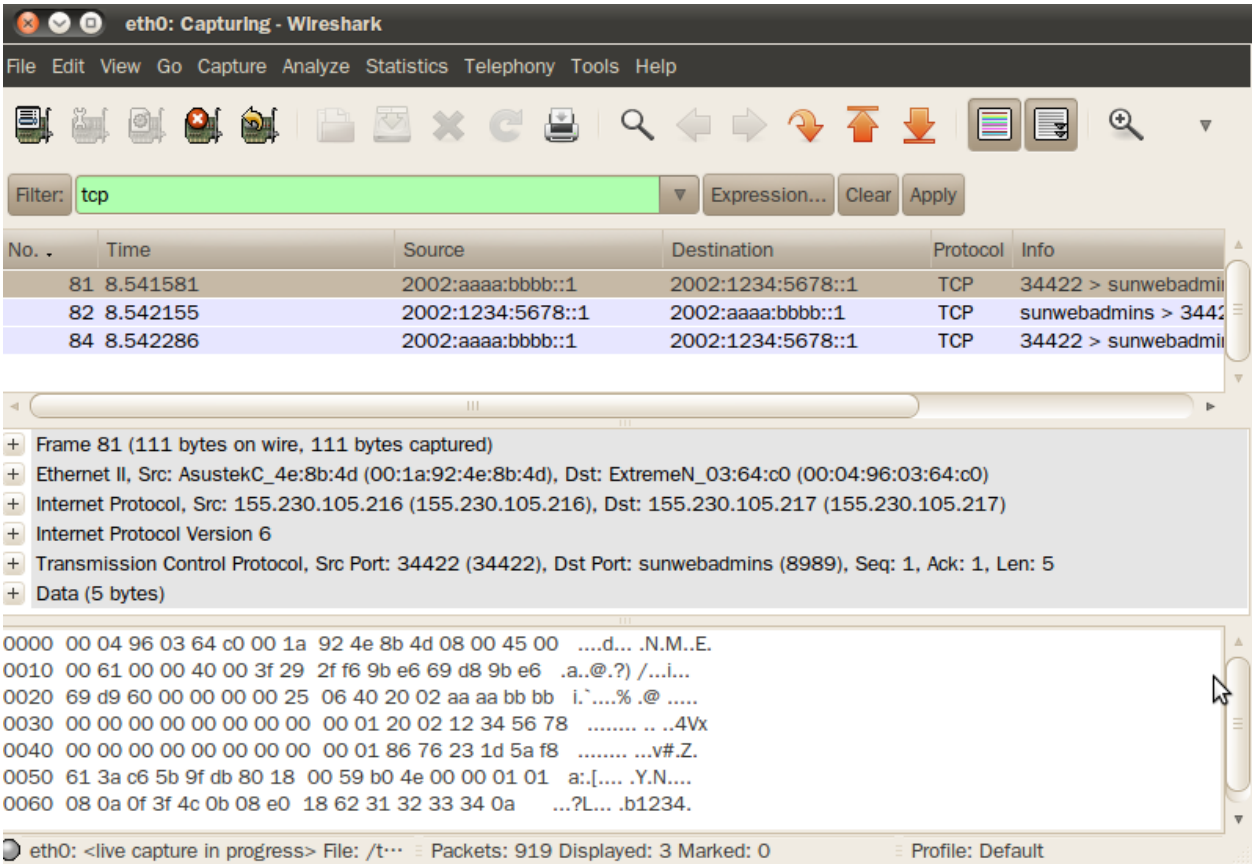

그림 6. Netfilter Hooking 확인을 위한 Wireshark 패킷 캡쳐 결과

위의 패킷 캡쳐 결과를 보면 프로그램 실행 후 기존의 6-to-4 tunneling 기법에서 사용하는 tunnelling mapping 서버가 아닌 도착지 AR의 주소(155.230.105.217)로 된 패킷 헤더로 변경이 된 것을 확인할 수 있다.

2.3 SMDC 구현

2.3.1 SDMC 란?

MOFI 에서 사용되는 기술은 HOST의 AS 정보는 변함 없이 Location 기반의 통신이 가능하게 하는 것이다. 그것을 위해서 HOST가 어떤 AR 과의 통신을 하고 있는지, 즉 HOST의 Location 정보를 알 수 있도록 해주는 Control Packet의 역할을 해주는 SDMC가 필요하다.

2.3.2 SDMC 구현 (HID Binding Protocol)

HID Binding Protocol(HBP)는 HOST가 AR에 접속을 하게 되었을 때 Binding Update를 통해서 자신이 연결 되었음을 LMC에 알려주는 역할을 하는 Protocol이다. 이것은 UDP 소스를 통해서 LMC쪽으로 Packet을 전달하게 되고 그 Packet을 받은 LMC에서 Binding Update가 되는 것이다. AR은 LMC를 통해서 현재 자신의 LOC 정보에 붙어 있는 HOST가 무슨 AS number를 가지고 있는지 알게 된다.

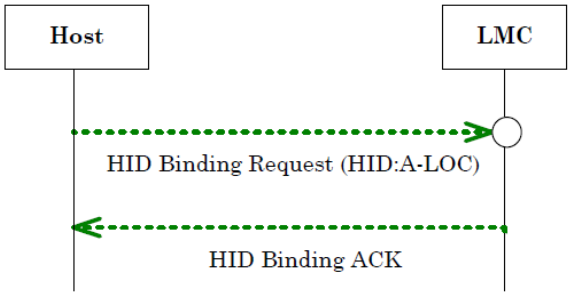

그림 7. HID Binding Update

2.3.3 SDMC 구현 (LOC Query)

LOC Query 는 Data를 보내는 Host 측면에서 Data의 목적지 AS number를 바탕으로 특정 HOST가 어떤 LOC 정보를 가지고 있는지 즉 어느 AR에 있는지 확인하기 위한 Query 문이다. '그림 1'을 바탕으로 설명을 하자면 Host1에서 Data packet을 전송을 하려고 하는 상태에서 Host2의 LOC 정보를 알기 위에서 AR1에 있는 LMC에서 AR2의 LMC와 AR3 LMC로 Query를 전송하게 된다. 그렇게 되면 정보를 받은 AR2,3의 LMC는 자신에게 일치 AS number를 가진 Host가 있는지 확인을 하게 된다. 그리고 일치하는 AS number Host가 존재 하는 경우에는 Query에 대한 Ack를 보내게 되고 AR1에서는 Ack를 받은 LMC쪽에 있는 AR로 Data packet을 전송하게 된다.

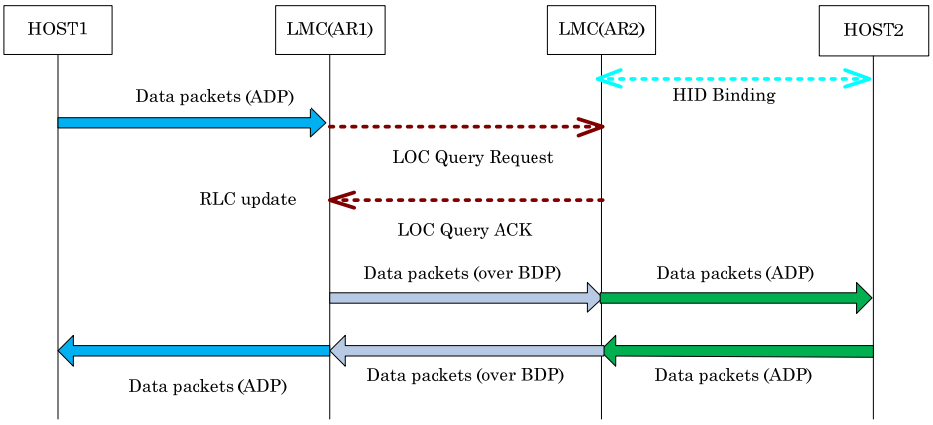

그림 8. LOC Query and DATA Delivery

2.3.4 SDMC 환경 설정

현재 구현환경의 한계상 LMC는 분리 되어 있는 형태가 아닌 각각의 AR 마다 LMC 정보를 직접 관리하는 형태의 구현이 이루어 졌다. 그로 인해서 지금 기존에 사용하던 AR Public IPv4 정보 이외에도 LMC 의 역할을 하기 위한 IPv4을 하나 더 할당 해줄 필요가 있다. Ethernet card를 하나 더 추가 해서 그 역할을 할 수 있는 IP할당 방법도 있겠지만 현재 Packet Forwarding 및 netfilter 후킹을 사용한 상태이므로 그 방법으로 하게 되면 통신 세팅하기가 힘들었다. 그렇기 때문에 간편하게 eth0에 하나의 IP를 더 할당하는 방법을 이용하였다.

```
#ifconfig eth0:1 <LMC IPv4 Address>
```

이 방법은 DNS하나에 여러 개의 IP를 할당하는 방법이지만 우리가 사용하는 MOFI 사용에 사용하는 방법으로 이용을 하였다.

3. MOFI 구현 테스트 결과

3.1 구현 Test 환경

6-to-4 Tunneling 기법을 이용한 MOFI 구현을 확인하기 위한 테스트 환경은 아래 '그림 9'과 같다.

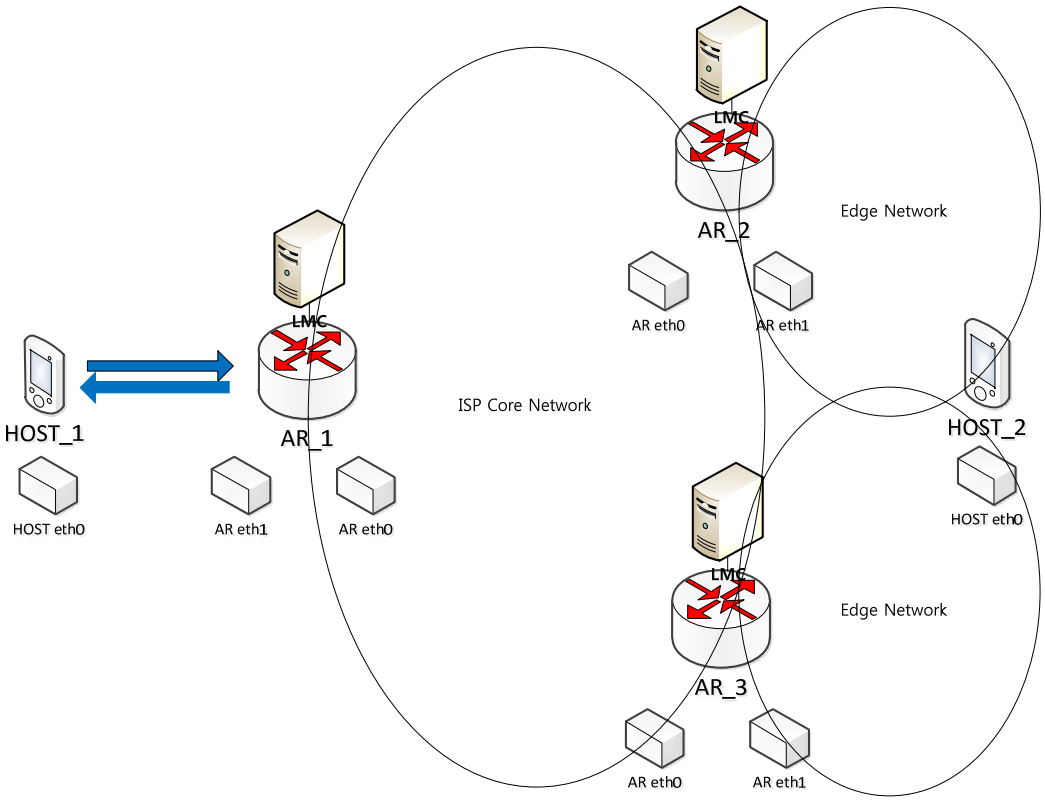

그림 9. 6-to-4 Tunneling 기법을 이용한 MOFI 구현 테스트 환경

'그림 9'에서 보면 알 수 있듯이 일단 HOST1은 AR1에서 통신을 시작하는 형태이고 HOST2는 AR2와 AR3 사이를 다니면서 움직이는 형태이다. AR2, AR3로 구현한 이유는 MOFI의 Mobility와 LOC 위치 변경을 보여 주기 위한 방법으로 위와 같은 Test환경을 구축하였습니다.

Test환경에 올라간 OS들은 모두 Linux Ubuntu 10.04 버전이 사용되었으며 kernel version 2.6.32로 설정하였습니다. Ubuntu 버전은 구현상 크게 상관없으나 Netfilter와 Iptables를 사용함으로 인해서 kernel version은 이상을 사용해야 이용하시는데 크게 문제가 없으실 듯 합니다.

Test를 위해서 Webcam을 이용하는 Streaming 서비스와 Query형태를 보여줄 수 있는 지도와 Wireshark를 통한 packet capture를 이용하였습니다.

3.2 구현 Test 세팅

Hbp.c라는 실행 파일은 AR2,3에서 사용을 해야 한다. Hbp.c는 HOST2가 어디에 접속을 하는가에 대한 것을 판단하는 부분과 AR1에서 Query가 날라와서 어디에 있는 확인을 요청하는것에 대한 정보를 확인할 수 있는 부분으로 구성되어 있다. Hbp_clnt.c 는 Host2에서 사용을 하여서 AR2 혹은 AR3에 접속을 한다는 메시지를 주는 것이다. 마지막으로는 ar.c는 AR1에서 실행을 하여 커널 모듈 변경을 수행하며 Host2의 LOC 정보를 확인하기 위한 Query message를 전송하는 기능을 수행합니다.

3.3 구현 Test 결과

그림 10은 LOC Query가 날아간다는 것을 표현할 방법이 없어서 저희가 Query에 맞추어 packet 경로를 지도로 나타낸 모습입니다.

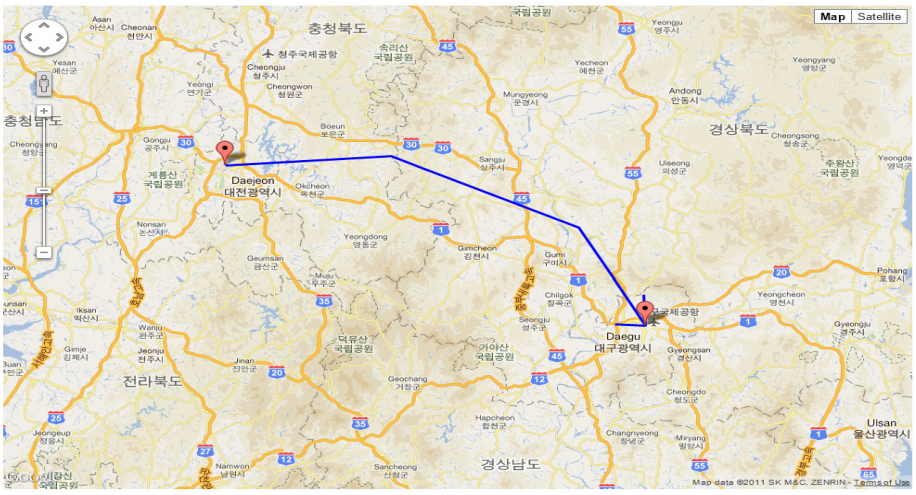

그림 10. LOC Query 경로 표시

그림 11은 한곳의 AR2에서 data가 올 때 packet capture를 나타낸 모습입니다. Wireshark 를 변형해서 IPv6정보가 아닌 HID를 표현을 하였습니다.

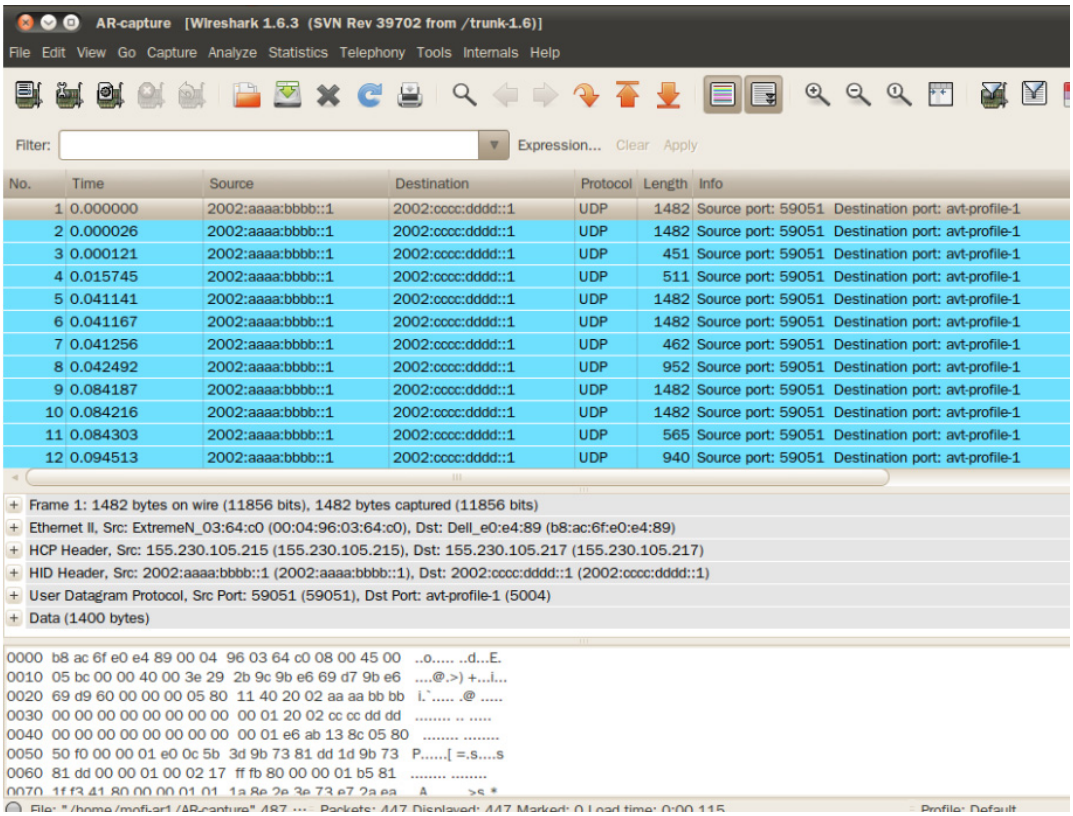

그림 11. AR Wireshark 캡쳐 장면

그림 12는 Host에서 실행한 webcam streaming service 모습입니다.

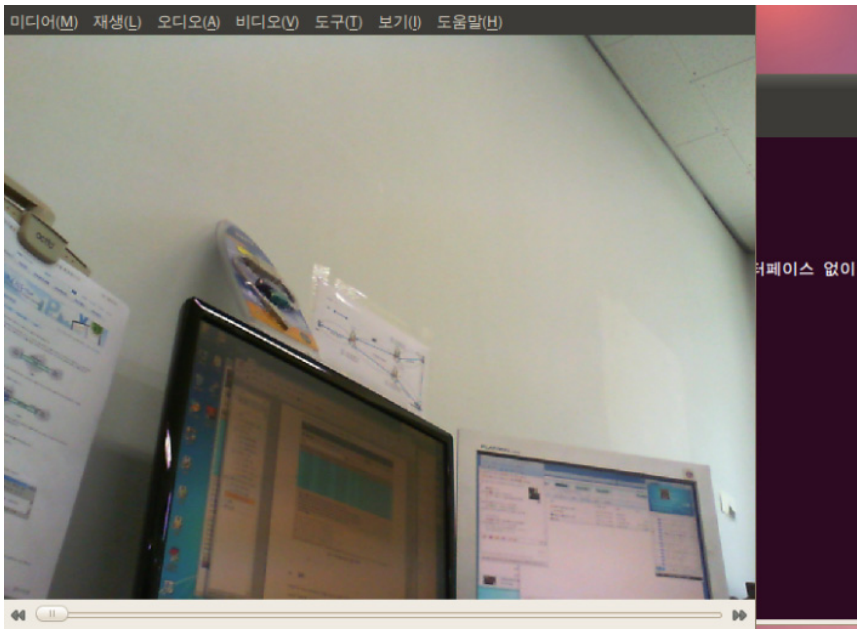

그림 12. Webcam Streaming service

4. 결론

이상으로 6-to-4 tunneling 기법을 이용한 MOFI 구현에 대하여 설명하였다. 현재 방법 적으로 100%로 SDMC를 구현했다고 볼 수는 없는 상황이다. 하지만 패킷의 전송 및 동영상 스트리밍 테스트를 통해서 정상적으로 구현이 완료된 것을 확인할 수 있었다.

참고문헌

[1] Mobile Oriented Future Internet (MOFI), http://www.mofi.re.kr/

[2] IETF Host Identity Protocol (HIP) WG, http://www.ietf.org/html.charters/hip-charter.html

[3] IETF RFC 4843, An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID), April 2007

[4] IETF RFC 3775, Mobility Support in IPv6, June 2004

[5] IETF RFC 5213, Proxy Mobile IPv6, August 2008

[6] Linux netfilter Hacking HOWTO, July 2002