路由器

郭恩惠2020010548

代码逻辑

```
void handlePacket(...) {
    // 通过以太网帧头部的type字段判断是arp还是ip包
    if(ethertype_arp) {
        handleArp();
    } else if(ethertype_ip) {
        handleIp();
    }
}

void handleArp() {
    // 通过arp头部的Opcode字段判断是request还是reply
    if(arp_op_request) {
        auto arp_pkt = ... ; // 构建arp包，封装为以太网帧
        auto outIface = getRoutingTable.lookup(...); //通过路由表查询出接口
        sendPacket(arp_pkt, outIface);
    } else if(arp_op_reply) {
        // 将IP-MAC映射存储在ARP Cache
        auto arp_request = m_arp.insertArpEntry(...);
        // 将队列中所有与arp_request相关的包全部发送出去
    }
}

void handleIp() {
    auto ip_h = ...; // IP头部
    if(/*目的ip在路由器*/) {
       // 判断IP头部字段Protocol
       if(ip_protocol_icmp) {    // 如果是ICMP
           if(/*是 Echo 消息*/) {
               sendIcmpEchoReply(); // 返回Echo Reply
           }
       } else {
           sendIcmp(3, 3); // 返回ICMP Port Unreachable
       }
    } else { // 待转发
        ip_h->ip_ttl -= 1; // TTL减1
        ip_h->ip_sum = 0;
        ip_h->ip_sum = cksum(...); // 重新计算校验和
        if(ip_h->ip_ttl == 0) {
            // TTL为0，不转发，返回超时消息
            sendIcmp(11, 0);
        } else {
            forwardPacket(...);
        }
    }
}
```

Ethernet Frame

将IP数据报封装为以太网帧时，设数据报的目的IP地址为 `dstip`

首先通过 `dstip` 查询**路由表**，得到**出接口**（即包从哪个接口发出）

然后通过 `dstip` 查询**ARP表**，得到**下一跳的MAC地址**

以太网帧的**源MAC地址**就是 出接口MAC地址

目的MAC地址是 下一跳MAC地址

IP

> 计算校验和之前先将头部的 `ip_sum` 字段置为0，因为它不参与校验和计算

只有IP头部参与校验和计算，IP数据部分不参与

`version` 和 `header length` 2个字段分别为4和5，在构建IP数据报的时候必须赋值

`total length` 是整个IP数据报的长度，包括IP头部+IP数据

ICMP

> 路由器生成ICMP消息时，IP数据报的源地址字段可以是路由器任意接口的ip地址
>
> 但我在实现的时候，ICMP的源IP都设置为以太网帧 入接口 的IP

入接口即以太网帧从哪个接口进入

ICMP头部 + ICMP数据 都参与校验和计算

以下所说的 Echo Reply/Request 指的是ICMP报文，不包含IP头部

Echo Reply 数据部分应该和 Echo Request 对应，因为有时间戳用于计算time

Echo Reply 头部的id和seq需要和Echo Request对应，用于发送方区分ICMP报文

Echo Reply 和 Echo Request 不同的字段只有 ICMP头部的 type 和 checksum

ArpCache

`ArpCache` 类的所有方法都会访问互斥资源 `m_cacheEntries` 和 `m_arpRequests`，并通过锁 `m_mutex` 保护互斥资源的访问

类 `ArpCache` 有一个函数 `periodicCheck...()`，该函数中需要判断同一个目的IP的ARP请求是否超过5次，如果是，需要返回 `ICMP Host Unreachable`

```
void ticker() {
    while (!m_shouldStop) {
        ...
        {
            std::lock_guard<std::mutex> lock(m_mutex); // 请求锁
            periodicCheckArpRequestsAndCacheEntries();
```

```
        }
        ...
    }
}
void periodicCheckArpRequestsAndCacheEntries() {
    ...
    if(ntimes ⩾ 5) {
        m_router.sendIcmp(3, 1); // Host Unreachable
    }
    ...
}
```

我在类 `SimpleRouter` 中定义了一个函数 `sendIcmp` 用于发送ICMP消息

```
void sendIcmp() {
    // 构建ICMP头部+数据
    auto ipdata = ...; // 构建IP数据报
    sendIp(ipdata);
}
void sendIp() {
    ...
    auto frame = ...; // 封装为以太网帧
    // 查询ARP缓存，找到下一跳的MAC地址
    auto arp_entry = m_arp.lookup(ip_h->ip_dst);
    if(arp_entry == nullptr) {
        // 添加到ARP请求队列
        m_arp.queueRequest(ip_h->ip_dst, frame, outIface->name, inIface);
        return;
    }
    memcpy(eth_h->ether_dhost, arp_entry->mac.data(), ETHER_ADDR_LEN);
    sendPacket(frame, outIface->name);
}
```

`sendIp` 函数调用了 `ArpCache::lookup` 和 `ArpCache::queueRequest` 2个函数

也就是说 `sendIcmp(sendIp)` 函数会请求锁 `m_mutex`

> 因为 `ticker` 函数先请求锁，再调用 `periodicCheck`
>
> 如果在 `periodicCheck` 函数中再执行 `sendIcmp` 函数
>
> 而 `sendIcmp` 又会请求锁，此时就导致死锁问题

解决办法是改写 `ticker` 和 `periodicCheck`

```
void ticker() {
    while (!m_shouldStop) {
        ...
        vector<Buffer> toSendPkts;
        {
            std::lock_guard<std::mutex> lock(m_mutex); // 请求锁
            toSendPkts = periodicCheckArpRequestsAndCacheEntries();
        }
        // 此时锁释放
```

```
        for(auto pkt : toSendPkts) {
            sendIcmp(3, 3);
        }
        ...
    }
}

vector<Buffer> periodicCheckArpRequestsAndCacheEntries() {
    ...
    vector<Buffer> toSendPkts;
    if(ntimes ⩾ 5) {
        toSendPkts.push_bach(pkt);
    }
    ...
    return toSendPkts
}
```