北京邮电大学计算机学院

2021-2022 学年第 1 学期实验报告

课程名称： 计算机网络

实验名称： 网络层实验

实验完成人：

姓名： 胡杨 **学号：** 2019212008 **成绩：** _____

指导教师： 雷友珣

日　　期： 2021 年 12 月 14 日

一、 实验目的

通过本实验使学生理解和掌握计算机网络层协议的基本工作机制，包括：网络层地址、IP 地址前缀、IP 分组转发、路由表、静态路由、动态路由算法；并通过本实验使学生掌握 Linux 系统网络配置基础命令。

二、 实验任务

1） 利用虚拟机平台，进行基于 Linux 系统的 IP 分组转发和静态路由实验：设计模拟实验环境，完成静态路由配置，实现 IP 分组转发功能。
2） 距离矢量路由算法的模拟实现。

三、 实验内容

1） 利用虚拟机平台软件（VirtualBox）创建三台 Server 版 Ubuntu 虚拟机，并使用虚拟机平台软件（VirtualBox）提供的虚拟以太局域网功能，搭建虚拟以太网络环境。将所创建的一台 Ubuntu 虚拟机作为实验中的路由器（下文记为 router），另外两台 Ubuntu 虚拟机作为实验中的主机（下文记为 host1 和 host2）,router 与 host1 在一个虚拟以太网上，router 与 host2 在另一虚拟以太网上。配置 router、host1, host2 的网卡（NIC），规划三台 Ubuntu 虚拟机所在网络的网络地址（IP 地址前缀）、网卡使用的 IP 地址，配置三台 Ubuntu 虚拟机的路由表，实现虚拟机 IP 地址间的互通。本实验采用下表中的 IP 地址。

表 1

	IP 地址	所在网络 IP 地址前缀
host1 网卡 1	192.168.53.1	192.168.53.0/24
router 网卡 1	192.168.53.2	192.168.53.0/24
router 网卡 2	192.168.58.7	192.168.58.0/24
host2 网卡 1	192.168.58.8	192.168.58.0/24

2） 在 Linux 系统中，开发距离矢量路由算法模拟软件。本实验中提供一个 C++语言编写的距离矢量路由算法模拟程序供参考。该软件实现了初步的距离矢量路由算法功能，包括：模拟计算机网络节点之间的消息的收发、网络节点和链路正常状态下的路由矢量算法模拟实现。

在学习距离矢量路由算法的基础上，基于实验中提供的参考程序，完善参考程序，完成网络节点或链路故障情况下的距离矢量路由过程模拟。

四、 实验环境

Ubuntu 18.0.6, g++1z

五、 实验过程描述

1. 搭建路由器

1.1 修改 host1 IP 地址

```
huyang@host1:~$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 100
0
    link/ether 08:00:27:90:65:56 brd ff:ff:ff:ff:ff:ff
    inet 192.168.53.1/24 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe90:6556/64 scope link
       valid_lft forever preferred_lft forever
```

1.2 修改 host2 IP 地址

```
huyang@host2:~$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 100
0
    link/ether 08:00:27:df:cd:36 brd ff:ff:ff:ff:ff:ff
    inet 192.168.53.2/24 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fedf:cd36/64 scope link
       valid_lft forever preferred_lft forever
```

1.3 从 host1 ping host2，在 host2 抓包

```
huyang@host1:~$ ping -c 2 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
64 bytes from 192.168.53.2: icmp_seq=1 ttl=64 time=0.461 ms
64 bytes from 192.168.53.2: icmp_seq=2 ttl=64 time=1.01 ms

--- 192.168.53.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1016ms
rtt min/avg/max/mdev = 0.461/0.740/1.019/0.279 ms
```

```
huyang@host2:~$ sudo tcpdump ip -c 4 -i enp0s3 host 192.168.53.1 -w ~/tcpdump.log
tcpdump: listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
4 packets captured
4 packets received by filter
0 packets dropped by kernel
```

1.4 查看抓包记录

```
huyang@host2:~$ sudo tcpdump -r tcpdump.log
reading from file tcpdump.log, link-type EN10MB (Ethernet)
03:55:28.679130 IP 192.168.53.1 > host2: ICMP echo request, id 1150, seq 1, length 64
03:55:28.679150 IP host2 > 192.168.53.1: ICMP echo reply, id 1150, seq 1, length 64
03:55:29.695461 IP 192.168.53.1 > host2: ICMP echo request, id 1150, seq 2, length 64
03:55:29.695493 IP host2 > 192.168.53.1: ICMP echo reply, id 1150, seq 2, length 64
```

1.5 配置虚拟子网后，host1 无法 ping 通 host2

```
huyang@host1:~$ ping -c 2 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
From 192.168.53.1 icmp_seq=1 Destination Host Unreachable
From 192.168.53.1 icmp_seq=2 Destination Host Unreachable

--- 192.168.53.2 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1006ms
pipe 2
```

1.6 然后，将三台虚拟机的 IP 地址改为表 1 中的值，此时（×表示不能 ping 通，√表示可以 ping 通）：

	Host 1	Host 2	Router 1	Router 2
Host 1	√	×	√	×
Host 2	×	√	×	√

1.7 host1 ping 通 router

```
huyang@host1:~$ ping -c 2 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
64 bytes from 192.168.53.2: icmp_seq=1 ttl=64 time=0.809 ms
64 bytes from 192.168.53.2: icmp_seq=2 ttl=64 time=0.487 ms

--- 192.168.53.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.487/0.648/0.809/0.161 ms
```

1.8 host2 ping 通 router

```
huyang@host2:~$ ping -c 2 192.168.58.7
PING 192.168.58.7 (192.168.58.7) 56(84) bytes of data.
64 bytes from 192.168.58.7: icmp_seq=1 ttl=64 time=0.794 ms
64 bytes from 192.168.58.7: icmp_seq=2 ttl=64 time=0.444 ms

--- 192.168.58.7 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.444/0.619/0.794/0.175 ms
```

1.9 在 host 1 中增加到达子网 2 的路由项，然后 ping 通了 router 的网卡 2

```
huyang@host1:~$ sudo ip route add 192.168.58.0/24 via 192.168.53.2 dev enp0s3
[sudo] password for huyang:
huyang@host1:~$ ping -c 2 192.168.58.7
PING 192.168.58.7 (192.168.58.7) 56(84) bytes of data.
64 bytes from 192.168.58.7: icmp_seq=1 ttl=64 time=0.430 ms
64 bytes from 192.168.58.7: icmp_seq=2 ttl=64 time=0.455 ms

--- 192.168.58.7 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1008ms
rtt min/avg/max/mdev = 0.430/0.442/0.455/0.024 ms
```

1.10 在 host 2 中增加到达子网 1 的路由项，host1 仍无法 ping 通 host2

```
huyang@host1:~$ ping -c 2 192.168.58.8
PING 192.168.58.8 (192.168.58.8) 56(84) bytes of data.

--- 192.168.58.8 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1017ms
```

1.11 启动 router 的 IPv4 转发

```
# Uncomment the next line to enable packet forwarding for IPv4
net.ipv4.ip_forward=1

# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
#net.ipv6.conf.all.forwarding=1


###################################################################
# Additional settings - these settings can improve the network
"/etc/sysctl.conf" 77L, 2682C written
huyang@router:~$ sudo sysctl -p
net.ipv4.ip_forward = 1
huyang@router:~$ _
```

1.12 host 1 成功 ping 通 host 2，host 2 抓包

```
huyang@host1:~$ ping -c 5 192.168.58.8
PING 192.168.58.8 (192.168.58.8) 56(84) bytes of data.
64 bytes from 192.168.58.8: icmp_seq=1 ttl=63 time=0.839 ms
64 bytes from 192.168.58.8: icmp_seq=2 ttl=63 time=0.816 ms
64 bytes from 192.168.58.8: icmp_seq=3 ttl=63 time=0.999 ms
64 bytes from 192.168.58.8: icmp_seq=4 ttl=63 time=1.17 ms
64 bytes from 192.168.58.8: icmp_seq=5 ttl=63 time=0.926 ms

--- 192.168.58.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 0.816/0.951/1.179/0.136 ms
```

```
huyang@host2:~$ sudo tcpdump ip -c 10 -i enp0s3 host 192.168.53.1 -w ~/tcpdump2.log
tcpdump: listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
10 packets captured
10 packets received by filter
0 packets dropped by kernel
```

1.13 host2 抓包内容

```
huyang@host2:~$ sudo tcpdump -r tcpdump2.log
reading from file tcpdump2.log, link-type EN10MB (Ethernet)
10:01:25.459185 IP 192.168.53.1 > host2: ICMP echo request, id 1333, seq 1, length 64
10:01:25.459205 IP host2 > 192.168.53.1: ICMP echo reply, id 1333, seq 1, length 64
10:01:26.460451 IP 192.168.53.1 > host2: ICMP echo request, id 1333, seq 2, length 64
10:01:26.460472 IP host2 > 192.168.53.1: ICMP echo reply, id 1333, seq 2, length 64
10:01:27.461916 IP 192.168.53.1 > host2: ICMP echo request, id 1333, seq 3, length 64
10:01:27.461935 IP host2 > 192.168.53.1: ICMP echo reply, id 1333, seq 3, length 64
10:01:28.464070 IP 192.168.53.1 > host2: ICMP echo request, id 1333, seq 4, length 64
10:01:28.464090 IP host2 > 192.168.53.1: ICMP echo reply, id 1333, seq 4, length 64
10:01:29.465439 IP 192.168.53.1 > host2: ICMP echo request, id 1333, seq 5, length 64
10:01:29.465463 IP host2 > 192.168.53.1: ICMP echo reply, id 1333, seq 5, length 64
```

1.14 添加 192.168.99.0/24 的路由项后，从 host 1 ping 192.168.99.2， 在 host 2 和 router 上抓包

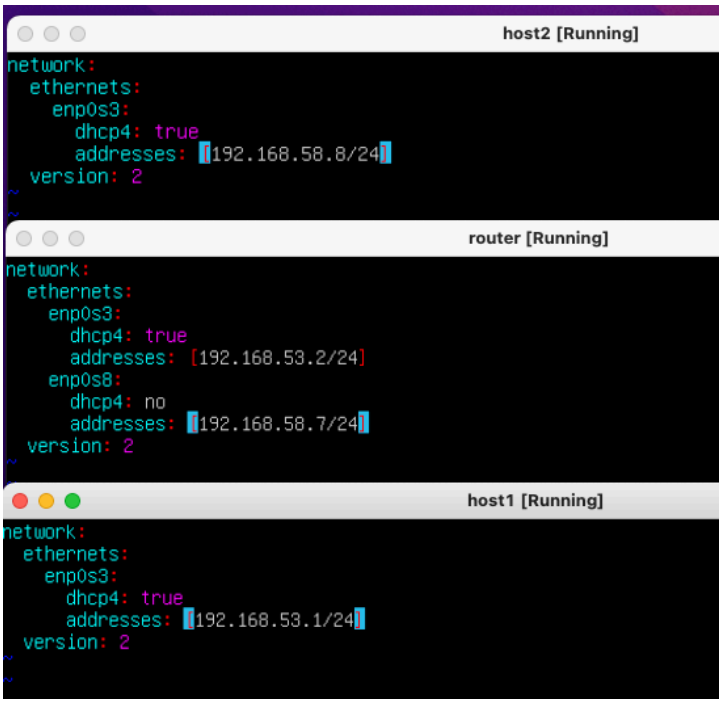
```
huyang@host1:~$ ping -c 6 192.168.99.2
PING 192.168.99.2 (192.168.99.2) 56(84) bytes of data.

--- 192.168.99.2 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5106ms
```

```
huyang@host2:~$ sudo tcpdump ip -c 12 -i enp0s3 host 192.168.53.1 -w ~/tcpdump3.log
tcpdump: listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
^C6 packets captured
6 packets received by filter
0 packets dropped by kernel
huyang@host2:~$ sudo tcpdump -r ~/tcpdump3.log
reading from file /home/huyang/tcpdump3.log, link-type EN10MB (Ethernet)
14:32:13.008296 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 1, length 64
14:32:14.018555 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 2, length 64
14:32:15.043057 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 3, length 64
14:32:16.067165 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 4, length 64
14:32:17.090725 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 5, length 64
14:32:18.114384 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 6, length 64
```

```
huyang@router:~$ sudo tcpdump ip -c 12 -i enp0s8 host 192.168.99.2 -w ~/tcpdump0.log
tcpdump: listening on enp0s8, link-type EN10MB (Ethernet), capture size 262144 bytes
^C6 packets captured
6 packets received by filter
0 packets dropped by kernel
huyang@router:~$ tcpdump -r ~/tcpdump0.log
tcpdump: /home/huyang/tcpdump0.log: Permission denied
huyang@router:~$ sudo tcpdump -r ~/tcpdump0.log
reading from file /home/huyang/tcpdump0.log, link-type EN10MB (Ethernet)
14:32:12.646041 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 1, length 64
14:32:13.656286 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 2, length 64
14:32:14.680779 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 3, length 64
14:32:15.704810 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 4, length 64
14:32:16.728472 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 5, length 64
14:32:17.752090 IP 192.168.53.1 > 192.168.99.2: ICMP echo request, id 1139, seq 6, length 64
```

1.15 修改三台虚拟机为静态地址



2. 改进距离矢量算法

2.1 实现功能：在参考代码的基础上，添加了以下功能：

- 在路由表更新后打印出新的路由表
- 让每个节点定期向其邻居发送路由表
- 添加了某个节点故障（下线）后的处理
- 缓解了无穷计数和循环路由的问题

2.2 软件结构

此次实验主要改动位于 main.cpp、dv_algorithm.cpp 和 networkrouting.cpp。提交的压缩包包含本地 Git 记录。

2.2.1 dv_algorithm.h/.cpp

网络层路由算法，例如发送或泛洪消息，检测节点在线状态等功能。

2.2.2 dv_msg.h/.cpp

消息类，负责编码、解码消息。

2.2.3 networkrouting.h/.cpp

用于存放路由相关信息，如路由表、邻居地址等。

2.2.4 transport.h/.cpp

传输层，调用 socket 接口来传送消息。

2.2.5 user_cmd.h/.cpp

用于处理用户在终端的输入。

2.2.6 main.cpp

初始化节点，并按照既定逻辑工作，实现距离矢量路由算法。

2.3 算法概述

2.3.1 在更新时打印新的路由表

在 main.cpp 中的主循环中，每次改动后打印路由表。使用 has_modification 变量记录是否有改动。

```
if (has_modification) {
    // print new routing table
    net_info.display_routing_table();
}
```

2.3.2 获取节点下线信息

获取在经过多次尝试（见 2.4 节）后，笔者决定使用一个数组 neigh_count 来记录一个节点发来的 ping 数。每个时钟周期结束时对每个已连接的节点的 neigh_count 递增 1，如果在任何时候收到了一个节点的 ping 消息，则将它的 neigh_count 置 0。这样，我们就可以设置一个阈值 N（代码中为 3，不建议设置为小于 3 的值），如果一个节点在连续的 N 个周期内都

没有发来一个 ping 消息，则认为它已经掉线。另外，如果按照原写法，在每次更新路由表后就泛洪，会导致以太网消息队列阻塞，引起消息严重延迟。因此，我们选择每 3 个 ping 周期进行一次泛洪。

为了实现以上功能，本人的改动主要包括：在 DV_Routing 类中实现了 increase_counter()和 reset_counter()方法；在 dv_algorithm.cpp 中，修改了 sigalrmHandler()函数的逻辑：每个时钟周期先递增计数器并检查是否有节点掉线，然后向邻居泛洪 ping 消息，最后，每 3 个时钟周期向邻居节点泛洪自己的路由表。另外，还修改了 main.cpp 中的主函数：加入了 ACTIVE_MODE 宏定义，如果开启才会及时泛洪，否则等待周期性泛洪。

```
int DV_Routing::increase_counter(Network_Info * p_net_info) {
    bool node_down = false; // node_down determines if a new flooding is needed
    // increase counter
    for (auto it1 = p_net_info->neigh_count.begin(); it1 != p_net_info->neigh_count.end();) {
        it1->second++;
        if (it1->second > thres) {
            node_down = true;
            // get node_id of the downed node
            std::string down_node_id = it1->first;

            // update neigh_count
            it1 = p_net_info->neigh_count.erase(it1);

            // update neigh_connected (topology)
            p_net_info->neigh_connected[down_node_id] = 0;

            // update routing table
            for (auto & it2 : p_net_info->distance_table) {
                // mark distance of node infinity to indicate it's down
                if (it2.first == down_node_id || it2.second.first == down_node_id) {
                    it2.second.second = INF;
                }
            }
        } else {
            it1++;
        }
    }
    // tell node to flood new routing table by returning -1
    if (node_down) {
        start_timer();
        return -1;
    }
    return 0;
}
```

增加计数器

```
int DV_Routing::reset_counter(Network_Info * p_net_info, std::string node_id) {

#ifdef VERBOSE
    std::cout << "Received Ping from " << node_id << "." << std::endl;
#endif

    p_net_info->neigh_count[node_id] = 0; // insert if it doesn't exist
    return 0;
}
```

重置计数器

```
static void sigalrmHandler(int sig) { // flood ping message at every interval
    gotAlarm = 1;
    counter++;

    // increase ping counter for connected nodes at every interval, see networkrouting.h for more
    if (dv_routing.increase_counter(&net_info) == -1) {
        // flood new routing table because a neighbor is down
        DV_Msg *p_distance_dv_msg = new DV_Msg;
        dv_routing.construct_dv_msg(&net_info, PATH_DISTANCE_MSG, p_distance_dv_msg);
        dv_routing.flood_dv_msg(p_t_l, &net_info, p_distance_dv_msg);
        delete p_distance_dv_msg;

#ifdef VERBOSE
        std::cout << "\nFound unconnected neighbor(s)!" << std::endl;
#endif

        net_info.display_routing_table();
    }

    // construct and flood ping message
    DV_Msg dv_msg;
    dv_routing.construct_dv_msg(&net_info, PING_MSG, &dv_msg);
    dv_routing.flood_ping_dv_msg(p_t_l, &net_info, & dv_msg);

    if (counter == 5) { // flood routing table to neighbors every 5 intervals
        counter = 0;

        DV_Msg *p_distance_dv_msg = new DV_Msg;
        dv_routing.construct_dv_msg(&net_info, PATH_DISTANCE_MSG, p_distance_dv_msg);
        dv_routing.flood_dv_msg(p_t_l, &net_info, p_distance_dv_msg);
        delete p_distance_dv_msg;

#ifdef VERBOSE
        std::cout << "[[ Flooded RT to neighbors! ]]" << std::endl;
#endif
    }
}
```

sigalrmHandler()

2.3.3 节点故障（下线）的处理

在一个节点下线后，它的邻居会检测到，并将路由表中涉及到下线节点的条目的距离设为无穷（用 INF，也就是 32767， 即 std::numeric_limits<int16_t>::max() 表示），**然后马上将新的路由表泛洪（无论 ACTIVE_MODE 是否启用，见 2.3.4）**。在一个节点接收到了某个节点的距离为无穷时（通过 new_path_cost 是否溢出判断），它也会对自己的路由表做相应的修改，**然后马上泛洪路由表（无论 ACTIVE_MODE 是否启用，见 2.3.4）**。这一部分细节逻辑较多，详见 main.cpp 代码注释。

```
} else if (net_info.distance_table.count(element.to_node_id) != 0) { // if an entry exists, check if a shorter path is discoverd
    int16_t cost_of_new_path = element.cost + net_info.neigh_cost[p_received_dv_msg->from_node_id];
    int16_t cost_of_old_path = net_info.distance_table[element.to_node_id].second;

    if (cost_of_new_path < 0 && // node down
        net_info.distance_table[element.to_node_id].first == p_received_dv_msg->from_node_id &&
        cost_of_old_path < INF) { // an infinity distance will result in overflow
        std::cout << "[ERROR]A node is down!" << std::endl;
        /**
         * Note that we are not able to find out exactly which node is down - We just know that a previously
         * reachable node is no longer reachable. If this node is a neighbor, then great! We know that a
         * neighbor is down. But also not-so-great, because we would have already noticed that from the timed-out
         * pings long before we receive an updated routing table from a neighbor telling us another neighbor
         * is down. However, in some extreme cases (i.e., it is faster to reach neighbor A via neighbor B than
         * directly accessing A), that might be the case. So, we include the following "redundant" code.
         *
         * EDIT: As it turns out, deprecated INF messages would also result in the following [WARNING] situation.
         * This is more of a bonus effect because then the observer could tell that the network is congested.
         */

        // update neigh_connected if it's a neighbor AND somehow it's not yet updated
        if (net_info.neighbour[element.to_node_id] == 1) {
            std::cout << "[WARNING]Neighbor " << element.to_node_id << " is down!" << std::endl;
            net_info.neigh_connected[element.to_node_id] = 0;
        }

        // Start INF_TIMER to avoid count-to-inf and circular routing problem
        cost_of_old_old_path = net_info.distance_table[element.to_node_id].second;
        dv_routing.start_timer();

        /**
         * Update routing table to infinity. Note that you can't just remove this entry altogether
         * because then you wouldn't be able to propagate this "node down" information to your neighbors.
         * Also, using infinity can allow later updates on the path to the destination node via another
         * node that is NOT the downed node.
         */
        has_modification = true;
        net_info.distance_table[element.to_node_id].second = INF;
```

```
// #ifdef ACTIVE_MODE
        /**
         * Unlike other news, for the timer mechanism to work, INF news are flooded immediately (regardless
         * of whether ACTIVE_MODE is set) when they are ready.
         */

        // flood an infinity route
        // generate a dv_msg containing ONLY the updated entry
        p_distance_dv_msg = new DV_Msg;
        p_distance_dv_msg->insert_from_node_id(PATH_DISTANCE_MSG, net_info.node_id);

        path_cost.to_node_id = element.to_node_id;
        path_cost.cost = INF;
        p_distance_dv_msg->insert_a_path_cost(&path_cost);

        // send message to neighbors
        dv_routing.flood_dv_msg(p_t_l, &net_info, p_distance_dv_msg);

        delete p_distance_dv_msg;
// #endif
```

2.3.4 无穷计数和循环路由问题

在完成基础逻辑后，笔者视图缓解这两个问题。经过调研和与室友讨论，决定使用抑制计时器的方法解决这个问题。抑制计时器的原理很简单：无穷计数和循环路由问题出现的原因是因为在一个节点下线后，还没等整个网络都接收到这个 INF 消息，它就被淹没了——因为在 INF 消息传播时，到下线节点的有穷路径也在传播。很快，所有节点又以为自己找到了到下线节点的更短路径，只有下线节点的邻居还会时不时发出 INF 消息，导致无穷计数问题。抑制计时器的基本思路是：当一个节点接收到 INF 消息，检查自己的路由表，如果自己的路由表有更新，则启动抑制计时器。**在这个时间段内，该节点不接收关于已被更新成 INF 的表项的更新，除非这个新的路径比更新成 INF 之前的路径还要短。我们把这种状态成为暂时的“中毒”，INF 消息就是“毒药”。**这样一来，如果我们把抑制计时器时长设置为一条 INF 消息传遍整个网络所需时长（这就是为什么我们要及时泛洪 INF 消息），就可以让整个网络都暂时“中毒”。在抑制计时器结束后，如果网络中还有节点能收到“下线节点”的 ping（也就是说它并没有下线，只是某条链路断了），那么到该节点的最短路径会被重新找到。这样做的话：

- 在一个节点下线后，大家迅速响应，不会导致无穷计数而浪费时间
- 如果一个节点没有下线，只是某一条链路断掉了，那么会造成一些不必要的延迟

```cpp
void DV_Routing::start_timer() {
    inf_timer_enabled = true;
    inf_timer_start = clock();
    std::cout << "<INF_TIMER started @ " << Network_Info::get_time_string() << ">" << std::endl;
}

void DV_Routing::stop_timer() {
    inf_timer_enabled = false;
    inf_timer_start = 0;
    std::cout << "<INF_TIMER expired @ " << Network_Info::get_time_string() << ">" << std::endl;
}
```

启动、关闭计时器

```cpp
// check inf timer
if (dv_routing.inf_timer_enabled && clock() - dv_routing.inf_timer_start >= INF_TIMER_THRES) {
    dv_routing.stop_timer();
}
```

在 main.cpp 中检查计时器

2.3.5 抑制计时器原理

理论上讲，如果我们可以准确估算出一条消息传遍整个网络所需时间，抑制计时器可以完全避免无穷计数问题。但是，实际中并非如此。因为 1）实际网络环境的延迟无法准确预测（如果不同节点启动计时器的时差太大，那么计时器会失去它的作用），2）已经发送但还未被接受的过期路由表也可能会导致无穷计数问题（比如当多个节点在短时间内同时下线时，不仅网络会变得拥塞，而且会导致很多过期的路由表被发出并接受。一种可能的解决办法是给每一种“毒药”都配备一个单独的计时器，但这会导致代码过于复杂，且因为还有 1）的限制，得不偿失）。但是无论如何，这个机制的出现不会加重无穷计数问题。哪怕出现了无穷计数问题，这个机制还有一定的概率被触发并终止它——这是一个纯粹的运气问题（见 2.5.6）。

2.3.6 阈值上限

为了解决网络拥塞导致的无穷计数，我们引入保险措施——阈值上限。当到达某个节点的距离超过这个上线，则将其视为无穷。这里，我们选择整个网络中所有路径的和作为上限，在初始化时计算得出。

2.3.7 初始化路由表

有了上下线机制，我们可以初始化节点的路由表全部为 INF，在节点上线后再进行更新。

```cpp
int Network_Info::init_routing_table(){
    // to initialize distance table

    map<string, int16_t>::iterator iter;
    iter = neigh_cost.begin();
    while (iter != neigh_cost.end()){
        auto pr = std::make_pair(iter->first, std::numeric_limits<int16_t>::max());
        auto table_pr = std::make_pair(iter->first, pr);
        auto ret_table_pr = distance_table.insert(table_pr);
        iter++;
    }

    return 1;
}
```

2.3.8 VERBOSE 机制

在 dv_algorithm.cpp 中，如果宏定义了 VERBOSE，会在接受到 Ping 和泛洪路由表后打印消息，方便进行观察程序运行过程。

2.3.9 路由表消息

按照真实的距离矢量路由算法，笔者在新建消息时，加入了到自己的距离为 0 的路由表项，这样可以保证一个节点能够在它的邻居下线后重新找到正确的路径。比如，当节点 D 下线后，如果没有这条消息，E 永远不能发现它有一条距离为 32 的路径直接到达它的邻居 B。

```cpp
int DV_Routing::construct_dv_msg(Network_Info * p_net_info, int msg_type, DV_Msg *p_dv_msg) {
    switch(msg_type) {
        case PING_MSG:
            // to construct ping message
            p_dv_msg->insert_from_node_id(PING_MSG, p_net_info->node_id);
            break;
        case PATH_DISTANCE_MSG:
            Path_Cost path_cost;
            p_dv_msg->insert_from_node_id(PATH_DISTANCE_MSG, p_net_info->node_id);
            for (const auto &element : p_net_info->distance_table) {
                path_cost.to_node_id = element.first;
                path_cost.cost = element.second.second;
                p_dv_msg->insert_a_path_cost(&path_cost);
            }
            // add a path to myself with cost of 0
            path_cost.to_node_id = p_net_info->node_id;
            path_cost.cost = 0;
            p_dv_msg->insert_a_path_cost(&path_cost);
            break;
    }

}
```

2.4 算法设计过程（淘汰的写法可以在 git 提交历史中找到）

2.4.1 返回式 Ping

最初，参考真实情况，本人先考虑了在节点每次收到 ping 后返回一个 response 消息，如果发送 ping 的节点没有在指定时间内收到回复，则认为没有 ping 通。为了实现这种方法，本人有定义了一个 PING_RESP 消息类型，然后使用一个 std::map<std::string, volatile bool>来记录是否收到返回。但是经过许久的测试，这种方式始终拥有奇怪的 bug，以至于在每个周

期内，发 ping 方很难准确接收到接受方发送的返回消息，误以为其已经掉线，故淘汰。

2.4.2 返回式 Ping 改进

后来，本人以为是使用信号量 signal.h 导致数组访问冲突等问题，但是后来换成了 while 循环加计时器的写法，但是仍然无法解决问题。根据肉眼观察，考虑是因为发送消息过多导致电脑以太网缓冲区溢出导致的。毕竟，每个节点在每次更新路由表后就会泛洪，泛洪出去的每个路由表项又会再次引起泛洪。但是，即使将间隔拉到 20 秒，这个 bug 仍存在，遂淘汰。

```
int DV_Routing::construct_dv_msg(Network_Info * p_net_info, int msg_type, DV_Msg *p_dv_msg) {
    switch(msg_type) {
        case PING_MSG: {
            // to construct ping message
            p_dv_msg->insert_from_node_id(PING_MSG, p_net_info->node_id);
            break;
        }
        case PATH_DISTANCE_MSG: {
            Path_Cost path_cost;
            p_dv_msg->insert_from_node_id(PATH_DISTANCE_MSG, p_net_info->node_id);
            for (const auto &element : p_net_info->distance_table) {
                path_cost.to_node_id = element.first;
                path_cost.cost = element.second.second;
                p_dv_msg->insert_a_path_cost(&path_cost);
            }
            break;
        }
        case PING_RESP: {
            p_dv_msg->insert_from_node_id(PING_RESP, p_net_info->node_id);
            break;
        }
    }

}
```

```
case PING_RESP: {
    // mark response received
    net_info.neigh_resp[p_received_dv_msg->from_node_id] = true;
    break;
}
```

```
int DV_Routing::send_rping_msg(std::string to_node_id, TransportLayer *p_t_l, Network_Info *p_net_info) {
    std::cout << "Ping from " << to_node_id << " received. Sending rping..." << std::endl;
    // encode the ping response message
    std::string encoded_msg_str;
    DV_Msg dv_msg;
    dv_routing.construct_dv_msg(&net_info, PING_RESP, &dv_msg);
    dv_msg.encode(encoded_msg_str);

    auto peer_port_no = p_net_info->node_addr[to_node_id];

    p_t_l->send_msg(dest_IP_address, peer_port_no, (char *) encoded_msg_str.c_str(), encoded_msg_str.length());
    return 0;
}      You, 2 weeks ago • counter is about to be deleted
```

但是，哪怕降低了泛洪和 Ping 的频率，以太网拥塞的问题依旧存在，此程序不宜长时间运行。

2.4.3. 初始化路由表过期问题

初始化的过期路由表问题：根据目前代码，初始化后的路由表是自己到每一个邻居的距离，但是这并不一定准确（因为它的邻居可能还没上线）。为了避免将这些错误的距离信息

发送出去，设置了一个 deprecated 变量，假设一个节点初始化后的路由表是上次关机时残留的、已弃用的。在对所有邻居进行连接性检查并更新了路由表后，才将路由表设为合法的，可以发送出去的。deprecated 变量本质上是一个计数器，用于记录是否完成了至少一次邻居连接性检查。最终这个机制因为不必要的复杂被淘汰了。

```
static void sigalrmHandler(int sig) { // flood ping message at every interval
    gotAlarm = 1;
    // check if respose are collected from last ping flood
    dv_routing.check_ping_resp(p_t_l, &net_info);
    if (deprecated < 2) deprecated++;

    // construct and flood ping message
    DV_Msg dv_msg;
    dv_routing.construct_dv_msg(&net_info, PING_MSG, &dv_msg);
    dv_routing.flood_ping_dv_msg(p_t_l, &net_info, &dv_msg);
}
```

2.4.4 其它

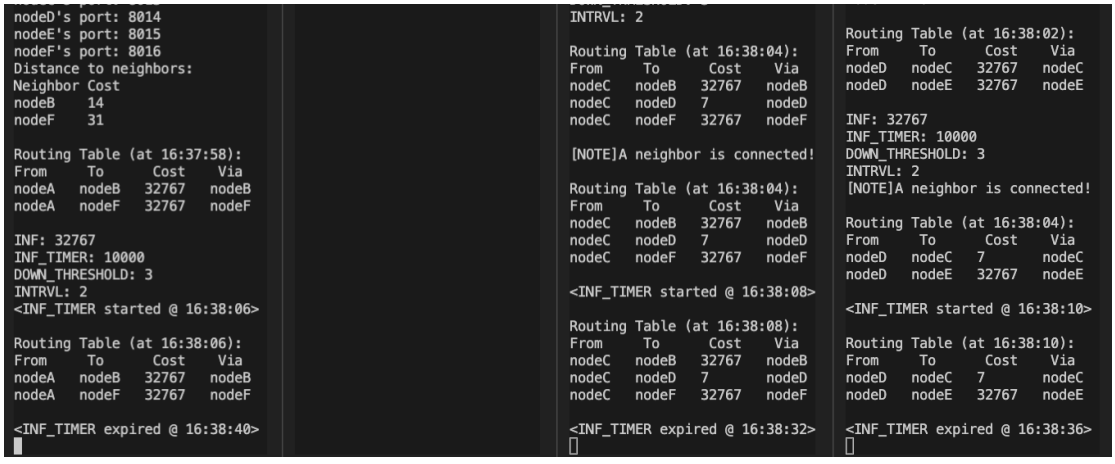
笔者还对代码进行了其它更改，详见代码注释和 git 提交记录。

2.5 运行过程

如无说明，截图中从左到右分别为 A~F 节点。为了方便观察路由表的变化，在大部分的截图中都没有启动 VERBOSE 模式。

2.5.1 启动节点 ACD

三者都很快达到了稳态。因为 BEF 仍没上线，因此无穷距离表项还触发了邻居的抑制计时器。

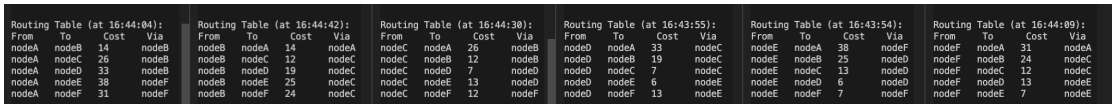


2.5.2 接着启动 BF

最下面的路由表为稳定状态的路由表，可见每一个节点都找到了当前状态下的最短路径，但是用时各不相同（观察路由表打印时间）。

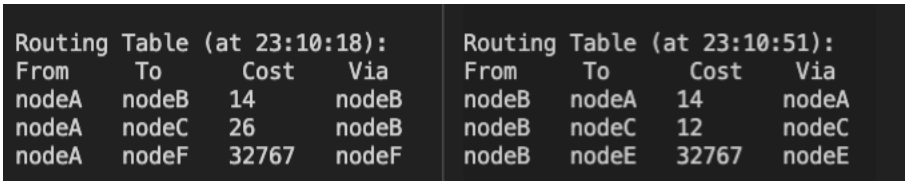
2.5.3 最后启动 E

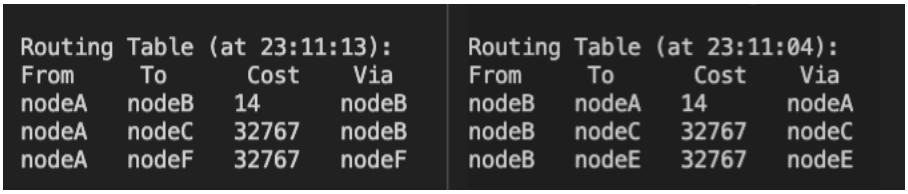
类似上图，可以大致观察到每个节点寻找最短路径并对 E 的上线做出相应的时间。

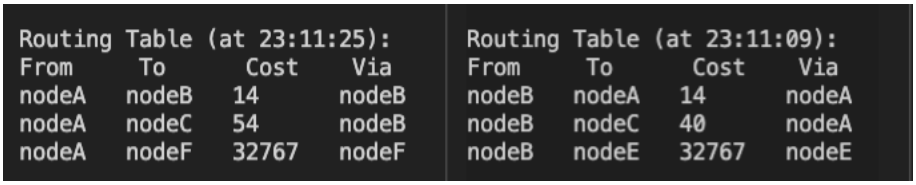


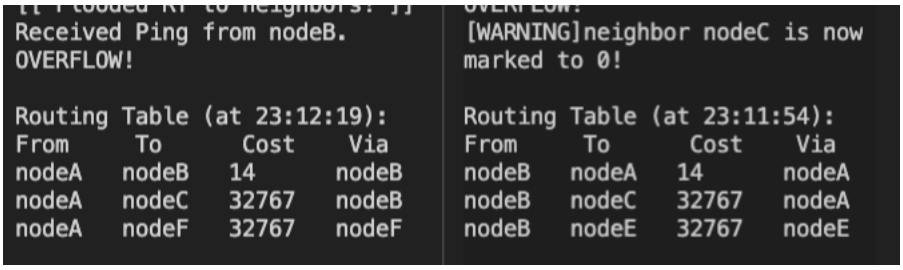
2.5.4 小范围观察无穷计数问题（ABC 上线，然后 C 下线）：

此时没有抑制计时器机制。为了方便观察路由表变化，以下截图省略了接收到 Ping 时打印的消息，并且，处在同一行的路由表并无时间上的关联（注意观察时间差距的增大），只是为了截图方便。

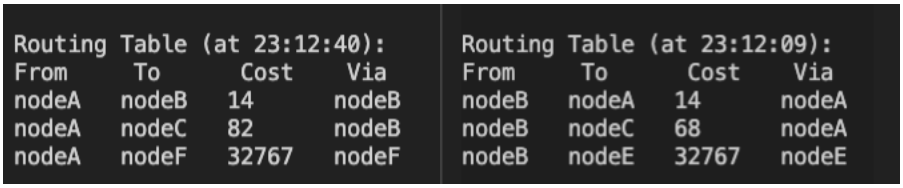


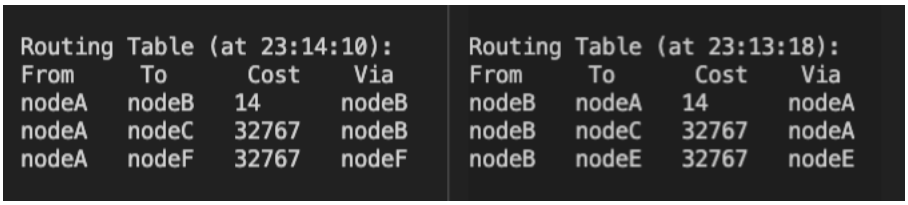






可以看到上图中 B 节点出现了[WARNING]消息，这种消息理应在真实世界中的网络延迟情况下才会出现（详见代码注释），但是以太网口的拥塞实在是太严重了，所以在本地模拟时也出现了。



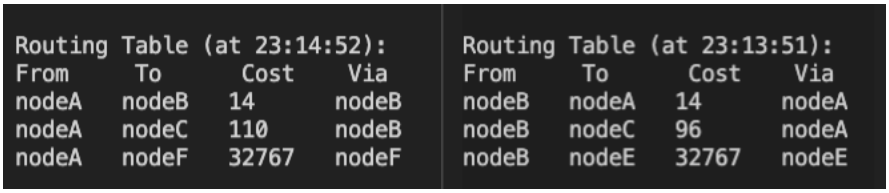


Routing Table (at 23:14:10):

From	To	Cost	Via
nodeA	nodeB	14	nodeB
nodeA	nodeC	32767	nodeB
nodeA	nodeF	32767	nodeF

Routing Table (at 23:13:18):

From	To	Cost	Via
nodeB	nodeA	14	nodeA
nodeB	nodeC	32767	nodeA
nodeB	nodeE	32767	nodeE

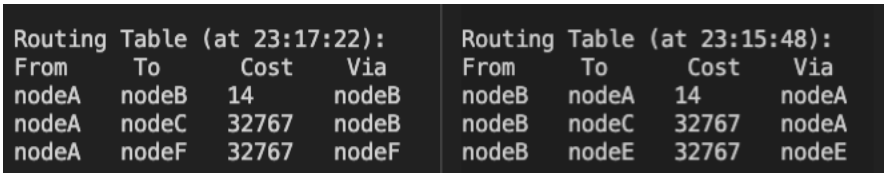


Routing Table (at 23:14:52):

From	To	Cost	Via
nodeA	nodeB	14	nodeB
nodeA	nodeC	110	nodeB
nodeA	nodeF	32767	nodeF

Routing Table (at 23:13:51):

From	To	Cost	Via
nodeB	nodeA	14	nodeA
nodeB	nodeC	96	nodeA
nodeB	nodeE	32767	nodeE

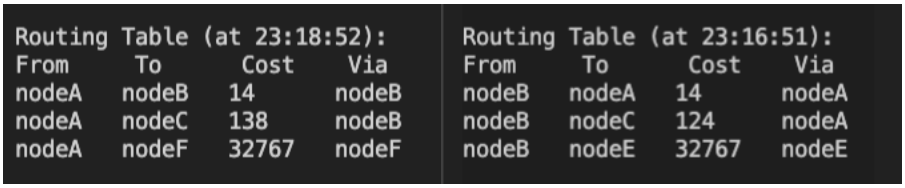


Routing Table (at 23:17:22):

From	To	Cost	Via
nodeA	nodeB	14	nodeB
nodeA	nodeC	32767	nodeB
nodeA	nodeF	32767	nodeF

Routing Table (at 23:15:48):

From	To	Cost	Via
nodeB	nodeA	14	nodeA
nodeB	nodeC	32767	nodeA
nodeB	nodeE	32767	nodeE



Routing Table (at 23:18:52):

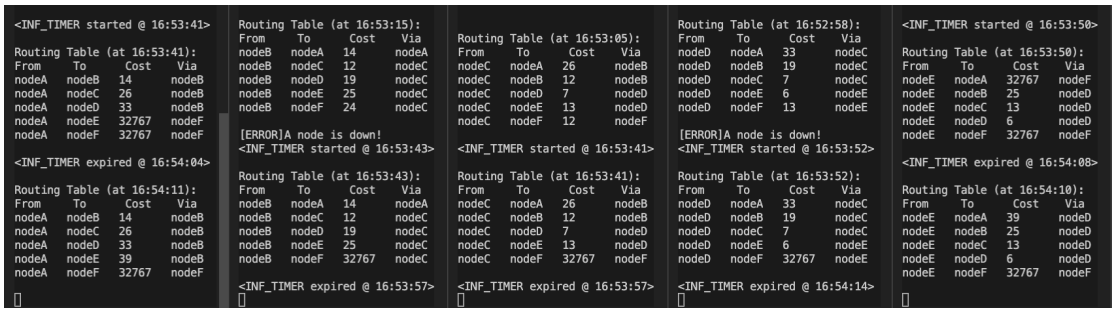
From	To	Cost	Via
nodeA	nodeB	14	nodeB
nodeA	nodeC	138	nodeB
nodeA	nodeF	32767	nodeF

Routing Table (at 23:16:51):

From	To	Cost	Via
nodeB	nodeA	14	nodeA
nodeB	nodeC	124	nodeA
nodeB	nodeE	32767	nodeE

2.5.5 使用抑制计时器避免无穷计数问题（截图为节点 A~E）

这里以节点 F 下线为例。如图，关闭 F 后，ACE 依次启动了抑制计时器，并且把 INF 消息泛洪。节点 BD 收到消息后也依次将 F 标记为不可达，并启动了计时器。在所有节点抑制计时器结束后，BCD 达到稳态，AE 又重新找到了连接到对方的最短路线。整个过程中没有出现无穷计数问题。



<INF_TIMER started @ 16:53:41>

Routing Table (at 16:53:41):

From	To	Cost	Via
nodeA	nodeB	14	nodeB
nodeA	nodeC	26	nodeB
nodeA	nodeD	33	nodeB
nodeA	nodeE	32767	nodeF
nodeA	nodeF	32767	nodeF

<INF_TIMER expired @ 16:54:04>

Routing Table (at 16:54:11):

From	To	Cost	Via
nodeA	nodeB	14	nodeB
nodeA	nodeC	26	nodeB
nodeA	nodeD	33	nodeB
nodeA	nodeE	39	nodeB
nodeA	nodeF	32767	nodeF

Routing Table (at 16:53:15):

From	To	Cost	Via
nodeB	nodeA	14	nodeA
nodeB	nodeC	12	nodeC
nodeB	nodeD	19	nodeC
nodeB	nodeE	25	nodeC
nodeB	nodeF	24	nodeC

[ERROR]A node is down!
<INF_TIMER started @ 16:53:43>

Routing Table (at 16:53:43):

From	To	Cost	Via
nodeB	nodeA	14	nodeA
nodeB	nodeC	12	nodeC
nodeB	nodeD	19	nodeC
nodeB	nodeE	25	nodeC
nodeB	nodeF	32767	nodeC

<INF_TIMER expired @ 16:53:57>

Routing Table (at 16:53:05):

From	To	Cost	Via
nodeC	nodeA	26	nodeB
nodeC	nodeB	12	nodeB
nodeC	nodeD	7	nodeD
nodeC	nodeE	13	nodeD
nodeC	nodeF	12	nodeF

<INF_TIMER started @ 16:53:41>

Routing Table (at 16:53:41):

From	To	Cost	Via
nodeC	nodeA	26	nodeB
nodeC	nodeB	12	nodeB
nodeC	nodeD	7	nodeD
nodeC	nodeE	13	nodeD
nodeC	nodeF	32767	nodeF

<INF_TIMER expired @ 16:53:57>

Routing Table (at 16:52:58):

From	To	Cost	Via
nodeD	nodeA	33	nodeC
nodeD	nodeB	19	nodeC
nodeD	nodeC	7	nodeC
nodeD	nodeE	6	nodeE
nodeD	nodeF	13	nodeE

[ERROR]A node is down!
<INF_TIMER started @ 16:53:52>

Routing Table (at 16:53:52):

From	To	Cost	Via
nodeD	nodeA	33	nodeC
nodeD	nodeB	19	nodeC
nodeD	nodeC	7	nodeC
nodeD	nodeE	6	nodeE
nodeD	nodeF	32767	nodeE

<INF_TIMER expired @ 16:54:14>

<INF_TIMER started @ 16:53:50>

Routing Table (at 16:53:50):

From	To	Cost	Via
nodeE	nodeA	32767	nodeF
nodeE	nodeB	25	nodeD
nodeE	nodeC	13	nodeD
nodeE	nodeD	6	nodeD
nodeE	nodeF	32767	nodeF

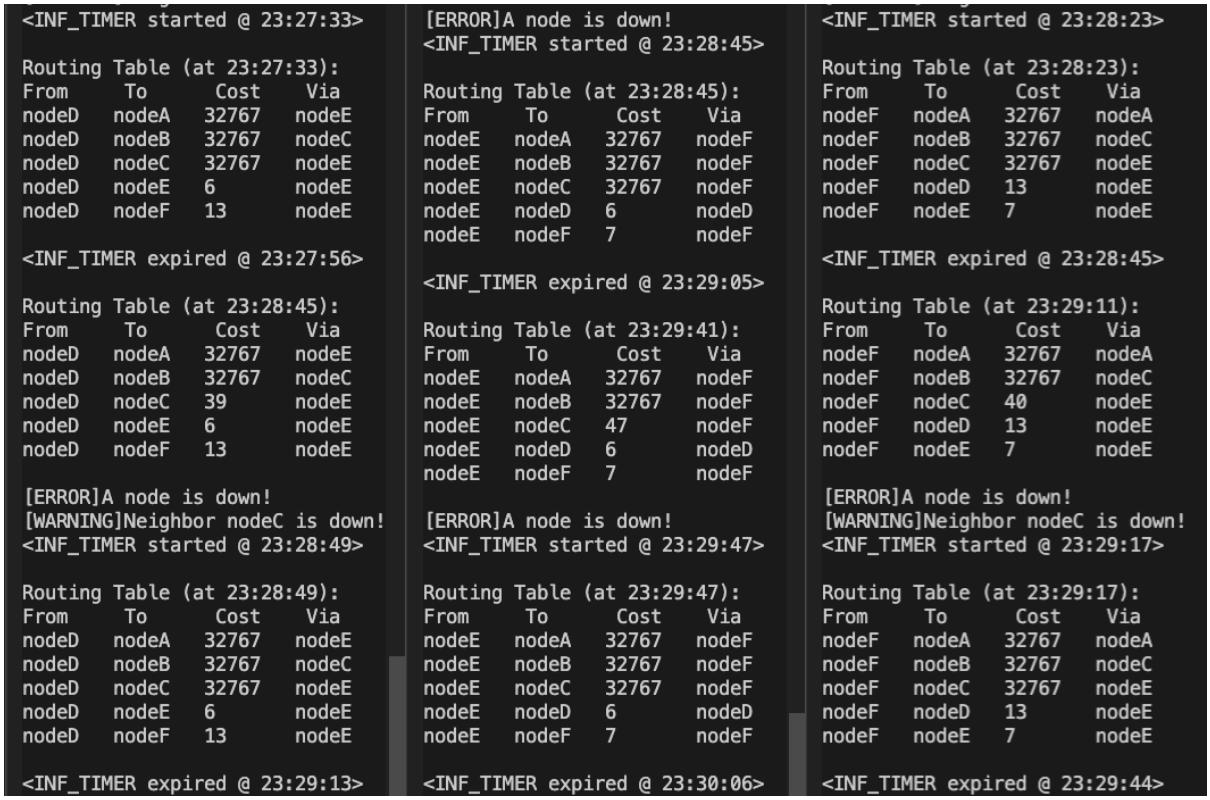
<INF_TIMER expired @ 16:54:08>

Routing Table (at 16:54:10):

From	To	Cost	Via
nodeE	nodeA	39	nodeD
nodeE	nodeB	25	nodeD
nodeE	nodeC	13	nodeD
nodeE	nodeD	6	nodeD
nodeE	nodeF	32767	nodeF

2.5.6 使用抑制计时器终止无穷计数问题

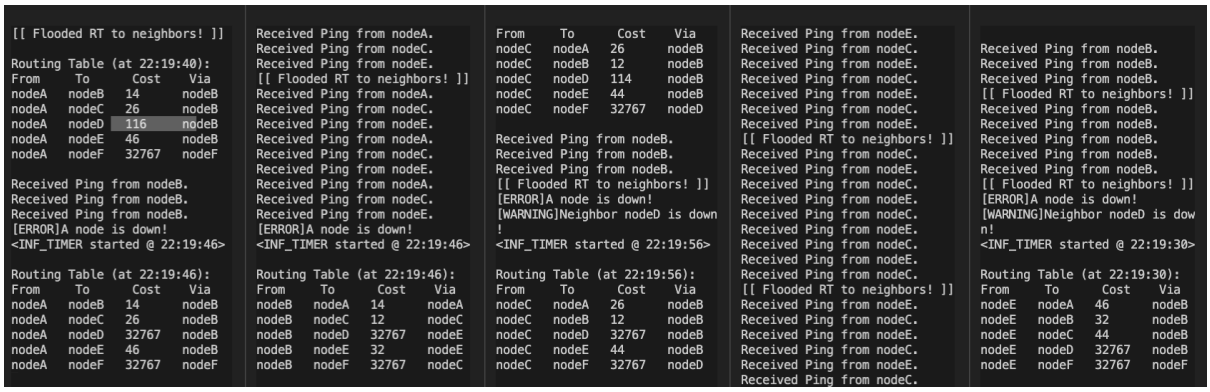
接下来，我们通过观察节点 ABC 在短时间内依次下线后节点 DEF 的反应来解释抑制计时器为何无法避免无穷计数问题，但又能巧妙地将其终止。



可以看到，在 ABC 断掉后，计时器帮助 DEF 避免了到 A 和 B 的路径的无穷计数问题。但是，由于消息延迟、计时器不同步（即三个节点计时器开启、关闭的时间不同）等原因，到 C 的路径还是出现了无穷计数问题（注意第二行的三个路由表）。他们都分别是在自己的上一个计时器过期后，接收到了过期路由信息导致的。但是，好在它们都及时发现了路由 C 不可达（F 通过没有收到 ping 消息发现，DE 通过过期的“C 不可达”消息发现），然后启动了新一轮的计时器（第三行的路由表上面的 INF_TIMER 消息）。在这个计时器周期内，它们处理掉了所有残留的“C 可达”的垃圾消息，并且也没有发出新的“C 可达”垃圾消息，因此很幸运地就此终止了无穷计数问题。这与计时器启动的时刻和延续的时长都有关系，所以说是“幸运的”。

2.5.7 使用阈值上限终止无穷计数问题

在最后的展示中，我们启用 VERBOSE 模式。可以看到 F、D 下线后，ABCE 经历了无穷计数（注意第一列第一个路由表中到 D 的路径涨到了 116）。但由于阈值为 121，最终无穷计数还是被终结了。



阈值法终结无穷计数问题通常具有很高的延迟。比如，如果此时我们再下线 E，经过了无穷计数，ABC 才达到稳态，但是 C 比 AB 晚了一分钟。

```
<INF_TIMER started @ 22:30:38>

Routing Table (at 22:30:38):
From      To       Cost     Via
nodeA     nodeB    14       nodeB
nodeA     nodeC    26       nodeB
nodeA     nodeD    32767    nodeB
nodeA     nodeE    32767    nodeB
nodeA     nodeF    32767    nodeF
```

```
<INF_TIMER started @ 22:30:35>

Routing Table (at 22:30:35):
From      To       Cost     Via
nodeB     nodeA    14       nodeA
nodeB     nodeC    12       nodeC
nodeB     nodeD    32767    nodeE
nodeB     nodeE    32767    nodeC
nodeB     nodeF    32767    nodeC
```

```
<INF_TIMER started @ 22:31:36>

Routing Table (at 22:31:36):
From      To       Cost     Via
nodeC     nodeA    26       nodeB
nodeC     nodeB    12       nodeB
nodeC     nodeD    32767    nodeB
nodeC     nodeE    32767    nodeB
nodeC     nodeF    32767    nodeD
```

2.5.8 节点上线

接着 2.5.7，我们将 DEF 重新上线，经历了好一会，全部节点才重新达到稳态（注意观察时间）。新上线的节点往往能快速达到稳态（例如 F），而离新上线的节点较远的节点需要花费很长的时间才能确定最短路径（例如 A）。