

# 交换机转发 实验设计报告

中国科学院大学

页

2022 年 4 月 9 日

## 一、广播网络实验

### 1. 实验内容

- (1) 实现节点广播的 `broadcast_packet` 函数。
- (2) 验证广播网络能够正常运行：从一个端节点 ping 另一个端节点
- (3) 验证广播网络的效率：在 `three_nodes_bw.py` 进行 `iperf` 测量两种场景：  
H1: `iperf client`; H2, H3: `servers` (h1 同时向 h2 和 h3 测量)  
H1: `iperf server`; H2, H3: `clients` (h2 和 h3 同时向 h1 测量)
- (4) 自己动手构建环形拓扑，验证该拓扑下节点广播会产生数据包环路。

### 2. 实验流程

- (1) 实现节点广播的 `broadcast_packet` 函数，调用 `list_for_each_entry`，对整个链表进行遍历当前主机不是发送消息的主机则调用 `iface_send_packet`，发包给这个主机。

```
void broadcast_packet(iface_info_t *iface, const char *packet, int len)
{
    // TODO: broadcast packet
    fprintf(stdout, "TODO: broadcast packet.\n");

    iface_info_t *iface_n = NULL;
    list_for_each_entry(iface_n, &instance->iface_list, list) {
        if (iface_n->fd != iface->fd)
            iface_send_packet(iface_n, packet, len);
    }
}
```

- (2) 编译 hub 程序运行程序 `three_node_bw.py`, 开启 h1、h2、h3 和 b1 四个结点。通过从 h1 ping h2 和 h3, 从 h2 ping h1 和 h3, 从 h3 ping h1 和 h2, 验证这些数据通路是否都是都是连通的。
- (3) 进行广播网络效率测试, 运行 `three_node_bw.py`, 开启 h1、h2、h3 和 b1 四个结点。先用 h1 作为服务器, h2 和 h3 作为客户进行访问。再用 h2 和 h3 作为服务器, h1 作为客户进行访问。

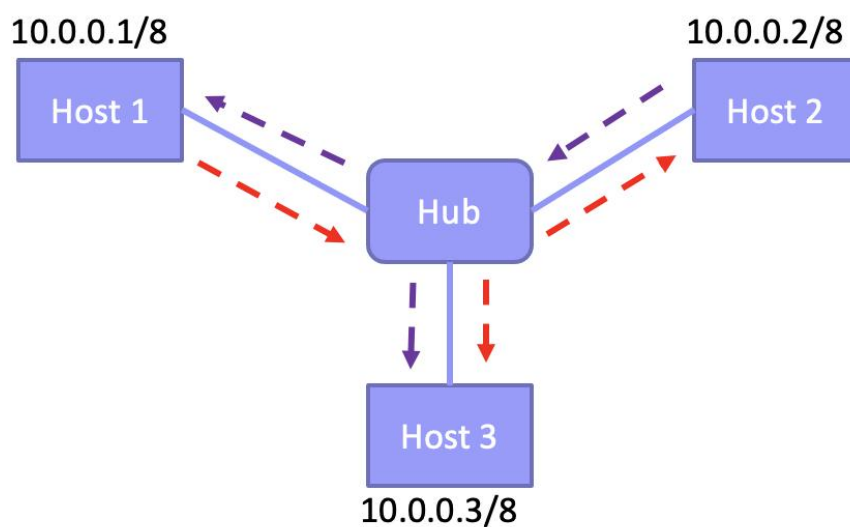


图 1: 3 结点拓扑

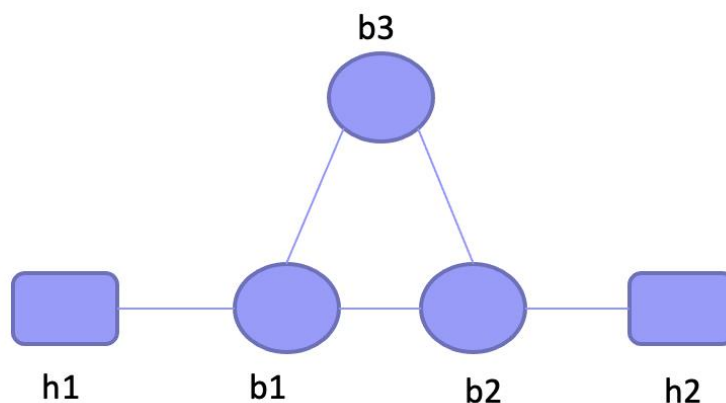


图 2: 环形拓扑

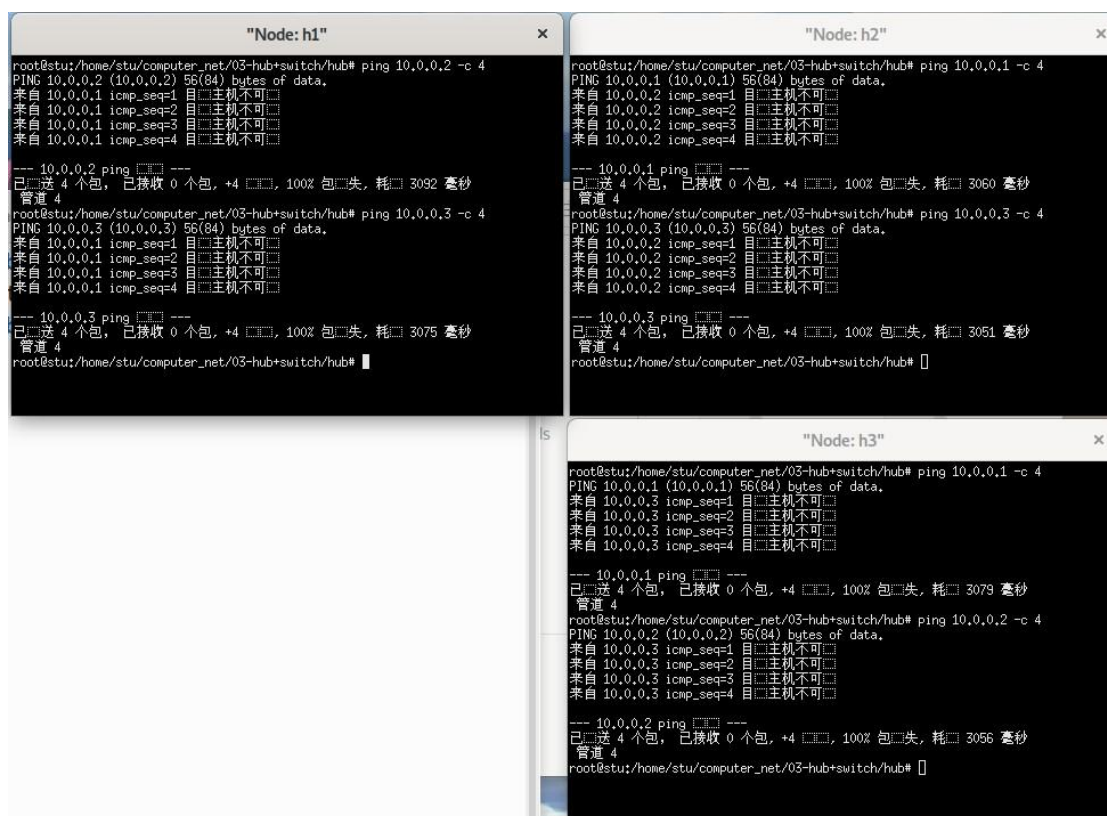
- (4) 修改拓扑结构, 验证环形拓扑下数据报形成环路。参照图 1 和图 2 的拓扑结构以及

源程序 `three_node_bw.py` 编写环形拓扑 `topo.py`，运行 `topo.py`，开启 `h1`、`h2`、`b1`、`b2` 和 `b3` 五个结点。`b1`、`b2` 和 `b3` 作为 hub，然后用 `h1` 发送一个数据包。然后在 `h2` 中打开 `wireshark` 抓包。

```
class BroadcastTopo(Topo):  
    def build(self):  
        h1 = self.addHost('h1')  
        h2 = self.addHost('h2')  
  
        b1 = self.addHost('b1')  
        b2 = self.addHost('b2')  
        b3 = self.addHost('b3')  
  
        self.addLink(h1, b1)  
        self.addLink(h2, b2)  
        self.addLink(b1, b2)  
        self.addLink(b2, b3)  
        self.addLink(b1, b3)
```

### 3. 实验结果与分析

#### (1) 广播网络功能测试 (ping)



```
"Node: h1"
root@stu:/home/stu/computer_net/03-hub+switch/hub# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
 来自 10.0.0.1 icmp_seq=1 自 主机不可
 来自 10.0.0.1 icmp_seq=2 自 主机不可
 来自 10.0.0.1 icmp_seq=3 自 主机不可
 来自 10.0.0.1 icmp_seq=4 自 主机不可

--- 10.0.0.2 ping ---
已 送 4 个包, 已接收 0 个包, +4 , 100% 包 , 耗 3092 毫秒
管道 4
root@stu:/home/stu/computer_net/03-hub+switch/hub# ping 10.0.0.3 -c 4
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
 来自 10.0.0.1 icmp_seq=1 自 主机不可
 来自 10.0.0.1 icmp_seq=2 自 主机不可
 来自 10.0.0.1 icmp_seq=3 自 主机不可
 来自 10.0.0.1 icmp_seq=4 自 主机不可

--- 10.0.0.3 ping ---
已 送 4 个包, 已接收 0 个包, +4 , 100% 包 , 耗 3075 毫秒
管道 4
root@stu:/home/stu/computer_net/03-hub+switch/hub#

"Node: h2"
root@stu:/home/stu/computer_net/03-hub+switch/hub# ping 10.0.0.1 -c 4
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
 来自 10.0.0.2 icmp_seq=1 自 主机不可
 来自 10.0.0.2 icmp_seq=2 自 主机不可
 来自 10.0.0.2 icmp_seq=3 自 主机不可
 来自 10.0.0.2 icmp_seq=4 自 主机不可

--- 10.0.0.1 ping ---
已 送 4 个包, 已接收 0 个包, +4 , 100% 包 , 耗 3060 毫秒
管道 4
root@stu:/home/stu/computer_net/03-hub+switch/hub# ping 10.0.0.3 -c 4
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
 来自 10.0.0.2 icmp_seq=1 自 主机不可
 来自 10.0.0.2 icmp_seq=2 自 主机不可
 来自 10.0.0.2 icmp_seq=3 自 主机不可
 来自 10.0.0.2 icmp_seq=4 自 主机不可

--- 10.0.0.3 ping ---
已 送 4 个包, 已接收 0 个包, +4 , 100% 包 , 耗 3051 毫秒
管道 4
root@stu:/home/stu/computer_net/03-hub+switch/hub#

"Node: h3"
root@stu:/home/stu/computer_net/03-hub+switch/hub# ping 10.0.0.1 -c 4
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
 来自 10.0.0.3 icmp_seq=1 自 主机不可
 来自 10.0.0.3 icmp_seq=2 自 主机不可
 来自 10.0.0.3 icmp_seq=3 自 主机不可
 来自 10.0.0.3 icmp_seq=4 自 主机不可

--- 10.0.0.1 ping ---
已 送 4 个包, 已接收 0 个包, +4 , 100% 包 , 耗 3079 毫秒
管道 4
root@stu:/home/stu/computer_net/03-hub+switch/hub# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
 来自 10.0.0.3 icmp_seq=1 自 主机不可
 来自 10.0.0.3 icmp_seq=2 自 主机不可
 来自 10.0.0.3 icmp_seq=3 自 主机不可
 来自 10.0.0.3 icmp_seq=4 自 主机不可

--- 10.0.0.2 ping ---
已 送 4 个包, 已接收 0 个包, +4 , 100% 包 , 耗 3056 毫秒
管道 4
root@stu:/home/stu/computer_net/03-hub+switch/hub#
```

图 3: 广播网络 ping 测试

测试结果如图 3。结果显示, 各 host 节点之间连接正常, hub 可以完成正常广播的功能。

## (2) 广播网络效率测试

① h1 作为主机, h2 和 h3 同时向 h1 测量, 结果如图 4 所示。

```

"Node: h1"
root@stu:/home/stu/computer_net/03-hub+switch/hub# iperf -s
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
[ 14] local 10.0.0.1 port 5001 connected with 10.0.0.3 port 53224
[ ID] Interval      Transfer    Bandwidth
[ 14] 0.0-30.9 sec  34.0 MBytes  9.22 Mbits/sec
[ 14] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 42396
[ 14] 0.0-30.2 sec  33.1 MBytes  9.19 Mbits/sec

"Node: h3"
root@stu:/home/stu/computer_net/03-hub+switch/hub# iperf -c 10.0.0.1 -t 30
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 144 KByte (default)
[ 13] local 10.0.0.3 port 53224 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 13] 0.0-30.2 sec  34.0 MBytes  9.43 Mbits/sec

"Node: b1"
root@stu:/home/stu/computer_net/03-hub+switch/hub# iperf -c 10.0.0.1 -t 30
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 144 KByte (default)
[ 13] local 10.0.0.2 port 42396 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 13] 0.0-30.0 sec  33.1 MBytes  9.26 Mbits/sec

"Node: h2"
root@stu:/home/stu/computer_net/03-hub+switch/hub# iperf -c 10.0.0.1 -t 30
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 144 KByte (default)
[ 13] local 10.0.0.2 port 42396 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 13] 0.0-30.0 sec  33.1 MBytes  9.26 Mbits/sec

```

图 4: h1 作主机 iperf 测试

h1 处的吞吐量为  $9.22+9.19 = 18.41\text{Mbps}$ ，带宽利用率为  $90.21\%$ 。h2 处的吞吐量为  $9.43\text{Mbps}$ ，利用率为  $94.3\%$ ，h3 处的吞吐量为  $9.26\text{Mbps}$ ，利用率  $92.6\%$ ，双向带宽利用都较高。

②h2 和 h3 作主机，h1 同时向 h2 和 h3 测量，结果如图 5 所示。

h1 处的吞吐量为  $5.31+4.00 = 9.31\text{Mbps}$ ，（单向）带宽利用率为  $46.55\%$ 。h2 处的吞吐量为  $5.23\text{Mbps}$ ，利用率为  $52.3\%$ ，h3 处的吞吐量为  $3.95\text{Mbps}$ ，利用率  $39.5\%$ 。带宽利用率很低，从图中我们可以看到 h2 和 h3 除了收到自己的包以外还收到了对方的包，这说明当 h1 同时向 h2 和 h3 发包时，在 b1 处包会被复制，在 b1→h2 有 h1 发往 h3 的包，在 b1→h3 有 b1 发往 h2 的包，这些冗余包浪费了带宽，因而带宽利用率很低。在情况①的时候 h2→b1 和 h3→b2 是独立的，因此带宽受影响很小，但是仍然达不到  $100\%$ 这是由于数据链路层会把数据包额外加上一些信息，这些信息占用的带宽不计入带宽计算中。

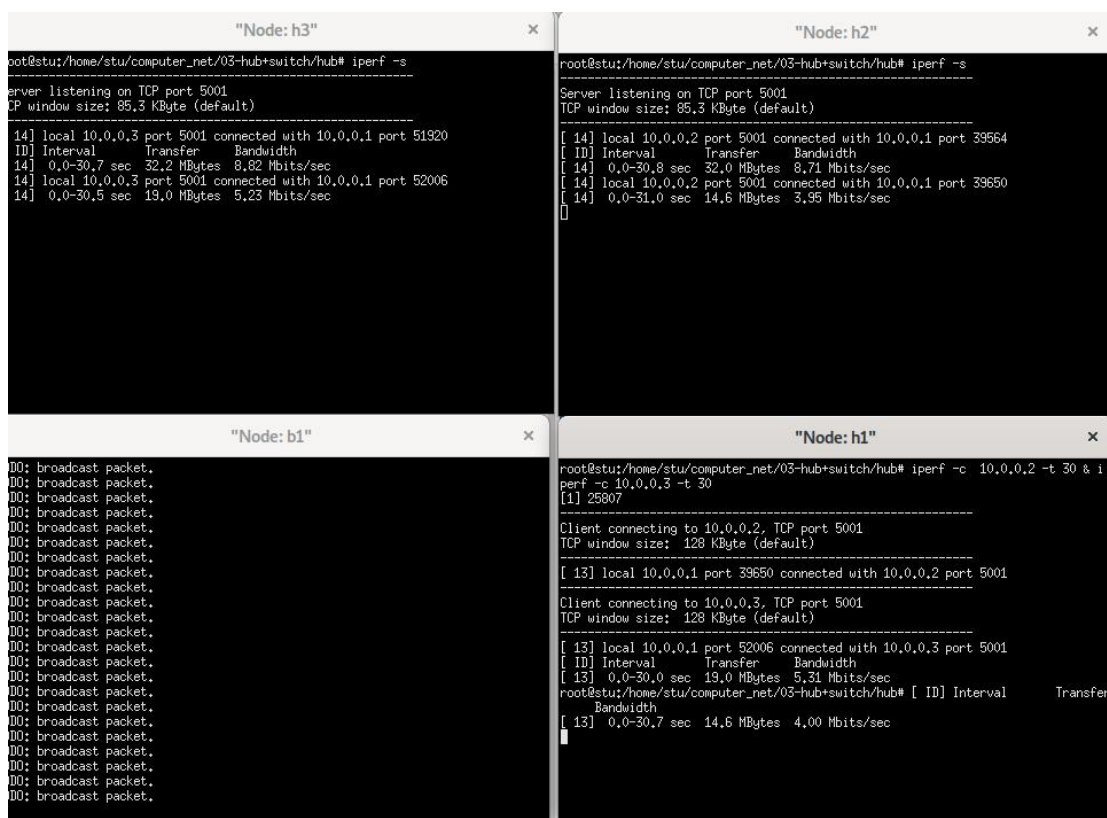


图 5: h2 和 h3 作主机 iperf

### (3) 环路广播测试

如图 6 所示, h1 向 h2 发出 ping 命令后, 各个转发节点的 hub 程序输出如图 10 所示, 说明环路中包被不断复制转发。当 h1 发送数据包后, b1、b2 和 b3 都在不停打印广播的信息, 同时从主机 h2 的 wireshark 如图 7 中抓包也可得到一直被转发的同样的数据包, 这说明数据包在环形拓扑中被不停地转发, 形成了广播风暴。

广播风暴(broadcast storm)简单的讲是指当广播数据充斥网络无法处理, 并占用大量网络带宽, 导致正常业务不能运行, 甚至彻底瘫痪, 这就发生了“广播风暴”。一个数据帧或包被传输到本地网段 (由广播域定义) 上的每个节点就是广播; 由于网络拓扑的设计和连接问题, 或其他原因导致广播在网段内大量复制, 传播数据帧, 导致网络性能下降, 甚至网络瘫痪。

广播风暴的产生有多种原因，如蠕虫病毒、交换机端口故障、网卡故障、链路冗余没有启用生成树协议、网线线序错误或受到干扰等。从目前来看，蠕虫病毒和 arp 攻击是造成网络广播风暴最主要的原因。虽说如今网络广播风暴已经很少见了，但在一些使用集线器的网络中仍然非常常见。解决网络广播风暴最快捷的方法是给集线器断电然后上电启动即可，但这只是治标不治本的方法，要彻底解决，最好使用交换机设备，并划分 vlan、通过端口控制网络广播风暴。否则，如果广播风暴是由于网卡损坏所致，要上百台计算机找出故障计算机是十分困难的事情。



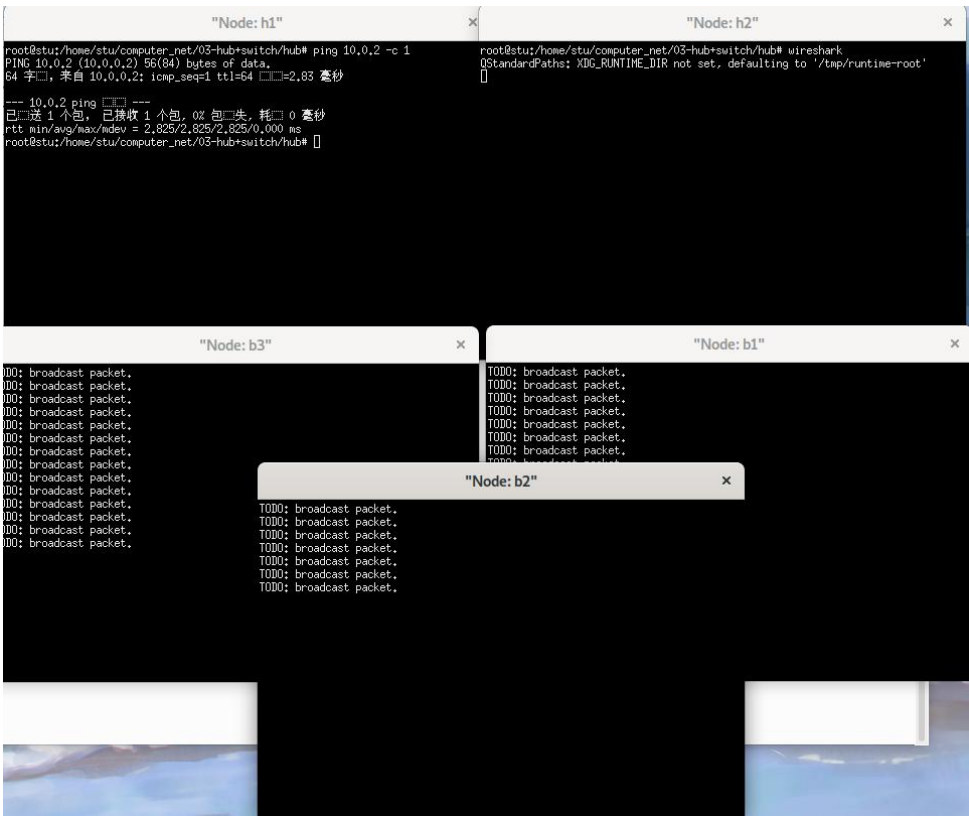


图 6：环路拓扑测试结果

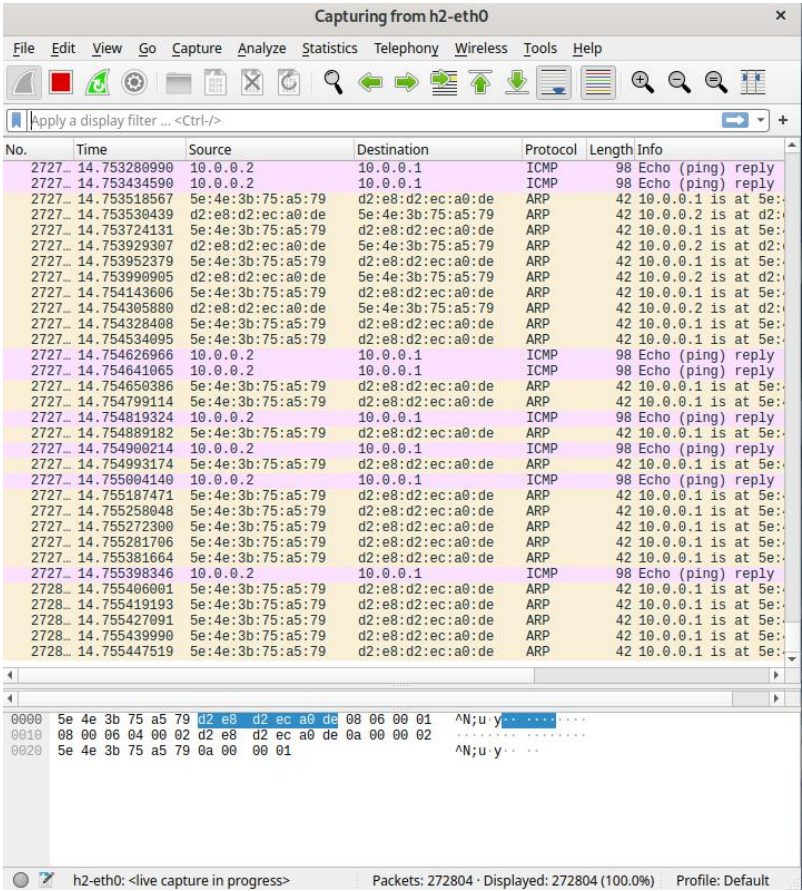


图 7：wireshark 抓包环路拓扑

## 二、交换机转发实验

### 1. 实验内容

(1) 实现对数据结构 `mac_port_map` 的所有操作，包括查询，插入和老化，以及数据包的转发和广播操作。

```
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);
```

(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。

### 2. 实验流程

- (1) 理解现有的代码框架，补充完善 `boradcast_packet.c` 中的 `boradcast_packet` 函数，`main.c` 中的 `handle_packet` 函数，`mac.c` 中的 `lookup_port`、`insert_mac_port` 和 `sweep_aged_mac_port_entry` 函数。
- (2) `make` 编译生成 `switch` 文件。
- (3) 执行命令生成 `h1`、`h2`、`h3` 三给主机和一个 `s1` 交换机结点，在 `h2`、`h3` 服务端，从 `h1` 发送数据包测试性能。

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

- (4) 观察实验结果，完成实验报告。

### 3. 实验函数设计

#### ① 数据包处理

交换机收到数据包后调用 `lookup` 查找转发表中是否有对应的条目，如果有对应的条目从记录的端口中转发出去，否则广播此数据包。如果没有对应的的条目根据这个数据包的源和入口，修改转发表

```
void handle_packet(iface_info_t *iface, char *packet, int len)
```



```

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

    struct ether_header *eh = (struct ether_header *)packet;
    iface_info_t * dest_iface = lookup_port(eh->ether_dhost);
    if (dest_iface) {
        iface_send_packet(dest_iface, packet, len);
    } else {
        broadcast_packet(iface, packet, len);
    }

    insert_mac_port(eh->ether_shost, iface);

    free(packet);
}

```

- ② 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。函数首先获得转发表的互斥锁，计算当前 mac 地址对应的 hash 值，然后利用 `list_for_each_entry()` 和 `memcmp()` 函数查找当前 hash 值对应的转发链表中是否有对应的条目，找到后占用的互斥锁。

```

iface_info_t *lookup_port(u8 mac[ETH_ALEN])
{
    // TODO: implement the lookup process here
    // fprintf(stdout, "TODO: implement the lookup process here.\n");
    mac_port_entry_t * mac_entry = NULL;
    u8 mac_hash = hash8((void *) mac, ETH_ALEN);
    pthread_mutex_lock(&mac_port_map.lock);
    list_for_each_entry(mac_entry, &mac_port_map.hash_table[mac_hash],
list) {
        if (memcmp(mac_entry->mac, mac, ETH_ALEN) == 0) {
            pthread_mutex_unlock(&mac_port_map.lock);
            return mac_entry->iface;
        }
    }
}

```

```

    }
}
pthread_mutex_unlock(&mac_port_map.lock);
return NULL;
}

```

- ③ 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。首先便利查找该 mac 对应的 hash 表中是否有对应条目，如果有的话修改信息和访问时间。没有对应的条目的话创建新的结点写入信息并将对应的结点插入到表中。

```

void insert_mac_port(u8 mac[ETH_ALEN], iface_info_t *iface)
{
    // TODO: implement the insertion process here
    //fprintf(stdout, "TODO: implement the insertion process here.\n");
    mac_port_entry_t * mac_entry = NULL;
    u8 mac_hash = hash8((void *) mac, ETH_ALEN);
    time_t now = time(NULL);
    pthread_mutex_lock(&mac_port_map.lock);
    list_for_each_entry(mac_entry, &mac_port_map.hash_table[mac_hash],
list) {
        if (memcmp(mac_entry->mac, mac, ETH_ALEN) == 0) {
            mac_entry->iface = iface;
            mac_entry->visited = now;
            pthread_mutex_unlock(&mac_port_map.lock);
            return;
        }
    }

    mac_entry = malloc(sizeof(mac_port_entry_t));
    memcpy(mac_entry->mac, mac, ETH_ALEN);
    mac_entry->iface = iface;
    mac_entry->visited = now;
    list_add_head(&mac_entry->list, &mac_port_map.hash_table[mac_hash]);
}

```

```
pthread_mutex_unlock(&mac_port_map.lock);
}
```

- ④ 老化：删除 30s 内未访问的条目，提高查表的效率。遍历 hash 转发表，如果发现超时的结点就删除对应的条目，释放相应的空间。

```
int sweep_aged_mac_port_entry()
{
    // TODO: implement the sweeping process here
    //fprintf(stdout, "TODO: implement the sweeping process here.\n");

    mac_port_entry_t *mac_entry = NULL;
    mac_port_entry_t *next_entry = NULL;
    time_t now = time(NULL);

    pthread_mutex_lock(&mac_port_map.lock);
    for (int i = 0; i < HASH_8BITS; i++) {
        list_for_each_entry_safe(mac_entry, next_entry,
                                &mac_port_map.hash_table[i], list) {
            if ((int)(now - mac_entry->visited) > MAC_PORT_TIMEOUT) {
                list_delete_entry(&mac_entry->list);
                free(mac_entry);
            }
        }
    }
    pthread_mutex_unlock(&mac_port_map.lock);
    return 0;
}
```

## 4. 实验结果与分析

### (1) 广播功能测试 (ping)

如图 8 所示，运行 three\_node\_bw.py，开启 h1、h2、h3 和 s1 四个结点。在 s1 打开 switch, 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 ping 通的，这说明实现的 switch 可以正常完成转发的功能。

```

"Node: h3"
root@stu:/home/stu/computer_net/03-hub+switch/switch# ping 10.0.0.1 -c 4
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data:
64 字节, 来自 10.0.0.1: icmp_seq=1 ttl=64 时间=0.225 毫秒
64 字节, 来自 10.0.0.1: icmp_seq=2 ttl=64 时间=0.294 毫秒
64 字节, 来自 10.0.0.1: icmp_seq=3 ttl=64 时间=0.151 毫秒
64 字节, 来自 10.0.0.1: icmp_seq=4 ttl=64 时间=0.147 毫秒

--- 10.0.0.1 ping 统计 ---
已发送 4 个包, 已接收 4 个包, 0% 包丢失, 耗时 3061 毫秒
rtt min/avg/max/ndev = 0.147/0.204/0.294/0.060 ms
root@stu:/home/stu/computer_net/03-hub+switch/switch# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 字节, 来自 10.0.0.2: icmp_seq=1 ttl=64 时间=0.144 毫秒
64 字节, 来自 10.0.0.2: icmp_seq=2 ttl=64 时间=0.158 毫秒
64 字节, 来自 10.0.0.2: icmp_seq=3 ttl=64 时间=0.240 毫秒
64 字节, 来自 10.0.0.2: icmp_seq=4 ttl=64 时间=0.213 毫秒

--- 10.0.0.2 ping 统计 ---
已发送 4 个包, 已接收 4 个包, 0% 包丢失, 耗时 3074 毫秒
rtt min/avg/max/ndev = 0.144/0.188/0.240/0.039 ms
root@stu:/home/stu/computer_net/03-hub+switch/switch#

"Node: h1"
root@stu:/home/stu/computer_net/03-hub+switch/switch# ping 10.0.0.2 -c 4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 字节, 来自 10.0.0.2: icmp_seq=1 ttl=64 时间=0.241 毫秒
64 字节, 来自 10.0.0.2: icmp_seq=2 ttl=64 时间=0.155 毫秒
64 字节, 来自 10.0.0.2: icmp_seq=3 ttl=64 时间=0.159 毫秒
64 字节, 来自 10.0.0.2: icmp_seq=4 ttl=64 时间=0.145 毫秒

--- 10.0.0.2 ping 统计 ---
已发送 4 个包, 已接收 4 个包, 0% 包丢失, 耗时 3063 毫秒
rtt min/avg/max/ndev = 0.145/0.175/0.241/0.038 ms
root@stu:/home/stu/computer_net/03-hub+switch/switch# ping 10.0.0.3 -c 4
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data:
64 字节, 来自 10.0.0.3: icmp_seq=1 ttl=64 时间=0.152 毫秒
64 字节, 来自 10.0.0.3: icmp_seq=2 ttl=64 时间=0.079 毫秒
64 字节, 来自 10.0.0.3: icmp_seq=3 ttl=64 时间=0.143 毫秒
64 字节, 来自 10.0.0.3: icmp_seq=4 ttl=64 时间=0.079 毫秒

--- 10.0.0.3 ping 统计 ---
已发送 4 个包, 已接收 4 个包, 0% 包丢失, 耗时 3059 毫秒
rtt min/avg/max/ndev = 0.079/0.113/0.152/0.034 ms
root@stu:/home/stu/computer_net/03-hub+switch/switch#

"Node: s1"
root@stu:/home/stu/computer_net/03-hub+switch/switch# ./switch
DEBUG: find the following interfaces: s1-eth0 s1-eth1 s1-eth2.
INFO: last visit = 1649552798
INFO: Entry: e2:e2:06:36:46:f3 -> s1-eth0 are removed.
INFO: last visit = 1649552810
INFO: Entry: 32:63:10:09:21:6c -> s1-eth1 are removed.
INFO: last visit = 1649552810
INFO: Entry: 5e:ad:93:91:19:20 -> s1-eth2 are removed.

"Node: h2"
root@stu:/home/stu/computer_net/03-hub+switch/switch# ping 10.0.0.1 -c 4
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data:
64 字节, 来自 10.0.0.1: icmp_seq=1 ttl=64 时间=0.167 毫秒
64 字节, 来自 10.0.0.1: icmp_seq=2 ttl=64 时间=0.391 毫秒
64 字节, 来自 10.0.0.1: icmp_seq=3 ttl=64 时间=0.165 毫秒
64 字节, 来自 10.0.0.1: icmp_seq=4 ttl=64 时间=0.175 毫秒

--- 10.0.0.1 ping 统计 ---
已发送 4 个包, 已接收 4 个包, 0% 包丢失, 耗时 3139 毫秒
rtt min/avg/max/ndev = 0.165/0.224/0.391/0.096 ms
root@stu:/home/stu/computer_net/03-hub+switch/switch# ping 10.0.0.3 -c 4
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data:
64 字节, 来自 10.0.0.3: icmp_seq=1 ttl=64 时间=0.089 毫秒
64 字节, 来自 10.0.0.3: icmp_seq=2 ttl=64 时间=0.103 毫秒
64 字节, 来自 10.0.0.3: icmp_seq=3 ttl=64 时间=0.104 毫秒
64 字节, 来自 10.0.0.3: icmp_seq=4 ttl=64 时间=0.086 毫秒

--- 10.0.0.3 ping 统计 ---
已发送 4 个包, 已接收 4 个包, 0% 包丢失, 耗时 3056 毫秒
rtt min/avg/max/ndev = 0.086/0.095/0.104/0.008 ms
root@stu:/home/stu/computer_net/03-hub+switch/switch#

```

图 8：交换机 ping 测试结果

## (2) 广播性能效率测试 (iperf)

运行 three\_node\_bw.py，开启 h1、h2、h3 和 s1 四个结点，用 h3 和 h3 作为服务器，h1 作为客户机同时访问 h2 和 h3。

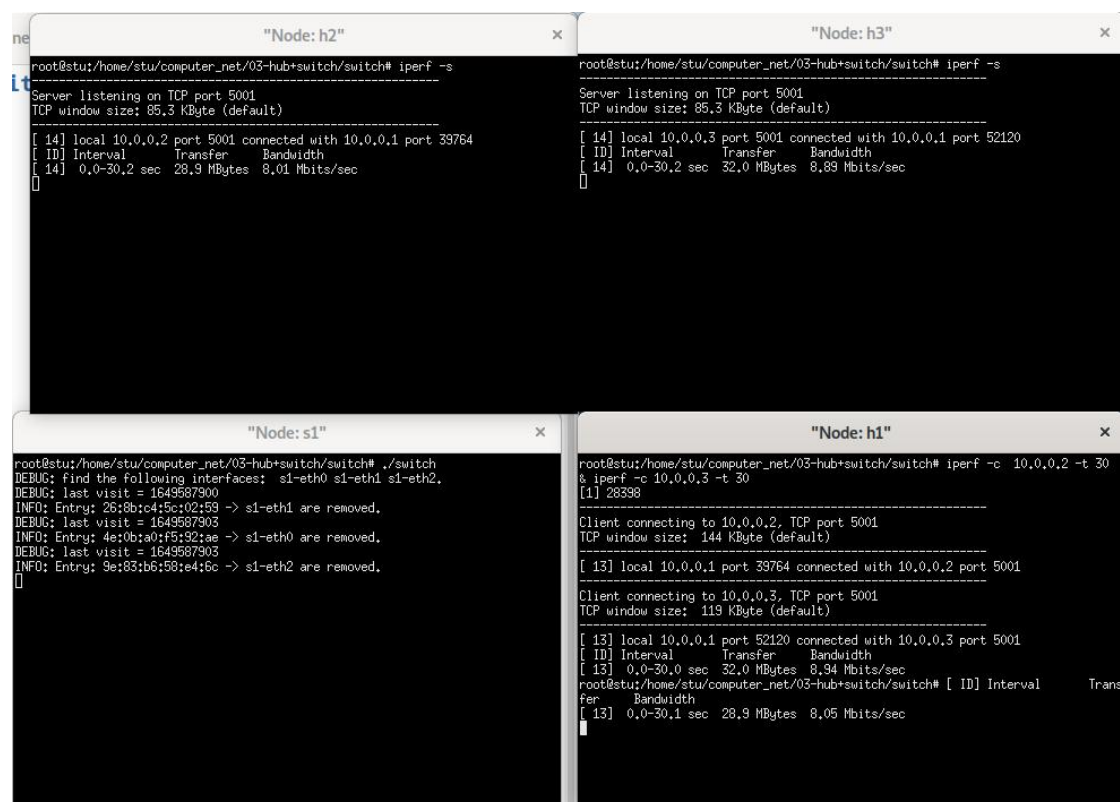


图 9：交换机广播性能测试结果

如图 9 测试结果显示：h1→h2 的实测带宽为 8.94Mbps，h1→h3 的实测带宽为 8.06Mbps。带宽利用率平均在 85.0%，h2 的处的吞吐量为 8.01Mbps，h3 处的为 8.89，而在 hub 同样实验实验结果为 h1 处的吞吐量为 5.31+4.00 = 9.31Mbps，带宽利用率为 46.55%。h2 处的吞吐量为 5.23Mbps，利用率为 52.3%，h3 处的吞吐量为 3.95Mbps，利用率 39.5%。带宽上涨 100%，这是因为相比 hub 的复制发送 switch 只向目的端口转发数据包而不再向所有端口转发数据包，避免了网络拓扑中的冗余包占用带宽。

与此同时我们可以从图 9 的 s1 节点发现 switch 的查询，插入和老化等功能正常运行。

### 三、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？）

答：

目的地址不需要更新，老化操作可以把过期的目的地址删除。如果更新目的地址，当目的 MAC 地址变更时，原来地址的老化时间可能一直被更新，从而无法被老化删除。考虑一

种情况：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 MAC 地址，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就一直不会被老化删除，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？

答：

这种广播包不会影响交换表转发逻辑。因全 0xFF 地址只会作为目的地址出现而不会作为源地址出现，将会被广播，不会更新交换机转发表。

(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

答：

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，两个节点之间只能有一条路径。当节点数量足够多时，网络中的一台交换机需要处理大量结点的请求，转发表需要为所有地址进行学习，交换机的计算和存储能力有限，达不到要求。其次，为了转发表中的信息不被很快刷新，转发表中需要记录大量的条目，查找时非常慢，效率反而会降低。

从交换机的连接方式上来说，单独一台交换机的端口数量是有限的，不足以满足网络终端设备接入网络的需求。为此我们需要使用多台交换机来提供终端接入功能，并将多台交换机互连，形成一个局域网。交换机的互连主要有级联(Uplink)和堆叠(Stack)两种方式。当交换机级联层数较多时，层次之间存在较大的收敛比，将出现较大的延时。超过一定数量的交换机进行级联，最终会因此网络广播风暴，导致网络性能严重下降，甚至瘫痪。因此交换机不应该无限制级联。而堆叠方式的数目也是有限的，一般最多允许对 8 台交换机进行堆叠，并且要求堆叠成员离自己的位置足够近，堆叠线缆最长只有几米，一般堆叠的交换机处于同一个机柜中，这种方式难以远程连接。

同时将所有终端连接在一起会降低网络入侵的难度，网络安全隐患很大

#### 参考资料：

【1】 广播风暴：<https://www.2lic.com/article/702749.html>。

【2】 交换机的互联技术：<https://www.cnblogs.com/gzxbkk/p/7910611.html>。