Router 实验设计报告

中国科学院大学

页

2022年4月30日

一、路由器转发实验

# 实验内容

1. 在给定框架的基础上实现具有IP 地址查找、 IP 数据包转发，处理 ARP 请求和应答、ARP 缓存管理发送和发送 ICMP 报文功能的路由器
2. 运⾏给定⽹络拓扑(router\_topo.py)在单路由器网络上完成 ping 测试。
3. 构造⼀个包含多个路由器节点组成的⽹络，在多路由网络上完成 ping 测试和 traceroute 测试

# 实验流程

1）搭建实验环境：

①在主机上安装arptables iptables，用于禁止每个节点的相应功能，

sudo apt install arptables iptables

②安装traceroute，用于输出路径上节点ip信息。

sudo apt install traceroute

2）在理解已有的代码框架的基础上补充完成以下函数，实现对应的功能：

① arp.c：

（1） arp\_send\_request：发送 ARP 请求；

（2） arp\_send\_reply：进⾏ARP 应答；

（3） handle\_arp\_packet：处理 ARP 数据包

②arpcache.c：

（1） arpcache\_lookup：查找 ARP cache 中是否有需要的IP 和 mac地址映射；

（2） arpcache\_append\_packet：查询 ARP cache 失败时，将包挂起并发送 ARP 请求；

（3） arpcache\_insert：插入IP 到 mac 的映射到cache，并发送等待该映射的数据包；

（4） arpcache\_sweep：老化删除已经不具备时效性的表项，处理未收到应答的包。

③icmp.c：

（1） icmp\_send\_packet：发送 ICMP 数据包。

④ip.c：

（1） handle\_ip\_packet：处理IP 数据包。

⑤ip\_base.c：

（1） longest\_prefix\_match：路由表最⻓前缀匹配；

（2） ip\_send\_packet：发送IP数据包。

3）make生成router，修改并运⾏给定⽹络拓扑(router\_topo.py)在单路由器网络上完成 ping 测试。给定的网络拓扑如下图1：

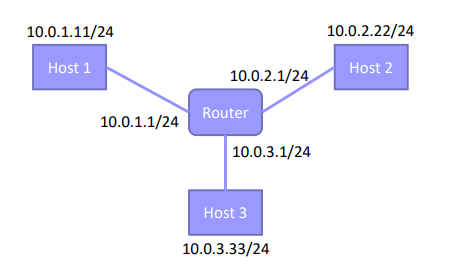


图1

4）构建包含2个 router 、2个 host 的⽹络拓扑结构，在多路由网络上完成 ping 测试和 traceroute 测试。构建拓扑结构如下图2所示：

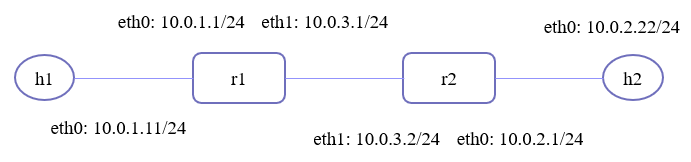


图2

5）攥写实验报告。

# 实验设计

路由器读取所有接口的信息后初始化路由表和arp-cache，然后开始处理从接口文件描

符得到的数据包，路由器会读这个数据包的头部，来判断这是⼀个 IP 数据包还是⼀个 ARP 数据包，然后调⽤ handle\_ip\_packet 函数处理ip数据包，调用handle\_arp\_packet函数处理arp数据包。handle\_ip\_packet 函数和handle\_arp\_packet函数以及它们需要调用的函数即为我们需要实现的实验部分，下文按照函数调用出现先后顺序介绍具体函数实现。

1. **handle\_ip\_packet** 函数（ip,c）

首先提取目的IP地址，如果目的IP地址等于接口IP地址，进一步检查如果报文是ICMP echo-request报文， 则发送ICMP echo-reply报文; 否则需要转发这个报文，先⽤最⻓前缀匹配查找路由表，如果查找成功且⽣存期没有耗尽，则重新计算校验和并转发数据包，否则调⽤icmp\_send\_packet函数向源主机报错。

void handle\_ip\_packet(iface\_info\_t \*iface, char \*packet, int len)

{

*// fprintf(stderr, "TODO: handle ip packet.\n");*

    struct iphdr \*IP\_head = packet\_to\_ip\_hdr(packet);

    u32 dest\_ip = ntohl(IP\_head->daddr);

*if* (dest\_ip == iface->ip)

    {

        struct icmphdr \*Icmp\_head = (struct icmphdr \*)(packet + ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE);

*if* (IP\_head->protocol == IPPROTO\_ICMP && Icmp\_head->type == ICMP\_ECHOREQUEST)

        {

            icmp\_send\_packet(packet, len, ICMP\_ECHOREPLY, 0);

        }

        free(packet);

    }

*else*

    {

        rt\_entry\_t \*rt = longest\_prefix\_match(dest\_ip);

*if* (rt)

        { *//路由表查找成功*

*if* (--IP\_head->ttl <= 0)

            {

                icmp\_send\_packet(packet, len, ICMP\_TIME\_EXCEEDED, ICMP\_EXC\_TTL);

                free(packet);

            }

*else*

            {

                IP\_head->checksum = ip\_checksum(IP\_head);

                ip\_send\_packet(packet, len);

            }

        }

*else*

        {

            icmp\_send\_packet(packet, len, ICMP\_DEST\_UNREACH, ICMP\_NET\_UNREACH);

        }

    }

}

1. **icmp\_send\_packet 函数**（icmp.c）

主要任务是按照格式填充 ICMP 包。首先读取数据包头部查看具体的type，若发送的是 reply，则是在回复 ping ，Rest of ICMP Header 拷贝 Ping 包中的相应字段，否则当

type=ICMP\_TIME\_EXCEEDED 时，说明⽣存期耗尽；当 type=ICMP\_DEST\_UNREACH 时，说明查找路由表时失败。然后根据不同的type和输入的参数按照icmp报文格式配置icmp数据包。 当type不是reply时，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。

void icmp\_send\_packet(const char \*in\_pkt, int len, u8 type, u8 code)

{

*// fprintf(stderr, "TODO: malloc and send icmp packet.\n");*

    struct iphdr \*ihr = packet\_to\_ip\_hdr(in\_pkt); *// in\_pkt的IP首部*

    int pkt\_len = 0;

*if* (type == ICMP\_ECHOREPLY)

        pkt\_len = len;

*else*

        pkt\_len = ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE + IP\_HDR\_SIZE(ihr) + ICMP\_HDR\_SIZE + 8;

    char \*packet = (char\*)malloc(pkt\_len \* sizeof(char));

    struct ether\_header \*eh = (struct ether\_header\*)packet;

    eh->ether\_type = htons(ETH\_P\_IP);

    struct iphdr \*packet\_IPhead = packet\_to\_ip\_hdr(packet);

    rt\_entry\_t \*rt = longest\_prefix\_match(ntohl(ihr->saddr));

    ip\_init\_hdr(packet\_IPhead, rt->iface->ip, ntohl(ihr->saddr), pkt\_len - ETHER\_HDR\_SIZE, 1);

    struct icmphdr \*Icmp\_head = (struct icmphdr\*)(packet + ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE);

    Icmp\_head->type = type;

    Icmp\_head->code = code;

    int Rest\_begin = ETHER\_HDR\_SIZE + IP\_HDR\_SIZE(packet\_IPhead) + 4;

*if* (type == ICMP\_ECHOREPLY)

        memcpy(packet + Rest\_begin, in\_pkt + Rest\_begin, pkt\_len - Rest\_begin);

*else* {

        memset(packet + Rest\_begin, 0, 4);

        memcpy(packet + Rest\_begin + 4, in\_pkt + ETHER\_HDR\_SIZE, IP\_HDR\_SIZE(ihr) + 8);

    }

    Icmp\_head->checksum = icmp\_checksum(Icmp\_head, pkt\_len - ETHER\_HDR\_SIZE - IP\_BASE\_HDR\_SIZE);

    ip\_send\_packet(packet, pkt\_len);

}

1. **ip\_send\_packet 函数**（ip\_base.c）

用于发送路由器依据 IP 相关协议生成的报文，如果网关不为0，该路由器任何端口IP都与目的IP不在同一网段，向默认网关发包。否则两台主机在同一网段，直接向目的ip发地址发包。

void ip\_send\_packet(char \*packet, int len)

{

*// fprintf(stderr, "TODO: send ip packet.\n");*

    struct iphdr \*pkt\_IPhead = packet\_to\_ip\_hdr(packet);

    u32 dst\_ip = ntohl(pkt\_IPhead->daddr);

    rt\_entry\_t \*rt = longest\_prefix\_match(dst\_ip);

*if* (rt->gw) *// 该路由器任何端口IP都与目的IP不在同一网段*

        iface\_send\_packet\_by\_arp(rt->iface, rt->gw, packet, len);

*else*

        iface\_send\_packet\_by\_arp(rt->iface, dst\_ip, packet, len);

}

1. **longest\_prefix\_match函数**（ip\_base.c）

因为子网掩码的缘故，我们匹配路由表在查找时需顺序遍历路由表，返回匹配的网络号最长的表项。

rt\_entry\_t \*longest\_prefix\_match(u32 dst)

{

*// fprintf(stderr, "TODO: longest prefix match for the packet.\n");*

    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;

}

iface\_send\_packet\_by\_arp 函数在arpcache中查找dst\_ip的mac地址。 如果找到，填充以太网头，通过iface\_send\_packet发送此报文，否则，将此报文挂起到arpcache中，发送arp请求。其中需要使用管理arpcache的函数 arpcache\_lookup 和arpcache\_append\_packet。

1. **arpcache\_lookup函数** （arpcache.c）

遍历表，查找是否有一个具有相同IP和mac地址的表项，如果查找到了返回1，否则返回0。

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;

}

1. **arpcache\_append\_packet 函数**（arpcache.c）

当查询 ARP cache 失败时调⽤将要发送的包挂起，并且发送 ARP 请求。在arp-cache待处理数据包的列表中查找，如果已经有一个具有相同IP地址和iface的条目(这意味着对应的arp请求已经发出)，只需将这个数据包附加到该条目的尾部(该条目可能包含多个数据包);否则，malloc指定IP地址和iface的新表项，附加报文，发送arp请求。

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");*

    struct arp\_req \*entry = NULL;

    struct cached\_pkt \*cached\_packet = (struct cached\_pkt\*)malloc(sizeof(struct cached\_pkt));

    cached\_packet->packet = packet;

    cached\_packet->len = len;

    int found=0;

    pthread\_mutex\_lock(&arpcache.lock);

    list\_for\_each\_entry(entry, &(arpcache.req\_list), list){

*if* (entry->ip4 == ip4 && entry->iface == iface){

            found = 1;

*break*;

        }

    }

*if* (found)

        list\_add\_tail(&(cached\_packet->list), &(entry->cached\_packets));

*else*{

        struct arp\_req \*req = (struct arp\_req\*)malloc(sizeof(struct arp\_req));

        req->iface = iface;

        req->ip4 = ip4;

        req->sent = time(NULL);

        req->retries = 1;

        init\_list\_head(&(req->cached\_packets));

        list\_add\_tail(&(cached\_packet->list), &(req->cached\_packets));

        list\_add\_tail(&(req->list), &(arpcache.req\_list));

        arp\_send\_request(iface, ip4);

    }

    pthread\_mutex\_unlock(&arpcache.lock);

}

到此icmp数据包的处理完毕，接下来回头看如何利用handle\_arp\_packet 函数处理ip数据包。

1. **handle\_arp\_packet函数**（arp.c）

当目的IP地址等于接口IP地址，如果报文时Request报文，回复⾃⼰的mac地址给源主机，如果时Reply报文说明自己的请求得到回应将信息源主机信息添加到自己的arp-cache中。 否则，丢弃这个报文。

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 \* ether\_arp\_pkt = (struct ether\_arp \*)(packet + ETHER\_HDR\_SIZE);

*// 判断⽬的地址是否为本端⼝*

*if* (ntohl(ether\_arp\_pkt->arp\_tpa) == iface->ip){

*// 请求报⽂得到回应，要保存到ARP缓存中*

*if* (ntohs(ether\_arp\_pkt->arp\_op) == ARPOP\_REPLY){

            arpcache\_insert(ntohl(ether\_arp\_pkt->arp\_spa), ether\_arp\_pkt->arp\_sha);

        }

*else* *if* (ntohs(ether\_arp\_pkt->arp\_op) == ARPOP\_REQUEST){

            arp\_send\_reply(iface, ether\_arp\_pkt);

        }

    }

}

1. **arpcache\_insert函数**（arpcache.c）

在arp-cache中插入IP->mac映射，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，如果缓存已满（32），随机选一个替换出去如果有等待这个映射的数据包，填充每个数据包的以太头，然后发送出去。insert和seweep修改 arp 表时需要加锁互斥访问。

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 index = rand() % 32;*// 随机⽣成的0～31的整数值,没找到无效项则对这一项覆盖*

*for* (int i=0; i<MAX\_ARP\_SIZE; i++){

*// valid==0说明表项⽆效，ip只对应一个mac*

*if* (!arpcache.entries[i].valid || arpcache.entries[i].ip4 == ip4){

            index = i;

*break*;

        }

    }

    arpcache.entries[index].ip4 = ip4;

    memcpy(arpcache.entries[index].mac, mac, ETH\_ALEN);

    arpcache.entries[index].added = time(NULL);

    arpcache.entries[index].valid = 1;

    struct arp\_req \*entry = NULL;

    struct arp\_req \*entry\_next = NULL;

    list\_for\_each\_entry\_safe(entry, entry\_next, &(arpcache.req\_list), list) {

*if* (entry->ip4 == ip4){

            struct cached\_pkt \*pkt = NULL;

            struct cached\_pkt \*pkt\_next;

            list\_for\_each\_entry\_safe(pkt, pkt\_next, &(entry->cached\_packets), list){

                memcpy(pkt->packet, mac, ETH\_ALEN);

                iface\_send\_packet(entry->iface, pkt->packet, pkt->len);

                free(pkt);

            }

            list\_delete\_entry(&(entry->list));

            free(entry);

        }

    }

    pthread\_mutex\_unlock(&arpcache.lock);

}

9）**arp\_send\_request和arp\_send\_reply函数**（arp.c）

按照ARP报文格式填充数据包并发送即可。其中当目的 MAC 地址不可知时，发送ARP 请求MAC地址写 FF: FF: FF: FF: FF: FF广播请求。

# 实验结果及分析

**（1）单路由器网络上完成 ping 测试**

脚本router\_topo.py增加测试语句如下所示：

    net.start()

    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'))

*#CLI(net)*

net.stop()

得到h1 ping r1, h2, h3能够 ping 通，结果如下图3所示，图4为 h1 ping 10.0.3.11（主机不可达）和 10.0.4.1（网络不可达）的结果，无法ping 通。

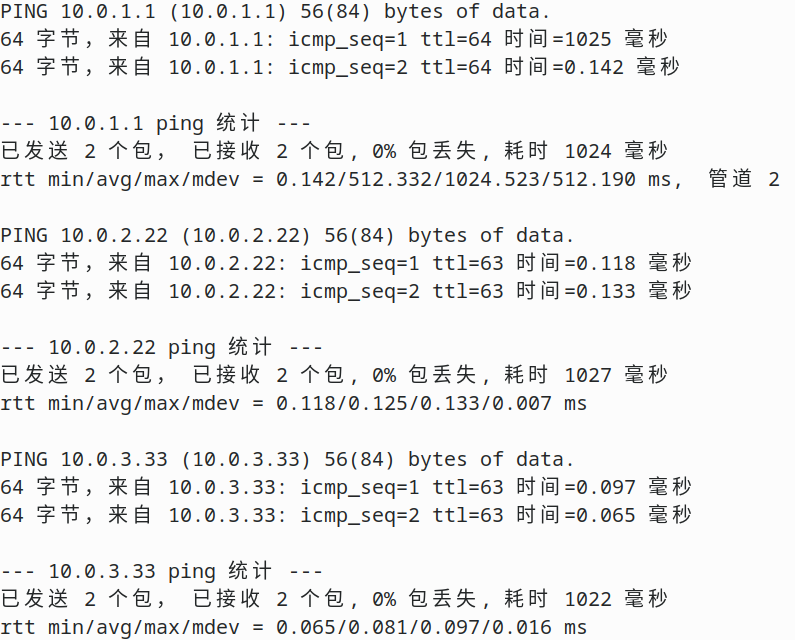


图3

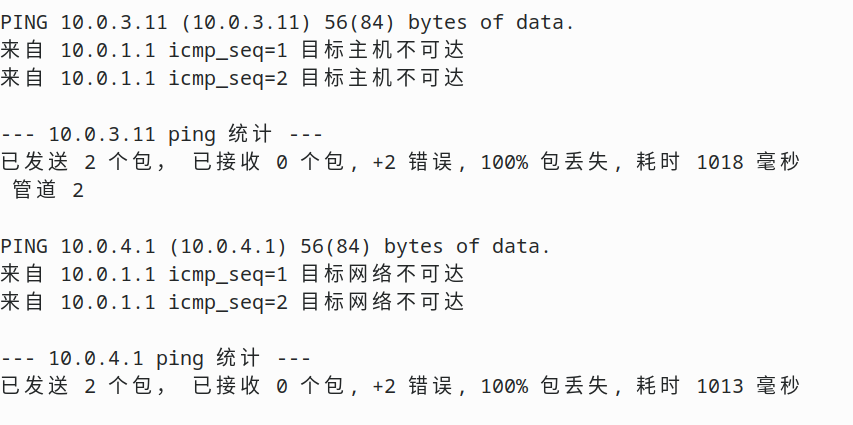


图4

**（2）在多路由网络上完成 ping 测试和 traceroute 测试**

h1 结点 ping r1, r2, h2 的结果和 h1 结点运行 traceroute 的结果如图5所示。结果显示连通性正常。另外traceroute能够正确输出路径上每个节点的IP信息，对应图2拓扑中h1->r1(eth0)->r1(eth1)->r2(eth1)->r2(eth0)->h2的线路信息，符合预期结果。

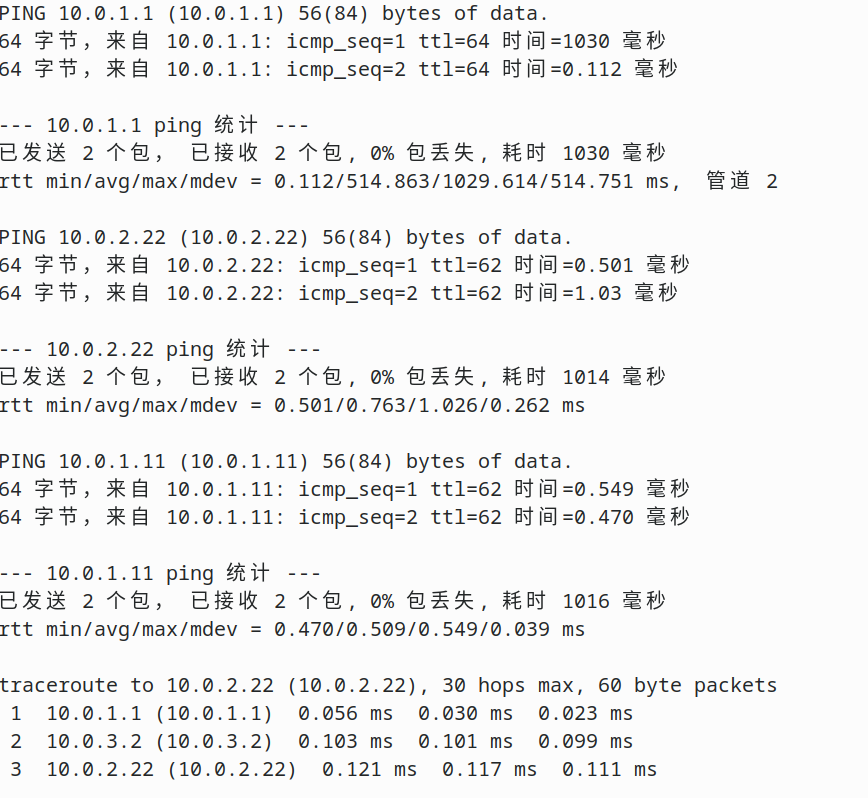


图5

# 实验总结

通过本次实验具体实现路由器，切身理解图6中TCP协议网络层的IP、ARP和ICMP的具体工作流程和它们的作用，对数据包的具体转发过程有了十分清晰深刻的认识。另外，在自己构建拓扑的时候ping失败过，trouceroute也打出了如图7所示奇怪的结果，探索一下发现改变addLink 的连接顺序会改变路由器的端口（如 eth0,eth1）连接顺序，因此并不是能随意连接的，同时也再次体会到了python简洁的同时要默默遵守一些类似语法块时由缩进决定的特别性质。

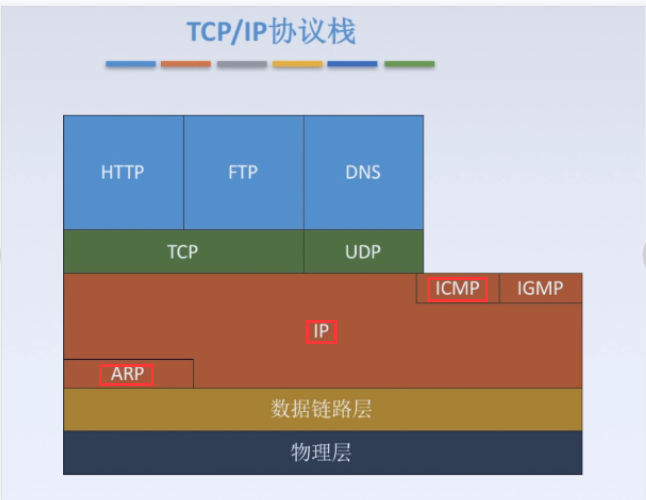


图6

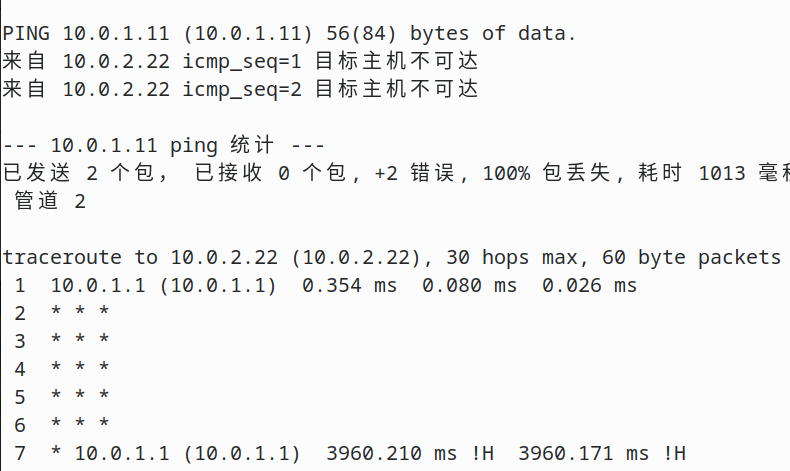


图7

**参考资料**：

1. [一个完整的网络数据包转发过程](https://blog.csdn.net/qq_38668258/article/details/84987881)。
2. [彻底搞懂系列之：ARP协议](https://zhuanlan.zhihu.com/p/370507243)。