Use Data Plane Development Kit (DPDK) to build my own UDP protocol stack

INF545 Independent Project

Author: Chuhao Tang

Internet of Things: Innovation & Management M1Ecole Polytechnique

Table of contents

1 Introduction:

1.1 The reason why I choose this topic as my independent project

Throughout the master's studies at m1 in Ecole Polytechnique, I have learnt a lot of knowledges about the networking and protocols. I learned how to use java to apply some network protocols in the INF557 course, and learned a lot of theoretical knowledge about network protocols in the INF566 course. As I become more and more interested in network programming, I want to understand how the protocol really works and how it is implemented in the application. So I chose not to apply the pre-provided protocol interface, but to find a way to gradually realize my own protocol stack. In this way, I choose this topic as my independent project.

1.2 The reason why we need DPDK to deal with the networking:

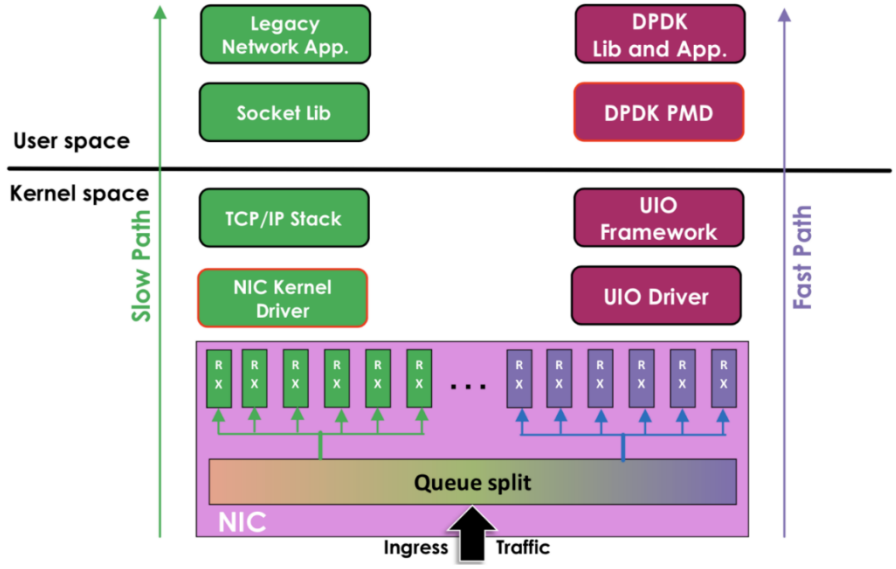
From the user aspect, we can feel that the network speed has been improving. The development of network technology has also evolved from 1GE to 100GE. From this point of view, we can conclude that the network IO capability of a single machine must keep up with the development and the demand of the network.

The traditional telecommunications area is based on the hardware solution, such as routers, switches, firewalls and other equipment. The disadvantage of the hardware is very obvious: it is not easy to debug and maintain. With the development of the mobile technology, hardware-based businesses cannot iterate quickly. At the same time, the emergence of private cloud is trending to share hardware through Network Functions Virtualization[1]. So now there is an urgent need for a high-performance network IO development framework with software architecture.

2 The main characteristics of DPDK

2.1 Linux User space I/O

The huge difference between DPDK and Linux kernel networking is that the former one uses Linux user space to deal with the protocol stack, and the latter uses Linux kernel to handle the protocol stack. From the previous analysis, we can know the way of IO implementation and the uncontrollable factors of data flow through the kernel, which are all implemented in the Linux kernel. The kernel is the cause of the bottleneck. In order to solve the problem, the kernel needs to be bypassed, we need to use a new space to deal with the packet in the protocol stack. Therefore, the mainstream solution is to bypass the network card IO, bypass the kernel and directly send and receive packets in the user mode to solve the bottleneck of the kernel.



[2]

On the left side of the picture is what data transfer in the traditional way: Network interface card, Driver, Protocol stack, Socket interface and finally to the Application. On the tight side is what DPDK deal with the data: Network interface card, DPDK polling mode, DPDK basic library and Application. In this way, DPDK bypasses the network driver module of the Linux kernel and directly reaches the user space from the network hardware without frequent memory copies and system calls. DPDK bare packet bounce requires 80 clock cycles per packet, while the traditional Linux kernel protocol stack requires 2k~4k clock cycles per packet.

UIO technology enables applications to directly operate the memory space of the device through user space drivers, avoiding multiple copies of data in the kernel buffer and application buffer, and improving data processing efficiency. DPDK enables faster development of high-speed packet networking applications, which means it allows building applications that can process packets faster. By bypassing the kernel processing, it reduces the time brought by CPU context switch.

2.2 DPDK core optimization: PMD

Before diving into the optimization down by the DPDK polling mode, we have to distinguish the polling and interrupt. Interrupts are a hardware mechanism that can occur at any time during the normal running of a program by the CPU. Polling is more like a protocol or policy down by the CPU rather than a hardware mechanism: Polling is a way for the CPU to decide how to provide services to peripheral devices[3]. In the polling process, the CPU regularly sends inquiries, and inquires whether each peripheral device needs its services in turn. The synchronous model completes an individual I/O faster and consumes less CPU clock cycles despite having to poll[4].

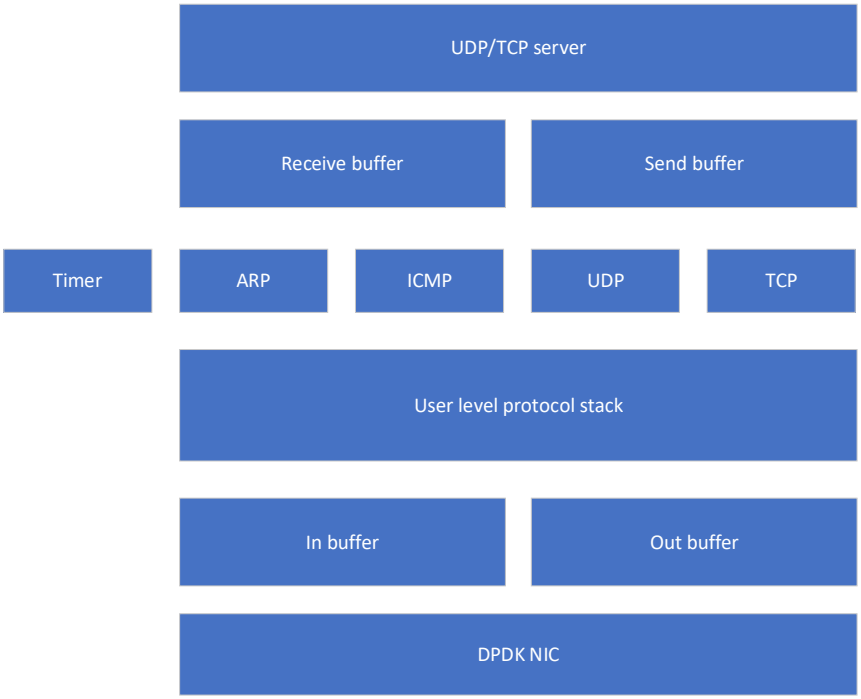
DPDK's UIO driver shields hardware from interrupts, and then uses active polling in user mode. This mode is called PMD, which is Poll Mode Driver. UIO bypasses the kernel and actively polls to remove hard interrupts, so that DPDK can process sending and receiving packets in user mode. It brings the benefits of Zero Copy[5] and no system calls, and synchronous processing reduces the Cache Miss caused by context switching.

3 Use DPDK to implement a UDP protocol stack

3.1 General idea

Because of the low speed's process in the Linux kernel, I want the whole stack analysis and process job bypass the Linux kernel so that we can get a higher speed. In this way, I can't use the protocol stack or the API function that the Linux kernel provide, which means that I need my own protocol stack for the DPDK. The general idea is to create the complete protocol stack to deal with the incoming packet by using DPDK. I envision the functionality to be implemented includes: Receive and send back the UDP message; Do the ARP request periodically and establish the ARP table for the protocol stack; Send back the ICMP check message back to where it requests; Implement a UDP server which can integrate various above function.

The overall architecture diagram is shown in the figure below:



In the following section, I will show every step of the implementation of the UDP protocol stack framework.

3.2 The implementation of the UDP protocol stack:

3.2.1 Bind the DPDK network interface card

Since I will not use the Linux kernel to process the packet, the first step is to receive the packet in a special NIC, which is the DPDK multi-queue network card. In this process, I make a IP address and MAC address record of the NIC, such as the network card eth0, and then let this NIC status to be down in order to avoid the packet receive from this NIC go into the Linux kernel. The final step is to bind the eth0 NIC to the DPDK by using the DPDK setup program. After that, we can get the packet through the NIC eth0 and analyze the packet through my own user level protocol stack.

3.2.2 Receive and send back the UDP packet

DPDK provide some of the API function to let us receive and send the packet: rte_eth_rx_burst() and rte_eth_tx_burst(). These two functions have the ability to communicate with the DPDK NIC, so the core of this section is, how to analyze that it is a UDP packet and how to arrange a UDP packet and send it back.

Analyzing and assembling packets is essentially a reverse process, so here I take the process of UDP packets assembling as an example to show the whole process. The most important part of this process is the assembling the different header. Since I didn't use any API function in the Linux kernel, the data I get from or send to the DPDK NIC is just a stream of data. I have to deconstruct it step by step to receive the UDP message, and on the sending side of process, I should assemble the UDP message with the header.

Here is the general structure of a UDP packet:

Mac header(14 bytes)	IP header(20 bytes)	UDP header(8 bytes)	Payload

The structure is clear to understand, when I try to send a message to the destination host, I put the UDP message into the payload first. Then I add various header to the payload, in which I should arrange the checksum, source IP address, destination IP address and so on into the header. The receiving process is quite the same. It deconstructs the packet layer-by-layer and finally get the payload message.

Here is the detail of the code when I assemble the header:

```
//package udp pkt
static int ng_encode_udp_pkt(uint8_t *msg, unsigned char *data, uint16_t total_len){
        //etherhdr:
        struct rte_ether_hdr *eth=(struct rte_ether_hdr *)msg;
        rte_memcpy(eth->s_addr.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
        rte_memcpy(eth->d_addr.addr_bytes, gDstMac, RTE_ETHER_ADDR_LEN);
        eth->ether_type=htons(RTE_ETHER_TYPE_IPV4);

        //iphdr:
        struct rte_ipv4_hdr *ip=(struct rte_ipv4_hdr *)(msg+sizeof(struct rte_ether_hdr));//set the offset
        ip->version_ihl= 0x45;
        ip->type_of_service= 0;
        ip->total_length = htons(total_len - sizeof(struct rte_ether_hdr));
        ip->packet_id=0;
        ip->fragment_offset=0;
        ip->time_to_live=64;//default value is 64
        ip->next_proto_id=IPPROTO_UDP;
        ip->src_addr=gSrcIp;
        ip->dst_addr=gDstIp;
        ip->hdr_checksum=0;//first set to 0
        ip->hdr_checksum=rte_ipv4_cksum(ip);

        //udphdr:
        struct rte_udp_hdr *udp=(struct rte_udp_hdr *)(msg+sizeof(struct rte_ether_hdr)+sizeof(struct rte_ipv4_hdr));
        udp->src_port=gSrcPort;
        udp->dst_port=gDstPort;
        uint16_t udplen=total_len-sizeof(struct rte_ether_hdr)-sizeof(struct rte_ipv4_hdr);
        udp->dgram_len=htons(udplen);
        rte_memcpy((uint8_t*)(udp+1),data,udplen);//udp+1, go to the data part
        udp->dgram_cksum=0;
        udp->dgram_cksum=rte_ipv4_udptcp_cksum(ip,udp);

        //to check the code:
        // struct in_addr addr;
        // addr.s_addr=gSrcIp;
        // printf("-->src: %s:%d, ",inet_ntoa(addr),ntohs(gSrcPort));
        // addr.s_addr=gDstIp;
        // printf("dst: %s:%d\n ",inet_ntoa(addr),ntohs(gDstPort));

        return 0;
}
```

It is worth nothing that, the premise of realizing data packet sending and receiving is that both parties in the communication can correctly resolve the other party's mac address through the IP address. I establish a new static ARP entry in my Windows machine in advance so that I can guarantee the communication between my Windows system and my Linux virtual machine. The problem here is that I must create a static ARP entry first to communicate, which seems ridiculous in our real life. The next section is aiming at dealing with this problem: to send a ICMP message back towards the ARP reply message.

3.2.3 Send back the ICMP check message

In this section, the goal is to let the Windows machine can get the Mac address automatically through ARP protocol. The general process is that I first delete the static ARP entry which I created in the last section, and then ping my eth0 IP address to get the feedback and the Mac address.

In my user level protocol stack, I should first analyze the ARP packet sent by the Windows machine through ping command. This process of deconstruct is similar to the same process of UDP message. We can analyze the ARP packet through its header, and then take the Windows IP and Mac address. I assemble these addresses into the out ARP packet to send it back to where it originated. The thing needs to be noticed is that the ARP packet is directly with the Mac header, not going through the IP protocol.

By only send back the ARP reply message, the Windows machine side will not get any feedback. Even though the IP and Mac address has already been added into the ARP table, the user of the Windows machine still think that the ping command is unsuccessful. Under this kind of situation, I also want to send a ICMP message back so that the one who send the ARP packet can get correct response information. The process is also the same as packeting the UDP and ARP packet.

Here are the code logic of the ARP and ICMP packet:

```
static int ng_encode_arp_pkt(uint8_t *msg, uint8_t *dst_mac, uint32_t sip, uint32_t dip){
        //it is a network layer protocol, includes ether hdr, arp hdr

        //etherhdr:
        struct rte_ether_hdr *eth=(struct rte_ether_hdr *)msg;
        rte_memcpy(eth->s_addr.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
        rte_memcpy(eth->d_addr.addr_bytes, dst_mac, RTE_ETHER_ADDR_LEN);
        eth->ether_type=htons(RTE_ETHER_TYPE_ARP);

        //arp hdr:
        struct rte_arp_hdr *arp=(struct rte_arp_hdr *)(eth+1);
        arp->arp_hardware=htons(1);
        arp->arp_protocol=htons(RTE_ETHER_TYPE_IPV4);
        arp->arp_hlen=RTE_ETHER_ADDR_LEN;
        arp->arp_plen=sizeof(uint32_t);
        arp->arp_opcode=htons(2);
        rte_memcpy(arp->arp_data.arp_sha.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
        rte_memcpy(arp->arp_data.arp_tha.addr_bytes, dst_mac, RTE_ETHER_ADDR_LEN);
        arp->arp_data.arp_sip=sip;
        arp->arp_data.arp_tip=dip;

        return 0;
}
```

```c
static int ng_encode_icmp_pkt(uint8_t *msg, uint8_t *dst_mac, uint32_t sip, uint32_t dip,uint16_t id,uint16_t seqnum){
        //it is an transportation layer protocol, include 3 header: ether, ip, icmp

        //ether hdr
        struct rte_ether_hdr *eth=(struct rte_ether_hdr *)msg;
        rte_memcpy(eth->s_addr.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
        rte_memcpy(eth->d_addr.addr_bytes, dst_mac, RTE_ETHER_ADDR_LEN);
        eth->ether_type=htons(RTE_ETHER_TYPE_IPV4);//define the network layer protocol

        //ip hdr
        struct rte_ipv4_hdr *ip=(struct rte_ipv4_hdr *)(msg+sizeof(struct rte_ether_hdr));//set the offset
        ip->version_ihl= 0x45;
        ip->type_of_service= 0;
        ip->total_length = htons(sizeof(struct rte_ipv4_hdr) + sizeof(struct rte_icmp_hdr));
        ip->packet_id=0;
        ip->fragment_offset=0;
        ip->time_to_live=64;//default value is 64
        ip->next_proto_id=IPPROTO_ICMP;//define the application layer protocol
        ip->src_addr=sip;
        ip->dst_addr=dip;
        ip->hdr_checksum=0;//first set to 0
        ip->hdr_checksum=rte_ipv4_cksum(ip);


        //icmp hdr
        struct rte_icmp_hdr *icmp=(struct rte_icmp_hdr *)(msg+sizeof(struct rte_ether_hdr)+sizeof(struct rte_ipv4_hdr));
        icmp->icmp_type=RTE_IP_ICMP_ECHO_REPLY;
        icmp->icmp_code=0;
        icmp->icmp_ident=id;
        icmp->icmp_seq_nb= seqnum;
        icmp->icmp_cksum=0;
        icmp->icmp_cksum = ng_checksum((uint16_t*)icmp, sizeof(struct rte_icmp_hdr));
        return 0;
}
```

3.2.4 Establish the ARP table for the protocol stack

Until now, the protocol stack has accomplished the function to respond the ARP request message and send back the ICMP message. But if I didn't use ping command to ask the Mac address of my Linux virtual machine IP address, they won't communicate to each other through the internet, which is definitely what I don't want to see. The current requirement is that, the protocol stack can automatically broadcast the ARP request message to the destination that it can reach. In this way, when we start up the Linux protocol stack, it can record the Mac address of the target machine without using ping command in the Windows machine. The final result I want to get is that it can establish its own ARP-table in the protocol stack.

In this module, there are three vital things needed: The first one is to arrange an ARP request packet and broadcast it to the neighbors. The second thing is to set a timer in order to broadcast periodically. The final thing is that after receive the ARP reply from other machine, the protocol stack can store the Mac address and the IP address into its own ARP-table, so that the next time it needs to communicate this host, the Mac address can be easily resolved through the IP address.

The most and foremost thing is to define the data structure of the ARP-table. Here I decided to use linked list to store the ARP entry and I also define the add and delete function related to the linked list. The code logic is shown below:

```c
//linked list
struct arp_entry {

        uint32_t ip;
        uint8_t hwaddr[RTE_ETHER_ADDR_LEN];
        uint8_t type;
        // the size is not matching


        struct arp_entry *next;
        struct arp_entry *prev;

};

struct arp_table {

        struct arp_entry *entries;
        int count;

};
```

The next part is to package an ARP request packet. The whole procedure will be quite similar to the package process above in this article, so I just simply skip it. It is worth noting that there is an important point which is related to the mac address that put into the header of Ether and ARP. Since right now the machine don't know the Mac address, it should broadcast in the same network segment. I grabbed an ARP package from the Wireshark software, and the structure of it will be like:

```
Ethernet II, Src: 4a:f1:c3:a0:d2:b3 (4a:f1:c3:a0:d2:b3), Dst:
  Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  Source: 4a:f1:c3:a0:d2:b3 (4a:f1:c3:a0:d2:b3)
  Type: ARP (0x0806)
  Trailer: 00000000000000000000000000000000
Address Resolution Protocol (request)
  Hardware type: Ethernet (1)
  Protocol type: IPv4 (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: request (1)
  Sender MAC address: 4a:f1:c3:a0:d2:b3 (4a:f1:c3:a0:d2:b3)
  Sender IP address: 10.222.12.242
  Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Target IP address: 10.222.14.136
```

From this we can see that the Target Mac address in ARP header is 00:00:00:00, but on the other hand, the destination broadcast Mac address in ethernet header is FF:FF:FF:FF. This is such an important thing when I arrange the ARP request packet.

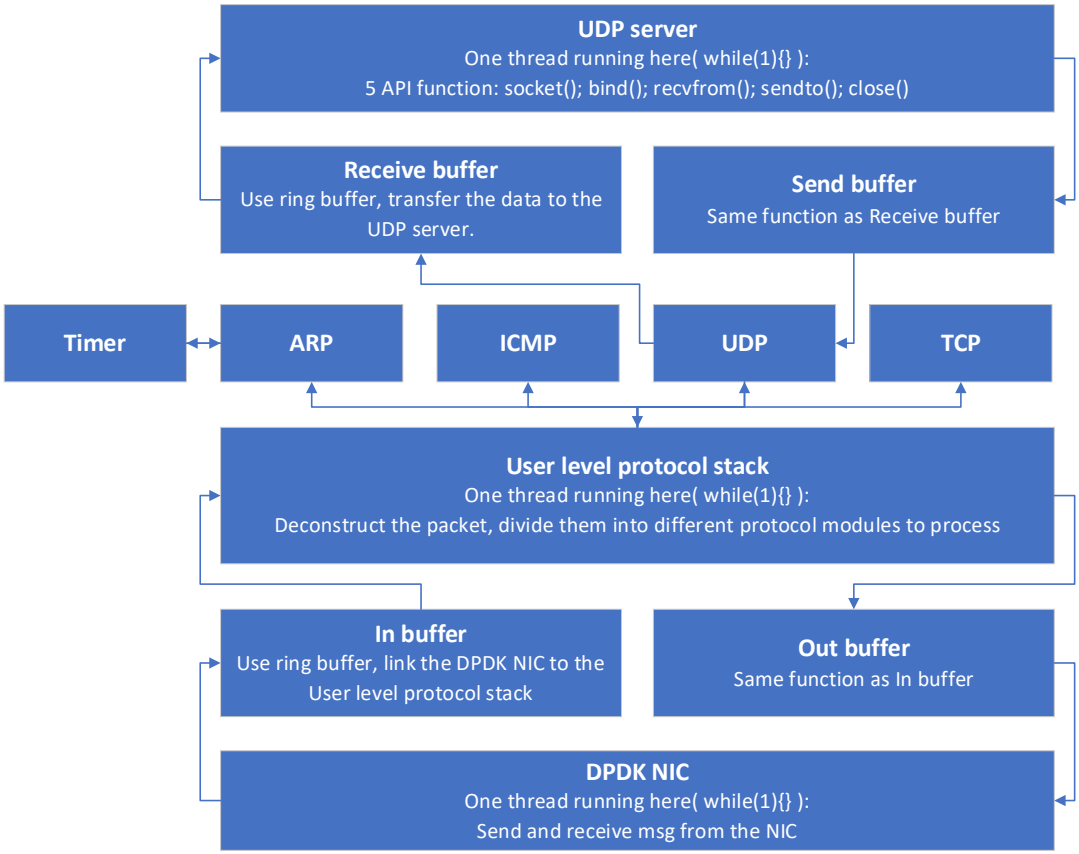
After defining the ARP-table and organizing the ARP request package, the only thing I left is to broadcast the packet periodically. In the following code, I establish a timer and set the reset time to 60 second, which means that the ARP request packet will be sent every 60 second:

```c
//setup a timer:
#if ENABLE_TIMER

        rte_timer_subsystem_init();

        struct rte_timer arp_timer;
        rte_timer_init(&arp_timer);

        uint64_t hz = rte_get_timer_hz();
        unsigned lcore_id = rte_lcore_id();
        rte_timer_reset(&arp_timer, hz, PERIODICAL, lcore_id, arp_request_timer_cb, mbuf_pool);//PERIODICAL: multiply trigger the timer

#endif
```

In general, in this chapter, I realized the establishment of the ARP table and sent ARP request messages regularly.

3.2.5 UDP network architecture

The detailed network architecture is shown in the figure below:



In general, this network architecture is composed of two loop structures and three threads running simultaneously. Next, I will disassemble and explain my UDP network architecture step by step.

The first layer of this architecture is the thread in DPDK NIC. In this thread or while loop, it aims at communicating with the DPDK multi-queue network card. It should keep listening to this port and convey the incoming messages to the in buffer, also keep receiving the packet from out buffer and send to the NIC. It is the external interface of the entire network structure and plays the role of connecting to the external network. Secondly, the connection between it and the protocol stack through buffer can better buffer the data, and the data processing in the whole architecture will be more reasonable.

The middle layer of the architecture is the thread in user level protocol stack. From my perspective, this is the most important part in my UDP network architecture. First of all, it needs to receive and send the data in the ring buffer of the lower layer. It assumes the role of connecting the relationship between the upper and lower layers. The most important thing is that this layer is responsible for parsing the data packets and determining which protocol module to distribute the parsed data packets to. In this while loop, it should not only link the upper layer and lower layer, but also contain the function of construct and deconstruct various protocols and distribute them to their own protocol process module.

The topmost layer of the architecture is the thread in UDP server. There are many API functions to implement UDP in the Linux kernel, and it is very convenient. Nevertheless, since I want to use DPDK to process the packet, I bypass the Linux kernel to handle packets, which

means that I have to implement these API functions for my own protocol stack. It is not difficult to see that I need to implement the five most basic functions: socket, bind, recvfrom, sendto and close. The following figure is the code structure of my own API functions:

```
//create a udp server:
static int udp_server_entry(__attribute__((unused))  void *arg) {

        //create a socket
        int connfd = nico_socket(AF_INET, SOCK_DGRAM, 0);
        if (connfd == -1) {
                printf("sockfd failed\n");
                return -1;
        }

        struct sockaddr_in localaddr, clientaddr; // struct sockaddr
        memset(&localaddr, 0, sizeof(struct sockaddr_in));

        //bind port and ip adress
        localaddr.sin_port = htons(8888);
        localaddr.sin_family = AF_INET;
        localaddr.sin_addr.s_addr = inet_addr("129.104.95.11");
        nico_bind(connfd, (struct sockaddr*)&localaddr, sizeof(localaddr));

        char buffer[UDP_APP_RECV_BUFFER_SIZE] = {0};
        socklen_t addrlen = sizeof(clientaddr);
        while (1) {

                if (nico_recvfrom(connfd, buffer, UDP_APP_RECV_BUFFER_SIZE, 0, (struct sockaddr*)&clientaddr, &addrlen) < 0) {

                        continue;

                } else {
                        printf("recv from %s:%d, data:%s\n", inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port), buffer);
                        nico_sendto(connfd, buffer, strlen(buffer), 0, (struct sockaddr*)&clientaddr, sizeof(clientaddr));
                }
        }
        nico_close(connfd);

}
```

The two loop structures are based on the ring buffer data structure. In the two ring structures, the buffer is disassembled into two parts, in and out (or receive and send). This is done for the following reasons: This can enable the isolation of output and output data, making the structure of the data clearer. It can also guarantee the data security when I implement multi-threading, in which I use mutex to protect the thread data in the in buffer and receive buffer. In order not to cause deadlock, the out buffer and send buffer shouldn't be implemented the mutex.

4 Tests and Results

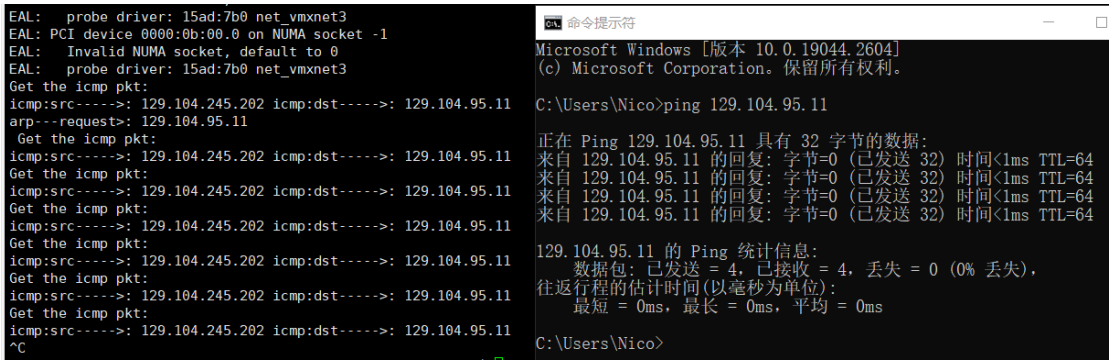
The testing process will follow the order of: ICMP message test, ARP broadcast function test, Receive and send function test.

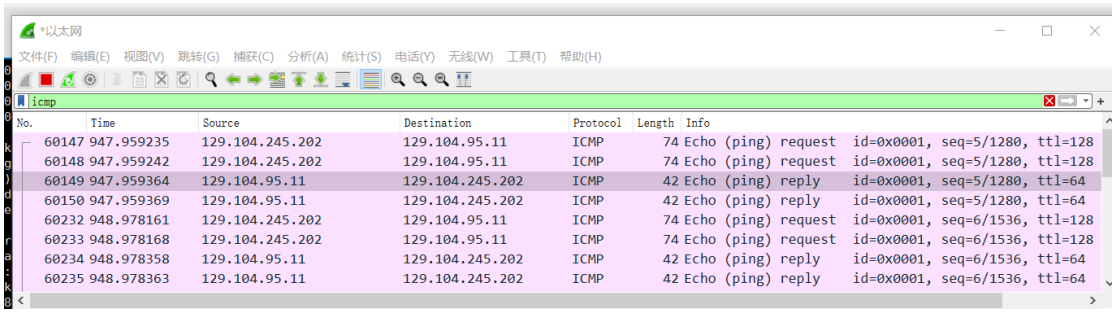
The testing tools include: Wireshark, Net assist, CMD in windows, VMware, Xshell.

4.1 ICMP message test

Usually, the easiest way for us to confirm whether the target IP is reachable is to use the

ping command. Also, I want to establish an ARP table for my Linux virtual machine in the next section. Here I use command “ping 129.104.95.11” to send ICMP message to the target Linux host and use Wireshark to trace the route of the message between the two host:

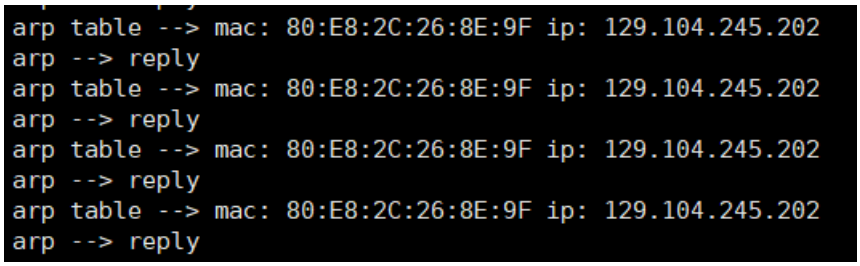




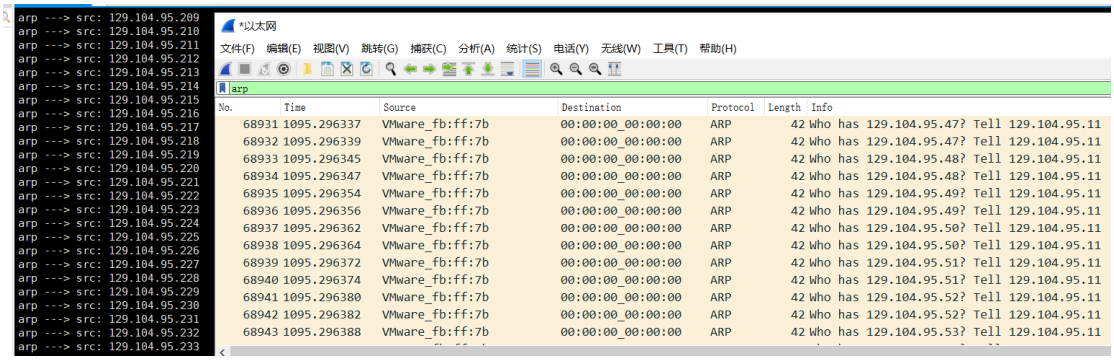
We can see that the ICMP message from my Windows host well send to the Linux host and successfully send back to where it came from.

4.2 ARP broadcast function test

Since I “ping” in the previous section, the Linux host has already know the IP and Mac address of the Windows host “129.104.245.202”. It has established the ARP table and store the mapping of IP and mac address. Now the two hosts both have known each other’s mapping of IP and Mac, which means that they can communicate to each other now without configure the ARP table manually. The process is shown below:

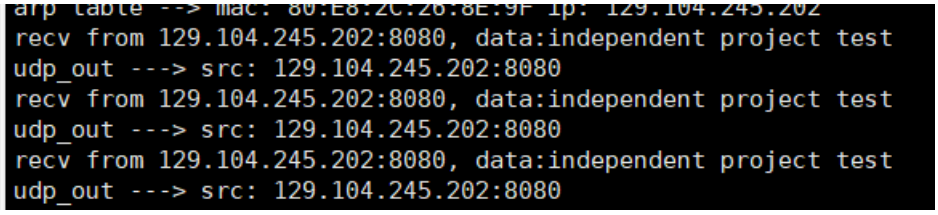


And also, in order to know that if there is any new host link to this network segment, I set a functionality to broadcast the ARP request message every 60s. All the message come from the ARP reply will be recorded into the ARP table. Since I don’t connect any new host to this network segment, this functionality I present here is just simply broadcast the ARP request message. I use terminal in Linux and Wireshark to show the result:



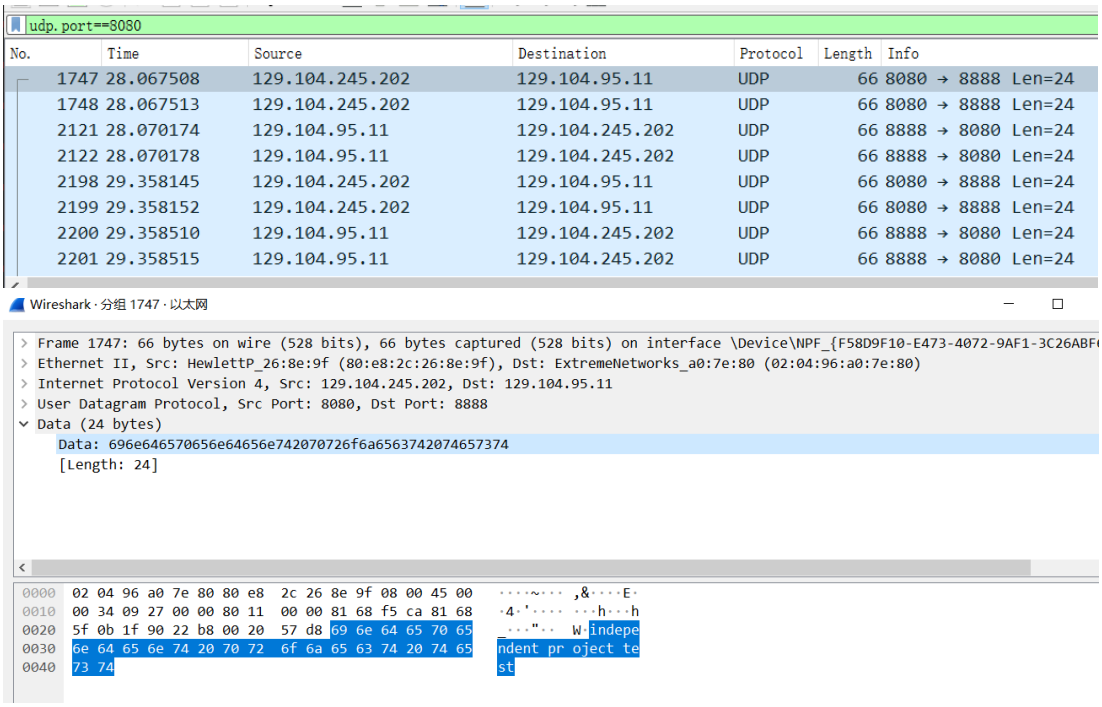
4.3 Receive and send function test

This test procedure is help by a third-party software named Net assist, it can use my host IP to send the message to my Linux virtual machine IP and show the received message. A brief summary of this software is: the black button in the upper left corner starts to bind the IP and port number of the windows host. I can select the destination IP address in the lower right corner and press the green button to send the message. In the monitor area, the blue message represents the message host send to the Linux machine, while the green one represents the incoming message from Linux machine. Here is the test result:



From the interface of Net assist and the terminal of the Linux machine, we can easily get the information that, my Windows host 129.104.245.202 can send and receive message “independent project test“ from the Linux virtual machine 129.104.95.11.

If I use the Wireshark to track the message in the ethernet of my computer, the result would be clearer, as shown below:



5 Conclusion

Because of the time constraints, I only realize some basic protocols and the UDP protocol stack. Therefore, the continuation of the independent project is to implement the TCP protocol stack, and then have a certain understanding of VPP. I will keep finishing those part in my following graduate project.

Overall, from this independent project, I have become more aware of the overall architecture of network protocols because I used DPDK to build the UDP user mode protocol stack step by step. This experience let me have a deeper understanding of protocol types, protocol headers, the process of parsing protocols and concepts such as IP, mac, and ARP table. I also improved my c code programming and debugging ability, and I was familiar with how to modify the server code, which greatly improved my Linux network programming ability.

6 Reference:

[1] Han B, Gopalakrishnan V, Ji L, et al. Network function virtualization: Challenges and opportunities for innovations[J]. IEEE communications magazine, 2015, 53(2): 90-97.

[2] Flow Bifurcation on Intel® Ethernet Controller X710/XL710

[3] Prasojo P T. How DPDK Enhance OvS Performance[J].

[4] Yang J, Minturn D B, Hady F. When poll is better than interrupt[C]//FAST. 2012, 12: 3-3.

[5] Tianhua L, Hongfeng Z, Guiran C, et al. The design and implementation of zero-copy for linux[C]//2008 Eighth International Conference on Intelligent Systems Design and Applications. IEEE, 2008, 1: 121-126.

7 Appendix:

All the code can be found in my GitHub website: https://github.com/nicottttt/dpdk-learning
Some of the codes refer to https://ke.qq.com/course/3941319#

7.1 udp.c

```
#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_mbuf.h>
#include <stdio.h>
#include "arp.h"
#include <arpa/inet.h>
#include <rte_malloc.h>
#include <rte_timer.h>

#define NUM_MBUFS (4096-1)
#define ENABLE_SEND 1
#define ENABLE_ARP 1
#define ENABLE_ICMP 1
#define ENABLE_ARP_REPLY 1
#define ENABLE_DEBUG 1
#define ENABLE_TIMER 1
#define TIMER_RESOLUTION_CYCLES 120000000000ULL // 10ms * 1000 = 10s * 6
#define ENABLE_RINGBUFFER 1
#define RING_SIZE 1024
#define ENABLE_MULTITHREAD 1
#define BURST_SIZE  32
#define ENABLE_UDP_APP 1
#define UDP_APP_RECV_BUFFER_SIZE 128

#if ENABLE_SEND
//ip,mac,port
//define it as a global para means that only allow one client
#define MAKDE_IPV4_ADDR(a,b,c,d) (a+(b<<8)+(c<<16)+(d<<24))//ipadress

static uint32_t gLocalIp=MAKDE_IPV4_ADDR(129,104,95,11);

static uint32_t gSrcIp;
static uint32_t gDstIp;

static uint8_t gSrcMac[RTE_ETHER_ADDR_LEN];
static uint8_t gDstMac[RTE_ETHER_ADDR_LEN];
```

```
static uint16_t gSrcPort;
static uint16_t gDstPort;
#endif

#if ENABLE_ARP_REPLY

static uint8_t gDefaultArpMac[RTE_ETHER_ADDR_LEN] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF};

#endif

//
#if ENABLE_RINGBUFFER
struct inout_ring{
  struct rte_ring *inring;
  struct rte_ring *outring;
};

static struct inout_ring *rInst=NULL;

//initialize
static struct inout_ring *ringInstance(void){
  if(rInst==NULL){
    rInst=rte_malloc("in/out ring", sizeof(struct inout_ring), 0);
    memset(rInst, 0, sizeof(struct inout_ring));
  }
  return rInst;
}

#endif


int gDpdkPortId=0;

static const struct rte_eth_conf port_conf_default={
  .rxmode={.max_rx_pkt_len=RTE_ETHER_MAX_LEN}
};

static void ng_init_port(struct rte_mempool *mbuf_pool){//initialize
  uint16_t nb_sys_ports=rte_eth_dev_count_avail();
  if(nb_sys_ports==0){
    rte_exit(EXIT_FAILURE,"No support\n");
  }

  struct rte_eth_dev_info dev_info;
```

```
  rte_eth_dev_info_get(gDpdkPortId, &dev_info);

  const int num_rx_qeues=1;
  const int num_tx_qeues=1;

  struct rte_eth_conf port_conf=port_conf_default;

  //write the configuration
  rte_eth_dev_configure(gDpdkPortId, num_rx_qeues, num_tx_qeues, &port_conf);
  //set rx queue
  if(rte_eth_rx_queue_setup(gDpdkPortId,0, 1024,
rte_eth_dev_socket_id(gDpdkPortId),NULL, mbuf_pool)<0){
    rte_exit(EXIT_FAILURE,"Could not setup RX queue\n");
  }

#if ENABLE_SEND
  //set tx queue
  struct rte_eth_txconf txq_conf=dev_info.default_txconf;
  txq_conf.offloads= port_conf.rxmode.offloads;//send and receive have the same size
  if(rte_eth_tx_queue_setup(gDpdkPortId,0, 1024,
rte_eth_dev_socket_id(gDpdkPortId),&txq_conf)<0){
    rte_exit(EXIT_FAILURE,"Could not setup TX queue\n");
  }
#endif

  if(rte_eth_dev_start(gDpdkPortId)<0){
    rte_exit(EXIT_FAILURE,"Could not start\n");
  }
  rte_eth_promiscuous_enable(gDpdkPortId);

};

#if ENABLE_SEND

static int ng_encode_udp_pkt(uint8_t *msg, unsigned char *data, uint16_t total_len)
{

  // encode

  // 1 ethhdr
  struct rte_ether_hdr *eth = (struct rte_ether_hdr *)msg;
  rte_memcpy(eth->s_addr.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
  rte_memcpy(eth->d_addr.addr_bytes, gDstMac, RTE_ETHER_ADDR_LEN);
  eth->ether_type = htons(RTE_ETHER_TYPE_IPV4);
```

```
  // 2 iphdr
  struct rte_ipv4_hdr *ip = (struct rte_ipv4_hdr *)(msg + sizeof(struct
rte_ether_hdr));
  ip->version_ihl = 0x45;
  ip->type_of_service = 0;
  ip->total_length = htons(total_len - sizeof(struct rte_ether_hdr));
  ip->packet_id = 0;
  ip->fragment_offset = 0;
  ip->time_to_live = 64; // ttl = 64
  ip->next_proto_id = IPPROTO_UDP;
  ip->src_addr = gSrcIp;
  ip->dst_addr = gDstIp;

  ip->hdr_checksum = 0;
  ip->hdr_checksum = rte_ipv4_cksum(ip);

  // 3 udphdr

  struct rte_udp_hdr *udp = (struct rte_udp_hdr *)(msg + sizeof(struct
rte_ether_hdr) + sizeof(struct rte_ipv4_hdr));
  udp->src_port = gSrcPort;
  udp->dst_port = gDstPort;
  uint16_t udplen = total_len - sizeof(struct rte_ether_hdr) - sizeof(struct
rte_ipv4_hdr);
  udp->dgram_len = htons(udplen);

  rte_memcpy((uint8_t*)(udp+1), data, udplen);

  udp->dgram_cksum = 0;
  udp->dgram_cksum = rte_ipv4_udptcp_cksum(ip, udp);

  struct in_addr addr;
  addr.s_addr = gSrcIp;
  printf(" --> src: %s:%d, ", inet_ntoa(addr), ntohs(gSrcPort));

  addr.s_addr = gDstIp;
  printf("dst: %s:%d\n", inet_ntoa(addr), ntohs(gDstPort));

  return 0;
}

//send msg(after receiving)
```

```
//the procedure is to package the udp pkt first, and then return the mbuf, and
finally send by the function burst.
static struct rte_mbuf *udp_send(struct rte_mempool *mbuf_pool, uint8_t *data,
uint16_t length){
  //mempool-->mbuf
  const unsigned total_len= length+14+20+8;//data length+eht_hdr+ip_hdr+udp_hdr
  struct rte_mbuf *mbuf=rte_pktmbuf_alloc(mbuf_pool);//set the starting point of the
mbuf
  if(!mbuf){
    rte_exit(EXIT_FAILURE,"alloc wrong\n");
  }
  mbuf->pkt_len=total_len;
  mbuf->data_len=total_len;

  uint8_t *pktdata= rte_pktmbuf_mtod(mbuf, uint8_t*);
  ng_encode_udp_pkt(pktdata,data,total_len);//package it to udp pkt
  return mbuf;
}

#endif

#if ENABLE_ARP

static int ng_encode_arp_pkt(uint8_t *msg, uint16_t opcode,uint8_t *dst_mac,
uint32_t sip, uint32_t dip){
  //it is a network layer protocol, includes ether hdr, arp hdr

  //if dont know where to send, the arphdr(dst_mac) will be FFFFFFFF and the machdr
will be 00000000, it is differendt!!!!

  //etherhdr:
  struct rte_ether_hdr *eth=(struct rte_ether_hdr *)msg;
  rte_memcpy(eth->s_addr.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
  if (!strncmp((const char *)dst_mac, (const char *)gDefaultArpMac,
RTE_ETHER_ADDR_LEN)) {
    uint8_t mac[RTE_ETHER_ADDR_LEN] = {0x0};//00000000
    rte_memcpy(eth->d_addr.addr_bytes, mac, RTE_ETHER_ADDR_LEN);
  } else {
    rte_memcpy(eth->d_addr.addr_bytes, dst_mac, RTE_ETHER_ADDR_LEN);
  }
  eth->ether_type = htons(RTE_ETHER_TYPE_ARP);

  //arp hdr:
  struct rte_arp_hdr *arp=(struct rte_arp_hdr *)(eth+1);
```

```
  arp->arp_hardware=htons(1);
  arp->arp_protocol=htons(RTE_ETHER_TYPE_IPV4);
  arp->arp_hlen=RTE_ETHER_ADDR_LEN;
  arp->arp_plen=sizeof(uint32_t);
  arp->arp_opcode=htons(opcode);
  rte_memcpy(arp->arp_data.arp_sha.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
  rte_memcpy(arp->arp_data.arp_tha.addr_bytes, dst_mac, RTE_ETHER_ADDR_LEN);
  arp->arp_data.arp_sip=sip;
  arp->arp_data.arp_tip=dip;

  return 0;
}

static struct rte_mbuf *arp_send(struct rte_mempool *mbuf_pool, uint16_t
opcode,uint8_t *dst_mac,uint32_t sip,uint32_t dip){
  const unsigned total_length=sizeof(struct rte_ether_hdr)+sizeof(struct
rte_arp_hdr);//14+28
  //allocate some mem for the buf
  struct rte_mbuf *mbuf=rte_pktmbuf_alloc(mbuf_pool);
  if(!mbuf){
    rte_exit(EXIT_FAILURE,"Allocate memory wrong\n");
  }

  mbuf->pkt_len=total_length;
  mbuf->data_len=total_length;

  uint8_t *pkt_data=rte_pktmbuf_mtod(mbuf,uint8_t *);
  ng_encode_arp_pkt(pkt_data, opcode, dst_mac, sip, dip);

  return mbuf;

}

#endif

#if ENABLE_ICMP

//cksum for icmp
static uint16_t ng_checksum(uint16_t *addr, int count) {

  register long sum = 0;

  while (count > 1) {
```

```
    sum += *(unsigned short*)addr++;
    count -= 2;

  }

  if (count > 0) {
    sum += *(unsigned char *)addr;
  }

  while (sum >> 16) {
    sum = (sum & 0xffff) + (sum >> 16);
  }

  return ~sum;
}


static int ng_encode_icmp_pkt(uint8_t *msg, uint8_t *dst_mac, uint32_t sip,
uint32_t dip,uint16_t id,uint16_t seqnum){
  //it is an transportation layer protocol, include 3 header: ether, ip, icmp

  //ether hdr
  struct rte_ether_hdr *eth=(struct rte_ether_hdr *)msg;
  rte_memcpy(eth->s_addr.addr_bytes, gSrcMac, RTE_ETHER_ADDR_LEN);
  rte_memcpy(eth->d_addr.addr_bytes, dst_mac, RTE_ETHER_ADDR_LEN);
  eth->ether_type=htons(RTE_ETHER_TYPE_IPV4);//define the network layer protocol

  //ip hdr
  struct rte_ipv4_hdr *ip=(struct rte_ipv4_hdr *)(msg+sizeof(struct
rte_ether_hdr));//set the offset
  ip->version_ihl= 0x45;
  ip->type_of_service= 0;
  ip->total_length = htons(sizeof(struct rte_ipv4_hdr) + sizeof(struct
rte_icmp_hdr));
  ip->packet_id=0;
  ip->fragment_offset=0;
  ip->time_to_live=64;//default value is 64
  ip->next_proto_id=IPPROTO_ICMP;//define the application layer protocol
  ip->src_addr=sip;
  ip->dst_addr=dip;
  ip->hdr_checksum=0;//first set to 0
  ip->hdr_checksum=rte_ipv4_cksum(ip);


  //icmp hdr
```

```
  struct rte_icmp_hdr *icmp=(struct rte_icmp_hdr *)(msg+sizeof(struct
rte_ether_hdr)+sizeof(struct rte_ipv4_hdr));//offset the ether header and the ip
header
  icmp->icmp_type=RTE_IP_ICMP_ECHO_REPLY;
  icmp->icmp_code=0;
  icmp->icmp_ident=id;
  icmp->icmp_seq_nb= seqnum;
  icmp->icmp_cksum=0;
  icmp->icmp_cksum = ng_checksum((uint16_t*)icmp, sizeof(struct rte_icmp_hdr));
  return 0;
}


static struct rte_mbuf *ng_send_icmp(struct rte_mempool *mbuf_pool,uint8_t
*dst_mac,uint32_t sip,uint32_t dip,uint16_t id,uint16_t seqnum){
  const unsigned total_length=sizeof(struct rte_ether_hdr)+sizeof(struct
rte_ipv4_hdr)+sizeof(struct rte_icmp_hdr);
  //allocate some mem for the buf
  struct rte_mbuf *mbuf=rte_pktmbuf_alloc(mbuf_pool);
  if(!mbuf){
    rte_exit(EXIT_FAILURE,"Allocate memory wrong\n");
  }

  mbuf->pkt_len=total_length;
  mbuf->data_len=total_length;

  uint8_t *pkt_data=rte_pktmbuf_mtod(mbuf,uint8_t *);
  ng_encode_icmp_pkt(pkt_data, dst_mac, sip, dip, id, seqnum);

  return mbuf;

}

#endif

// print ethernet mac adress
static void print_ethaddr(const char *name, const struct rte_ether_addr *eth_addr)
{
  char buf[RTE_ETHER_ADDR_FMT_SIZE];
  rte_ether_format_addr(buf, RTE_ETHER_ADDR_FMT_SIZE, eth_addr);
  printf("%s%s", name, buf);
}


#if ENABLE_TIMER
```

```
static void arp_request_timer_cb(__attribute__((unused)) struct rte_timer *tim,
void *arg) {

  struct rte_mempool *mbuf_pool = (struct rte_mempool *)arg;
  struct inout_ring *ring=ringInstance();
#if 0
  struct rte_mbuf *arpbuf = ng_send_arp(mbuf_pool, RTE_ARP_OP_REQUEST,
ahdr->arp_data.arp_sha.addr_bytes, ahdr->arp_data.arp_tip, ahdr->arp_data.arp_sip);
  rte_eth_tx_burst(gDpdkPortId, 0, &arpbuf, 1);
  rte_pktmbuf_free(arpbuf);

#endif

  int i = 0;
  //send to every machine link to this local network
  for (i = 1;i <= 254;i ++) {

    uint32_t dstip = (gLocalIp & 0x00FFFFFF) | (0xFF000000 & (i << 24));

    //print arp msg
    struct in_addr addr;
    addr.s_addr = dstip;
    printf("arp ---> src: %s \n", inet_ntoa(addr));

    struct rte_mbuf *arpbuf = NULL;
    uint8_t *dstmac = ng_get_dst_macaddr(dstip);
    if (dstmac == NULL) {

      //arphdr:FFFFFFFF
      //machdr:00000000 !!!!!
      arpbuf = arp_send(mbuf_pool, RTE_ARP_OP_REQUEST, gDefaultArpMac, gLocalIp,
dstip);

    } else {

      arpbuf = arp_send(mbuf_pool, RTE_ARP_OP_REQUEST, dstmac, gLocalIp, dstip);
    }

    // rte_eth_tx_burst(gDpdkPortId, 0, &arpbuf, 1);
    // rte_pktmbuf_free(arpbuf);
    rte_ring_mp_enqueue_burst(ring->outring, (void**)&arpbuf , 1, NULL);

  }
```

```
}

#endif


#if ENABLE_MULTITHREAD

#if ENABLE_UDP_APP

struct localhost{

  int fd;
  //unsigned int status;
  uint32_t localip;
  uint8_t localmac[RTE_ETHER_ADDR_LEN];
  uint16_t localport;
  int protocol;

  struct rte_ring *sndbuffer;
  struct rte_ring *rcvbuffer;

  struct localhost *prev;
  struct localhost *next;

  pthread_cond_t cond;
  pthread_mutex_t mutex;

};

static struct localhost *lhost=NULL;

#define DEFAULT_FD_NUM 3

static int get_fd_frombitmap(void) {
  int fd = DEFAULT_FD_NUM;
  return fd;
}

static struct localhost * get_hostinfo_fromip_port(uint32_t dip, uint16_t port,
uint8_t proto) {

  struct localhost *host;
```

```
  for (host = lhost; host != NULL;host = host->next) {
    if (dip == host->localip && port == host->localport && proto == host->protocol)
{
      return host;
    }
  }
  return NULL;

}

static int ng_encode_udp_apppkt(uint8_t *msg, uint32_t sip, uint32_t dip,
  uint16_t sport, uint16_t dport, uint8_t *srcmac, uint8_t *dstmac,
  unsigned char *data, uint16_t total_len) {

  // encode

  // 1 ethhdr
  struct rte_ether_hdr *eth = (struct rte_ether_hdr *)msg;
  rte_memcpy(eth->s_addr.addr_bytes, srcmac, RTE_ETHER_ADDR_LEN);
  rte_memcpy(eth->d_addr.addr_bytes, dstmac, RTE_ETHER_ADDR_LEN);
  eth->ether_type = htons(RTE_ETHER_TYPE_IPV4);


  // 2 iphdr
  struct rte_ipv4_hdr *ip = (struct rte_ipv4_hdr *)(msg + sizeof(struct
rte_ether_hdr));
  ip->version_ihl = 0x45;
  ip->type_of_service = 0;
  ip->total_length = htons(total_len - sizeof(struct rte_ether_hdr));
  ip->packet_id = 0;
  ip->fragment_offset = 0;
  ip->time_to_live = 64; // ttl = 64
  ip->next_proto_id = IPPROTO_UDP;
  ip->src_addr = sip;
  ip->dst_addr = dip;

  ip->hdr_checksum = 0;
  ip->hdr_checksum = rte_ipv4_cksum(ip);

  // 3 udphdr

  struct rte_udp_hdr *udp = (struct rte_udp_hdr *)(msg + sizeof(struct
rte_ether_hdr) + sizeof(struct rte_ipv4_hdr));
```

```
  udp->src_port = sport;
  udp->dst_port = dport;
  uint16_t udplen = total_len - sizeof(struct rte_ether_hdr) - sizeof(struct
rte_ipv4_hdr);
  udp->dgram_len = htons(udplen);

  rte_memcpy((uint8_t*)(udp+1), data, udplen);

  udp->dgram_cksum = 0;
  udp->dgram_cksum = rte_ipv4_udptcp_cksum(ip, udp);

  return 0;
}

static struct rte_mbuf * ng_udp_pkt(struct rte_mempool *mbuf_pool, uint32_t sip,
uint32_t dip, uint16_t sport, uint16_t dport, uint8_t *srcmac, uint8_t *dstmac,
uint8_t *data, uint16_t length) {

  // mempool --> mbuf

  const unsigned total_len = length + 42;

  struct rte_mbuf *mbuf = rte_pktmbuf_alloc(mbuf_pool);
  if (!mbuf) {
    rte_exit(EXIT_FAILURE, "rte_pktmbuf_alloc\n");
  }
  mbuf->pkt_len = total_len;
  mbuf->data_len = total_len;

  uint8_t *pktdata = rte_pktmbuf_mtod(mbuf, uint8_t*);

  ng_encode_udp_apppkt(pktdata, sip, dip, sport, dport, srcmac, dstmac, data,
total_len);

  return mbuf;

}

struct offload{//udp packet

  uint32_t sip;
  uint32_t dip;
  uint16_t sport;
  uint16_t dport;
```

```
  uint8_t protocol;
  unsigned char *data;
  uint16_t length;

};

static int udp_process(struct rte_mbuf *udpmbuf){

  struct rte_ipv4_hdr *iphdr=rte_pktmbuf_mtod_offset(udpmbuf ,struct rte_ipv4_hdr
*,sizeof(struct rte_ether_hdr));//get the ip hdr
  struct rte_udp_hdr *udphdr=(struct rte_udp_hdr *)(iphdr+1);//get the udp hdr,从
iphdr 开始偏移 iphdr 的长度

  struct localhost *host=get_hostinfo_fromip_port(iphdr->dst_addr, udphdr->dst_port,
iphdr->next_proto_id );
  if(host==NULL){//if dont find, just exit
    rte_pktmbuf_free(udpmbuf);
    return -3;
  }

  struct offload *ol=rte_malloc("offload", sizeof(struct offload),0);
  if(ol==NULL){
    rte_pktmbuf_free(udpmbuf);
    return -1;
  }

  ol->dip=iphdr->dst_addr;
  ol->sip=iphdr->src_addr;
  ol->dport=udphdr->dst_port;
  ol->sport=udphdr->src_port;
  ol->protocol=IPPROTO_UDP;
  ol->length=ntohs(udphdr->dgram_len);

  ol->data=rte_malloc("unsigned cahr*", ol->length -sizeof(struct rte_udp_hdr),0);
  if(ol->data==NULL){
    rte_pktmbuf_free(udpmbuf);
    rte_free(ol);
    return -2;
  }

  rte_memcpy(ol->data, (unsigned char *)(udphdr+1), ol->length - sizeof(struct
rte_udp_hdr));
  rte_ring_mp_enqueue(host->rcvbuffer, ol);//push into the udp server recv buffer
```

```
  //wake the thread here after enqueue some elements in the queue
  pthread_mutex_lock(&host->mutex);
  pthread_cond_signal(&host->cond);
  pthread_mutex_unlock(&host->mutex);

  rte_pktmbuf_free(udpmbuf);
  return 0;

}

//offload--->mbufs
static int udp_out(struct rte_mempool *mbuf_pool){

  struct localhost *host;
  for (host = lhost; host != NULL; host = host->next) {

    struct offload *ol;
    int nb_snd = rte_ring_mc_dequeue(host->sndbuffer, (void **)&ol);
    if (nb_snd < 0) continue;

    // struct in_addr addr;
    // addr.s_addr = ol->dip;
    // printf("udp_out ---> src: %s:%d \n", inet_ntoa(addr), ntohs(ol->dport));

    uint8_t *dstmac = ng_get_dst_macaddr(ol->dip);
    if (dstmac == NULL) {//if there is no mac adress in arp table, send an arp
request

      struct rte_mbuf *arpbuf = arp_send(mbuf_pool, RTE_ARP_OP_REQUEST,
gDefaultArpMac, ol->sip, ol->dip);

      struct inout_ring *ring = ringInstance();
      rte_ring_mp_enqueue_burst(ring->outring, (void **)&arpbuf, 1, NULL);//put the
arp msg into the out ring buffer

      rte_ring_mp_enqueue(host->sndbuffer, ol);//here put the msg back to the snd
buffer because we must send the arp msg first and get the mac adress

    } else {
      //package an udp pkt
      struct rte_mbuf *udpbuf = ng_udp_pkt(mbuf_pool, ol->sip, ol->dip, ol->sport,
ol->dport, host->localmac, dstmac, ol->data, ol->length);
      struct inout_ring *ring = ringInstance();
```

```
      rte_ring_mp_enqueue_burst(ring->outring, (void **)&udpbuf, 1, NULL);//put the
udp pkt into the out ring buffer

    }

  }

  return 0;

}


#endif

static int pkt_process(void *arg) {

  struct rte_mempool *mbuf_pool = (struct rte_mempool *)arg;
  struct inout_ring *ring=ringInstance();
  while(1){

    struct rte_mbuf *mbufs[BURST_SIZE];
    unsigned num_recvd = rte_ring_mc_dequeue_burst(ring->inring, (void**)mbufs,
BURST_SIZE, NULL);//pop the msg in the ring buffer


    //analyze the pkt
    unsigned i=0;
    for(i=0;i<num_recvd;i++){
      struct rte_ether_hdr *ehdr=rte_pktmbuf_mtod(mbufs[i],struct
rte_ether_hdr*);//analyze the ethernet header

#if ENABLE_ARP
      if(ehdr->ether_type == rte_cpu_to_be_16(RTE_ETHER_TYPE_ARP)){
        struct rte_arp_hdr *ahdr=rte_pktmbuf_mtod_offset(mbufs[i],struct rte_arp_hdr
*,sizeof(struct rte_ether_hdr));//offset the header of ethernet

        // struct in_addr addr;
        // addr.s_addr=gLocalIp;
        // printf("local----->: %s\n, ",inet_ntoa(addr));

        if(ahdr->arp_data.arp_tip==gLocalIp){//only response to the host ip

          if(ahdr->arp_opcode == rte_cpu_to_be_16(RTE_ARP_OP_REQUEST)){//deal with
arp request msg
            struct in_addr addr;
```

```
            addr.s_addr=ahdr->arp_data.arp_tip;
            printf("arp---request>: %s\n ",inet_ntoa(addr));

            struct rte_mbuf *arpbuf= arp_send(mbuf_pool, RTE_ARP_OP_REPLY,
ahdr->arp_data.arp_sha.addr_bytes, ahdr->arp_data.arp_tip, ahdr->arp_data.arp_sip);
            rte_ring_mp_enqueue_burst(ring->outring, (void**)&arpbuf, 1, NULL);
            rte_pktmbuf_free(arpbuf);

          }else if(ahdr->arp_opcode == rte_cpu_to_be_16(RTE_ARP_OP_REPLY)){//receive
arp repley msg, put the adress into the arp table

            printf("arp --> reply\n");

            struct arp_table *table = arp_table_instance();
            uint8_t *hwaddr = ng_get_dst_macaddr(ahdr->arp_data.arp_sip);
            if (hwaddr == NULL) {

              struct arp_entry *entry = rte_malloc("arp_entry",sizeof(struct
arp_entry), 0);
              if (entry) {
                memset(entry, 0, sizeof(struct arp_entry));

                entry->ip = ahdr->arp_data.arp_sip;
                rte_memcpy(entry->hwaddr, ahdr->arp_data.arp_sha.addr_bytes,
RTE_ETHER_ADDR_LEN);
                entry->type = 0;

                LL_ADD(entry, table->entries);
                table->count ++;
              }

            }


#if ENABLE_DEBUG
            struct arp_entry *iter;
            for (iter = table->entries; iter != NULL; iter = iter->next) {
              struct in_addr addr;
              addr.s_addr = iter->ip;
              print_ethaddr("arp table --> mac: ", (struct rte_ether_addr
*)iter->hwaddr);
              printf(" ip: %s \n", inet_ntoa(addr));

            }
```

```
#endif
            rte_pktmbuf_free(mbufs[i]);

          }// end if it is request or repley arp msg

        }//end if if(ahdr->arp_data.arp_tip=...

        continue;
      }//end if(ehdr->ether_type....
#endif

//udp
      if(ehdr->ether_type != rte_cpu_to_be_16(RTE_ETHER_TYPE_IPV4)){
        rte_pktmbuf_free(mbufs[i]);
        continue;//if not ipv4,then skip
      }

      struct rte_ipv4_hdr *iphdr=rte_pktmbuf_mtod_offset(mbufs[i],struct
rte_ipv4_hdr *,sizeof(struct rte_ether_hdr));//get the ip hdr

      //udp
      if(iphdr->next_proto_id==IPPROTO_UDP){

        udp_process(mbufs[i]);

      }//udp if end

#if ENABLE_ICMP//icmp is on the same layer of udp

      if(iphdr->next_proto_id==IPPROTO_ICMP){//icmp
        struct rte_icmp_hdr *icmphdr=(struct rte_icmp_hdr *)(iphdr + 1);
        if(icmphdr->icmp_type==RTE_IP_ICMP_ECHO_REQUEST){
          //print the msg:
          printf("Get the icmp pkt:\n");
          struct in_addr addr;
          addr.s_addr=iphdr->src_addr;
          printf("icmp:src----->: %s ",inet_ntoa(addr));
          addr.s_addr=iphdr->dst_addr;
          printf("icmp:dst----->: %s\n",inet_ntoa(addr));

          struct rte_mbuf *txbuf=ng_send_icmp(mbuf_pool, ehdr->s_addr.addr_bytes,
iphdr->dst_addr, iphdr->src_addr, icmphdr->icmp_ident, icmphdr->icmp_seq_nb);
          rte_ring_mp_enqueue_burst(ring->outring, (void**)&txbuf, 1, NULL);
```

```
          rte_pktmbuf_free(mbufs[i]);
        }

      }//icmp if end

#endif
    }// for end

#if ENABLE_UDP_APP

  udp_out(mbuf_pool);

#endif


  }//while 1 end

  return 0;

}

#endif


#if ENABLE_UDP_APP



static struct localhost * get_hostinfo_fromfd(int sockfd){

  struct localhost *host;
  for(host=lhost;host!=NULL;host=host->next){
    if(sockfd==host->fd){
      return host;//bind the local var host to one element of the global lhost
    }
  }

  return NULL;

}
```

```
static int nico_socket(__attribute__((unused)) int domain, int type,
__attribute__((unused)) int protocol){
  //distribute a fd and link a type to the socket

  int fd=get_fd_frombitmap();
  struct localhost *host = rte_malloc("localhost", sizeof (struct localhost),0);
  if(host == NULL){
    return -1;
  }

  memset(host, 0, sizeof(struct localhost));

  host->fd=fd;
  if(type==SOCK_DGRAM){
    host->protocol=IPPROTO_UDP;
  }
  // else if(type==SOCK_STGREAM){
  //  host->protocol=IPPROTO_TCP;
  // }

  //create receive buffer
  host->rcvbuffer=rte_ring_create("recv buffer",
RING_SIZE,rte_socket_id(),RING_F_SP_ENQ | RING_F_SC_DEQ);
  if(host->rcvbuffer==NULL){
    rte_free(host);
    return -1;
  }

  //create send buffer
  host->sndbuffer=rte_ring_create("send buffer",
RING_SIZE,rte_socket_id(),RING_F_SP_ENQ | RING_F_SC_DEQ);
  if(host->sndbuffer==NULL){
    rte_ring_free(host->rcvbuffer);
    rte_free(host);
    return -1;
  }

  pthread_cond_t blank_cond=PTHREAD_COND_INITIALIZER;
  rte_memcpy(&host->cond, &blank_cond, sizeof(pthread_cond_t));

  pthread_mutex_t blank_mutex=PTHREAD_MUTEX_INITIALIZER;
  rte_memcpy(&host->cond, &blank_mutex, sizeof(pthread_mutex_t));


  LL_ADD(host, lhost);
```

```
  return fd;

}

static int nico_bind(int sockfd, const struct sockaddr *addr,
__attribute__((unused)) socklen_t addrlen){
  //bind the port, ipaddres and macadress
  struct localhost *host = get_hostinfo_fromfd(sockfd);//find the host through the
fd
  if(host == NULL)return -1;

  //bind here
  const struct sockaddr_in *laddr = (const struct sockaddr_in *)addr;
  host->localport = laddr->sin_port;
  rte_memcpy(&host->localip, &laddr->sin_addr.s_addr, sizeof(uint32_t));
  rte_memcpy(host->localmac, gSrcMac, RTE_ETHER_ADDR_LEN);

  return 0;
}

static ssize_t nico_recvfrom(int sockfd, void *buf, size_t len,
__attribute__((unused)) int flags, struct sockaddr *src_addr,
__attribute__((unused)) socklen_t *addrlen){
  struct localhost *host=get_hostinfo_fromfd(sockfd);
  if(host==NULL)return -1;

  unsigned char *ptr=NULL;
  struct offload *ol=NULL;
  int nb=-1;

  //set mutex in the thread
  pthread_mutex_lock(&host->mutex);
  while((nb = rte_ring_mc_dequeue(host->rcvbuffer,(void **)&ol))<0){//take the
offload from the recvbuffer
    pthread_cond_wait(&host->cond, &host->mutex);
  }
  pthread_mutex_unlock(&host->mutex);

  struct sockaddr_in *saddr = (struct sockaddr_in *)src_addr;
  saddr->sin_port = ol->sport;
  rte_memcpy(&saddr->sin_addr.s_addr, &ol->sip, sizeof(uint32_t));

  if(len<ol->length){
```

```
    rte_memcpy(buf, ol->data, len);
    //malloc a new place to the data that we haven't take
    ptr=rte_malloc("unsigned char", ol->length, 0);
    rte_memcpy(ptr, ol->data+len, ol->length-len);
    ol->length-=len;
    rte_free(ol->data);
    ol->data=ptr;

    //enqueue the data that exceed the len:
    rte_ring_mp_enqueue(host->rcvbuffer, ol);
    return len;
  }else{

    rte_memcpy(buf, ol->data, ol->length);
    //printf("into here: ol->data------>%s\n",ol->data);
    rte_free(ol->data);
    rte_free(ol);
    return ol->length;
  }

}

static ssize_t nico_sendto(int sockfd, const void *buf, size_t len,
__attribute__((unused)) int flags, const struct sockaddr *dest_addr,
__attribute__((unused)) socklen_t addrlen){


  struct localhost *host=get_hostinfo_fromfd(sockfd);
  if(host==NULL)return -1;

  const struct sockaddr_in *daddr = (const struct sockaddr_in *)dest_addr;

  struct offload *ol=rte_malloc("offload", sizeof(struct offload), 0);
  if(ol==NULL) return -1;

  ol->dip=daddr->sin_addr.s_addr;
  ol->dport=daddr->sin_port;
  ol->sip=host->localip;
  ol->sport=host->localport;
  ol->length=len;
  ol->data=rte_malloc("unsigned char*", len, 0);
  if(ol->data==NULL){
    rte_free(ol);
    return -1;
  }
```

```
  rte_memcpy(ol->data, buf, len);

  rte_ring_mp_enqueue(host->sndbuffer, ol);

  return len;



}


static int nico_close(int fd){

  struct localhost *host = get_hostinfo_fromfd(fd);//find the fd through host
  if(host == NULL)return -1;

  LL_REMOVE(host, lhost);
  if(host->rcvbuffer){
    rte_ring_free(host->rcvbuffer);
  }
  if(host->sndbuffer){
    rte_ring_free(host->sndbuffer);
  }
  rte_free(host);

}


//create a udp server:
static int udp_server_entry(__attribute__((unused))  void *arg) {

  //create a socket
  int connfd = nico_socket(AF_INET, SOCK_DGRAM, 0);
  if (connfd == -1) {
    printf("sockfd failed\n");
    return -1;
  }

  struct sockaddr_in localaddr, clientaddr; // struct sockaddr
  memset(&localaddr, 0, sizeof(struct sockaddr_in));

  //bind port and ip adress
  localaddr.sin_port = htons(8888);
  localaddr.sin_family = AF_INET;
```

```
  localaddr.sin_addr.s_addr = inet_addr("129.104.95.11");
  nico_bind(connfd, (struct sockaddr*)&localaddr, sizeof(localaddr));

  char buffer[UDP_APP_RECV_BUFFER_SIZE] = {0};
  socklen_t addrlen = sizeof(clientaddr);
  while (1) {

    if (nico_recvfrom(connfd, buffer, UDP_APP_RECV_BUFFER_SIZE, 0, (struct
sockaddr*)&clientaddr, &addrlen) < 0) {

      continue;

    } else {
      printf("recv from %s:%d, data:%s\n", inet_ntoa(clientaddr.sin_addr),
ntohs(clientaddr.sin_port), buffer);
      nico_sendto(connfd, buffer, strlen(buffer), 0, (struct sockaddr*)&clientaddr,
sizeof(clientaddr));
    }
  }
  nico_close(connfd);

}


#endif



int main(int argc, char*argv[]){
  if(rte_eal_init(argc,argv)<0){
    rte_exit(EXIT_FAILURE,"Error with EAL init\n");
  }

  struct rte_mempool *mbuf_pool=rte_pktmbuf_pool_create("mbuf
pool",NUM_MBUFS,0,0,RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
  if(mbuf_pool==NULL){
    rte_exit(EXIT_FAILURE,"Cloud not create mbuf\n");
  }
  ng_init_port(mbuf_pool);//initialize the port(eth0)

  rte_eth_macaddr_get(gDpdkPortId,(struct rte_ether_addr *)gSrcMac);


//setup a timer:
#if ENABLE_TIMER
```

```
  rte_timer_subsystem_init();

  struct rte_timer arp_timer;
  rte_timer_init(&arp_timer);

  uint64_t hz = rte_get_timer_hz();
  unsigned lcore_id = rte_lcore_id();
  rte_timer_reset(&arp_timer, hz, PERIODICAL, lcore_id, arp_request_timer_cb,
mbuf_pool);//PERIODICAL: multiply trigger the timer

#endif


#if ENABLE_RINGBUFFER
  struct inout_ring *ring=ringInstance();
  if(ring==NULL){rte_exit(EXIT_FAILURE,"ring init fail\n");}

  if(ring->inring==NULL){
    ring->inring=rte_ring_create("in ring", RING_SIZE,rte_socket_id(),RING_F_SP_ENQ
| RING_F_SC_DEQ);
  }

  if(ring->outring==NULL){
    ring->outring=rte_ring_create("out ring",
RING_SIZE,rte_socket_id(),RING_F_SP_ENQ | RING_F_SC_DEQ);
  }

#endif

#if ENABLE_MULTITHREAD

  //启动线程，和 cpu 粘合的
  lcore_id = rte_get_next_lcore(lcore_id, 1, 0);
  rte_eal_remote_launch(pkt_process, mbuf_pool, lcore_id);

#endif


#if ENABLE_UDP_APP

  //启动线程，和 cpu 粘合的
  lcore_id = rte_get_next_lcore(lcore_id, 1, 0);//bind another core
  rte_eal_remote_launch(udp_server_entry, mbuf_pool, lcore_id);
```

```
#endif

//up to here, establish 3 thread, which are: main thread, thread for the protocol
analyze and thread for udp server

  while(1){

    //rx
    //Parse the application layer package
    struct rte_mbuf *rx[32];
    unsigned num_recvd=rte_eth_rx_burst(gDpdkPortId, 0, rx, 32);//receive msg
    if(num_recvd>32){
      rte_exit(EXIT_FAILURE,"Cloud not create mbuf\n");
    }
    else if(num_recvd>0){
      rte_ring_sp_enqueue_burst(ring->inring, (void**)rx, num_recvd, NULL);//receive
msg and push into the ring buffer
    }


    //tx
    struct rte_mbuf *tx[32];
    unsigned nb_tx = rte_ring_sc_dequeue_burst(ring->outring, (void**)tx,
BURST_SIZE, NULL);//dequeue the msg and send through the network card
    if(nb_tx > 0){
      rte_eth_tx_burst(gDpdkPortId, 0, tx, nb_tx);//send out
      //release the memory
      unsigned i=0;
      for(i=0;i<nb_tx;i++){
        rte_pktmbuf_free(tx[i]);
      }
    }

#if ENABLE_TIMER

    static uint64_t prev_tsc = 0, cur_tsc;
    uint64_t diff_tsc;

    cur_tsc = rte_rdtsc();
    diff_tsc = cur_tsc - prev_tsc;
    if (diff_tsc > TIMER_RESOLUTION_CYCLES) {
      rte_timer_manage();
      prev_tsc = cur_tsc;
    }
```

```
#endif

  }//while 1 end

}//main end
```

7.2 arp.h:

```
//only let this .h file define once
#ifndef __NG_ARP_H__//if not define
#define __NG_ARP_H__

#include <rte_ether.h>
#include <rte_malloc.h>


#define ARP_ENTRY_STATUS_DYNAMIC  0
#define ARP_ENTRY_STATUS_STATIC   1


#define LL_ADD(item, list) do {   \
  item->prev = NULL;        \
  item->next = list;        \
  if (list != NULL) list->prev = item; \
  list = item;          \
} while(0)


#define LL_REMOVE(item, list) do {    \
  if (item->prev != NULL) item->prev->next = item->next;  \
  if (item->next != NULL) item->next->prev = item->prev;  \
  if (list == item) list = item->next;  \
  item->prev = item->next = NULL;     \
} while(0)

/*the arp table is like this:

Internet adress         mac adress       type
127.0.0.1               xx:xx:xx:xx      dynamic

so next define ip adress, mac adress and type

*/

//linked list
struct arp_entry {

  uint32_t ip;
  uint8_t hwaddr[RTE_ETHER_ADDR_LEN];
  uint8_t type;
```

```
  // the size is not matching

  struct arp_entry *next;
  struct arp_entry *prev;

};

struct arp_table {

  struct arp_entry *entries;
  int count;

};



static struct  arp_table *arpt = NULL;

static struct  arp_table *arp_table_instance(void) {

  if (arpt == NULL) {

    arpt = rte_malloc("arp table", sizeof(struct  arp_table), 0);
    if (arpt == NULL) {
      rte_exit(EXIT_FAILURE, "rte_malloc arp table failed\n");
    }
    memset(arpt, 0, sizeof(struct  arp_table));
  }

  return arpt;

}

//find the mac add
static uint8_t* ng_get_dst_macaddr(uint32_t dip) {

  struct arp_entry *iter;
  struct arp_table *table = arp_table_instance();

  for (iter = table->entries;iter != NULL;iter = iter->next) {
    if (dip == iter->ip) {
      return iter->hwaddr;
    }
  }
```

```
  return NULL;
}

#endif
```