09-路由器转发实验报告

陈彦帆 2018K8009918002

1、实验内容

（1）在给定框架的基础上实现路由器的以下功能：转发 IP 包，根据错误信息或收到的 ping 包返回 ICMP 包，发送和接收 ARP 包，查找路由表，管理 ARPcache。

（2）在单路由器网络上完成 ping 测试。

（3）在多路由网络上完成 ping 测试和 traceroute 测试。

2、实验流程

（1）实现路由器的以下功能：转发 IP 包，根据错误信息或收到的 ping 包返回 ICMP 包，发送和接收 ARP 包，查找路由表，管理 ARPcache。

（2）在单路由器网络上完成 ping 测试：

给定的网络拓扑如下图：

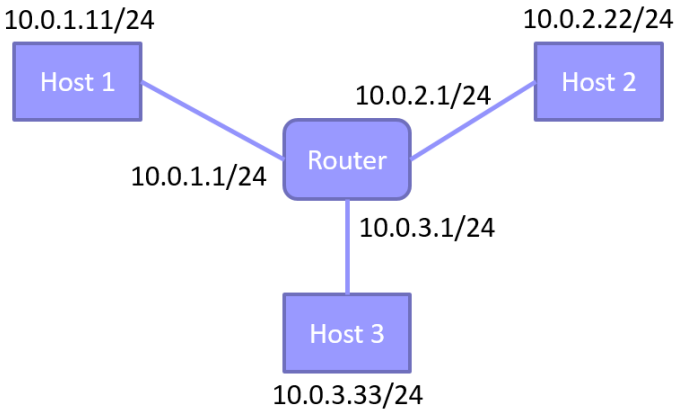

图 1

修改 router_topy.py 并执行。其中测试部分语句如下：

```
r1.cmd('./router &')
print(h1.cmd('ping -c 2 10.0.1.1'))
print(h1.cmd('ping -c 2 10.0.2.22'))
print(h1.cmd('ping -c 2 10.0.3.33'))
print(h1.cmd('ping -c 2 10.0.3.11'))
print(h1.cmd('ping -c 2 10.0.4.1'))
```

（3）在多路由网络上完成 ping 测试和 traceroute 测试。

构建的网络拓扑如下图。

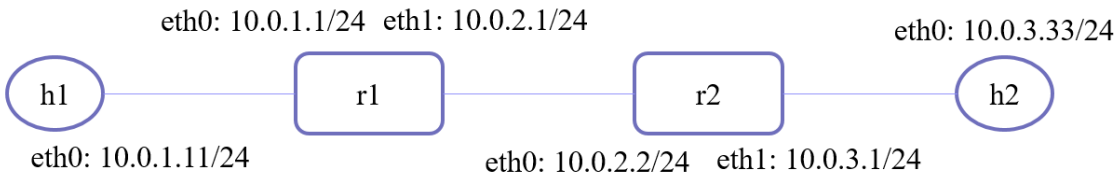

图 2

脚本见 traceroute.py。

测试部分语句如下：

```
for r in (r1,r2):
    r.cmd('./router > %s-output.txt 2>&1 &' % r)
print(h1.cmd('ping -c 1 10.0.1.1'))
print(h1.cmd('ping -c 1 10.0.2.2'))
print(h1.cmd('ping -c 1 10.0.3.33'))
print(h1.cmd('traceroute 10.0.3.33'))
```

3、分析与实现

（1）转发 IP 包

handle_ip_packet 函数：

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。

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    // fprintf(stderr, "TODO: handle ip packet.\n");
    struct iphdr *iph = packet_to_ip_hdr(packet);
    if(ntohl(iph->daddr) == iface->ip){
        if(iph->protocol == IPPROTO_ICMP) {
            struct icmphdr *icph = (struct icmphdr*)(IP_DATA(iph));
            if(icph->type == ICMP_ECHOREQUEST)
                icmp_send_packet(iface,packet,len,ICMP_ECHOREPLY,0);
        }
        free(packet);
    }
    else {
        #ifdef MYDEBUG
        fprintf(stderr, "forward.\n");
```

```
        #endif
        ip_forward_packet(iface, packet, len);
    }
}
```

ip_forward_packet 函数：

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。

```
void ip_forward_packet(iface_info_t *iface, char *packet, int len)
{
    struct iphdr *ihr = packet_to_ip_hdr(packet);
    u32 daddr = ntohl(ihr->daddr);
    if(--ihr->ttl<=0) {
        icmp_send_packet(iface,packet,len,ICMP_TIME_EXCEEDED,ICMP_EXC_TTL);
        free(packet);
    }
    else {
        rt_entry_t *rt_entry = longest_prefix_match(daddr);
        if(!rt_entry) {
            icmp_send_packet(iface,packet,len,ICMP_DEST_UNREACH,ICMP_NET_UNREACH);
            free(packet);
        }
        else{
            ihr->checksum = ip_checksum(ihr);
            u32 next_ip = daddr;
            if(rt_entry->gw)
                next_ip = rt_entry->gw;
            iface_send_packet_by_arp(rt_entry->iface,next_ip,packet,len);
        }
    }
}
```

iface_send_packet_by_arp 函数：

根据下一跳 ip 查找 arp 表，若找到，则修改包的 mac 地址为找到的地址并发送，否则把包挂在 arpcache 上并发送 arp 请求。

（2）根据错误信息或收到的 ping 包返回 ICMP 包

icmp_send_packet 函数：

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。

由于收到包和返回相应的 ICMP 包之间的间隔较短，可认为网络拓扑没有发生变化，故 mac 地址和 ip 地址只需原路返回即可，无须查找 arp 表和路由表。

```
void icmp_send_packet(iface_info_t *iface, char *in_pkt, int len, u8 type, u8 code)
{
    struct ether_header *eh = (struct ether_header *)in_pkt;
    struct iphdr *ihr = packet_to_ip_hdr(in_pkt);
    int pkt_len = type==ICMP_ECHOREPLY ? len + IP_BASE_HDR_SIZE - IP_HDR_SIZE(ihr)
                                       : ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + ICMP_H
DR_SIZE + IP_HDR_SIZE(ihr) + 8;
    char *packet = malloc(pkt_len);

    struct ether_header *n_eh = (struct ether_header *)packet;
    memcpy(n_eh->ether_dhost,eh->ether_shost,ETH_ALEN);
    memcpy(n_eh->ether_shost,iface->mac,ETH_ALEN);
    n_eh->ether_type = htons(ETH_P_IP);

    struct icmphdr *n_ichr = (struct icmphdr*)(packet + ETHER_HDR_SIZE + IP_BASE_HD
R_SIZE);
    n_ichr->code = code;
    n_ichr->type = type;
    if(type == ICMP_ECHOREPLY) {
        int start = ETHER_HDR_SIZE+IP_HDR_SIZE(ihr)+4; // 4 is offsetof u of icmphd
r
        memcpy(&n_ichr->u, in_pkt+start, pkt_len-start);
    }
    else {
        memset(&n_ichr->u,0,sizeof(n_ichr->u));
        memcpy((char*)n_ichr + ICMP_HDR_SIZE, in_pkt+ETHER_HDR_SIZE, IP_HDR_SIZE(ih
r)+8);
    }
    n_ichr->checksum = icmp_checksum(n_ichr,pkt_len-
(ETHER_HDR_SIZE + IP_BASE_HDR_SIZE));

    struct iphdr *n_ihr = packet_to_ip_hdr(packet);
    ip_init_hdr(n_ihr,iface->ip,ntohl(ihr->saddr),pkt_len-
ETHER_HDR_SIZE,IPPROTO_ICMP);

    iface_send_packet(iface, packet, pkt_len);
}
```

（3）发送和接收 ARP 包

arp_send_request 和 arp_send_reply 函数：

按照 arp 格式填充数据包并发送即可。其中，当目的 MAC 地址不可知时，写 FF: FF: FF: FF: FF: FF，为广播包，当为 ARP 请求时，Target HW Addr 置空。

（4）查找路由表

路由表的实现已经给出，查找时，只需顺序遍历路由表，返回匹配的网络号最长的表项。若无匹配的网络号的项，返回 NULL。

```
rt_entry_t *longest_prefix_match(u32 dst)
{
    rt_entry_t *rt_entry = NULL, *rt_longest = NULL;
    list_for_each_entry(rt_entry, &rtable, list){
        if((rt_entry->dest & rt_entry->mask) == (dst & rt_entry->mask)) {
            if(!rt_longest || rt_longest->mask < rt_entry->mask)
                rt_longest = rt_entry;
        }
    }
    return rt_longest;
}
```

（5）管理 ARPcache

初始化：框架已给出。

查找：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。访问 arp 表时需要加锁，与 sweep 线程互斥。

```
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;
}
```

插入：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。访问 arp 表时需要加锁，与 sweep 线程互斥。

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。访问 arpreq 表时也需要加锁，与 sweep 线程互斥。

append：遍历 arpreq 表，找到对应 iface 和 ip 的项（若没有则新建一个），把给定的包挂在该项的链表中。发送相应的 arp 请求。

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。

4、实验结果

（1）在单路由器网络上完成 ping 测试:

网络拓扑如图 1。测试结果如图 3-4。

图 3 为 h1 ping r1, h2, h3 的结果，能够 ping 通。

```
alphabet@ubuntu:~/netexp/09/09-router$ sudo python router_topo.py
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=1031 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.115 ms

--- 10.0.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1031ms
rtt min/avg/max/mdev = 0.115/515.926/1031.738/515.812 ms, pipe 2

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=0.157 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.103 ms

--- 10.0.2.22 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1020ms
rtt min/avg/max/mdev = 0.103/0.130/0.157/0.027 ms

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=0.129 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.395 ms

--- 10.0.3.33 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.129/0.262/0.395/0.133 ms
```

图 3

图 4 为 h1 ping 10.0.3.11（ARP 找不到）和 10.0.4.1（路由表找不到）的结果，无法 ping 通。

```
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

--- 10.0.3.11 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1014ms
pipe 2

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

--- 10.0.4.1 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1010ms
```

图 4

（2）在多路由网络上完成 ping 测试和 traceroute 测试。

网络拓扑如图 2。测试结果如图 5。

图 5 显示了 h1 结点 ping r1，r2，h2 的结果和 h1 结点运行 traceroute 的结果。路由器正常运行，结果均符合预期。

```
alphabet@ubuntu:~/netexp/09/09-router$ sudo python traceroute.py
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=1030 ms

--- 10.0.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1030.841/1030.841/1030.841/0.000 ms

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=0.277 ms

--- 10.0.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.277/0.277/0.277/0.000 ms

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=62 time=0.416 ms

--- 10.0.3.33 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.416/0.416/0.416/0.000 ms

traceroute to 10.0.3.33 (10.0.3.33), 30 hops max, 60 byte packets
 1  _gateway (10.0.1.1)  0.273 ms  0.215 ms  0.207 ms
 2  10.0.2.2 (10.0.2.2)  0.323 ms  0.325 ms  0.324 ms
 3  10.0.3.33 (10.0.3.33)  0.411 ms  0.416 ms  0.413 ms
```

图 5

5、问题与解决

（1）对于不同的收发包过程，执行 free(packet)的时机可能不同，必须小心操作以防内存泄漏或多次 free。

（2）遇到了 ip 报头校验和更新错误的问题，在 wireshark 打开 ip 校验功能并抓包后得以确定。

（3）发现自己写的程序比参考程序性能更差，最后确定是打印语句影响运行速度，删除打印语句后，经 iperf 测试，性能强于参考程序。

（4）在配置路由表时遇到问题。最后发现是因为 mininet addLink 的顺序会影响路由器的端口（如 eth0,eth1）连接顺序。

（5）ARP 缓存老化操作时，会有死锁现象。怎么办？我的解决办法是，从发送最初的 arp 请求到发送 5 次超出上限经过 5s，时间较短，可认为网络拓扑没有发生变化，故 mac 地址和 ip 地址只需根据一开始收到的包原路返回即可，不必查找 arp 表，也就避免了死锁。