实验报告

路由器转发实验

一、实验内容

了解路由器转发的原理，包括路由表结构和最长前缀匹配查找方法。了解路由器转发数据包的流程，实现路由器查询和转发模块。了解根据 IP 查询 MAC 地址的方法，即 ARP 协议内容。学习如何构建 ARP 缓存表以及如何实现 ARP 缓存机制。了解 ARP 协议报文格式，实现处理 ARP 请求和应答。最后了解 ICMP 协议格式，在路由器遇到路由表查找失败、ARP 查询失败、TTL 为 0 时能发送 ICMP 报文，并且支持处理 ping 请求报文。

二、实验流程

1. 实现一个具有处理 ARP 请求和应答、ARP 缓存管理、IP 地址查找和 IP 数据包转发、发送和处理 ICMP 数据包功能的路由器。

2. 在给定拓扑下进行测试，进行 ping 实验，验证 ARP 处理功能、IP 报文转发功能、以及 ICMP Destination Host Unreachable 和 Destination Net Unreachable 功能。

3. 构建多个路由器节点的复杂拓扑，通过连通性测试，进一步验证 ARP 处理功能、IP 报文转发功能；通过路径测试，验证 ICMP 功能。

三、实验结果及分析

（一）路由器设计思路

1、IP 数据包转发

本实验中路由器将收到的报文通过以太网帧的 ether_type 分为两类，ARP 报文和 IP 报文，分别处理。

对于 IP 报文，有一种情况需要特别处理，那就是该报文是 ICMP 协议的 ECHOREQUEST（请求回复）报文并且其目标 ip 地址为该端口的 ip 地址。此时不需要转发数据包，而是需要向源地址发送一个 ICMP 回复报文。

而对于其他情况的 IP 数据包（包括其他 ICMP 报文），正常转发即可。正常转发的流程中有 2 种特殊情况，路由表查找失败、TTL 为 0，此时无法转发，向源发送相应 ICMP 的报错报文。

还有需要注意的一点是，路由表查询成功时，如果下一跳网关是 0，意味着目的主机在本网络内，此时直接向目的 IP 转发数据包；否则向下一跳网关转发数据包。

2、路由表查询

路由表查询采用最长前缀匹配查找方法，具体实现如下。

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

遍历整个路由表，当地址匹配且 mask 大于当前最大 mask 时，更新目标表项和最大 mask。注意到有一个特殊的判断条件，rtb->mask==0 && max_mask==0，此时也更新。这是为了处理默认路由，默认路由其 mask 为全 0，如果只在 rtb->mask > max_mask 时更新，对于只有默认路由匹配的情况，则会查询失败。

3、ARP 缓存机制

在 IP 数据包处理模块找到下一跳目的 IP 后，调用 iface_send_packet_by_arp 完成转发。这一模块先完成从目的 IP 到目的 MAC 地址的转换，如果在 arpcache 里查询到映射，完成转发；没有查到，把该数据包挂起到 ARP 等待队列。如果已经向该目的地址发出 ARP 请求，那么只用把该数据包添加到该 IP 对应的队列中；如果没有还没有发过请求，新建一个该 IP 对应的等待队列，然后向目的 IP 发一个 ARP 查询请求。

另一方面是使用一个 sweep 线程来维护 ARP 缓存。每秒钟，运行一次 arpcache_sweep 操作，遍历整个 ARP 等待队列和 ARP 缓存条目。如果一个缓存条目在缓存中已存在超过了 15 秒，将该条目清除。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。

4、ARP 报文处理

接收到的 ARP 报文分为 2 种，请求和回复。对于请求报文，当其请求的目的 IP 为该端口地址时，回复一个 ARP 回应报文，并且把源的 IP 和 MAC 映射关系加入 ARP 缓存。对于回应报文，当其回复的目的 IP 为该端口地址时，说明我们前面发出的一个 ARP 请求得到了回复，此时把源的 IP 和 MAC 映射关系加入 ARP 缓存。

上面两种加入 ARP 缓存的操作后，都需要访问 ARP 等待队列，查看有没有正在等待该条映射的数据包。如果有，把这些数据包依次填写目的 MAC 地址，转发出去，并删除掉相应缓存数据包。

这里需要注意一点是，在把映射关系加入 ARP 缓存时，需要查重。

```
// if the mapping of ip to mac already exist, update
for (int i=0; i<MAX_ARP_SIZE; i++) {
    if (arpcache.entries[i].valid == 1 && arpcache.entries[i].ip4 == ip4) {
        memcpy(arpcache.entries[i].mac, mac, ETH_ALEN);
        arpcache.entries[i].added = time(NULL);
        pthread_mutex_unlock(&arpcache.lock);
        return;
    }
}
```

遍历整个缓存条目表，如果有 IP 相同的条目，更新其添加时间和 MAC 地址。此时就不需要再访问 ARP 等待队列了，因为本身映射关系就在表中，肯定没有等待该条映射的数据包。

5、ICMP 报文与死锁处理

icmp_send_packet 函数中需要我们构建 ICMP 报文并发送。此时发送的目标 IP 地址即是发来这条出错数据包的源 IP 地址。然后端口根据目的 IP 查找路由表可得。那么 MAC 地址呢，一个自然的想法是根据目的 IP 查找 ARP 缓存，但是这样就出现了死锁问题。

ARP sweep 进程执行开始时会获取 ARP 缓存互斥锁，遇到重发 ARP 请求超过 5 次的数据包，需要发送 ICMP 不可达消息。如果发送 ICMP 时又去查找 ARP 表，就需要再次申请互斥锁，导致死锁。

讲义上给了一种方法，将 sweep 进程分为两步，先把要发送 ICMP 消息的数据包拿出来，放到一个临时链表中，然后释放互斥锁。然后对临时链表中每个数据包，发送 ICMP 消息。这个方法功能上没有问题，但是实现比较麻烦，并且效率不高。如果发送 ICMP 消息时，查询 ARP 没有查到，需要再次添加到等待队列并且发送 ARP 请求，函数调用链冗长。

我们仔细思考一下，向本来的源地址发 ICMP 消息时，有必要通过 ARP 查询 MAC 地址吗。其实没有必要，我们既然能从原始数据包中提取源 IP 地址，那么自然也能提取源 MAC 地址，这个映射由发来的数据自然携带，并且是可靠的。

```
struct ether_header *eh = (struct ether_header *)packet;
memcpy(eh->ether_dhost, in_eh->ether_shost, ETH_ALEN);
memcpy(eh->ether_shost, in_eh->ether_dhost, ETH_ALEN);
eh->ether_type = htons(ETH_P_IP);
```

如上图代码所示，直接调换原始数据包中 ether_header 的源、目的 MAC 地址，填入 ICMP 包的 ether_header。经过测试，对于路由表查找失败、TTL 为 0、ping 回应的 ICMP 数据包发送都正确无误，而对于 ARP 查询失败却存在一些问题。

这个其实是框架的问题，对于前三者，在构建 ICMP 数据包时，其原始数据包没有被修改、被破坏。而对于 ARP 查询失败的数据包，它首先调用了 iface_send_packet_by_arp 函数，然后再被挂到 ARP 等待队列。而

iface_send_packet_by_arp 函数做了什么呢？

```
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);
```

我们可以看到，iface_send_packet_by_arp 函数一进来就把原始数据包里的 ether_header 的源 MAC 地址给换成了目的端口的 MAC 地址，这就导致了源 MAC 地址被破坏。这个做法本身没有什么问题，毕竟不管是查到 ARP 映射直接转发，还是等待映射之后再转发，这个源 MAC 地址都要改成发出时端口的 MAC 地址。但是如果我们之后还要利用原始数据包的信息，这个修改就显得太早了。

解决方法也很简单，等到实际要发送数据包时，我们再修改 ether_header 的源 MAC 地址就行。

```
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;

    u8 dst_mac[ETH_ALEN];
    int found = arpcache_lookup(dst_ip, dst_mac);
    if (found) {
        // log(DEBUG, "found the mac of %x, send this packet", dst_ip);
        memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
        eh->ether_type = htons(ETH_P_IP);
        memcpy(eh->ether_dhost, dst_mac, ETH_ALEN);
        iface_send_packet(iface, packet, len);
        free(packet);
    }
```

ARP 等待队列那里转发数据包时也作相应修改，这里就不贴代码了。

修改后测试，对于 4 种情况的 ICMP 数据包发送都正确无误。

这种方法相比讲义的方法更简单，ICMP 发送过程避免了繁琐的 ARP 函数调用，更加高效。

（二）给定拓扑下 ping 测试

1、Ping 10.0.1.1 (r1)

Ping R1 端口，结果如下，可以 ping 通。

"Node: h1"

```
root@joker-linux:/mnt/hgfs/share/09-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=0.241 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.154 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.561 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.207 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3041ms
rtt min/avg/max/mdev = 0.154/0.290/0.561/0.159 ms
```

2、Ping 10.0.2.22 (h2)

Ping h2 主机，结果如下，可以 ping 通。

```
root@joker-linux:/mnt/hgfs/share/09-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=0.255 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.164 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.157 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.157 ms

--- 10.0.2.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3083ms
rtt min/avg/max/mdev = 0.157/0.183/0.255/0.041 ms
```

3、Ping 10.0.3.33 (h3)

Ping h3 主机，结果如下，可以 ping 通。

```
root@joker-linux:/mnt/hgfs/share/09-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=0.243 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.825 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.684 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.158 ms

--- 10.0.3.33 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3019ms
rtt min/avg/max/mdev = 0.158/0.477/0.825/0.283 ms
```

4、Ping 10.0.3.11

Ping 不存在的主机，结果如下，返回 ICMP Destination Host Unreachable。

```
root@joker-linux:/mnt/hgfs/share/09-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 3086ms
pipe 4
```

5、Ping 10.0.4.1

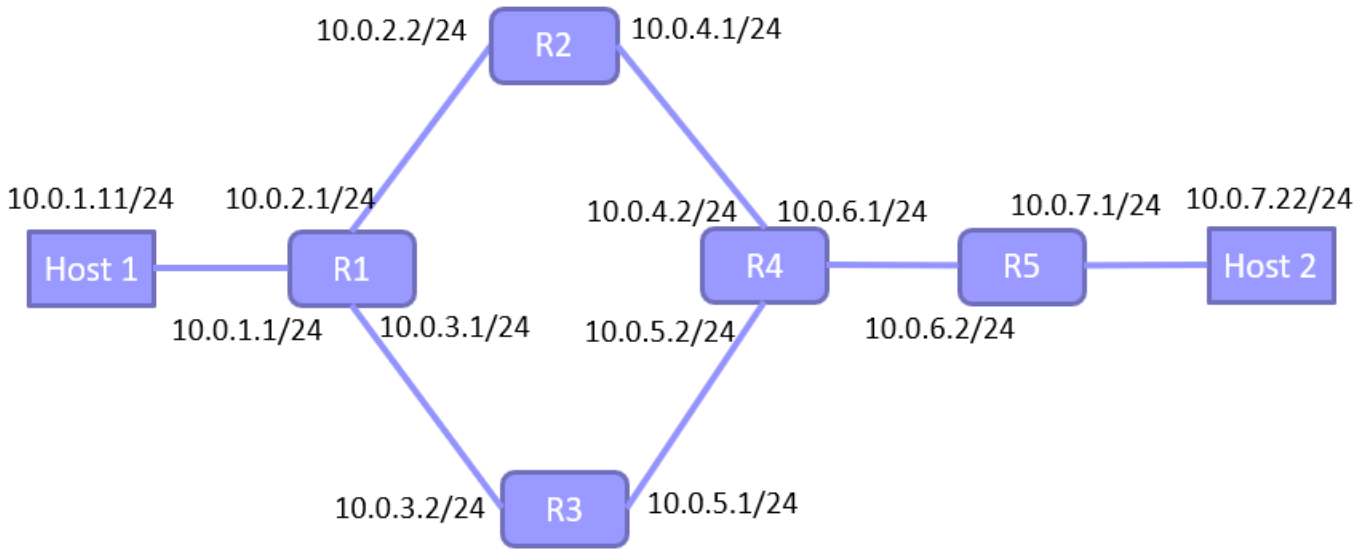
Ping 不存在的网络，结果如下，返回 ICMP Destination Net Unreachable。

```
root@joker-linux:/mnt/hgfs/share/09-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 3067ms
```

（三）多路由器节点的复杂拓扑下测试

1、多路由器节点拓扑结构



2、连通性测试

H1 节点 ping 每个路由器节点的入端口 IP 地址，能够 ping 通。

"Node: h1"

```
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.1.1 -c 2
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=0.353 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.182 ms

--- 10.0.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1018ms
rtt min/avg/max/mdev = 0.182/0.267/0.353/0.085 ms
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.2.2 -c 2
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=1.10 ms
64 bytes from 10.0.2.2: icmp_seq=2 ttl=63 time=0.469 ms

--- 10.0.2.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1014ms
rtt min/avg/max/mdev = 0.469/0.783/1.097/0.314 ms
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.3.2 -c 2
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=63 time=0.556 ms
64 bytes from 10.0.3.2: icmp_seq=2 ttl=63 time=1.66 ms

--- 10.0.3.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1036ms
rtt min/avg/max/mdev = 0.556/1.109/1.663/0.553 ms
```

```
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.4.2 -c 2
PING 10.0.4.2 (10.0.4.2) 56(84) bytes of data.
64 bytes from 10.0.4.2: icmp_seq=1 ttl=62 time=0.590 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=0.694 ms

--- 10.0.4.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1012ms
rtt min/avg/max/mdev = 0.590/0.642/0.694/0.052 ms
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.5.2 -c 2
PING 10.0.5.2 (10.0.5.2) 56(84) bytes of data.
64 bytes from 10.0.4.2: icmp_seq=1 ttl=62 time=0.902 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=0.808 ms

--- 10.0.5.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 0.808/0.855/0.902/0.047 ms
```

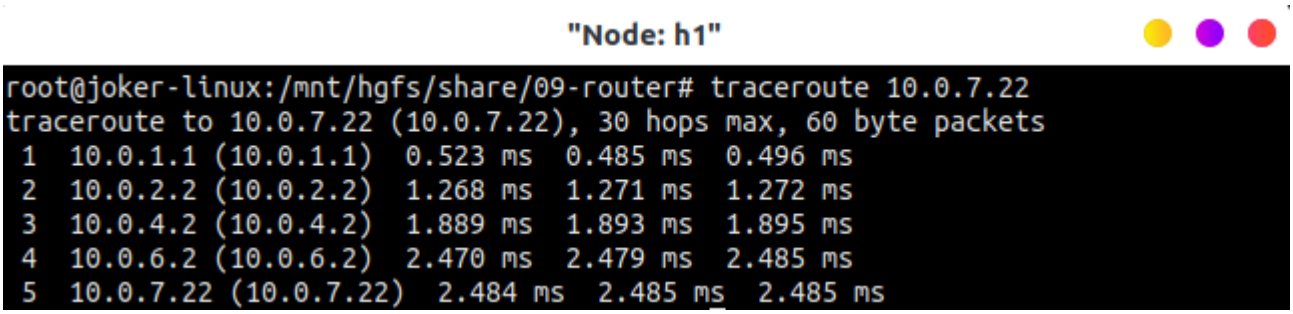
```
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.6.2 -c 2
PING 10.0.6.2 (10.0.6.2) 56(84) bytes of data.
64 bytes from 10.0.6.2: icmp_seq=1 ttl=61 time=0.635 ms
64 bytes from 10.0.6.2: icmp_seq=2 ttl=61 time=0.693 ms

--- 10.0.6.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1023ms
rtt min/avg/max/mdev = 0.635/0.664/0.693/0.029 ms
root@joker-linux:/mnt/hgfs/share/09-router# ping 10.0.7.22 -c 2
PING 10.0.7.22 (10.0.7.22) 56(84) bytes of data.
64 bytes from 10.0.7.22: icmp_seq=1 ttl=60 time=0.760 ms
64 bytes from 10.0.7.22: icmp_seq=2 ttl=60 time=1.34 ms

--- 10.0.7.22 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1008ms
rtt min/avg/max/mdev = 0.760/1.047/1.335/0.287 ms
```

3、路径测试

在一个 H1 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。

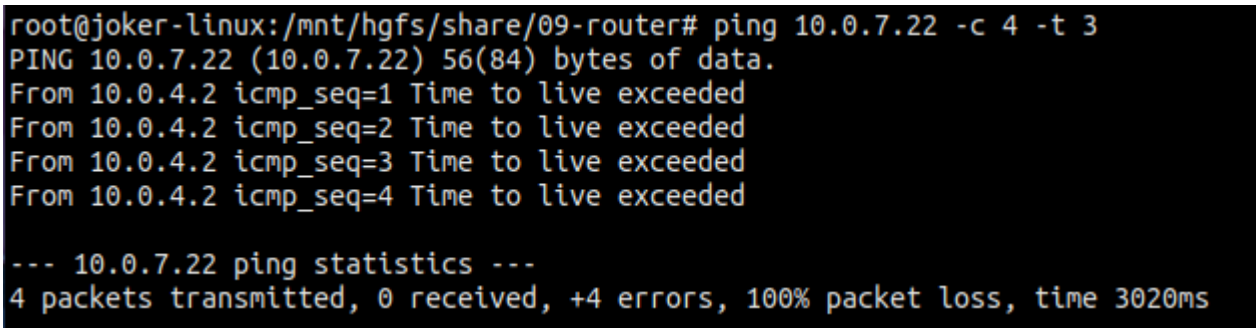


如上图所示，H1 到 H2 路径为：H1 - R1 - R2 - R4 - R5 - H2。

4、ICMP TTL 值减为 0 测试

其实 traceroute 能返回正确结果本身就验证了 TTL 为 0 功能正确，因为 traceroute 是从 1 开始不断增加 TTL 来获取从源到目的地址的每一跳信息。

这里再直接验证一下 TTL 为 0 功能。我们已经知道从 H1 到 H2 共 5 跳路径长度，如果我们使用 TTL 小于 5 的数据包来 ping H2 节点，那么必然中途的路由器会返回 TTL 减为 0 错误。



设置 TTL 为 3，确实收到 Time to live exceeded 的 ICMP 数据包，该功能正确。

四、实验总结

通过本次实验，我了解了路由器转发的原理、路由表结构，并且手动实现了最长前缀匹配查找方法。通过实际实现路由器的查询和转发功能，更加理解了路由器转发数据包的流程。我还学到了如何构建 ARP 缓存表、如何实现 ARP 缓存机制、以及如何处理 ARP 请求和应答，这让我对理论课上讲过的 ARP 协议有了更多的理解。最后我还了解了 ICMP 协议，它在 IP 报文转发中起到了重要的报错、查询功能，也让我更加理解了 ping、traceroute 等功能的原理。