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。

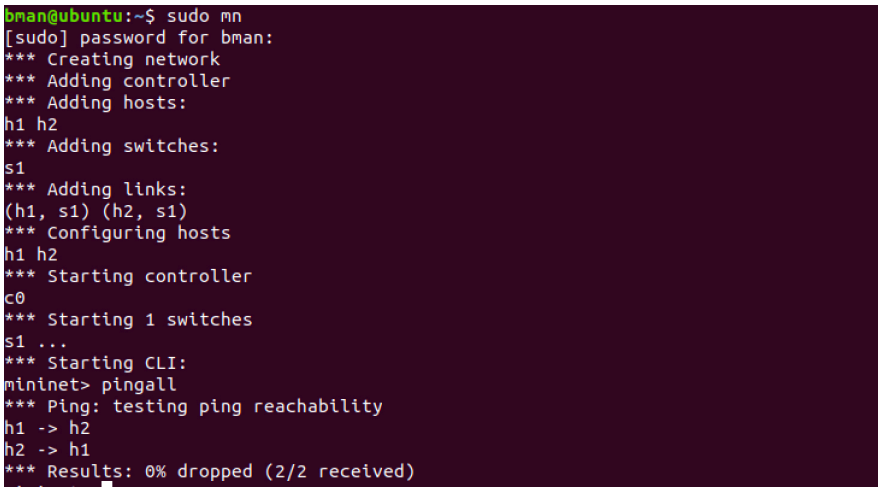

图 1: mininet 成功安装验证图

2. 编写 arpcache_lookup 函数

```
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN]) {
    //     ARP           ip4      mac                          1              0
    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)
{
    //     IPv4        MAC                          ARP
    //                 IPv4
    //

    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(flag == 0) {
        //
        struct arp_req *new_req = (struct arp_req *)malloc(sizeof(struct arp_req));
        //
        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);
        //
        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])
{
    //            ip4
    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) {
        //
        //
        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;
    }

    //                        IP              MAC
    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

```
void *arpcache_sweep(void *arg)
{
    int i = 0;
    time_t now;
    while (1) {
        sleep(1);
        //     arp             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;
            }
        }

        //                          1s
        //               5              icmp
        struct arp_req *req = NULL;
        struct cached_pkt *pkt = NULL;
        list_for_each_entry(req, &arpcache.req_list, list) {
            if(req->retries >= 5) {
                pkt = NULL;
                list_for_each_entry(pkt, &req->cached_packets, list) {
                    printf("arp request failed, send icmp packet\n");
                    icmp_send_packet(pkt->packet, pkt->len, 3, 1);
                }
                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;
                    req->sent = now;
                    arp_send_request(req->iface, req->ip4);
                }
            }
        }
    }

    return NULL;
}
```

6. 编写 arp_send_request 函数

```
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    //          arp
    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);

    //
    // 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);

    //
    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)
{
    //     arp                 arp
    //                ip    mac
    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);

    //
    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);

    //                    ip
    //
    //
    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

```
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_arp * ea = packet_to_arp_hdr(packet);
    //
    ea->arp_op = ntohs(ea->arp_op);
    ea->arp_tpa = ntohl(ea->arp_tpa);
    ea->arp_spa = ntohl(ea->arp_spa);
    //       arp        arp                 ARP
    arpcache_insert(ea->arp_spa, ea->arp_sha);
    if(ea->arp_op == ARPOP_REQUEST) {
        //     arp
        arp_send_reply(iface, ea);
    }
    free(packet);
}
```

9. 编写 iface_send_packet_by_arp 函数

```
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)
{
    //      ip
    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;
        //
        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:         //
                // 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
            // todo
            printf("this packet icmp type is 8, todo \n");
            break;
        case 11:      //
            // 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));  //              checksum
            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);
    // (                    TTL    1                    )
    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 {
        //     ICMP Destination Net Unreachable
        icmp_send_packet(packet, len, 3, 0);
        free(packet);
        return;
    }
}
```

14. 完成实验内容一

(1) 运行 router_topo.py 生成的网络结构如图 2 所示。

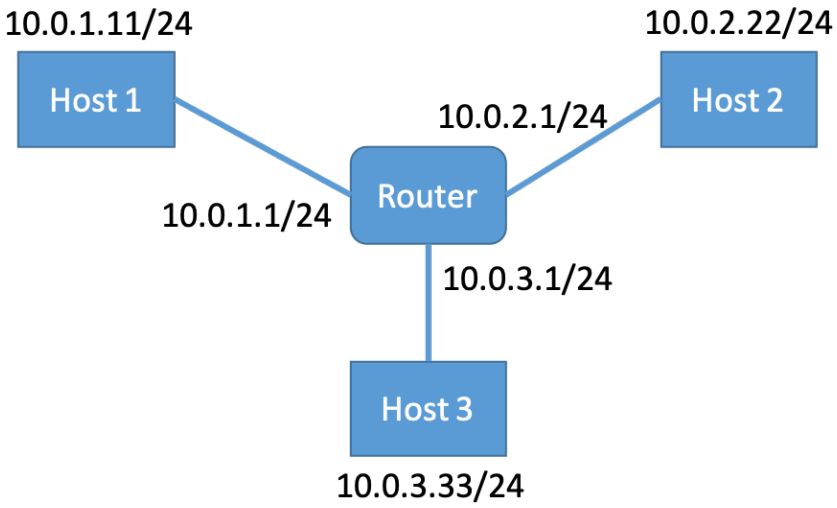

图 2: router_topo.py 生成的网络

15. 完成实验内容二

六、 实验结果与分析

七、 说明