广播网络实验报告

李昊宸

2017K8009929044

（一）广播网络实现

一、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

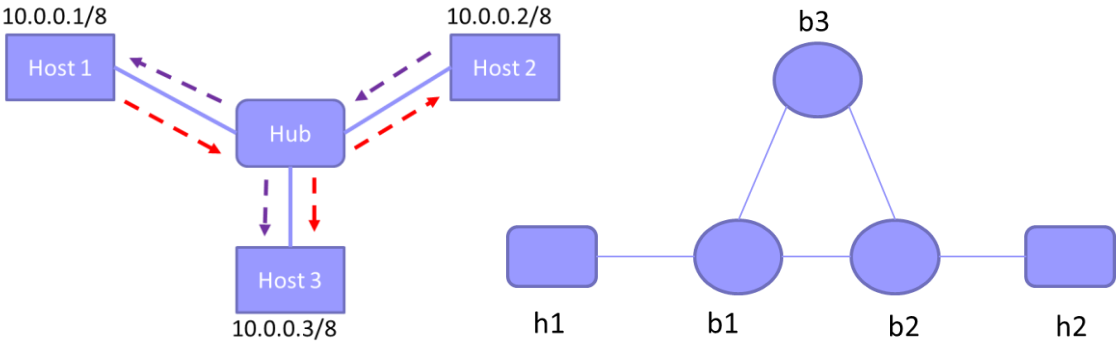
   从一个端节点 ping 另一个端节点
3. 验证广播网络的效率

   在 three_nodes_bw.py 进行 iperf 测量
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

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： Hub 的代码实现，编译后在 Hub 结点上运行
three_nodes_bw.py：实现如下左图的节点拓扑
loop_topo.py：实现如下右图的节点拓扑



图一 三节点网络拓扑（左）与环形网络拓扑（右）

2. 启动脚本

1)广播网络功能测试

```
make all
sudo python three_nodes_bw.py
mininet> xterm h1 h2 h3 b1
b1# ./hub
h1# ping 10.0.0.2 -c 4
h1# ping 10.0.0.3 -c 4
h2# ping 10.0.0.1 -c 4
h2# ping 10.0.0.3 -c 4
h3# ping 10.0.0.1 -c 4
h3# ping 10.0.0.2 -c 4
mininet> quit
```

2)广播网络效率测试

```
sudo python three_nodes_bw.py
mininet> xterm h1 h2 h3 b1
b1# ./hub
h1# iperf -s
h2# iperf -c 10.0.0.1 -t 30
h3# iperf -c 10.0.0.1 -t 30
mininet> quit
```

上述过程是以 h1 作为服务器，h2、h3 作为客户进行向 h1 的访问

```
sudo python three_nodes_bw.py
mininet> xterm h1 h1 h2 h3 b1
b1# ./hub
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

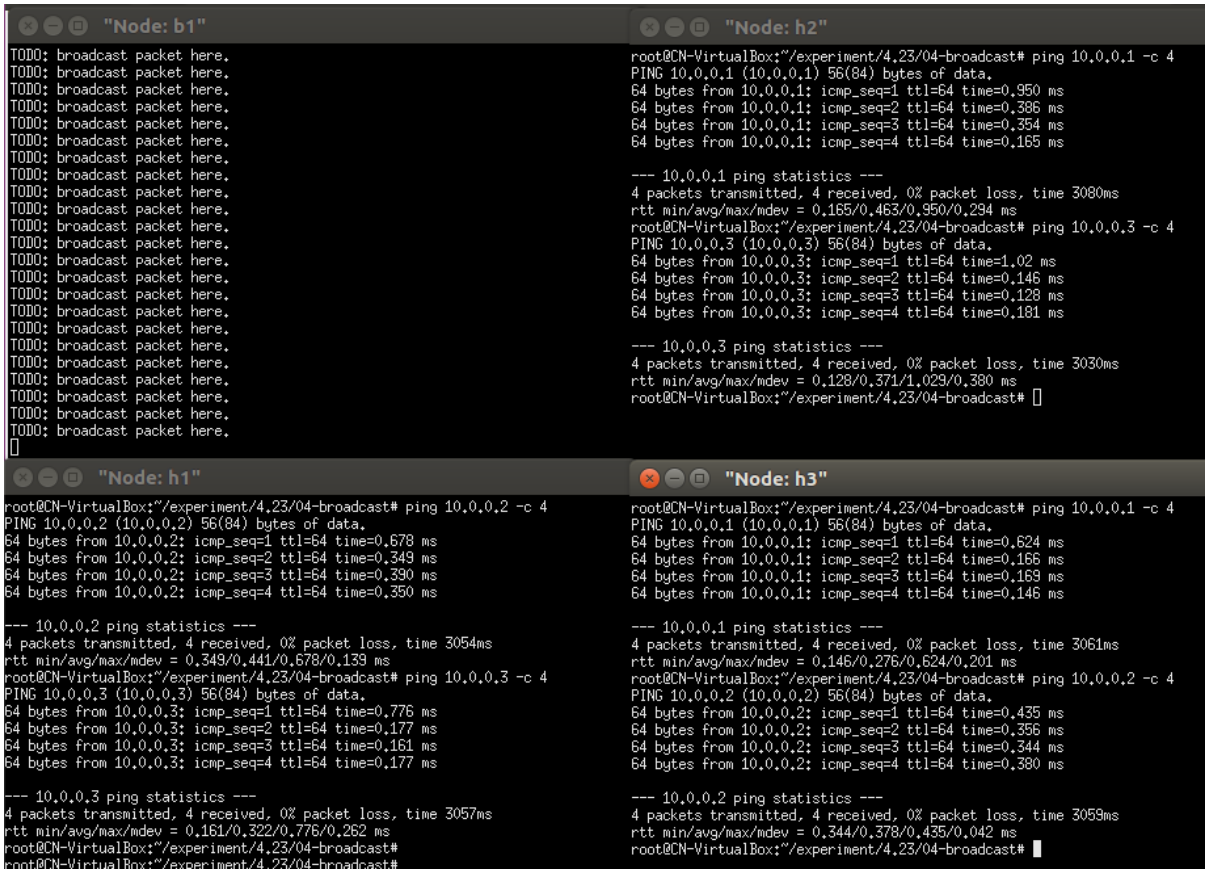
3)环形拓扑数据包环路测试

```
sudo python loop_topo.py
mininet> xterm h1 h2 b1 b2 b3
b1# ./hub
b2# ./hub
```

```
b3# ./hub
h2# wireshark &
h1# ping -c 1 10.0.0.2
mininet> quit
```

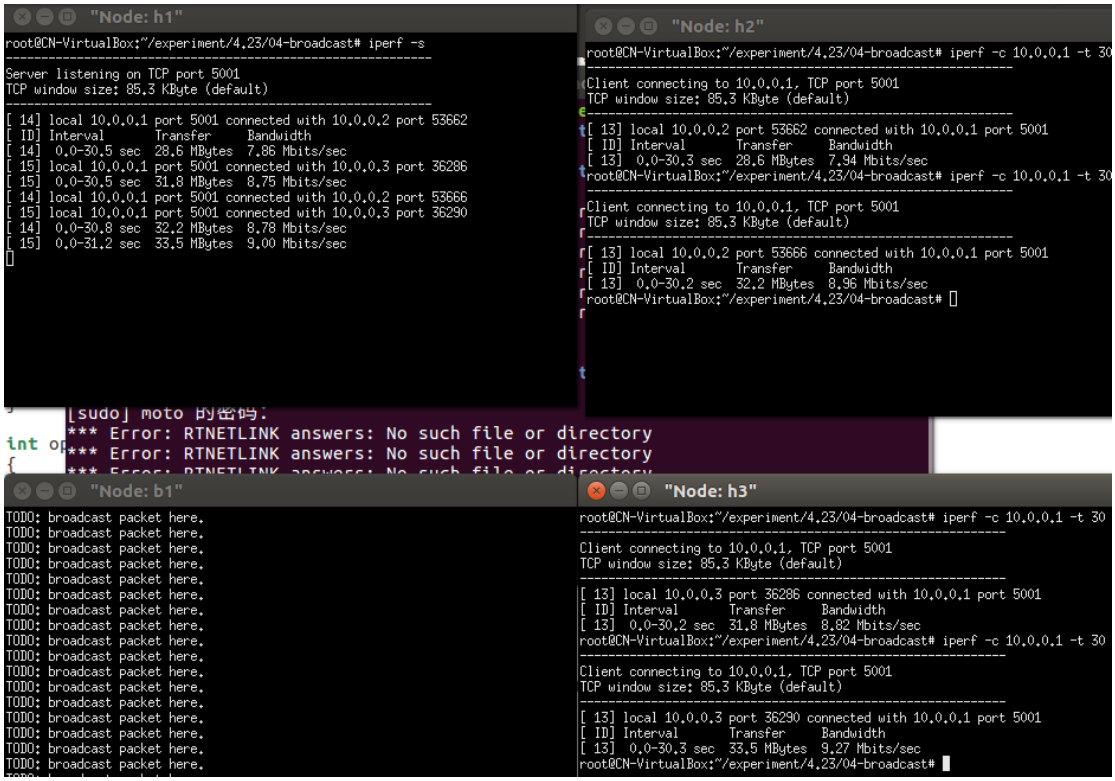
三、实验结果及分析

1. 实验结果

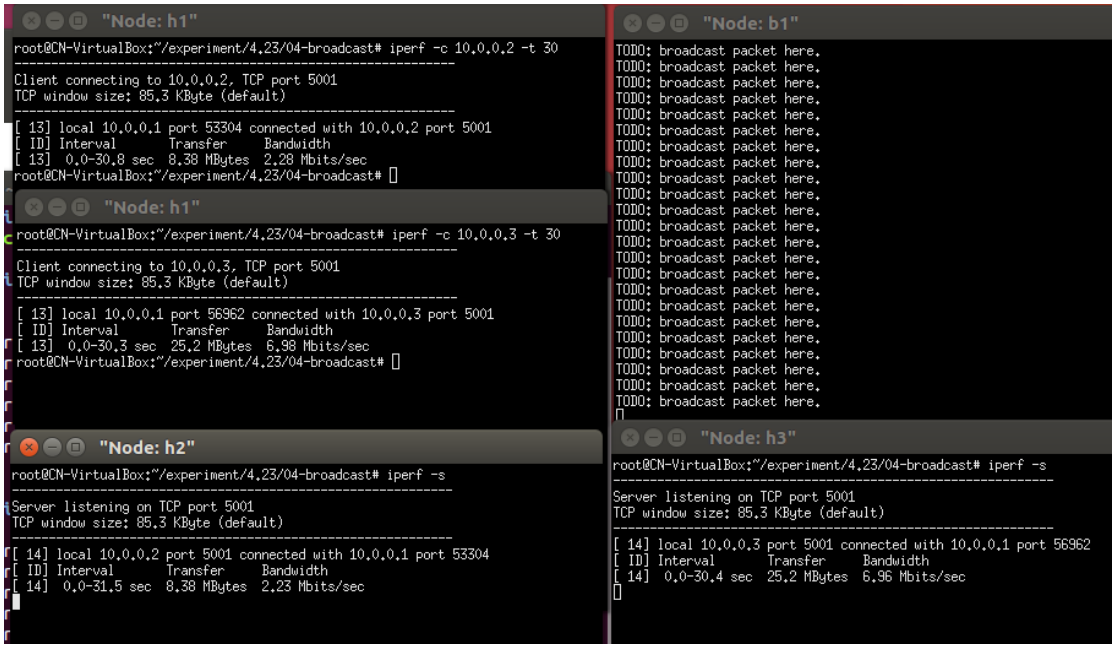


图二 广播网络功能测试

从打印结果可以看出，从 h1 ping h2、h3，从 h2 ping h1、h3，从 h3 ping h1、h2 的数据通路都是通的，这就说明我们实现的 hub 可以完成广播的目的。



图三 广播网络效率测试—h1 为 server，h2、h3 为 client



图四 广播网络效率测试—h2、h3 为 server，h1 为 client

当 h1 作为 Server，h2 和 h3 作为 Client 时，做了两组测试：1）h2 先进行 iperf 测试，结束发包之后再启动 h3 的测试。测试结果：h2-h1：7.86Mbps h3-h1：8.75Mbps 2）h2 和 h3 同时启动测试。测试结果：h2-h1：8.78Mbps h3-h1：9.00 Mbps

实验结果发现 h2 与 h3 是否并行测试对实时带宽影响不大，这说明了带宽的双向性：理论上，当 h2 与 h3 同时以 10Mbps 的速率向 h1 发送数据包时，从 h2 发出的一个数据包到达 b1 结点后，会被复制成两份，一份发往 h1，一份发往 h3。从 h3 发出的一个数据包到达 b1 结点后，会被复制成两份，一份发往 h1，一份发往 h2。于是，h1 与 b1 之间的通路上，

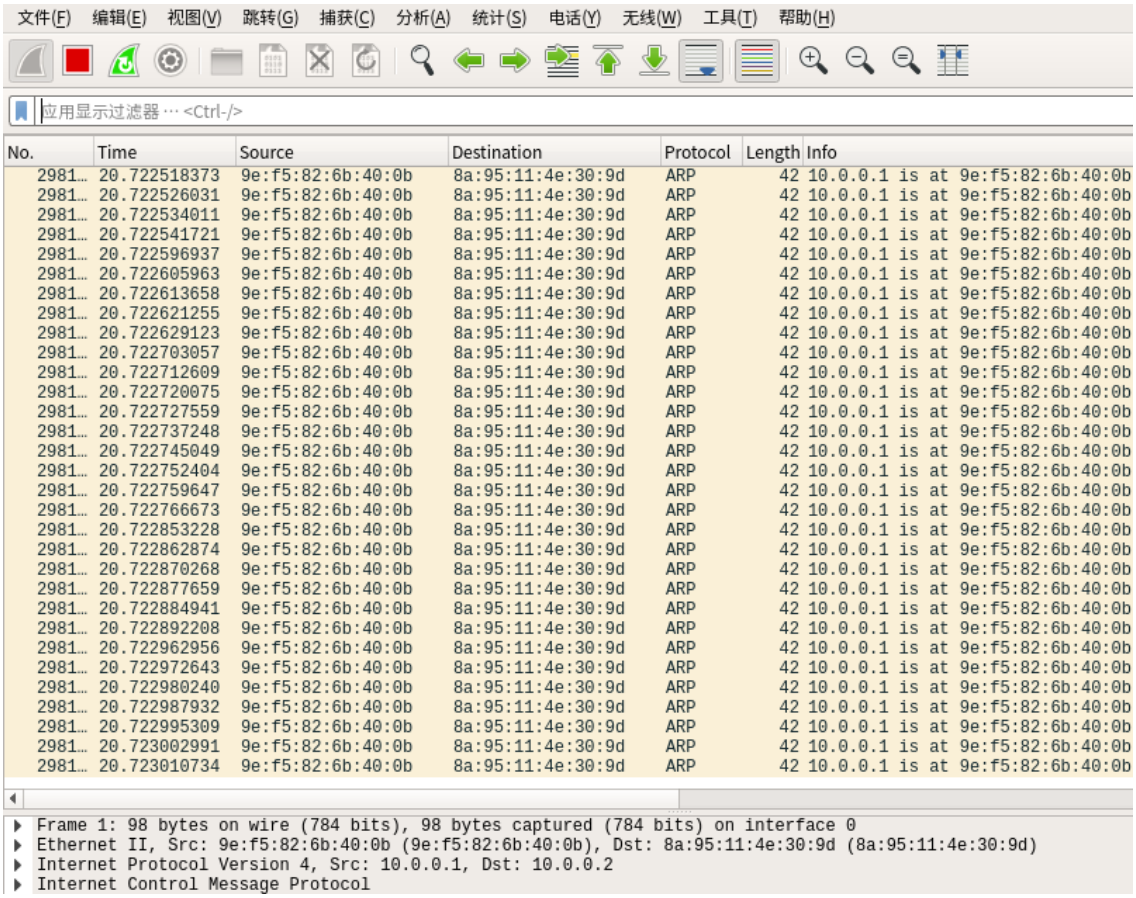
从 b1 到 h1 的方向上是传输速率为 10Mbps+10Mbps = 20Mbps 的数据里流，h2 与 b1、h3 与 b1 之间的通路上都存在着双向的传输速率为 10Mbps 的数据流。理论分析结果与实际测量结果较为相似。

当 h1 作为 Client，h2 和 h3 作为 Client 时，也做了两组测试：1）h1 分别进行与 h2 和 h3 的 iperf 测试。测试结果：h1-h2：7.59Mbps h1-h3：6.78Mbps 2）h1 同时与 h2 和 h3 进行 iperf 测试。测试结果：h1-h2：2.28Mbps h1-h3：6.98 Mbps

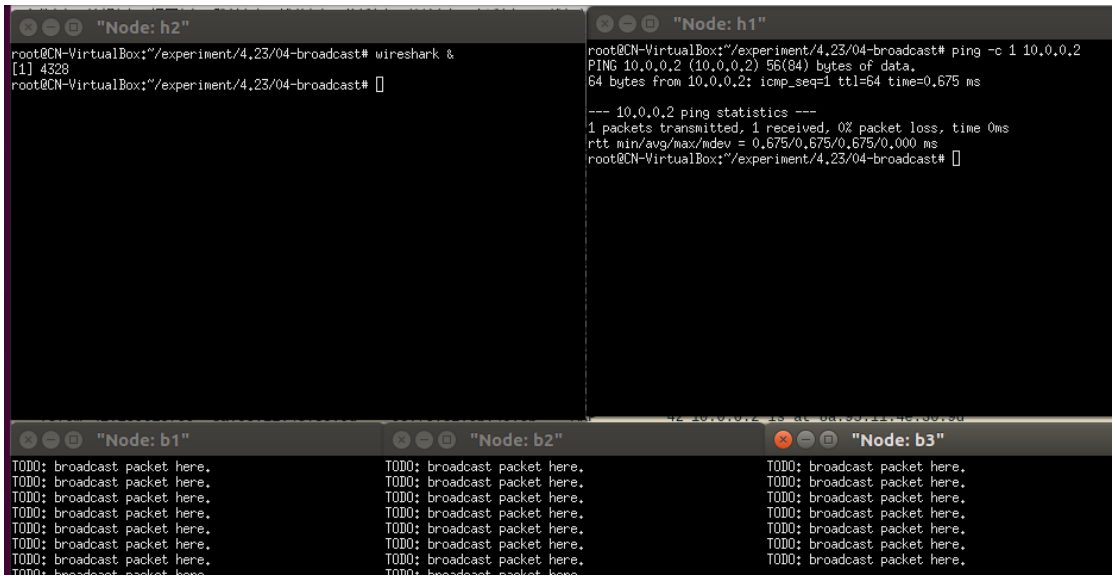
实验结果发现，并发对带宽的影响较大，这说明了广播转发会占用无效带宽：
单发时，h1 以 20Mbps 的速率将数据发送到 b1，b1 会将数据包复制成两份发往 h2 和 h3。但是 b1 与 h2、h3 之间的带宽只有 10Mbps，从 h1 发到 b1 的包会在 b1 中排队等待发送，总时延增加了排队时延。再加上数据处理时间，于是实际测试中测试速率不会达到理想的 10Mbps。

并发时，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。

测试时，发现 h1-h3 之间的带宽最高，其次是 h2-h3，最后是 h1-h2。多次测试结果基本满足该规律。猜测是 hub 在确定接收消息和转发消息时，是按照一定次序开始遍历的，也就是先遍历到的先处理，相对带宽就高；后遍历到的就后处理，相对带宽就低。



图五 环形拓扑测试—在 h2 结点上的抓包



图六　环形拓扑测试—h1 只发送一个数据包，该包在环路中陷入循环

实验过程中，可以看到 b1、b2、b3 不停的在打印新进行广播的信息，h2 结点的抓包显示接收到的包全部是 h1 起初发往 h2 的一个包，这说明该包在环形拓扑中不停的被转发不停旋转，对资源造成了极大的浪费。

另外发现的有趣的事情是，结束实验时，当退出 h1 和 h2 节点时，能看到 b1、b2、b3 之间仍然在继续着数据报的接收和广播。但是，当继续退出任意一个节点后，数据报的转发会立即结束，因为广播不会发送给广播源。

2. 实验分析

我们来看看实现 hub 的代码：

```
int main(int argc, const char **argv)
{
    if (getuid() && geteuid()) {
        printf("Permission denied, should be superuser!\n");
        exit(1);
    }
    init_ustack();
    ustack_run();
    return 0;
}
```

首先通过 getuid 和 geteuid 获得调用程序的真实用户 ID 和执行目前进程有效的用户识别码。随后通过 init_ustack 进行初始化。

```
void init_ustack()
{
                    instance = malloc(sizeof(ustack_t));
                    bzero(instance, sizeof(ustack_t));
                    init_list_head(&instance->iface_list);
                    init_all_ifaces();
}
```

初始化过程要先建立 ustack_t 类型的一个变量 instance，然后将 instance 中的 iface_list 链表初始化，再将所有的 iface（）初始化。

之后执行 ustack_run()，运行 hub 服务。

```
void ustack_run()
{
        struct sockaddr_ll addr;
        socklen_t addr_len = sizeof(addr);
        char buf[ETH_FRAME_LEN];
        int len;

        while (1) {
                int ready = poll(instance->fds, instance->nifs, -1);
                if (ready < 0) {
                                perror("Poll failed!");
                                break;
                            }
                else if (ready == 0)
                continue;

                for (int i = 0; i < instance->nifs; i++) {
                    if (instance->fds[i].revents & POLLIN) {
                    len = recvfrom(instance->fds[i].fd, buf, ETH_FRAME_LEN,
0, \ (struct sockaddr*)&addr, &addr_len);
                    if (len <= 0) {
                    log(ERROR, "receive packet error: %s", strerror(errno));
                                    }
                    else if (addr.sll_pkttype == PACKET_OUTGOING) {
              // XXX: Linux raw socket will capture both incoming and
              // outgoing packets, while we only care about the incoming ones.

              // log(DEBUG, "received packet which is sent from the "
              //             "interface itself, drop it.");
                                    }
                    else {
                        iface_info_t               *iface               =
fd_to_iface(instance->fds[i].fd);
                        if (!iface)
                        continue;

                        char *packet = malloc(len);
                        if (!packet) {
                                log(ERROR, "malloc failed when receiving
packet.");
```

```
                    continue;
                        }
                memcpy(packet, buf, len);
                handle_packet(iface, packet, len);
                }
                    }
                }
               }
    }
```

首先将 instance->fds 文件指针挂载到内部的等待队列，然后循环遍历每一个 instance->fd 的组分,如果有消息出入就使用 recvfrom 函数抓取。由于 linux 的 raw socket 指令会抓取发送出去的包和收到的包，需要检测是不是发送出去的，如果是就将该包丢弃。如果该包是接收到的包，就使用 fd_to_iface 函数对端口链表进行遍历，将该包的发送端口值返回，并记录到 iface 中。随后在内存中 malloc 一个区域用来存放接收到的报文，之后再调用 handle_packet 函数将收到的包广播出去。handle_packet 函数非常简单，只是调用了 broadcast_packet 广播函数之后，将之前的接收包 free 掉。

（二）实验代码详解

一、broadcast_packet

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

    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);
    }
}
```

当调用到 broadcast_packet 函数时，首先会在 Node 界面打印标准输出，告知程序员包已经到达 Hub，准备广播。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，如果当前遍历到的主机不是发送该消息的主机，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。

二、loop_topo.py

```
#!/usr/bin/python

import os
import sys
import glob

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.cli import CLI

script_deps = [ 'ethtool' ]

def check_scripts():
    dir = os.path.abspath(os.path.dirname(sys.argv[0]))

    for fname in glob.glob(dir + '/' + 'scripts/*.sh'):
        if not os.access(fname, os.X_OK):
            print '%s should be set executable by using `chmod +x
```

```
$script_name`` % (fname)
                sys.exit(1)

#以上部分是检测 script 的可执行权限

        for program in script_deps:
            found = False
            for path in os.environ['PATH'].split(os.pathsep):
                exe_file = os.path.join(path, program)
                if os.path.isfile(exe_file) and os.access(exe_file, os.X_OK):
                    found = True
                    break
            if not found:
                print ``%s` is required but missing. which could be installed
via `apt` or `aptitude`` % (program)
                sys.exit(2)

#以上部分是检测是否安装需要用到的库

    # Mininet will assign an IP address for each interface of a node
    # automatically, but hub or switch does not need IP address.
    def clearIP(n):
        for iface in n.intfList():
            n.cmd('ifconfig %s 0.0.0.0' % (iface))

    #以上宏为清 IP，将中间路由器的 IP 端口设置为 0.0.0.0

    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)

    if __name__ == '__main__':
        check_scripts()
```

```
    topo = BroadcastTopo()
    net = Mininet(topo = topo, link = TCLink, controller = None)

    h1, h2, b1, b2, b3 = net.get('h1', 'h2', 'b1', 'b2', 'b3')
    h1.cmd('ifconfig h1-eth0 10.0.0.1/8')
    h2.cmd('ifconfig h2-eth0 10.0.0.2/8')

#将 h1 和 h2 的 IP 地址设置好

    clearIP(b1)
    clearIP(b2)
    clearIP(b3)

    for h in [ h1, h2, b1, b2, b3 ]:
        h.cmd('./scripts/disable_offloading.sh')
        h.cmd('./scripts/disable_ipv6.sh')

#禁用 offloading 和 IPV6

    net.start()
    CLI(net)
    net.stop()
```