交换机转发实验报告

李昊宸

2017K8009929044

（一）交换机转发实现

一、实验内容

1. 交换机学习实现：

查询操作：每收到一个数据包，根据目的 MAC 地址查询相应转发条目，如果查询到对应条目，则根据相应转发端口转发数据包，更新访问时间；否则，广播该数据包

插入操作：每收到一个数据包，如果其源 MAC 地址在转发表中，更新访问时间；否则，将该地址与入端口的映射关系写入转发表

老化操作：每秒钟运行一次老化操作，删除超过 30 秒未访问的转发条目

2. 使用 iperf 和给定的拓扑进行实验，对比交换机转发与集线器广播的性能从一个端节点 ping 另一个端节点

3. 思考题：

我们知道，网络中存在广播包，其目的 MAC 地址设置为全 0xFF ，例如 ARP 请求数据包。这种广播包对交换机行为逻辑有什么影响？

理论上，足够多个交换机可以连接起全世界所有的终端。请问，使用这种方式连接亿万台主机是否技术可行？并说明理由。

二、实验流程

1. 搭建实验环境

include：相关头文件

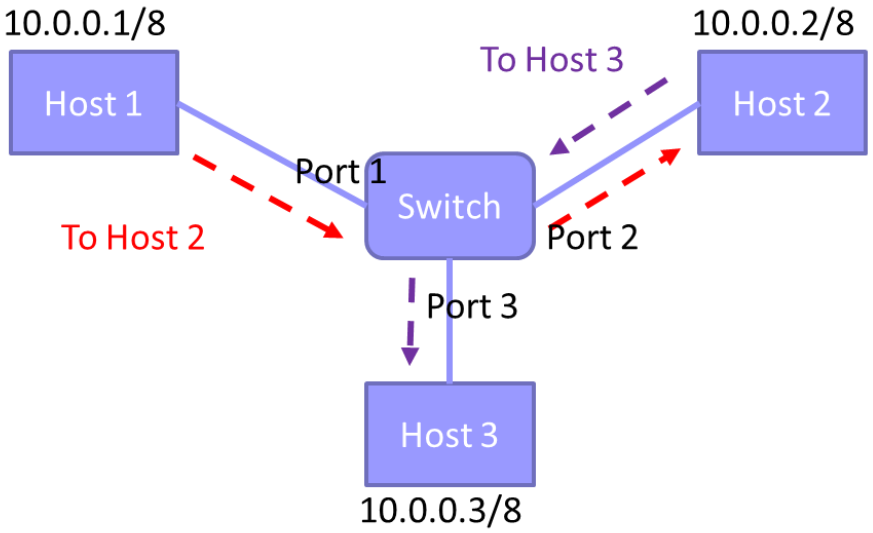
scripts：禁用 TCP Offloading、IPV6 功能，避免抓到无用包

main.c： Switch 的代码实现，编译后在交换机结点上运行

mac.c： 实现的 mac_port_mac 相关操作：

```
iface_info_t *lookup_port(u8 mac[ETH_ALEN]);
void insert_mac_port(u8 mac[ETH_ALEN], iface_info_t *iface);
int sweep_aged_mac_port_entry();
void broadcast_packet(iface_info_t *iface, const char *packet, int len);
void handle_packet(iface_info_t *iface, char *packet, int len);
```

three_nodes_bw.py：实现如下图的节点拓扑



图一 三节点网络拓扑

交换机的实现：

1）转发表的**构造**

目的地址	转发端口	老化时间
Host 1 MAC Addr	Port 1	30 sec
Host 2 MAC Addr	Port 2	30 sec
Host 3 MAC Addr	Port 3	30 sec

图二 转发表

转发表用于存储目的地址和转发端口的映射关系，实际建立的转发表每一个条目还附带老化时间一项，用于表示转发表项的有效期。

交换机转发表的构造基于一个**基本假设**：如果收到的数据包的源 MAC 地址为 X，端口为 Y，那么将数据包从端口 Y 发送出去就可以到达 X。

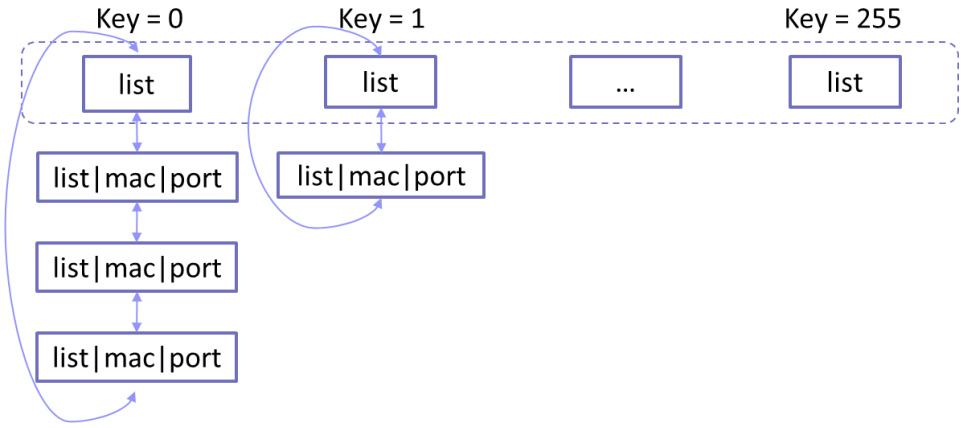
所以，转发表的插入操作为：每收到一个数据包，如果源 MAC 地址不在表中，就将该 MAC 地址与收到该数据包的端口的映射关系写入转发表；否则，就更新转发表中该项的访问时间（也即刷新老化时间）。老化操作每秒钟运行一次，删除超过 30 秒未访问的老化条目。

对于交换机而言，每收到一个数据包，都要根据目的 MAC 地址查询转发所需要的端口名字。如果该目的 MAC 地址在转发表中，就根据对应转发表项中的转发端口将该包发送出去；否则就广播该数据包。

2）转发表的**查询**

构造转发表一个比较自然的想法是使用一个链表。但是，查询链表的时间过长，最差情况下可能要遍历整个链表，造成大量时间的浪费。

所以，我们采用**对 MAC 地址 Hash**，根据 key 值进入对应链表表项中查找。



图三 转发表的 Hash 结构

2. 启动脚本

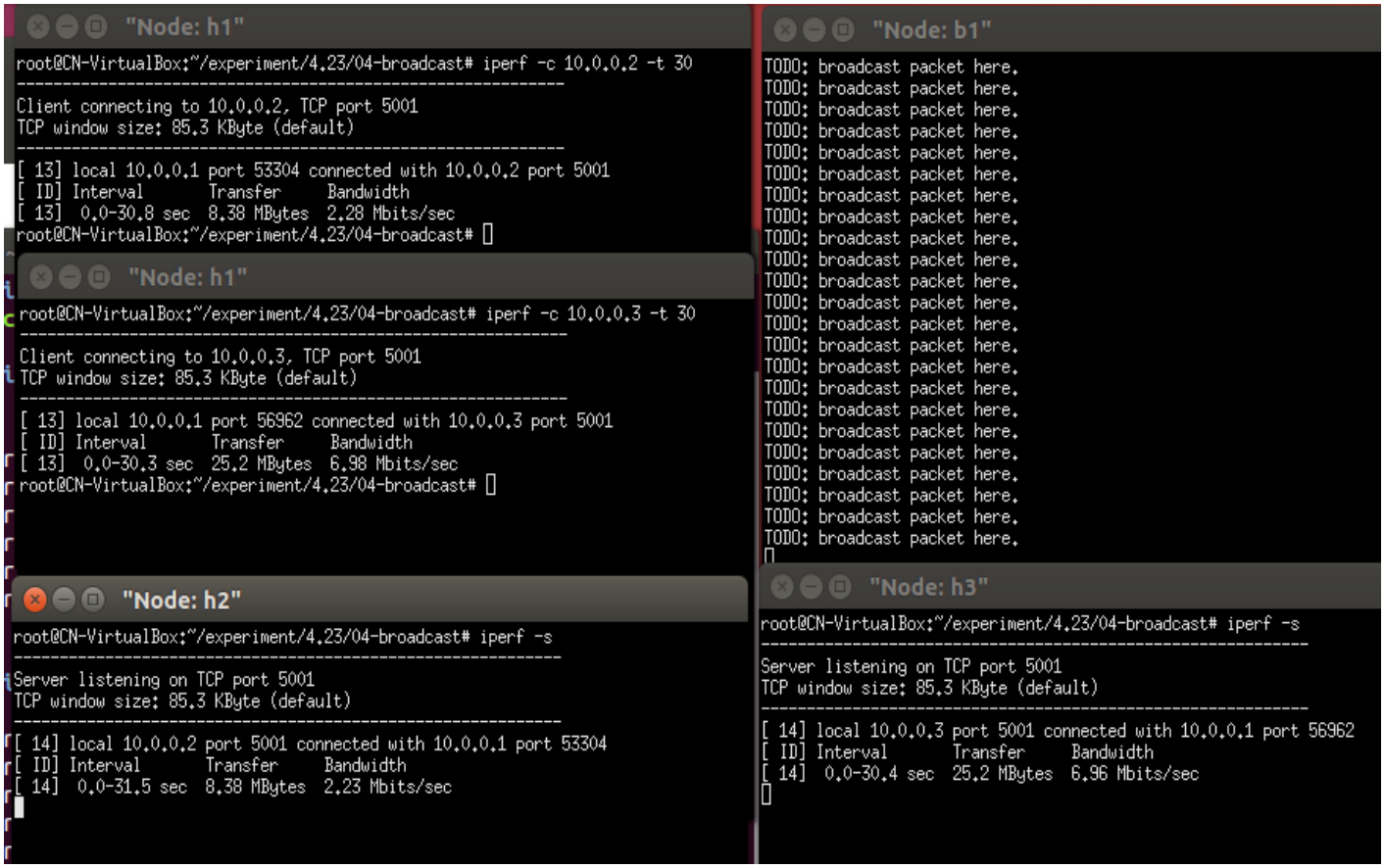
1) 交换机转发效率测试

```
make all
sudo python three_nodes_bw.py
mininet> xterm h1 h1 h2 h3 s1
s1# ./switch
h2# iperf -s
h3# iperf -s
h1# iperf -c 10.0.0.2 -t 30
h1# iperf -c 10.0.0.3 -t 30
mininet> quit
```

上述过程是以 h2、h3 作为服务器，h1 作为客户（通过启动两个终端）同时向二者进行访问

三、实验结果及分析

1. 实验结果



图四 Hub 的转发效率测试

我们先来回顾一下上周实验中广播网络的效率测试。

实验条件：h1：client，同时向 h2 和 h3 请求服务

h2：server

h3：server

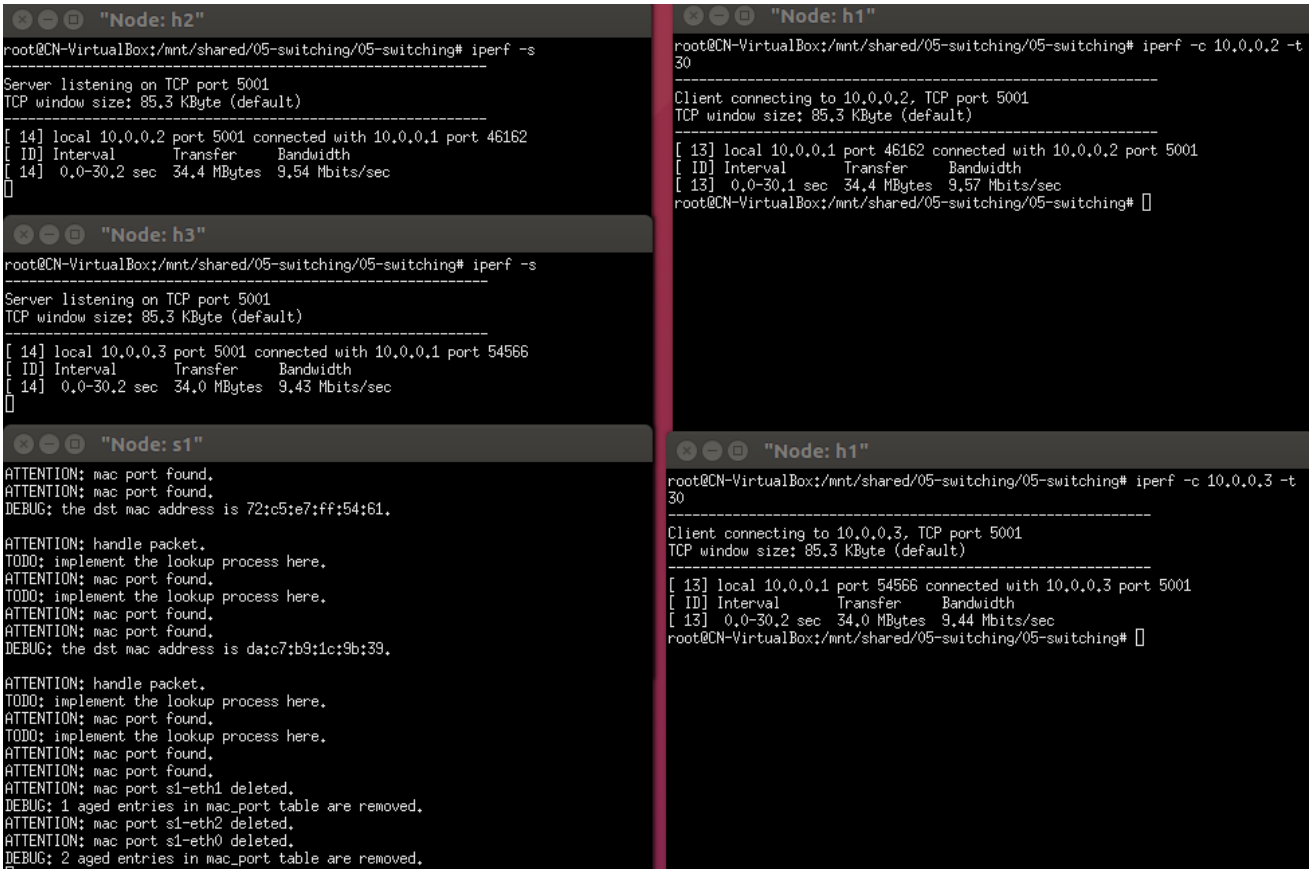
b1：hub

并发请求时，h1 向 b1 以 20Mbps 的速率发送数据包，其中一半的目的主机是 h2，另一半的目的主机是 h3。数据包到达 b1 后，开始向 h2 和 h3 转发。每一个数据包，都会被复制后发往 h2 和 h3，所以对于 h2，虽然其接受速率理论上最高为 10Mbps，但是其中约有一半是 h1 要发往 h3 的包，对于 h2 来言属于无效包，白白占用带宽，真正有用的是 h1 要发给 h2 的包。对于 h3 也是类似的情况。

所以，如果 hub 对包的转发顺序是绝对随机的话，h1 与 h2 之间的传输速率和 h1 与 h3 之间的传输速率都应在 5Mbps 之下。

实际测试中，h1-h2：2.28Mbps h1-h3：6.98Mbps，二者相加为 9.26Mbps，小于 10Mbps，满足理论分析。

再来看这次实验实现的交换机的转发效率：



图五 Switch 的转发效率测试

下面详细描述交换机进行转发的过程：

实验条件：h1：client，同时向 h2 和 h3 请求服务

h2：server

h3：server

s1：hub

h1 想要给 h2、h3 发包，首先需要知道二者的 MAC 地址。（以 h1 给 h2 通信为例）h1（IP:10.0.0.1）首先在自己的 **ARP 缓存中查找** h2（IP：10.0.0.2）的 MAC 地址，查找发现不存在，于是向 s1 节点发送一个 **ARP 广播包**，目的 MAC 地址为 FF:FF:FF:FF:FF:FF，意图为向局域网内所有主机询问 h2 的 MAC 地址。

s1 节点从 PORT 1 收到 h1 发来的数据包。解析该包，获得 h1 的 MAC 地址为 Host 1 MAC Addr，于是向转发表中加入 h1 的转发项：

目的地址	转发端口	老化时间
Host 1 MAC Addr	Port 1	30 sec

图六 实时更新的 Switch 的转发表

继续解析，s1 发现该数据包的目的 MAC 地址为 FF:FF:FF:FF:FF:FF，转发表中没有该项目，于是 s1 将该包向所有其他端口**广播**。

该包从 PORT 2 发送出去，到达 h2。h2 向自己的 **ARP 缓存中写入** h1 的 MAC 地址，随后向 s1 节点发送一个 **ARP 单播包**，源 MAC 地址为 h2，目的 MAC 地址为 h1。

s1 节点从 PORT 2 收到 h2 发来的数据包。解析该包，获得 h2 的 MAC 地址为 Host 2 MAC Addr，于是向转发表中加入 h2 的转发项：

目的地址	转发端口	老化时间
Host 1 MAC Addr	Port 1	30 sec
Host 2 MAC Addr	Port 2	30 sec

图七 实时更新的 Switch 的转发表

继续解析，s1 发现该数据包的目的 MAC 地址为 Host 1 MAC Addr，转发表中该项的转发端口为 PORT 1，于是 s1 将该包从 PORT 1 **转发**。

h1 接收了 s1 转发来的数据包。解析获得 h2 的 MAC 地址，**写入 ARP 缓存**。随后 h1 再向 h2 发包就从 ARP 缓存中查找 h2 的 MAC 地址，然后填入报文首部。h1 与 h3 间的通信与上类似。

并发请求时，h1 向 b1 以 20Mbps 的速率发送数据包，其中一半的目的主机是 h2，另一半的目的主机是 h3。数据包到达 s1 后，发给 h2 的包从 PORT 2 转发，发给 h3 的包从 PORT 3 转发。所以，s1-h2 和 s1-h3 间的链路始终满载着 10Mbps 的包，并且全都是有效包。考虑到中间延迟，实际测量速率：h1-h2：9.54Mbps h1-h3：9.44Mbps，符合理论分析。

目的地址	转发端口	老化时间
Host 1 MAC Addr	Port 1	0 sec
Host 2 MAC Addr	Port 2	0 sec
Host 3 MAC Addr	Port 3	0 sec

图八 老化执行前的 Switch 的转发表

最后，传输全部结束后，等待 30 秒，可以在 s1 的终端中看到，转发表中 s1-eth0、s1-eth1 和 s1-eth2 端口对应的项被删除，证明交换机实现了 30s 老化功能。

目的地址	转发端口	老化时间

图九 老化执行后的 Switch 的转发表

2. 实验分析

本次实验实现的具体代码见第二部分实验代码详解。

实现 switch 的 main 函数与 hub 基本一致，在上次的实验报告中已详细阐述，在此不再赘述，但有所不同的是：

```
void init_ustack()
{
    instance = safe_malloc(sizeof(ustack_t));

    bzero(instance, sizeof(ustack_t));
    init_list_head(&instance->iface_list);

    init_all_ifaces();
```

```
    init_mac_port_table();
}
```

在 init_ustack 函数增加了 init_mac_port_table 函数，代码见下：

```
void init_mac_port_table()
{
    bzero(&mac_port_map, sizeof(mac_port_map_t));

    for (int i = 0; i < HASH_8BITS; i++) {
        init_list_head(&mac_port_map.hash_table[i]);
    }

    pthread_mutex_init(&mac_port_map.lock, NULL);

    pthread_create(&mac_port_map.thread, NULL, sweeping_mac_port_thr
ead, NULL);
}
```

主要功能为初始化转发表的哈希结构，初始化转发表的互斥锁，创建老化线程。

```
void *sweeping_mac_port_thread(void *nil)
{
    while (1) {
        sleep(1);
        int n = sweep_aged_mac_port_entry();

        if (n > 0)
            log(DEBUG, "%d aged entries in mac_port table are remove
d.", n);
    }

    return NULL;
}
```

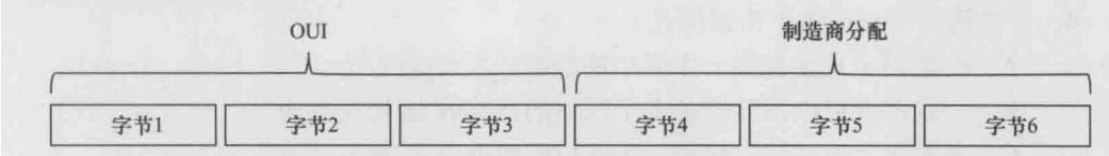
老化线程每隔 1s 执行一次，调用 sweep_aged_mac_port_entry 函数清除老化时间到达 0 的转发表项，n 的值为该次清除的转发表表项个数。

3. 思考题

1） 我们知道，网络中存在广播包，其目的 MAC 地址设置为全 0xFF ，例如 ARP 请求数据包。这种广播包对交换机行为逻辑有什么影响？

首先我们来看一个设备的 MAC 地址。制造商在生产制造网卡的过程中，会往每一块网卡的 ROM 中烧入一个 48bit 的 BIA（Burned-In Address，固化地址）地址，BIA 地址的前 3 个字节就是该制造商的 OUI，后 3 个字节由该制造商自己确定，但不同的网卡，其 BIA

地址的后 3 个字节不相同。烧入进网卡的 BIA 地址是不能被更改的，只能被读取出来使用。



图十 BIA 地址格式

实际上，BIA 地址只是 MAC 地址中的一部分，是一种单播 MAC 地址。MAC 地址共分为 3 种，分别为单播 MAC 地址、组播 MAC 地址、广播 MAC 地址。这 3 种 MAC 地址的定义分别如下：

1）单播 MAC 地址是指第一个字节的最低位是 0 的 MAC 地址。

2）组播 MAC 地址是指第一个字节的最低位是 1 的 MAC 地址。

3）广播 MAC 地址是指每个比特都是 1 的 MAC 地址。广播 MAC 地址是组播 MAC 地址的一个特例。

也就是说，不存在某个主机的 MAC 地址为全 F。对于交换机而言，一方面，交换机收到的数据包的源 MAC 地址永远不会是全 F，因为不存在这样的主机，所以全 F 的 MAC 地址永远不会被学习到交换机的转发表中；另一方面，交换机也永远不会在转发表中查询到全 F 的 MAC 地址对应的端口映射。两方面共同导致了交换机在面对目的 MAC 地址为全 F 时，采取的策略是广播，这也与 ARP 协议等设置广播报文的目的 MAC 地址为全 F 的初衷一致。

对于 MAC 地址分类的详细描述记录在我的 csdn 博客下：

https://blog.csdn.net/Therock_of_lty/article/details/105864601

2）理论上，足够多个交换机可以连接起全世界所有的终端。请问，使用这种方式连接亿万台主机是否技术可行？并说明理由。

答案肯定是不行的。

首先，目前我们实现的交换网络拓扑非常简单，树状结构，如果出现了环路，目前的交换机算法将会在数据转发过程中形成环路。所以，目前局域网交换机应用较多的算法是生成树协议。

但是，无论使用哪个算法，交换机要想实现正确的功能，其转发表需要记录网络中的所有主机。10 亿台主机所在量级为 2^30，仅仅是转发表的项数就有 1G，也就是转发表本身占据数十 GB 的空间，无论是哪个算法都无法快速处理如此庞大的数据，首先从技术上这是不可行的。

其次，任何一个交换机内部都保存网络上所有主机的 MAC 地址，这意味着网络安全性非常差，因为任何人都可以获取到任何一个主机的真实 MAC 地址并发起攻击，这对网络环境的安全是不可接受的。

（二）实验代码详解

一、handle_packet

```
void handle_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_header *eh = (struct ether_header *)packet;
    log(DEBUG, "the dst mac address is " ETHER_STRING ".\n", ETHER_F
MT(eh->ether_dhost));

    // TODO: implement the packet forwarding process here
    fprintf(stdout, "ATTENTION: handle packet.\n");

    iface_info_t* dest_iface = lookup_port(eh->ether_dhost);

    if(dest_iface != NULL)
        iface_send_packet(dest_iface, packet, len);
    else
        broadcast_packet(iface, packet, len);

    insert_mac_port(eh->ether_shost, iface);

}
```

调用 handle_packet 函数的时机在每次收到新的数据包时。首先使用 lookup_port 函数在交换机转发表中查找是否有目的 MAC 地址的表项。如果找到的话，就将该数据包从该表项中记录的端口发送出去；如果没有找到，就向所有其他端口进行广播。

最后，对该数据包的源 MAC 地址执行 insert_mac_port 函数，修改转发表。

二、lookup_port

```
iface_info_t *lookup_port(u8 mac[ETH_ALEN])
{
    // TODO: implement the lookup process here

    pthread_mutex_lock(&mac_port_map.lock);

    fprintf(stdout, "TODO: implement the lookup process here.\n");

    uint8_t hash_val = hash8((char*)mac, ETH_ALEN);
    mac_port_entry_t *mac_entry = NULL;
    int found = 0;
    list_for_each_entry(mac_entry, &mac_port_map.hash_table[hash_val
], list)
    {
```

```
            found = 1;
            for(int i = 0; i < ETH_ALEN; i++)
            {
                if(mac_entry->mac[i] != mac[i])
                    found = 0;
            }
            if(found)
            {
                fprintf(stdout,"ATTENTION: mac port found. \n");
                mac_entry->visited = time(NULL);

                pthread_mutex_unlock(&mac_port_map.lock);
                return mac_entry->iface;
            }
        }
        fprintf(stdout,"ATTENTION: mac port not found. \n");
        pthread_mutex_unlock(&mac_port_map.lock);
        return NULL;
    }
```

调用 lookup_port 函数，首先要获取转发表的互斥锁。获得锁后，计算当前 MAC 地址的哈希值，调用宏 list_for_each_entry 查找当前哈希值所在的转发链表中是否存在于当前 MAC 地址匹配的转发表项。如果有，就标准输出找到 mac 端口映射，将该表项的访问时间修改为当前的时间，释放互斥锁，返回找到的端口。如果没有，就标准输出没有找到相应映射表项，释放互斥锁，返回 NULL。

三、broadcast_packet

```
    void broadcast_packet(iface_info_t *iface, char *packet, int len)
    {
        // TODO: implement the broadcast process here


        iface_info_t *IFACE = NULL;
        list_for_each_entry(IFACE, &instance->iface_list,list)
        {
            if(IFACE->fd != iface->fd)
                iface_send_packet(IFACE, packet, len);
        }
        fprintf(stdout, "ATTENTION: broadcast completed.\n");
    }
```

当调用到 broadcast_packet 函数时，首先新建一个 iface_info_t 变量，为防止野指针赋值 NULL。随后调用 list_for_each_entry 宏，对整个链表进行遍历。遍历过程中，如果当前遍历到的端口不是发送该消息的端口，那么就调用 iface_send_packet 函数将收到的包发送给该主机；如果是发送该消息的主机，就跳过。最后打印广播完成的标识。

四、insert_mac_port

```
void insert_mac_port(u8 mac[ETH_ALEN], iface_info_t *iface)
{

    iface_info_t *IFACE = lookup_port(mac);
    //DONT USE lookup_port between lock and unlock!!!!!!
    if (IFACE) {
        (list_entry(IFACE, mac_port_entry_t, iface))->visited = time
(NULL);
        fprintf(stdout, "ATTENTION: mac port found.\n");
        return;
    }
    pthread_mutex_lock(&mac_port_map.lock);
    uint8_t hash_val = hash8((char*)mac, ETH_ALEN);
    mac_port_entry_t *mac_entry = malloc(sizeof(mac_port_entry_t));
    for (int i = 0; i < ETH_ALEN; i++)
        mac_entry->mac[i] = mac[i];
    mac_entry->iface = iface;
    mac_entry->visited = time(NULL);

    list_add_tail(&mac_entry->list, &mac_port_map.hash_table[hash_va
l]);

    fprintf(stdout, "ATTENTION: mac port %s inserted.\n", iface->nam
e);
    pthread_mutex_unlock(&mac_port_map.lock);
}
```

对一个到达的数据包解析源 MAC 地址后，启动 insert_mac_port 函数。首先调用 lookup_port 函数查看转发表中是否有该 MAC 地址，如果有，就用宏 list_entry 找到该端口的真实节点，修改其访问时间为当前时间，标准输出找到 mac 端口映射，返回。

如果没找到，申请转发表的互斥锁（之所以在前面不用加锁，是因为 lookup_port 函数本身需要申请互斥锁）。计算 MAC 地址的哈希值 hash_val，新分配一个转发表项节点，将其 mac 位修改为 MAC 地址，端口修改为收到该数据包的端口，访问时间修改为当前时间，调用宏 list_add_tail 将该节点加到 hash_val 对应的转发表链表的末尾，最后标准输出插入 mac 端口映射，释放互斥锁退出。

五、sweep_aged_mac_port_entry

老化操作有多种实现方式，本次代码选择的是老化时间栏记录最近一次访问的时间。如果当前时间与最近一次访问的时间相差超过 30s，就将该表项删除。

```
int sweep_aged_mac_port_entry()
```

```
{
    // TODO: implement the sweeping process here
    pthread_mutex_lock(&mac_port_map.lock);

    //fprintf(stdout, "TODO: implement the sweeping process here.\n"
);
    int number = 0;
    mac_port_entry_t *mac_entry, *q;
    for(int i = 0; i < HASH_8BITS; i++)
    {
        list_for_each_entry_safe(mac_entry, q, &mac_port_map.hash_ta
ble[i], list)
        {
            if(mac_entry->visited + MAC_PORT_TIMEOUT < time(NULL))
            {
                fprintf(stdout, "ATTENTION: mac port %s deleted.\n",
mac_entry->iface->name);
                list_delete_entry(&mac_entry->list);
                free(mac_entry);
                number++;
            }
        }
    }
    pthread_mutex_unlock(&mac_port_map.lock);
    return number;
}
```

首先获得转发表的互斥锁，调用宏 list_for_each_entry_safe 安全遍历每一个哈希值的转发链表，遍历时如果发现某一表项超时（算法见上），就标准输出删除该表项，调用宏 list_delete_entry 删除链表上的该节点，释放该节点的空间，计数器加 1。全部结束后，释放互斥锁，返回删除的表项数。