实验报告

孔静 2014K8009929022

November 9, 2017

1 实验题目

路由器转发实验

2 实验内容

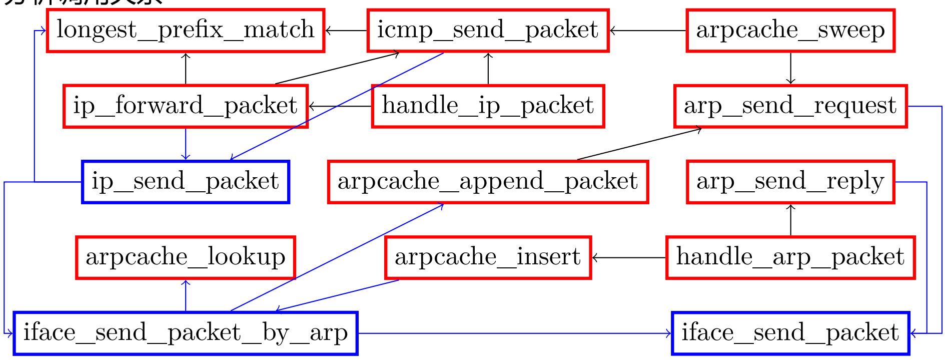
- 运行给定网络拓扑 (topo/router_topo.py)
- 在 r1 上执行路由器程序
  - 首先, 执行脚本 (disable_arp.sh, disable_icmp.sh, disable_ip_forward.sh), 禁止协议栈的相应功能
  - 在 r1 中运行./router, 进行数据包的处理
- 在 h1 上进行 ping 实验
  - Ping 10.0.1.1 (r1), 能够 ping 通
  - Ping 10.0.2.22 (h2), 能够 ping 通
  - Ping 10.0.3.33 (h3), 能够 ping 通
  - Ping 10.0.2.11, 返回 ICMP Destination Host Unreachable
  - Ping 10.0.4.1, 返回 ICMP Destination Net Unreachable

3 实验流程

- 确认函数功能
  - longest_prefix_match
  - ip_forward_packet
  - handle_ip_packet
  - icmp_send_packet

  - arpcache_lookup
  - arpcache_append_packet
  - arpcache_insert
  - arpcache_sweep
  - arp_send_request
  - arp_send_reply
  - handle_arp_packet

- 分析调用关系



- 编写函数代码

```
////////
//ip.c//
////////
rt_entry_t *longest_prefix_match(u32 dst)
{
  // fprintf(stderr, "TODO: longest prefix match for the packet.\n
    ");
  // return NULL;
  rt_entry_t *entry, *longest = NULL;
  list_for_each_entry(entry, &rtable, list){
    if((dst & entry->mask) == (entry->dest & entry->mask)){
      if(longest == NULL){
        longest = entry;
      }
      else if(entry->mask > longest->mask){
        longest = entry;
      }
    }
  }
  return longest;
}
void ip_forward_packet(u32 ip_dst, char *packet, int len)
{
  // fprintf(stderr, "TODO: forward ip packet.\n");
  struct iphdr *ip = packet_to_ip_hdr(packet);
  rt_entry_t *entry = longest_prefix_match(ip_dst);
  if(!entry){
    icmp_send_packet(packet, len, (u8)3, (u8)0);
    free(packet);
    return;
  }
  ip->ttl--;
  ip->checksum = ip_checksum(ip);
  if(ip->ttl <= 0){
    icmp_send_packet(packet, len, (u8)11, (u8)0);
    free(packet);
    return;
  }
  ip_send_packet(packet, len);
}
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
  // fprintf(stderr, ”TODO: handle ip packet: echo the ping packet, and forward other IP packets.\n”);
  struct iphdr *ip = packet_to_ip_hdr(packet);
  u32 ip_dst = ntohl(ip->daddr);
  if(iface->ip == ip_dst){
    icmp_send_packet(packet, len, (u8)0, (u8)0);
    free(packet);
  }
  else{
    ip_forward_packet(ip_dst, packet, len);
  }
}
//////////
//icmp.c//
//////////
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 *in_ip = packet_to_ip_hdr(in_pkt);
  u32 dst = ntohl(in_ip->saddr);
  rt_entry_t *entry = longest_prefix_match(dst);
  int out_len, source, destination, new_len;
  switch(type){
    case 0:
      out_len = len - IP_HDR_SIZE(in_ip) + IP_BASE_HDR_SIZE;
      source = 0; destination = IP_HDR_SIZE(in_ip);
      break;
    case 3:
    case 11:
      out_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + ICMP_HDR_SIZE + IP_HDR_SIZE(in_ip) + 8;
      source = ICMP_HDR_SIZE; destination = 0;
      break;
  }
  char *out_pkt;
  out_pkt = malloc(out_len);
  struct icmphdr* icmp_hdr = (struct icmphdr *)(out_pkt + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
  memset(icmp_hdr, 0, ICMP_HDR_SIZE);
  new_len = out_len - IP_BASE_HDR_SIZE - ETHER_HDR_SIZE;
  memcpy((char *)icmp_hdr + source, (char *)in_ip + destination, new_len - source);
  icmp_hdr->type = type;
  icmp_hdr->code = code;
  icmp_hdr->checksum = icmp_checksum(icmp_hdr, new_len);
  struct ether_header *out_eh = (struct ether_header *)out_pkt;
  out_eh->ether_type = ntohs(ETH_P_IP);
  struct iphdr *out_ip = packet_to_ip_hdr(out_pkt);
  ip_init_hdr(out_ip, entry->iface->ip, dst, out_len -
    ETHER_HDR_SIZE, 1);
  ip_send_packet(out_pkt, out_len);
}
/////////
//arp.c//
/////////
//request和reply功能相似,顾合并成一个函数
void arp_send_option(iface_info_t *iface, u32 dst_ip, struct
    ether_arp *req_hdr, u16 option){

  char *packet;
  int len = ETHER_HDR_SIZE + sizeof(struct ether_arp);
  packet = (char *)malloc(len);
  struct ether_header* eh = (struct ether_header*)packet;
  struct ether_arp* eh_arp = (struct ether_arp*)(packet +
    ETHER_HDR_SIZE);
  eh->ether_type = htons(ETH_P_ARP);
  memcpy((char *)eh->ether_shost, (char *)iface->mac, ETH_ALEN);
  eh_arp->arp_hrd = htons(0x01);
  eh_arp->arp_pro = htons(ETH_P_IP);
  eh_arp->arp_hln = ETH_ALEN;
  eh_arp->arp_pln = 4;
  eh_arp->arp_op = htons(option);
  eh_arp->arp_spa = htonl(iface->ip);

  memset(eh_arp->arp_tha, 0, ETH_ALEN);
  memcpy((char *)eh_arp->arp_sha, (char *)iface->mac, ETH_ALEN);
  switch(option){
    case ARPOP_REQUEST:
      memset(eh->ether_dhost, 0xff, ETH_ALEN);
      eh_arp->arp_tpa = htonl(dst_ip);
      break;
    case ARPOP_REPLY:
      memcpy(eh->ether_dhost, req_hdr->arp_sha, ETH_ALEN);
      eh_arp->arp_tpa = req_hdr->arp_spa;
      break;
  }
  iface_send_packet(iface, packet, len);
}
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
  // fprintf(stderr, "TODO: send arp request when lookup failed in
      arpcache.\n");
  arp_send_option(iface, dst_ip, NULL, ARPOP_REQUEST);
}
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr
    )
{
  // fprintf(stderr, "TODO: send arp reply when receiving arp
    request.\n");
```

```c
  arp_send_option(iface, 0, req_hdr, ARPOP_REPLY);
}
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* eh_arp = (struct ether_arp*)(packet + ETHER_HDR_SIZE);
  u16 op = ntohs(eh_arp->arp_op);
  u32 ip = ntohl(eh_arp->arp_tpa);

  if(op == ARPOP_REQUEST && ip == iface->ip){
    arp_send_reply(iface, eh_arp);
  }
  else if(op == ARPOP_REPLY && ip == iface->ip){
    arpcache_insert(ntohl(eh_arp->arp_spa), eh_arp->arp_sha);
  }
}
//////////////
//arpcache.c//
//////////////
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
  // fprintf(stderr, "TODO: lookup ip address in arp cache.\n");
  // return 0;
  int result = 0;
    pthread_mutex_lock(&arpcache.lock);
    for(int i = 0; i < MAX_ARP_SIZE; i++){
        if(ip4 == arpcache.entries[i].ip4 && arpcache.entries[i].valid){
            memcpy(mac, arpcache.entries[i].mac, ETH_ALEN);
            result = 1;
            break;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
    return result;
}
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;
    struct cached_pkt *pkt_entry;
    list_for_each_entry(req_entry, &arpcache.req_list, list){
        if (ip4 == req_entry->ip4){
            goto append_packet;
        }
    }
    req_entry = (struct arp_req *)malloc(sizeof(struct arp_req));
    init_list_head(&req_entry->list);
    list_add_tail(&req_entry->list, &arpcache.req_list);
    init_list_head(&req_entry->cached_packets);
    req_entry->iface = (iface_info_t *)malloc(sizeof(iface_info_t));
    memcpy(req_entry->iface, iface, sizeof(iface_info_t));
    req_entry->ip4 = ip4;
    req_entry->sent = 0;
    req_entry->retries = 0;
    arp_send_request(iface, ip4);
append_packet:
    pkt_entry = (struct cached_pkt *)malloc(sizeof(struct
    cached_pkt));
    init_list_head(&pkt_entry->list);
    list_add_tail(&pkt_entry->list, &req_entry->cached_packets);
    pkt_entry->packet = packet;
    // pkt_entry->packet = (char *)malloc(len);
    // memcpy(pkt_entry->packet, packet, len);
    pkt_entry->len = len;

    pthread_mutex_unlock(&arpcache.lock);
}
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);
  struct arp_req *req_entry, *req_q;
  list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list),
     list) {
    if(req_entry->ip4 == ip4){
      break;
    }
  }

  int index;
  for(index = 0; index < MAX_ARP_SIZE; index++){
    if(arpcache.entries[index].valid == 0){
      break;
    }
  }
  if(index == MAX_ARP_SIZE){
    index = rand() % MAX_ARP_SIZE;
  }
  memcpy(arpcache.entries[index].mac, mac, ETH_ALEN);
  arpcache.entries[index].ip4 = ip4;
  arpcache.entries[index].valid = 1;
  arpcache.entries[index].added = 0;
  pthread_mutex_unlock(&arpcache.lock);
  struct cached_pkt *pkt_entry, *pkt_q;
  list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->
    cached_packets), list) {
    iface_send_packet_by_arp(req_entry->iface, ip4, pkt_entry->
    packet, pkt_entry->len);
    free(pkt_entry);
  }
  list_delete_entry(&(req_entry->list));
  free(req_entry->iface);
  free(req_entry);
}
void *arpcache_sweep(void *arg)
{
  struct arp_req *req_entry , *req_q;
  struct cached_pkt *pkt_entry, *pkt_q;
  while (1) {
    sleep(1);
    // fprintf(stderr, "TODO: sweep arpcache periodically: remove old entries, resend arp requests.\n");
    pthread_mutex_lock(&arpcache.lock);
    for(int index = 0; index < MAX_ARP_SIZE; index++){
      if(++arpcache.entries[index].added > ARP_ENTRY_TIMEOUT){
        arpcache.entries[index].valid = 0;
      }
    }
    list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
      if(req_entry->retries >= ARP_REQUEST_MAX_RETRIES){
        list_delete_entry(&(req_entry->list));
        pthread_mutex_unlock(&arpcache.lock);
        list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list){
          icmp_send_packet(pkt_entry->packet, pkt_entry->len, (u8)3, (u8)1);
          free(pkt_entry->packet);
          free(pkt_entry);
        }
        pthread_mutex_lock(&arpcache.lock);

        free(req_entry->iface);
        free(req_entry);
      }
      else{
        arp_send_request(req_entry->iface, req_entry->ip4);
        req_entry->sent = time(NULL);
        req_entry->retries += 1;
      }

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

- 进行结果测试

```
#略去一些代码
os.system("make clean")
os.system("make")
net.start()
h1, r1 = net.get('h1', 'r1')

r1.cmd('./scripts/disable_arp.sh')
r1.cmd('./scripts/disable_icmp.sh')
r1.cmd('./scripts/disable_ip_forward.sh')
r1.cmd('./router &')
time.sleep(0.2)
print h1.cmd('ping 10.0.1.1 -c 5')
print h1.cmd('ping 10.0.2.22 -c 5')
print h1.cmd('ping 10.0.3.33 -c 5')
print h1.cmd('ping 10.0.2.11 -c 5')
print h1.cmd('ping 10.0.4.1 -c 5')
net.stop()
```

4 实验结果

```
kj@12-ubuntu:~/Desktop/07-router$ sudo python test.py
rm -f *.o router
gcc -c -g -Wall -Iinclude arp.c -o arp.o
In file included from arp.c:12:0:
include/log.h:12:23: warning: 'this_log_level' defined but not used [-Wunused-variable]
static enum log_level this_log_level = DEBUG;
^
include/log.h:14:20: warning: 'log_level_str' defined but not used [-Wunused-variable]
static const char *log_level_str[] = { "DEBUG", "INFO", "WARNING", "ERROR" };
^
gcc -c -g -Wall -Iinclude arpcache.c -o arpcache.o
gcc -c -g -Wall -Iinclude icmp.c -o icmp.o
gcc -c -g -Wall -Iinclude ip.c -o ip.o
gcc -c -g -Wall -Iinclude main.c -o main.o
gcc -c -g -Wall -Iinclude packet.c -o packet.o
gcc -c -g -Wall -Iinclude rtable.c -o rtable.o
gcc  arp.o arpcache.o icmp.o ip.o main.o packet.o rtable.o -o router -lpthread
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=0.130 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.051 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.084 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.233 ms
64 bytes from 10.0.1.1: icmp_seq=5 ttl=64 time=0.118 ms

--- 10.0.1.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4103ms
rtt min/avg/max/mdev = 0.051/206.291/1030.972/412.340 ms, pipe 2

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=0.147 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.057 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.100 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.155 ms
64 bytes from 10.0.2.22: icmp_seq=5 ttl=63 time=0.140 ms

--- 10.0.2.22 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4092ms
rtt min/avg/max/mdev = 0.057/0.119/0.155/0.039 ms

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=0.077 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.067 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.088 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.052 ms
```

```
64 bytes from 10.0.3.33: icmp_seq=5 ttl=63 time=0.140 ms

—— 10.0.3.33 ping statistics ——
5 packets transmitted, 5 received, 0% packet loss, time 4091ms
rtt min/avg/max/mdev = 0.052/0.084/0.140/0.032 ms

PING 10.0.2.11 (10.0.2.11) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Host Unreachable

—— 10.0.2.11 ping statistics ——
5 packets transmitted, 0 received, +1 errors, 100% packet loss, time 4087ms
pipe 5

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
From 10.0.1.1 icmp_seq=5 Destination Net Unreachable

—— 10.0.4.1 ping statistics ——
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 4077ms
```

5 结果分析

ping 命令会通过 ICMP 协议发送 ICMP 数据包, 步骤如下:

1. 应用程序构造数据包, 该示例是产生 ICMP 包, 被提交给内核 (网络驱动程序)

2. 内核检查是否能够转化该 IP 地址为 MAC 地址, 也就是在本地的 ARP 缓存中查看 IP-MAC 对应表

3. 如果存在该 IP-MAC 对应关系, 那么跳到步骤 7; 否则继续以下步骤

4. 内核进行 ARP 广播, 目的地的 MAC 地址是 FF-FF-FF-FF-FF-FF,ARP 命令类型为 REQUEST(1), 其中包含有自己的 MAC 地址

5. 当 192.168.1.2 主机接收到该 ARP 请求后, 就发送一个 ARP 的 REPLY(2) 命令, 其中包含自己的 MAC 地址

6. 本地获得 192.168.1.2 主机的 IP-MAC 地址对应关系, 并保存到 ARP 缓存中

7. 内核将把 IP 转化为 MAC 地址, 然后封装在以太网头结构中, 再把数据发送出去