一、实验内容

1、 实验内容一

(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) 运行给定网络拓扑(router_topo.py)

- 路由器节点 r1 上执行脚本(disable_arp.sh, disable_icmp.sh, disable_ip_forward.sh)，禁止协议栈的相应功能
- 终端节点 h1-h3 上执行脚本 disable_offloading.sh

(3) 在 r1 上执行路由器程序

- 在 r1 中运行./router，进行数据包的处理

(4) 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

2、 实验内容二

(1) 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

(2) 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

(3) 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

二、实验流程

1、 实验内容一

(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) 修改 `arpcache.c` → ARP 缓存管理：

```
#include "log.h"

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 == 1 && 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");
        pthread_mutex_lock(&arpcache.lock);

        struct arp_req *req_entry = NULL, *req_q;
        list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
                if (req_entry->ip4 == ip4) {
                        struct cached_pkt *pkt_entry = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
                        memset(pkt_entry, 0, sizeof(struct cached_pkt));
                        init_list_head(&(pkt_entry->list));
                        pkt_entry->packet = packet;
                        pkt_entry->len = len;
                        list_add_tail(&pkt_entry->list, &req_entry->cached_packets);

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

        req_entry = (struct arp_req *)malloc(sizeof(struct arp_req));
        memset(req_entry, 0, sizeof(struct arp_req));
        init_list_head(&(req_entry->list));
        req_entry->iface = iface;
        req_entry->ip4 = ip4;
```

```
	req_entry->sent = time(NULL);
	req_entry->retries = 0;
	init_list_head(&(req_entry->cached_packets));
	list_add_tail(&req_entry->list, &(arpcache.req_list));

	struct cached_pkt *pkt_entry = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
	memset(pkt_entry, 0, sizeof(struct cached_pkt));
	pkt_entry->packet = packet;
	pkt_entry->len = len;
	list_add_tail(&pkt_entry->list, &req_entry->cached_packets);

	pthread_mutex_unlock(&arpcache.lock);

	arp_send_request(iface, ip4);
}

int get_an_empty_entry(void)
{
	for (int i=0; i<MAX_ARP_SIZE; i++) {
		if (arpcache.entries[i].valid == 0) {
			return i;
		}
	}
	return rand() % MAX_ARP_SIZE;
}

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);

	// 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;
		}
```

```
    }

    // find an empty entry and fill it with new mapping
    int entry_id = get_an_empty_entry();
    arpcache.entries[entry_id].ip4 = ip4;
    arpcache.entries[entry_id].added = time(NULL);
    arpcache.entries[entry_id].valid = 1;
    memcpy(arpcache.entries[entry_id].mac, mac, ETH_ALEN);

    struct arp_req *req_entry = NULL, *req_q;
    list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
        if (req_entry->ip4 == ip4) {
            struct cached_pkt *pkt_entry, *pkt_q;
            list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list) {
                struct ether_header *eh = (struct ether_header *)pkt_entry->packet;
                memcpy(eh->ether_dhost, mac, ETH_ALEN);
                memcpy(eh->ether_shost, req_entry->iface->mac, ETH_ALEN);
                eh->ether_type = htons(ETH_P_IP);
                iface_send_packet(req_entry->iface, pkt_entry->packet, pkt_entry->len);
                list_delete_entry(&(pkt_entry->list));
                free(pkt_entry->packet);
                free(pkt_entry);
            }

            list_delete_entry(&(req_entry->list));
            free(req_entry);
        }
    }

    pthread_mutex_unlock(&arpcache.lock);
}

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);
        for (int i=0; i<MAX_ARP_SIZE; i++) {
            if ( arpcache.entries[i].valid == 1 && (time(NULL) - arpcache.entries[i].added >
ARP_ENTRY_TIMEOUT) ) {
                arpcache.entries[i].valid = 0;
            }
        }

        struct arp_req *req_entry = NULL, *req_q;
        list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
            if (time(NULL) - req_entry->sent > 1) {
                (req_entry->retries)++;
                if(req_entry->retries > ARP_REQUEST_MAX_RETRIES){
                    struct cached_pkt *pkt_entry, *pkt_q;
                    list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list) {
                        //ICMP Destination Host Unreachable
                        log(DEBUG, "handle icmp host unreach packet\n");
                        icmp_send_packet(pkt_entry->packet,                 pkt_entry->len,
ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);

                        list_delete_entry(&(pkt_entry->list));
                        free(pkt_entry->packet);
                        free(pkt_entry);
                    }

                    list_delete_entry(&(req_entry->list));
                    free(req_entry);
                }
                else{
                    req_entry->sent = time(NULL);
                    arp_send_request(req_entry->iface, req_entry->ip4);
                }
            }
        }

        pthread_mutex_unlock(&arpcache.lock);
    }

    return NULL;
```

```
}
```

(3) 修改 arp.c → ARP 请求和应答处理：

```
#include "log.h"

const u8 eth_broadcast_addr[] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
const u8 arp_request_addr[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

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));
    memset(packet, 0, ETHER_HDR_SIZE + sizeof(struct ether_arp));

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

    struct ether_arp *arp = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
    arp->arp_hrd = htons(ARPHRD_ETHER);
    arp->arp_pro = htons(ETH_P_IP);
    arp->arp_hln = ETH_ALEN;
    arp->arp_pln = 4;
    arp->arp_op = htons(ARPOP_REQUEST);
    memcpy(arp->arp_sha, iface->mac, ETH_ALEN);
    arp->arp_spa = htonl(iface->ip);
    memcpy(arp->arp_tha, arp_request_addr, ETH_ALEN);
    arp->arp_tpa = htonl(dst_ip);

    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));
    free(packet);
    log(DEBUG, "handle arp send request packet\n");
}

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 *)req_hdr - ETHER_HDR_SIZE;
    struct ether_header *eh = (struct ether_header *)packet;
    //log(DEBUG, "arp_shost:" ETHER_STRING " eh_shost:" ETHER_STRING, ETHER_FMT(req_hdr->arp_sha), ETHER_FMT(eh->ether_shost));

    memcpy(eh->ether_dhost, eh->ether_shost, ETH_ALEN);
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    eh->ether_type = htons(ETH_P_ARP);

    req_hdr->arp_op = htons(ARPOP_REPLY);
    memcpy(req_hdr->arp_tha, req_hdr->arp_sha, ETH_ALEN);
    req_hdr->arp_tpa = req_hdr->arp_spa;
    memcpy(req_hdr->arp_sha, iface->mac, ETH_ALEN);
    req_hdr->arp_spa = htonl(iface->ip);

    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));
}

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 = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
    log(DEBUG, "handle arp packet\n");

    if (ntohs(arp->arp_op) == ARPOP_REQUEST) {
        if (ntohl(arp->arp_tpa) == iface->ip) {
            //log(DEBUG, "got packet from %s, %d bytes, proto_id: %d\n", iface->name, len, ntohs(arp->arp_op));
            arpcache_insert(ntohl(arp->arp_spa), arp->arp_sha);
            arp_send_reply(iface, arp);
        }
    }
    else if (ntohs(arp->arp_op) == ARPOP_REPLY) {
        if (ntohl(arp->arp_tpa) == iface->ip) {
            arpcache_insert(ntohl(arp->arp_spa), arp->arp_sha);
        }
    }
    else {
        log(ERROR, "Unknown arp packet type 0x%04hx, ingore it.", ntohs(arp->arp_op));
    }
```

```
        free(packet);
}

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);

        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);
        }
        else {
                // log(DEBUG, "lookup %x failed, pend this packet", dst_ip);
                arpcache_append_packet(iface, dst_ip, packet, len);
        }
}
```

(4) 修改 ip.c → IP 数据包的处理：

```
#include "types.h"
#include "rtable.h"
#include "icmp.h"
#include "arp.h"
#include "log.h"

void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
        fprintf(stderr, "TODO: handle ip packet.\n");
        struct iphdr *iphdr = packet_to_ip_hdr(packet);
        log(DEBUG, "handle ip packet\n");

        if (iphdr->protocol == IPPROTO_ICMP) {
```

```
        unsigned char *icmp_type = (unsigned char *)iphdr + IP_HDR_SIZE(iphdr);
        if (*icmp_type == ICMP_ECHOREQUEST && ntohl(iphdr->daddr) == iface->ip) {
            log(DEBUG, "handle icmp request packet\n");
            icmp_send_packet(packet, len, ICMP_ECHOREPLY, 0);
            return;
        }
        log(DEBUG, "forward icmp packet\n");
    }

    u32 dest_ip = ntohl(iphdr->daddr);
    rt_entry_t *rt_dest = longest_prefix_match(dest_ip);
    if (rt_dest) { // forward the packet
        iphdr->ttl = iphdr->ttl - 1;
        if (iphdr->ttl <= 0) { //ICMP TTL equals 0 during transit
            log(DEBUG, "handle icmp ttl 0 packet\n");
            icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL);
        }
        else {
            iphdr->checksum = ip_checksum(iphdr);
            if (rt_dest->gw == 0) {
                iface_send_packet_by_arp(rt_dest->iface, dest_ip, packet, len);
            }
            else {
                iface_send_packet_by_arp(rt_dest->iface, rt_dest->gw, packet, len);
            }
        }
    }
    else { //ICMP Dest Network Unreachable
        log(DEBUG, "handle icmp net unreach packet\n");
        icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
    }
}
```

(5) 修改 icmp.c → 发送 ICMP 包：

```
#include <string.h>

void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    fprintf(stderr, "TODO: malloc and send icmp packet.\n");
    struct ether_header *in_eh = (struct ether_header *)in_pkt;
    struct iphdr *in_iphdr = packet_to_ip_hdr(in_pkt);
    int len_icmp = len - ETHER_HDR_SIZE - IP_HDR_SIZE(in_iphdr);
    int packet_len;

    if(type == ICMP_ECHOREPLY)
        packet_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + len_icmp;
    else
        packet_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + ICMP_HDR_SIZE +
IP_HDR_SIZE(in_iphdr) + ICMP_COPIED_DATA_LEN;

    char *packet = (char *)malloc(packet_len);

    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);

    struct iphdr *iphdr = packet_to_ip_hdr(packet);
    rt_entry_t *src_entry = longest_prefix_match(ntohl(in_iphdr->saddr));
    ip_init_hdr(iphdr, src_entry->iface->ip, ntohl(in_iphdr->saddr), packet_len - ETHER_HDR_SIZE,
IPPROTO_ICMP);

    struct icmphdr *icmp = (struct icmphdr *)(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
    if (type == ICMP_ECHOREPLY) {
        memcpy((char*)icmp, (in_pkt + ETHER_HDR_SIZE + IP_HDR_SIZE(in_iphdr)), len_icmp);
        icmp->type = type;
        icmp->code = code;
    }
    else {
        icmp->type = type;
        icmp->code = code;
        memset((char*)icmp + 4, 0, 4);
        memcpy((char*)icmp + 8, (char*)in_iphdr, IP_HDR_SIZE(in_iphdr) +
ICMP_COPIED_DATA_LEN);
    }
    icmp->checksum = icmp_checksum(icmp, packet_len - ETHER_HDR_SIZE -
IP_BASE_HDR_SIZE);

    ip_send_packet(packet, packet_len);
```

```
	free(packet);
}
```

(6) 修改 ip_base.c → 最长前缀匹配和发送 IP 包：

```
#include "log.h"

rt_entry_t *longest_prefix_match(u32 dst)
{
	fprintf(stderr, "TODO: longest prefix match for the packet.\n");
	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;
}

void ip_send_packet(char *packet, int len)
{
	fprintf(stderr, "TODO: send ip packet.\n");
	struct iphdr *iphdr = packet_to_ip_hdr(packet);
	u32 dest_ip = ntohl(iphdr->daddr);
	rt_entry_t *rt_dest = longest_prefix_match(dest_ip);

	iface_send_packet(rt_dest->iface, packet, len);
}
```

(7) 修改 device_internal.c：

```
void iface_send_packet(iface_info_t *iface, const char *packet, int len)
{
	struct sockaddr_ll addr;
	memset(&addr, 0, sizeof(struct sockaddr_ll));
	addr.sll_family = AF_PACKET;
	addr.sll_ifindex = iface->index;
	addr.sll_halen = ETH_ALEN;
	addr.sll_protocol = htons(ETH_P_ARP);
```

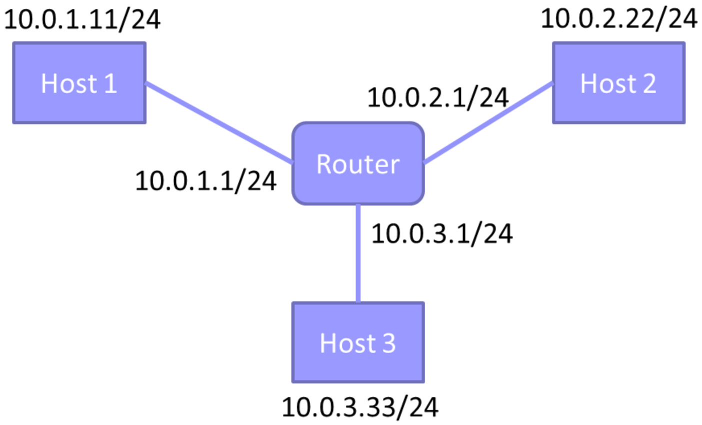
```
    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(addr.sll_addr, eh->ether_dhost, ETH_ALEN);

    if (sendto(iface->fd, packet, len, 0, (const struct sockaddr *)&addr,
                sizeof(struct sockaddr_ll)) < 0) {
        perror("Send raw packet failed");
    }

    // free((char *)packet);
}
```

(8) 执行命令 make，生成可执行程序 router：

```
wasder@WASDER:~/exp2/4-router$ make
```

(9) 运行给定网络拓扑(router_topo.py)：

```
wasder@WASDER:~/exp2/4-router$ sudo python3 router_topo.py
```



(10) 路由器节点 r1 上执行脚本(disable_arp.sh, disable_icmp.sh, disable_ip_forward.sh)，禁止协议栈的相应功能

```
mininet> xterm r1
r1# cd scripts/
r1# ./disable_arp.sh
r1# ./disable_icmp.sh
r1# ./disable_ip_forward.sh
```

(11) 终端节点 h1-h3 上执行脚本 disable_offloading.sh

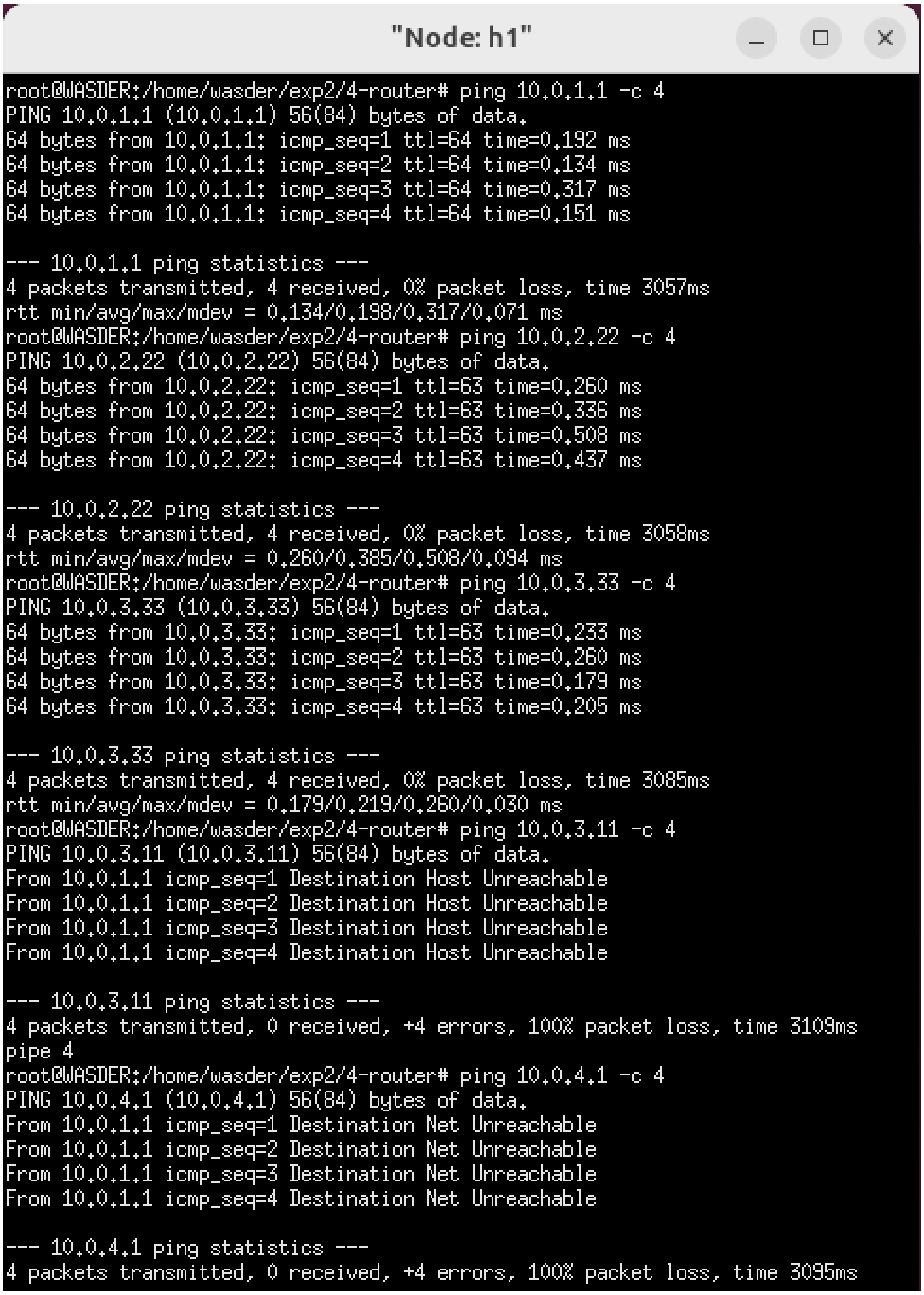
```
mininet> xterm h1 h2 h3
h?# cd scripts/
h?# ./disable_offloading.sh
```

(12) 在 r1 上执行路由器程序（在 r1 中运行./router，进行数据包的处理）

```
r1# cd ..
r1# ./router
```

(13) 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



```
root@WASDER:/home/wasder/exp2/4-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=0.192 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.134 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.317 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.151 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3057ms
rtt min/avg/max/mdev = 0.134/0.198/0.317/0.071 ms
root@WASDER:/home/wasder/exp2/4-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=0.260 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.336 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.508 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.437 ms

--- 10.0.2.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3058ms
rtt min/avg/max/mdev = 0.260/0.385/0.508/0.094 ms
root@WASDER:/home/wasder/exp2/4-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=0.233 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.260 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.179 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.205 ms

--- 10.0.3.33 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3085ms
rtt min/avg/max/mdev = 0.179/0.219/0.260/0.030 ms
root@WASDER:/home/wasder/exp2/4-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 3109ms
pipe 4
root@WASDER:/home/wasder/exp2/4-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 3095ms
```

2、 实验内容二

(1) 建立 four_node_ring.py 的副本 seven_node_ring.py：

```
wasder@WASDER:~/exp2/4-router$ cp router_topo.py router_topo_copy.py
```

(2) 根据所需拓扑修改 router_topo_copy.py:

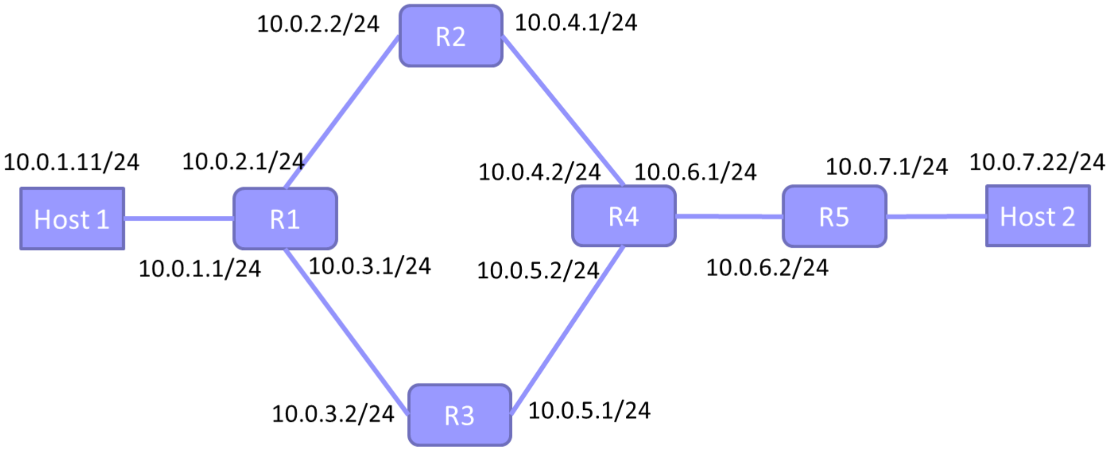
```
class RouterTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        r1 = self.addHost('r1')
        r2 = self.addHost('r2')
        r3 = self.addHost('r3')
        r4 = self.addHost('r4')
        r5 = self.addHost('r5')

        self.addLink(h1, r1)
        self.addLink(r1, r2)
        self.addLink(r1, r3)
        self.addLink(r2, r4)
        self.addLink(r3, r4)
        self.addLink(r4, r5)
        self.addLink(r5, h2)

if __name__ == '__main__':
    check_scripts()

    topo = RouterTopo()
    net = Mininet(topo = topo, controller = None)

    h1, h2, r1, r2, r3, r4, r5 = net.get('h1', 'h2', 'r1', 'r2', 'r3', 'r4', 'r5')
    h1.cmd('ifconfig h1-eth0 10.0.1.11/24')
    h2.cmd('ifconfig h2-eth0 10.0.7.22/24')

    h1.cmd('route add default gw 10.0.1.1')
    h2.cmd('route add default gw 10.0.7.1')

    r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
    r1.cmd('ifconfig r1-eth1 10.0.2.1/24')
    r1.cmd('ifconfig r1-eth2 10.0.3.1/24')
```

```
r2.cmd('ifconfig r2-eth0 10.0.2.2/24')
r2.cmd('ifconfig r2-eth1 10.0.4.1/24')

r3.cmd('ifconfig r3-eth0 10.0.3.2/24')
r3.cmd('ifconfig r3-eth1 10.0.5.1/24')

r4.cmd('ifconfig r4-eth0 10.0.4.2/24')
r4.cmd('ifconfig r4-eth1 10.0.5.2/24')
r4.cmd('ifconfig r4-eth2 10.0.6.1/24')

r5.cmd('ifconfig r5-eth0 10.0.6.2/24')
r5.cmd('ifconfig r5-eth1 10.0.7.1/24')

r1.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
r1.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.3.2 dev r1-eth2')
r1.cmd('route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
r1.cmd('route add -net 10.0.7.0 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 10.0.3.0 netmask 255.255.255.0 gw 10.0.2.1 dev r2-eth0')
r2.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.4.2 dev r2-eth1')
r2.cmd('route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.4.2 dev r2-eth1')
r2.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.4.2 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')
r3.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.5.2 dev r3-eth1')
r3.cmd('route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.5.2 dev r3-eth1')
r3.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.5.2 dev r3-eth1')

r4.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.4.1 dev r4-eth0')
r4.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.4.1 dev r4-eth0')
r4.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.5.1 dev r4-eth1')
r4.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.6.2 dev r4-eth2')

r5.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
```

```
r5.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')

for n in (h1, h2, r1, r2, r3, r4, r5):
        n.cmd('./scripts/disable_offloading.sh')
        n.cmd('./scripts/disable_ipv6.sh')

for n in (r1, r2, r3, r4, r5):
        n.cmd('./scripts/disable_arp.sh')
        n.cmd('./scripts/disable_icmp.sh')
        n.cmd('./scripts/disable_ip_forward.sh')
```

(3) 运行给定网络拓扑(router_topo.py)：

```
wasder@WASDER:~/exp2/4-router$ sudo python3 router_topo_copy.py
```

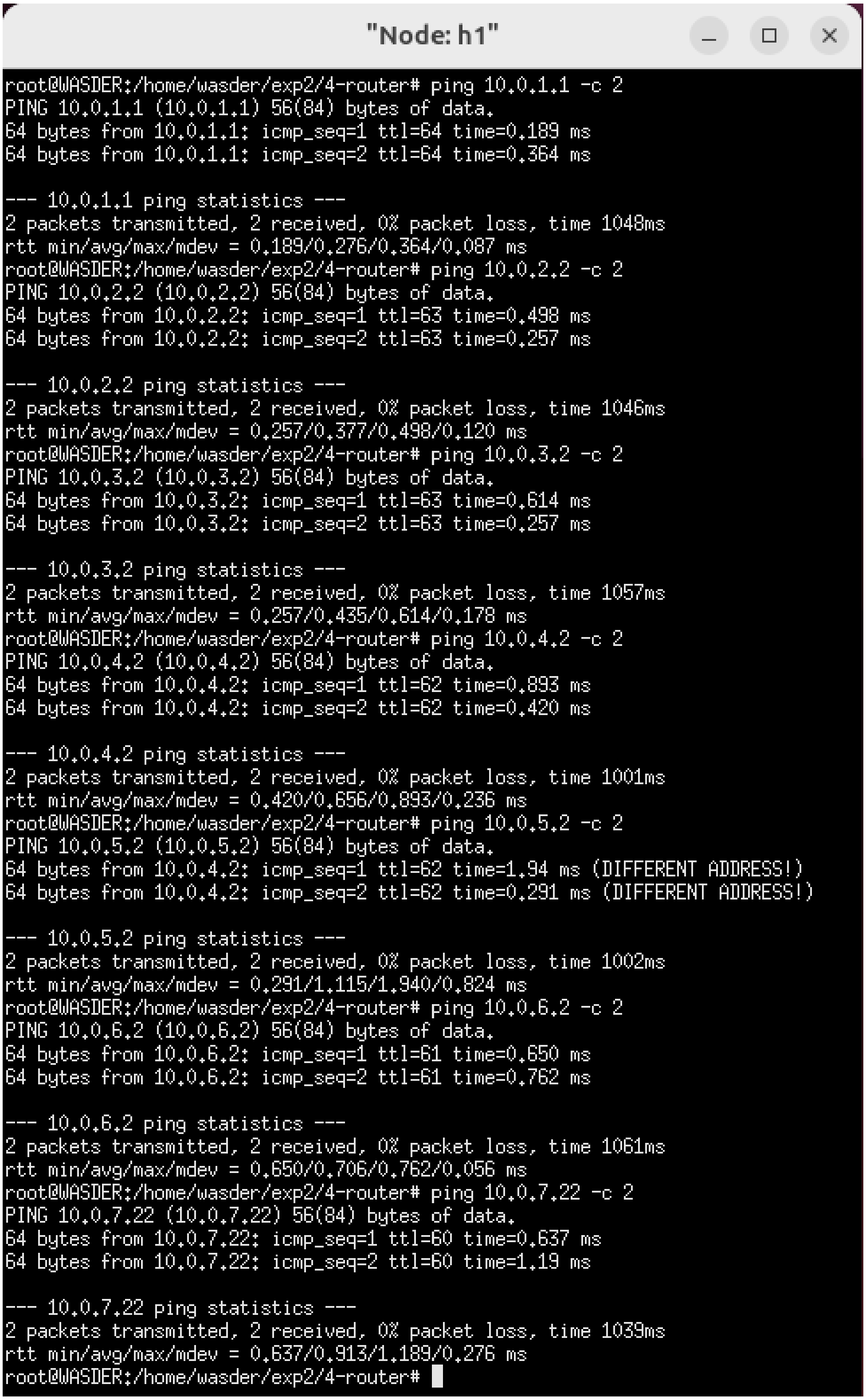


(4) 在 r1-r5 上执行路由器程序（在 r1-r5 中运行./router，进行数据包的处理）

```
mininet> xterm r1 r2 r3 r4 r5
r?# ./router
```

(5) 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）：

```
mininet> xterm h1
```

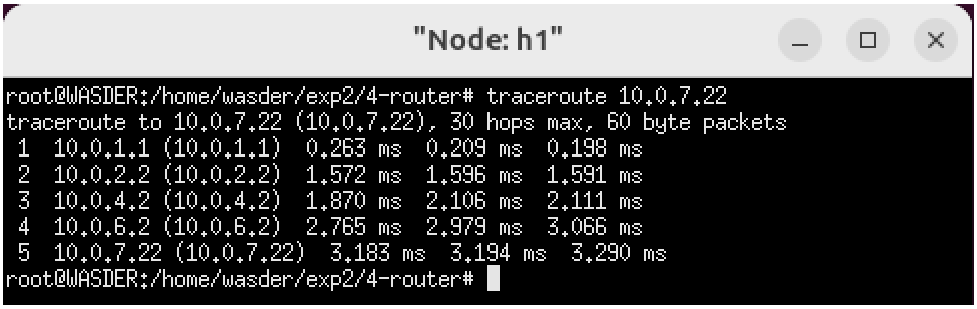


```
root@WASDER:/home/wasder/exp2/4-router# ping 10.0.1.1 -c 2
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=0.189 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.364 ms

--- 10.0.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1048ms
rtt min/avg/max/mdev = 0.189/0.276/0.364/0.087 ms
root@WASDER:/home/wasder/exp2/4-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=0.498 ms
64 bytes from 10.0.2.2: icmp_seq=2 ttl=63 time=0.257 ms

--- 10.0.2.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1046ms
rtt min/avg/max/mdev = 0.257/0.377/0.498/0.120 ms
root@WASDER:/home/wasder/exp2/4-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=0.614 ms
64 bytes from 10.0.3.2: icmp_seq=2 ttl=63 time=0.257 ms

--- 10.0.3.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1057ms
rtt min/avg/max/mdev = 0.257/0.435/0.614/0.178 ms
root@WASDER:/home/wasder/exp2/4-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=0.893 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=0.420 ms

--- 10.0.4.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.420/0.656/0.893/0.236 ms
root@WASDER:/home/wasder/exp2/4-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=1.94 ms (DIFFERENT ADDRESS!)
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=0.291 ms (DIFFERENT ADDRESS!)

--- 10.0.5.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.291/1.115/1.940/0.824 ms
root@WASDER:/home/wasder/exp2/4-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=0.650 ms
64 bytes from 10.0.6.2: icmp_seq=2 ttl=61 time=0.762 ms

--- 10.0.6.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1061ms
rtt min/avg/max/mdev = 0.650/0.706/0.762/0.056 ms
root@WASDER:/home/wasder/exp2/4-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=0.637 ms
64 bytes from 10.0.7.22: icmp_seq=2 ttl=60 time=1.19 ms

--- 10.0.7.22 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1039ms
rtt min/avg/max/mdev = 0.637/0.913/1.189/0.276 ms
root@WASDER:/home/wasder/exp2/4-router# 
```

(6) 安装 `traceroute`：

```
wasder@WASDER:~/exp2/4-router$ sudo apt install traceroute
```

(7) 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):



```
root@WASDER:/home/wasder/exp2/4-router# traceroute 10.0.7.22
traceroute to 10.0.7.22 (10.0.7.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.263 ms  0.209 ms  0.198 ms
 2  10.0.2.2 (10.0.2.2)  1.572 ms  1.596 ms  1.591 ms
 3  10.0.4.2 (10.0.4.2)  1.870 ms  2.106 ms  2.111 ms
 4  10.0.6.2 (10.0.6.2)  2.765 ms  2.979 ms  3.066 ms
 5  10.0.7.22 (10.0.7.22)  3.183 ms  3.194 ms  3.290 ms
root@WASDER:/home/wasder/exp2/4-router# 
```

如上图所示，h1 到 h2 路径为：h1 → r1 → r2 → r4 → r5 → h2。