Project2: Static routing forwarding

徐磊 201828018670040

崔天宇 201818018670007

陈洁 2018E8018661111

一 实验目的

- 理解 IP、ARP、ICMP 协议的工作机制
- 实现 IP 地址查找与数据包转发
- 实现 ARP 请求和应答、ARP 数据包缓存管理
- 实现 ICMP 消息的发送
- 学习 ping，traceroute 等网络命令的基本功能和使用方法

二 实验环境

- Ubuntu16.04
- Mininet

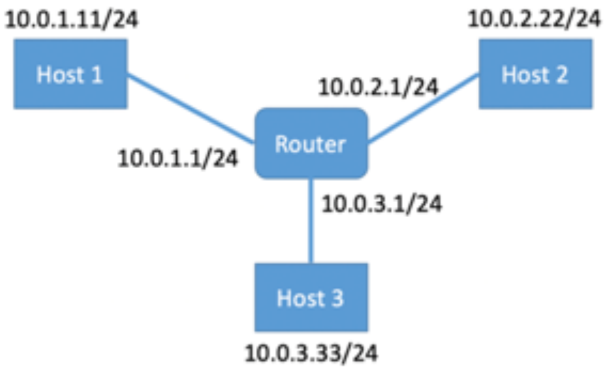
三 实验内容

根据已有的基础代码，继续完成 TODO 部分。在两种网络拓扑结构下，主要完 成以下三个部分:

- **IP协议相关**：查找转发(基于最长前缀匹配规则)
- **ARP协议相关**：IP-Mac 地址映射基本概念，ARP 请求、ARP 回应数据包格式，ARP 条目查询，ARP 缓存管理

- **ICMP 协议**：实验中涉及到的几种 ICMP 数据包格式(路由表查找失败、ARP 查 询失败、TTL 值减为 0、收到 ping 本端又的数据包)

> 实验一：对应的星型网络拓扑结构：



完善代码中的 TODO 部分，实现路由器的连通性测试。即在 h1 上进行 ping 实 验:

Ping 10.0.1.1 (r1)，可以 ping 通

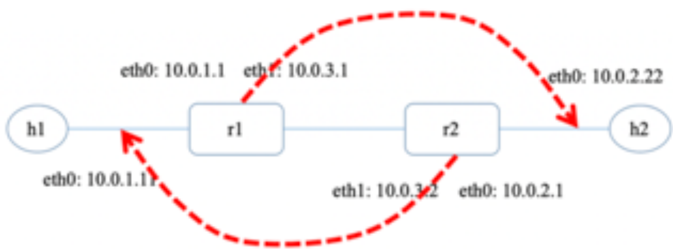
Ping 10.0.2.22 (h2)，可以 ping 通

Ping 10.0.3.33 (h3)，可以 ping 通

Ping 10.0.3.11，返回 ICMP Destination Host Unreachable

Ping 10.0.4.1，返回 ICMP Destination Net Unreachable

> 实验二:根据要求配置网络拓扑环境:



建立如上线性网络拓扑，实现路由器的连通性测试。以及路径测试:在一台主 机上可以 traceroute 其他主机，获取数据包经过每个节点入端又的 IP 地址。

四 实验流程

实验待完善内容为：

- **处理IP数据包**：

  判断是否为 ICMP echo 请求，否则利用最长前缀匹配规则转发 IP 包，转发 IP 包 之前需要检查 TTL，更新 checksum 等

- **发送 ICMP 数据包**：

满足下述四个条件时发送 ICMP 数据包

  - TTL 值为 0
  - 查找不到路由表条目收到
  - ping 本端又的包
  - ARP 查询失败

- **ARP 缓存管理**：

  在没有触发前三个条件时，将数据包添加到 ARP 缓存的等待队列中，并发送 ARP 请求。注意缓存管理要避免死锁的出现(lock unlock)

- **处理 ARP 请求和应答**：

  收到 ARP 数据包时，先判断是 ARP 请求还是 ARP 应答

  处理 ARP 请求数据包时，先发送 ARP 回复数据包，再将 IP-Mac 地址映射对插 入 ARP 缓存中，并查找等待队列中是否有数据包符合地址映射对，如有，则将对应 的数据包发送 处理 ARP 回复数据包时，同理也先将 IP-Mac 地址映射对插入 ARP 缓存中，再查找等待队列中是否有数据包符合地址映射对，如有，则将对应的数据包发送。

4.1 相关代码的TODO部分

> 代码：arp.c

- 发送arp请求

```c
// send an arp request: encapsulate an arp request packet, send it out through
// iface_send_packet
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;
	packet = (char*)malloc(sizeof(struct ether_header) + sizeof(struct ether_arp));
	struct ether_header* eh;
	eh = (struct ether_header*)(packet);
	memcpy(eh->ether_shost, iface->mac, ETH_ALEN*sizeof(u8));
	show(eh->ether_shost);
	u8 temp = 255;
	for(int q = 0; q < ETH_ALEN; q++) eh->ether_dhost[q] = temp;
	show(eh->ether_dhost);
	eh->ether_type = htons(ETH_P_ARP);

	struct ether_arp* eh_arp;
	eh_arp = (struct ether_arp*)(packet + ETHER_HDR_SIZE);
	eh_arp->arp_hrd = htons(0x01);
	eh_arp->arp_pro = htons(0x0800);
	eh_arp->arp_hln = 6;
	eh_arp->arp_pln = 4;
	eh_arp->arp_op = htons(0x01);
	memcpy(eh_arp->arp_sha, iface->mac, ETH_ALEN*sizeof(u8));
	printf("************************\n");
	show(iface->mac);
	show(eh_arp->arp_sha);
	eh_arp->arp_spa = htonl(iface->ip);
	eh_arp->arp_tpa = htonl(dst_ip);
	memset(eh_arp->arp_tha,0,ETH_ALEN*sizeof(u8));
	// show(eh_arp->arp_tha);
	iface_send_packet(iface, packet, ETHER_HDR_SIZE+sizeof(struct ether_arp));
}
```

- 发送arp reply

```c
// send an arp reply packet: encapsulate an arp reply packet, send it out
// through iface_send_packet
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;
    packet = (char*)malloc(ETHER_HDR_SIZE + sizeof(struct ether_arp));
    struct ether_header* eh;
    eh = (struct ether_header*)packet;
    struct ether_arp* eh_arp;
    eh_arp = (struct eh_arp*)(packet + ETHER_HDR_SIZE);
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN*sizeof(u8));
    memcpy(eh->ether_dhost, req_hdr->arp_sha, ETH_ALEN*sizeof(u8));
    eh->ether_type = htons(ETH_P_ARP);
    eh_arp->arp_hrd = htons(0x01);
    eh_arp->arp_pro = htons(0x0800);
    eh_arp->arp_hln = 6;
    eh_arp->arp_pln = 4;
    eh_arp->arp_op = htons(ARPOP_REPLY);
    eh_arp->arp_spa = htonl(iface->ip);
    u32 my_ip = ntohl(req_hdr->arp_spa);
    eh_arp->arp_tpa = htonl(my_ip);
    memcpy(eh_arp->arp_sha,iface->mac,ETH_ALEN*sizeof(u8));
    memcpy(eh_arp->arp_tha,req_hdr->arp_sha,ETH_ALEN*sizeof(u8));

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

- 发送arp request和arp reply，并且将ip->mac映射和相关信息放入arpcache中

```c
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_header* eh;
    struct ether_arp* eh_arp;
    eh = (struct ether_header*)(packet);
    eh_arp = (struct ether_arp*)(packet + ETHER_HDR_SIZE);
    if(ntohs(eh_arp->arp_op) == 0x01){
        if(ntohl(eh_arp->arp_tpa) == iface->ip){
            arp_send_reply(iface, eh_arp);
            arpcache_insert(ntohl(eh_arp->arp_spa),eh_arp->arp_sha);
        }
    }
    else if(ntohs(eh_arp->arp_op) == 0x02){
        if(ntohl(eh_arp->arp_tpa) == iface->ip){
            arpcache_insert(ntohl(eh_arp->arp_spa), eh_arp->arp_sha);
        }
    }
    free(packet);
}
```

```c
// send (IP) packet through arpcache lookup
//
// Lookup the mac address of dst_ip in arpcache. If it is found, fill the
// ethernet header and emit the packet by iface_send_packet, otherwise, pending
// this packet into arpcache, and send arp request.
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) {
        memcpy(eh->ether_dhost, dst_mac, ETH_ALEN*sizeof(u8));
        if(ntohs(eh->ether_type) == ETH_P_IP){
            struct iphdr* myiph;
            myiph = (struct iphdr*)(packet + ETHER_HDR_SIZE);
            struct icmphdr* myicmph;
            myicmph = (struct icmphdr*)(packet + ETHER_HDR_SIZE + IP_HDR_SIZE(myiph));
            u32 sip = ntohl(myiph->saddr);
            u32 dip = ntohl(myiph->daddr);
        }
        iface_send_packet(iface, packet, len);
    }
    else {
        arpcache_append_packet(iface, dst_ip, packet, len);
    }

}
```

> 代码 Arpcache.c

- 查找arpchae中已将有相同的IP项

```c
// lookup the IP->mac mapping
//
// traverse the table to find whether there is an entry with the same IP
// and mac address with the given arguments
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
	pthread_mutex_lock(&arpcache.lock);
	fprintf(stderr, "TODO: lookup ip address in arp cache.\n");
	show_cache(arpcache);
	int i=0;
	for(i=0; i < MAX_ARP_SIZE; i++){
		IP_int2string(ip4);
		if(arpcache.entries[i].ip4 == ip4 && arpcache.entries[i].valid == 1){
			printf("find in chache!\n");
			// pthread_mutex_lock(&arpcache.lock);
			memcpy(mac, arpcache.entries[i].mac, ETH_ALEN*sizeof(u8));
			// printf("--------------------\n");
			// show_mac(mac);
			// show_mac(arpcache.entries[i].mac);
			pthread_mutex_unlock(&arpcache.lock);
			return 1;
		}
	}
	printf("did not find in cache\n");
	pthread_mutex_unlock(&arpcache.lock);
	return 0;
}
```

- Append the packet to arpcache

```c
// append the packet to arpcache
//
// Lookup in the list which stores pending packets, if there is already an
// entry with the same IP address and iface (which means the corresponding arp
// request has been sent out), just append this packet at the tail of that entry
// (the entry may contain more than one packet); otherwise, malloc a new entry
// with the given IP address and iface, append the packet, and send arp request.
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");
	//find
	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->iface->ip == iface->ip && req_entry->ip4 == ip4){
			struct cached_pkt *cache;
			cache = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
			cache->packet = packet;
			cache->len = len;
			list_add_tail(&(cache->list), &(req_entry->cached_packets));
			pthread_mutex_unlock(&arpcache.lock);
			return;
		}
	}
	//add
	struct arp_req* req_q_new;
	req_q_new = (struct arp_req*)malloc(sizeof(struct arp_req));
	req_q_new->iface = iface;
	req_q_new->ip4 = ip4;
	req_q_new->sent = time(NULL);
	req_q_new->retries = 0;
	init_list_head(&(req_q_new->cached_packets));
	list_add_tail(&(req_q_new->list), &(arpcache.req_list));

	struct cached_pkt *cache_new;
	cache_new = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
	cache_new->packet = packet;
	cache_new->len = len;
	list_add_tail(&(cache_new->list), &(req_q_new->cached_packets));
	pthread_mutex_unlock(&arpcache.lock);
	arp_send_request(iface, ip4);
}
```

将IP->mac的映射插入arpcache中，如果arpcache中没有位置可以添加映射，则随机插入某项。如果有待处理的数据包等待这个映射，请为每个数据填充以太网头，并将它们发送出去

```c
// insert the IP->mac mapping into arpcache, if there are pending packets
// waiting for this mapping, fill the ethernet header for each of them, and send
// them out
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);
    //insert the IP->mac mapping into arpcache
    show_cache(arpcache);
    int flag = 0;
    for(int i = 0; i < MAX_ARP_SIZE; i++){
        pthread_mutex_lock(&arpcache.lock);
        if(arpcache.entries[i].ip4 == 0){
            pthread_mutex_lock(&arpcache.lock);
            if(isRepeat(ip4))
            {
            arpcache.entries[i].ip4 = ip4;
            memcpy(arpcache.entries[i].mac, mac, ETH_ALEN*sizeof(u8));
            arpcache.entries[i].added = time(NULL);
            arpcache.entries[i].valid = 1;
            }
            pthread_mutex_unlock(&arpcache.lock);
            flag = 1;
            break;
        }
        else
            pthread_mutex_unlock(&arpcache.lock);
    }
    //arpcache has been filled totally, insert randomly
    if(!flag){
        int index = rand()%MAX_ARP_SIZE;
        pthread_mutex_lock(&arpcache.lock);
        arpcache.entries[index].ip4 = ip4;
        memcpy(arpcache.entries[index].mac, mac, ETH_ALEN*sizeof(u8));
        arpcache.entries[index].added = time(NULL);
        arpcache.entries[index].valid = 1;
        pthread_mutex_unlock(&arpcache.lock);
    }
```

```c
    //deal with pending packets
    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 *cache = NULL, *cache_q;
            list_for_each_entry_safe(cache, cache_q, &(req_entry->cached_packets), list) {
                char* packet = cache->packet;
                int len = cache->len;
                struct ether_header* eh;
                eh = (struct ether_header*)(packet);
                memcpy(eh->ether_dhost, mac, ETH_ALEN*sizeof(u8));
                iface_send_packet(req_entry->iface, packet,len);
                list_delete_entry(&(cache->list));
            }
        }
        list_delete_entry(&(req_entry->list));
    }
    // show_cache(arpcache);
}
```

- 定期扫描arpcache

  对于IP-> mac条目，如果条目在表中已超过15秒，将其从表中删除。对于挂起的数据包，如果arp请求在1秒前发送出去，而未收到回复，则重新发送arp请求。 如果arp请求已经发送了5次而没有收到arp回复，则对于每个挂起的数据包，发送icmp数据包（DEST_HOST_UNREACHABLE），并丢弃这些数据包。

```c
// sweep arpcache periodically
//
// For the IP->mac entry, if the entry has been in the table for more than 15
// seconds, remove it from the table.
// For the pending packets, if the arp request is sent out 1 second ago, while
// the reply has not been received, retransmit the arp request. If the arp
// request has been sent 5 times without receiving arp reply, for each
// pending packet, send icmp packet (DEST_HOST_UNREACHABLE), and drop these
// packets.
void *arpcache_sweep(void *arg)
{
	while (1) {
		sleep(1);
		fprintf(stderr, "TODO: sweep arpcache periodically: remove old entries, resend arp requests .\n");
		show_cache(arpcache);
		//避免死锁
		pthread_mutex_unlock(&arpcache.lock);
		pthread_mutex_lock(&arpcache.lock);
		//f the entry has been in the table for more than 15
		// seconds, remove it from the table
		for(int i = 0; i < MAX_ARP_SIZE; i++){
			if(time(NULL) - arpcache.entries[i].added >= 15 && arpcache.entries[i].ip4!=0){
				arpcache.entries[i].ip4 = 0;
				arpcache.entries[i].valid = 0;
			}
		}
		//For the pending packets, if the arp request is sent out 1 second ago, while the reply has not been r
		struct arp_req *arp_req,*arp_req_q;
		time_t nowtime = time((time_t*)NULL);
		list_for_each_entry_safe(arp_req,arp_req_q,&(arpcache.req_list),list){
			if(nowtime - arp_req->sent>1 && arp_req->retries<5){
				iface_info_t *iface = arp_req->iface;
				u32 ip4 = arp_req->ip4;
				arp_send_request(iface,ip4);
				arp_req->sent= nowtime;
				arp_req->retries +=1;
			}
```

```c
			//more than 5, resend arp requests
			else if(arp_req->retries>2){
				struct cached_pkt *cache,*cache_q;
				list_for_each_entry_safe(cache,cache_q,&(arp_req->cached_packets),list){

					char* packet = (char*)cache->packet;
					struct ether_header* eh = (struct ether_header*)packet;
					struct iphdr* iph = packet_to_ip_hdr(packet);
					u32 source_ip = ntohl(iph->saddr);
					u32 dst_ip = ntohl(iph->daddr);
					pthread_mutex_unlock(&arpcache.lock);
					u32 dst = ntohl(iph->saddr);
					rt_entry_t* entry = longest_prefix_match(dst);
					u32 sip = entry->iface->ip;
					printf("sending ICMP packet\n");
					icmp_send_packet((char*)cache->packet,cache->len,3,1,sip);
					pthread_mutex_lock(&arpcache.lock);
					list_delete_entry(&(cache->list));
				}
				pthread_mutex_unlock(&arpcache.lock);
				pthread_mutex_lock(&arpcache.lock);
				list_delete_entry(&(arp_req->list));
			}
		}
		pthread_mutex_unlock(&arpcache.lock);
	}

	return NULL;
}
```

> 代码 Icmp.c

- 发送ICMP包

```c
// send icmp packet
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code, u32 sip)
{
	fprintf(stderr, "TODO: malloc and send icmp packet.\n");
	int offset;
	char* icmp_packet;
	struct ether_header* eh_temp = (struct ether_header*)(in_pkt);
	struct iphdr* iph_temp = (struct iphdr*)(in_pkt + ETHER_HDR_SIZE);
	struct icmphdr* icmp_temp = (struct icmp*)(in_pkt + ETHER_HDR_SIZE + IP_HDR_SIZE(iph_temp));
	if(type == 0 && code == 0){
		icmp_packet = (char*)malloc(len);
		struct ether_header* eh = (struct ether_header*)icmp_packet;
		struct iphdr* iph = (struct iphdr*)(icmp_packet + ETHER_HDR_SIZE);
		struct icmphdr* icmph = (struct icmphdr*)(icmp_packet + ETHER_HDR_SIZE + IP_HDR_SIZE(iph_temp));
		eh->ether_type = htons(ETH_P_IP);
		memcpy(eh->ether_shost, eh_temp->ether_dhost, ETH_ALEN*sizeof(u8));
		memcpy(eh->ether_dhost, eh_temp->ether_shost, ETH_ALEN*sizeof(u8));
		u32 daddr_temp = sip;
		u32 saddr_temp = ntohl(iph_temp->saddr);
		ip_init_hdr(iph,daddr_temp,saddr_temp,len - ETHER_HDR_SIZE,IPPROTO_ICMP);
		icmph->type = 0;
		icmph->code = 0;

		memcpy(((char*)icmph) + 4,((char*)icmp_temp) + 4,len-ETHER_HDR_SIZE-IP_HDR_SIZE(iph_temp));

		icmph->checksum=icmp_checksum(icmph,len-ETHER_HDR_SIZE-IP_HDR_SIZE(iph_temp));
		offset = len;
	}
	else{
		offset = ETHER_HDR_SIZE + 2*IP_HDR_SIZE(iph_temp) + 16;
		icmp_packet = (char*)malloc(ETHER_HDR_SIZE + IP_HDR_SIZE(iph_temp) + 8 + IP_HDR_SIZE(iph_temp) + 8);
		struct ether_header* eh = (struct ether_header*)icmp_packet;
		struct iphdr* iph = (struct iphdr*)(icmp_packet + ETHER_HDR_SIZE);
		struct icmphdr* icmph = (struct icmphdr*)(icmp_packet + ETHER_HDR_SIZE + IP_HDR_SIZE(iph_temp));
		eh->ether_type = htons(ETH_P_IP);
		memcpy(eh->ether_dhost, eh_temp->ether_shost, ETH_ALEN*sizeof(u8));
		memcpy(((char*)icmph) + 8, (char*)iph_temp, IP_HDR_SIZE(iph_temp) + 8);
		u32 daddr_temp = sip;
		u32 saddr_temp = ntohl(iph_temp->saddr);
		ip_init_hdr(iph,daddr_temp,saddr_temp,offset - ETHER_HDR_SIZE,IPPROTO_ICMP);
		memset(((char*)icmph) + 4, 0, 4);
		icmph->type = type;
		icmph->code = code;
		icmph->checksum=icmp_checksum(icmph,offset-ETHER_HDR_SIZE-IP_HDR_SIZE(iph_temp));
	}
	ip_send_packet(icmp_packet, offset);
}
```

> 代码 Ip_forwarding.c

```c
// forward the IP packet from the interface specified by longest_prefix_match,
// when forwarding the packet, you should check the TTL, update the checksum,
// determine the next hop to forward the packet, then send the packet by
// iface_send_packet_by_arp
void ip_forward_packet(u32 ip_dst, char *packet, int len)
{
	fprintf(stderr, "TODO: forward ip packet.\n");

	struct ether_header* eh = (struct ether_header*)packet;
	struct iphdr* iph = packet_to_ip_hdr(packet);
	struct icmphdr* my_icmph = (struct icmphdr *)(packet + ETHER_HDR_SIZE + IP_HDR_SIZE(iph));

	// deal with the packet
	u32 dst = ntohl(iph->daddr);
	rt_entry_t* my_entry = longest_prefix_match(dst);
	//failed
	if(!my_entry){
		u32 dst = ntohl(iph->saddr);
        rt_entry_t* my_entry = longest_prefix_match(dst);
		u32 sip = my_entry->iface->ip;
		icmp_send_packet(packet, len, 3, 0, sip);
		free(packet);
		return;
	}
	//TTL - 1 <= 0
	iph->ttl -= 1;
	if(iph->ttl <= 0){
                u32 dst = ntohl(iph->saddr);
                rt_entry_t* my_entry = longest_prefix_match(dst);
                u32 sip = my_entry->iface->ip;

		icmp_send_packet(packet, len, 11, 0,sip);
		free(packet);
		return;
	}
	iph->checksum = ip_checksum(iph);
	memcpy(eh->ether_shost, my_entry->iface->mac, ETH_ALEN);
	u32 next_hop = my_entry->gw;
	if (!next_hop)
		next_hop = ntohl(iph->daddr);
	iface_send_packet_by_arp(my_entry->iface, next_hop, packet, len);
}
```

- 处理ip包

  如果数据包是ICMP echo请求，目的IP地址等于iface的IP地址，则发送ICMP echo reply; 否则，转发数据包。

```
// handle ip packet
//
// If the packet is ICMP echo request and the destination IP address is equal to
// the IP address of the iface, send ICMP echo reply; otherwise, forward the
// packet.
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_header* eh = (struct ether_header*)packet;
    struct iphdr* iph = packet_to_ip_hdr(packet);
    struct icmphdr* my_icmph = (struct icmphdr *)(packet + ETHER_HDR_SIZE + IP_HDR_SIZE(iph));

    if(my_icmph->type == 8 && ntohl(iph->daddr) == iface->ip){
                u32 dst = ntohl(iph->saddr);
                rt_entry_t* my_entry = longest_prefix_match(dst);
                u32 sip = my_entry->iface->ip;

                icmp_send_packet(packet, len, 0, 0, sip);
                free(packet);
                return;
    }
    else {
        ip_forward_packet(iph->daddr, packet, len);
    }

}
```

> 代码 ip.c

- 在路由表中查找，找到具有相同和最长前缀的条目。输入地址按主机字节顺序排列

```
// lookup in the routing table, to find the entry with the same and longest prefix.
// the input address is in host byte order
rt_entry_t *longest_prefix_match(u32 dst)
{
    fprintf(stderr, "TODO: longest prefix match for the packet.\n");

    rt_entry_t* input = NULL;
    rt_entry_t* output = NULL;
    u32 number = 0;
    list_for_each_entry(input, &rtable, list){
        u32 net = input->dest&input->mask;
        u32 net1 = input->mask & dst;
        if(net1 == net){
            if(input->mask > number){
                number = input->mask;
                output = input;
            }
        }
    }
    return output;
    //return NULL;
}
```

- 发送IP包

  与ip_forward_packet不同，ip_send_packet发送路由器自身生成的数据包。 此函数用于发送ICMP数据包。

```
// send IP packet
//
// Different from ip_forward_packet, ip_send_packet sends packet generated by
// router itself. This function is used to send ICMP packets.
void ip_send_packet(char *packet, int len)
{
    fprintf(stderr, "TODO: send ip packet.\n");

    struct iphdr *ip = packet_to_ip_hdr(packet);
    u32 dst = ntohl(ip->daddr);
    rt_entry_t *entry = longest_prefix_match(dst);
    if (!entry) {
        log(ERROR, "Could not find forwarding rule for IP (dst:"IP_FMT") packet.", 
                HOST_IP_FMT_STR(dst));
        free(packet);
        return ;
    }

    u32 next_hop = entry->gw;
    if (!next_hop)
        next_hop = dst;

    struct ether_header *eh = (struct ether_header *)packet;
    eh->ether_type = htons(ETH_P_IP);
    struct iphdr* iph;
    iph =(struct iphdr*)(packet + ETHER_HDR_SIZE);
    iph->saddr = htonl(entry->iface->ip);
    memcpy(eh->ether_shost, entry->iface->mac, ETH_ALEN*sizeof(u8));
    iface_send_packet_by_arp(entry->iface, next_hop, packet, len);
}
```

- 拓扑结构

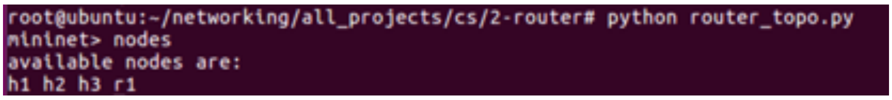
```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.cli import CLI

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

        self.addLink(h1, r1)
        self.addLink(h2, r2)
        self.addLink(r1, r2)

if __name__ == '__main__':
    topo = RouterTopo()
    net = Mininet(topo = topo, controller = None)

    h1, h2, r1, r2 = net.get('h1', 'h2', 'r1', 'r2')
    h1.cmd('ifconfig h1-eth0 10.0.1.11/24')
    h2.cmd('ifconfig h2-eth0 10.0.2.22/24')

    h1.cmd('route add default gw 10.0.1.1')
    h2.cmd('route add default gw 10.0.2.1')

    for h in (h1, h2):
        h.cmd('./scripts/disable_offloading.sh')
        h.cmd('./scripts/disable_ipv6.sh')

    r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
    r1.cmd('ifconfig r1-eth1 10.0.3.1/24')
    r1.cmd('route add -
net 10.0.2.0 netmask 255.255.255.0 gw 10.0.3.2 dev r1-eth1')

    r2.cmd('ifconfig r2-eth0 10.0.2.1/24')
    r2.cmd('ifconfig r2-eth1 10.0.3.2/24')
    r2.cmd('route add -
net 10.0.1.0 netmask 255.255.255.0 gw 10.0.3.1 dev r2-eth1')

    r1.cmd('./scripts/disable_arp.sh')
    r1.cmd('./scripts/disable_icmp.sh')
    r1.cmd('./scripts/disable_ip_forward.sh')

    r2.cmd('./scripts/disable_arp.sh')
    r2.cmd('./scripts/disable_icmp.sh')
    r2.cmd('./scripts/disable_ip_forward.sh')
```

```
    net.start()
    CLI(net)
    net.stop()
```

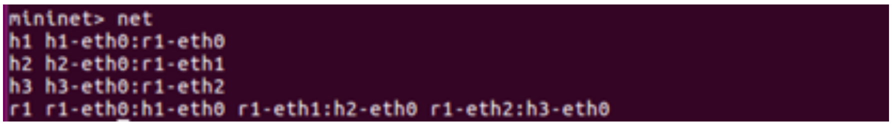
五 实验结果

5.1 实验一

执行router_topo.py文件，显示mininet下节点



显示组网信息

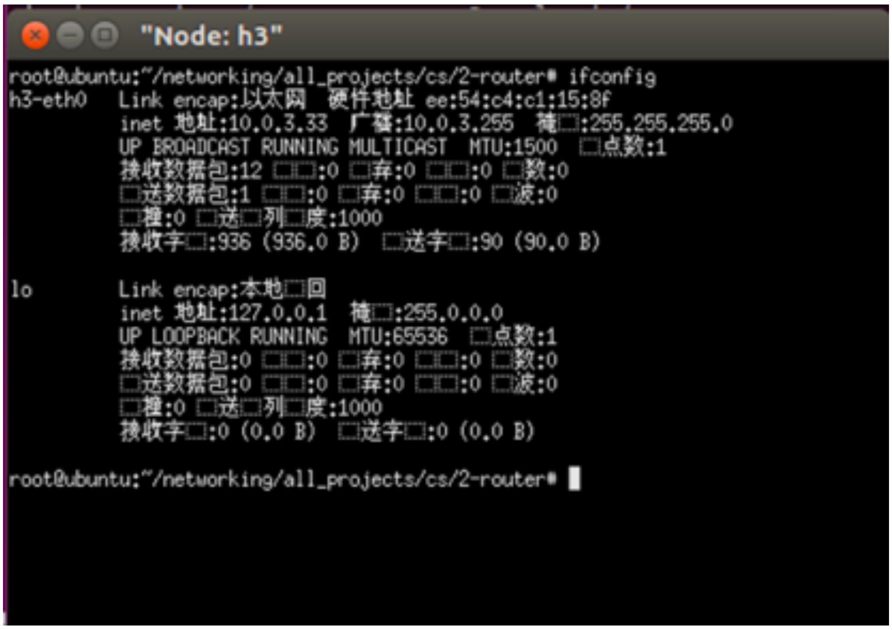


打开所有节点终端

H1节点的ifconfig结果，显示节点的IP地址

H2节点的ifconfig结果

H3节点的ifconfig结果



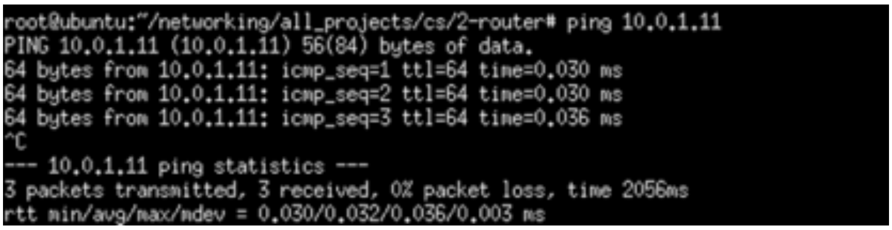
R1节点的ifconfig结果，其包括3个网口eth0，eth1和eth2分别连接h1，h2，h3

```
"Node: r1"
root@ubuntu:~/networking/all_projects/cs/2-router# ifconfig
lo        Link encap:本地□回
          inet 地址:127.0.0.1  掩□:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  □点数:1
          接收数据包:0 □□:0 □弃:0 □□:0 □数:0
          □送数据包:0 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:0 (0.0 B)  □送字□:0 (0.0 B)

r1-eth0   Link encap:以太网  硬件地址 72:08:68:a2:74:60
          inet 地址:10.0.1.1  广播:10.0.1.255  掩□:255.255.255.0
          inet6 地址: fe80::7008:68ff:fea2:7460/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  □点数:1
          接收数据包:1 □□:0 □弃:0 □□:0 □数:0
          □送数据包:12 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:90 (90.0 B)  □送字□:936 (936.0 B)

r1-eth1   Link encap:以太网  硬件地址 5a:43:62:3e:65:fd
          inet 地址:10.0.2.1  广播:10.0.2.255  掩□:255.255.255.0
          inet6 地址: fe80::5843:62ff:fe3e:65fd/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  □点数:1
          接收数据包:1 □□:0 □弃:0 □□:0 □数:0
```

在R1上执行router程序

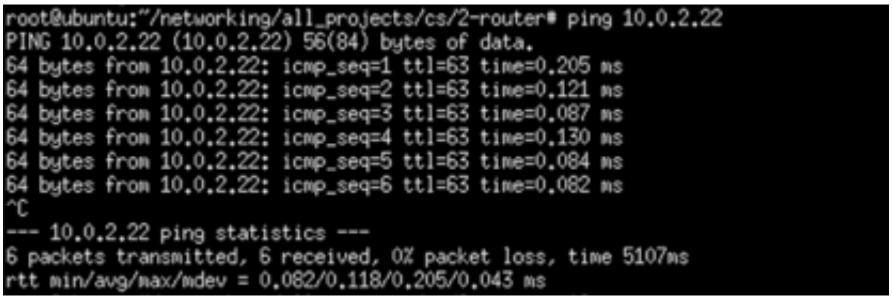
```
root@ubuntu:~/networking/all_projects/cs/2-router# ./router
DEBUG: find the following interfaces:  r1-eth0 r1-eth1 r1-eth2.
Routing table of 3 entries has been loaded.
```

在h1节点上ping自己显示的结果，表明能够ping通



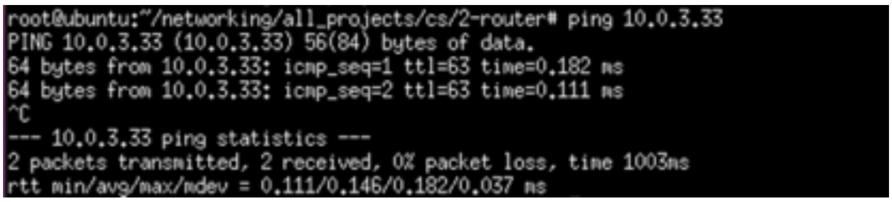
```
root@ubuntu:~/networking/all_projects/cs/2-router# ping 10.0.1.11
PING 10.0.1.11 (10.0.1.11) 56(84) bytes of data.
64 bytes from 10.0.1.11: icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from 10.0.1.11: icmp_seq=2 ttl=64 time=0.030 ms
64 bytes from 10.0.1.11: icmp_seq=3 ttl=64 time=0.036 ms
^C
--- 10.0.1.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2056ms
rtt min/avg/max/mdev = 0.030/0.032/0.036/0.003 ms
```

在h1上ping h2节点的IP地址，结果表明能够ping通，router程序执行正常



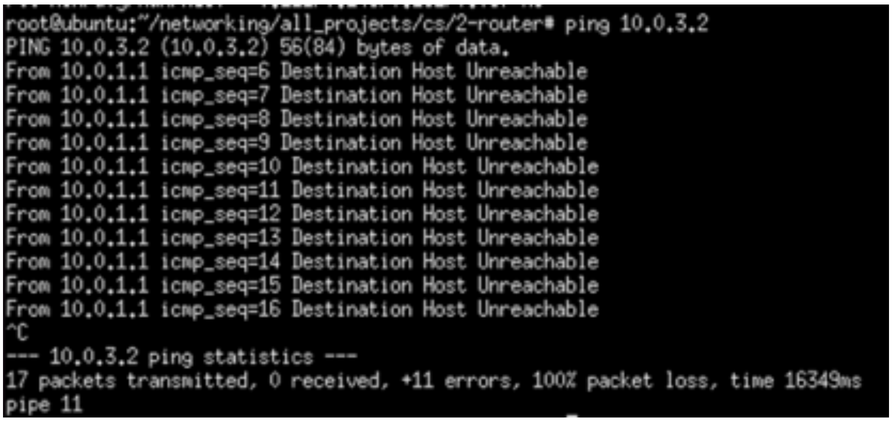
```
root@ubuntu:~/networking/all_projects/cs/2-router# ping 10.0.2.22
PING 10.0.2.22 (10.0.2.22) 56(84) bytes of data.
64 bytes from 10.0.2.22: icmp_seq=1 ttl=63 time=0.205 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.121 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.087 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.130 ms
64 bytes from 10.0.2.22: icmp_seq=5 ttl=63 time=0.084 ms
64 bytes from 10.0.2.22: icmp_seq=6 ttl=63 time=0.082 ms
^C
--- 10.0.2.22 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5107ms
rtt min/avg/max/mdev = 0.082/0.118/0.205/0.043 ms
```

在h1上ping h3的结果



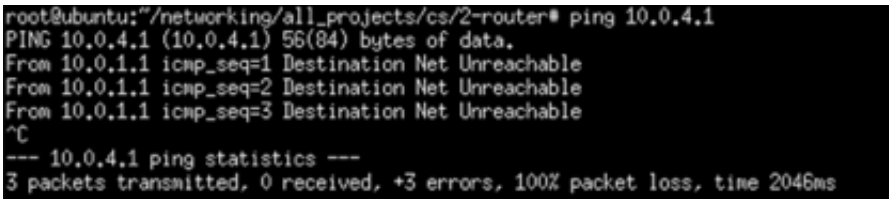
```
root@ubuntu:~/networking/all_projects/cs/2-router# ping 10.0.3.33
PING 10.0.3.33 (10.0.3.33) 56(84) bytes of data.
64 bytes from 10.0.3.33: icmp_seq=1 ttl=63 time=0.182 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.111 ms
^C
--- 10.0.3.33 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 0.111/0.146/0.182/0.037 ms
```

在h1节点ping在网段中不存在的节点ip时，显示Destination Host Unreachable



```
root@ubuntu:~/networking/all_projects/cs/2-router# ping 10.0.3.2
PING 10.0.3.2 (10.0.3.2) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=6 Destination Host Unreachable
From 10.0.1.1 icmp_seq=7 Destination Host Unreachable
From 10.0.1.1 icmp_seq=8 Destination Host Unreachable
From 10.0.1.1 icmp_seq=9 Destination Host Unreachable
From 10.0.1.1 icmp_seq=10 Destination Host Unreachable
From 10.0.1.1 icmp_seq=11 Destination Host Unreachable
From 10.0.1.1 icmp_seq=12 Destination Host Unreachable
From 10.0.1.1 icmp_seq=13 Destination Host Unreachable
From 10.0.1.1 icmp_seq=14 Destination Host Unreachable
From 10.0.1.1 icmp_seq=15 Destination Host Unreachable
From 10.0.1.1 icmp_seq=16 Destination Host Unreachable
^C
--- 10.0.3.2 ping statistics ---
17 packets transmitted, 0 received, +11 errors, 100% packet loss, time 16349ms
pipe 11
```

在h1节点ping不存在的网段IP时，显示Destination Net Unreachable



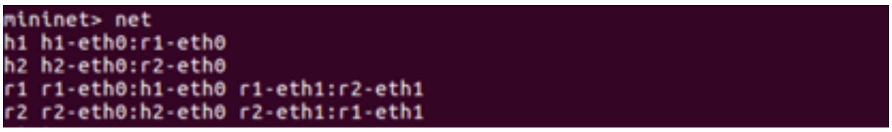
```
root@ubuntu:~/networking/all_projects/cs/2-router# ping 10.0.4.1
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Net Unreachable
From 10.0.1.1 icmp_seq=2 Destination Net Unreachable
From 10.0.1.1 icmp_seq=3 Destination Net Unreachable
^C
--- 10.0.4.1 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2046ms
```

5.2 实验二

首先组网显示如下，节点包括h1，h2，r1，r2

```
mininet> nodes
available nodes are:
h1 h2 r1 r2
```

使用net命令查看，h1连接r1，r1连接r2，r2连接h2

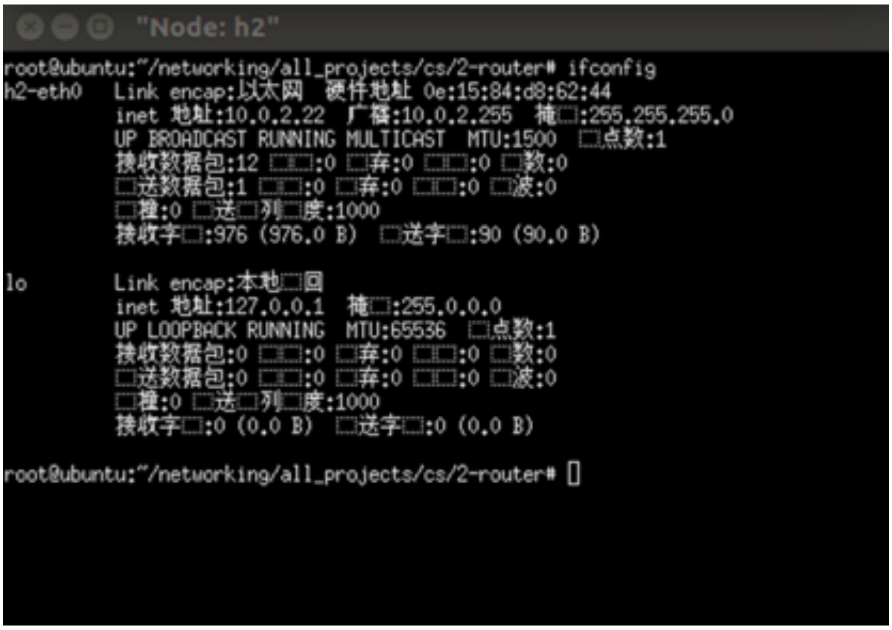


```
mininet> net
h1 h1-eth0:r1-eth0
h2 h2-eth0:r2-eth0
r1 r1-eth0:h1-eth0 r1-eth1:r2-eth1
r2 r2-eth0:h2-eth0 r2-eth1:r1-eth1
```

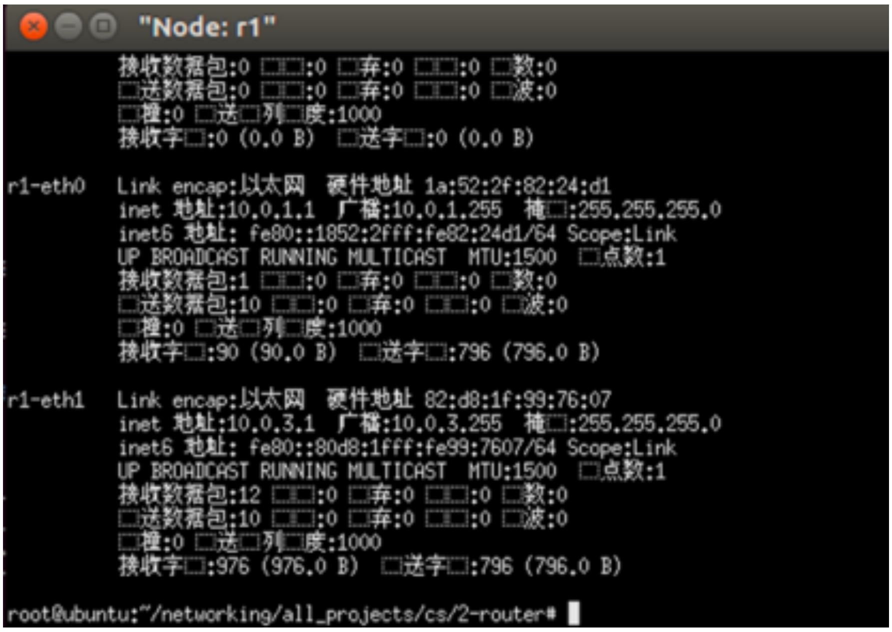
Xterm命令打开终端，在终端下分别使用ifconfig命令查看IP和网口，h1节点显示如下

```
"Node: h1"
root@ubuntu:~/networking/all_projects/cs/2-router# ifconfig
h1-eth0   Link encap:以太网  硬件地址 12:3b:56:47:58:ec
          inet 地址:10.0.1.11  广播:10.0.1.255  掩□:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  □点数:1
          接收数据包:10 □□:0 □弃:0 □□:0 □数:0
          □送数据包:1 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:796 (796.0 B)  □送字□:90 (90.0 B)

lo        Link encap:本地□回
          inet 地址:127.0.0.1  掩□:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  □点数:1
          接收数据包:0 □□:0 □弃:0 □□:0 □数:0
          □送数据包:0 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:0 (0.0 B)  □送字□:0 (0.0 B)

root@ubuntu:~/networking/all_projects/cs/2-router#
```

H2节点显示如下



R1节点显示如下



R2节点显示如下

```
"Node: r2"
          接收数据包:0 □□:0 □弃:0 □□:0 □数:0
          □送数据包:0 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:0 (0.0 B)  □送字□:0 (0.0 B)

r2-eth0   Link encap:以太网  硬件地址 72:88:27:4b:e4:a0
          inet 地址:10.0.2.1  广播:10.0.2.255  掩□:255.255.255.0
          inet6 地址: fe80::7088:27ff:fe4b:e4a0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  □点数:1
          接收数据包:1 □□:0 □弃:0 □□:0 □数:0
          □送数据包:12 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:90 (90.0 B)  □送字□:976 (976.0 B)

r2-eth1   Link encap:以太网  硬件地址 76:49:b6:84:b9:a3
          inet 地址:10.0.3.2  广播:10.0.3.255  掩□:255.255.255.0
          inet6 地址: fe80::7449:b6ff:fe84:b9a3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  □点数:1
          接收数据包:10 □□:0 □弃:0 □□:0 □数:0
          □送数据包:12 □□:0 □弃:0 □□:0 □波:0
          □撞:0 □送□列□度:1000
          接收字□:796 (796.0 B)  □送字□:976 (976.0 B)

root@ubuntu:~/networking/all_projects/cs/2-router#
```

在r1和r2节点分别执行router程序

```
root@ubuntu:~/networking/all_projects/cs/2-router# ./router
DEBUG: find the following interfaces:  r1-eth0 r1-eth1.
Routing table of 3 entries has been loaded.
```

```
root@ubuntu:~/networking/all_projects/cs/2-router# ./router
DEBUG: find the following interfaces:  r2-eth0 r2-eth1.
Routing table of 3 entries has been loaded.
```

在h1节点ping h2节点，显示结果能够正常ping通

```
root@ubuntu:~/networking/all_projects/cs/2-router# ping 10.0.2.22
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=62 time=0.238 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=62 time=0.150 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=62 time=0.272 ms
^C
--- 10.0.2.22 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2050ms
rtt min/avg/max/mdev = 0.150/0.220/0.272/0.051 ms
```

在h1节点执行traceroute，能够发现其r1入端口通路

```
root@ubuntu:~/networking/all_projects/cs/2-router# traceroute 10.0.2.22
traceroute to 10.0.2.22 (10.0.2.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.299 ms  0.280 ms  0.273 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * 10.0.2.22 (10.0.2.22)  0.207 ms  0.204 ms
```

我们的拓扑结构由两个host和两个router构成，因此ping后他的arpcahe为：

- R1

```
*****************
TODO: forward ip packet.
TODO: longest prefix match for the packet.
TODO: lookup ip address in arp cache.
arpcache is:
10.0.1.11       6a:92:23:86:53:f3       1558492325      1
10.0.3.2        62:b7:1c:53:37:4a       1558492326      1
*****************
```

- R2

```
****************
TODO: forward ip packet.
TODO: longest prefix match for the packet.
TODO: lookup ip address in arp cache.
arpcache is:
10.0.3.1        7a:96:f6:05:24:88       1558492325      1
10.0.2.22       6e:c2:31:8b:ef:ee       1558492326      1
****************
```