实验四 运用ns-3分析路由协议

专业：计算机科学与技术

学号：22920192204097

姓名：吴雨娟

一、实验目的

学习运用ns-3进行网络仿真的基本方法与流程；

了解ns-3中路由协议实现框架；

运用ns-3构造网络拓扑分析网络数据，分析掌握RIP路由协议的工作原理与过程。

二、实验内容

任务1：IPv6报文观察（已完成）

使用ping命令，捕获ipv6数据包，比较分析IPv6与IPv4报文头变化。

任务2：ns-3的基本使用

2.1 在教程示列first.cc中，使用运行参数传入DataRate和Delay值，观察并解析数据报文收发时间变化；修改LOG级别，观察输出变化；生成pcap文件，记录结点发出的报文。

2.2 在教程示例third.cc中，加个回调函数，输出观察最后一个移动结点的位置变化。

任务3：构建网络拓扑

3.1 建立网络拓扑， 配置网络结点与应用

3.2 设置仿真事件时序、跟踪记录仿真数据

任务4：结合RIP工作原理，分析其工作过程

4.1 从任务3中抓取的数据，分析路由收敛过程。

4.2 构造 “坏消息”，观察路由收敛过程，以及期间的丢包现象。

三、实验结果和分析

1. 任务1：IPv6报文观察（已完成）

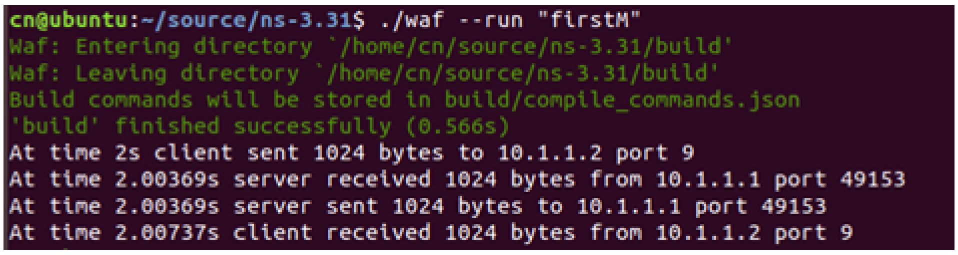
2.任务2：ns-3的基本使用

2.1在教程示列first.cc中，使用运行参数传入DataRate、Delay和MaxPacket值，观察并解析数据报文收发时间变化； 修改LOG级别，观察输出变化；生成pcap文件，记录结点发出的报文。

（1）使用运行参数传入DataRate、Delay和MaxPacket值，观察并解析数据报文收发时间变化；

①未做任何修改，运行firstM.cc文件（此时DataRate= 5Mbps，Delay=2ms，MaxPackets=1）

结果如下：



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "firstM"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.566s)
At time 2s client sent 1024 bytes to 10.1.1.2 port 9
At time 2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time 2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time 2.00737s client received 1024 bytes from 10.1.1.2 port 9
```

②使用运行参数传入DataRate，观察并解析数据报文收发时间变化

修改firstM.cc文件如下：

声明一个命令行类的对象，然后调用Parse

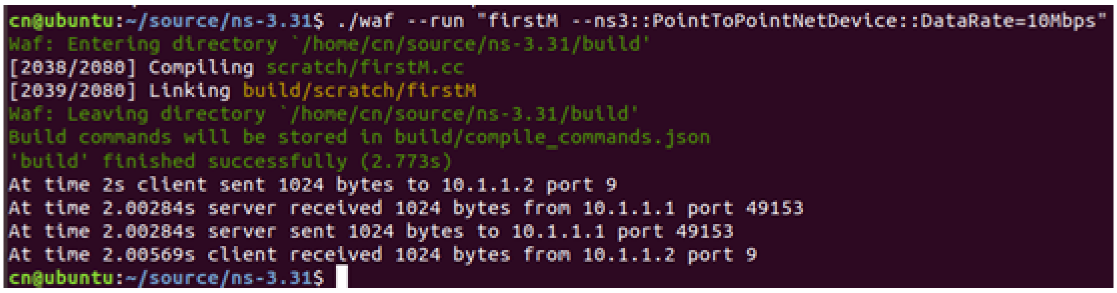
```
//CommandLine cmd (__FILE__);
  CommandLine cmd;
  cmd.Parse (argc, argv);
```

把代码中原有的对DataRate的赋值语句注释掉

```
//pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
```

在命令行中修改DataRate= 10Mbps，并运行

结果如下：



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "firstM --ns3::PointToPointNetDevice::DataRate=10Mbps"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
[2038/2080] Compiling scratch/firstM.cc
[2039/2080] Linking build/scratch/firstM
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (2.773s)
At time 2s client sent 1024 bytes to 10.1.1.2 port 9
At time 2.00284s server received 1024 bytes from 10.1.1.1 port 49153
At time 2.00284s server sent 1024 bytes to 10.1.1.1 port 49153
At time 2.00569s client received 1024 bytes from 10.1.1.2 port 9
cn@ubuntu:~/source/ns-3.31$
```

对比①和②中修改DataRate参数前后的两张结果图可知：由于把数据的发送速率从5Mbit/s增大到了10Mbit/s，所以服务器收到数据的事件相应地提前了一段时间，客户机收到数据的时间节点也前移了。

③使用运行参数传入Delay，观察并解析数据报文收发时间变化

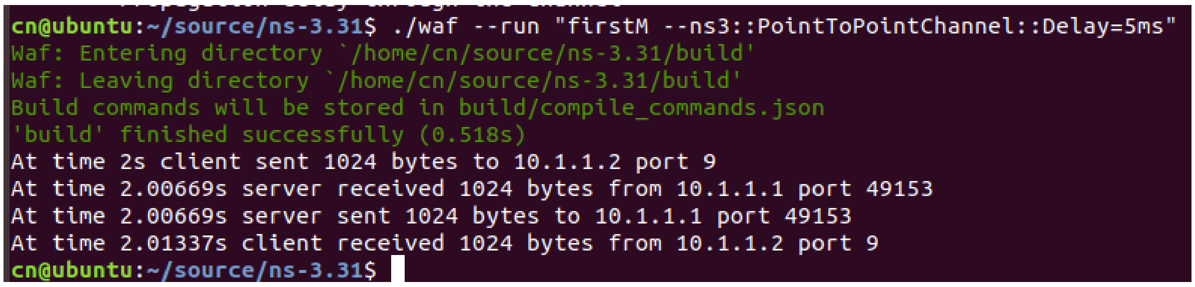
修改firstM.cc文件如下：

把代码中原有的对Delay的赋值语句注释掉

```
//pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

在命令行中修改Delay= 5ms，并运行

结果如下：



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "firstM --ns3::PointToPointChannel::Delay=5ms"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.518s)
At time 2s client sent 1024 bytes to 10.1.1.2 port 9
At time 2.00669s server received 1024 bytes from 10.1.1.1 port 49153
At time 2.00669s server sent 1024 bytes to 10.1.1.1 port 49153
At time 2.01337s client received 1024 bytes from 10.1.1.2 port 9
cn@ubuntu:~/source/ns-3.31$
```

对比①和③中修改Delay参数前后的两张结果图可知：由于把数据的延迟从2ms增大到了5ms，所以服务器收到数据的事件相应地推迟了一段时间，客户机收到数据的时间节点也后移了。

④使用运行参数传入MaxPackets，观察并解析数据报文收发时间变化

修改firstM.cc文件如下：

新增变量nPackets，默认值为1，可通过命令行修改变量值

```
  uint32_t nPackets = 1;
  //CommandLine cmd (__FILE__);
  CommandLine cmd;
  cmd.AddValue("nPackets", "Number of packets to echo", nPackets);
  cmd.Parse (argc, argv);
```

修改对MaxPackets的赋值语句

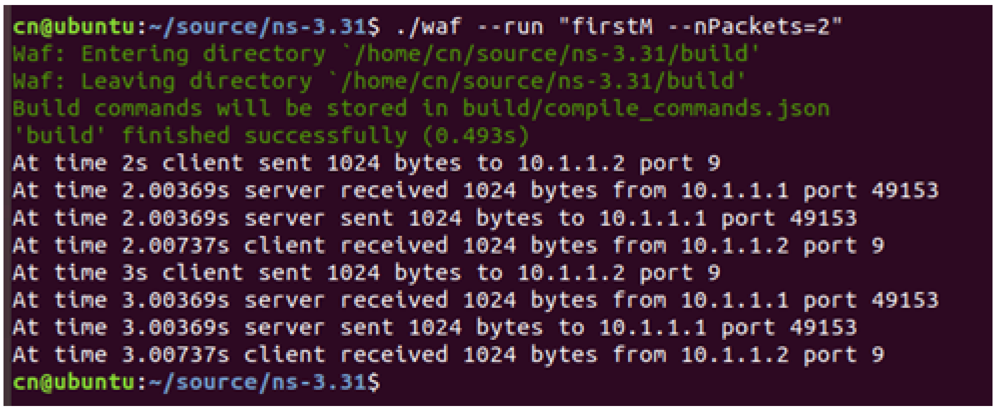
```
  //echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ( "MaxPackets",UintegerValue(nPackets));
```

并且把DataRate和Delay修改回原本设定的值

```
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

在命令行设置发送两次分组，即令nPackets=2，运行

结果如下：



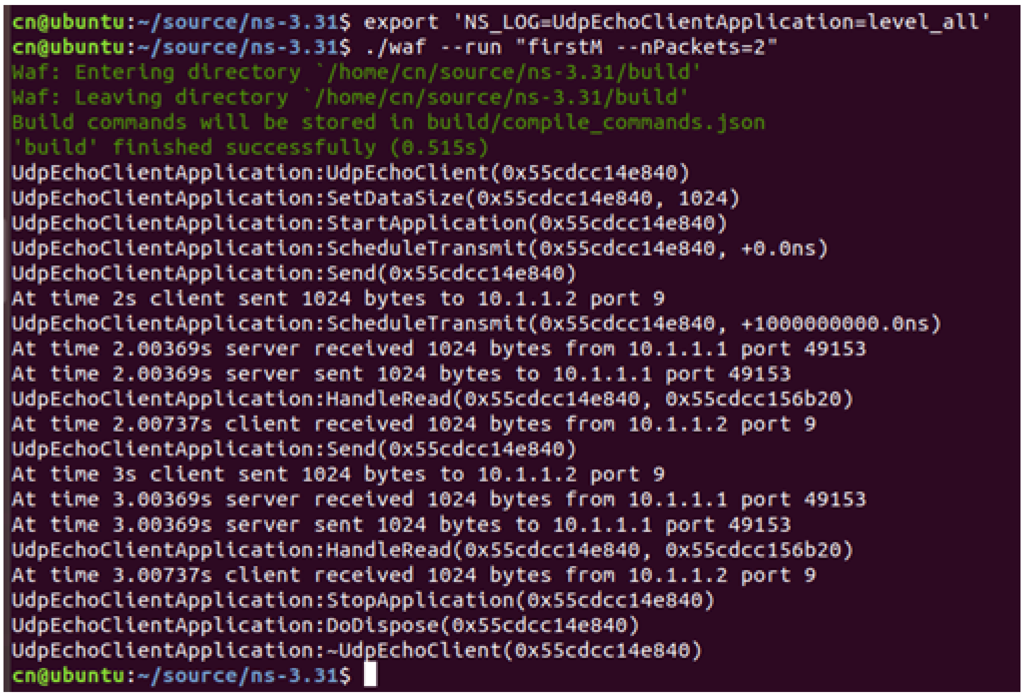
```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "firstM --nPackets=2"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.493s)
At time 2s client sent 1024 bytes to 10.1.1.2 port 9
At time 2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time 2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time 2.00737s client received 1024 bytes from 10.1.1.2 port 9
At time 3s client sent 1024 bytes to 10.1.1.2 port 9
At time 3.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time 3.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time 3.00737s client received 1024 bytes from 10.1.1.2 port 9
cn@ubuntu:~/source/ns-3.31$
```

对比①和④中修改MaxPackets参数前后的两张结果图可知：每次发送的分组数由1变成了2，修改后客户端发送了两次分组，而之前只发送了一次分组。

（2）修改LOG级别，观察输出变化

命令行中输入命令export 'NS_LOG=UdpEchoClientApplication=level_all’

再运行firstM.cc，结果如下：



```
cn@ubuntu:~/source/ns-3.31$ export 'NS_LOG=UdpEchoClientApplication=level_all'
cn@ubuntu:~/source/ns-3.31$ ./waf --run "firstM --nPackets=2"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.515s)
UdpEchoClientApplication:UdpEchoClient(0x55cdcc14e840)
UdpEchoClientApplication:SetDataSize(0x55cdcc14e840, 1024)
UdpEchoClientApplication:StartApplication(0x55cdcc14e840)
UdpEchoClientApplication:ScheduleTransmit(0x55cdcc14e840, +0.0ns)
UdpEchoClientApplication:Send(0x55cdcc14e840)
At time 2s client sent 1024 bytes to 10.1.1.2 port 9
UdpEchoClientApplication:ScheduleTransmit(0x55cdcc14e840, +1000000000.0ns)
At time 2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time 2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
UdpEchoClientApplication:HandleRead(0x55cdcc14e840, 0x55cdcc156b20)
At time 2.00737s client received 1024 bytes from 10.1.1.2 port 9
UdpEchoClientApplication:Send(0x55cdcc14e840)
At time 3s client sent 1024 bytes to 10.1.1.2 port 9
At time 3.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time 3.00369s server sent 1024 bytes to 10.1.1.1 port 49153
UdpEchoClientApplication:HandleRead(0x55cdcc14e840, 0x55cdcc156b20)
At time 3.00737s client received 1024 bytes from 10.1.1.2 port 9
UdpEchoClientApplication:StopApplication(0x55cdcc14e840)
UdpEchoClientApplication:DoDispose(0x55cdcc14e840)
UdpEchoClientApplication:~UdpEchoClient(0x55cdcc14e840)
cn@ubuntu:~/source/ns-3.31$
```

对比修改LOG级别前后两张结果图可知：输出的信息比之前增加了。这些额外的调试信息是来自NS_LOG_FUNTION级别的日志。这些信息显示了在脚本运行期间程序中每个函数调用过程。

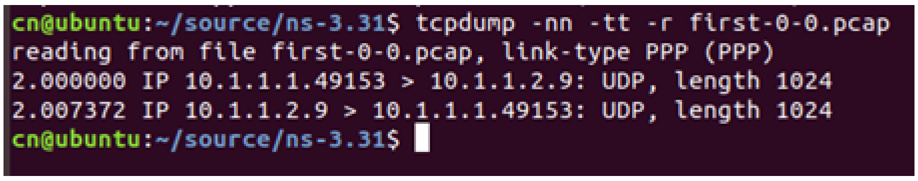
（3）生成pcap文件，记录结点发出的报文。

修改firstM.cc文件如下：

在运行前添加一行代码，生成pcap文件

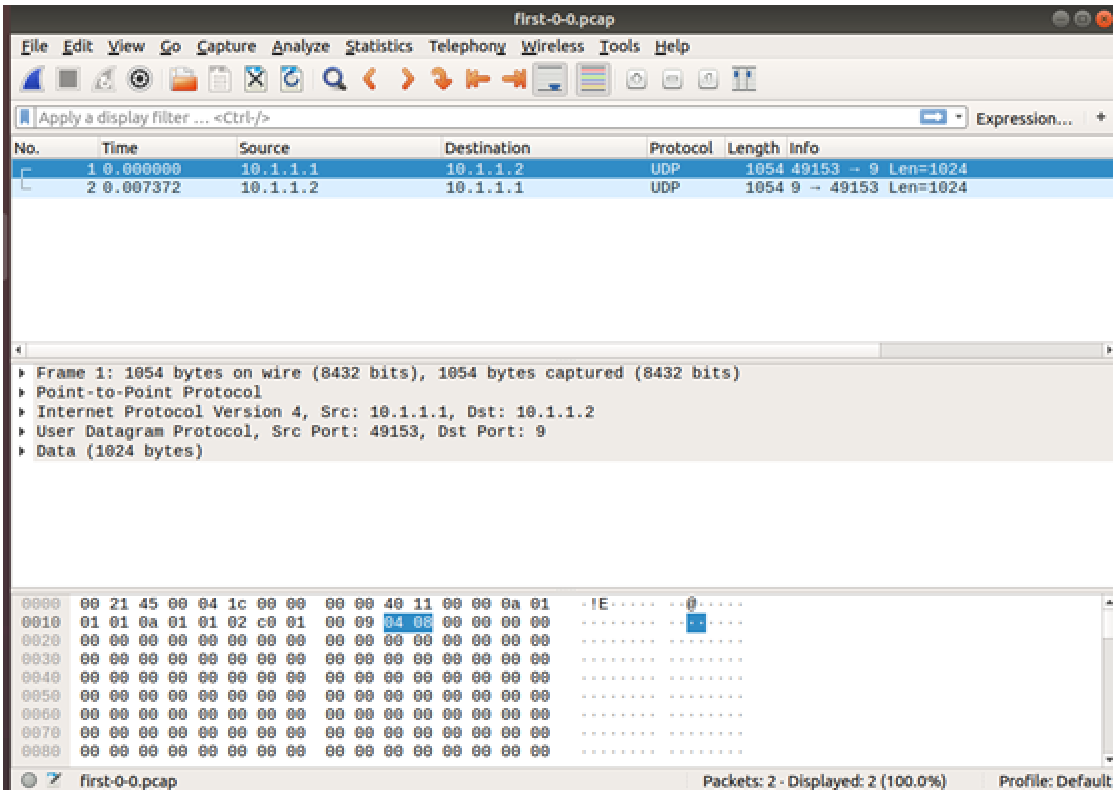
```
pointToPoint.EnablePcapAll ("first");

Simulator::Run ();
Simulator::Destroy ();
```

编译后在终端查看pcap文件：



```
cn@ubuntu:~/source/ns-3.31$ tcpdump -nn -tt -r first-0-0.pcap
reading from file first-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 1024
2.007372 IP 10.1.1.2.9 > 10.1.1.1.49153: UDP, length 1024
cn@ubuntu:~/source/ns-3.31$
```

通过wireshark打开：


first-0-0.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
Expression…

```
No.  Time       Source     Destination  Protocol  Length  Info
   1 0.000000   10.1.1.1   10.1.1.2     UDP       1054 49153 → 9  Len=1024
   2 0.007372   10.1.1.2   10.1.1.1     UDP       1054 9 → 49153  Len=1024
```

```
▸ Frame 1: 1054 bytes on wire (8432 bits), 1054 bytes captured (8432 bits)
▸ Point-to-Point Protocol
▸ Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.1.1.2
▸ User Datagram Protocol, Src Port: 49153, Dst Port: 9
▸ Data (1024 bytes)
```

```
0000  00 21 45 00 04 1c 00 00  00 00 40 11 00 00 0a 01   ·!E·····  ··@·····
0010  01 01 0a 01 01 02 c0 01  00 09 04 08 00 00 00 00   ········  ········
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
```

first-0-0.pcap   Packets: 2 · Displayed: 2 (100.0%)   Profile: Default

在wireshark中可以看到pcap文件中的内容，即UDP报文。

2.2在教程示例third.cc中，加个回调函数，输出观察最后一个移动结点的位置变化。

（1）在教程示例third.cc中，加个回调函数，修改代码如下（具体代码见thirdM.cc）

定义一个回调函数如下

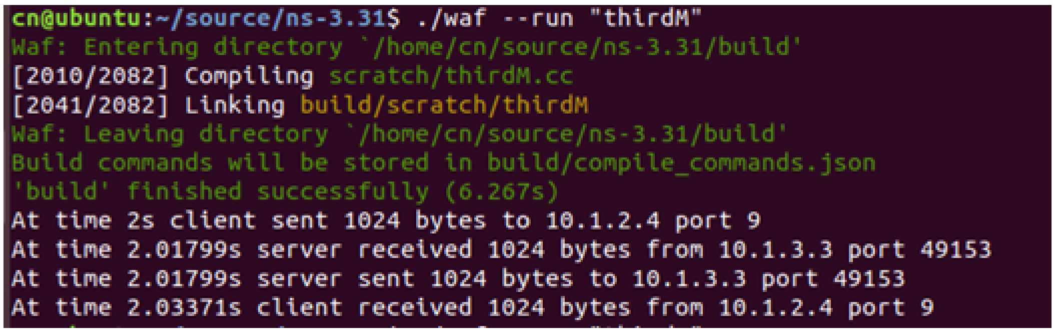
```
/*设置把这个函数与位置改变回调联系起来*/
void
CourseChange (std::string context, Ptr<const MobilityModel> model)
{
  Vector position = model->GetPosition ();
  NS_LOG_UNCOND (context <<
    " x = " << position.x << ", y = " << position.y<<" time:"
<<Simulator::Now().GetSeconds());
}
```

调用回调函数

```
/*为最后一个wifi节点绑定回调函数*/
  std::ostringstream oss;
  oss << "/NodeList/"
      << wifiStaNodes.Get (nWifi - 1)->GetId ()
      << "/$ns3::MobilityModel/CourseChange";

  Config::Connect (oss.str (), MakeCallback (&CourseChange));
```

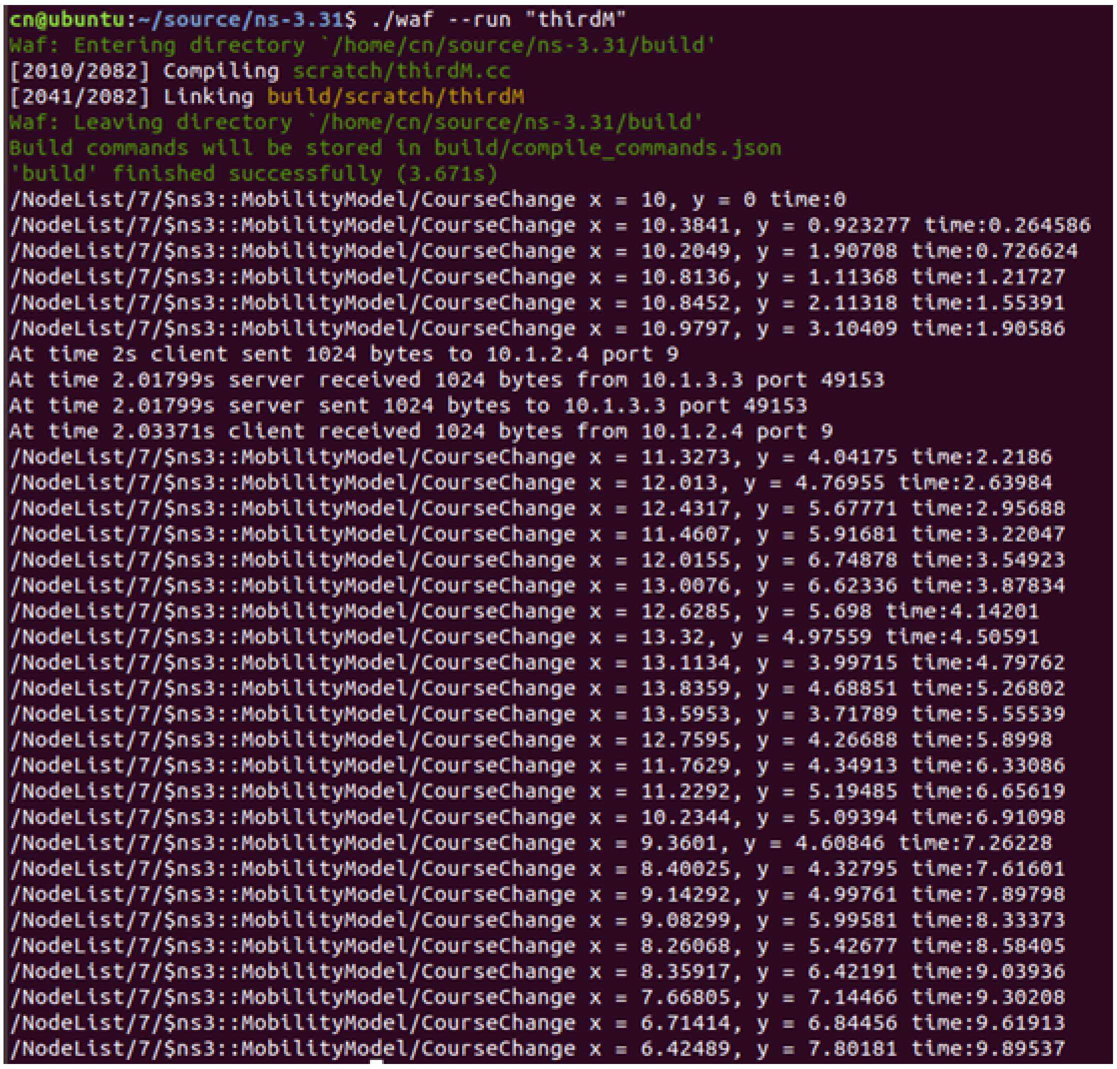
（2）运行，输出观察最后一个移动结点的位置变化

修改前，运行原thirdM.cc的输出：



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "thirdM"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
[2010/2082] Compiling scratch/thirdM.cc
[2041/2082] Linking build/scratch/thirdM
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (6.267s)
At time 2s client sent 1024 bytes to 10.1.2.4 port 9
At time 2.01799s server received 1024 bytes from 10.1.3.3 port 49153
At time 2.01799s server sent 1024 bytes to 10.1.3.3 port 49153
At time 2.03371s client received 1024 bytes from 10.1.2.4 port 9
```

修改后，运行thirdM.cc的输出：



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "thirdM"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
[2010/2082] Compiling scratch/thirdM.cc
[2041/2082] Linking build/scratch/thirdM
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (3.671s)
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10, y = 0 time:0
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.3841, y = 0.923277 time:0.264586
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.2049, y = 1.90708 time:0.726624
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.8136, y = 1.11368 time:1.21727
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.8452, y = 2.11318 time:1.55391
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.9797, y = 3.10409 time:1.90586
At time 2s client sent 1024 bytes to 10.1.2.4 port 9
At time 2.01799s server received 1024 bytes from 10.1.3.3 port 49153
At time 2.01799s server sent 1024 bytes to 10.1.3.3 port 49153
At time 2.03371s client received 1024 bytes from 10.1.2.4 port 9
/NodeList/7/$ns3::MobilityModel/CourseChange x = 11.3273, y = 4.04175 time:2.2186
/NodeList/7/$ns3::MobilityModel/CourseChange x = 12.013, y = 4.76955 time:2.63984
/NodeList/7/$ns3::MobilityModel/CourseChange x = 12.4317, y = 5.67771 time:2.95688
/NodeList/7/$ns3::MobilityModel/CourseChange x = 11.4607, y = 5.91681 time:3.22047
/NodeList/7/$ns3::MobilityModel/CourseChange x = 12.0155, y = 6.74878 time:3.54923
/NodeList/7/$ns3::MobilityModel/CourseChange x = 13.0076, y = 6.62336 time:3.87834
/NodeList/7/$ns3::MobilityModel/CourseChange x = 12.6285, y = 5.698 time:4.14201
/NodeList/7/$ns3::MobilityModel/CourseChange x = 13.32, y = 4.97559 time:4.50591
/NodeList/7/$ns3::MobilityModel/CourseChange x = 13.1134, y = 3.99715 time:4.79762
/NodeList/7/$ns3::MobilityModel/CourseChange x = 13.8359, y = 4.68851 time:5.26802
/NodeList/7/$ns3::MobilityModel/CourseChange x = 13.5953, y = 3.71789 time:5.55539
/NodeList/7/$ns3::MobilityModel/CourseChange x = 12.7595, y = 4.26688 time:5.8998
/NodeList/7/$ns3::MobilityModel/CourseChange x = 11.7629, y = 4.34913 time:6.33086
/NodeList/7/$ns3::MobilityModel/CourseChange x = 11.2292, y = 5.19485 time:6.65619
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10.2344, y = 5.09394 time:6.91098
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.3601, y = 4.60846 time:7.26228
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.40025, y = 4.32795 time:7.61601
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.14292, y = 4.99761 time:7.89798
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.08299, y = 5.99581 time:8.33373
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.26068, y = 5.42677 time:8.58405
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.35917, y = 6.42191 time:9.03936
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.66805, y = 7.14466 time:9.30208
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.71414, y = 6.84456 time:9.61913
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.42489, y = 7.80181 time:9.89537
```

对比修改前和修改后的结果图可知：修改代码后，可以观察到移动结点的位置变化，具体变化由坐标值的变化来体现。

在thirdM.cc中加入为最后一个wifi节点绑定回调函数的代码，所以在输出中可以观察到最后一个wifi节点的位置变化。

3.任务3：构建网络拓扑

3.1 构建网络拓扑-1，应用Ipv4GlobalRouting，输出观察结点路由

（1）根据PPT给出的网络拓扑-1，编写代码global.cc实现全局路由。

具体代码见global.cc，以下只列出重要部分：

开启全局路由相应借口事件

```
Config::SetDefault("ns3::Ipv4GlobalRouting::RespondToInterfaceEvents",
BooleanValue(true));
```

创建结点

```
NS_LOG_INFO("Create nodes.");

    NodeContainer c;
    c.Create(9);
    NodeContainer n0n1 = NodeContainer(c.Get(0), c.Get(1));
    NodeContainer n1n2 = NodeContainer(c.Get(1), c.Get(2));
    NodeContainer n1n3 = NodeContainer(c.Get(1), c.Get(3));
    NodeContainer n2n4 = NodeContainer(c.Get(2), c.Get(4));
    NodeContainer n2n5 = NodeContainer(c.Get(2), c.Get(5));
    NodeContainer n3n4 = NodeContainer(c.Get(3), c.Get(4));
    NodeContainer n3n6 = NodeContainer(c.Get(3), c.Get(6));
    NodeContainer n4n5 = NodeContainer(c.Get(4), c.Get(5));
    NodeContainer n4n6 = NodeContainer(c.Get(4), c.Get(6));
    NodeContainer n6n7 = NodeContainer(c.Get(6), c.Get(7));
    NodeContainer n5n7 = NodeContainer(c.Get(5), c.Get(7));
    NodeContainer n7n8 = NodeContainer(c.Get(7), c.Get(8));

    InternetStackHelper internet;
    internet.Install(c);
```

创建信道

```
NS_LOG_INFO("Create channels.");

    PointToPointHelper p2p;

    p2p.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
    p2p.SetChannelAttribute("Delay", StringValue("2ms"));
    NetDeviceContainer d0d1 = p2p.Install(n0n1);
    NetDeviceContainer d1d2 = p2p.Install(n1n2);
    NetDeviceContainer d1d3 = p2p.Install(n1n3);
    NetDeviceContainer d2d4 = p2p.Install(n2n4);
    NetDeviceContainer d2d5 = p2p.Install(n2n5);
    NetDeviceContainer d3d4 = p2p.Install(n3n4);
    NetDeviceContainer d3d6 = p2p.Install(n3n6);
    NetDeviceContainer d4d5 = p2p.Install(n4n5);
    NetDeviceContainer d4d6 = p2p.Install(n4n6);
    NetDeviceContainer d6d7 = p2p.Install(n6n7);
    NetDeviceContainer d5d7 = p2p.Install(n5n7);
    NetDeviceContainer d7d8 = p2p.Install(n7n8);

    p2p.SetDeviceAttribute("DataRate", StringValue("1500kbps"));
    p2p.SetChannelAttribute("Delay", StringValue("10ms"));
```

添加ip地址

```
NS_LOG_INFO("Assign IP Addresses.");
    Ipv4AddressHelper ipv4;
    ipv4.SetBase("10.0.0.0", "255.255.255.0");
    Ipv4InterfaceContainer i0i1 = ipv4.Assign(d0d1);

    ipv4.SetBase("10.0.1.0", "255.255.255.0");
    ipv4.Assign(d1d2);

    ipv4.SetBase("10.0.2.0", "255.255.255.0");
    ipv4.Assign(d1d3);

    ipv4.SetBase("10.0.3.0", "255.255.255.0");
    ipv4.Assign(d2d4);

    ipv4.SetBase("10.0.4.0", "255.255.255.0");
    ipv4.Assign(d2d5);

    ipv4.SetBase("10.0.5.0", "255.255.255.0");
    ipv4.Assign(d3d4);

    ipv4.SetBase("10.0.6.0", "255.255.255.0");
    ipv4.Assign(d3d6);

    ipv4.SetBase("10.0.7.0", "255.255.255.0");
    ipv4.Assign(d4d5);

    ipv4.SetBase("10.0.8.0", "255.255.255.0");
    ipv4.Assign(d4d6);

    ipv4.SetBase("10.0.9.0", "255.255.255.0");
    ipv4.Assign(d6d7);

    ipv4.SetBase("10.0.10.0", "255.255.255.0");
    ipv4.Assign(d5d7);

    ipv4.SetBase("10.0.11.0", "255.255.255.0");
    ipv4.Assign(d7d8);
```

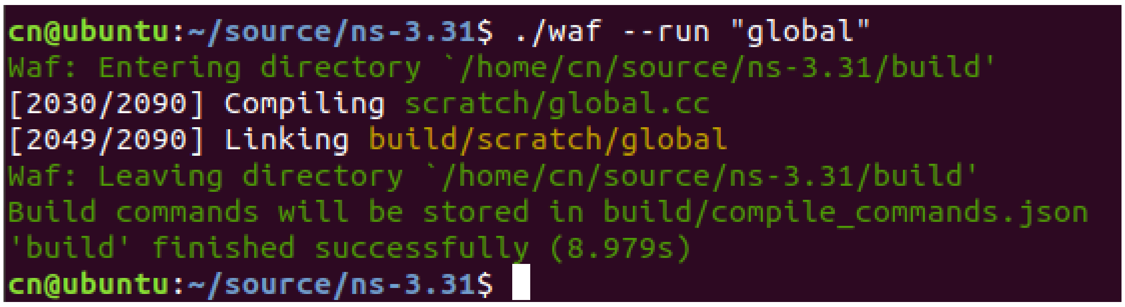
创建路由器结点

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables();
```

输出路由表

```
// 查探路由表
    Ipv4GlobalRoutingHelper g;
    Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper>
("dynamic-global-routing.routes", std::ios::out);
    //g.PrintRoutingTableAllAt(Seconds(12), routingStream);
    double i = 0.0;
    while (i < 20) {//每隔0.5s就打印一次路由表
        g.PrintRoutingTableAt(Seconds(i), c.Get(0),routingStream);
        i = i + 0.5;
    }
```

（2）运行global.cc，查看路由表



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "global"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
[2030/2090] Compiling scratch/global.cc
[2049/2090] Linking build/scratch/global
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (8.979s)
cn@ubuntu:~/source/ns-3.31$ 
```

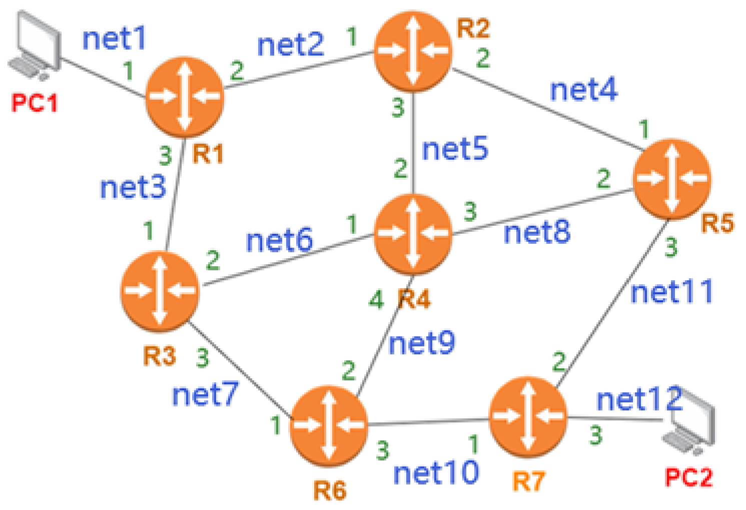
运行后无法输出路由表。

询问过老师，可能因为全局路由具有上帝视角，没有输出路由表这个功能。

3.2在拓扑1基础上，改用RIP路由协议, 增加拓扑-2所示Metric值；PC1和PC2间运行ping应用测试、并观察报文走向

（1）创建网络拓扑

①网络拓扑示意图（附端口号和网络）如下



②网络信息如下：

NodeContainer	网络号	子网掩码
net1 (src, r1)	10.0.0.0	255.255.255.0
net2 (r1, r2)	10.0.1.0	255.255.255.0
net3 (r1, r3)	10.0.2.0	255.255.255.0
net4 (r2, r5)	10.0.3.0	255.255.255.0
net5 (r2, r4)	10.0.4.0	255.255.255.0
net6 (r3, r4)	10.0.5.0	255.255.255.0
net7 (r3, r6)	10.0.6.0	255.255.255.0
net8 (r4, r5)	10.0.7.0	255.255.255.0
net9 (r4, r6)	10.0.8.0	255.255.255.0
net10 (r6, r7)	10.0.9.0	255.255.255.0
net11 (r5, r7)	10.0.10.0	255.255.255.0
net12 (r7, dst)	10.0.11.0	255.255.255.0

（2）代码主要修改部分如下

完整代码见ripM.cc，以下仅展示重要部分：

创建结点

```
NS_LOG_INFO ("Create nodes.");
  Ptr<Node> src = CreateObject<Node> ();
  Names::Add ("SrcNode", src);
  Ptr<Node> dst = CreateObject<Node> ();
  Names::Add ("DstNode", dst);
  Ptr<Node> r1 = CreateObject<Node> ();
  Names::Add ("Router1", r1);
  Ptr<Node> r2 = CreateObject<Node> ();
  Names::Add ("Router2", r2);
  Ptr<Node> r3 = CreateObject<Node> ();
  Names::Add ("Router3", r3);
  Ptr<Node> r4 = CreateObject<Node> ();
  Names::Add ("Router4", r4);
  Ptr<Node> r5 = CreateObject<Node>();
  Names::Add("Router5", r5);
  Ptr<Node> r6 = CreateObject<Node>();
  Names::Add("Router6", r6);
  Ptr<Node> r7 = CreateObject<Node>();
  Names::Add("Router7", r7);
  NodeContainer net1 (src, r1);
  NodeContainer net2 (r1, r2);
  NodeContainer net3 (r1, r3);
  NodeContainer net4 (r2, r5);
  NodeContainer net5 (r2, r4);
  NodeContainer net6 (r3, r4);
  NodeContainer net7 (r3, r6);
  NodeContainer net8(r4, r5);
  NodeContainer net9(r4, r6);
  NodeContainer net10(r6, r7);
  NodeContainer net11(r5, r7);
  NodeContainer net12(r7, dst);
  NodeContainer routers (r1, r2, r3, r4, r5);
  routers.Add(r6);
  routers.Add(r7);
  NodeContainer nodes (src, dst);
```

创建信道

```
 NS_LOG_INFO ("Create channels.");
  CsmaHelper csma;
  csma.SetChannelAttribute ("DataRate", DataRateValue (5000000));
  csma.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (2)));
  NetDeviceContainer ndc1 = csma.Install (net1);
  NetDeviceContainer ndc2 = csma.Install (net2);
  NetDeviceContainer ndc3 = csma.Install (net3);
  NetDeviceContainer ndc4 = csma.Install (net4);
  NetDeviceContainer ndc5 = csma.Install (net5);
  NetDeviceContainer ndc6 = csma.Install (net6);
  NetDeviceContainer ndc7 = csma.Install (net7);
  NetDeviceContainer ndc8 = csma.Install(net8);
  NetDeviceContainer ndc9 = csma.Install(net9);
  NetDeviceContainer ndc10 = csma.Install(net10);
  NetDeviceContainer ndc11 = csma.Install(net11);
  NetDeviceContainer ndc12 = csma.Install(net12);
```

创建路由器结点

```
  NS_LOG_INFO ("Create IPv4 and routing");
  RipHelper ripRouting;
```

设置Metric值

```
//去除掉与两台主机相连接的接口
  ripRouting.ExcludeInterface (r1, 1);
  ripRouting.ExcludeInterface (r7, 3);

  //括号内的三个参数分别是节点，接口号，距离。这个函数是成对出现的
  ripRouting.SetInterfaceMetric (r1, 2, 3);
  ripRouting.SetInterfaceMetric (r2, 1, 3);
  ripRouting.SetInterfaceMetric(r1, 3, 3);
  ripRouting.SetInterfaceMetric(r3, 1, 3);
  ripRouting.SetInterfaceMetric(r2, 2, 2);
  ripRouting.SetInterfaceMetric(r5, 1, 2);
  ripRouting.SetInterfaceMetric(r2, 3, 8);
  ripRouting.SetInterfaceMetric(r4, 2, 8);
  ripRouting.SetInterfaceMetric(r3, 2, 6);
  ripRouting.SetInterfaceMetric(r4, 1, 6);
  ripRouting.SetInterfaceMetric(r3, 3, 2);
  ripRouting.SetInterfaceMetric(r6, 1, 2);
  ripRouting.SetInterfaceMetric(r4, 3, 3);
  ripRouting.SetInterfaceMetric(r5, 2, 3);
  ripRouting.SetInterfaceMetric(r4, 4, 4);
  ripRouting.SetInterfaceMetric(r6, 2, 4);
```

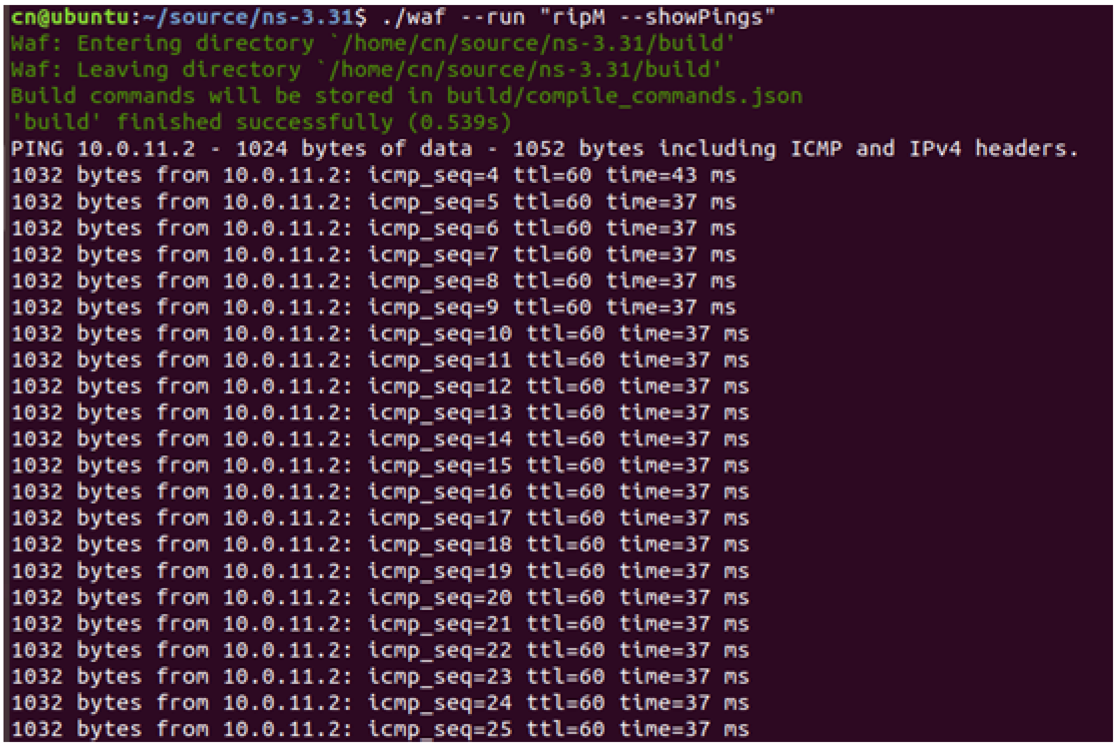
```
    ripRouting.SetInterfaceMetric(r6, 3, 5);
    ripRouting.SetInterfaceMetric(r7, 1, 5);
    ripRouting.SetInterfaceMetric(r5, 3, 10);
    ripRouting.SetInterfaceMetric(r7, 2, 10);
```

去除掉与两台主机相连接的接口，因为这两个网络不涉及路由选择协议。括号内的三个参数分别是节点，接口号，距离。这个函数是成对出现的，因为修改的距离关于两个接口号。

分配IP地址

```
//分配IP地址
    ipv4.SetBase (Ipv4Address ("10.0.0.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic1 = ipv4.Assign (ndc1);

    ipv4.SetBase (Ipv4Address ("10.0.1.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic2 = ipv4.Assign (ndc2);

    ipv4.SetBase (Ipv4Address ("10.0.2.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic3 = ipv4.Assign (ndc3);

    ipv4.SetBase (Ipv4Address ("10.0.3.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic4 = ipv4.Assign (ndc4);

    ipv4.SetBase (Ipv4Address ("10.0.4.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic5 = ipv4.Assign (ndc5);

    ipv4.SetBase (Ipv4Address ("10.0.5.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic6 = ipv4.Assign (ndc6);

    ipv4.SetBase (Ipv4Address ("10.0.6.0"), Ipv4Mask ("255.255.255.0"));
    Ipv4InterfaceContainer iic7 = ipv4.Assign (ndc7);

    ipv4.SetBase(Ipv4Address("10.0.7.0"), Ipv4Mask("255.255.255.0"));
    Ipv4InterfaceContainer iic8 = ipv4.Assign(ndc8);

    ipv4.SetBase(Ipv4Address("10.0.8.0"), Ipv4Mask("255.255.255.0"));
    Ipv4InterfaceContainer iic9 = ipv4.Assign(ndc9);

    ipv4.SetBase(Ipv4Address("10.0.9.0"), Ipv4Mask("255.255.255.0"));
    Ipv4InterfaceContainer iic10 = ipv4.Assign(ndc10);

    ipv4.SetBase(Ipv4Address("10.0.10.0"), Ipv4Mask("255.255.255.0"));
    Ipv4InterfaceContainer iic11 = ipv4.Assign(ndc11);

    ipv4.SetBase(Ipv4Address("10.0.11.0"), Ipv4Mask("255.255.255.0"));
    Ipv4InterfaceContainer iic12 = ipv4.Assign(ndc12);
```

（3）PC1和PC2间运行ping应用测试



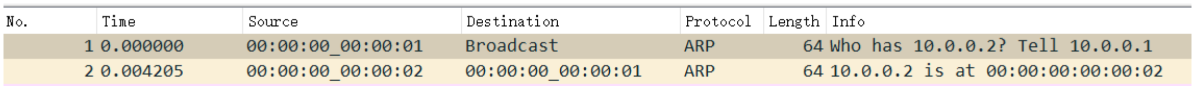
```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM --showPings"
Waf: Entering directory `/home/cn/source/ns-3.31/build'
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.539s)
PING 10.0.11.2 - 1024 bytes of data - 1052 bytes including ICMP and IPv4 headers.
1032 bytes from 10.0.11.2: icmp_seq=4 ttl=60 time=43 ms
1032 bytes from 10.0.11.2: icmp_seq=5 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=6 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=7 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=8 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=9 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=10 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=11 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=12 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=13 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=14 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=15 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=16 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=17 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=18 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=19 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=20 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=21 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=22 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=23 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=24 ttl=60 time=37 ms
1032 bytes from 10.0.11.2: icmp_seq=25 ttl=60 time=37 ms
```

由上图结果可知：在icmp_seq=4的时候才有信息被打印出来，说明icmp_seq=0-3的报文由于中间的几个路由还没有交换好信息得到目的地址的路由导致了超时传输，TTL减为0后报文被丢弃。在icmp_seq=4时才找到正确的路径，但是此时的time为43ms，后续为37ms，说明此时的转发路径还不是最短的（路由未收敛），后续找到了更短的转发路径（路由收敛）。

（4）用wireshark观察报文走向

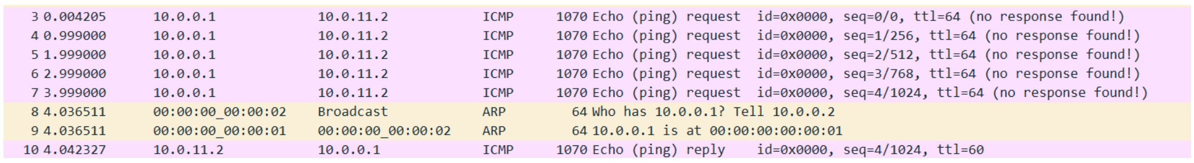
用wireshark打开ripM-SrcNode-0.pcap，观察PC1的报文。

主机src（PC1）ping主机dst（PC2）即为src向dst发送icmp报文（请求帧），然后获得来自dst的回应帧。



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	00:00:00_00:00:01	Broadcast	ARP	64	Who has 10.0.0.2? Tell 10.0.0.1
2	0.004205	00:00:00_00:00:02	00:00:00_00:00:01	ARP	64	10.0.0.2 is at 00:00:00:00:00:02

观察PC1的报文，我们可以发现，PC1先在所处的网络内发送arp报文，想要获取目的主机PC2的硬件地址，不过由于PC2与其不在同一个网络内，arp报文将交付给默认网关。然后将网关接口的硬件地址存了下来，后续各个网段内都进行如上操作，获取同一网络内默认网关的硬件地址。



3	0.004205	10.0.0.1	10.0.11.2	ICMP	1070	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!)
4	0.999000	10.0.0.1	10.0.11.2	ICMP	1070	Echo (ping) request id=0x0000, seq=1/256, ttl=64 (no response found!)
5	1.999000	10.0.0.1	10.0.11.2	ICMP	1070	Echo (ping) request id=0x0000, seq=2/512, ttl=64 (no response found!)
6	2.999000	10.0.0.1	10.0.11.2	ICMP	1070	Echo (ping) request id=0x0000, seq=3/768, ttl=64 (no response found!)
7	3.999000	10.0.0.1	10.0.11.2	ICMP	1070	Echo (ping) request id=0x0000, seq=4/1024, ttl=64 (no response found!)
8	4.036511	00:00:00_00:00:02	Broadcast	ARP	64	Who has 10.0.0.1? Tell 10.0.0.2
9	4.036511	00:00:00_00:00:01	00:00:00_00:00:02	ARP	64	10.0.0.1 is at 00:00:00:00:00:01
10	4.042327	10.0.11.2	10.0.0.1	ICMP	1070	Echo (ping) reply id=0x0000, seq=4/1024, ttl=60

观察ICMP报文。NO为3，4，5，6，7的ICMP报文分别是五条请求报文，seq为0，1，2，3，4。NO为10的报文是seq=4的请求报文的回复报文，而我们用showPings观察到在icmp_seq=4的时候才有信息被打印出来，两者是一致的。

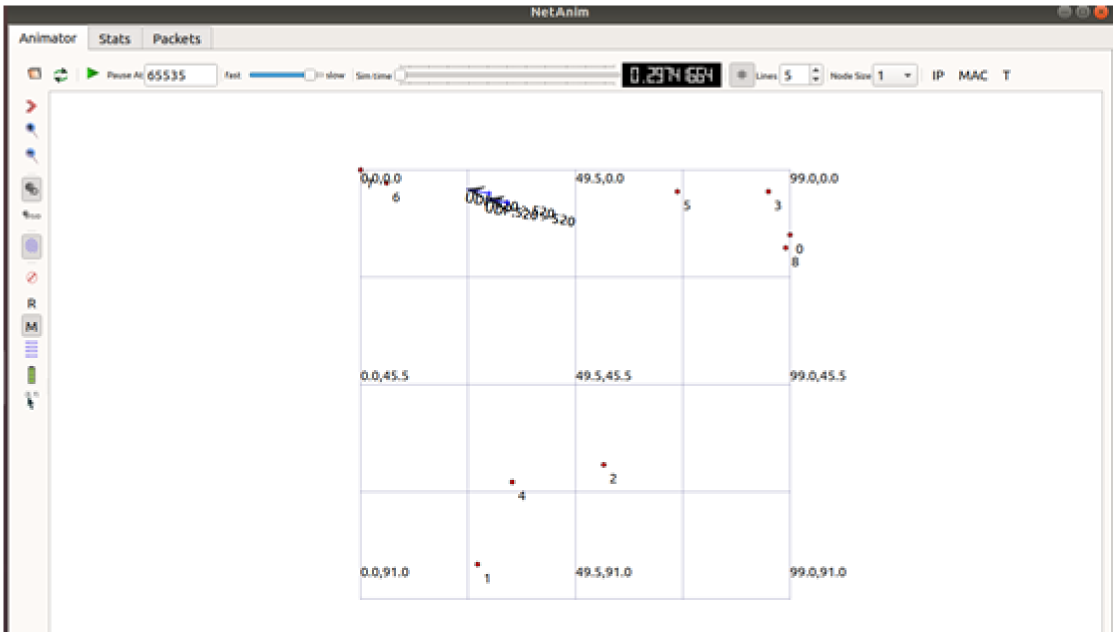
（5）借助AnimationInterface来生成“ripM-anim.xml”文件, 用NetAnim工具打开就可以直观的看到报文的走向。

①修改ripM.cc如下：

```
  AnimationInterface anim ("ripM-anim.xml");
  anim.EnablePacketMetadata(true);
  anim.AddSourceDestination(0,"10.3.1.2");

  anim.EnableIpv4RouteTracking ("ripM-track.xml", Seconds(1.0), Seconds(131.0),
nodes, Seconds(0.1));
```

②编译后运行生成"ripM-anim.xml"文件（见附件），用NetAnim工具打开：

结果如下：

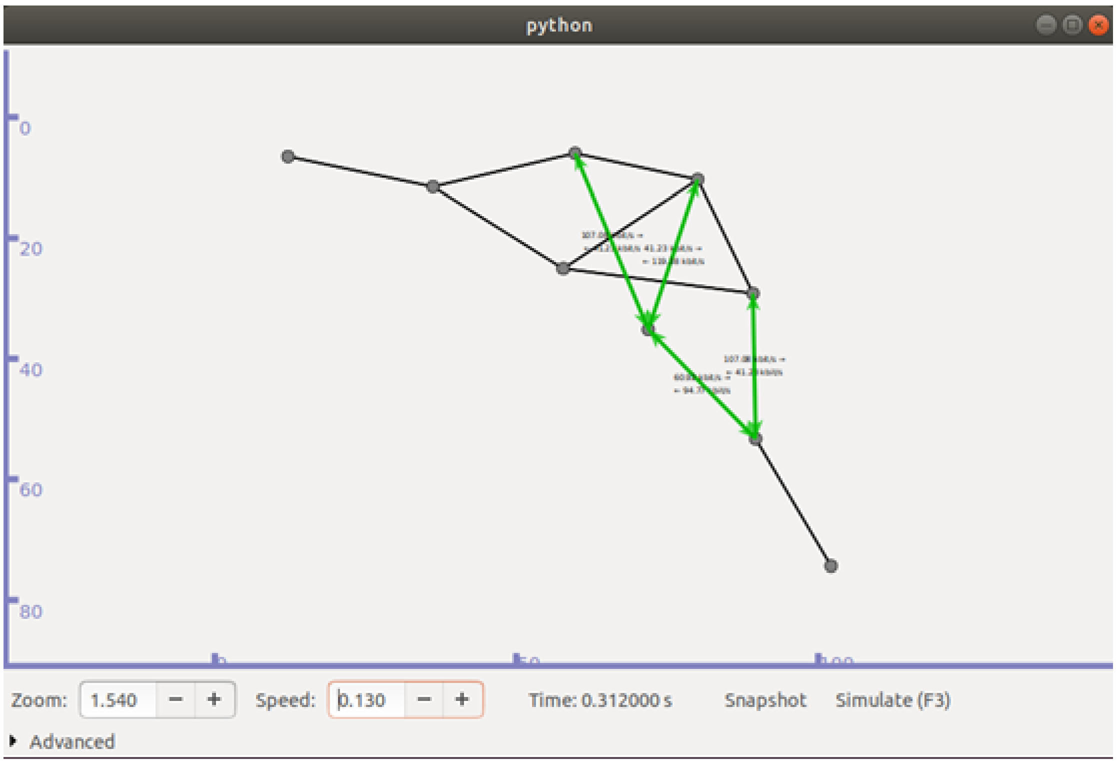


播放动画，可以清楚地看到报文走向。

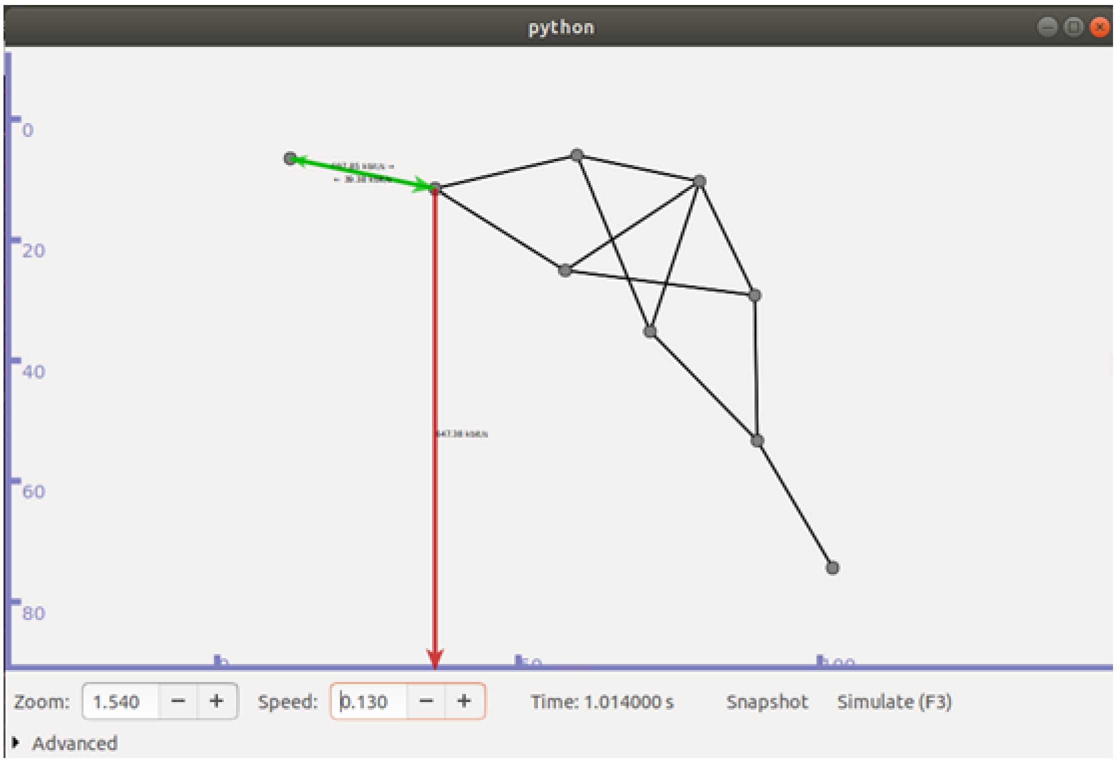
3.3观察可视化仿真效果，在路由结点上跟踪数据、捕获并存储；

（1）输入./waf --run ripM –vis，观察可视化仿真效果。

①路由处在收敛状态中（此时正在转发路由表）

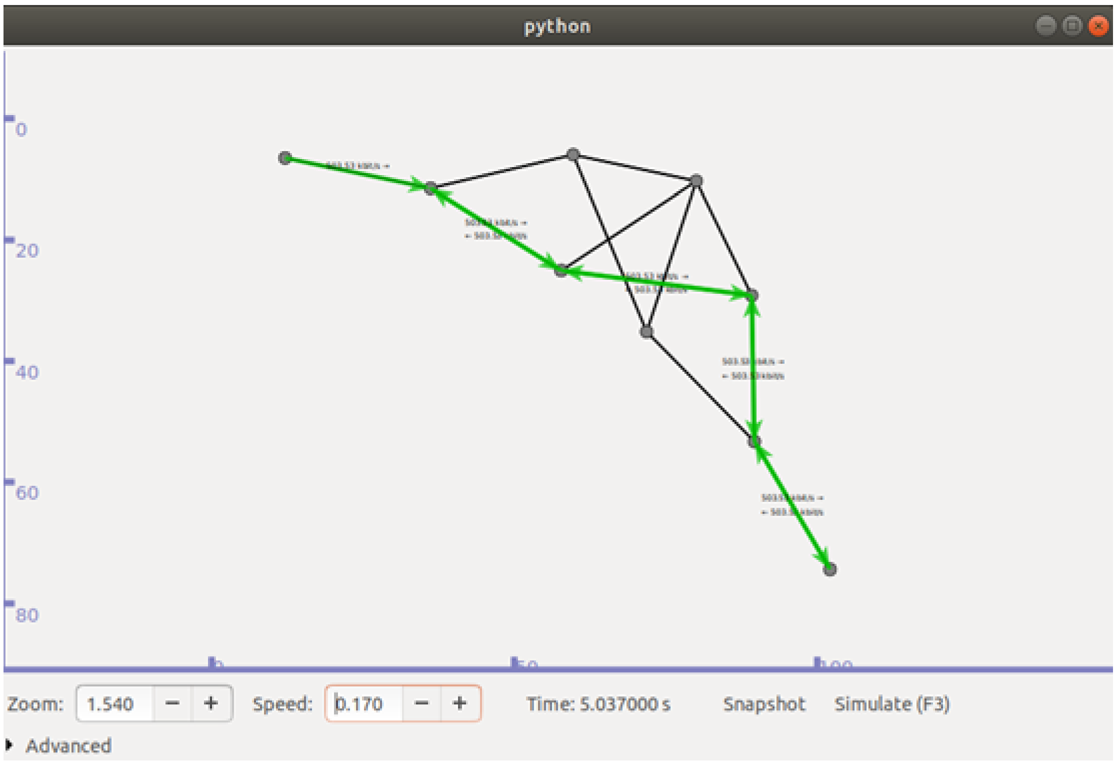


②发生超时丢包



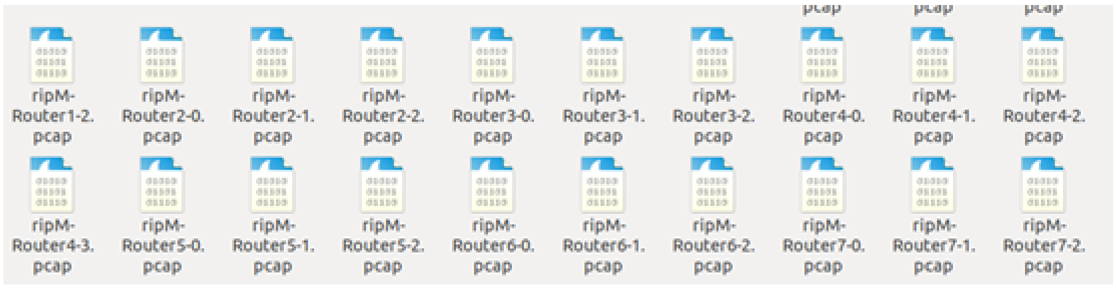
③路由收敛

经过路由r1,r3,r6,r7的路径最短，下图路径中经过的路由分别为r1,r3,r6,r7，可知这是最短路径，路由已收敛。



（2）在路由结点上跟踪数据、捕获并存储

存储下来的数据文件如下：（仅显示部分，完整的见附件）



4.任务4：结合RIP工作原理，分析其工作过程

4.1分时段打印某些结点的路由表

修改ripM.cc代码如下：

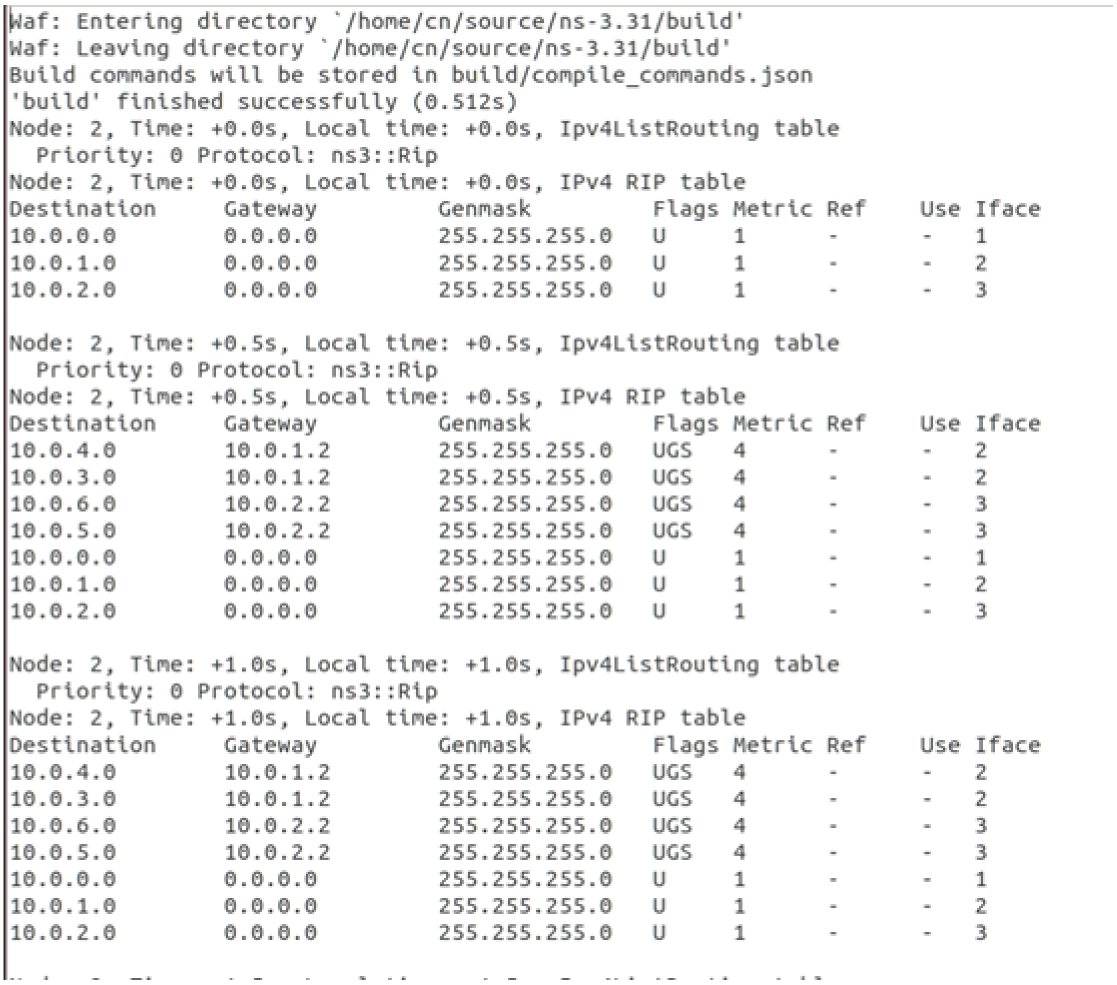
设置路由表可打印

```
bool verbose = false;
bool printRoutingTables = true;
bool showPings = false;
```

循环打印路由表

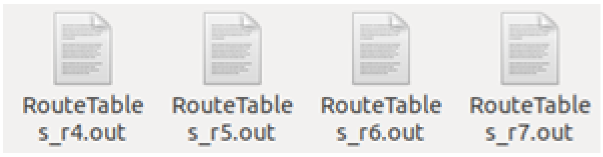
```
    double i = 0.0;
    while (i < 50) {//每隔0.5s就打印一次路由表
        routingHelper.PrintRoutingTableAt(Seconds(i), r1, routingStream);
        i = i + 0.5;
    }
```

输入./waf --run "ripM">RouteTables_r1.out 2>&1命令打印路由r1的路由表，如下所示。（完整路由表见附件文件夹路由表）



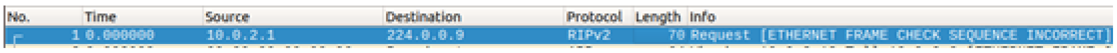
```
Waf: Entering directory `/home/cn/source/ns-3.31/build'
Waf: Leaving directory `/home/cn/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.512s)
Node: 2, Time: +0.0s, Local time: +0.0s, Ipv4ListRouting table
  Priority: 0 Protocol: ns3::Rip
Node: 2, Time: +0.0s, Local time: +0.0s, IPv4 RIP table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        0.0.0.0         255.255.255.0   U     1      -      -   1
10.0.1.0        0.0.0.0         255.255.255.0   U     1      -      -   2
10.0.2.0        0.0.0.0         255.255.255.0   U     1      -      -   3

Node: 2, Time: +0.5s, Local time: +0.5s, Ipv4ListRouting table
  Priority: 0 Protocol: ns3::Rip
Node: 2, Time: +0.5s, Local time: +0.5s, IPv4 RIP table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.4.0        10.0.1.2        255.255.255.0   UGS   4      -      -   2
10.0.3.0        10.0.1.2        255.255.255.0   UGS   4      -      -   2
10.0.6.0        10.0.2.2        255.255.255.0   UGS   4      -      -   3
10.0.5.0        10.0.2.2        255.255.255.0   UGS   4      -      -   3
10.0.0.0        0.0.0.0         255.255.255.0   U     1      -      -   1
10.0.1.0        0.0.0.0         255.255.255.0   U     1      -      -   2
10.0.2.0        0.0.0.0         255.255.255.0   U     1      -      -   3

Node: 2, Time: +1.0s, Local time: +1.0s, Ipv4ListRouting table
  Priority: 0 Protocol: ns3::Rip
Node: 2, Time: +1.0s, Local time: +1.0s, IPv4 RIP table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.4.0        10.0.1.2        255.255.255.0   UGS   4      -      -   2
10.0.3.0        10.0.1.2        255.255.255.0   UGS   4      -      -   2
10.0.6.0        10.0.2.2        255.255.255.0   UGS   4      -      -   3
10.0.5.0        10.0.2.2        255.255.255.0   UGS   4      -      -   3
10.0.0.0        0.0.0.0         255.255.255.0   U     1      -      -   1
10.0.1.0        0.0.0.0         255.255.255.0   U     1      -      -   2
10.0.2.0        0.0.0.0         255.255.255.0   U     1      -      -   3
```

同理，可得到7个路由器的路由表。



（1）分析典型RIP报文的主要结构及功能

①用wireshark打开ripM-Router3-0.pcap，观察路由r3的报文。

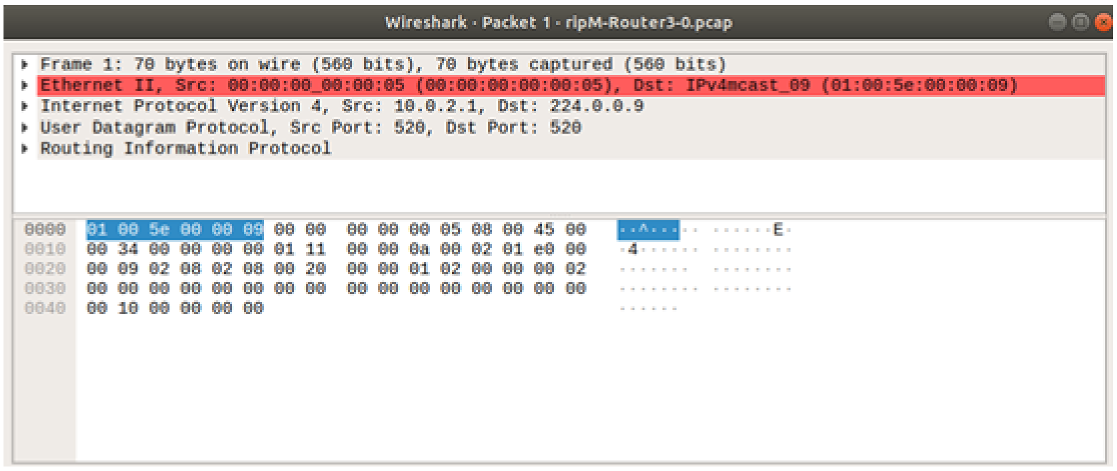


```
No.   Time       Source      Destination   Protocol  Length  Info
    1 0.000000   10.0.2.1    224.0.0.9     RIPv2         70  Request [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
```

如上图所示，NO为1的报文是一个request报文。在时间为0时，源IP为10.0.2.1（路由r1发出，从路由r3的1号端口进入）向目的IP为224.0.0.9发送了一个RIPv2协议的报文，长度为70字节，并且是一个请求报文。224.0.0.9是RFC2453文档分配RIP version2使用的IP组播地址。

②RIP v2报文的格式分析

以ripM-Router3-0.pcap中NO为1的报文为例。



Wireshark · Packet 1 · ripM-Router3-0.pcap

```
▸ Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
▸ Ethernet II, Src: 00:00:00_00:00:05 (00:00:00:00:00:05), Dst: IPv4mcast_09 (01:00:5e:00:00:09)
▸ Internet Protocol Version 4, Src: 10.0.2.1, Dst: 224.0.0.9
▸ User Datagram Protocol, Src Port: 520, Dst Port: 520
▸ Routing Information Protocol
```

```
0000  01 00 5e 00 00 09 00 00  00 00 00 05 08 00 45 00   ··^·····  ······E·
0010  00 34 00 00 00 00 01 11  00 00 0a 00 02 01 e0 00   ·4······  ········
0020  00 09 02 08 02 08 00 20  00 00 01 02 00 00 00 02   ·······   ········
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0040  00 10 00 00 00 00                                  ······
```

每个字段的信息如下表所示：

	该报文的信息	**说明**
目的地址硬件地址（6个字节）	01 00 5e 00 00 09	
源地址硬件地址(6个字节)	00 00 00 00 00 05	
类型（2个字节）	08 00	
版本号（4位）	4	
首部长度（4位）	5	
服务类型（1个字节）	00	
总长度（2个字节）	00 34	
标识（2个字节）	00 00	
标志位+片偏移(2个字节)	00 00	
TTL(1个字节)	01	
协议(1个字节)	11	表示协议为UDP
首部检验和（2个字节）	00 00	
IP源地址(4个字节)	0a 00 02 01	
IP目的地址(4个字节)	e0 00 00 09	
源端口号	02 08	
目的端口号	02 08	
长度	00 20	
Command	01	表示这是请求报文
Version	02	表示版本为RIPv2
Unused	00 00	
Address Family	00 02	地址族标识，其值为2时表示IP协议。
Route Tag	00 00	外部路由标记
IP Address	00 00 00 00	该路由的目的IP地址
Netmask	00 00 00 00	目的地址的掩码

	该报文的信息	说明
Next Hop	00 00 00 00	提供一个更好的下一跳地址。如果为0.0.0.0，则表示发布此路由的路由器地址就是最优下一跳地址。
Metric	00 00 00 10	路由的开销。对于请求报文，此字段为16。

表中源端口号和目的端口号都是02 08，即520，查阅RFC2453文档得知，在RIPv2中，每个使用RIP路由协议的路由器都在UDP端口号520上发送和接收数据报路由。

（2）分析路由初次收敛过程和时间

列出7个路由器的路由表收敛过程如下：（其中列出的时间点是发生变化的时间点）

路由器r1	**+0.0s**	**+0.5s**	**+3.5s**	**+4.0s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0	1	1	1	1
10.0.1.0	1	1	1	1
10.0.2.0	1	1	1	1
10.0.3.0		4	4	4
10.0.4.0		4	4	4
10.0.5.0		4	4	4
10.0.6.0		4	4	4
10.0.7.0			6	6
10.0.8.0			9	6
10.0.9.0				6
10.0.10.0			6	6
10.0.11.0				11

路由器r2	**+0.0s**	**+1.0s**	**+4.5s**
Destination	Metric	Metric	Metric
10.0.0.0		4	4
10.0.1.0	1	1	1
10.0.2.0		4	4
10.0.3.0	1	1	1
10.0.4.0	1	1	1
10.0.5.0		6	6
10.0.6.0		7	7
10.0.7.0		3	3
10.0.8.0		6	6
10.0.9.0		13	9
10.0.10.0		3	3
10.0.11.0		13	13

路由器r3	**+0.0s**	**+1.0s**
Destination	Metric	Metric
10.0.0.0		4
10.0.1.0		4
10.0.2.0	1	1
10.0.3.0		7
10.0.4.0		7
10.0.5.0	1	1
10.0.6.0	1	1
10.0.7.0		7
10.0.8.0		3
10.0.9.0		3
10.0.10.0		8
10.0.11.0		8

路由器r4	**+0.0s**	**+0.5s**	**+3.5s**	**+7.0s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0			10	9
10.0.1.0		6	6	6
10.0.2.0		7	7	7
10.0.3.0		4	4	4
10.0.4.0	1	1	1	1
10.0.5.0	1	1	1	1
10.0.6.0		5	5	5
10.0.7.0	1	1	1	1
10.0.8.0	1	1	1	1
10.0.9.0		5	5	5
10.0.10.0		4	4	4
10.0.11.0		10	10	10

路由器r5	**+0.0s**	**+0.5s**	**+3.5s**	**+5.0s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0			6	6
10.0.1.0		3	3	3
10.0.2.0			6	6
10.0.3.0	1	1	1	1
10.0.4.0		3	3	3
10.0.5.0		4	4	4
10.0.6.0			9	8
10.0.7.0	1	1	1	1
10.0.8.0		4	4	4
10.0.9.0		11	11	8
10.0.10.0	1	1	1	1
10.0.11.0		11	11	11

路由器r6	**+0.0s**	**+0.5s**	**+4.0s**	**+5.0s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0			6	6
10.0.1.0			6	6
10.0.2.0		3		3
10.0.3.0			9	8
10.0.4.0		5	5	5
10.0.5.0		3	3	3
10.0.6.0	1	1	1	1
10.0.7.0		5	5	5
10.0.8.0	1	1	1	1
10.0.9.0	1	1	1	1
10.0.10.0		6		6
10.0.11.0		6		6

路由器r7	**+0.0s**	**+0.5s**	**+3.0s**	**+4.5s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0				11
10.0.1.0			13	11
10.0.2.0				8
10.0.3.0		11	11	11
10.0.4.0			13	10
10.0.5.0			14	8
10.0.6.0		6	6	6
10.0.7.0		11	11	10
10.0.8.0		6	6	6
10.0.9.0	1	1	1	1
10.0.10.0	1	1	1	1
10.0.11.0	1	1	1	1

比较各个路由最后更新路由表的时间，并取其中的最大值，得到路由表收敛的时间。

除了r4以外，其他路由器在5.0s的时候均已收敛，而r4是在7.0s打印路由表的时候收敛的，说明在我们测试的精度下，链路在7.0s时初次收敛。

4.2构造“坏消息”：断开R3与R6间的连接，观察路由变化

修改ripM.cc如下：

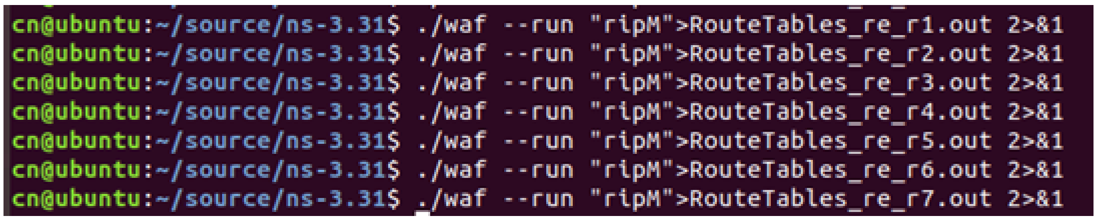
改为水平分割方式

```
//std::string SplitHorizon ("PoisonReverse");
std::string SplitHorizon("SplitHorizon");
```

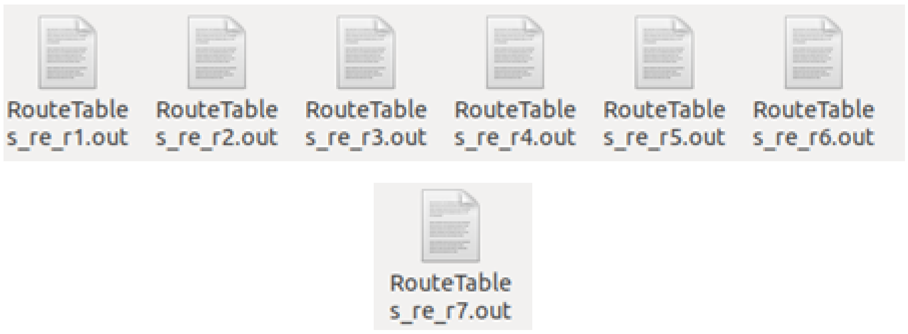
在第20秒时断开r3与r6的连接

```
Simulator::Schedule (Seconds (20), &TearDownLink, r3, r6, 3, 1);
```

运行并保存结果：



```
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r1.out 2>&1
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r2.out 2>&1
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r3.out 2>&1
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r4.out 2>&1
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r5.out 2>&1
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r6.out 2>&1
cn@ubuntu:~/source/ns-3.31$ ./waf --run "ripM">RouteTables_re_r7.out 2>&1
```



（1）观察路由再次收敛过程

观察20s后的路由表，其中列出的时间点为初始时间点和数据产生变化的时间点。

路由器r1	**+20.0s**	**+24.5s**	**+40.5s**	**+41.5s**	**+44.0s**
Destination	Metric	Metric	Metric	Metric	Metric
10.0.0.0	1	1	1	1	1
10.0.1.0	1	1	1	1	1
10.0.2.0	1	1	1	1	1
10.0.3.0	4	4	4	4	4
10.0.4.0	4	4	4	4	4
10.0.5.0	4	4	4	4	4
10.0.6.0	4				
10.0.7.0	6	6	6	6	6
10.0.8.0	6			10	9
10.0.9.0	6		13	13	13
10.0.10.0	6	6	6	6	6
10.0.11.0	11				

路由器r2	**+20.0s**	**+27.5s**	**+37.5s**	**+40.0s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0	4	4	4	4
10.0.1.0	1	1	1	1
10.0.2.0	4	4	4	4
10.0.3.0	1	1	1	1
10.0.4.0	1	1	1	1
10.0.5.0	6	6	6	6
10.0.6.0	7			
10.0.7.0	3	3	3	3
10.0.8.0	6	6	6	6
10.0.9.0	9		13	10
10.0.10.0	3	3	3	3
10.0.11.0	13	13	13	13

路由器r3	**+20.0s**	**+20.5s**	**+37.5s**	**+38.0s**	**+76.5s**
Destination	Metric	Metric	Metric	Metric	Metric
10.0.0.0	4	4	4	4	4
10.0.1.0	4	4	4	4	4
10.0.2.0	1	1	1	1	1
10.0.3.0	7	7	7	7	7
10.0.4.0	7	7	7	7	7
10.0.5.0	1	1	1	1	1
10.0.6.0	1				16
10.0.7.0	7		7	7	7
10.0.8.0	3		7	7	7
10.0.9.0	3		11	11	11
10.0.10.0	8		10	9	9
10.0.11.0	8				16

路由器r4	**+20.0s**	**+23.0s**	**+40.0s**
Destination	Metric	Metric	Metric
10.0.0.0	9	9	9
10.0.1.0	6	6	6
10.0.2.0	7		9
10.0.3.0	4	4	4
10.0.4.0	1	1	1
10.0.5.0	1	1	1
10.0.6.0	5		
10.0.7.0	1	1	1
10.0.8.0	1	1	1
10.0.9.0	5	5	5
10.0.10.0	4	4	4
10.0.11.0	10	10	10

路由器r5	**+20.0s**	**+27.5s**
Destination	Metric	Metric
10.0.0.0	6	6
10.0.1.0	3	3
10.0.2.0	6	6
10.0.3.0	1	1
10.0.4.0	3	3
10.0.5.0	4	4
10.0.6.0	8	
10.0.7.0	1	1
10.0.8.0	4	4
10.0.9.0	8	8
10.0.10.0	1	1
10.0.11.0	11	11

路由器r6	**+20.0s**	**+20.5s**	**+37.5s**	**+45.0s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0	6		13	13
10.0.1.0	6		10	10
10.0.2.0	3			13
10.0.3.0	8	8	8	8
10.0.4.0	5	5	5	5
10.0.5.0	3		5	5
10.0.6.0	1			
10.0.7.0	5	5	5	5
10.0.8.0	1	1	1	1
10.0.9.0	1	1	1	1
10.0.10.0	6	6	6	6
10.0.11.0	6	6	6	6

路由器r7	**+20.0s**	**+23.0s**	**+40.0s**	**+41.5s**
Destination	Metric	Metric	Metric	Metric
10.0.0.0	11			
10.0.1.0	11		13	13
10.0.2.0	8			
10.0.3.0	11	11	11	11
10.0.4.0	10	10	10	10
10.0.5.0	8		14	10
10.0.6.0	6			
10.0.7.0	10	10	10	10
10.0.8.0	6	6	6	6
10.0.9.0	1	1	1	1
10.0.10.0	1	1	1	1
10.0.11.0	1	1	1	1

设置20s时断开R3与R6间的连接，观察路由表可以看出，在我们测试的精度下，链路在76.5s时再次收敛，再次收敛经过了约56.5s。

（2）比较两次路由收敛所花费的时间

初次收敛花费了约7.0s，再次收敛花费了约56.5s。坏消息出现后经过了约56.5s路由才重新收敛，而一开始只经过约7.0s就收敛了，可以明显地感受到坏消息传得慢。

四、实验小结

通过本次实验，学习了仿真软件ns3的基本操作和代码格式。通过ns3，可以清楚地看到链路中报文的传播以及路由表的更新过程，在可视化时，设置不同的速度，可以看到网络上的情况是瞬息万变的，什么时候会产生超时丢包，什么时候路由才会收敛，我们在运行之前都无法给出准确的数值，只有在实验过程中才能得到答案，这也验证了实践出真知的道理。

在调试全局路由的代码时，出现了一些小问题，比如我一开始在路由器之间建立的是csma信道，但是运行后会报错。经老师提醒后发现，路由器之间应设置p2p信道，但是运行后发现无法打印路由表。老师分析了全局路由的缺点，说明全局路由具有上帝视角，是没有打印路由表这个功能的。

在这次实验中践行了课本中学到的理论知识，受益匪浅。

五、相关代码文档和文件记录

1.任务一

捕获的ipv6报文在**ipv6.pcapng**中

2.任务二

修改后的代码见**firstM.cc**以及**thirdM.cc**

firstM.cc生成的pcap文件是**first-0-0.pcap**以及**first-1-0.pcap**

3.任务三

应用Ipv4GlobalRouting构建网络拓扑-1的代码文件是**global.cc**

在拓扑1基础上，改用RIP路由协议, 增加拓扑-2所示Metric值的代码文件是**ripM.cc**

PC1和PC2间运行ping应用测试、并观察报文走向的记录文件在**文件夹ping**和**文件夹netanim**中

在路由结点上跟踪数据、捕获并存储的文件在**文件夹ping**中

4.任务四

分时段打印所有结点的路由表在**文件夹路由表**中

在20s时断开R3与R6间的连接候产生的路由表在**文件夹20s断开后的路由表**中