路由器转发实验报告

热伊莱·图尔贡　2018K8009929030

一、 实验题目：

路由器转发实验

二、 实验内容：

➢ 内容一：

实现路由器转发机制，对于给定拓扑（router_topo.py ），在 r1 上执行路由器程序，进行数据包的处理。

在 h1 上进行 ping 实验：

Ping 10.0.1.1 (r1)，能够 ping 通

Ping 10.0.2.22 (h2)，能够 ping 通

Ping 10.0.3.33 (h3)，能够 ping 通

Ping 10.0.3.11，返回 ICMP Destination Host Unreachable

Ping 10.0.4.1，返回 ICMP Destination Net Unreachable

➢ 内容二：

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。

连通性测试：

终端节点 ping 每个路由器节点的入端口 IP 地址，能够 ping 通

路径测试

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的 IP 信息

三、 实验过程：

➢ 完成 arp.c，实现处理 ARP 请求和应答

收到 ARP 请求时，如果 Target Proto Addr 为本端口地址，则 ARP 应答。

转发数据包时，如果 ARP 缓存中没有相应条目，则发送 ARP 请求。

- 实现 arp_send_request(iface_info_t *iface, u32 dst_ip)发送 arp 请求

```
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    // fprintf(stderr, "TODO: send arp request when lookup failed in arpcache.\n");
    char *packet = (char *)malloc(ETHER_HDR_SIZE + sizeof(struct ether_arp));
    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    memset(eh->ether_dhost, 0xff, ETH_ALEN);
    eh->ether_type = htons(ETH_P_ARP);
    struct ether_arp *arp_pkt = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
    arp_pkt->arp_hrd = htons(ARPHRD_ETHER);
    arp_pkt->arp_pro = htons(ETH_P_IP);
    arp_pkt->arp_hln = (u8)ETH_ALEN;
    arp_pkt->arp_pln = (u8)4;
    arp_pkt->arp_op = htons(ARPOP_REQUEST);
    memcpy(arp_pkt->arp_sha, iface->mac, ETH_ALEN);
    arp_pkt->arp_spa = htonl(iface->ip);
    memset(arp_pkt->arp_tha, 0, ETH_ALEN);
    arp_pkt->arp_tpa = htonl(dst_ip);
    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));
}
```

- 实现 arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)发送 arp 回复

```
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)
{
    // fprintf(stderr, "TODO: send arp reply when receiving arp request.\n");
    char *packet = (char *)malloc(ETHER_HDR_SIZE + sizeof(struct ether_arp));
    struct ether_header *eh = (struct ether_header *)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);
    struct ether_arp *arp_pkt = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
    arp_pkt->arp_hrd = htons(ARPHRD_ETHER);
    arp_pkt->arp_pro = htons(ETH_P_IP);
    arp_pkt->arp_hln = (u8)ETH_ALEN;
    arp_pkt->arp_pln = (u8)4;
    arp_pkt->arp_op = htons(ARPOP_REPLY);
    memcpy(arp_pkt->arp_sha, iface->mac, ETH_ALEN);
    arp_pkt->arp_spa = htonl(iface->ip);
    memcpy(arp_pkt->arp_tha, req_hdr->arp_sha, ETH_ALEN);
    arp_pkt->arp_tpa = req_hdr->arp_spa;
    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));
}
```

- 实现 handle_arp_packet(iface_info_t *iface, char *packet, int len)根据收到的 arp 包的 op 部分内容，执行相应操作

```
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    // fprintf(stderr, "TODO: process arp packet: arp request & arp reply.\n");
    struct ether_arp *arp_pkt = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
    if (ntohs(arp_pkt->arp_op) == ARPOP_REPLY)
    {
        arpcache_insert(ntohl(arp_pkt->arp_spa), arp_pkt->arp_sha);
    }
    else if (ntohs(arp_pkt->arp_op) == ARPOP_REQUEST && ntohl(arp_pkt->arp_tpa) == iface->ip)
    {
        arp_send_reply(iface, arp_pkt);
    }
    free(packet);
}
```

➢ 完成 arpcache.c，实现 ARP 缓存管理

进行 ARP 查询、更新等操作

- Lookup：遍历 arp 表，若找到 ip 项与给定 ip 相同，拷贝 mac 地址并返回 1，否则返回 0。
- Append：遍历 arpreq 表，找到对应 iface 和 ip 的项，把给定的包挂在该项的链表中。发送相应的 arp 请求。
- Insert：遍历 arp 表，若找到 ip 项与给定 ip 相同，则更新。否则寻找一个空的项填入，若无空项，随机替换一项。插入后遍历 arpreq 表，找到所有 ip 项与给定 ip 相同的项，把该项下挂的所有包填上相应的 mac 地址并发出，然后删除该项。

- Sweep：每隔 1 秒，遍历 arp 表，将更新时间超过 15s 的条目设为无效。遍历 arpreq 表，如果一个 IP 对应的 ARP 请求发出去已经超过了 1 秒，重新发送 ARP 请求；如果发送超过 5 次仍未收到 ARP 应答，则对该队列下的数据包依次回复 ICMP（Destination Host Unreachable）消息，并删除等待的数据包。然后删除该项。

```
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    // fprintf(stderr, "TODO: lookup ip address in arp cache.\n");
    pthread_mutex_lock(&arpcache.lock);
    for (int i = 0; i < MAX_ARP_SIZE; i++)
    {
        if (arpcache.entries[i].valid && arpcache.entries[i].ip4 == ip4)
        {
            memcpy(mac, arpcache.entries[i].mac, ETH_ALEN);
            pthread_mutex_unlock(&arpcache.lock);
            return 1;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
    return 0;
}
```

```
void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len)
{
    // fprintf(stderr, "TODO: append the ip address if lookup failed, and send arp request if necessary.\n");
    arp_send_request(iface, ip4);
    pthread_mutex_lock(&arpcache.lock);
    struct arp_req *entry = NULL, *q;
    int cached = 0;
    list_for_each_entry_safe(entry, q, &arpcache.req_list, list)
    {
        if (entry->ip4 == ip4 && entry->iface == iface)
        {
            cached = 1;
            break;
        }
    }
    if (!cached)
    {
        entry = malloc(sizeof(struct arp_req));
        init_list_head(&entry->cached_packets);
        entry->iface = iface;
        entry->ip4 = ip4;
        entry->retries = 0;
        list_add_tail(&entry->list, &arpcache.req_list);
    }
    struct cached_pkt *pkt = malloc(sizeof(struct cached_pkt));
    pkt->len = len;
    pkt->packet = packet;
    list_add_tail(&pkt->list, &entry->cached_packets);

    entry->retries++;
    entry->sent = time(NULL);
    pthread_mutex_unlock(&arpcache.lock);
}
```

```c
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
{
    // fprintf(stderr, "TODO: insert ip->mac entry, and send all the pending packets.\n");
    pthread_mutex_lock(&arpcache.lock);
    int vid = rand() % MAX_ARP_SIZE;
    for (int i = 0; i < MAX_ARP_SIZE; i++){
        if (arpcache.entries[i].valid == 0)
        {vid = i;}
        else{
            if (arpcache.entries[i].ip4 == ip4){
            vid = i;
            break;}}}
    arpcache.entries[vid].valid = 1;
    arpcache.entries[vid].ip4 = ip4;
    memcpy(arpcache.entries[vid].mac, mac, ETH_ALEN);
    arpcache.entries[vid].added = time(NULL);
    struct arp_req *entry = NULL, *q;
    list_for_each_entry_safe(entry, q, &arpcache.req_list, list){
        if (entry->ip4 == ip4){
            struct cached_pkt *pkt = NULL, *q1;
            list_for_each_entry_safe(pkt, q1, &entry->cached_packets, list)
            {
                struct ether_header *eh = (struct ether_header *)pkt->packet;
                memcpy(eh->ether_dhost, mac, ETH_ALEN);
                memcpy(eh->ether_shost, entry->iface->mac, ETH_ALEN);
                iface_send_packet(entry->iface, pkt->packet, pkt->len);
                list_delete_entry(&pkt->list);
                free(pkt);
            }
            list_delete_entry(&entry->list);
            free(entry);
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
}
```

```c
void *arpcache_sweep(void *arg)
{
    while (1)
    {
        sleep(1);
        // fprintf(stderr, "TODO: sweep arpcache periodically: remove old entries, resend arp requests .\n");
        pthread_mutex_lock(&arpcache.lock);
        time_t now = time(NULL);
        for (int i = 0; i < MAX_ARP_SIZE; i++)
        {
            if (arpcache.entries[i].valid && now > arpcache.entries[i].added + ARP_ENTRY_TIMEOUT)
                arpcache.entries[i].valid = 0;
        }
        struct arp_req *entry = NULL, *q;
```

```
        list_for_each_entry_safe(entry, q, &arpcache.req_list, list)
        {
            if (entry->retries >= ARP_REQUEST_MAX_RETRIES)
            {
                struct cached_pkt *pkt = NULL, *ql;
                list_for_each_entry_safe(pkt, ql, &entry->cached_packets, list)
                {
                    struct ether_header *eh = (struct ether_header *)pkt->packet;
                    u8 *mac = eh->ether_dhost;
                    iface_info_t *iface = NULL;
                    list_for_each_entry(iface, &instance->iface_list, list)
                    {
                        if (memcmp(mac, iface->mac, ETH_ALEN) == 0)
                            break;
                    }
                    icmp_send_packet(iface, pkt->packet, pkt->len, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
                    free(pkt->packet);
                    list_delete_entry(&pkt->list);
                    free(pkt);
                }
                list_delete_entry(&entry->list);
                free(entry);
            }
            else if (now > entry->sent + 1)
            {
                arp_send_request(entry->iface, entry->ip4);
                entry->retries++;
            }
        }
        pthread_mutex_unlock(&arpcache.lock);
    }
    return NULL;
}
```

- 完成 ip_base.c 和 ip.c，实现 IP 地址查找和 IP 数据包转发

收到数据包后，查找对应的转发端口；更新 IP 头部，转发数据包

若当前包的目的 ip 为当前端口 ip 且为 ping 包，则返回 icmpreply 包，否则转发该 IP 包。将 ttl 减一，若减为 0 返回出错 ICMP 包。否则重新计算 checksum，路由查找下一跳 ip 和端口号，若找到则转发，若查找失败，返回出错 ICMP 包。

- 完成 icmp.c，实现发送 ICMP 数据包

路由表查找失败；ARP 查询失败； TTL 值为 0；收到 ping 本端口的包

按照格式填充 ICMP 包。其中，若发送的是 reply，则 Rest of ICMP Header 拷贝 Ping 包
中的相应字段，否则 Rest of ICMP Header 前 4 字节设置为 0，接着拷贝收到数据包的 IP 头 部和随后的 8 字节。按照格式填充 ip 报头。

四、 实验结果：

- 内容一：对于给定拓扑（router_topo.py），在 r1 上执行路由器程序，进行数据包的处理。

```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# 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=0.179 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.001 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.074 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3048ms
rtt min/avg/max/mdev = 0.001/0.086/0.179/0.063 ms
```

```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# 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=0.091 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.101 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.128 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.082 ms

--- 10.0.2.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3079ms
rtt min/avg/max/mdev = 0.082/0.100/0.128/0.017 ms
```

```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# 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=0.084 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.088 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.137 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.103 ms

--- 10.0.3.33 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3063ms
rtt min/avg/max/mdev = 0.084/0.103/0.137/0.020 ms
```

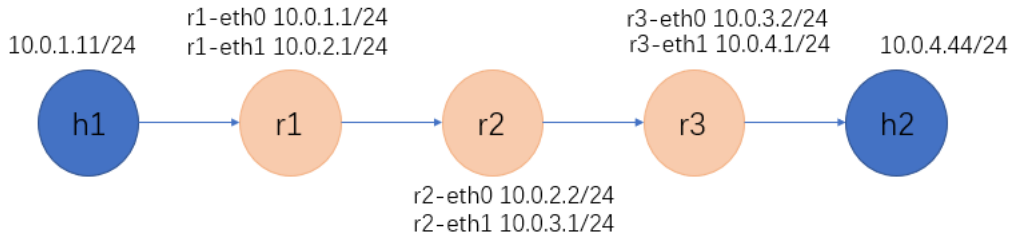
```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# ping 10.0.3.11 -c 4
PING 10.0.3.11 (10.0.3.11) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Host Unreachable
From 10.0.1.1 icmp_seq=2 Destination Host Unreachable
From 10.0.1.1 icmp_seq=3 Destination Host Unreachable
From 10.0.1.1 icmp_seq=4 Destination Host Unreachable

--- 10.0.3.11 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3059ms
pipe 4
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# 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 3063ms
```

Ping 10.0.1.1（r1）：ping 路由器入端口 ip，能够 ping 通
Ping 10.0.2.22（h2），Ping 10.0.3.33（h3）：
ping 能够连接到的节点，能够 ping 通
Ping 10.0.3.11 ：ping 不存在的节点，返回 ICMP Destination Host Unreachable
Ping 10.0.4.1 ：ping 不存在的网段，返回 ICMP Destination Net Unreachable
与理论结果相同，验证成功。

- 内容二：

构造一个包含多个路由器节点组成的网络

```
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 10.0.4.44/24')

r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
r1.cmd('ifconfig r1-eth1 10.0.2.1/24')

r2.cmd('ifconfig r2-eth0 10.0.2.2/24')
r2.cmd('ifconfig r2-eth1 10.0.3.1/24')

r3.cmd('ifconfig r3-eth0 10.0.3.2/24')
r3.cmd('ifconfig r3-eth1 10.0.4.1/24')

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

r1.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
r1.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')

r2.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.2.1 dev r2-eth0')
r2.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.3.2 dev r2-eth1')

r3.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.3.1 dev r3-eth0')
r3.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.3.1 dev r3-eth0')
```



```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# 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=0.132 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.071 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.070 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.084 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3069ms
rtt min/avg/max/mdev = 0.070/0.089/0.132/0.025 ms
```

```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# ping 10.0.2.2 -c 4
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=63 time=0.320 ms
64 bytes from 10.0.2.2: icmp_seq=2 ttl=63 time=0.102 ms
64 bytes from 10.0.2.2: icmp_seq=3 ttl=63 time=0.102 ms
64 bytes from 10.0.2.2: icmp_seq=4 ttl=63 time=0.103 ms

--- 10.0.2.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3056ms
rtt min/avg/max/mdev = 0.102/0.156/0.320/0.094 ms
```

```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# ping 10.0.3.2 -c 4
PING 10.0.3.2 (10.0.3.2) 56(84) bytes of data.
64 bytes from 10.0.3.2: icmp_seq=1 ttl=62 time=0.158 ms
64 bytes from 10.0.3.2: icmp_seq=2 ttl=62 time=0.123 ms
64 bytes from 10.0.3.2: icmp_seq=3 ttl=62 time=0.139 ms
64 bytes from 10.0.3.2: icmp_seq=4 ttl=62 time=0.131 ms

--- 10.0.3.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3075ms
rtt min/avg/max/mdev = 0.123/0.137/0.158/0.013 ms
```

```
root@rayilam-VirtualBox:/home/rayilam/CN/06-router# traceroute 10.0.4.44
traceroute to 10.0.4.44 (10.0.4.44), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.657 ms  0.629 ms  0.621 ms
 2  10.0.2.2 (10.0.2.2)  0.460 ms  0.456 ms  0.451 ms
 3  10.0.3.2 (10.0.3.2)  0.446 ms  0.442 ms  0.437 ms
 4  10.0.4.44 (10.0.4.44)  0.433 ms  0.430 ms  0.426 ms
```

终端节点 ping 每个路由器节点的入端口 IP 地址：能够 ping 通

在 h1 上 traceroute h2 ，正确输出路径上每个节点的 IP 信息

与预期结果相同，证明连通性良好，路径正确。