國立交通大學

資訊工程系

碩士論文

在 NCTUns 模擬器平台上提供 IPv6 的 Simulation 和 Emulation 的功能

Supporting the Simulation and Emulation of IPv6 on the NCTUns Network Simulator

研究生:周家敏

指導教授:王協源

中華民國九十四年六月

中文摘要

隨著全球資訊網際網路的使用者急遽增加，原本的 IPv4 的位址空間(addressing space) 已漸漸不足以應付網際網路使用者的需求。於是，下一個世代的網際網路協定 IPv6 已被訂定出來以提供更大的位址空間及增加網際網路協定的使用彈性和能力。IPv6 已逐漸獲得重視並有相當多相關此一通訊協定的研究。然而，若在真實環境下進行網路研究要花費大量的時間和金錢，且結果不易重複呈現。在模擬的環境下，所有的網路狀況及設定都是可以簡單地重覆呈現且花費低廉，因此使用者不僅可以容易地得到重覆的實驗結果，還可以節省大量的時間和金錢。

基於上述關於在模擬平台上進行研究的優點，我們開始研究如何在網路模擬器的平台下開發 IPv6 的模擬。NCTUns 是一個創新的模擬器，它整合了 OS kernel、simulation engine 和應用程式。所以在本篇的論文中，可分成三大部份：第一部份是在 NCTUns 環境下提供 IPv6 的模擬。在此部份中，我們利用修改 Linux kernel 和網路模擬器的 engine 及 module code 達成在 NCTUns 上提供純 IPv6 的模擬。第二部份是在 NCTUns 環境下同時提供 IPv4 和 IPv6 的模擬，在這一部份中，我們增加二個 module 來達成 IPv4 和 IPv6 的混合模擬。最後一部份則是在 NCTUns 環境下提供 IPv6 的 Emulation，在此部份,我們修改 Linux kernel 和增加一個 IPv6 emulation daemon。修改 Linux kernel 是為了使 Linux 具備類似 Divert Socket 的功能。增加 IPv6 emulation daemon 是為了使 simulation machine 能正確的和 external host 建立 connection 及傳輸封包。

Abstract

Due to the rapid growth of Internet, IPv4 (Internet Protocol Version 4) may not provide enough address space in the future. Thus, IPv6, the next generation of Internet protocol, has been proposed to deal with this issue. IPv6 has opened many novel aspects of researching areas and become a hot topic. Therefore, developing an IPv6-based evaluation environment is valuable and important. In the network-research domain, simulation is a useful approach for users to evaluate the performance of IPv6 with acceptable cost. This thesis describes how to make NCTUns support IPv6 simulations and emulations, including the mixed networks of IPv4 and IPv6.

Since NCTUns is a novel network simulator that integrates the OS kernel, the simulation engine, and user-level applications into a whole simulation environment, the work of this thesis involves the modifications of Linux kernel, the simulation engine, and the development of new protocol modules in NCTUns. First of all, to support the pure IPv6 simulation in NCTUns, the Linux kernel, the simulation engine and modules are modified to provide the pure IPv6 simulation. Second, to support the simulation of mixed networks of IPv4 and IPv6 in NCTUns, two modules are developed to accomplish the conversion between IPv4 packets and IPv6 packets. Finally, to support the IPv6 emulation in NCTUns, we modified the Linux kernel to let it be capable of redirecting some specified packets to the user-level applications (This mechanism is called “divert socket” in FreeBSD) and wrote new user-level programs to allow packets with different IP versions to be transmitted correctly between a real host and a simulated host.

致謝

感謝恩師王協源教授二年來對我的悉心指導，研究所二年，由於老師給我們的札實訓練，讓我在專業領域上獲益良多，相信以這二年札實的，不論是在工作或研究上都會是重要的基石。

感謝林華君教授以及廖婉君教授撥冗來到交通大學進行口試。

感謝父母親多年來的支持，讓我在求學過程中順暢無後顧之優，能專心於學業之中。

最後感謝網路與系統實驗室的所有成員，因為有你們的陪伴以及彼此間的鼓勵，讓我在研究所二年不僅學到很多，生活也很充實快樂。

Table of Contents

List of Figures

Part I: Supporting the pure IPv6 simulation in NCTUns

1. Introduction

Nowadays, due to the exponential growth of Internet users around the world, the address space provided by the Internet protocol version 4 (IPv4) may run out someday. To overcome this problem, IPv6, the next generation of the network-layer protocol, has been proposed, and more and more researchers devote themselves to studying IPv6-related issues.

To do IPv6-related researches, such as Mobile IPv6, AODV IPv6 and so forth, one has to conduct lots of experiments. Nevertheless, conducting experiments in the real world is not an easy job. It takes not only a great deal of time but much money as well. On the contrary, if the experiments of these IPv6-related researches can be conducted by using a network simulator, researchers are able to save lots of money and time to gain the almost equivalent experimental results. Based on this reason, we analyzed how to support IPv6 simulation in NCTUns, a novel simulator which has lots of merits, including an easy-to-develop environment, a diversity of protocols, and so forth. In part I, we describe what and how we did to support pure IPv6 simulations.

2. The Overview of the NCTUns Simulator

Figure 1.1 shows the overview of how NCTUns network simulator works.

When a daemon program (a “daemon program” usually refers to as a user-level program running on the background) wants to send a packet from one node to another during a simulation, it sends that packet from the user space to the kernel space. When the packet enters the kernel, its destination IP address is transformed to another special address format (called SSDD format) used by NCTUns to route it. After the packet passes the protocol stack in the kernel, it is placed in the queue of a tunnel interface, the device ID of which is ranged from 1 to 4095. At the same time, a special event packet that notifies the simulation engine of the arrival of a pending packet is created and placed into a special tunnel device, tOe0.

Each time when the simulation engine gets the resource of CPU, it will first check if any packet is queued in the FIFO queue of tOe0 by the read() system call. If so, the simulation engine will read the special packets from tOe0 and knows which tunnel interfaces have pending packets. Next, the simulation engine reads those pending packets from tunnel interfaces and sends them to the protocol modules associated to those tunnel interfaces. A tunnel interface has a group of protocol module instances to form its protocol stack below IP layer, such ARP module, MAC802.3 module, etc. After a packet passes through a protocol stack simulating protocols under IP layer, the simulation engine sends it back to the kernel by the write() system call to make it processed by protocols at IP layer and above. Since NCTUns adopts real-world TCP/IP protocol stacks and real application programs, this simulator provides higher fidelity and accuracy simulations than traditional ones.

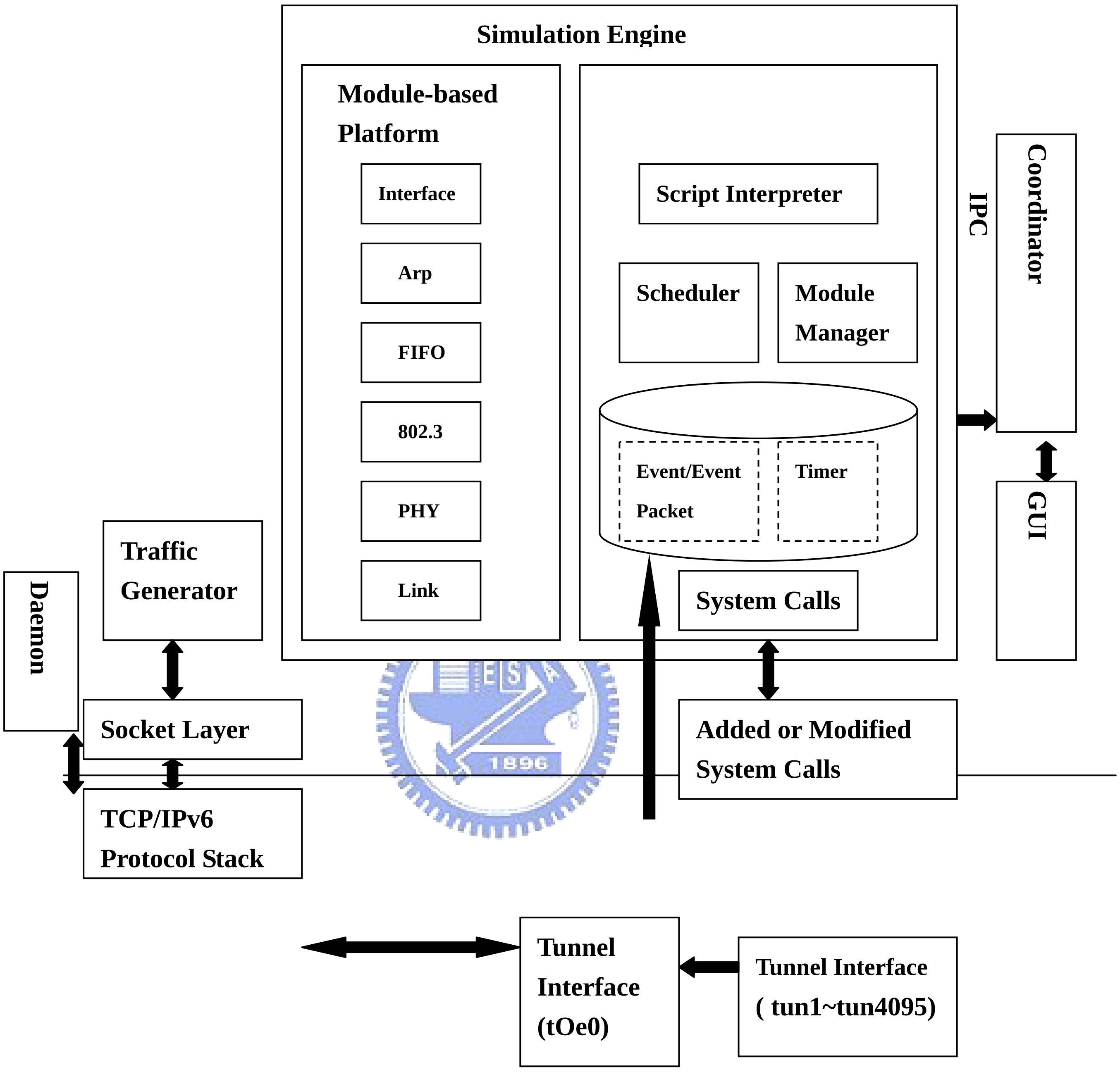


Figure 1.1: The simulation network topology

3. IPv6 Address and IPv6 Routing Scheme

Because the Linux kernel and most of the operating system have only one routing table to store the routing information of the whole network, it may result in conflicts of routing entries if one tries to simulate multiple network nodes on a machine. For example, both node A and node B have their own routing entries to node C. Assume that nodes A and B prefer nodes D and E as their next hops to node C, respectively. In such case, two different routing entries for node C exist in the kernel's routing table simultaneously. One suggests that the kernel has to use node D as the next hop to route packets destined to node C, but the other suggests that the next hop to node C is node E. To deal with this problem, we designed a special address scheme to make routing entries of all simulated nodes can be stored together in the kernel's routing table. The following is a brief description for the address scheme and the routing scheme for IPv6 in NCTUns.

I. IPv6 address scheme:

Before the routing scheme is explained, we have to introduce the basic IPv6 address format used in NCTUns first. Every tunnel interface in NCTUns is assigned with an IPv6 address in the special format. After the assignment of IP addresses for each tunnel interface, we are able to send the packet to one node assigned with this kind of IPv6 address. The brief description of the format is shown as follows:

3fff: 0: 0: NetID: 0: 0: NetID: HostID

II. IPv6 routing scheme:

To simulate the routing of a packet among simulated nodes on a single machine, we designed a novel IP format called SSDDv6. SSDDv6 makes a destination IP address contain enough info, such as the source subnet ID, the source host ID, the destination subnet ID and the destination host ID. With the help of this format, a routing entry describes not only the IP addresses of a next hop and a destination, but also the IP address of a source node. With this scheme, although the kernel still has only one routing table, the routing table of each simulated node is able to be put together with SSDDv6 format.

Network route:

3fff:0:0:0:SrcNetID:SrcHostID:DstNetID::

We execute commands with this address format to build necessary routing entries to make the routing of packets correct during simulation. Here is an example:

```
Route –A inet6 add 3fff:0:0:0:0001:0101:0002::/112 dev tun1
```

We use the above command to direct packets, which are generated from a subnet 1 and destined to the subnet 2, to the tun1 interface.

In this scheme, the kernel is capable of routing packets to correct simulated network nodes by putting packets into correct tunnel interfaces. As we mentioned previously, a tunnel interface is associated with a simulated network node. When a packet passes the kernel, the kernel alters the destination IP address of this packet to an IP address with the format of SSDDv6. Since routing entries of each node are built with the format of SSDDv6, the routing entry for this packet is able to be found successfully, and the kernel can direct this packet to the correct tunnel interface.

3.1. An IPv6 Routing Example

To make readers more precisely understand the routing scheme we have implemented in NCTUns, we take an example of how IPv6 routing works in NCTUns in the following subsections. In subsection 3.1.1, we state the IPv6 address scheme in NCTUns and how an IPv6 address is altered. In subsection 3.1.2, we describe how to set up the IPv6 routing table in NCTUns.

As shown in figure 1.2, the topology is that two hosts are connected with an intermediate router. Node 1 is assigned an IPv6 address 3fff::1:0:0:0100:0101, and node 2 is assigned an IPv6 address 3fff::2:0:0:0100:0202. The intermediate router has two interfaces. One is assigned an IPv6 address 3fff::1:0:0:0100:0102, and the other is assigned an IPv6 address of 3fff::2:0:0:0100:0201.

3.1.1. IPv6 Address Scheme

In figure 1.2, when node 1, whose IP address is 3fff::1::0100:0101, runs the ping program for node 2, it will execute the command like this, "ping6 3fff::2:0:0:0100:0202". When a packet generated by this ping6 program enters the kernel, the kernel modifies the destination IP address of this packet to 3fff::0001:0101:0002:0202 (SSDDv6 address). At this time, the kernel makes use of this SSDDv6 address to look up its routing table and discovers the correspondent tunnel interface. The kernel changes the SSDDv6 address back to the original IPv6 address (3fff::2::0100:0202) when it puts this packet into the queue of the correspondent tunnel interface. Then, the simulation engine captures the packet from the tunnel interface to simulate the processing of protocols below IP layer.

After the processing is finished, the packet is put back to the queue of another tunnel interface corresponding to the next-hop node by the write() system call. The packet then is sent to the protocol stack in the kernel as if it were received from a real NIC. At the same time, the tunnel interface routine first examines whether or not this packet has reached its destination node. If so, the destination IP address will not be altered. Otherwise, the tunnel interface routine modifies the destination IP address of this packet to SSDDv6 format (3fff::0100:0102:0002:0202). In the IP layer, the kernel looks up the routing table and finds that the packet doesn't reach its destination. So, the kernel redirect this packet to a tunnel interface based on the found routing entry. This process is repeated until the packet reaches its destination node. When the packet enters the tunnel interface corresponding to the destination node, the tunnel interface routine detects this packet has reached its destination node by finding that the source ID is the same as the destination ID in SSDDv6 format, for example, 3fff::0002:0202:0002:0202 in this case. Therefore, the tunnel interface routine transforms the destination IP address of this packet back to the original one and delivers the packet to the upper layer for further processing.

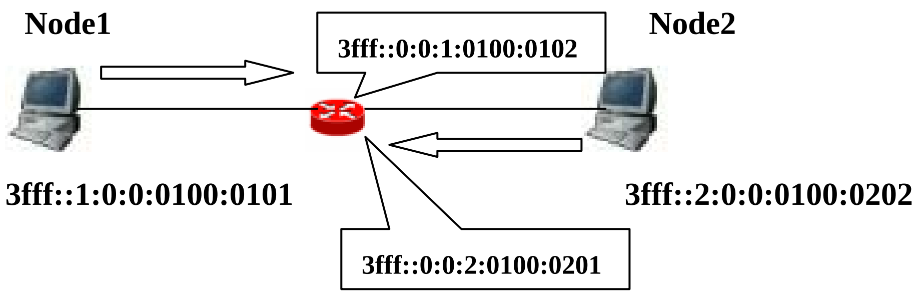


Figure 1.2: An example of IPv6 address

3.1.2. Setting the IPv6 Routing Table

Before simulation cases are run in NCTUns simulator, the routing table should be set up. After the entries are built up, a packet could be delivered to its correspondent tunnel interface by looking up the routing table. To build up the routing entries of the figure 1.2, we execute commands as follows:

```
route –A inet6 add 3fff::0001:0101:0001::/112 dev tun1
route –A inet6 add 3fff::0002:0202:0002::/112 dev tun4
route –A inet6 add 3fff::0001:0101:0002::/112 gw 3fff::0001:0101:0001:0102
route –A inet6 add 3fff:0002:0202:0001::/112 gw 3fff::0002:0202:0002:0201
…
…
```

After running these commands, we obtain the routing table as follow:

```
Destination   ags  Metric  Ref  Use  Iface   Next Hop
------------------------------------------------------------------
3fff::0001:0101:0002::/112  0   4   0   3fff::0001:0101:0001:0102
3fff::0002:0202:0001::/112  0   4   0   3fff::0002:0202:0002:0201
3fff::0001:0101:0001::/112  0   4   0   tun1
3fff::0002:0202:0002::/112  0   4   0   tun4
….
….
```

For letting users more understand of this mechanism, we describe the actions of looking up one of the correspondent routing entries as an example here. The packet from the node one to the node two is assigned with the destination address of 3fff:00:00:0002:00:00:0100:0202. Before the routing entries are looked up, the destination IPv6 address is replaced with the destination IPv6 address of SSDDv6 format (3fff:00:00:00:0001:0101:0002:0202). Then, the routing table listed above is looked up, and the correspondent tunnel interface is tun1. Although we merely describe how one entry is inquired, other routing entries could be looked up according to the same rules. Based on the built-up routing entries during the simulation, the packet can be directed to the correspondent tunnel interface correctly according to the destination IPv6 address prefix.

4. Kernel Modification

In chapter 3, we describe the overview of IPv6 address format in NCTUns. In this chapter, we explain the modification to the Linux kernel to support several mechanisms required by NCTUns. In sections 4.1 and 4.2, what works we did to make the SSDDv6 address scheme function correctly is explained. In section 4.3, what works we did to make the port mapping and translation function correctly is introduced.

4.1. IPv6 Address Translation and Source-Destination-Pair IPv6 Scheme

After the discussion in chapter 3, we now understand what an IPv6 address format in NCTUns looks like. Based on that concept, we are to show what we should modify in the Linux kernel to let our address scheme function correctly.

In the kernel function of inet_sendmsg() and inet_stream_connect() defined in the file of af_inet.c, we transform the IPv6 address into a SSDDv6 address format (3fff:0:0:0:SrcNetID:SrcHostID:DstNetID:DstHostID) to route a packet to its correspondent tunnel interface. Here, we only take the function of inet_sendmsg as an example.

```
int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
                size_t size)
{
        struct sock *sk = sock->sk;

//NCTUNS
/*
  * We intercept the destination address here
  * and modify the original destination address from 3fff:0:0:X:0:0:0100:XX format
  * to 3fff:0:0:0:SrcNetID:SrcHostID:DstNetID:DstHostID format.
  */
{
                …
                …
        //NCTUNS_V6
        if(sk->sk_family == 10)
        {
```

This means the packets belong to the type of IPv6.

```
            struct sockaddr_in6 *addr_v6 = (struct sockaddr_in6 *)msg->msg_name;
            struct in6_addr srcipv6;

            if(addr_v6 && (sk->nodeID > 0))
            {
                …
                …
                if(!((srcipv6.s6_addr32[0] ==0) && (srcipv6.s6_addr32[1] ==0) &&
                    (srcipv6.s6_addr32[2] ==0) && (srcipv6.s6_addr32[3] ==0)))
                {

                    addr_v6->sin6_addr.s6_addr[13] =
                    addr_v6->sin6_addr.s6_addr[7];
                    addr_v6->sin6_addr.s6_addr[12] =
                    addr_v6->sin6_addr.s6_addr[6];
```

We transform the destination address into the format of 3fff:0:0:0:SrcNetID: SrcHostID:DstNetID :DstHostID.

```
                    addr_v6->sin6_addr.s6_addr16[5] = srcipv6.s6_addr16[7];
                    addr_v6->sin6_addr.s6_addr16[4] = srcipv6.s6_addr16[3];
                    addr_v6->sin6_addr.s6_addr16[3] = 0;
                }

            }
        }
    }
    //NCTUNS_V6
    …
```

```
…
}
```

In the kernel function of inet_recvmsg() defined in the file of af_inet.c, we transform the SSDDv6 address format back to the original address format before the packet is transmitted up to the user-level program. The detailed codes are explained as follows:

```
int inet_recvmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
             size_t size, int flags)
{
      …
      …
      //NCTUNS_V6
{
      if(sk->sk_family == 10)
      {
            struct sockaddr_in6 *addr_v6 = (struct sockaddr_in6 *)msg->msg_name;
            if(addr_v6 && (sk->nodeID > 0) &&
(strncmp(sk->sk_prot->name,"RAW",3)!=0))
            {
```

We transform the special format (3fff:0:0:0:SrcNetID:SrcHostID:DstNetID:DstHostID) back to the normal format(3fff:0:0:NetID:0:0:0100:HostID).

```
                  addr_v6->sin6_addr.s6_addr[6]=addr_v6->sin6_addr.s6_addr[12];
                  addr_v6->sin6_addr.s6_addr[7]=addr_v6->sin6_addr.s6_addr[13];
                  addr_v6->sin6_addr.s6_addr16[4] = 0;
                  addr_v6->sin6_addr.s6_addr16[5] = 0;
```

```
                        addr_v6->sin6_addr.s6_addr[12] = 1;
                        addr_v6->sin6_addr.s6_addr[13] = 0;
                }

        }
          …
        …
}
//NCTUNS_V6
…
…
}
```

4.2. The Tunnel Interface

A tunnel Interface is a pseudo network interface. In other words, there is no physical network attached to it. A tunnel interface is regarded as being equivalent to a real Ethernet network interface. Therefore, NCTUns makes use of tunnel interfaces to simulate network devices on its simulated nodes. The modifications to tunnel interface include the supports for the SSDDv6 format and the divert socket. In the section, we describe how to make a tunnel interface support the SSDDv6 format. The introduction to the modification for the divert socket is explained in Part III.

In the kernel function of tun_net_xmit defined in tun.c, the destination address we obtain on the point of packet's entering tunnel interface is in a SSDDv6 address format (3fff:0:0:0:SrcNetID:SrcHostID:DstNetID:DstHostID). On the verge of packet’s leaving tunnel interface, the destination IPv6 address would be altered back to a normal IPv6 address (3fff:0:0:DstNetID:0:0:0100:DstHostID). Supposing the source node is also the destination node, the source IPv6 address will be modified into a special IPv6 address. Otherwise, the source and destination IPv6 address are all in normal address format before the packet is delivered from the tunnel interface. By the way, we also obtain the next gateway address and fill in that address in the MAC header for later use by NCTUns. The detailed info is shown as follows:

In the function tun_net_xmit() (defined in the file of tun.c):

```
/* Net device start xmit */
static int tun_net_xmit(struct sk_buff *skb, struct net_device *dev)
{
     struct tun_struct *tun = netdev_priv(dev);
     …
     …
     p_ipv6 = (struct in6_addr *)&ipv6->daddr;
     …
     …
//NCTUNS_V6
          if(ipv6->version == 6)
          {
                    …
                    …
                    nid1=mt_ipv6tonid(s_v6);
                    nid2=mt_ipv6tonid(d_v6);
```

We use the normal source and destination addresses to look up the corresponding node IDs.

```
            }
//NCTUNS_V6
//NCTUNS_V6
if(nid1 && nid2){
                if(nid1 == nid2){
                     int count=0;

                     if(ipv6->version == 6)
                          {
                          p_ipv6->s6_addr16[3] =
                          p_ipv6->s6_addr16[6];
                          p_ipv6->s6_addr16[4] = 0;
                          p_ipv6->s6_addr16[5] = 0;
                          p_ipv6->s6_addr[12] = 1;
                          p_ipv6->s6_addr[13] = 0;
                          p_ipv6 = (struct in6_addr *)&ipv6->saddr;
                          p_ipv6->s6_addr16[4] = d_v6.s6_addr16[3];
                          p_ipv6->s6_addr16[5] = d_v6.s6_addr16[7];
                          p_ipv6->s6_addr16[6] = p_ipv6->s6_addr16[3];
                          p_ipv6->s6_addr16[3] = 0;
                     }
                     …
                     …
                     return count;
     }
}
```

If this node is the destination node, we transform the SSDDv6 destination address (3fff:0:0:0:SrcNetID:SrcHostID :DstNetID:DstHostID) back to a normal IPv6 address (3fff:0:0:DstNetID:0:0:1:DstHostID) and alter the normal source IPv6 address to a SSDDv6 source address (3fff:0:0:0:SrcNetID:SrcHostID :DstNetID:DstHostID).

```
…

…

  p_ipv6->s6_addr16[3] = p_ipv6->s6_addr16[6];

  p_ipv6->s6_addr16[4] = 0;

  p_ipv6->s6_addr16[5] = 0;

  p_ipv6->s6_addr[12] = 1;

  p_ipv6->s6_addr[13] = 0;

  p_ipv6 = (struct in6_addr *)&ipv6->saddr;

  if(!(p_ipv6->s6_addr32[2] == 0))

  {

                p_ipv6->s6_addr16[3] = p_ipv6->s6_addr16[6];

                p_ipv6->s6_addr16[4] = 0;

                p_ipv6->s6_addr16[5] = 0;

                p_ipv6->s6_addr[12] = 1;

                p_ipv6->s6_addr[13] = 0;

   }

   struct rt6_info *rt = (struct rt6_info*)skb->dst;

   gt_ipv6 = (struct in6_addr *)&rt->rt6i_gateway;

   …

   dst_gt[0] = gt_ipv6->s6_addr[0];

   dst_gt[1] = gt_ipv6->s6_addr[1];

   dst_gt[2] = gt_ipv6->s6_addr[6];

   dst_gt[3] = gt_ipv6->s6_addr[7];

   dst_gt[4] = gt_ipv6->s6_addr[14];

   dst_gt[5] = gt_ipv6->s6_addr[15];

   …
```

If this node is not the destination node, we transform the SSDDv6 format of the source and destination address (3fff:0:0:0:SrcNetID:SrcHostID :DstNetID:DstHostID) back to the normal address format (3fff:0:0:DstNetID:0:0:1:DstHo stID).

We obtain the next gateway address and fill in that address in the MAC header.

```
        (void)memcpy(eh->h_dest, dst_gt, sizeof(eh->h_dest));
        …
//NCTUNS_V6
…
…
}
```

In the kernel function of tun_get_user defined in tun.c, on condition that the received node is also the target node, the destination IPv6 address will be altered back to a original IPv6 address and the source IPv6 address will be altered to a SSDDv6 address for later routing purpose. However, if the received node is not the destination node, the destination IPv6 address will be modified to a SSDDv6 address for the later routing entry's inquiry. After these modifications, the packet would be delivered to the upper layer for further processing.

In the function tun_get_user() (defined in the file of tun.c):

```
/* Get packet from user space buffer */
static __inline__ ssize_t tun_get_user(struct tun_struct *tun, struct iovec *iv, size_t
count)
{
        struct tun_pi pi = { 0, __constant_htons(ETH_P_IPV6) };
        struct sk_buff *skb;
        size_t len = count;
        …
        …
        if(ip_hdr->version == 4)
```

We set the protocol of receiving side to IPv6 in default and set the protocol to IPv4 once we decide the packet we received is belong to the ipv4 protocol.

```
{
        skb->protocol = __constant_htons(ETH_P_IP);
        ip = mt_tidtoip(tid);
…
…
}
if(ip_hdr->version == 6)
  {
        struct in6_addr ipv6 = mt_tidtoipv6(tid);
        ipv6_hdr = (struct ipv6hdr *)skb->data;
        ipv6_hdr->saddr.s6_addr16[4] = ipv6.s6_addr16[3];
        ipv6_hdr->saddr.s6_addr16[5] = ipv6.s6_addr16[7];
        ipv6_hdr->saddr.s6_addr16[6] = ipv6_hdr->saddr.s6_addr16[3];
        ipv6_hdr->saddr.s6_addr16[3] = 0;
        …
        …

        if((n1 = mt_ipv6tonid(ipv6)) < 1){
                printk("nid1 fail!!\n");
                                        goto bypass1;
        }
        if((n2 = mt_ipv6tonid(ipv6_hdr->daddr)) < 1){
                goto bypass1
        }
        if(n1 != n2){
                        ipv6_hdr->daddr.s6_addr16[4] = ipv6.s6_addr16[3];
```

We obtain this node's address from node id and transform its address format to a special address format (3fff:0:0:0:SrcNetID:SrcHostID:DstNetID:DstHostID)

If this node's id is not equal to destination node id, we are in the middle node. At the middle node, we just forward the packet so we need to transform the destination IPv6 address to a special IPv6 address for routing purpose.

```
                ipv6_hdr->daddr.s6_addr16[5] = ipv6.s6_addr16[7];
                ipv6_hdr->daddr.s6_addr16[6] = ipv6_hdr->daddr.s6_addr16[3];
                ipv6_hdr->daddr.s6_addr16[3] = 0;
        }
bypass1:
        …
        …
}
```

4.3. Port Number Mapping and Translation

Before viewing the details of this topic, we have to define what the real ports as well as virtual ports are at first. The port number used by the user-level application is called the virtual port. In addition, the unique port number used by the kernel is called the real port.

If user-level programs all want to bind to the same port, due to using the same copy of Linux kernel, they cannot bind to the same port at the same time. To solve this problem, we will have to provide a special mechanism to overcome this. Therefore, the port number mapping is implemented in kernel to make users consider as if they were all bound to the same port they want on surface. In reality, the kernel transforms the virtual port to the real port. Above, we have briefly explained what the port number mapping is and we are about to see what the port number translation is in the following. When the packet is received by the destination node, the real port number has to be transformed back to the virtual port number and the procedure of performing those works is called the port number translation. To help you better understand of this mechanism, an example is listed below.

Example:

As we all know, the web-sever applications all bind to the same well-known port number 80. Based on this, when two web-sever applications are run in NCTUns, these two applications are bound to the virtual port number 80 at the same time. In kernel, each of these virtual port numbers is transformed to a unique port number called the real port, such as 5000 and 5001. When the packet is received, the kernel transforms the real port number back to the virtual port number 80.

4.3.1. Port Number Mapping

Because only one copy of kernel is used, applications cannot bind to the same port number at the same time. However, during the simulation, multiple user-level programs may be concurrently bound to the same port. For example, two web-sever applications running on the different simulated nodes all want to bind to the port number 80. Based on this, when the simulation is run, the kernel has to replace the virtual port number with the real one. Thus, user-level programs can all bind to the same port on surface.

The kernel functions related to the port number mapping are tcp_v6_get_port(), tcp_v6_hash_connect() and udp_v6_get_port(). In each of these functions, the real port number is obtained and the info of the virtual port number is recorded for later use. During the port number translation, the real port would used as a key to look up its correspondent virtual port. Here, the kernel function of udp_v6_get_port is taken as an example and shown below:

```
static int udp_v6_get_port(struct sock *sk, unsigned short snum)
{

//NCTUNS
          /*
           * 1. If the virtual port (snum) is not zero, we record it and set it to zero.
           * Then, we use original procedure to get a real port.
           * 2. Else if virtual port (snum) is zero, we choose one by ourselves.
           * And we set snum to zero. Then, we use original procedure to get a real
           * port.
           */

     if(sk->nodeID > 0){
                    int nid = sk->nodeID;
                    if(snum){
                              if(!mt_lookupVport(nid, snum)){
                                        sk->sk_vport = snum;
                                        snum = 0;
                              }else{
                                        printk("[udp_v4_get_port] already in use\n");
                                        goto fail;
                              }
                    }else{
```

We check whether the virtual port has been used or not here

We get the unused virtual port here

```
                                sk->sk_vport = mt_getunusevport(nid);

                                if(!sk->sk_vport)

                                          printk("[udp_v4_get_port] mt_getunusevport
fail!!\n");

                    }

          }
//NCTUNS

<Original Procedure>

     if (snum == 0) {

     }

          …

          …

}
```

4.3.2. Port Number Translation

The port translation is to transform the real port number back to the virtual one. When an application program wants to connect to the web server, it just knows that the server is bound to the well-known port (virtual port) 80. Therefore, it connects to the server by the port of 80. However, the port a web sever has bound to in kernel is not 80. Web server actually binds to the real port 5000, and users who wish to connect to the servers don't know the real port the server has really bound to is 5000. Those would just think servers bind to the port 80 and use port 80 as the destination they wish to connect to. Therefore, the kernel performs the port translation before the packets are transmitted to the web server. In the following, we take the udpv6_rcv()

as an example.

```
 static int udpv6_rcv(struct sk_buff **pskb, unsigned int *nhoffp)
{
        …
        …
//NCTUNS_V6
        memcpy((void *)&v6_daddr, (const void *)daddr, sizeof(struct in6_addr));
          nodeIDd = mt_ipv6tonid1(v6_daddr, skb->dev);
          if(nodeIDd > 0){
                /*
                                  * Note: XXX
                                  * If the packet belongs to virtual conncection
                                  * then we should follow the following rules:
                                  * src: R -> V (rport -> vport)
                                  * dst: V -> R (vport -> rport)
                                  */
                                rport = mt_VtoRport(nodeIDd, ntohs(uh->dest));
                                if(rport > 0)
                                                uh->dest = htons(rport);

                                vport = mt_RtoVport( ntohs(uh->source));
                                if(vport > 0)
                                                uh->source = htons(vport);

//NCTUNS_V6
```

We obtain the real port by the virtual port and replace the virtual port with the real port in the destination-port field.

We obtain the virtual port by the real port and replace the real port with the virtual port in the source-port field.

```
        …
        …
}
```

5. Modifications for Simulation

Since lots of NCTUns components are related to the internet protocol, to support the simulation of internet protocol version 6 (IPv6) in NCTUns, the modifications of simulator are a must. In section 5.1, the necessary modified modules are explained. In section 5.2, how the simulator engine is modified is explained. In section 5.3, what kinds of useful APIs are added for IPv6 is explained.

5.1 Modifications of Modules

Modules are an easy way for researchers to develop their interested topics. In NCTUns, we simulate a diversity of protocols by modules, such as MAC module, ARP module, and so forth. Therefore, researchers using the simulator of NCTUns could easily implement their interested studies by modules. For the purpose of making the modules we have developed in the platform of IPv4 function correctly in IPv6, some parts of the simulator's modules need to be altered. These necessary modified modules are listed below.

I. The HUB module (hub.cc), the MAC 802.11 module (mac-802_11-dcf.cc) and the MAC module (mac.cc):

The motivation of modifications of these modules is to make the IPv6 ptr record be generated correctly. The ptr record is the record used by GUI to draw the diagram of traffic flow. Supposing we don't modify these modules to support the network protocol of IPv6, the ptr record will be generated faultily and the GUI will draw the erroneous diagram of traffic flow. Here, we just take the hub module as an example. The modifications of the hub module are shown below.

```
int hub::sslog(ePacket_ *pkt, u_int32_t portNum)
{
   …
   …
               if ( __ip ) {
                     IPV6_SRC(ipv6Src,__ip);
                     IPV6_DST(ipv6Dst,__ip);


              }
   …
   …
   char
   ipv6_Src_[INET6_ADDRSTRLEN],ipv6_Dst_[INET6_ADDRSTRLEN];

   inet_ntop(AF_INET6,&ipv6Src,ipv6_Src_,sizeof(INET6_ADDRSTRLEN));

   inet_ntop(AF_INET6,&ipv6Dst,ipv6_Dst_,sizeof(INET6_ADDRSTRLEN));
   ss8023log->IP_Src = ipv6addr_to_nodeid(ipv6_Src_);
   ss8023log->IP_Dst = ipv6addr_to_nodeid(ipv6_Dst_);
   …
```

We get the source and destination IPv6 address from the IPv6 protocol header.

We get the node ID from the IPv6 address which is obtained from the IPv6 header.

```
    …
}
```

II. The Interface module (module/nctuns-dep/interface.cc):

The modifications we have made in this module are to set the next IPv6 gateway. The next IPv6 gateway address is obtained from the MAC header. By setting the next gateway, the simulator could know where the packet should be delivered next. The modifications of this module are shown below.

```
int interface::signal(Event_ *ep) {
    …
    …
    if(tmp_ether.ether_type != 8)
    {
                //convert the next gateway address to the form of ipv6

                bzero(gw_ipv6, sizeof(struct in6_addr));

                gw_ipv6.s6_addr[0] = tmp_ether.ether_dhost[0];
                gw_ipv6.s6_addr[1] = tmp_ether.ether_dhost[1];
                gw_ipv6.s6_addr[6] = tmp_ether.ether_dhost[2];
                gw_ipv6.s6_addr[7] = tmp_ether.ether_dhost[3];
                gw_ipv6.s6_addr[12] = 1;
                gw_ipv6.s6_addr[13] = 0;
                gw_ipv6.s6_addr[14] = tmp_ether.ether_dhost[4];
                gw_ipv6.s6_addr[15] = tmp_ether.ether_dhost[5];
```

We get the next hop IPv6 address from the Ethernet header.

```
                pkt->rt_setgwipv6(&gw_ipv6);
                pkt->rt_settype(6);
    }
    else
    {
          …
          …
    }
…
}
```

We set the next IPv6 gateway address and set the packet type to IPv6.

III. The ARP module (module/arp/arp.cc):

The ARP module does the job of looking up the MAC address for the specified IP address. To make the ARP module function correctly not only in IPv4 but IPv6 as well, some pars of this module are modified and shown below.

```
int arp::send(ePacket_ *pkt) {
      Packet *pkt_ = (Packet *)pkt->DataInfo_;
      u_short pk_type = pkt_->rt_gettype();
      if(pk_type == 6)
      {
                struct in6_addr ipv6_Dst = getDstIpv6(pkt);
                   int         recordExistButNoMac;
                   if ( u_char *macDst = findArpTbl_ipv6(ipv6_Dst,
```

We first find if the Mac record exists in the table. If the record exists, we just fill in the source and destination Mac addresses in the Mac header.

```
recordExistButNoMac))
{

            atchMacHdr(pkt, macDst, ETHERTYPE_IP);
              return(NslObject::send(pkt));

} else {
            if( ARP_MODE && !strcmp(ARP_MODE, "RunARP") ) {
  /* if record exists but has no mac, we only
    * need to update buffer space.
    */
  if ( recordExistButNoMac ) {

    updatePktBuf_ipv6(ipv6_Dst, pkt);
  }
  else {



    addArpTbl_ipv6(ipv6_Dst, 0, pkt, arpTable);


    return(arpRequest_ipv6(ipv6_Dst));
    }
  }
  else {
            freePacket(pkt);
        return (1);
```

If the record exists without Mac address, we just update the buffer space where we store the packet temporarily.

If there is no record for this packet, we just store the packet temporarily in the table and send the ARP request message to the peer node to ask the MAC address. If the procedure of looking up the MAC address is completed, we keep on delivering the packet stored in the table.

```
                    }
                }
        }
    }
```

IV. The AODV module (module/route/aodv/AODV.cc,AODVrt.cc)

Here, Ad hoc On-demand Distance Vector (AODV) routing protocol could be used for the internetworking between wireless ad hoc networks and the IPv6 internet. To make the IPv6 scheme run correctly in this module, we altered all parts of this module concerned with the address format. After modifications, we could make the packet work correctly not only in the IPv4 protocol but the IPv6 protocol as well. Because those parts we have to modify are too much, they cannot clearly be explained here. However, we could at least briefly list what we have modified in the following.

a. int updateSimpleRRoute_ipv6(struct in6_addr *prevhop_ip);

To make the immediately reverse route run correctly, this function has to be altered.

b. int sendRREQ_ipv6(struct in6_addr *dst, const u_char ttl);

To make the mechanism of transmitting the IPv6 RREQ packets to neighbors function correctly, this function has to be altered.

c. int sendRREP_ipv6(struct in6_addr *dst, struct in6_addr *src, struct in6_addr *toward, u_int8_t hopcount, u_int32_t deqno, u_int64_t lifetime);

To make the node receiving the IPv6 RREQ packets deliver the IPv6 RREP packets correctly, this function has to be altered.

d. int forwardRREQ_ipv6(struct RREQ_msg *my_rreq, u_char cur_ttl);

To make the middle node forward the IPv6 RREQ packets correctly, this function has to be altered.

e. int sendRERR_ipv6(struct in6_addr *,Unreach_list *);

To make a node send the IPv6 RERR packets to its precursors in un-reached list, this function has to be altered.

f. int processBuffered_ipv6(struct in6_addr *new_rt_addr);

To make the buffered IPv6 packets be transmitted correctly, this function has to be altered.

V. The GOD module (module/route/god-routed.cc, myrouted.cc)

God routing daemon (GOD) acts like a god; it knows all of the routing topology in advance. Therefore, it could do the job of setting all the routing tables in the start point of the simulation. We modified this module to make the function of god-routing daemon (GOD) run correctly not only in the platform of IPv4 but IPv6 as well. All the parts relative to the address formats are modified and listed below.

```
Int myRouted::send(ePacket_ *pkt)
{
   …
   if(pk_type == 6)
   {
        IPv6_DST(dstipv6, p->pkt_sget());
```

We obtain the destination IPv6 address in IPv6 header.

```
        …
    }
    …
    if(routing_ipv6(&dstipv6, p) < 0)
```

We use the destination IPv6 address to look up the routing path of nexthop.

```
    {
        freePacket(pkt);
        printf("myRoutd:: No routing entry found…");
        return 1;
    }
    …
}
```

5.2. Modifications of Simulator Engine

The simulator engine, a user-level program, provides multiple simulation services for the modules, such as event scheduler, timer management, script interpreter and so forth. In addition, it also manages all of the tools used in NCTUns, such as ttcp_v6, ping6, etc., and decides when to start the daemon or when to stop the daemon during the simulation.

To let the simulation of IPv6 function correctly in NCTUns, all parts of simulator engine influenced by the network protocol of IPv6 should be altered. The details of these modifications are listed as follows:

I. The engine function of umtbl_configtun() (maptable.cc):

Before the modification we have made, the configuration of the IPv4 tunnels is the only thing done in this function. To configure the IPv6 tunnels in NCTUns simulator, we have to add some additional codes as follows:

This format is for configuring IPv6 tunnels. There is an example listed as follows: /usr/local/nctuns/tools/ifconfig tun1 inet6 add 3fff:00:00:0001:00:00:0100:0101

```
int umtbl_configtun() {

    char                cmd1[100];
    u_char                  *p1;
    char format[] = "%s/ifconfig tun%d %d.%d.%d.%d netmask %d.%d.%d.%d";
    struct if_info          *io;
    struct maptable             *mt;

    SLIST_FOREACH(mt, &mtable, nextnode) {
        SLIST_FOREACH(io, &(mt->ifinfo), nextif) {

            p1 = (u_char *)io->netmask;
            sprintf(cmd1, format1, getenv("NCTUNS_TOOLS"),
                                          io->tid,io->ip_v6);

            system(cmd1);
        }
    }
    return(1);
}
```

We run the system call here to configure the tunnels for IPv6.

II. The engine function of set_tuninfo() (nctuns_api.cc):

To store the tunnel info of IPv6 for later use, for example, we use the node id to look up the correspondent IPv6 address, the modification we have done are listed as follows:

```
int set_tuninfo(u_int32_t nid, u_int32_t portid, u_int32_t tid,
         u_long *ip, u_long *netmask, u_char *mac,char *ipv6)
{

   return(umtbl_add(nid, portid, tid, ip, netmask, mac,ipv6));
}
int umtbl_add(u_int32_t nid, u_int32_t portid, u_int32_t tid,
            u_long *ip, u_long *netmask, u_char *mac,char *ipv6)
{
   …
   …
   /* fill if_info information */
   io->tid = tid;
   io->portid = portid;
   io->mac = mac;
   io->ip = ip;
   if(ipv6 != NULL)
   {
         io->ip_v6 = ipv6;
   }
```

The variable of ipv6 contains the info we want to store for later use.

We record the IPv6 tunnel information here.

```
…
}
```

After all of the modifications explained above are done, the simulator engine could be supported both in the network protocol of IPv4 and IPv6. However, the new APIs (one part of the simulator engine) used to support the modules are not explained here. The discussion of these newly added APIs for IPv6 is listed in the later section.

5.3. New IPv6 APIs

Some new APIs, which are used to support the network protocol of IPv6 in NCTUns, are added into the simulation engine. When we write new modules, these new added APIs would be useful tools for us to obtain the wanted info. Therefore, this section focuses on the introduction of newly added APIs. The new APIs added to NCTUns are listed and briefly explained below.

I. nodeid_to_ipv6addr (nctuns_api.cc):

To obtain the IPv6 address by this API, we use the node id as a key variable to look up the correspondent IPv6 address. Parts of the modifications are shown below.

```
…
SLIST_FOREACH(io, &(mt->ifinfo), nextif) {
/* we want to find ip of node */
if (io->portid == port)
```

```
                return((char *)(io->ip_v6));
        }
```

We return the IPv6 address stored in the entry with the equivalent node id.

II. ipv6addr_to_nodeid (nctuns_api.cc):

To obtain the correspondent node ID from by this API, we use the IPv6 address as a key variable to look up the correspondent node id. Parts of the modifications are shown below.

```
…
SLIST_FOREACH(mt, &mtable, nextnode) {
                        SLIST_FOREACH(io, &(mt->ifinfo), nextif) {
                if(!strcmp(io->ip_v6,ipv6))
                        return(mt->nodeID);
                        }
}
```

We search through all the entries in mtable and return the node id with the equivalent IPv6 address.

III. macaddr_to_ipv6addr (nctuns_api.cc):

To obtain the correspondent IPv6 address by this API, we use the Mac address as a key variable to look up the correspondent IPv6 address. Parts of the modifications are shown below.

```
…
SLIST_FOREACH(mt, &mtable, nextnode) {
                        SLIST_FOREACH(io, &(
                                        if(!bcmp(io->mac, mac, 6))
                                                        return((char *)(io->ip_v6));
                        }
```

We search through all the entries in mtable and return the IPv6 address with the equivalent MAC address.

Part II: Supporting the mixed simulation of IPv4 and IPv6

1. Introduction

Nowadays, more and more researchers devote themselves to studying the IPv6-related topics. In addition, more and more network-related merchandise supports the function of IPv6. In the future, the network of IPv6 would get increasingly common. However, transferring the currently used network (IPv4) into the next generation network (IPv6) is not an easy job. How to let the IPv4 and IPv6 coexist becomes a hot topic these days.

In this chapter, we are about to describe the works that we have done for the communication between IPv4 and IPv6 networks. When an IPv6 packet is transmitted through a network which does not support the IPv6 protocol, it will be dropped and not be able to reach its desired destination. Therefore, researchers proposed several approaches to supporting the communication between the IPv4 and IPv6 networks. Here, only two of these approaches were adopted and implemented in NCTUns. The first one is called tunneling. This approach is primarily intended to support the mixed IPv4 and IPv6 simulation networks. The second one is called NAT-PT (Network Address Translation - Protocol Translation). That approach is primarily intended adopted to support the direct transmission between IPv4 and IPv6. The detailed design of these mechanisms is explained in the following sections.

2. Design Goals

When all communication devices within the network topology run in the equivalent protocol stack, this kind of network is called the pure network. On the contrary, a network topology with a diversity of protocol stacks is called the mixed networks.

A variety of protocol stacks exists in a real-world network, and the communication between different protocol stacks is necessary. Therefore, researchers in the world have the desire of conducting the experiments not only in the environment of pure networks but mixed networks as well. Furthermore, recently there are lots of mixed-network-related studies in progress. Based on this, we write two modules, the tunneling and NAT-PT, to support the mixed networks of IPv4 and IPv6.

3. Supporting New Modules

To make the coexistence of IPv4 and IPv6 networks work correctly in NCTUns, we write two modules－tunneling and NAT-PT. In section 3.1, what is tunneling and how tunneling is implemented are explained. In section 3.2, what is NAT-PT and how NAT-PT is implemented are introduced.

3.1. Tunneling

Transforming the network of IPv4 straightly into the network of IPv6 is impossible because the IPv4 network is wide-spread currently. Therefore, the coexistence of IPv4 and IPv6 is a must. Based on this, the mechanism of tunneling is proposed to the world to solve this difficulty.

3.1.1. The Introduction of Tunneling

When the packets are transmitted through multiple networks to the desired destination, we may encounter the issue of transmitting packets through the network in different protocol stacks. In other words, the packet may be transmitted through the network of IPv4-IPv6-IPv4 or IPv6-IPv4-IPv6. Therefore, to overcome this problem, we propose one resolution called tunneling to solve this problem. Tunneling is primarily intended to aid the migration to IPv6 networks and its mechanism is explained by figure 2.1 and figure 2.2.

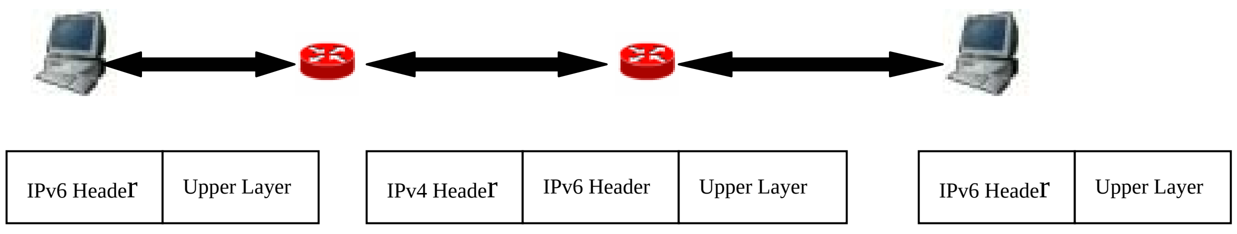


Figure 2.1: Transmission between IPv6 networks

As shown in figure 2.1, when a packet is transmitted from the IPv6 network to another IPv6 network with IPv4 network in the middle of the transmission path, an action of appending the IPv4 header would occur in the middle transmission path. On the verge of packet's entering the IPv4 networks from the IPv6 networks, an additional IPv4 header would be appended in front of the original packet. Likewise, on the point of packet's leaving the IPv4 networks to the IPv6 networks, the appended IPv4 header would be retrieved.

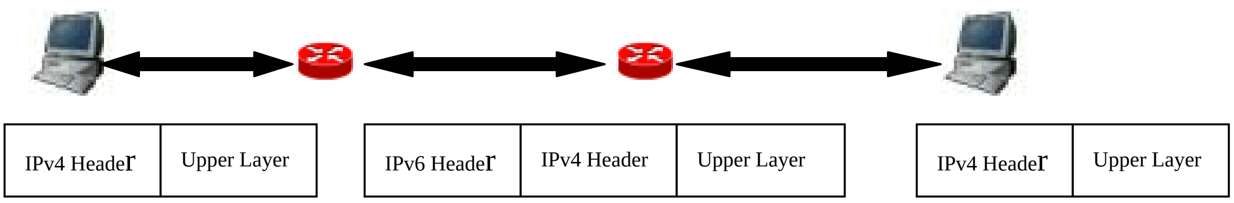


Figure 2.2: Transmission between IPv4 networks

Like the above descriptions, actions we see in figure 2.2 are almost equivalent with the actions in figure 2.1 except the type of the appended header. The appended header is the type of IPv6 instead of IPv4.

3.1.2 Modules for Supporting Tunneling

To support the mechanism of tunneling in NCTUns, a new module tunneling is developed. This module is used to support the packet's transmission of mixed networks, such as IPv6-IPv4-IPv6 or IPv4-IPv6-IPv4 networks in NCTUns. We take IPv6-IPv4-IPv6 as an example and the details of our implements are shown as follows:

/*tunnel.cc*/

```
int tunnel::recv(ePacket_ *pkt) {
    …
    …
    if(*ip_ == dst_ip)
    {
        /*this means we achieve the outer side of the tunnel
          *so we have to remove the ipv4 header from the packet
        */
        …
        char     *ipv4_hdr = (char *)pkt_->pkt_sget();
```

We get the start point of the IPv4 header.

```
        char *tmp1 = (char *)pkt_->pkt_sget();
        struct ipv6hdr    *tmp = (struct ipv6hdr*)
                                (tmp1+sizeof(struct ip));
```

We get the start point of the IPv6 header.

```
        char *ipv6_hdr = (char *)malloc(pbuf_h_n->p_tlen - sizeof(struct ip));
      int size_len = pbuf_h_n->p_tlen-sizeof(struct ip);

        memcpy((char *)ipv6_hdr,(char *)(ipv4_hdr + sizeof(struct ip)),size_len);
```

We strip off the IPv4 header here.

```
        memcpy((char *)ipv4_hdr,(char *)ipv6_hdr,size_len);
        free(ipv6_hdr);
        pbuf_h->p_len = size_len;
      pbuf_h_n->p_tlen = size_len;
```

We store the new packet size back to the total-length field in the packet buffer's header.

```
        struct ipv6hdr    *ip_header =
                            (struct ipv6hdr *)pkt_->pkt_sget();

    }
```

```
else
{
    /*this means we achieve the entrance point of the tunnel
    * so we have to add the ipv4 header in front of the ipv6 header
    */
    struct ipv6hdr    *ip_header =    (struct ipv6hdr *)pkt_->pkt_sget();
    …
    char *ip_ip_hdr = (char *)
                          malloc(sizeof(struct ip)+pbuf_h_n->p_tlen);
    int size_len = sizeof(struct ip)+pbuf_h_n->p_tlen;
    struct ip *ip_ip = (struct ip *)ip_ip_hdr;
    ip_ip->ip_tos = 0;
    ip_ip->ip_v = 4;
    ip_ip->ip_hl = 5;
    ip_ip->ip_off = htons(IP_DF);
    ip_ip->ip_ttl = 64;
    ip_ip->ip_p = htons(ETHERTYPE_IP);
    ip_ip->ip_len = htons(size_len);

    memcpy((char *)&ip_ip->ip_src,(char *)&ip_header->saddr.s6_addr32[3],
    sizeof(unsigned long));
    memcpy((char *)&ip_ip->ip_dst, (char *)&ip_header->daddr.s6_addr32[3],
     sizeof(unsigned long));
    ip_ip->ip_sum = ip_fast_csum((unsigned char *)ip_ip, ip_ip->ip_hl);
    memcpy((char *)(ip_ip_hdr+sizeof(struct ip)),(char
    *)ip_header,pbuf_h_n->p_tlen);
```

We generate an IPv4 header and append it to the original packet. Thus, we form a packet format as follows:
IPv4 header -- IPv6 header -- uplayer protocol

We obtain the IPv4 address from the last part of the IPv6 address.

We generate the IPv4 checksum here.

```
            struct ipv6hdr *tmp_hdr = (struct ipv6hdr *)(ip_ip_hdr+sizeof(struct ip));
            memcpy((char *)ip_header,(char *)ip_ip_hdr,size_len);
            free(ip_ip_hdr);
            pbuf_h->p_len = size_len;
            pbuf_h_n->p_tlen = size_len;

        }
…
}
```

We store the new length back to the packet buffer’s header.

3.2. IPv6 to IPv4 (IPv4 to IPv6) Address Translation

When one surfs the web pages, s/he may not know if the web pages s/he currently browses is running in the protocol of IPv4 or IPv6. Furthermore, the IP protocol one uses currently could be IPv4 or IPv6. Thus, one may need to communicate with another server in the different Internet protocol. Based on this, we have to adopt and develop a scheme called NAT-PT in NCTUns to achieve the direct communication.

3.2.1 The Introduction of Address Translation

Nowadays, the newly-produced network product the IPv6 function is appended in is getting more and more common. Thus, the problem of the coexistence of IPv4 and IPv6 becomes increasingly important for researchers to study. A great number of solutions are proposed by researchers and we adopt one of these solutions here—

NAT-PT is the method we use.

NAT-PT is primarily intended to aid the straight communication between v4 realm and v6 realm. This mechanism is by means of translating the IPv4 header to the IPv6 header and vice versa. Besides, some parts of ICMP header or ICMPv6 header also need to be modified. After all these works are done, the node in IPv6 protocol could communicate with one in IPv4 protocol directly and vice versa.

3.2.2 Modules for Supporting NAT-PT

For the purpose of supporting the straight communication between the IPv4 and IPv6 nodes, we add a new module, NAT-PT, to NCTUns. Two things are performed in this module.

First of all, the IPv6 header is directly transformed to the IPv4 header and vice versa. Second, parts of ICMP header or ICMPv6 header need to be modified. The header format of ICMP is almost the same with the header format of ICMPv6 except one field called the type. Therefore, we fill in the correspondent value in the type field of ICMPv4 and ICMPv6 header during the header translation. One of the directions (IPv4-to-IPv6) is taken as an example here, and the detailed implementations are listed as follows:

```
/*translator.cc*/
int translator::recv(ePacket_ *pkt) {
 …
```

```
…
if(ipv6_hdr->version == 6)
{
        /*we received IPv6 packet and want to translate it into IPv4 packet*/
        /*1.we generate the IPv4 header*/
        char *ipv4_hdr = (char *)malloc(pbuf_h_n->p_tlen –
                                        sizeof(struct ipv6hdr) + sizeof(struct ip));
        size_len = pbuf_h_n->p_tlen-
                                        sizeof(struct ipv6hdr)+sizeof(struct ip);
        struct ip *tmp_ipv4hdr = (struct ip *)ipv4_hdr;
        tmp_ipv4hdr->ip_tos = ipv6_hdr->priority;
        tmp_ipv4hdr->ip_v = 4;
        tmp_ipv4hdr->ip_hl = 5;
        tmp_ipv4hdr->ip_off = htons(IP_DF);
        tmp_ipv4hdr->ip_ttl = ipv6_hdr->hop_limit;
```

We generate an IPv4 header and fill in the correspondent values here!!

```
        if(ipv6_hdr->nexthdr == NEXTHDR_ICMP)
        {
                        tmp_ipv4hdr->ip_p = IPPROTO_ICMP;
        }
        else
                tmp_ipv4hdr->ip_p = ipv6_hdr->nexthdr;
```

Do remember to fill in the correct protocol field!!

```
          tmp_ipv4hdr->ip_len = htons(size_len);
```

We fill in the new total length in the field of IPv4 header.

```
        /*generate IPv4 address from IPv6 address*/
          memcpy((char *)&tmp_ipv4hdr->ip_src,
      (char *)&ipv6_hdr->saddr.s6_addr32[3],sizeof(unsigned long));
       memcpy((char *)&tmp_ipv4hdr->ip_dst,
      (char *)&ipv6_hdr->daddr.s6_addr32[3],sizeof(unsigned long));


       tmp_ipv4hdr->ip_sum = ip_fast_csum((unsigned char
   *)tmp_ipv4hdr,           tmp_ipv4hdr->ip_hl);


      /*2.change the type of icmpv6 header to icmp header*/
      if(ipv6_hdr->nexthdr == NEXTHDR_ICMP)
      {
          struct icmp6hdr     *icmpv6_hdr =
                          (struct icmp6hdr *)(pkt_->pkt_sget()+
                          sizeof(struct ipv6hdr));
          if(icmpv6_hdr->icmp6_type == ICMPV6_ECHO_REQUEST)
                  icmpv6_hdr->icmp6_type = ICMP_ECHO;
          if(icmpv6_hdr->icmp6_type == ICMPV6_ECHO_REPLY)
                  icmpv6_hdr->icmp6_type = ICMP_ECHOREPLY;


      }
      /*3.recompute the TCP or DUP checksum*/
      if(ipv6_hdr->nexthdr == IPPROTO_UDP)
      {
          struct udphdr *uh =
                      (struct udphdr *)(pkt_->pkt_sget()+
                      sizeof(struct ipv6hdr));
          if(uh->uh_sum != 0)
```

We compute the new check sum and fill in this value in the checksum filed of IPv4 header.

We replace the ICMPv6 type with the ICMP type.

```
                {
                        uh->uh_sum =
                        csum_tcpudp_magic(tmp_ipv4h                              >ip_dst,s
                        ize_len,IPPROTO_UDP,0);
                }
        }
        else if(ipv6_hdr->nexthdr == IPPROTO_TCP)
        {
                struct tcphdr   *th =
                                (struct tcphdr *)(pkt_->pkt_sget()+
                                sizeof(struct ipv6hdr));
                th->th_sum = csum_tcpudp_magic(tmp_ipv4hdr->ip_src,
                tmp_ipv4hdr->ip_dst,size_len,IPPROTO_TCP,0);
        }
        /*4.we copy the new ipv4 header back to PT_SDATA and the translation is
        *completed
        */
        char *tmp1 = (char *)(pkt_->pkt_sget()+sizeof(struct ipv6hdr));
        char *tmp2 = (char *)(pkt_->pkt_sget());
        char *tmp3 = (char *)(ipv4_hdr + sizeof(struct ip));
        memcpy((char *)tmp3,(char *)tmp1,pbuf_h->p_len-
                                                sizeof(struct ipv6hdr));
        memcpy((char *)tmp2,(char *)ipv4_hdr,size_len);
    ...
}
```

We recomputed the checksum here!!

We replace the IPv6 header with the IPv4 header generated earlier.

Part III: Supporting IPv6 emulation

1. The Emulation Host

The emulation host is a mechanism for users to simulate the transmission between the simulation machine and real host. When one puts the external host (the emulation host) in the simulated network topology, the packets transmitted to the node of external host are actually delivered to the real host. Therefore, researchers could gain the experimental results not only in the simulated environment but the real world as well. This chapter focuses on how this mechanism works correctly.

1.1. The Introduction of IPv6 Emulation Host

To direct the packets from the simulated host to the other genuine host through the real physical link and vice versa, a user-level program, an emulation daemon, captures the packets from the kernel and directs them to the correct direction. The works of this daemon are capturing packets according to the filtering rules and making the decision of putting the packets ether into the simulation or Ethernet links. How the packets are transmitted between the simulated host and the real host is shown in the diagram of figure 3.1. In the figure 3.1, when a traffic-generating program (ttcp) running on the real host delivers a UDP packet to the simulated host (3fff:00:00:0001:00:00:0100:0101), the packet is transmitted into the real world links from the Ethernet interface. After the packet is received by the real host, it would be routed back to the simulated host according to the routing entries set in real host. The emulation daemon running on the simulated host captures the packet coming from the Ethernet interface and makes some modifications of the internet protocol address for routing purpose. Then, we transmit the packet back to the tunnel interface; the simulator reads it from the tunnel interface and puts it into a series of simulations. When the simulated host node 1 receives the packet and discovers the packet is the desired destination, it delivers this packet up to the user level for further processing.

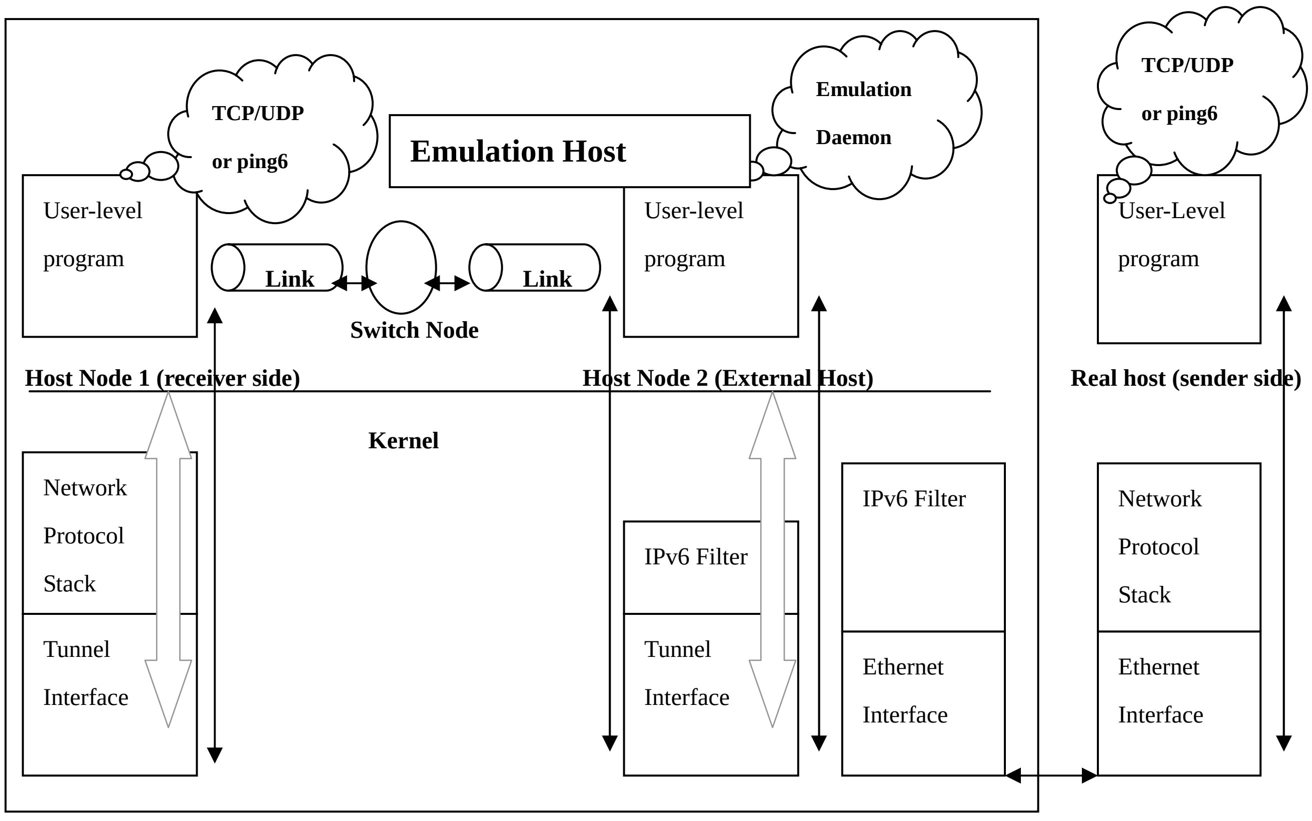


Figure 3.1: The topology of IPv6 emulation host

1.2 Mechanisms for the IPv6 Emulation Host Daemon

In the figure 3.1, there is an emulation daemon running on the simulated host node 2. The emulation daemon does the job of translating the simulated IPv6 address to the real host IPv6 address and vice versa. When the emulation daemon receives the packet, it will first check the direction (INPUT or OUTPUT) of this packet. If the direction of this packet is INPUT, the destination address of this packet will be stored in the source address field of the table for looking up the desired entry later, and the source address of this packet will need to be stored in the destination address field of the table as well. Besides, if the protocol of this packet belongs to TCP or UDP, the source and destination port numbers will need to be stored in the table. There is however one notable place—the source-port number of this packet is stored in the field of destination port and vice versa.

After the direction is determined and the necessary info is recorded, the simulated IPv6 address of this host node is stored in the source address field of the IPv6 protocol header. In addition, the destination address in the IPv6 header is transformed into the SSDDv6 address format (3fff::SrcNet:SrcHost:DstNet:DstHost). Why an original destination IPv6 address is transformed to a SSDDv6 address is that the kernel can make use of a SSDDv6 address to look up the correspondent tunnel interface and route the packet to that tunnel interface.

Previous descriptions merely explain what actions the emulation host daemon will perform in the direction of INPUT. In the following paragraph, we are about to see what actions need to be done in the direction of OUTPUT. When the packet is received by the daemon and the OUTPUT direction is determined, the source IPv6 address obtained from the table is stored in the source address field of the IPv6 header. The real IPv6 address of the foreign host is stored in the destination address field of the IPv6 header. Furthermore, the source port number obtained from the table is recorded in the IPv6 header.

Above, how the header is modified during the transmission of packets is briefly discussed. We will see the details pertaining how to transform the address and port fields within the protocol header in the following sections.

1.2.1 Translating the IPv6 Address

By one example below, this mechanism could be clearly understood.

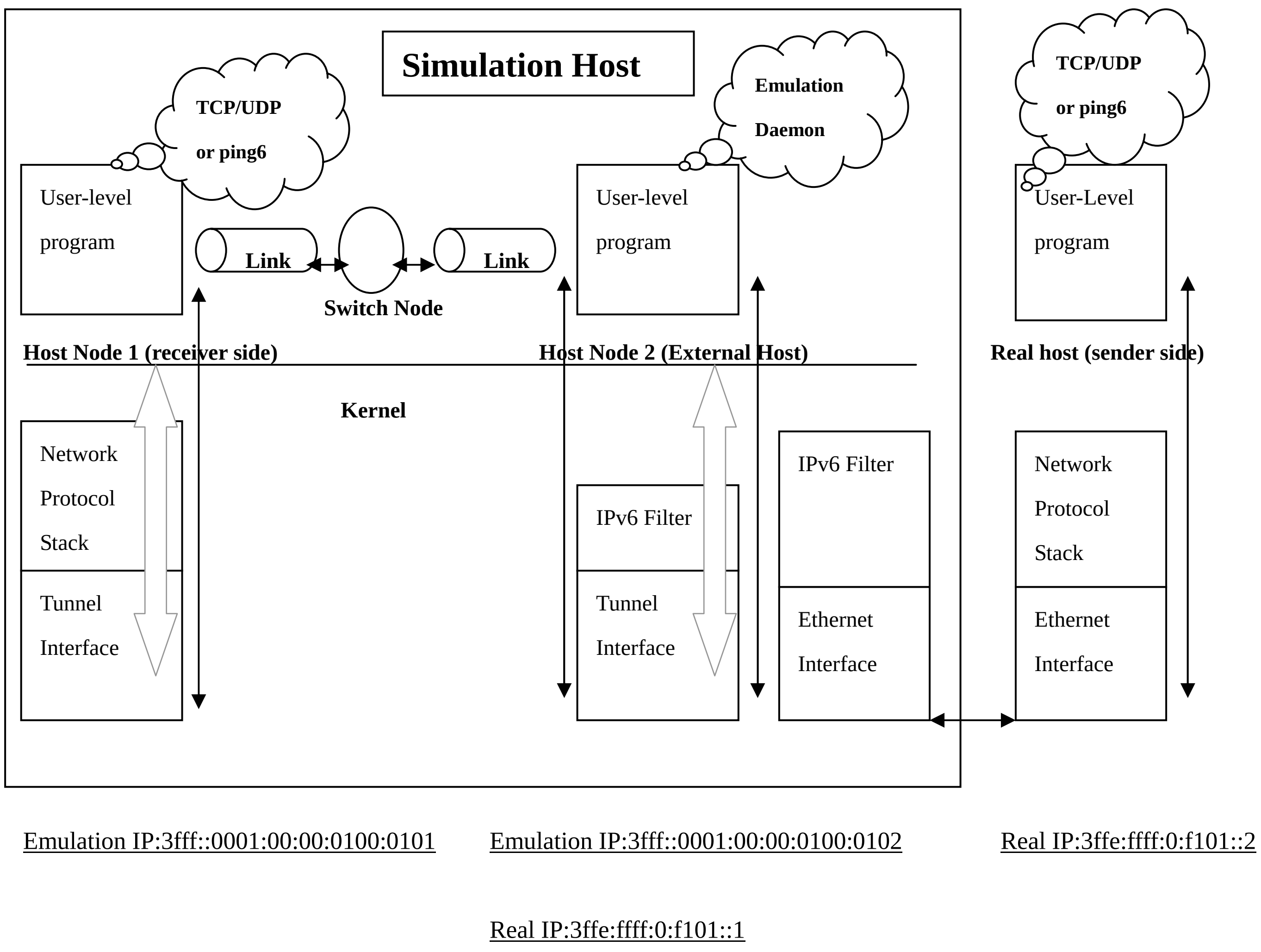


Figure 3.2: Translating the IPv6 address for running emulation

In the figure 3.2, when the sender wants to transmit the packet to the simulated host node 1, it first sends the packet with the source address of 3ffe:ffff:0:f101::2 and destination address of 3fff:00:00:0001:00:00:0100:0101 to the host with the real IPv6 address of 3ffe:ffff:0:f101::1. After the host with the IPv6 address of 3ffe:ffff:0:f101::1 receives the packet, the packet is captured by the emulation daemon. Then, the daemon records the source IPv6 address, the destination IPv6 address and port numbers and replaces the source IPv6 address of 3ffe:ffff:0:f101::2 with the simulated source address of 3fff::0001:00:00:0100:0102. The destination IPv6 address of 3fff:00:00:0001:00:00:0100:0101 is also replaced with the simulated SSDDv6 destination IPv6 address of 3fff::0001:0102:0001:0101. After those necessary modified fields of IPv6 protocol header are changed, this packet is able to be sent straightly by the system call of sendto().

After the packet is transmitted to NCTUns, it is put into a series of simulation. If this packet belongs to the type of ICMP or TCP, the emulation host node 1 must send the reply packet back to the emulation host node 2. When the node 2 receives the reply packet, this reply packet is captured by emulation daemon. The emulation daemon replaces the source IPv6 address with the one stored in the table earlier (3fff:00:00:0001:00:00:0100:0101) and replaces the destination IPv6 address with the IPv6 address of the real host (3ffe:ffff:0:f101::2). After altering these fields, the daemon sends the packet out with the system call sendto() straightly.

1.2.2 Translating the Port Number

When the real host with the source-port number 4500 sends the packet to the simulated host binding to the port number 7000, it considers that it would create the connection with the peer node by the port 7000. However, due to the special port mapping mechanism implemented in the NCTUns, the genuine port we create connection with is replaced by the real port 5000. Thus, when the packet is captured by the emulation daemon running on the simulated host node 2, the destination port 7000 is stored in the source port field of the table and the source port 4500 is stored in the destination port field of the table. After the info is stored, the packet is sent directly to the peer simulated host node 1.

When the host node 1 receives the packet, it sends the reply packet with the source port 5000 and destination port 4500 to the host node 2. By the time of receiving the captured packet in host node 2, the daemon running on the host node 2 compares the IPv6 source address in IPv6 header with the source IPv6 address stored in the table; it also compares the destination port with the one stored in the table. Supposing all the comparisons are equal, we will store the source port and destination port obtained from the table to the source port and destination port fields of the IPv6 protocol header. Therefore, we could create connection of the port 7000 instead port 5000.

1.3 Setting the Routing Entries for the Emulation Host

Based on the previous sections, we could clearly understand how the packet is transmitted and altered. Nevertheless, there is one notable point. Above, we don't mention how the packet is routed from the real host to the external host and vice versa. This section focuses on how the packet is routed according to our built-up routing entries. The detailed description is stated below.

I. Setting the IPv6 address:

```
ip -6 addr add [IPv6-address] dev [device name]
```

Before the simulated host and real host are running, we execute this kind of command to set real IPv6 addresses to the simulated host and the real external hosts. When the routing entries are set, these assigned real IPv6 address will be used.

Example:

```
ip -6 addr add 3ffe:ffff:0:f101::1 dev eth0
ip -6 addr add 3ffe:ffff:0:f101::2 dev eth0
```

Above, we execute those commands to set the real IPv6 address of the simulated host to 3ffe:ffff:0:f101::1 and set the real IPv6 address of real host to 3ffe:ffff:0:f101::2.

II. Setting the routing:

```
route –A inet6 add [IPv6-address prefix] gw [IPv6-address]
```

Before the simulated host and real host are running, we execute this kind of command for directing the packet from the simulated host to the real host and vice versa. The IPv6 address in the position after "gw" is the gateway address we set in part I.

Example:

```
route –A inet6 add 3ffe::/16 gw 3ffe:ffff:0:f101::2 ………………..(1)
route –A inet6 add 3ffe::/16 gw 3ffe:ffff:0:f101::1 ………………..(2)
```

The command 1 is executed in the simulated host. After executing the command 1, the address prefix with 3ffe can be routed to the gateway address of 3ffe:ffff:0:f101::2 (the real IP address of a real host). That is to say, the packet can be directed from the simulated to the real host. The command 2 is executed in the real host and its function works like command 1. After the execution of command 2, the packet can be directed from the real host to the simulated host.

1.4 Mixed IPv6 and IPv4 Emulation Host

Due to the exponentially increasing internet users in the world, the network protocol of IPv4 is getting less and less sufficient to satisfy those users. Therefore, the next generation network protocol, IPv6, is introduced to the world. However, one problem accompanies the newly introduced network protocol (IPv6). How to transform the original network of IPv4 into the network of IPv6 is a hard job. People around the world start to study and propose lots of transferring methodologies to solve this problem. Therefore, the mixed network is still a valuable topic for researchers. We have introduced how we solve this problem in earlier chapters. Nevertheless, those solutions are all for the simulation. Here, we propose a solution to provide the mixed network in the emulation.

One module (NAT-PT) is added between the sender and receiver for the mixed network emulation. By the way, the network protocol of the sender and the receiver must be different. On condition that the sender and receiver are all in the same network protocol, the NAT-PT shall be unnecessary to be added between them. Otherwise, the NAT-PT is a must. When the packet is transmitted from the sender to the receiver or from the receiver to the sender, its packet header would be completely transformed into another protocol header. Therefore, people could run the different network-protocol applications on each side. We take the situation of the receiver with IPv4 application and sender with IPv6 application as an example below.

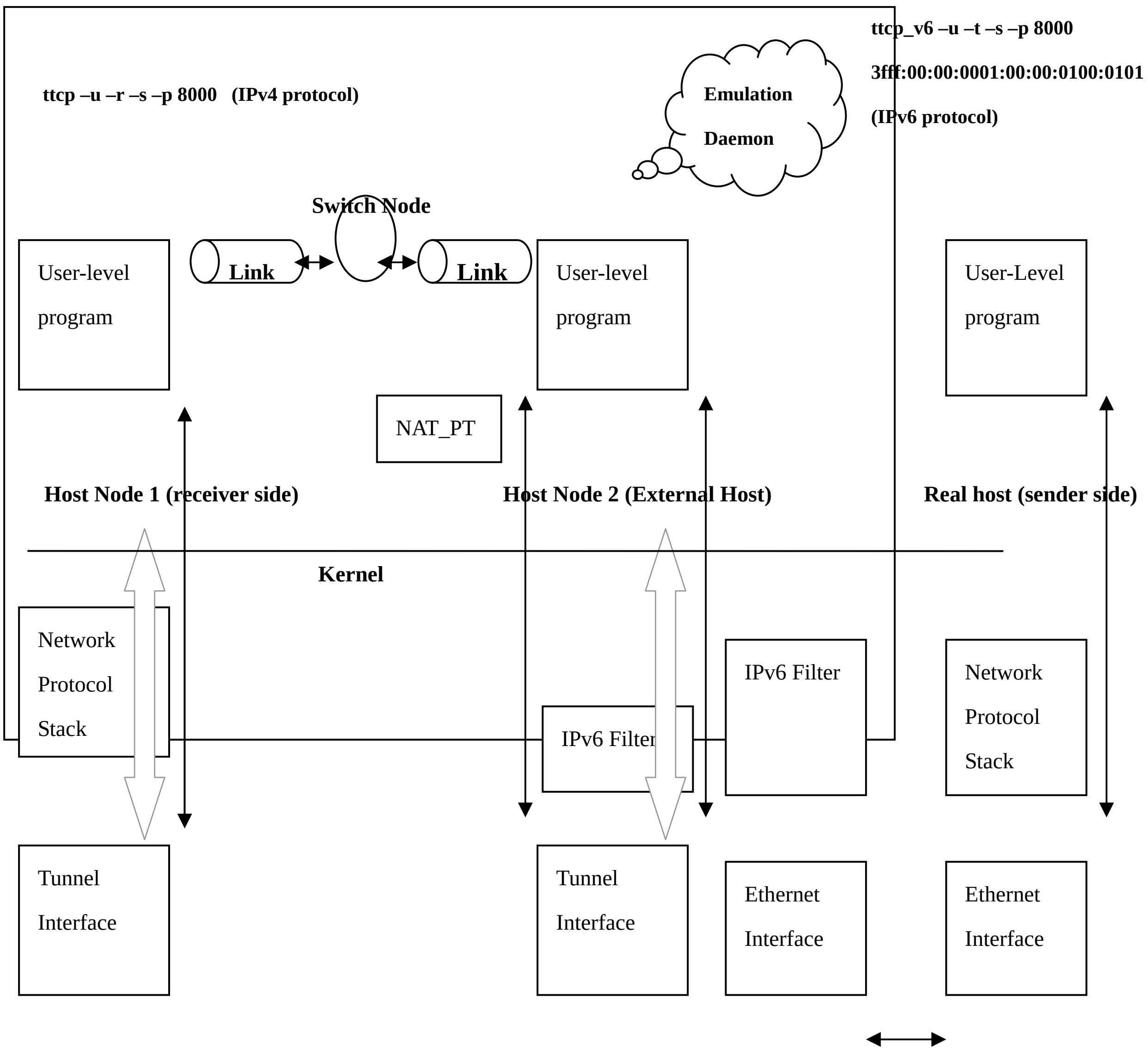


Figure 3.3: Mixed IPv6 and IPv4 Emulation Host

From the figure 3.3 above, we could see that when the packet is transmitted from the emulation daemon to the kernel, it would go through the simulator's transmission path. After the packet is sent from the specific tunnel interface to the simulation engine, it would be transmitted through a series of modules and one of these modules is NAT-PT. NAT-PT does the work of completely translating the IPv6 header to IPv4 header and vice versa. By the way, the ICMPv6 header also needs to be modified into the ICMP header by this module and vice versa. After the translation, the machine running IPv6 applications could communicates with the host running IPv4 applications straightly.

2 The Emulation Router

The topology of the network is composed of lots of elements, such as the host, the hub, the router, the switch and so on. Therefore, people conducting the experiments of emulation will interest in not only the host but other components as well. Based on this, we provide the other emulation methodology－the emulation router. By using the emulation router within the simulated network topology, we could gain the experimental results under the environment of mixed components－simulated hosts, simulated routers and real routers.

2.1. The Introduction of IPv6 Emulation Router

The emulation router is a real router and we put it between the simulation networks to make the packets be sent through the real router during the transmission.

From the figure 3.4 below, we could see that when the simulation node 1 sends the packet to the simulation node 2 through the external router, the packet would be captured by the emulation router daemon. After the packet is captured, the daemon would alter the address into a specially designed address and deliver it out directly. A packet with the special address format would be directed to the real router. On the point of receiving the packet sent from the simulation router, the real router would look up the routing table and forward the packet back to the simulation router according to the routing rules we have built up. When the simulation router receives the packet, the packet would be captured by emulation router daemon again. At this time, the emulation daemon would alter the special address format back to the normal one and deliver it out to keep on the simulation.

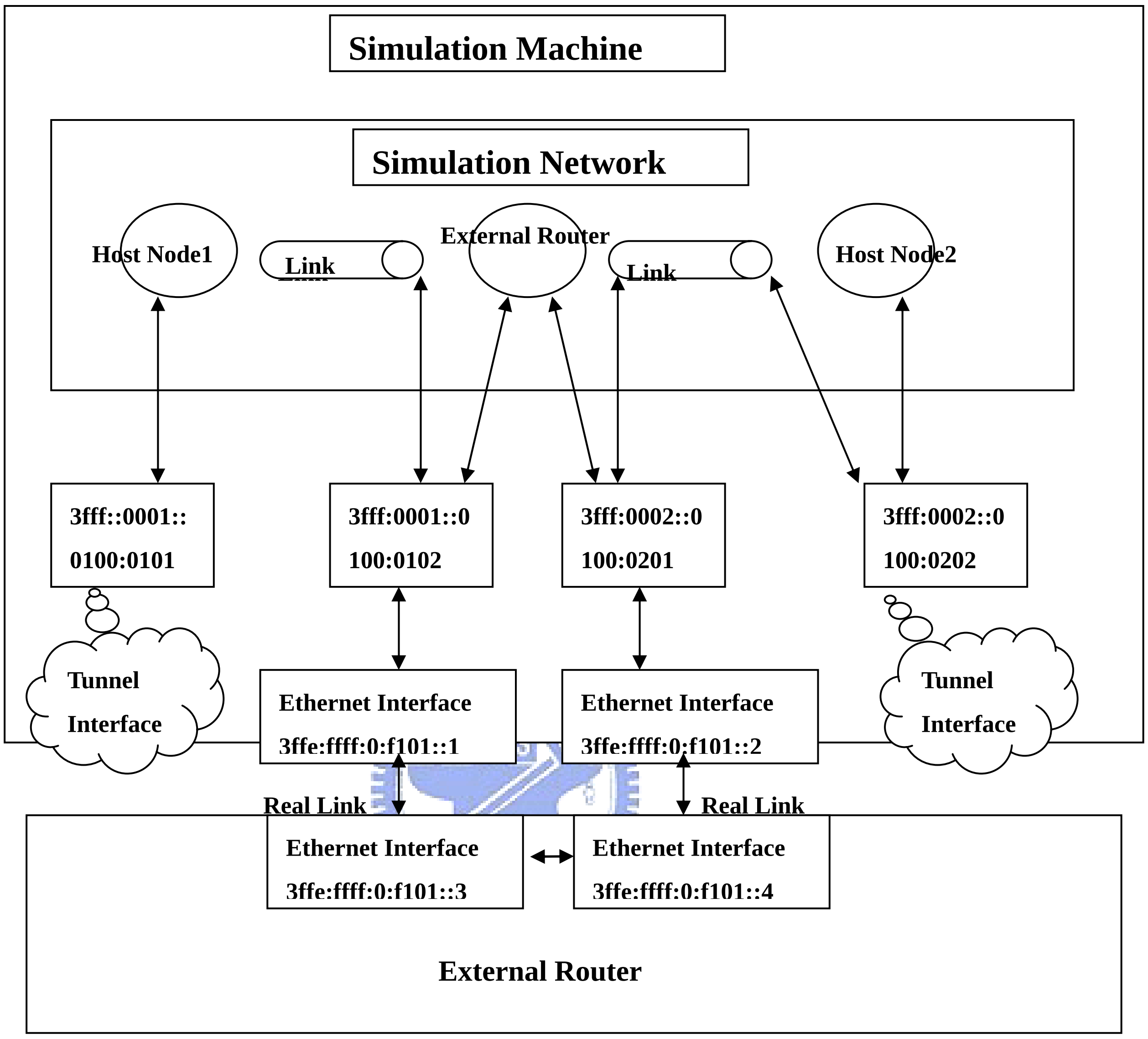


Figure 3.4: The topology of IPv6 emulation router

2.2. Mechanisms for the IPv6 Emulation Router Daemon

In the figure 3.4, when the packet is received by the external router, it will first be captured by emulation router daemon. After the packet is captured, the daemon would replace the destination IPv6 address (3fff::0002:00:00:0100:0202) with a special address (3fff:00:00:0002:00:00:c801:0202) we contrive. Unlike the emulation host daemon, the emulation router daemon doesn't record the info of port because works the router does are just forwarding the packet. Furthermore, the packet is not transmitted up to the transmission protocol layer, such as TCP or UDP. Thus, after the address is altered to the special one, the packet is sent straightly. According to the routing entries set in the simulation machine, the packet will be sent to the real router with the IPv6 address of 3ffe:ffff:0:f101::3. Likewise, the packet will be sent back to the emulation machine (3ffe:ffff:0:f101::2) based on the routing entries set in the real router. When the packet is received by the simulated router, the packet would be captured by the router daemon. The router daemon translates the special destination IPv6 address (3fff:00:00:0002:00:00:c801:0202) back to the SSDDv6 destination (3fff::0002:00:00:0100:0202) IPv6 address and transmits it directly to experience the simulation.

2.3. Translating the IPv6 Address

To make the routing entries which we set for directing the packet to the emulation router not be confused with those we set for the execution of simulation in NCTUns, the destination IPv6 address needs to be altered. On the point of the packet's leaving the simulated router, the modified destination IPv6 address is used to route the packet to the real router. In addition, on the point of the packet's leaving the real router, the modified destination address is also used to route the packet to the simulated router. How the normal address format is modified to the special one is explained as follows:

The original IPv6 address is:

3fff:00:00:DstNetID:00:00:0100:DstHostID

Example:

3fff:00:00:0002:00:00:0100:0202

The modified IPv6 address is:

3fff:00:00:DstNetID:00:00:c8/SrcNetID:DstNetID/DstHostID

Example:

3fff:00:00:0002:00:00:c801:0202

As the modified IPv6 address listed above, we can see that the address is translated according to a special scheme. The special scheme is that we put the source subnet ID in the place after the special number c8 and put destination subnet ID and destination host ID in the last field of the IPv6 address format. After the above modifications, the normal IPv6 address format (3fff:00:00:0002:00:00:0100:0202) would be altered to the special one (3fff:00:00:0002:00:00:c801:0202). Why we put the special number c8 before the source subnet ID is that we could differentiate the routing entries set for directing the packet to the real host from those set for directing the packet to other simulated hosts in NCTUns.

2.4. Setting the Routing Entries for the Emulation Router

Based on the Figure 3.4 and the previous sections, we could clearly understand how the packet is transmitted and altered. However, there is one notable point. Above, we don't mention how the packet is routed from the real router to the emulation router and vice versa. In this section, the routing settings will be introduced by an example in the following. When the packet is transmitted from the normal simulated host node to the external router node, it will first be directed from the simulated machine to the real router by the commands as follows:

```
route –A inet6 add 3fff:00:00:0002:00:00:c801:0200/120 gw 3ffe:ffff:0:f101::3
route –A inet6 add 3fff:00:00:0001:00:00:c802:0100/120 gw 3ffe:ffff:0:f101::4
```

After the routing entries are built up by executing above commands, the kernel can direct the packet, which is from the subnet 1 to the subnet 2, to the real router with the IPv6 address 3ffe:ffff:0:f101::3 set to one of its interface. Furthermore, the kernel can direct the packet, which is from the subnet 2 to the subnet 1, to the real router with the IPv6 address 3ffe:ffff:0:f101::4 set to one of its another interface; however, the commands listed above are just to route the packet to the real router. Because the packet directed to the real router also needs to be directed back to the simulation machine, we list the commands used in this situation as follows:

```
route –A inet6 add 3fff:00:00:0002:00:00:c801:0200/120 gw 3ffe:ffff:0:f101::2
route –A inet6 add 3fff:00:00:0001:00:00:c802:0100/120 gw 3ffe:ffff:0:f101::1
```

Like the above descriptions, the similar actions would be taken in the real router. Once the packet is received, the real router directs the packet, which is from the subnet 1 to subnet 2, to the simulation machine with the IPv6 address 3ffe:ffff:0:f101::2 set to one of its interface. Likewise, the real router directs the packet, which is from the subnet 2 to subnet 1, to the simulation machine with an IPv6 address 3ffe:ffff:0:f101::1 set to one of its another interface. After all the above commands are executed in a real router and a simulated router, the packets can be routed between these hosts.

3 Kernel Modification

The necessary modified parts in the kernel for making the IPv6 emulation function correctly are system calls, the netfilter and the divert-socket-emulated mechanism. In FreeBSD, we have the mechanism of divert socket and this mechanism could be used to filter the specified packet. However, In Linux, this mechanism no longer exists but we could emulate this method by the mechanism of netfilter. What is netfilter and how the kernel is modified are explained in the following sections.

3.1. The IPv6 Netfilter

Before the modifications of Linux kernel for simulating the divert socket are introduced, the mechanism of IPv6 netfilter needs to be explained at first. Unlike Freebsd, Linux doesn't support divert socket; nevertheless, it supplies other mechanism called netfilter to emulate the divert socket function. Netfilter is formed by five hooks and each hook is registered with a callback function. In the figure 3.5, when a packet is transmitted through one of the five hooks, such as NF_IP6_FORWARD, this packet would be delivered to the registered call back function (divert_hook). In the call back function, the decision of whether the packet should be captured or not is made. If this packet needs to be captured, it will be put into the socket queue for user-level programs to read it out of the queue. Otherwise, the kernel returns to the original data-flow path and keeps on the transmission of packets. Above descriptions are just the explanations for this mechanism. The real works we have done in the kernel are shown in the following sections.

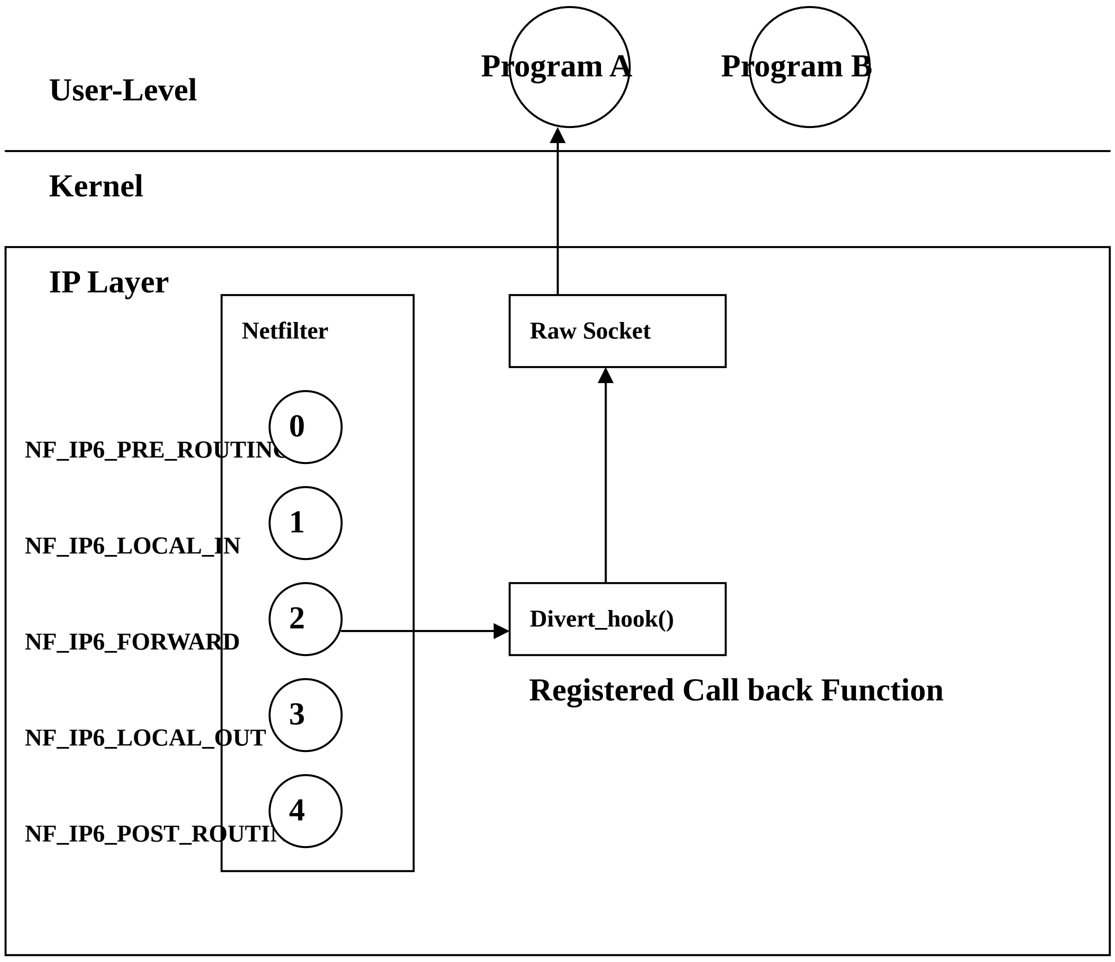


Figure 3.5: The IPv6 netfilter

3.2. Implementation for IPv6 Netfilter

There are five hooks for the IPv6 netilter. Each hook is registered with a callback function. When the packet is transmitted through any of these hooks, it would be transmitted to the callback function. The callback function performs the works of making the decision of whether the packet should be captured or not. Before the mechanism of IPv6 Netfilter functions correctly, the callback function needs to be registered for each hook. How to register the callback function in the kernel is shown below.

To register the callback function (defined in tun.c):

The callback function is registered for each IPv6 hook during the initialization of the tunnel interface. After the callback function is registered, the packet transmitted through each of these IPv6 hooks will be delivered to the registered callback function.

```
…
static struct nf_hook_ops ipv6_redir_ops0= {
            .hook    =divert_hook,
            .owner =THIS_MODULE,
            .pf       =PF_INET6,
            .hooknum =NF_IP6_PRE_ROUTING,
            .priority =0,
  };
…
err = nf_register_hook(&ipv6_redir_ops0);
err = nf_register_hook(&ipv6_redir_ops1);
err = nf_register_hook(&ipv6_redir_ops2);
err = nf_register_hook(&ipv6_redir_ops3);
err = nf_register_hook(&ipv6_redir_ops4);
…
```

This is the callback function.

IPv6 hooks are registered here!!

3.3. Implementation for Divert Sockets

Each time when the packet is transmitted through any point of divert hooks, this packet is delivered to the callback function (divert_hook) registered in the phase of tunnel initialization. In the divert_hook function, first, whether this packet should be retrieved or not is determined. Second, if the packet needs to be captured, it will be appended to the socket queue for further processing by application. The real modifications we have done to implement the divert sockets are shown below.

The callback function (defined in nctuns_divert.c):

```
unsigned int divert_hook(unsigned int hook,
                         struct sk_buff **pskb,
                    const struct net_device *indev,
                    const struct net_device *outdev,
                    int (*okfn)(struct sk_buff *))
{
     …
     …
     if (curr = nctuns_pkt_match(hook, *pskb)) {
//NCTUNS_V6
          if(ip->version == 6)
          {
               dr_ipv6 = list_entry(curr, struct divert_rule_ipv6, nextdr);

                         if (!dr_ipv6->sk) {
```

Whether this packet should be captured or not is determined here.

```
                                        printk("divert_hook() : Fatal error! Can not find
the sock!!\n");
                            }


          }
          else
          {
          …
          …
          }
     if(ip->version == 6)
          {
               if (sock_queue_rcv_skb(dr_ipv6->sk, *pskb) < 0) {
                                   kfree_skb(*pskb);
                                        printk("divert_hook() : now %lu,
                                   Enqueue fail!\n", NCTUNS_ticks);
                              }


          }
     …
     …
}
```

After the packet is determined to be captured, it is appended into the socket queue for further processing by the user-level program.

3.4. Modifications in the External Node

To make the ICMPv6 packet be transmitted correctly to the real host and back to the simulated host, we have to alter a little of the kernel codes in the real host. In case the kernel codes of the real host are not modified, the checksum error will occur and the packet will be thrown away. Based on this, we modified some parts of kernel codes and these modifications are listed as follows.

In the function of rawv6_recvmsg (defined in net/ipv6/raw.c) and tcp_v6_rcv (defined in net/ipv6/tcp_ipv6.c):

The field of checksum is set to the value of CHECKSUM_UNNECESSARY. After these codes are added, the checksum error can be avoided, and the packet will be received by the real host without any problems.

```
…
if(skb->nh.ipv6h->daddr.s6_addr16[0] != htons(0x3fff) &&
skb->nh.ipv6h->saddr.s6_addr16[0] == htons(0x3fff))
{
    skb->ip_summed = CHECKSUM_UNNECESSARY;
}
…
```

If the packet is from the simulated host to the real host, its checksum should be set to unnecessary. Thus, the packet would not be dropped during transmission.

3.5. System Calls

A system call 280 is added in the Linux kernel for registering the divert-socket information of IPv6. The data passed by system 280 is the filtering rules, the specified action, and the specified hook. When the packet is transmitted through any one of the five hooks and delivered into the callback function, the packet would be filtered according to the rules registered by system call 280. How the system call is called and implemented is shown below.

I. The IPv6 emulation daemon (emudiv.c):

The IPv6 emulation daemon is a program performing the job of translating the simulated IPv6 address to the real IPv6 address and vice versa. In the beginning of the emulation daemon, some system calls are called to register or flush the information of the packet-filtering rules in the specified hook. The details are explained as follows:

```
int main(argc,argv)
int argc;
char *argv[];
{
…
…
        ipv6_addr_set(&de_ipv6.srcip,0,0,0,0);
        ipv6_addr_set(&de_ipv6.smask,0,0,0,0);
        memcpy((void *)&de_ipv6.dstip, (const void *)&myipv6, sizeof(struct
        in6_addr));
        ipv6_addr_set(&de_ipv6.dmask,0xffffffff, 0xffffffff,0xffffffff,0xffffffff);
        syscall(280, DIVERT_ADDTAIL, divfs, NF_IP6_LOCAL_IN, (char
        *)&de_ipv6, len_ipv6);
```

The daemon fills in some data, such as packet-filtering info, the wanted action, and the specified hook. Then, it passes them to the kernel by system call 280.

```
…
…
}
```

After the info is sent from the user-level program to the kernel, the kernel will perform some correspondent works according to the system call 280. How the kernel implements the system call 280 is introduced in the part II below.

II. The system calls (nctuns_syscall.c):

The kernel defines a series of system calls, and one of the system calls, system call 280, is used in the IPv6 emulation. After the info is passed from the emulation daemon to the kernel by the system call 280, the kernel will store or flush the packet-filtering info. How this system call is implemented in the kernel is explained in the following.

Implementations of system call 280:

When the system call 280 is called by the user-level program, the data will be passed from the application to the kernel function of sys_NCTUNS_divert_ipv6. Then, in sys_NCTUNS_divert_ipv6(), the other kernel function — nctuns_reg_divert_rule_ipv6()— will be called.

In the kernel function (nctuns_reg_divert_rule_ipv6()), a series of actions are implemented, such as DIVERT_ADD, DIVERT_FLUSH, etc. The kernel takes the correspondent action according to the choice passed from the user-level program by system call 280.

Example:

On condition that the action of DIVERT_ADD is chosen, the packet-filtering info passed by the user-level program will be stored in the correspondent hook. Then, when a packet is transmitted through that hook, it will be filtered according to the info stored in the hook. The detailed kernel codes are listed as follows:

```
asmlinkage int sys_NCTUNS_divert_ipv6(int action, int fd, u_long hook, struct
divert_entry_ipv6 *de, int addr_len)
```

This is the system call 280.

```
{
                        …
          ret = nctuns_reg_divert_rule_ipv6(action, sock ? sock->sk : NULL, hook,
de,addr_len);
```

This kernel function is called for further processing of the info passed from the user-level program.

```
          return ret;
}
int nctuns_reg_divert_rule_ipv6(int action, struct sock *sk, u_long hook, struct
divert_entry_ipv6 *de, int addr_len)
{
         …
         …
         switch (action) {
```

The correspondent works are performed according to the action passed by the user-level program.

```
          case DIVERT_INFO:
         …
          case DIVERT_ADDHEAD:
```

```
        case DIVERT_ADDTAIL:
                …
                  memcpy((void *)&dr->srcip, (const void *)&myde->srcip,
                  sizeof(struct in6_addr));
                  memcpy((void *)&dr->smask, (const void *)&myde->smask,
                  sizeof(struct in6_addr));
                  memcpy((void *)&dr->dstip, (const void *)&myde->dstip,
                  sizeof(struct in6_addr));
                  memcpy((void *)&dr->dmask, (const void *)&myde->dmask,
                  sizeof(struct in6_addr));

                  dr->sport          = myde->sport;
                  dr->dport          = myde->dport;
                ...
                  if (action == DIVERT_ADDHEAD)
                    list_add(&dr->nextdr, &rule_table[hook]);
                  else if (action == DIVERT_ADDTAIL)
                        list_add_tail(&dr->nextdr, &rule_table[hook]);
                  break;
              …
}
```

The packet-filtering info is stored in the hook which the user-level program chooses.

Future Work

Although the simulation and emulation environment of IPv6 has been built up, there are still some valuable topics needed to be implemented in NCTUns. These uncompleted works, such as mobile IPv6, RIPv6, OSPFv6, and so forth, are listed in the table below for further development by others who are interested in these topics.

The Mobile IPv6	Not done
The QOS of IPv6	Not done
RIPv6	Not done
OSPFv6	Not done
Optical Networks of IPv6	Not done
VoIPv6	Not done
Command console	Not done

Conclusions

Due to the exponentially increasing internet users and newly designed devices, such as 3G mobile phones, embedded vehicular network devices, and so on, the next generation network protocol (IPv6) is becoming increasingly important in the future. Therefore, we provide an environment for users to conduct their interested IPv6-related topics at low costs in NCTUns. In the previous sections and chapters, we explain what we did to support the IPv6 simulation and emulation in NCTUns. That is to say, the environment of the simulation and emulation of IPv6 has already been built up. Therefore, researchers can now conduct any IPv6-related study in the environment that we have developed. To make the readers more clearly understand how much work we have done, completed works are listed in the table below.

Phase one	
Modify the current traffic generator to generate IPv6 packets	We modify the ttcp, stg , rtg, stcp and rtcp to support IPv6 traffic generator.
The pure IPv6 simulation with simple network topology	The simulation of all of these types of packets (TCP/UDP/ICMPv6) is OK. In this stage, we modify the Linux kernel and simulator codes to support the basic IPv6 simulation.

Modify the Linux kernel to support the function of broadcast for IPv6	This part is completed.
Modify the AODV module to support the IPv6 simulation	This part is completed.
Add modules to support mixed simulation of IPv4 and IPv6	This part is completed.
Phase two	
Support IPv6 emulations	This part is completed

As shown in the above table, we can see how much we have achieved for the IPv6 simulation. Currently, NCTUns is capable of supporting simulations of fixed networks and wireless networks with AODV and GOD routing protocols, and simulations with mixed IPv4 and IPv6 networks. In addition, NCTUns supports the emulation of the mixed IPv4 and IPv6 networks as well.

Appendix

1. The IPv6 Routing

To make the IPv6 packets be routed correctly, there is one topic worth to be paid attention to－the IPv6 forward system variables need to be set. How these variables are set is listed below.

```
sysctl –w net.ipv6.conf.all.forwarding=1
sysctl –w net.ipv6.conf.default.forwarding=1
```

If we don’t set these system variables, the packets will not be forwarded by the middle node. Therefore, the works of setting these system variables are necessary. For the purpose of saving the efforts for users to set these two variables, we have added these actions in nctuns.cc, the file of simulator engine.

2. Traffic Generators for IPv6

Due to the traffic generators supplied by NCTUns all in IPv4 protocol, if these user-level programs are not modified to support the network protocol of IPv6, the IPv6 packet cannot be generated, and the environment of IPv6 we develop cannot be tested. Therefore, before the environment of IPv6 is built up, traffic generators, such as ttcp_v6, stcp_ipv6, and so on, are modified to support the protocol of IPv6. By the way, the ping, an application program generating the ICMP packets, is needless to be modified because there is a real one supported by Linux called ping6.

3. The Added Broadcast Function for IPv6

Unlike the protocol of IPv4, no broadcasting function exists in the protocol of IPv6. The network protocol of IPv6 replaces the ability of broadcast with the ability of multicast; nevertheless, that mechanism is hard to be supported in NCTUns. Based on this, we implement a mechanism by assigning a specially designed address as the destination address to emulate the broadcast function in NCTUns. If the destination address format is like the format listed below, we will regard this type of packet as the broadcast.

- **The special address Format:**

3fff:00:00:DstNetID:00:00:ffff:ffff

The destination address in the special address format will be accepted by all of its neighbors. To make this scheme function correctly, the kernel function of tun_get_user() needs to be modified. The modifications are listed as follows:

```
static __inline__ ssize_t tun_get_user(struct tun_struct *tun, struct iovec *iv, size_t
count)
{
    …
    …
    if((ipv6_hdr->daddr.s6_addr16[7] & 0xffff)==0xffff)
```

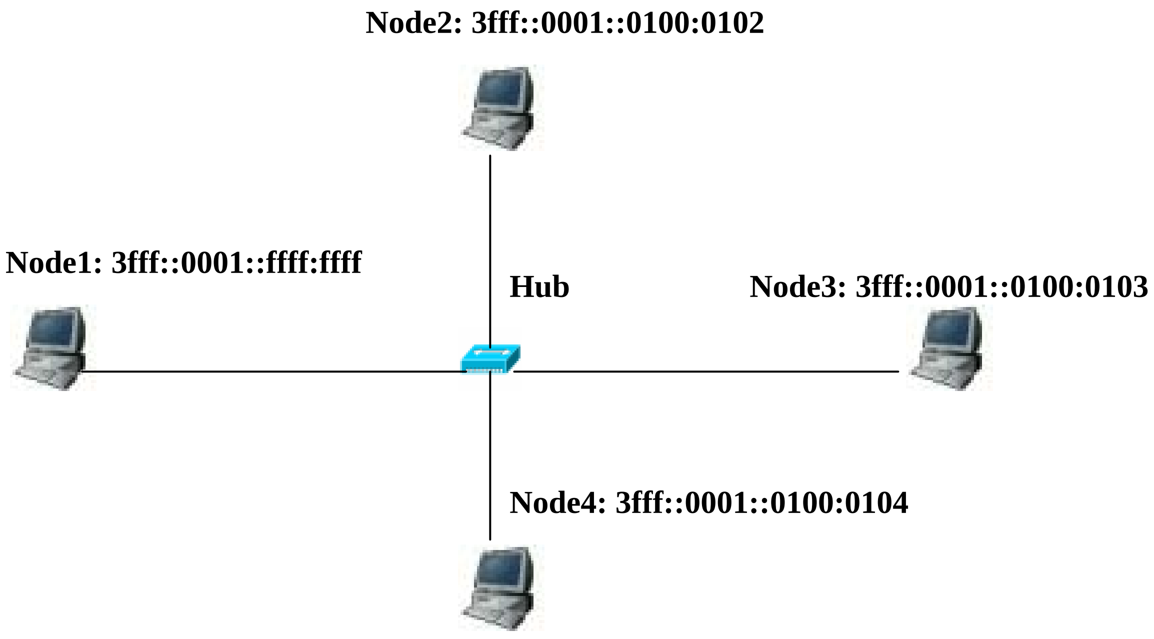
The address with special word (0xffff) will be replaced with the address of this node. Therefore, this special address could be accepted by every node.

```
                {
                        memcpy((void *)&ipv6_hdr->daddr,
(const void *)&ipv6, sizeof(struct in6_addr));
                }
        …
}
```

To better understand of this scheme, an example is given below.

Example:

When node 1 wants to broadcast the packet to its neighbors within the specified subnet it belongs to, it sends the packet with the destination address of 3fff::0001::ffff:ffff. This kind of packet would be accepted by all of its neighbors (node2, node3 and node4).



Node2: 3fff::0001::0100:0102
Node1: 3fff::0001::ffff:ffff
Hub
Node3: 3fff::0001::0100:0103
Node4: 3fff::0001::0100:0104

Figure 4.1: Supporting the broadcast function for IPv6

Reference

[1] Thomson & Narten, IPv6 Stateless Address Autoconfiguration, RFC 2462, December 1998

[2] Hinden & Deering, Internet Protocol Version 6 (IPv6) Specification, RFC 2460, December 1998

[3] Hinden & Deering, IP Version 6 Addressing Architecture, RFC 2373, July 1998

[4] IAB & IESG, IPv6 Address Allocation Management, RFC 1881, December 1995

[5] Hinden & Deering, IPv6 Multicast Address Assignments, RFC 2375, July 1998

[6] McCann, Deering & Mogul, Path MTU Discovery for IP version 6, RFC 1981, August 1996

[7] Conta & Deering, Generic Packet Tunneling in IPv6, RFC 2473, December 1998

[8] Gilligan & Nordmark, Transition Mechanisms for IPv6 Hosts and Router, RFC 2893, August 2000

[9] Carpenter & Moore, Connection of IPv6 Domains via IPv4 Clouds, RFC 3056, February 2001

[10] K.C. Liao, Porting the NCTUns Network Simulatior to Linux and Supporting Emulation, June 2004

[11] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin, The Design and Implementation of the NCTUns 1.0 Network

Simulation Engine, Computer Networks, Vol. 42, Issue 2, June 2003, pp. 175-197

[12] Porting applications to IPv6 How To, http://gsyc.escet.urjc.es/~eva/IPv6-web/ipv6.html#RFC2731

[13] Tomohiro Fujisaki and Junya Kato, Tech Tutorials, http://www.ipv6style.jp/en/tech/20050509/index.shtml