实验 8 · 路由器转发实验

吴嘉皓
2015K8009915007

一、实验内容

1. 补全 ip.c、arp.c、arpcache.c、icmp.c 文件中的相关函数，实现路由器转发 ip 包；
2. 构造一个包含多个 router 的网络拓扑结构，并补全路由表，使用 1 中完成的路由器逻辑函数实现在新的拓扑结构中 ip 包的转发；

二、实验流程

（一）代码目录

```
2015K8009915007_吴嘉皓_08.tar.gz
    |—08-router
      ├── arp.c
      ├── arpcache.c
      ├── icmp.c
      ├── include
      │   ├── arpcache.h
      │   ├── arp.h
      │   ├── base.h
      │   ├── checksum.h
      │   ├── ether.h
      │   ├── hash.h
      │   ├── icmp.h
      │   ├── ip.h
      │   ├── list.h
      │   ├── log.h
      │   ├── packet.h
      │   ├── rtable.h
      │   └── types.h
      ├── ip.c
      ├── main.c
      ├── Makefile
      ├── packet.c
      ├── router_topo.py
      ├── rtable.c
      ├── rtable_internal.c
      ├── scripts
      │   ├── disable_arp.sh
```

```
│       ├── disable_icmp.sh
│       ├── disable_ip_forward.sh
│       ├── disable_ipv6.sh
│       └── disable_offloading.sh
└── three_router_topo.py
```

（二）实验流程

1. 按序在 08-router 目录下输入如下命令：

```
make
sudo python router_topo.py
mininet> xterm h1 r1
(xterm) r1> ./router
(xterm) h1> ping 10.0.1.1  -c 4
(xterm) h1> ping 10.0.2.22 -c 4
(xterm) h1> ping 10.0.3.33 -c 4
(xterm) h1> ping 10.0.3.11 -c 4
(xterm) h1> ping 10.0.4.1  -c 4
```

2. 按序在 08-router 目录下输入如下命令：

```
make
sudo python three_router_topo.py
mininet> xterm h1 h2

(xterm) h1> ping 10.0.1.1  -c 4
(xterm) h1> ping 10.0.2.2  -c 4
(xterm) h1> ping 10.0.3.2  -c 4

(xterm) h2> ping 10.0.2.1  -c 4
(xterm) h2> ping 10.0.3.1  -c 4
(xterm) h2> ping 10.0.4.1  -c 4

(xterm) h1> traceroute 10.0.4.44
(xterm) h2> traceroute 10.0.1.11
```

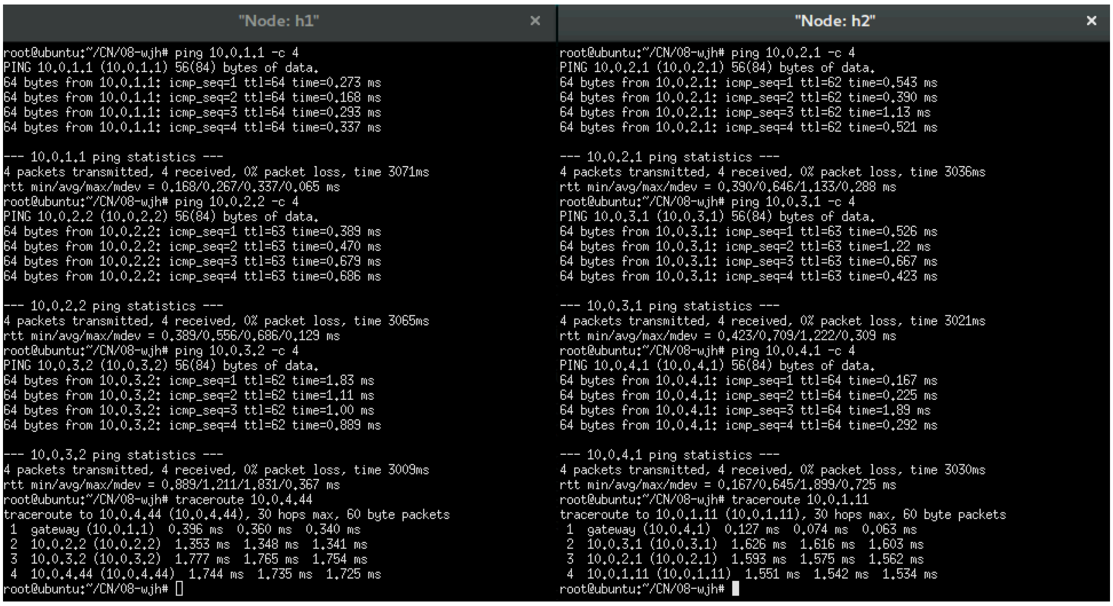
三、实验结果

（一）单路由拓扑的输出结果

"Node: h1"

```
root@ubuntu:~/CN/08-wjh# ping 10.0.1.1 -c 4
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=0.272 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.069 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.072 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.159 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3052ms
rtt min/avg/max/mdev = 0.069/0.143/0.272/0.082 ms
root@ubuntu:~/CN/08-wjh# ping 10.0.2.22 -c 4
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=0.230 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.158 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.294 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.140 ms

--- 10.0.2.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3050ms
rtt min/avg/max/mdev = 0.140/0.205/0.294/0.062 ms
root@ubuntu:~/CN/08-wjh# ping 10.0.3.33 -c 4
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=0.212 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.140 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.178 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.126 ms

--- 10.0.3.33 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3077ms
rtt min/avg/max/mdev = 0.126/0.164/0.212/0.033 ms
root@ubuntu:~/CN/08-wjh# ping 10.0.3.11 -c 4
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
From 10.0.1.1 icmp_seq=2 Destination Host Unreachable
From 10.0.1.1 icmp_seq=3 Destination Host Unreachable
From 10.0.1.1 icmp_seq=4 Destination Host Unreachable

--- 10.0.3.11 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3073ms
pipe 4
root@ubuntu:~/CN/08-wjh# ping 10.0.4.1 -c 4
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
From 10.0.1.1 icmp_seq=2 Destination Net Unreachable
From 10.0.1.1 icmp_seq=3 Destination Net Unreachable
From 10.0.1.1 icmp_seq=4 Destination Net Unreachable

--- 10.0.4.1 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3077ms
```

（二）多路由节点拓扑结构输出结果



四、结果分析

（一）单路由拓扑结果分析

10.0.1.1、10.0.2.22 和 10.0.3.33 都能 PING 通；

10.0.3.34 能够匹配上 ip，因为掩码为 255.255.255.0；但是其 Host 部分并不存在，因此返回的结果是 Host Unreachable；

10.0.4.1 匹配不上 ip，因为在路由表中查找不到对应的项，于是返回的结果是 Net Unreachable。

（二）多路由节点拓扑结果分析

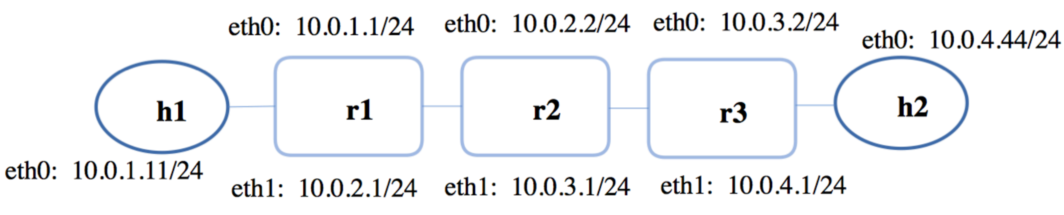


图 1 三路由节点拓扑图

由拓扑图对比可知，

H1 PING 10.0.1.1、10.0.2.22、10.0.3.2 能够 PING 通；

H2 PING 10.0.2.1、10.0.3. 1、10.0.4.1 能够 PING 通；

Traceroute 的结果也正是其通过的路径；

（三）代码实现分析

➪ handle_ip_packet()

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    struct iphdr *ip = packet_to_ip_hdr(packet);
    u32 dst_ip = ntohl(ip->daddr);

    if (ip->protocol == IPPROTO_ICMP) {
        struct icmphdr *icmp = packet_to_icmp_hdr(packet, ip);
        if((dst_ip == iface->ip) && (icmp->type == ICMP_ECHOREQUEST)){
            icmp_send_packet(packet, len, ICMP_ECHOREPLY, 0);
            free(packet);
            return ;
        }
    }
    ip_forward_packet(dst_ip, packet, len);
}
```

➪ ip_forward_packet(): ip 包传递

```
void ip_forward_packet(u32 ip_dst, char *packet, int len)
{
    // check the TTL
    struct iphdr *ip = packet_to_ip_hdr(packet);
    if((--ip->ttl) <= 0){
        icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL);
        free(packet);
        return ;
    }

    // look up router table to find the rt_entry
    rt_entry_t *entry = longest_prefix_match(ip_dst);
    if (!entry) {
        icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
        free(packet);
        return ;
    }

    // determine the next hop to forward the packet
    u32 next_hop = (!entry->gw) ? ip_dst : entry->gw;

    // update the source mac address in packet as the mac address of the forward port
    ether_header_t *eth = (ether_header_t *)packet;
    eth->ether_type = htons(ETH_P_IP);
    memcpy(eth->ether_shost, entry->iface->mac, ETH_ALEN);

    // update the checksum
    ip->checksum = ip_checksum(ip);

    iface_send_packet_by_arp(entry->iface, next_hop, packet, len);
}
```

➪ longest_prefix_match(): 在路由表中查找最长 mask 的匹配项

```c
rt_entry_t *longest_prefix_match(u32 dst){
    u32 entry_addr = 0, dst_addr = 0, mask = 0;
    rt_entry_t *entry = NULL, *desired_entry = NULL;
    list_for_each_entry(entry, &rtable, list){
        entry_addr = entry->dest & entry->mask;
        dst_addr   = dst         & entry->mask;
        if((dst_addr == entry_addr) && (mask < entry->mask)){
            desired_entry = entry;
            mask = entry->mask;
        }
    }
    return desired_entry;
}
```

⇨ arp_send_request()

```c
void arp_send_request(iface_info_t *iface, u32 dst_ip){
    // if dest mac is unknown, set it as 0x ff:ff:ff:ff:ff:ff
    u8 dst_mac[ETH_ALEN];
    memset(dst_mac, 0xff, ETH_ALEN);

    // set up a new packet
    char *packet = (char *)malloc(ETHER_HDR_SIZE+ARP_SIZE);

    // set ether header string
    ether_header_t *eth_hdr = (ether_header_t *)packet;
    eth_hdr->ether_type = htons(ETH_P_ARP);
    memcpy(eth_hdr->ether_shost, iface->mac, ETH_ALEN);
    memcpy(eth_hdr->ether_dhost, dst_mac, ETH_ALEN);

    // set arp request protocol string
    ether_arp_t *arp_hdr = packet_to_arp_hdr(packet);
    arp_init_header(arp_hdr, iface, ARPOP_REQUEST, iface->ip, dst_ip);

    // send packet through the provided iface
    iface_send_packet(iface, packet, ETHER_HDR_SIZE+ARP_SIZE);
}
```

⇨ arp_send_reply()

```
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr){
    char *packet = (char *)malloc(ETHER_HDR_SIZE+ARP_SIZE);

    // set ether header string
    ether_header_t *eth_hdr = (ether_header_t *)packet;
    eth_hdr->ether_type = htons(ETH_P_ARP);
    memcpy(eth_hdr->ether_shost, iface->mac, ETH_ALEN);
    memcpy(eth_hdr->ether_dhost, req_hdr->arp_sha, ETH_ALEN);

    // set arp reply protocol string
    u32 reply_tpa = ntohl(req_hdr->arp_spa);
    ether_arp_t *reply_hdr = packet_to_arp_hdr(packet);
    arp_init_header(reply_hdr, iface, ARPOP_REPLY, iface->ip, reply_tpa);
    memcpy(reply_hdr->arp_tha, req_hdr->arp_sha, ETH_ALEN);

    // send packet through the provided iface
    iface_send_packet(iface, packet, ETHER_HDR_SIZE+ARP_SIZE);
}
```

⇨ handle_arp_packet()

```
void handle_arp_packet(iface_info_t *iface, char *packet, int len){
    // resolve the received packet
    ether_arp_t *arp = packet_to_arp_hdr(packet);
    u32 tpa = ntohl(arp->arp_tpa);
    // handle different arp operations
    switch(ntohs(arp->arp_op)){
        case ARPOP_REQUEST:
            if(tpa == iface->ip){
                arp_send_reply(iface, arp);
                arpcache_insert(arp->arp_spa, arp->arp_sha);
            }
            break;
        case ARPOP_REPLY:
            if (tpa == iface->ip)
                arpcache_insert(arp->arp_spa, arp->arp_sha);
            break;
        default:
            // log(ERROR, "Unknown arp operation type, ingore it.");
            printf("Unknown arp operation type, ingore it.\n");
            break;
    }
}
```

⇨ arpcache_lookup()

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

⇨ arp_append_packet()

```
void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len){
    struct cached_pkt *new_cached_pkt = (struct cached_pkt *)malloc(CACHE_PKT_SIZE);
    if(!new_cached_pkt) exit(-1);
    new_cached_pkt->len = len;
    new_cached_pkt->packet = (char *)malloc(len);//packet;
    if(!new_cached_pkt->packet) exit(-1);
    memcpy(new_cached_pkt->packet, packet, len);
    free(packet);

    pthread_mutex_lock(&arpcache.lock);

    struct arp_req *req_entry = NULL, *req_q;
    list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
        if((req_entry->ip4 == ip4) && (req_entry->iface == iface)) {
            list_add_tail(&(new_cached_pkt->list), &(req_entry->cached_packets));
            pthread_mutex_unlock(&arpcache.lock);
            return ;
        }
    }
    struct arp_req *new_req_entry = (struct arp_req *)malloc(ARP_REQ_SIZE);
    if(!new_req_entry){
        pthread_mutex_unlock(&arpcache.lock);
        exit(-1);
    }
    new_req_entry->ip4     = ip4;
    new_req_entry->iface   = iface;
    new_req_entry->sent    = time(NULL);
    new_req_entry->retries = 0;

    init_list_head(&(new_req_entry->cached_packets));
    init_list_head(&(new_req_entry->list));
    list_add_tail(&(new_cached_pkt->list), &(new_req_entry->cached_packets));
    list_add_tail(&(new_req_entry->list), &(arpcache.req_list));
    pthread_mutex_unlock(&arpcache.lock);
    arp_send_request(iface, ip4);
}
```

⇨ arp_insert()

```c
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN]){
    pthread_mutex_lock(&arpcache.lock);
    // check whether the arpcache is full
    int index = -1;
    for(int i = 0; i < MAX_ARP_SIZE; i++)
        if(!arpcache.entries[i].valid){
            index = i;
            break;
        }
    // if arpcache is full,
    // randomly choose an index to insert the mapping
    if(index == -1) {
        srand((unsigned) time(NULL));
        index = (int)(rand()%MAX_ARP_SIZE);
    }
    arpcache.entries[index].ip4 = ntohl(ip4);
    arpcache.entries[index].valid = 1;
    arpcache.entries[index].added = time(NULL);
    memcpy(arpcache.entries[index].mac, mac, ETH_ALEN);

    // handle pending packets waiting for this mapping
    struct ether_header *eh = NULL;
    struct arp_req *req_entry = NULL, *req_q = NULL;
    struct cached_pkt *pkt_entry = NULL, *pkt_q = NULL;
    list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list){
        if(req_entry->ip4 == ntohl(ip4)){
            list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list){
                eh = (struct ether_header *)(pkt_entry->packet);
                memcpy(eh->ether_dhost, mac, ETH_ALEN);
                iface_send_packet(req_entry->iface, pkt_entry->packet, pkt_entry->len);
                list_delete_entry(&(pkt_entry->list));
                free(pkt_entry);
                pkt_entry = NULL;
            }
            list_delete_entry(&(req_entry->list));
            free(req_entry);
            req_entry = NULL;
            break;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
}
```

⇨ arp_sweep()

```
void *arpcache_sweep(void *arg)
{
    while (1) {
        sleep(1);
        pthread_mutex_lock(&arpcache.lock);
        // For the IP->mac entry
        time_t cur_time = time(NULL);
        for(int i = 0; i < MAX_ARP_SIZE; i++)
            if((cur_time - arpcache.entries[i].added) > 15){
                arpcache.entries[i].valid = 0;
            }

        // For the pending packets
        struct arp_req *req_entry = NULL, *req_q;
        struct cached_pkt *pkt_entry = NULL, *pkt_q;
        list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
            if((cur_time - req_entry->sent) >= 1){
                if((++req_entry->retries) > 5){
                    list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list){
                        pthread_mutex_unlock(&arpcache.lock);
                        icmp_send_packet(pkt_entry->packet, pkt_entry->len,
                                         ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
                        pthread_mutex_lock(&arpcache.lock);
                        list_delete_entry(&(pkt_entry->list));
                        free(pkt_entry->packet);
                        pkt_entry->packet = NULL;
                        free(pkt_entry);
                        pkt_entry = NULL;
                    }
                    list_delete_entry(&(req_entry->list));
                    free(req_entry);
                    req_entry = NULL;
                }
                else
                    arp_send_request(req_entry->iface, req_entry->ip4);
            }
        }
        pthread_mutex_unlock(&arpcache.lock);
    }
    return NULL;
}
```

```c
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    // prepare for out+pkt
    long out_len = 0;
    char *out_pkt = NULL;

    // resolve ip header of in_pkt
    struct iphdr *in_ip_hdr = packet_to_ip_hdr(in_pkt);
    u32 out_daddr = ntohl(in_ip_hdr->saddr);
    u32 out_saddr = longest_prefix_match(out_daddr)->iface->ip;

    // set up ICMP
    if (type != ICMP_ECHOREPLY) {
        out_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + ICMP_HDR_SIZE +
            IP_HDR_SIZE(in_ip_hdr) + 8;
    } else {
        out_len = len - IP_HDR_SIZE(in_ip_hdr) + IP_BASE_HDR_SIZE;
    }

    // set up out_pkt
    out_pkt = (char*) malloc(out_len);

    struct icmphdr *icmp = PACKET_TO_ICMP(out_pkt);

    // set ether header
    struct ether_header *eh = (struct ether_header *)out_pkt;
    eh->ether_type = htons(ETH_P_IP);

    // set ip header
    struct iphdr *out_ip_hdr = packet_to_ip_hdr(out_pkt);
    ip_init_hdr(out_ip_hdr, out_saddr, out_daddr,
                (out_len - ETHER_HDR_SIZE), IPPROTO_ICMP);

    // set icmp header
    memset(icmp, 0, ICMP_HDR_SIZE);
    icmp->code = code;
    icmp->type = type;

    // set the rest of icmp
    int size = 0;
    char *src = NULL, *dst = NULL;
    if (type != ICMP_ECHOREPLY) {
        dst = ((char*)icmp + ICMP_HDR_SIZE);
        src = (char*)in_ip_hdr;
        size = IP_HDR_SIZE(in_ip_hdr) + 8;
        memcpy(dst, src, size);
    }
    else{
        dst = ((char *)icmp) + ICMP_HDR_SIZE - 4;
        src = (char *)(in_pkt + ETHER_HDR_SIZE + IP_HDR_SIZE(in_ip_hdr) + 4);
        size = len - ETHER_HDR_SIZE - IP_HDR_SIZE(in_ip_hdr) - 4;
        memcpy(dst, src, size);
    }
    icmp->checksum = icmp_checksum(icmp, ICMP_SIZE(out_len));

    // send ICMP
    ip_send_packet(out_pkt, out_len);
}
```