计算机网络实验 7 报告

学号　2017K8009929059
姓名　於修远

路由器转发实验

一、实验内容

- 安装 arptables 和 iptables 以禁止每个节点的固有功能；安装 traceout 以实现路径测试。
- 编写并实现路由器的 IP 包处理、ICMP 包生成、ARP 包生成处理、ARP 缓存维护等路由器功能。
- 运行给定网络拓扑` router_topo.py `，验证 ping 功能在各种情况下均处理正常。
- 构造包含多个路由器节点的网络拓扑，手动配置路由表，并测试连通性，完成路径测试。

二、实验流程

（一）IP 数据包处理

1、IP 包分析

对于接收到的 IP 数据包，调用` handle_ip_packet `函数进行处理。第一步是提取 IP 数据包的目的地址，如果恰为目的 IP 为自身的数据包，则进行应答处理。在本实验中，仅支持 ping_request 包的应答，即生成并发送该包对应的 ICMP 的 ping_echo 包。由于每次收包后都开辟空间进行分析管理，所以处理完成后需要释放接受包的空间，否则会造成巨大的内存占用。

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    // TODO: handle ip packet

    struct iphdr* hdr = packet_to_ip_hdr(packet);
    u32 pip = iface->ip, dip = ntohl(hdr->daddr);
    if (pip == dip) {
        struct icmphdr* echo = (struct icmphdr*)(IP_DATA(hdr));
        if (echo->type == ICMP_ECHOREQUEST)
            icmp_send_packet(packet, len, ICMP_ECHOREPLY, code: 0);
        free(packet);
    } else {...}
}
```

对于目的结点不是本结点的情况，需要依次检查路由表条目、数据包 TTL 值，在必要时回复 ICMP 的 NET_UNREACH 或 EXC_TTL 信息，并结束处理。通过所有检查后由于更新了 TTL 值，还需要相应更新校验和，再通过 ARP 机制向下一结点发出。值得注意的是在路由器端口与目的地址不在同一网段时，应以路由表条目中的对应 IP 地址作为下一跳地址；否则应直接向在同一网段的目的 IP 发送即可。

```
rt_entry_t* entry = longest_prefix_match(dip);
if (!entry) {
    icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
    free(packet);
} else {
    (hdr->ttl)--;
    if (hdr->ttl == 0) {
        icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL);
        free(packet);
    } else {
        u32 dst_ip = entry->gw;
        if (!dst_ip) dst_ip = dip;
        hdr->checksum = ip_checksum(hdr);
        iface_send_packet_by_arp(entry->iface, dst_ip, packet, len);
    }
}
```

2、最长前缀匹配

在路由表查询过程中，需要遵循最长前缀匹配原则。考虑到路由表表项存储可能是无规律的，所以需要遍历路由表，过程中暂存包含最长匹配前缀的条目，初始最大匹配长度设为 0。每次比对时，若被比对项的掩码位数不大于已匹配位数，则直接跳过。

```
rt_entry_t *longest_prefix_match(u32 dst)
{
    // TODO: longest prefix match for the packet
    rt_entry_t *result = NULL, *entry;
    u32 result_mask = 0;
    list_for_each_entry(entry, head: &rtable, list) {
        if (entry->mask > result_mask) {
            u32 ip_mask = entry->mask & dst;
            if (ip_mask == (entry->dest & entry->mask)) {
                result = entry;
                result_mask = entry->mask;
            }
        }
    }
    return result;
}
```

（二）ICMP 数据包生成

1、ICMP 包构建

考虑到 ICMP 包的许多内容需要重新填写，所以需要额外申请存储空间。对于非 ping_echo 的 ICMP 数据包，数据长度可以用以太网头+IP 头+ICMP 头+原数据包 IP 头+拷贝数据长度进行计算；对于 ping_echo 包，这一计算可以简化为原数据包长度减去 IP 头长度，再加上新 IP 头长度，这是因为 ping_echo 包与 ping_request 包结构基本一致，只可能在 IP 头部分出现长度不一的情况。

```
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    // TODO: malloc and send icmp packet.
    //struct ether_header *in_eh = (struct ether_header *)in_pkt;
    struct iphdr *in_ih = packet_to_ip_hdr(in_pkt);
    int out_len;
    if (type == ICMP_ECHOREPLY) out_len = len - IP_HDR_SIZE(in_ih) + IP_BASE_HDR_SIZE;
    else out_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + ICMP_HDR_SIZE + IP_HDR_SIZE(in_ih) + ICMP_COPIED_DATA_LEN;
    char* out_pkt = (char*)malloc( size: out_len * sizeof(char));
```

开辟 IP 数据包空间后需填写 IP 和 ICMP 信息。IP 头填写过程中源 IP 地址暂时留空，发出前查询端口时一并填写；目的 IP 地址则填写原数据包的源 IP 地址。ICMP 头部则根据传入数据，填写 ICMP 类型码。

```
struct iphdr *out_ih = packet_to_ip_hdr(out_pkt);
ip_init_hdr(out_ih,  saddr: 0, ntohl(in_ih->saddr),  len: out_len - ETHER_HDR_SIZE,  proto: 1);

struct icmphdr *out_ich = (struct icmphdr*)(IP_DATA(out_ih));
out_ich->type = type;
out_ich->code = code;
```

最后，ICMP 包剩余部分可以直接从原数据包中拷贝。ping_echo 包直接拷贝 ping_request 包的对应内容，非 ping_echo 包则拷贝原数据包数据部分的前 8 字节。另外，非 ping_echo 包还需要将 ICMP 检验和后的四字节置 0，对应于` icmphdr `结构体中的联合体` u `字段。完成信息拷贝后计算并填入校验和，即可准备发送。

```
char *out_rest = (char*)out_ich + ICMP_HDR_SIZE;
if (type == ICMP_ECHOREPLY) {
    int rest_offset = ETHER_HDR_SIZE + IP_HDR_SIZE(in_ih) + 4;
    memcpy( dest: out_rest - 4,  src: in_pkt + rest_offset,  n: len - rest_offset);
} else {
    memset(&out_ich->u,  c: 0,  n: 4);
    memcpy(out_rest, in_ih,  n: IP_HDR_SIZE(in_ih) + ICMP_COPIED_DATA_LEN);
}
out_ich->checksum = icmp_checksum(out_ich,  len: out_len - ETHER_HDR_SIZE - IP_BASE_HDR_SIZE);

ip_send_packet(out_pkt, out_len);
```

2、ICMP 包发出

发包前需要先根据先前填写的目的 IP 查找路由表，确定发送端口。由于在先前组建数据包时没有填写源 IP 地址，在这里也把查找得到的发送端口的 IP 信息一并填入，并计算校验和。选择下一跳 IP 的过程与直接处理 IP 包的过程类似，故不赘述。最后同样通过` iface_send_packet_by_arp `函数的 ARP 机制发出。

```
void ip_send_packet(char *packet, int len)
{
    // TODO: send ip packet
    struct iphdr* ih = packet_to_ip_hdr(packet);
    u32 dst = ntohl(ih->daddr);
    rt_entry_t* entry = longest_prefix_match(dst);
    u32 saddr = entry->iface->ip;
    ih->saddr = htonl(saddr);
    ih->checksum = ip_checksum(ih);
    u32 next_ip = entry->gw;
    if (!next_ip) next_ip = dst;
    iface_send_packet_by_arp(entry->iface, next_ip, packet, len);
}
```

（三）ARP 数据包处理

1、ARP 包分析

对于接受到的 ARP 包，首先根据目的 IP 地址筛选，仅处理目的 IP 为本端口 IP 的 ARP 包。对于 ARP 请求，调用函数发送应答；对于 ARP 应答，调用函数将接收到的映射关系插入 ARP 缓存的 IP-mac 映射表中。

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

2、ARP 包生成

ARP 应答包和请求包的构造过程相似。由于 ARP 包的规格固定，首先可以申请固定长度的空间，填写以太网头部和 ARP 包的部分共通信息。由于请求包是广播，应答包是单播，所以二者的目的 mac 地址填写不同：在以太网头部中，前者为各位全 1（通过 memset 置各字节为-1），后者则是特定的目的 mac 地址；在 ARP 信息中，前者改为各位全 0（通过 memset 置各字节为 0）。ARP 包直接通过端口发送，不再涉及 ARP 查询等缓存相关操作。

```
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    // TODO: send arp request when lookup failed in arpcache.
    int len = ETHER_HDR_SIZE + sizeof(struct ether_arp);
    char* packet = (char*)malloc( size: len * sizeof(char));
    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    memset(eh->ether_dhost,  c: -1, ETH_ALEN);
    eh->ether_type = htons(ETH_P_ARP);
    struct ether_arp* ah = (struct ether_arp*)(packet + ETHER_HDR_SIZE);
    arp_init_hdr(ah, iface, ARPOP_REQUEST, dst_ip);
    memset(ah->arp_tha,  c: 0, ETH_ALEN);
    iface_send_packet(iface, packet, len);
}
```

（四）ARP 缓存维护

1、ARP 缓存机制

所有 IP 数据包的发送都经由 ARP 缓存机制的管理。若 ARP 缓存表中存在有效匹配条目，直接填写以太网头部并发送；否则，将数据包加入缓存表的数据包缓存中，并发送 ARP 请求。这部分是实验固有代码，未作出修改，故不赘述。由于存在一个缓存清理进程，所有 ARP 缓存相关操作都需要在互斥锁内完成。

2、ARP 缓存查找

ARP 缓存查找在数据包组建完成后、准备发送前进行。遍历缓存表中的所有有效映射条目，检查其中存储的 IP 项是否为需要查找的 IP，查找成功后填写其对应的下一跳 mac 地址。返回值表示查找情况。

```
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    // TODO: lookup ip address in arp cache
    pthread_mutex_lock(&arpcache.lock);

    struct arp_cache_entry* entry;
    for (int i = 0; i < MAX_ARP_SIZE; ++i) {
        entry = &arpcache.entries[i];
        if (!entry->valid) continue;
        if (entry->ip4 == ip4) {
            memcpy(mac, entry->mac, ETH_ALEN);
            pthread_mutex_unlock(&arpcache.lock);
            return 1;
        }
    }

    pthread_mutex_unlock(&arpcache.lock);
    return 0;
}
```

3、添加待应答数据包

新增等待 ARP 应答的数据包时，应申请 ARP 缓存的数据包条目空间，并填入该数据包的指针和长度等信息。若该数据包请求的 IP-mac 映射已发送，即存在该映射关系的等待队列，则直接将新条目加入队尾。

```
pthread_mutex_lock(&arpcache.lock);

struct cached_pkt* new_pkt = (struct cached_pkt*)malloc(sizeof(struct cached_pkt));
new_pkt->packet = packet;
new_pkt->len = len;

struct arp_req *req_entry = NULL;
list_for_each_entry(req_entry, head: &(arpcache.req_list), list) {
    if (req_entry->ip4 == ip4 && req_entry->iface == iface) {
        list_add_tail(&new_pkt->list, &req_entry->cached_packets);
        pthread_mutex_unlock(&arpcache.lock);
        return;
    }
}
```

若该条目对应一个新的申请队列，则需要相应地开辟新的队列标识，加入等待响应的映射表中。

```
struct arp_req* new_req = (struct arp_req*)malloc(sizeof(struct arp_req));
new_req->iface = iface;
new_req->ip4 = ip4;
new_req->retries = ARP_REQUEST_MAX_RETRIES;
init_list_head(&new_req->cached_packets);
list_add_tail(&new_pkt->list, &new_req->cached_packets);
list_add_tail(&new_req->list, &arpcache.req_list);
arp_send_request(iface, ip4);
new_req->sent = time( timer: NULL);

pthread_mutex_unlock(&arpcache.lock);
```

4、添加映射表条目

接收到格式正确的 ARP 响应后需要将信息填入映射表中，优先在无效条目上覆盖填写，若条目已满则根据当前时间戳模 32 来替换条目，与以时间做种获取随机数的效果基本一致。

```
struct arp_cache_entry* entry = NULL;
for (int i = 0; i < MAX_ARP_SIZE; ++i) {
    if (!arpcache.entries[i].valid) {
        entry = &arpcache.entries[i];
        break;
    }
}

time_t now = time( timer: NULL);
if (!entry) {
    int idx = now % 32;
    entry = &arpcache.entries[idx];
}
entry->ip4 = ip4;
entry->added = now;
entry->valid = 1;
memcpy(entry->mac, mac, ETH_ALEN);
```

完成条目填写后，检查等待应答的数据包队列。对于正在等候本条映射关系的队列，遍历队列中所有的条目，逐一发出并删除条目。最后删除本条等待队列，即完成了映射条目的插入。

```
struct arp_req *req_entry = NULL, *req_q;
list_for_each_entry_safe(req_entry, req_q,  head: &(arpcache.req_list), list) {
    if (req_entry->ip4 == ip4) {
        struct cached_pkt *pkt_entry = NULL, *pkt_q;
        list_for_each_entry_safe(pkt_entry, pkt_q,  head: &(req_entry->cached_packets), list) {
            struct ether_header *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);
        }
        list_delete_entry(&(req_entry->list));
        free(req_entry);
        break;
    }
}
```

5、检查清理 ARP 缓存

清理 ARP 缓存对应于其结构的两个部分，清理进程每秒唤醒一次。首先检查 IP-mac 地址映射，对于留存时间超过 15 秒的条目，设置为无效条目。

```
time_t now = time( timer: NULL);
for (int i = 0; i < MAX_ARP_SIZE; ++i) {
    struct arp_cache_entry* entry = &arpcache.entries[i];
    entry->valid = (now - entry->added < ARP_ENTRY_TIMEOUT);
}
```

对于等待应答的数据包队列组，将队首信息中的剩余发送次数逐个减 1 并重发 ARP 请求。对于减 1 后重发次数小于 0 的组次不重发 ARP 请求，而是直接遍历队列，对每一个包的源 IP 地址回复一个 ICMP_HOST_UNREACHABE 包，随后释放队列及其中的每一个节点。

```
struct arp_req *req_entry = NULL, *req_q;
list_for_each_entry_safe(req_entry, req_q, head: &(arpcache.req_list), list) {
    now = time( timer: NULL);
    if (now - req_entry->sent > 1) {
        if (--(req_entry->retries) > 0) {
            arp_send_request(req_entry->iface, req_entry->ip4);
            req_entry->sent = now;
        } else {
            struct cached_pkt *pkt_entry = NULL, *pkt_q;
            list_for_each_entry_safe(pkt_entry, pkt_q, head: &(req_entry->cached_packets), list) {
                list_delete_entry(&(pkt_entry->list));
                icmp_send_packet(pkt_entry->packet, pkt_entry->len, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
                free(pkt_entry->packet);
                free(pkt_entry);
            }
            list_delete_entry(&(req_entry->list));
            free(req_entry);
        }
    }
}
```

（五）多路由器节点网络构造

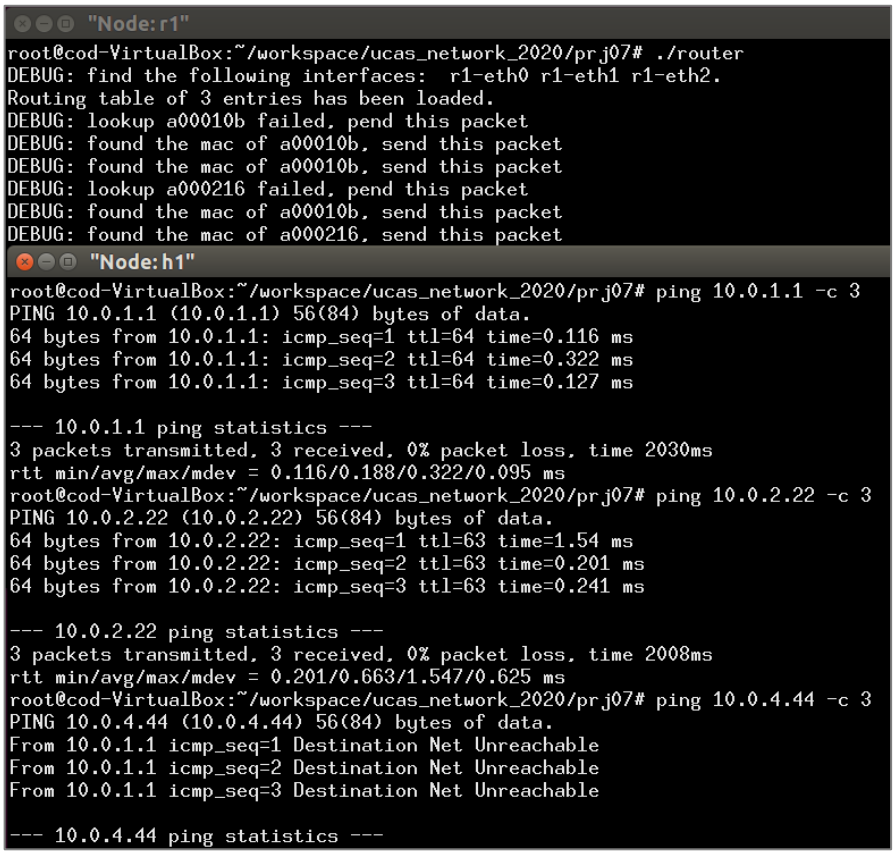
对于实验要求的多路由器节点，这里采用了最简单的线性串联结构：2 台主机结点用 3 台路由器结点线性逐个连接。端口定义和脚本运行等代码略去。若只作连通两个主机结点的用途，则只需为 3 个路由器配置 4 条路由表项：为 r1 配置向 h2 的条目，为 r2 配置向 h1 和 h2 的条目，为 r3 配置向 h1 的条目。代码如下。

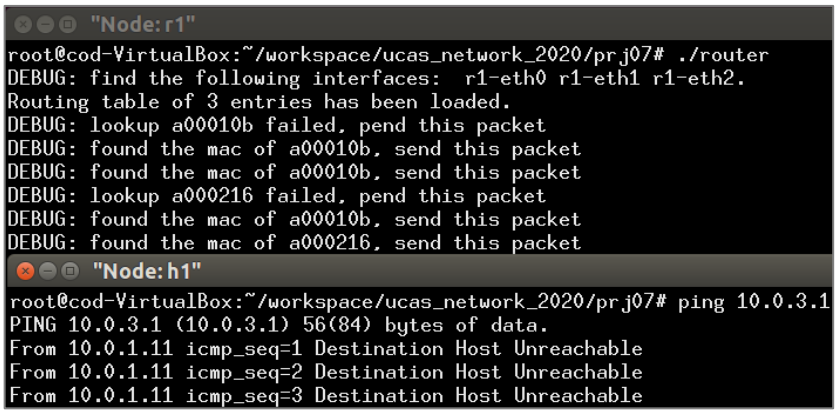
```
h1.cmd('ifconfig h1-eth0 10.0.1.11/24')
h2.cmd('ifconfig h2-eth0 10.0.4.44/24')

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

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')

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 10.0.4.0 netmask 255.255.255.0 gw 10.0.3.2 dev r2-eth1')
r2.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.2.1 dev r2-eth0')
r3.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.3.1 dev r3-eth0')
```

三、结果分析

（一）原拓扑 ping 功能测试

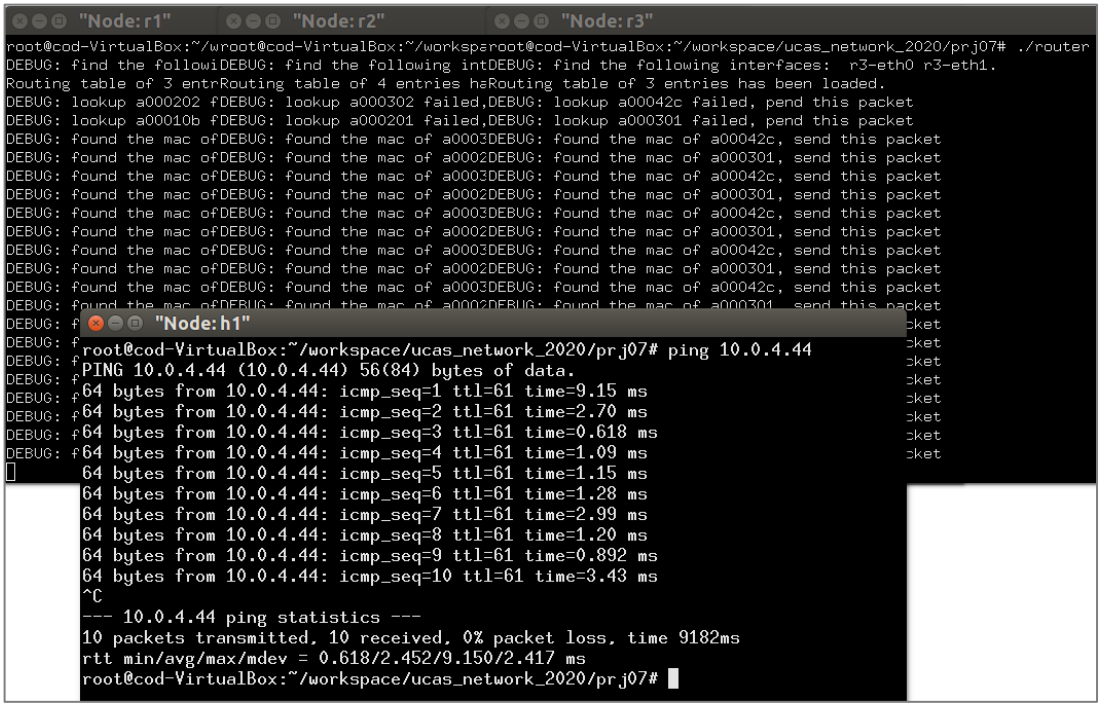
执行课程提供的` router_topo.py `脚本后，在 r1 结点上启动 router，在 h1 结点上对分别对 10.0.1.1（路由器结点）和 10.0.2.22 和 10.0.3.1 和 10.0.4.44 使用 ping 指令，得到的结果如下图所示。可见各 ICMP 包生成转发功能正确。



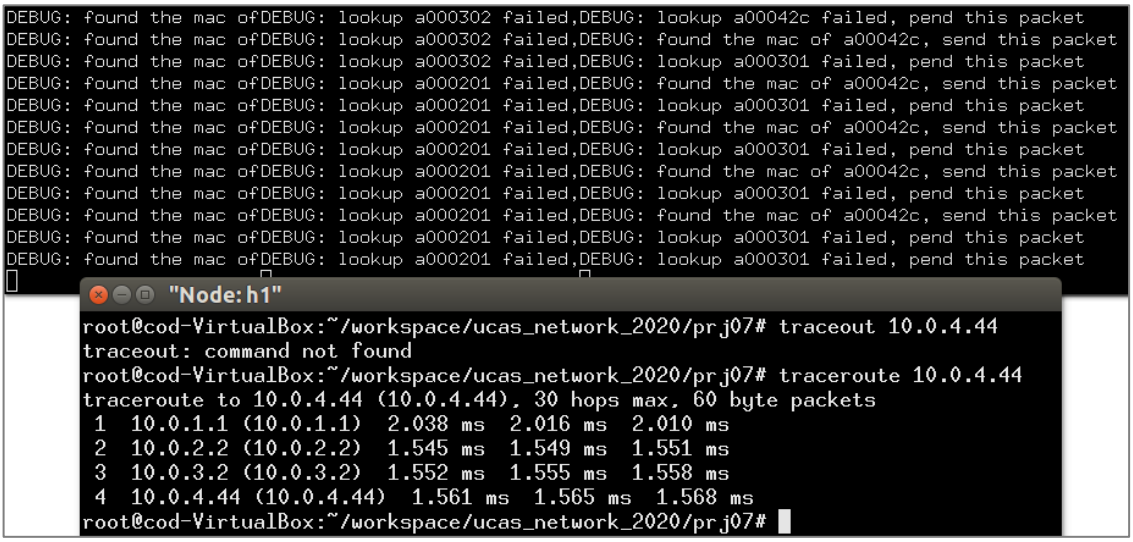


（二）多路由器节点网络 ping 及 traceout 测试

执行自己编写的` router_trace.py `脚本后，在 r1 和 r2 和 r3 结点上分别启动 router，在 h1 结点上对 h2 结点使用 ping 指令。如下图所示，两节点可以连通。



随后执行` traceroute 10.0.4.44 `指令，结果如下图所示，两结点之间路经检测可以正常进行。



（三）性能比对

最后，我又尝试运行了课程提供的` router-reference `程序，发现虽然在单路由节点时差距不明显，但在多路由节点下，我的实现中 ping 平均延时高出了许多倍，traceroute 时第一跳延时也明显较长。推测原因在于部分实现的细节：如 ICMP 包组建过程中的频繁内存拷贝，ARP 缓存中待响应表项的组织等。细节优化后路由器的处理性能应当还可以有不小的提升。