中国科学院大学

实验报告

静态路由转发实验

课程名称： 计算机网络

学院： 计算机学院

专业： 计算机体系结构

姓名： 卞留念

卫一宁

学号： 201828013229131

2018E8013261055

2019 年 6 月 5 日

中国科学院大学实验报告

专业：计算机体系结构
姓名：卞留念
卫一宁
学号：201828013229131
2018E8013261055
日期：2019 年 6 月 5 日

课程名称：计算机网络　　实验名称：静态路由转发实验　　指导老师：谢高岗

一、　实验要求

本实验要求学生在已有代码基础上，完善其中的 TODO 部分，实现路由器的 IP 查找转发、ARP 请求和应答、ARP 缓存管理、发送 ICMP 消息等功能。

二、　实验内容和步骤

1.　实验内容一

(1) 运行给定网络拓扑 (router_topo.py)

(2) Ping 10.0.1.1 (r1)，能够 ping 通

(3) Ping 10.0.2.22 (h2)，能够 ping 通

(4) Ping 10.0.3.33 (h3)，能够 ping 通

(5) Ping 10.0.3.11，返回 ICMP Destination Host Unreachable

(6) Ping 10.0.4.1，返回 ICMP Destination Net Unreachable

2.　实验内容二

(1) 构造一个包含多个路由器节点组成的网络，手动配置每个路由器节点的路由表，有两个终端节点，通过路由器节点相连，两节点之间的跳数不少于 3 跳，手动配置其默认路由表

(2) 终端节点 ping 每个路由器节点的入端口 IP 地址，能够 ping 通

(3) 在一个终端节点上 traceroute 另一节点，能够正确输出路径上每个节点（入端口）的 IP 信息

三、　主要仪器设备

计算机，Mininet 软件，Wireshark 软件

四、　实验步骤

1.　安装 mininet 与 wireshark 软件

下载并安装 mininet 与 wireshark 等软件。

2. 编写代码 TODO 部分

根据所学知识完成代码的 TODO 部分。

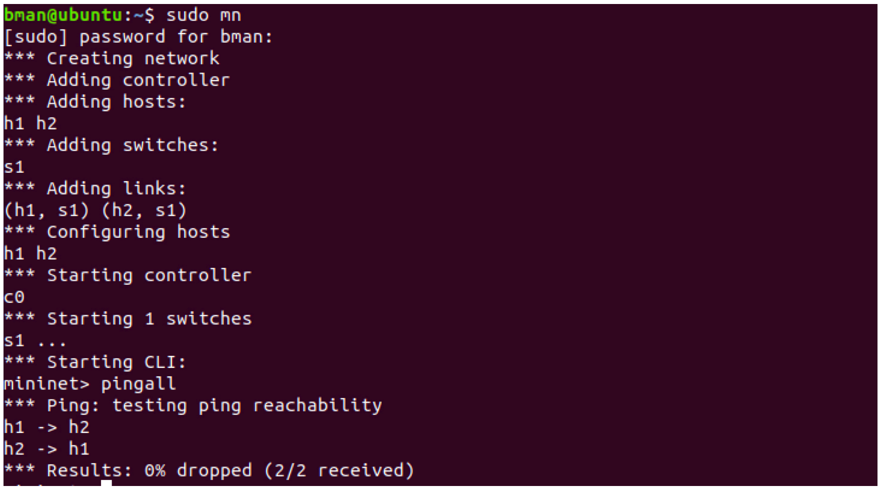
3. 完成实验内容

编译代码生成程序，在路由器节点上运行此程序，依次完成实验内容。

五、 实验过程

1. 安装 mininet 与 wireshark 软件

在 Ubuntu 下输入 sudo apt install mininet 与 sudo apt install build-essential xterm wireshark ethtool iperf traceroute iptables arptables 命令进行软件安装，安装完成后，运行 sudo mn 验证 mininet 是否正确安装，验证结果如图 1 所示。



```
bman@ubuntu:~$ sudo mn
[sudo] password for bman:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

图 1: mininet 安装成功

2. 编写 arpcache_lookup 函数

```c
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN]) {
    // traverse arp table
    int i = 0;
    int flag = 0;
    for (i = 0; i < MAX_ARP_SIZE; i += 1) {
        if(arpcache.entries[i].ip4 == ip4 && arpcache.entries[i].valid == 1) {
            flag = 1;
            memcpy(mac, arpcache.entries[i].mac, sizeof(u8) * ETH_ALEN);
        }
    }
    return flag;
}
```

3. 编写 arpcache_append_packet 函数

```
void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len)
{
    // append a packet into arp cache packet list
    // if there has been one list with given ipv4 addr, insert packet into that list
    // if not, create a list

    int flag = 0;
    struct arp_req *ele = NULL;
    list_for_each_entry(ele, &arpcache.req_list, list) {
        if (ele->ip4 == ip4) {
            flag = 1;
            struct cached_pkt *new_pkt = (struct cached_pkt *)malloc(sizeof(struct
                cached_pkt));
            new_pkt->packet = (char *)malloc((size_t)len);
            memcpy(new_pkt->packet, packet, len);
            new_pkt->len = len;
            init_list_head(&new_pkt->list);
            list_add_tail(&new_pkt->list, &ele->cached_packets);
        }
    }

    // if not found
    if(flag == 0) {
        // firstly, create cache object
        struct arp_req *new_req = (struct arp_req *)malloc(sizeof(struct arp_req));
        // init value
        new_req->ip4 = ip4;
        new_req->retries = 0;
        new_req->iface = iface;
        init_list_head(&new_req->cached_packets);
        init_list_head(&new_req->list);
        list_add_tail(&new_req->list, &arpcache.req_list);
        // create packet object and insert it into list
        struct cached_pkt *new_pkt = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
        new_pkt->packet = (char *)malloc((size_t)len);
        memcpy(new_pkt->packet, packet, len);
        new_pkt->len = len;
        init_list_head(&new_pkt->list);
        list_add_tail(&new_pkt->list, &new_req->cached_packets);

        arp_send_request(iface, ip4);
    }
}
```

4. 编写 arpcache_insert 函数

```
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
{
    // check is there has been one item with given ipv4 addr, if yes, replace it
    int i = 0;
    int flag = 0;
    for (i = 0; i < MAX_ARP_SIZE; i += 1) {
        if(arpcache.entries[i].ip4 == ip4) {
            flag = 1;
            memcpy(arpcache.entries[i].mac, mac, ETH_ALEN);
            arpcache.entries[i].valid = 1;
            time(&arpcache.entries[i].added);
        }
    }

    if(flag == 0) {
        // if no, create one, put it in the first place, FIFO
        struct arp_cache_entry *entry = (struct arp_cache_entry *)malloc(sizeof(struct
             arp_cache_entry));
        entry->ip4 = ip4;
        memcpy(entry->mac, mac, ETH_ALEN);
        time(&entry->added);
        entry->valid = 1;
        for (i = MAX_ARP_SIZE - 1; i >= 0; i -= 1) {
            arpcache.entries[i] = arpcache.entries[i - 1];
        }
        arpcache.entries[0] = *entry;
    }

    struct arp_req *req = NULL;
    struct cached_pkt *pkt = NULL;
    list_for_each_entry(req, &arpcache.req_list, list) {
        if(req->ip4 == ip4) {
            pkt = NULL;
            list_for_each_entry(pkt, &req->cached_packets, list) {
                struct ether_header * eh = (struct ether_header *)(pkt->packet);
                memcpy(eh->ether_dhost, mac, ETH_ALEN);
                iface_send_packet(req->iface, pkt->packet, pkt->len);
                list_delete_entry(&pkt->list);
            }
            list_delete_entry(&req->list);
        }
    }
}
```

5. 编写 arpcache_sweep 函数

```c
void *arpcache_sweep(void *arg)
{
    int i = 0;
    time_t now;
    while (1) {
        sleep(1);
        // traverse arp cache, delete item which has been staying over 15s
        for(i = 0; i < MAX_ARP_SIZE; i += 1) {
            time(&now);
            if((long)now - (long)arpcache.entries[i].added > 15) {
                arpcache.entries[i].valid = 0;
            }
        }

        // traverse packet list, if waiting time is more than 1s, re-send it
        // if retries over 5 times, send icmp packet
        struct arp_req *req = NULL, *req_q;
        struct cached_pkt *pkt = NULL, *pkt_q;
        list_for_each_entry_safe(req, req_q, &(arpcache.req_list), list) {
            if(req->retries >= 5) {
                pkt = NULL;
                list_for_each_entry_safe(pkt, pkt_q, &(req->cached_packets), list) {
                    printf("arp request failed, send icmp packet\n");
                    icmp_send_packet(pkt->packet, pkt->len, 3, 1);
                    list_delete_entry(&(pkt->list));
                }
                list_delete_entry(&(req->list));
            } else {
                time(&now);
                if((long)now - (long)req->sent >= 1) {
                    printf("arp request retry one more time\n");
                    req->retries += 1;
                    arp_send_request(req->iface, req->ip4);
                } else {
                    printf("now is %ld, req->sent is %ld \n", (long)now, (long)req->sent);
                }
            }
        }
    }

    return NULL;
}
```

6. 编写 arp_send_request 函数

```c
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    // encapsulate an arp reply and send it out
    char *packet = (char *)malloc(sizeof(struct ether_header) + sizeof(struct ether_arp));
    bzero(packet, sizeof(struct ether_header) + sizeof(struct ether_arp));

    struct ether_header *eh = (struct ether_header *)packet;
    struct ether_arp *ea = packet_to_arp_hdr(packet);

    // set value
    // ether
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    int i;
    for(i = 0; i < ETH_ALEN; i += 1) {
        eh->ether_dhost[i] = 0xFF;
    }
    eh->ether_type = htons(ETH_P_ARP);

    // arp
    ea->arp_hrd = htons(1);
    ea->arp_pro = htons(0x0800);
    ea->arp_hln = ETH_ALEN;
    ea->arp_pln = 4;
    ea->arp_op = htons(ARPOP_REQUEST);

    ea->arp_spa = htonl(iface->ip);
    memcpy(ea->arp_sha, iface->mac, ETH_ALEN);
    ea->arp_tpa = htonl(dst_ip);
    // tha  00 00 00 00 00 00
    printf("arp request for target ip address: %x is sending \n", dst_ip);

    // send it out
    iface_send_packet(iface, packet, (int)(sizeof(struct ether_header) + sizeof(struct
        ether_arp)));
}
```

7. 编写 arp_send_reply 函数

```
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)
{
    // encapsulate an arp request packet and send it out
    // when response one packet, it's spa and sha is arp request result
    char *packet = (char *)malloc(sizeof(struct ether_header) + sizeof(struct ether_arp));
    bzero(packet, sizeof(struct ether_header) + sizeof(struct ether_arp));

    struct ether_header *eh = (struct ether_header *)packet;
    struct ether_arp *ea = packet_to_arp_hdr(packet);

    // set value
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    memcpy(eh->ether_dhost, req_hdr->arp_sha, ETH_ALEN);
    eh->ether_type = htons(ETH_P_ARP);

    ea->arp_hrd = htons(1);
    ea->arp_pro = htons(0x0800);
    ea->arp_hln = ETH_ALEN;
    ea->arp_pln = 4;
    ea->arp_op = htons(ARPOP_REPLY);

    ea->arp_tpa = htonl(req_hdr->arp_spa);
    memcpy(ea->arp_tha, req_hdr->arp_sha, ETH_ALEN);
    ea->arp_spa = htonl(req_hdr->arp_tpa);

    // firstly, check is this host's ip
    // if it is, fill the ether and send it out
    // if not, find it in arpcache
    if(req_hdr->arp_tpa == iface->ip) {
        memcpy(ea->arp_sha, iface->mac, ETH_ALEN);
        iface_send_packet(iface, packet, (int)(sizeof(struct ether_header) + sizeof(struct
            ether_arp)));
    } else {
        int found = arpcache_lookup(req_hdr->arp_tpa, req_hdr->arp_tha);
        if(found) {
            iface_send_packet(iface, packet, (int)(sizeof(struct ether_header) +
                sizeof(struct ether_arp)));
        } else {
            free(packet);
        }
    }
}
```

8. 编写 handle_arp_packet 函数

```c
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_arp * ea = packet_to_arp_hdr(packet);
    // change byte order
    ea->arp_op = ntohs(ea->arp_op);
    ea->arp_tpa = ntohl(ea->arp_tpa);
    ea->arp_spa = ntohl(ea->arp_spa);
    // put the source information into arp table
    arpcache_insert(ea->arp_spa, ea->arp_sha);
    if(ea->arp_op == ARPOP_REQUEST) {
        // give arp reply
        arp_send_reply(iface, ea);
    }
    free(packet);
}
```

9. 编写 iface_send_packet_by_arp 函数

```c
void iface_send_packet_by_arp(iface_info_t *iface, u32 dst_ip, char *packet, int len)
{
    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    eh->ether_type = htons(ETH_P_IP);

    u8 dst_mac[ETH_ALEN];
    int found = arpcache_lookup(dst_ip, dst_mac);
    if (found) {
        memcpy(eh->ether_dhost, dst_mac, ETH_ALEN);
        iface_send_packet(iface, packet, len);
    } else {
        arpcache_append_packet(iface, dst_ip, packet, len);
    }
}
```

10. 编写 longest_prefix_match 函数

```
rt_entry_t *longest_prefix_match(u32 dst)
{
    rt_entry_t * result = NULL;
    int max_mask = 0;
    rt_entry_t * ele = NULL;
    list_for_each_entry(ele, &rtable, list) {
        if((ele->dest & ele->mask) == (dst & ele->mask)) {
            if(ele->mask >= max_mask) {
                max_mask = ele->mask;
                result = ele;
            }
        }
    }
    return result;
}
```

11. 编写 handle_ip_packet 函数

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    struct iphdr *ip = packet_to_ip_hdr(packet);
    u32 daddr = ntohl(ip->daddr);
    if (daddr == iface->ip) {
        icmp_send_packet(packet, len, 0, 0);
        free(packet);
    }
    else {
        ip_forward_packet(daddr, packet, len);
    }
}
```

12. 编写 icmp_send_packet 函数

```
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    // get ip destination addr
    struct iphdr * pkt_ip_hdr = packet_to_ip_hdr(in_pkt);
    struct icmphdr * pkt_icmp_hdr = (struct icmphdr *)(in_pkt + ETHER_HDR_SIZE +
        pkt_ip_hdr->ihl * 4);

    rt_entry_t * rt = longest_prefix_match(ntohl(pkt_ip_hdr->saddr));
    if(rt) {
        size_t packet_length;
        char * packet;
        struct iphdr * packet_iphdr;
        struct icmphdr * packet_icmphdr;
        // judge type and process
        switch(type) {
            case 0:
                // malloc an icmp packet
                // ip header
                packet_length = (size_t)len;
                packet = (char *)malloc(packet_length);
                packet_iphdr = packet_to_ip_hdr(packet);
                ip_init_hdr(packet_iphdr, rt->iface->ip, ntohl(pkt_ip_hdr->saddr),
                    (u16)(packet_length - sizeof(struct ether_header)), 1);
                // icmp header
                packet_icmphdr = (struct icmphdr *)(packet + ETHER_HDR_SIZE +
                    packet_iphdr->ihl * 4);
                packet_icmphdr->code = code;
                packet_icmphdr->type = type;
                packet_icmphdr->icmp_identifier = pkt_icmp_hdr->icmp_identifier;
                packet_icmphdr->icmp_sequence = pkt_icmp_hdr->icmp_sequence;

                memcpy(
                        packet + ETHER_HDR_SIZE + packet_iphdr->ihl * 4 + sizeof(struct
                            icmphdr),
                        in_pkt + ETHER_HDR_SIZE + pkt_ip_hdr->ihl * 4 + sizeof(struct
                            icmphdr),
                        len - ETHER_HDR_SIZE - pkt_ip_hdr->ihl * 4 - sizeof(struct icmphdr)
                );

                packet_icmphdr->checksum = icmp_checksum(packet_icmphdr, (int)packet_length
                    - ETHER_HDR_SIZE - sizeof(struct iphdr));
                break;
            case 3:         // dst unreachable
                // malloc an icmp packet
                // ip header
                packet_length = (size_t)(ETHER_HDR_SIZE + sizeof(struct iphdr) +
                    sizeof(struct icmphdr) + sizeof(struct iphdr) + 8);
                packet = (char *)malloc(packet_length);
                packet_iphdr = packet_to_ip_hdr(packet);
                ip_init_hdr(packet_iphdr, rt->iface->ip, ntohl(pkt_ip_hdr->saddr),
                    (u16)(packet_length - sizeof(struct ether_header)), 1);
                // icmp header
                packet_icmphdr = (struct icmphdr *)(packet + ETHER_HDR_SIZE +
                    packet_iphdr->ihl * 4);
                packet_icmphdr->code = code;
                packet_icmphdr->type = type;

                memcpy(
                        packet + ETHER_HDR_SIZE + packet_iphdr->ihl * 4 + sizeof(struct
                            icmphdr),
                        in_pkt + ETHER_HDR_SIZE,
                        sizeof(struct iphdr) + 8
                );
                packet_icmphdr->checksum = icmp_checksum(packet_icmphdr, (int)packet_length
                    - ETHER_HDR_SIZE - sizeof(struct iphdr));
                break;
            case 8:          // icmp request
                // todo
                printf("this packet icmp type is 8, todo \n");
                break;
            case 11:      // ttl exceed
                // malloc an icmp packet
                // ip header
                packet_length = (size_t)(ETHER_HDR_SIZE + sizeof(struct iphdr) +
                    sizeof(struct icmphdr) + sizeof(struct iphdr) + 8);
                packet = (char *)malloc(packet_length);
                packet_iphdr = packet_to_ip_hdr(packet);
                ip_init_hdr(packet_iphdr, rt->iface->ip, ntohl(pkt_ip_hdr->saddr),
                    (u16)(packet_length - sizeof(struct ether_header)), 1);
                // icmp header
                packet_icmphdr = (struct icmphdr *)(packet + ETHER_HDR_SIZE +
                    packet_iphdr->ihl * 4);
                packet_icmphdr->code = code;
                packet_icmphdr->type = type;

                memcpy(
                        packet + ETHER_HDR_SIZE + packet_iphdr->ihl * 4 + sizeof(struct
                            icmphdr),
                        in_pkt + ETHER_HDR_SIZE,
                        sizeof(struct iphdr) + 8
                );
                packet_icmphdr->checksum = icmp_checksum(packet_icmphdr, (int)packet_length
                    - ETHER_HDR_SIZE - sizeof(struct iphdr));
                break;
            default:
                printf("unknown icmp type! \n");
                return;
                }

                iface_send_packet_by_arp(rt->iface, ntohl(pkt_ip_hdr->saddr), packet,
                    (int)packet_length);
            }
}
```

13. 编写 ip_forward_packet 函数

```
void ip_forward_packet(u32 ip_dst, char *packet, int len)
{
    // check TTL
    struct iphdr * pkt_ip_hdr = packet_to_ip_hdr(packet);
    // if ttl === 1, discard it
    if(pkt_ip_hdr->ttl - 1 <= 0) {
        icmp_send_packet(packet, len, 11, 0);
        free(packet);
        return;
    }

    // forward packet
    rt_entry_t * rt = longest_prefix_match(ip_dst);
    if(rt) {
        pkt_ip_hdr->ttl -= 1;
        pkt_ip_hdr->checksum = ip_checksum(pkt_ip_hdr);
        iface_send_packet_by_arp(rt->iface, rt->gw == 0 ? ntohl(pkt_ip_hdr->daddr) :
            rt->gw, packet, len);
    } else {
        // return ICMP Destination Net Unreachable
        icmp_send_packet(packet, len, 3, 0);
        free(packet);
        return;
    }
}
```

六、 实验结果与分析

1. 实验一

(1) 运行 router_topo.py 生成的网络如图 2 所示。

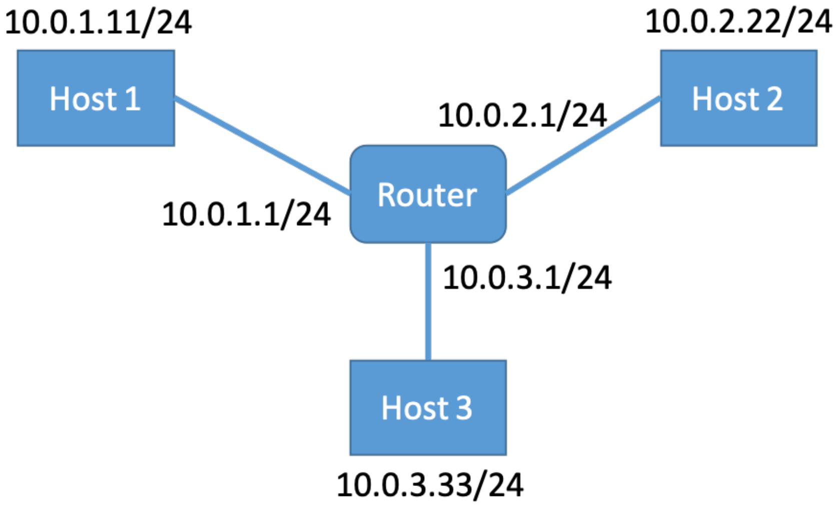
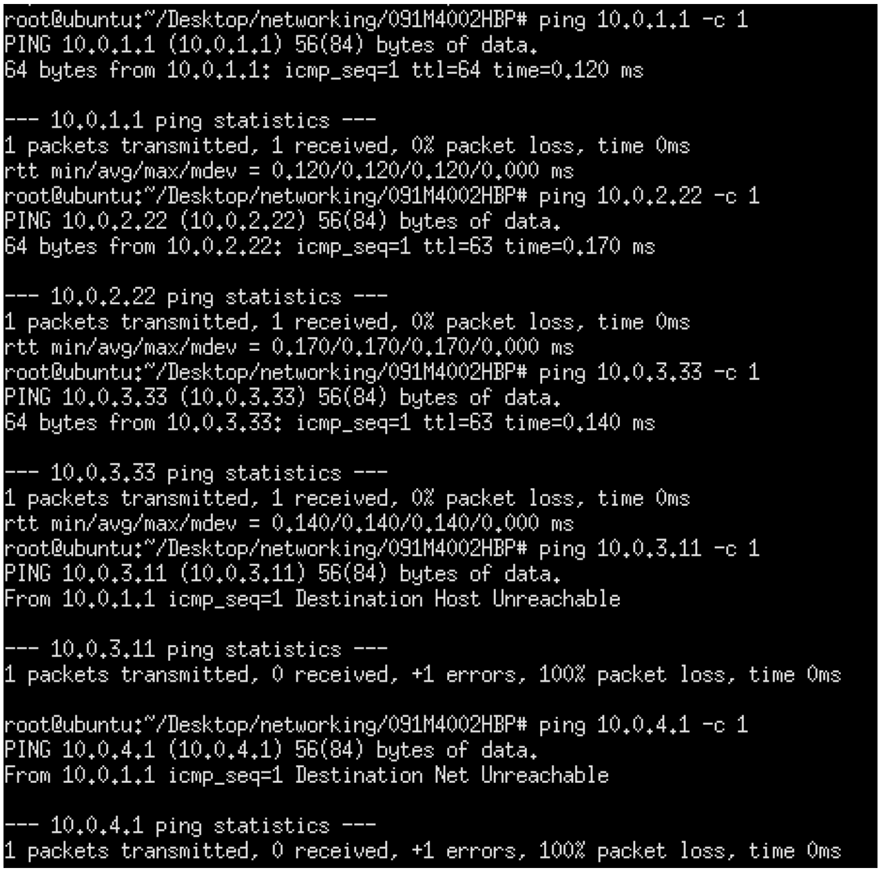


图 2: router_topo.py 生成的网络

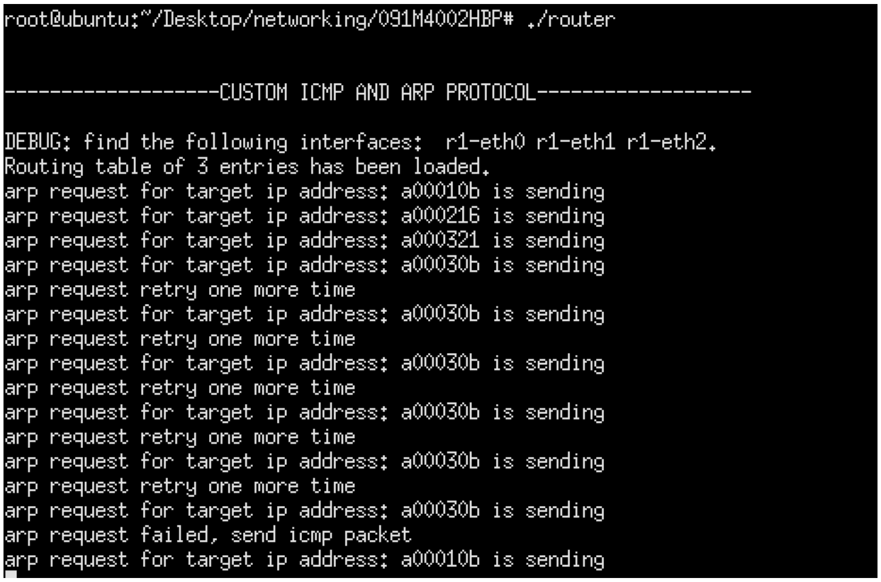
(2) 实验一 (2)-(6)ping 结果如图 3 所示。

(3) 实验一 (2)-(6) 路由程序运行结果如图 4所示。



```
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.1.1 -c 1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=0.120 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.120/0.120/0.120/0.000 ms
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.2.22 -c 1
PING 10.0.2.22 (10.0.2.22) 56(84) bytes of data.
64 bytes from 10.0.2.22: icmp_seq=1 ttl=63 time=0.170 ms

--- 10.0.2.22 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.170/0.170/0.170/0.000 ms
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.3.33 -c 1
PING 10.0.3.33 (10.0.3.33) 56(84) bytes of data.
64 bytes from 10.0.3.33: icmp_seq=1 ttl=63 time=0.140 ms

--- 10.0.3.33 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.140/0.140/0.140/0.000 ms
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.3.11 -c 1
PING 10.0.3.11 (10.0.3.11) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Host Unreachable

--- 10.0.3.11 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.4.1 -c 1
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Net Unreachable

--- 10.0.4.1 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
```

图 3: 实验一 (2)-(6)ping 程序结果



```
root@ubuntu:~/Desktop/networking/091M4002HBP# ./router


-------------------CUSTOM ICMP AND ARP PROTOCOL-------------------

DEBUG: find the following interfaces:  r1-eth0 r1-eth1 r1-eth2.
Routing table of 3 entries has been loaded.
arp request for target ip address: a00010b is sending
arp request for target ip address: a000216 is sending
arp request for target ip address: a000321 is sending
arp request for target ip address: a00030b is sending
arp request retry one more time
arp request for target ip address: a00030b is sending
arp request retry one more time
arp request for target ip address: a00030b is sending
arp request retry one more time
arp request for target ip address: a00030b is sending
arp request retry one more time
arp request for target ip address: a00030b is sending
arp request retry one more time
arp request for target ip address: a00030b is sending
arp request failed, send icmp packet
arp request for target ip address: a00010b is sending
```

图 4: 实验一 (2)-(6) 路由程序运行结果

2. 实验二

(1) 根据要求编写 router_topo2.py, 代码如下。

```
class RouterTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        r1 = self.addHost('r1')
        r2 = self.addHost('r2')
        r3 = self.addHost('r3')

        self.addLink(h1, r1)
        self.addLink(r1, r2)
        self.addLink(r2, r3)
        self.addLink(r3, h2)

if __name__ == '__main__':
    topo = RouterTopo()
    net = Mininet(topo = topo, controller = None)

    h1, h2, r1, r2, r3 = net.get('h1', 'h2', 'r1', 'r2', 'r3')
    h1.cmd('ifconfig h1-eth0 10.0.1.11/24')
    h2.cmd('ifconfig h2-eth0 10.0.4.4/24')

    h1.cmd('route add default gw 10.0.1.1')
    h2.cmd('route add default gw 10.0.4.1')

    for h in (h1, h2):
        h.cmd('./scripts/disable_offloading.sh')
        h.cmd('./scripts/disable_ipv6.sh')

    r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
    r1.cmd('ifconfig r1-eth1 10.0.2.2/24')

    r2.cmd('ifconfig r2-eth0 10.0.2.3/24')
    r2.cmd('ifconfig r2-eth1 10.0.3.3/24')

    r3.cmd('ifconfig r3-eth0 10.0.3.4/24')
    r3.cmd('ifconfig r3-eth1 10.0.4.4/24')

    r1.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.2.3 dev r1-eth1')
    r1.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.2.3 dev r1-eth1')

    r2.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.2.2 dev r2-eth0')
    r2.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.3.4 dev r2-eth1')

    r3.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.3.3 dev r3-eth0')
    r3.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.3.3 dev r3-eth0')
```

```
for r in (r1, r2, r3):
    r.cmd('./scripts/disable_arp.sh')
    r.cmd('./scripts/disable_icmp.sh')
    r.cmd('./scripts/disable_ip_forward.sh')

net.start()
CLI(net)
net.stop()
```

(2) router_topo2.py 生成的网络如图 5 所示。

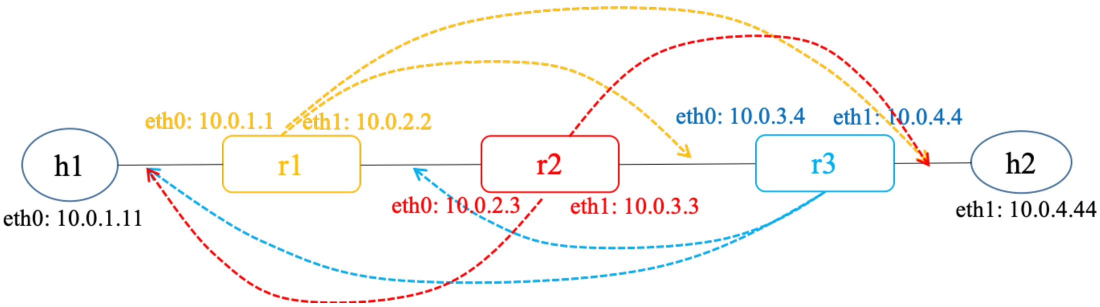
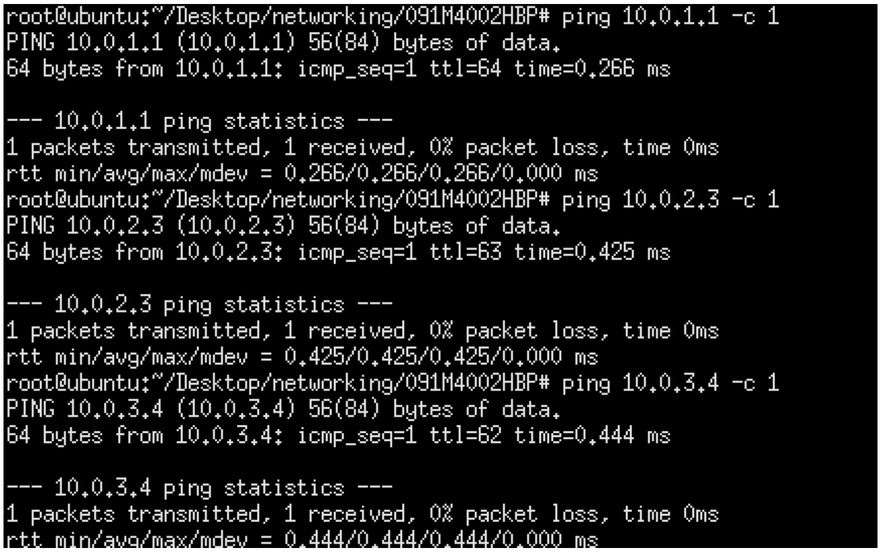


图 5: router_topo2.py 生成的网络

(3) 实验二 (2)ping 结果如图 6 所示。



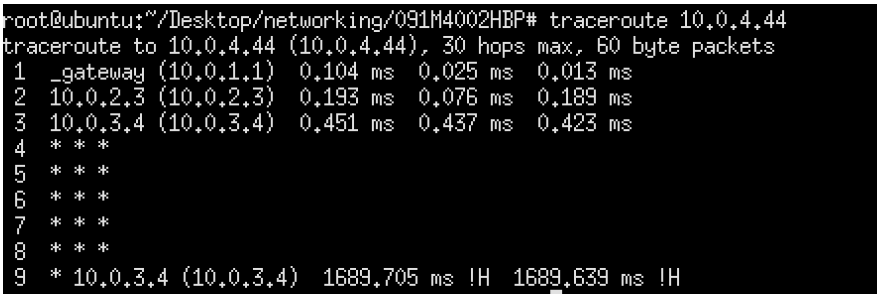
```
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.1.1 -c 1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=0.266 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.266/0.266/0.266/0.000 ms
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.2.3 -c 1
PING 10.0.2.3 (10.0.2.3) 56(84) bytes of data.
64 bytes from 10.0.2.3: icmp_seq=1 ttl=63 time=0.425 ms

--- 10.0.2.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.425/0.425/0.425/0.000 ms
root@ubuntu:~/Desktop/networking/091M4002HBP# ping 10.0.3.4 -c 1
PING 10.0.3.4 (10.0.3.4) 56(84) bytes of data.
64 bytes from 10.0.3.4: icmp_seq=1 ttl=62 time=0.444 ms

--- 10.0.3.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.444/0.444/0.444/0.000 ms
```

图 6: 实验二 (2)ping 程序结果

(4) 实验二 (2)traceroute 结果如图 7所示。

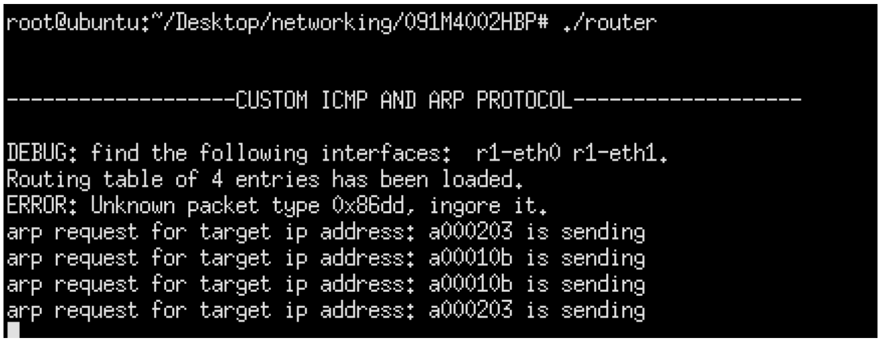
(5) 实验二 (2)-(3) 路由 1 程序运行结果如图 8所示。

(6) 实验二 (2)-(3) 路由 2 程序运行结果如图 9所示。



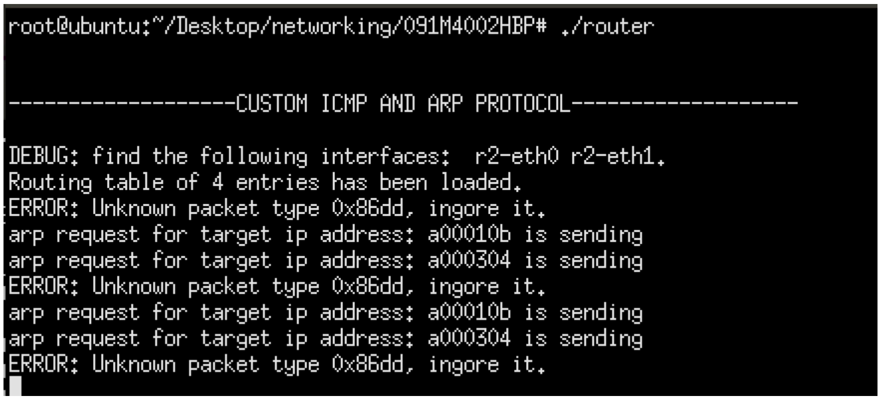
```
root@ubuntu:~/Desktop/networking/091M4002HBP# traceroute 10.0.4.44
traceroute to 10.0.4.44 (10.0.4.44), 30 hops max, 60 byte packets
 1  _gateway (10.0.1.1)  0.104 ms  0.025 ms  0.013 ms
 2  10.0.2.3 (10.0.2.3)  0.193 ms  0.076 ms  0.189 ms
 3  10.0.3.4 (10.0.3.4)  0.451 ms  0.437 ms  0.423 ms
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * 10.0.3.4 (10.0.3.4)  1689.705 ms !H  1689.639 ms !H
```

图 7: 实验二 (3)traceroute 程序结果



```
root@ubuntu:~/Desktop/networking/091M4002HBP# ./router


-------------------CUSTOM ICMP AND ARP PROTOCOL-------------------

DEBUG: find the following interfaces:  r1-eth0 r1-eth1.
Routing table of 4 entries has been loaded.
ERROR: Unknown packet type 0x86dd, ingore it.
arp request for target ip address: a000203 is sending
arp request for target ip address: a00010b is sending
arp request for target ip address: a00010b is sending
arp request for target ip address: a000203 is sending
```

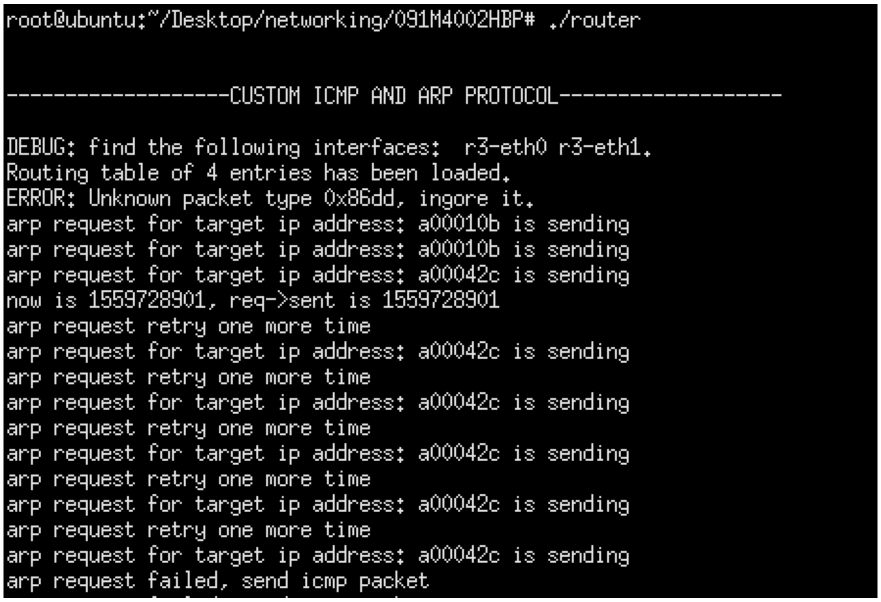
图 8: 实验二 (2)-(3) 路由 1 程序运行结果



```
root@ubuntu:~/Desktop/networking/091M4002HBP# ./router


-------------------CUSTOM ICMP AND ARP PROTOCOL-------------------

DEBUG: find the following interfaces:  r2-eth0 r2-eth1.
Routing table of 4 entries has been loaded.
ERROR: Unknown packet type 0x86dd, ingore it.
arp request for target ip address: a00010b is sending
arp request for target ip address: a000304 is sending
ERROR: Unknown packet type 0x86dd, ingore it.
arp request for target ip address: a00010b is sending
arp request for target ip address: a000304 is sending
ERROR: Unknown packet type 0x86dd, ingore it.
```

图 9: 实验二 (2)-(3) 路由 2 程序运行结果

(7) 实验二 (2)-(3) 路由 3 程序运行结果如图 10所示。



```
root@ubuntu:~/Desktop/networking/091M4002HBP# ./router


-------------------CUSTOM ICMP AND ARP PROTOCOL-------------------

DEBUG: find the following interfaces:  r3-eth0 r3-eth1.
Routing table of 4 entries has been loaded.
ERROR: Unknown packet type 0x86dd, ingore it.
arp request for target ip address: a00010b is sending
arp request for target ip address: a00010b is sending
arp request for target ip address: a00042c is sending
now is 1559728901, req->sent is 1559728901
arp request retry one more time
arp request for target ip address: a00042c is sending
arp request retry one more time
arp request for target ip address: a00042c is sending
arp request retry one more time
arp request for target ip address: a00042c is sending
arp request retry one more time
arp request for target ip address: a00042c is sending
arp request retry one more time
arp request for target ip address: a00042c is sending
arp request failed, send icmp packet
```

图 10: 实验二 (2)-(3) 路由 3 程序运行结果

七、 其他

1. 不足

本实验完成的不足之处有：arp 缓存表的替换算法应该是随机替换，而代码中实现的是先入先出；arp 缓存没有释放缓存包和缓存条目的内存，有内存泄露问题。

2. 疑问

为什么实验一和实验二只能 ping 通入端口，不能 ping 通出端口？实验二两个主机节点也相互 ping 不通（router-reference 也是如此）。

3. 收获

通过对 icmp 协议和 arp 协议底层的代码书写，深刻学习了 icmp 协议和 arp 协议的机制。

4. 想法

mininet 软件对于底层测试不够友好，没有集成的 GUI 界面，测试的自动化程度不够，修改一处代码以后，需要手动在 mininet 里重启应用，需要手动发包，需要手动使用 wireshark 查看包，自动化程度太低，制约了生产力。

5. 说明

报告文档通过 tex 程序输出，源文件在代码目录的 report 目录下；代码可以在 Ubuntu 下通过 make router 命令编译链接生成可执行程序；代码仓库地址：`https://github.com/mrbian/091M4002HBP`。联系邮箱为：`mrbianliunian@outlook.com`。