交换机转发实验

中国科学院大学
袁欣怡 2018K8009929021
2021.4.13

实验内容

---

1. 实现交换机转发

在广播网络中，广播结点会将收到的数据包从其他所有端口转发出去，这样会造成线路上有很多无用的数据包，降低线路传播数据的效率。为了提高效率，交换机在转发数据包的时候，会将数据包沿目的主机的方向转发，这是设计交换机的初衷。

那么问题就是，交换机如何确定哪个方向是目的主机的方向？为了解决这个问题，我们构建了一个交换机转发表。简单来说，就是在转发表中存储目的地址和转出端口的对应关系，等到需要转发的时候进行查表，即可知道该向什么方向转发数据包。

构建转发表的三个关键步骤是**查询**，**插入**和**老化**。

查询：收到数据包时，根据目的MAC地址查询转发表，如果查询到则从该端口转发，否则进行广播。

插入：收到数据包时，可以获得收到该数据包的端口和数据包的源MAC地址，更新转发表中的条目。

老化：删除30s内未访问的条目，保持转发表简洁，提高查表的效率。

还有一点需要注意的是，在查找转发表的时候，先进行Hash，也可以提高查找的效率。

2. 完成思考题

（1）交换机在转发数据包时有两个查表操作：根据源MAC地址、根据目的MAC地址，为什么在查询源MAC地址时更新老化时间，而查询目的MAC地址时不更新呢？（提示：1、查询目的MAC地址时是否有必要更新；2、如果更新的话，当一个主机从交换机的一个网口切换到了另一个网口，会有什么问题？）

（2）网络中存在广播包，即发往网内所有主机的数据包，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转发表逻辑有什么影响?

（3）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。

实验流程

1. 搭建实验环境

本实验中涉及到的文件主要有：

`main.c`：编译后生成 `switch` 。需要完成其中的 `handle_packet` 函数。

`mac.c`：实现的 `mac_port_mac` 相关操作的函数，包括 `lookup_port` ， `insert_mac_port` ，`sweep_aged_mac_port_entry` 等。

`broadcast.c`：包含了 `boradcast_packet` 函数，实现广播的功能。

`Makefile` ：处理 `make clean` 和 `make all` 指令。

`three_nodes_bw.py` ：构建三结点网络拓扑结构。

2. 实验代码设计

转发表用于存储目的地址和转发端口的映射关系，同时还需要记录每一条目上次访问的时间，超过30s未被访问的条目会被清除。

`handle_packet` 函数：

```
// handle packet
// 1. if the dest mac address is found in mac_port table, forward it; otherwise,
// broadcast it.
// 2. put the src mac -> iface mapping into mac hash table.
void handle_packet(iface_info_t *iface, char *packet, int len){
    // lab05 added: implement the packet forwarding process here
    fprintf(stdout, "NOTICE: implement the packet forwarding process here.\n");

    struct ether_header *eh = (struct ether_header *)packet;
    log(DEBUG, "the dst mac address is " ETHER_STRING ".\n", ETHER_FMT(eh->ether_dhost));

    iface_info_t* dest_iface = lookup_port(eh->ether_dhost);
    // 调用lookup查找转发表中是否有对应的条目

    if(dest_iface != NULL) // 有对应的条目
        iface_send_packet(dest_iface, packet, len); // 从记录的端口中转发出去
    else
        broadcast_packet(iface, packet, len); // 广播此数据包

```

```
    insert_mac_port(eh->ether_shost, iface); // 根据这个数据包的源和入口，修改转发表

    free(packet);

}
```

`lookup_port` 函数：

```
// lookup the mac address in mac_port table
iface_info_t *lookup_port(u8 mac[ETH_ALEN])
{
    // lab05 added: implement the lookup process here
    fprintf(stdout, "NOTICE: implement the lookup process here.\n");

    pthread_mutex_lock(&mac_port_map.lock); // 获取转发表的互斥锁

    uint8_t hash_val = hash8((char*)mac, ETH_ALEN); // 计算当前mac地址对应的hash值
    mac_port_entry_t *mac_entry = NULL;
    int i=0;
    int found = 0;

    list_for_each_entry(mac_entry, &mac_port_map.hash_table[hash_val], list) { // 查找当前hash值对应的转发链表中是否有对应的条目
        found = 1;
        for(i=0; i<ETH_ALEN; i++){
            if(mac_entry->mac[i] != mac[i])
                found = 0;
        }
        if(found){ // 查找到
            fprintf(stdout,"NOTICE: mac port found. \n");
            mac_entry->visited = time(NULL); // 修改最后访问时间

            pthread_mutex_unlock(&mac_port_map.lock);
            return mac_entry->iface;
        }
    }

    // 没有查找到
    fprintf(stdout,"ERROR: mac port not found. \n");
    pthread_mutex_unlock(&mac_port_map.lock);
    return NULL;
}
```

`insert_mac_port` 函数：

```
// insert the mac -> iface mapping into mac_port table
void insert_mac_port(u8 mac[ETH_ALEN], iface_info_t *iface)
{
```

```
    // lab05 added: implement the insertion process here
    fprintf(stdout, "NOTICE: implement the insertion process here.\n");

    iface_info_t *IFACE = lookup_port(mac); // 查找转发表中是否有该mac地址
    if (IFACE) {
        (list_entry(IFACE, mac_port_entry_t, iface))->visited = time(NULL);
// 找到该端口的真实结点，并修改访问时间
        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)); // 分配一
个新结点

    int i=0;
    for (i=0; i<ETH_ALEN; i++)
        mac_entry->mac[i] = mac[i]; // 修改为当前mac地址
    mac_entry->iface = iface; // 修改为收到数据包的端口
    mac_entry->visited = time(NULL); // 修改为当前时间

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

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

`sweep_aged_mac_port_entry` 函数：

```
// sweeping mac_port table, remove the entry which has not been visited in
the
// last 30 seconds.
int sweep_aged_mac_port_entry()
{
    // lab05 added: implement the sweeping process here
    // fprintf(stdout, "NOTICE: implement the sweeping process here.\n");

    pthread_mutex_lock(&mac_port_map.lock);

    int number=0;
    int i=0;
    mac_port_entry_t *mac_entry, *q;

    for(i=0; i<HASH_8BITS; i++){
        list_for_each_entry_safe(mac_entry, q, &mac_port_map.hash_table[i],
list){ // 遍历每一个hash值的转发链表
            if(mac_entry->visited + MAC_PORT_TIMEOUT < time(NULL)){ // 发现超
时
```

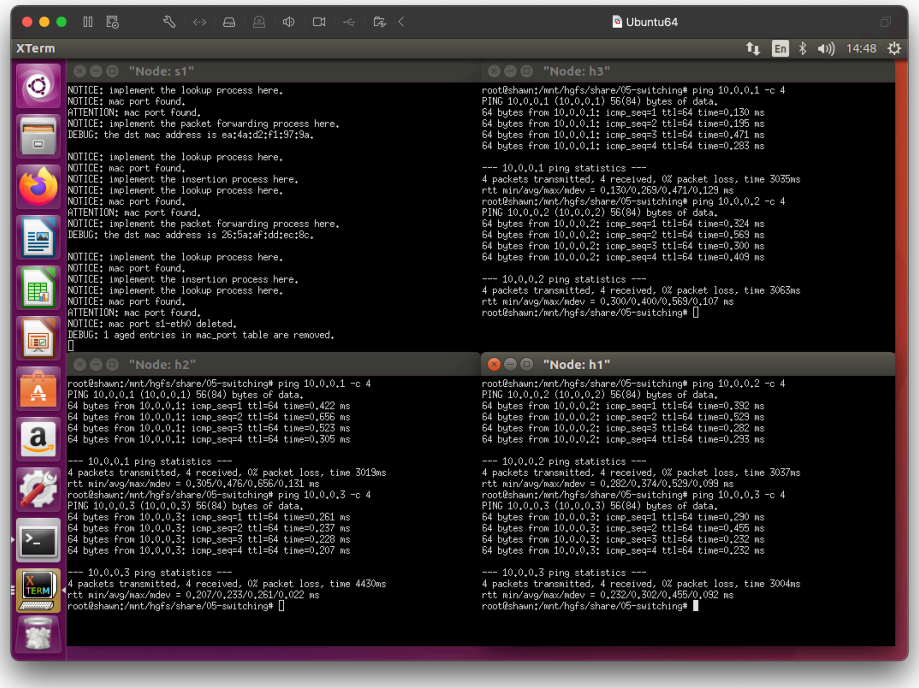
```
                fprintf(stdout, "NOTICE: mac port %s deleted.\n",mac_entry->iface->name);
                list_delete_entry(&mac_entry->list);
                free(mac_entry);
                number++;
            }
        }
    }

    pthread_mutex_unlock(&mac_port_map.lock);
    return number; // 返回删除的条目的数目
}
```

3. 启动脚本进行测试

（1）switch功能测试

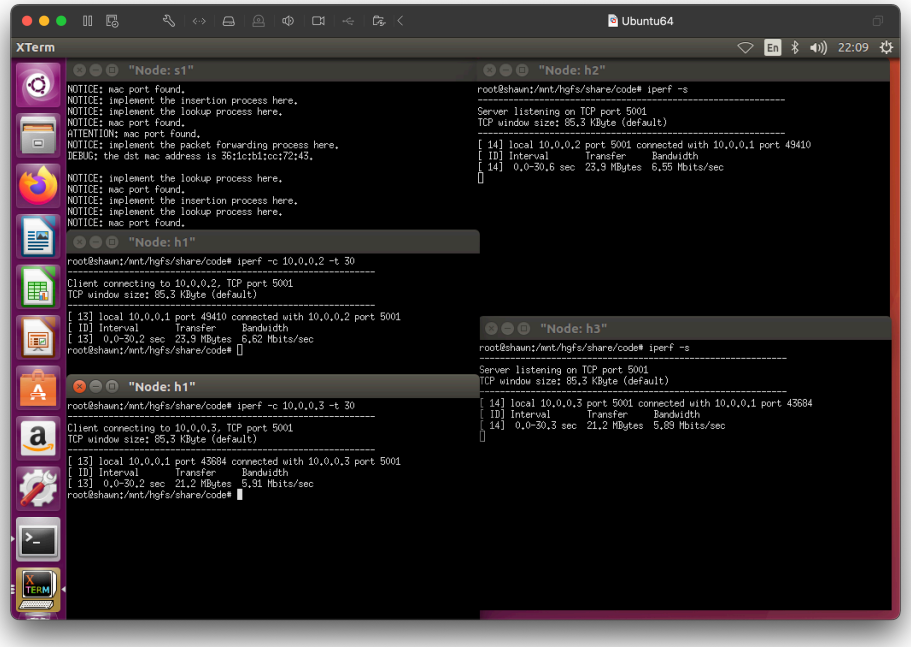
运行 `three_node_bw.py` ，开启 `h1` 、 `h2` 、 `h3` 和 `s1` 四个结点。通过从 `h1` ping `h2` 和 `h3`，从 `h2` ping `h1` 和 `h3` ，从 `h3` ping `h1` 和 `h2` ，可以发现这些数据通路都是连通的，这说明我们实现的 `switch` 可以完成转发的功能。



（2）switch性能测试

运行 `three_node_bw.py` ，开启 `h1` 、 `h2` 、 `h3` 和 `s1` 四个结点。用 `h2` 和 `h3` 作为服务器， `h1` 作为客户进行访问（需要打开两个 `h1` 的终端）。

当 `h1` 同时访问 `h2` 和 `h3` 时，得到的传输速率如图所示：



测试结果：h1-h2：6.62 Mbps，h1-h3：5.91 Mbps。

回顾上次实验，同样条件下的测试结果：h1-h2：6.33 Mbps，h1-h3：3.51 Mbps。因此我们不难发现，传输速率有提升，但没有达到期望幅度。

在进行交换机转发的过程中，以 `h1` 向 `h2` 发包为例， `h1` 先在自己的ARP缓存中查找 `h2` 的 `MAC` 地址，没有查找到对应条目，因此向 `s1` 发送一个 `ARP` 广播包，向局域网范围内的所有主机询问 `h2` 的 `MAC` 地址。当 `s1` 收到广播包时，先获得 `h1` 的 `MAC` 地址，并且在转发表中新增这一条目，然后将广播包发送给周围所有结点，最终到达 `h2` 。 `h2` 收到数据包后，会通过 `s1` 向 `h1` 发送一个 `ARP` 单播包，此时 `s1` 收到包后就可以将 `h2` 的 `MAC` 地址添加到转发表中，且此时转发表中已经有关于 `h1` 的信息， `s1` 可以确定从哪个端口转发数据包。 `h1` 收到数据包后，解析后获得 `h2` 的地址并写入 `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。

因此理论上h1-h2和h1-h3的传播速率都应在 `10Mbps` 左右，实际测试中速率只达到了 `6Mbps` 左右，可能的原因有：（1）网络结构简单，性能提升不明显，而且增加了发送 `ARP` 单播包的消耗，所以实际的速率没有达到预期。（2）给虚拟机分配的 `CPU` 资源较少，导致难以达到可用的带宽瓶颈。

实验总结与思考题

1. 交换机在转发数据包时有两个查表操作：根据源MAC地址、根据目的MAC地址，为什么在查询源MAC地址时更新老化时间，而查询目的MAC地址时不更新呢？（提示：1、查询目的MAC地址时是否有必要更新；2、如果更新的话，当一个主机从交换机的一个网口切换到了另一个网口，会有什么问题？）

考虑一种极端情况：假如结点 `h1` 一直不停地向结点 `h2` 发送数据包，这个过程中结点 `h2` 更换了接入网口，那么交换机就无法通过转发表中记录的网口将数据包发送给 `h2` ，但是由于 `h1` 一直在查询 `h2` 的 `MAC` 地址，所以转发表中的条目一直不会被老化舍弃，最终这个数据包一致无法送达。所以不能在查询目的 `MAC` 地址时更新老化时间。

2. 网络中存在广播包，即发往网内所有主机的数据包，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转发表逻辑有什么影响?

   这种广播包不会影响交换表转发逻辑。因为按照 `MAC` 地址的设计规则，不存在某个主机的 `MAC` 地址为全 `0xFF` ，所以这个地址不会被添加到转发表中。但是广播包可能会引起数据包环路的现象。

3. 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。

   不行。首先，所有终端构成的网络中势必存在很多环路，需要用生成树算法去处理这些环路。其次，转发表中需要记录大量的条目，总量在 `10GB` 量级，查找时非常慢，效率反而会降低。最后，将所有终端连接在一起会降低网络入侵的难度，对网络安全造成极大威胁。

参考资料

1. ARP协议的工作机制详解：http://c.biancheng.net/view/6388.html
2. MAC地址分类：https://blog.csdn.net/Therock_of_lty/article/details/105864601