# 信息网络建模与仿真

# 实验二：信息网络仿真中的随机量

# 指导书

2021年11月

# 一、实验目的

1、掌握NS-3中对业务源、拓扑模型、运动模型和信道模型的建模方法。

2、进一步熟悉NS-3的使用方法，学会对NS-3中的实例进行修改、编译和测试。

# 二、实验要求

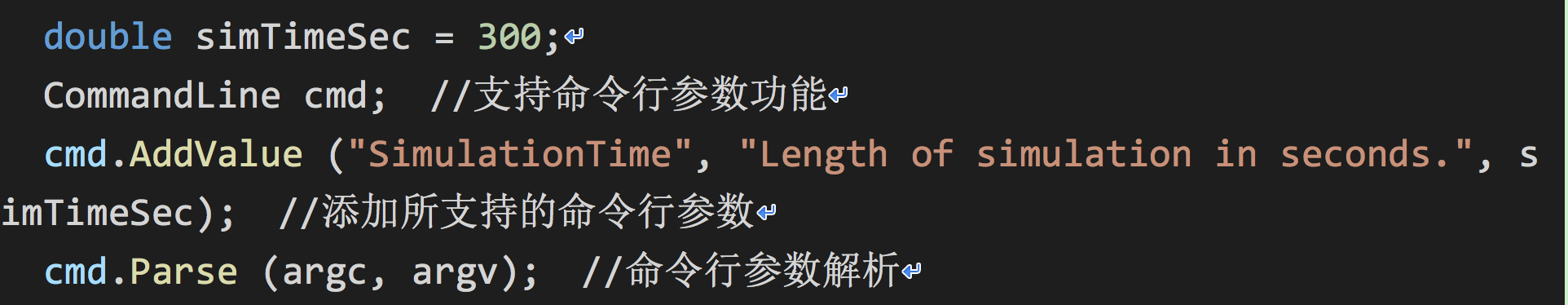
1. 结合课上对于three-gpp-http业务建模的分析，分析、编译并运行src/application/examples/three-gpp-http-example.cc，并分析运行结果。修改模型中所包含的随机分布的参数，重新编译、运行程序，并分析修改参数对运行结果所带来的变化，观察程序执行结果的变化情况。
2. 分析src/mobility/examples/main-random-walk.cc以及所对应的位置分配和移动性模型代码，以此为基础，修改节点个数、移动性模型类型和所对应的参数，观察程序执行结果的变化情况。
3. 分析和运行src/propagation/examples/main-propagation-loss.cc代码，以该代码为基础，观察程序执行结果的变化情况。
4. 分析和运行src/brite/examples/brite-generic-example.cc代码，学习在NS3中调用brite生成网络拓扑的方法。
5. 学习爱课堂上的NS3网络仿真实例（WiFi自组网），熟悉运动模型、路径损耗模型、路由协议等相关模块的修改方法。

# 三、实验说明

## 3.1 Three-gpp-http-example.cc程序说明

该程序完成点对点拓扑建模，用两个节点分别作为服务器和客户端模拟http协议的通信和数据交换过程。

1. **CommandLine类功能**

****

（1）cmd.Addvalue

添加一个可以通过命令行修改的参数，格式为：（属性名称，属性说明，变量）。例如：

./waf --run “scratch/three-gpp-http-example --SimulationTime=30”

（2）cmd.Parse (argc,argv);

实现：将命令行输入的参数作为类CommandLine的参数进行分析。

1. **日志**
2. 日志的不同级别

NS3提供了不同详尽程度的日志级别，以帮助用户获得仿真过程的所需信息。有7个程度递增的日志级别，级别越高内容越详细，包括：

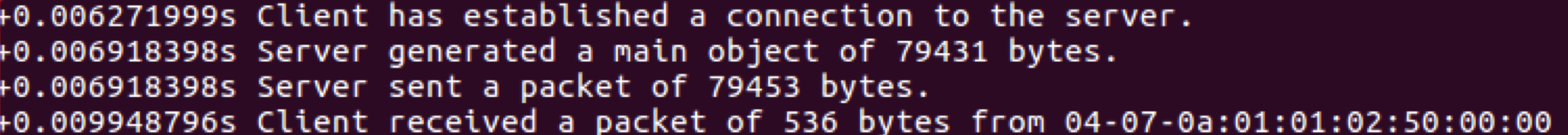
* LOG\_ERROR：记录错误信息
* LOG\_WARN：记录警告信息
* LOG\_DEBUG：记录调试信息
* LOG\_INFO：记录程序进展信息
* LOG\_FUNCTION：记录每个调用函数信息
* LOG\_LOGIC：记录每一个函数内部描述逻辑流程信息
* LOG\_ALL：记录所有信息

（2）日志的配置

A．使用LOG\_LEVEL\_INFO级别的日志配置



使用日志系统函数LogComponentEnable实现日志的配置。设置日志为LOG\_LEVEL\_INFO级别，则打开了这个日志级别和它之下的所有级别。

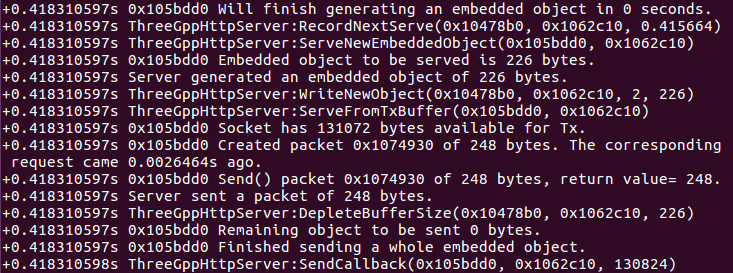


B．使用LOG\_LEVEL\_ALL级别的日志配置

上述结果只显示了基本收发信息，信息内容不够充实。在Shell界面中，可输入对环境变量NS\_LOG的设置，如下：

export NS\_LOG=ThreeGppHttpServer=level\_all

运行结果如下：



从上图可以看出，显示内容更加丰富。

**程序如下：**

int

main (int argc, char \*argv[])

{

  double simTimeSec = 300;

  CommandLine cmd; //支持命令行参数功能

  cmd.AddValue ("SimulationTime", "Length of simulation in seconds.", simTimeSec); //添加所支持的命令行参数

  cmd.Parse (argc, argv); //命令行参数解析

  Time::SetResolution (Time::NS); //设置时间分辨率

  LogComponentEnableAll (LOG\_PREFIX\_TIME);//log前打上时间

  //LogComponentEnableAll (LOG\_PREFIX\_FUNC);

  //LogComponentEnable ("ThreeGppHttpClient", LOG\_INFO);

  ///LogComponentEnable ("ThreeGppHttpServer", LOG\_INFO);

  LogComponentEnable ("ThreeGppHttpExample", LOG\_INFO);

  // Setup two nodes

  NodeContainer nodes; //节点容器

  nodes.Create (2); //创建2个节点

  PointToPointHelper pointToPoint; //点对点拓扑

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

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

//设置链路参数

  NetDeviceContainer devices;//设备容器

  devices = pointToPoint.Install (nodes);

  InternetStackHelper stack; //协议栈

  stack.Install (nodes); //在两个节点上安装协议栈

  Ipv4AddressHelper address;

  address.SetBase ("10.1.1.0", "255.255.255.0");//IP基地址和子网掩码

  Ipv4InterfaceContainer interfaces = address.Assign (devices);为两个设备分配地址

  Ipv4Address serverAddress = interfaces.GetAddress (1);

//服务器端地址

  // Create HTTP server helper

  ThreeGppHttpServerHelper serverHelper (serverAddress);

//创建3GPP HTTP Server

  // Install HTTP server

  ApplicationContainer serverApps = serverHelper.Install (nodes.Get (1)); //安装Http server

  Ptr<ThreeGppHttpServer> httpServer = serverApps.Get (0)->GetObject<ThreeGppHttpServer> (); //获得Server指针

  // Example of connecting to the trace sources

  httpServer->TraceConnectWithoutContext ("ConnectionEstablished",

                                          MakeCallback (&ServerConnectionEstablished));

  httpServer->TraceConnectWithoutContext ("MainObject", MakeCallback (&MainObjectGenerated));

  httpServer->TraceConnectWithoutContext ("EmbeddedObject", MakeCallback (&EmbeddedObjectGenerated));

  httpServer->TraceConnectWithoutContext ("Tx", MakeCallback (&ServerTx));

  // Setup HTTP variables for the server

  PointerValue varPtr;

  httpServer->GetAttribute ("Variables", varPtr);

  Ptr<ThreeGppHttpVariables> httpVariables = varPtr.Get<ThreeGppHttpVariables> ();

  httpVariables->SetMainObjectSizeMean (102400); // 100kB

  httpVariables->SetMainObjectSizeStdDev (40960); // 40kB

//设置HTTP模型有关的变量：主对象大小均值和标准差

  // Create HTTP client helper

  ThreeGppHttpClientHelper clientHelper (serverAddress);

  // Install HTTP client

  ApplicationContainer clientApps = clientHelper.Install (nodes.Get (0));

  Ptr<ThreeGppHttpClient> httpClient = clientApps.Get (0)->GetObject<ThreeGppHttpClient> ();

//生成客户端，绑定到另一节点，并获取对象指针

  // Example of connecting to the trace sources

  httpClient->TraceConnectWithoutContext ("RxMainObject", MakeCallback (&ClientMainObjectReceived));

  httpClient->TraceConnectWithoutContext ("RxEmbeddedObject", MakeCallback (&ClientEmbeddedObjectReceived));

  httpClient->TraceConnectWithoutContext ("Rx", MakeCallback (&ClientRx));

  // Stop browsing after 30 minutes

  clientApps.Stop (Seconds (simTimeSec));

  Simulator::Run ();

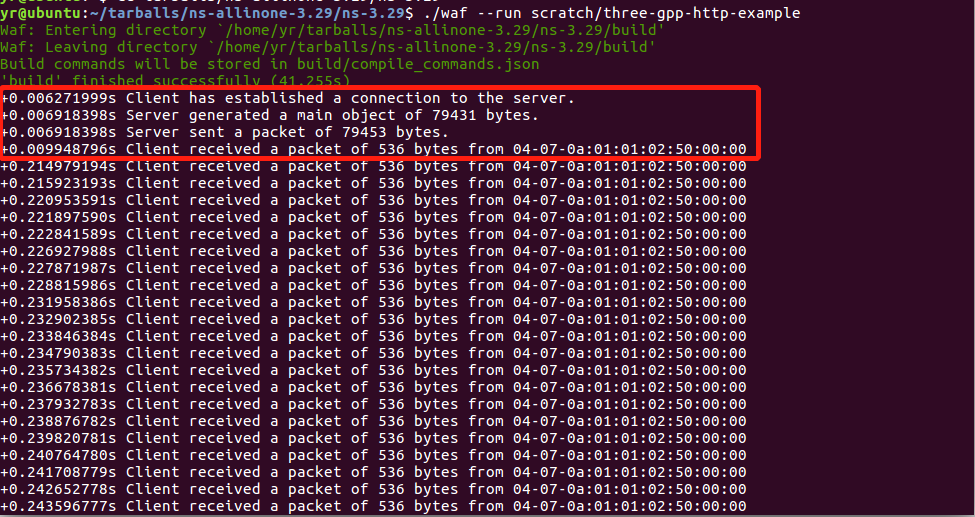
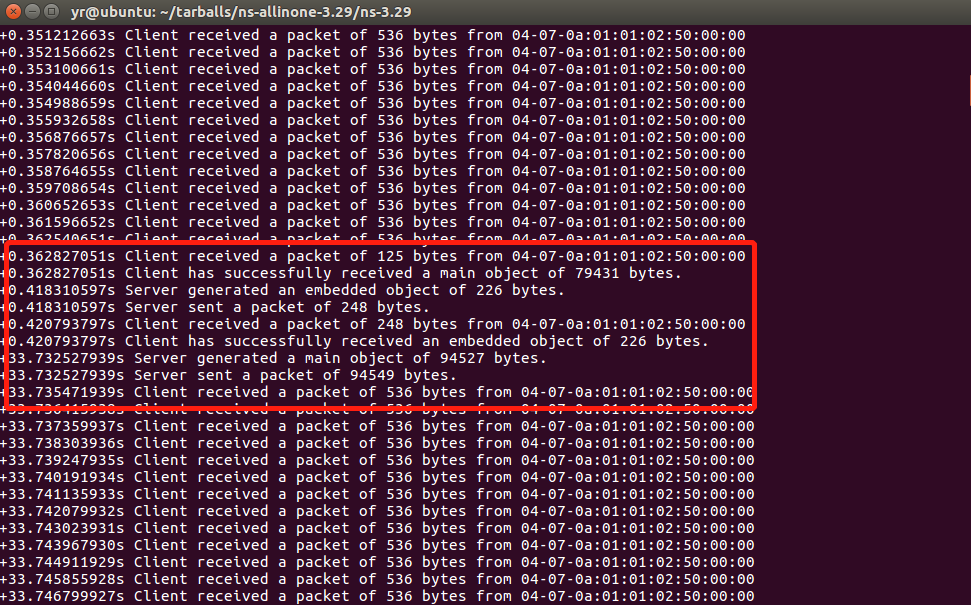
  Simulator::Destroy ();

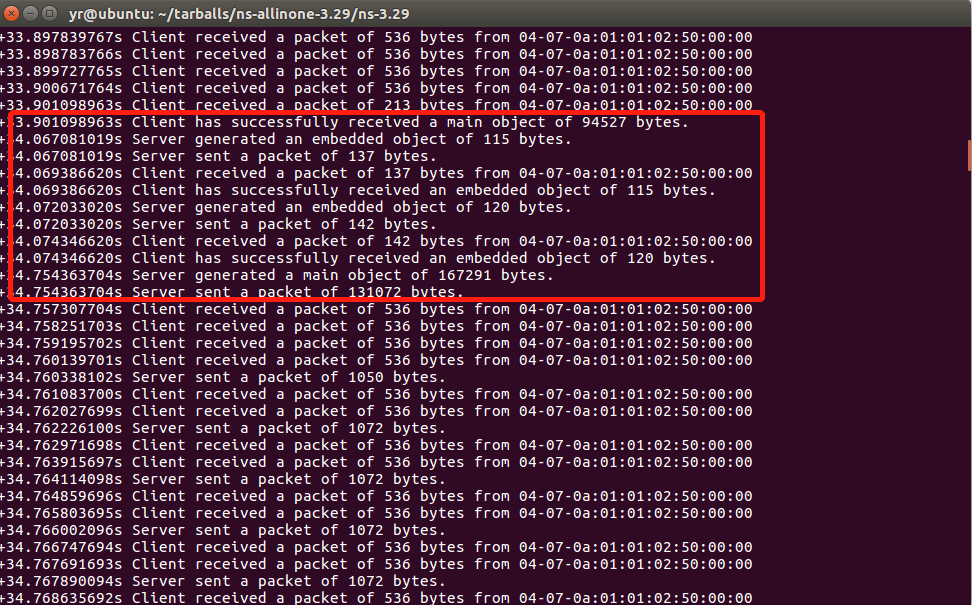
  return 0;

}

模型的实现代码在src/application/model目录中，以three-gpp开头的文件中。

运行结果的部分截图如下：



## 3.2 main-random-walk.cc程序说明

该程序是一个简单的随机游走模型示例。

/\* -\*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -\*- \*/

/\*

 \* Copyright (c) 2006,2007 INRIA

 \*

 \* This program is free software; you can redistribute it and/or modify

 \* it under the terms of the GNU General Public License version 2 as

 \* published by the Free Software Foundation;

 \*

 \* This program is distributed in the hope that it will be useful,

 \* but WITHOUT ANY WARRANTY; without even the implied warranty of

 \* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the

 \* GNU General Public License for more details.

 \*

 \* You should have received a copy of the GNU General Public License

 \* along with this program; if not, write to the Free Software

 \* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

 \*/

#include "ns3/core-module.h"

#include "ns3/mobility-module.h"

//在此处添加#include <iostream>可以看到打印的输出

using namespace ns3;

static void

CourseChange (std::string foo, Ptr<const MobilityModel> mobility)

{

  Vector pos = mobility->GetPosition ();

  Vector vel = mobility->GetVelocity ();

  std::cout << Simulator::Now () << ", model=" << mobility << ", POS: x=" << pos.x << ", y=" << pos.y

            << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y

            << ", z=" << vel.z << std::endl;//可以修改此处代码，使得不同坐标打印在不同的文件中，便于分析结果

}//输出各个节点不同时刻位置变化

int main (int argc, char \*argv[])

{

  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Mode", StringValue ("Time"));

  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Time", StringValue ("2s"));

  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Speed", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"));

  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Bounds", StringValue ("0|200|0|200"));

//配置了二维随机游走模型的参数

  CommandLine cmd;

  cmd.Parse (argc, argv);

  NodeContainer c;

  c.Create (100); //创建100个节点，可修改

  MobilityHelper mobility;

  mobility.SetPositionAllocator ("ns3::RandomDiscPositionAllocator",

                                 "X", StringValue ("100.0"),

                                 "Y", StringValue ("100.0"),

                                 "Rho", StringValue ("ns3::UniformRandomVariable[Min=0|Max=30]"));

//初始位置分配，对应于拓扑模型

  mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",

                             "Mode", StringValue ("Time"),

                             "Time", StringValue ("2s"),

                             "Speed", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"),

                             "Bounds", StringValue ("0|200|0|200"));

//设置移动性模型和对应的参数

  mobility.InstallAll ();

  Config::Connect ("/NodeList/\*/$ns3::MobilityModel/CourseChange",

                   MakeCallback (&CourseChange)); //连接回调函数

  Simulator::Stop (Seconds (100.0));

  Simulator::Run ();

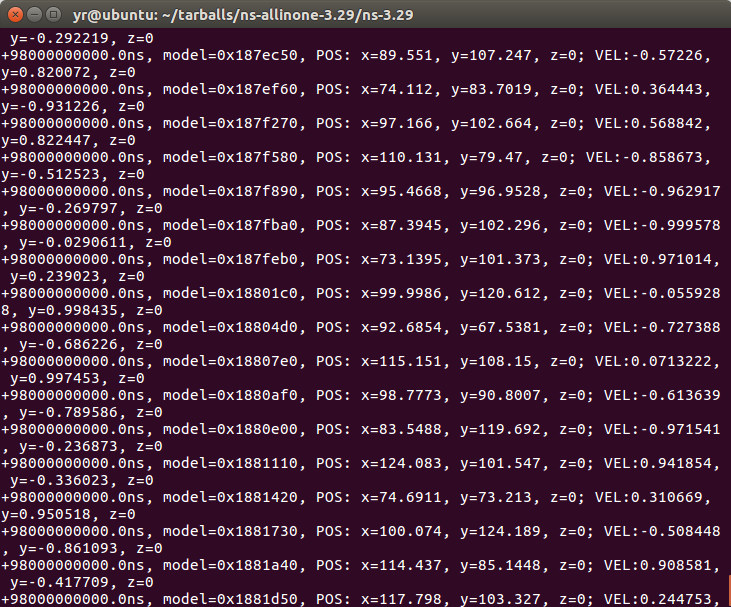
  Simulator::Destroy ();

  return 0;

}

节点数过多不利于分析节点的移动情况，可通过减少节点数量、不同节点的数据分别写入不同文件等方式使数据分析变得便利。可尝试修改为二维位点模型等其他移动性模型，也可修改随机游走模型参数并分析结果。

（1）运行结果的部分截图如下：



1. 单个节点信息输出：

通过修改connect函数中tracesource的参数，可以达到输出单个节点的信息，再修改coursechange函数。可以将单个节点信息输出到文件中，并进行后续的分析、绘图等操作。例如：跟踪节点0的位置信息。

* 修改connect函数：

  Config::Connect ("/NodeList/0/$ns3::MobilityModel/CourseChange",

                    MakeCallback (&CourseChange));

* 修改coursechange函数：

static void

CourseChange (std::string foo, Ptr<const MobilityModel> mobility)

{

  Vector pos = mobility->GetPosition ();

  Vector vel = mobility->GetVelocity ();

  std::cout << Simulator::Now () << ", model=" << mobility << ", POS: x=" << pos.x << ", y=" << pos.y

            << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y

            << ", z=" << vel.z << std::endl;

  //创建文件并输出

  std::ofstream datafile;

  datafile.open("random\_walk\_data.plt",std::ios::app);

  datafile << Simulator::Now () << ", model=" << mobility << ", POS: x=" << pos.x << ", y=" << pos.y

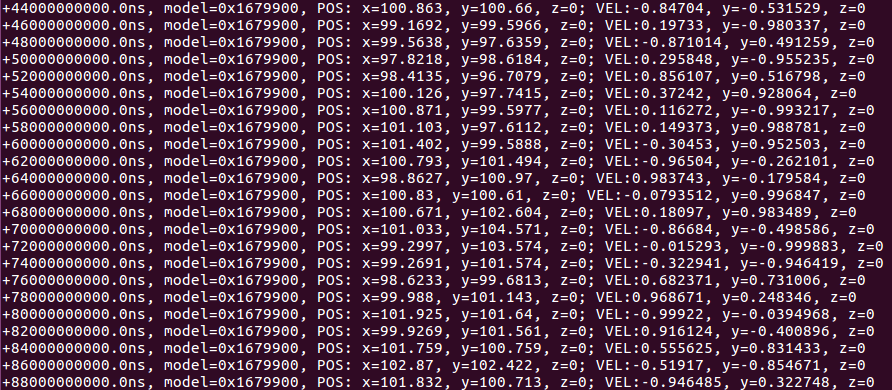
            << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y

            << ", z=" << vel.z << std::endl;

  datafile.close();

}

* 单个节点输出信息，注意model地址相同，说明是同一节点：



1. 多个节点信息输出

输出特定的多个节点信息，需要多次调用connect函数创建trace。

  for (int iter=0;iter<3;iter++)

  {

        Config::Connect ("/NodeList/"+std::to\_string(iter)+"/$ns3::MobilityModel/CourseChange",

                        MakeCallback (&CourseChange));

  }

并修改作为tracesink的CourseChange函数：

static void

CourseChange (std::string foo, Ptr<const MobilityModel> mobility)

{

  Vector pos = mobility->GetPosition ();

  Vector vel = mobility->GetVelocity ();

  std::cout << Simulator::Now () << ", model=" << mobility << ", POS: x=" << pos.x << ", y=" << pos.y

            << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y

            << ", z=" << vel.z << std::endl;

  std::stringstream stest;//stest将指针地址转化为字符串

  std::string name;//name会记录当前触发节点的地址，用来区分节点

  stest<<mobility;

  stest>>name;

  std::ofstream datafile;

  datafile.open("random\_walk\_data\_"+name+".plt",std::ios::app);//根据不同节点的地址创建对应的文件

  datafile << Simulator::Now () << ", model=" << mobility << ", POS: x=" << pos.x << ", y=" << pos.y

            << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y

            << ", z=" << vel.z << std::endl;

  datafile.close();

}

如上面两图，使用一个for循环为NodeList中索引为0-2的三个节点建立了trace，并在CourseChange中完成数据的输出。需要注意的是，此法使用节点的地址作为区分节点的唯一标识符。而计算机每次仿真的地址分配是不同的，因此在进行新的仿真之前一定要将旧的数据文件挪走，以免造成数据混乱。

1. 节点移动的可视化：

使用ns3自带的netanim完成节点移动的可视化工作，首先引入相关库：

#include "ns3/netanim-module.h"

然后在使用Simulator语句开始仿真之前插入建立输出文件的语句，如下图红色下划线语句：

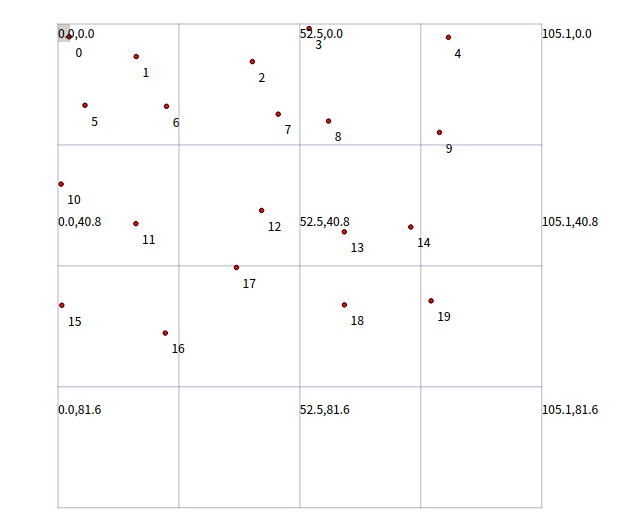
  AnimationInterface anim("random\_walk.xml");

  Simulator::Stop (Seconds (100.0));

  Simulator::Run ();

  Simulator::Destroy ();

最后在终端中的ns3.xx/netanim-3.xxx路径下执行./NetAnim命令启动软件，然后选中声称在ns3.xx中的xml文件，即可完成可视化，在该软件上方的工具栏点击播放（▶）按钮即可看到节点的具体运动过程。某一时刻的位置如下图：



注意：为了便于可视化，此图没有使用原程序中的randomDiscPositionAllocator位置分配器来分配节点的初始位置，而是创建了20个节点，并使用GridPositionAllocator完成的初始位置分配。参考源码如下：

  NodeContainer c;

  c.Create (20);//原文为100

  MobilityHelper mobility;

  //原位置分配器

  //mobility.SetPositionAllocator ("ns3::RandomDiscPositionAllocator",

  //                               "X", StringValue ("100.0"),

  //                               "Y", StringValue ("100.0"),

  //                               "Rho", StringValue ("ns3::UniformRandomVariable[Min=0|Max=30]"));

  //仿真图中所采用的位置分配器

  mobility.SetPositionAllocator ("ns3::GridPositionAllocator",

                                        "GridWidth",StringValue("5"),

                                        "DeltaX",StringValue("20"),

                                        "DeltaY",StringValue("20"));

## 3.3 main-propagation-loss.cc程序说明

该程序完成多个常用信道损耗模型的建模和仿真，得到信号功率与节点间距离的关系，并将仿真结果以图像的形式进行输出。

int main (int argc, char \*argv[])

{

  CommandLine cmd;

  cmd.Parse (argc, argv);

  GnuplotCollection gnuplots ("main-propagation-loss.pdf");

//GNUplot是一个将数据图形化的库，GnuplotCollection可以将多个图放在一个文件中

  {

    Ptr<FriisPropagationLossModel> friis = CreateObject<FriisPropagationLossModel> ();

//testDeterministic方法是计算相应信道模型功率和距离的对应关系

    Gnuplot plot = TestDeterministic (friis); //生成一个plot对象

    plot.SetTitle ("ns3::FriisPropagationLossModel (Default Parameters)"); //设置plot标题

    gnuplots.AddPlot (plot);//往gnuplots 图集合中增加一个图

  }

  {

    Ptr<LogDistancePropagationLossModel> log = CreateObject<LogDistancePropagationLossModel> ();

    log->SetAttribute ("Exponent", DoubleValue (2.5));

    Gnuplot plot = TestDeterministic (log);

    plot.SetTitle ("ns3::LogDistancePropagationLossModel (Exponent = 2.5)");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<RandomPropagationLossModel> random = CreateObject<RandomPropagationLossModel> ();

    Ptr<ExponentialRandomVariable> expVar = CreateObjectWithAttributes<ExponentialRandomVariable> ("Mean", DoubleValue (50.0));

    random->SetAttribute ("Variable", PointerValue (expVar));

    Gnuplot plot = TestDeterministic (random);

    plot.SetTitle ("ns3::RandomPropagationLossModel with Exponential Distribution");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<JakesPropagationLossModel> jakes = CreateObject<JakesPropagationLossModel> ();

    // doppler frequency shift for 5.15 GHz at 100 km/h

    Config::SetDefault ("ns3::JakesProcess::DopplerFrequencyHz", DoubleValue (477.9));

//Jakes模型中的多普勒频率

// TestDeterministicByTime计算相应信道模型在固定距离下功率和时间的关系

    Gnuplot plot = TestDeterministicByTime (jakes, Seconds (0.001), Seconds (1.0));

    plot.SetTitle ("ns3::JakesPropagationLossModel (with 477.9 Hz shift and 1 millisec resolution)");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<JakesPropagationLossModel> jakes = CreateObject<JakesPropagationLossModel> ();

    // doppler frequency shift for 5.15 GHz at 100 km/h

    Config::SetDefault ("ns3::JakesProcess::DopplerFrequencyHz", DoubleValue (477.9));

    Gnuplot plot = TestDeterministicByTime (jakes, Seconds (0.0001), Seconds (0.1));

    plot.SetTitle ("ns3::JakesPropagationLossModel (with 477.9 Hz shift and 0.1 millisec resolution)");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<ThreeLogDistancePropagationLossModel> log3 = CreateObject<ThreeLogDistancePropagationLossModel> ();

    Gnuplot plot = TestDeterministic (log3);

    plot.SetTitle ("ns3::ThreeLogDistancePropagationLossModel (Defaults)");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<ThreeLogDistancePropagationLossModel> log3 = CreateObject<ThreeLogDistancePropagationLossModel> ();

    // more prominent example values:

    log3->SetAttribute ("Exponent0", DoubleValue (1.0));

    log3->SetAttribute ("Exponent1", DoubleValue (3.0));

    log3->SetAttribute ("Exponent2", DoubleValue (10.0));

    Gnuplot plot = TestDeterministic (log3);

    plot.SetTitle ("ns3::ThreeLogDistancePropagationLossModel (Exponents 1.0, 3.0 and 10.0)");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<NakagamiPropagationLossModel> nak = CreateObject<NakagamiPropagationLossModel> ();

// TestProbabilistic方法计算每个距离下功率损耗的概率分布

    Gnuplot plot = TestProbabilistic (nak);

    plot.SetTitle ("ns3::NakagamiPropagationLossModel (Default Parameters)");

    gnuplots.AddPlot (plot);

  }

  {

    Ptr<ThreeLogDistancePropagationLossModel> log3 = CreateObject<ThreeLogDistancePropagationLossModel> ();

    Ptr<NakagamiPropagationLossModel> nak = CreateObject<NakagamiPropagationLossModel> ();

    log3->SetNext (nak);

    Gnuplot plot = TestProbabilistic (log3);

    plot.SetTitle ("ns3::ThreeLogDistancePropagationLossModel and ns3::NakagamiPropagationLossModel (Default Parameters)");

    gnuplots.AddPlot (plot);

  }

  gnuplots.GenerateOutput (std::cout);

//修改std::cout为一输出文件，则可将数据输出到文件，并在程序执行完毕后，执行gnuplot命令画图

  // produce clean valgrind

  Simulator::Destroy ();

  return 0;

}

在文件的开头添加#include <fstream>，使得程序能够将仿真结果输出到文件中以便后续绘图和分析。

* 在源程序末尾gnuplots.GenerateOutput之前，定义文件：

如：std::ofstream plotFile(“propagation\_loss.plt”);

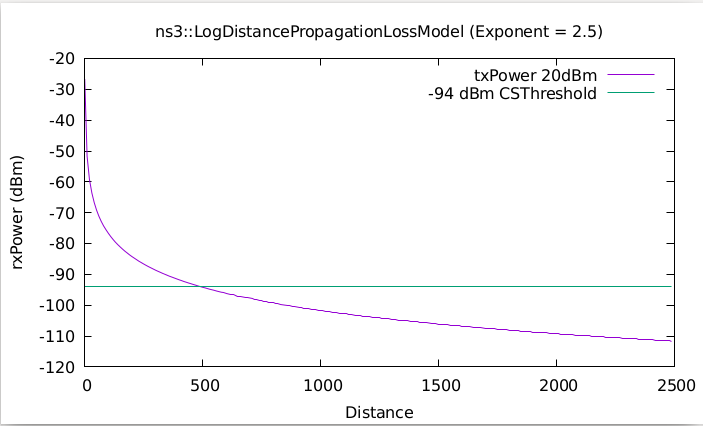
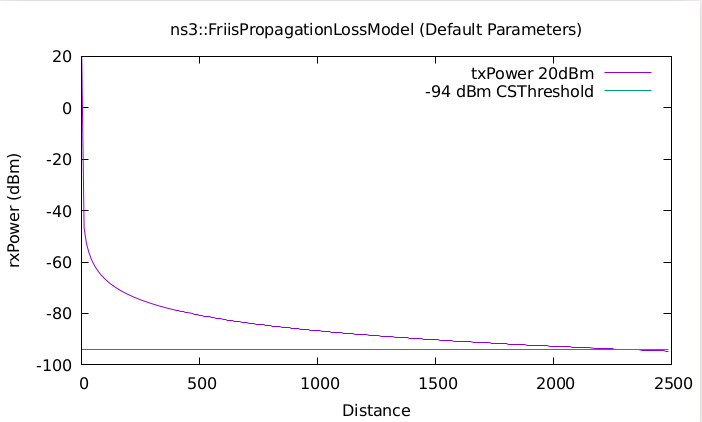
* 用这个声明的流对象作为GenerateOutput的参数：

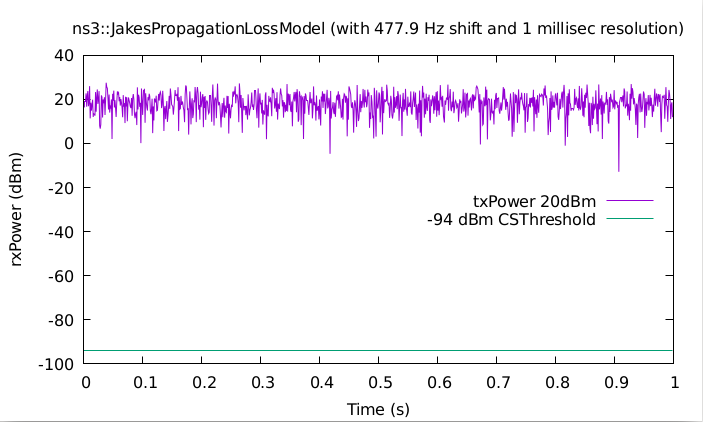
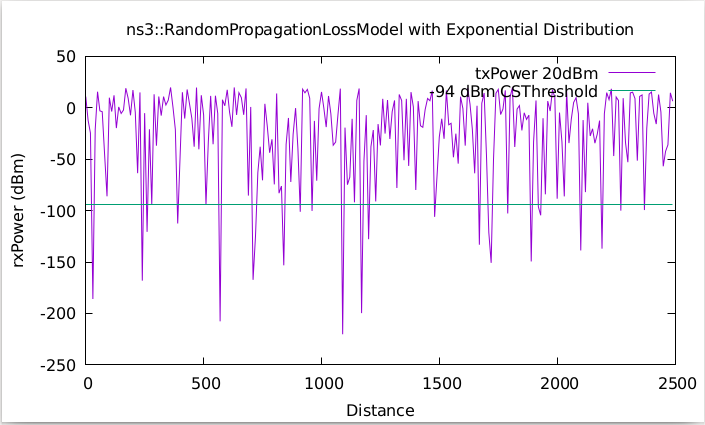
gnuplots.GenerateOutput(plotFile);

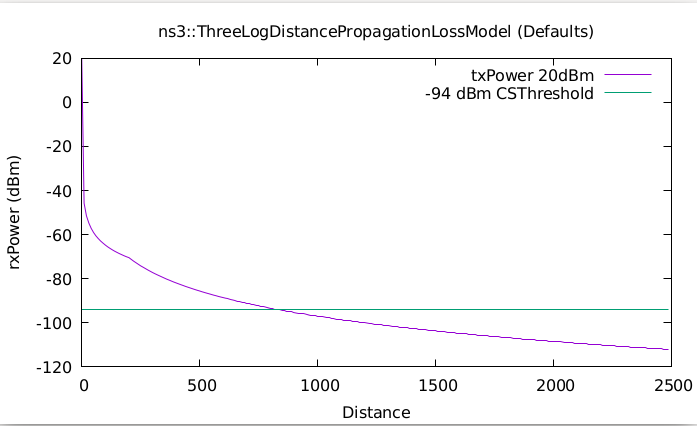
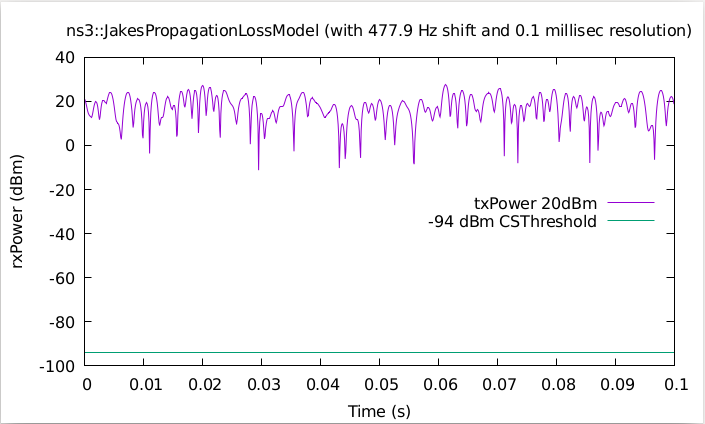
在程序执行完之后，该文件会出现在ns-3.xx文件夹下。进入该文件夹下，执行gnuplot propagation\_loss.plt命令，会在该文件夹下生成一个名为main-propagation-loss.pdf的文件，打开该文件，可以看到上述代码所生成的路径损耗模型图形。

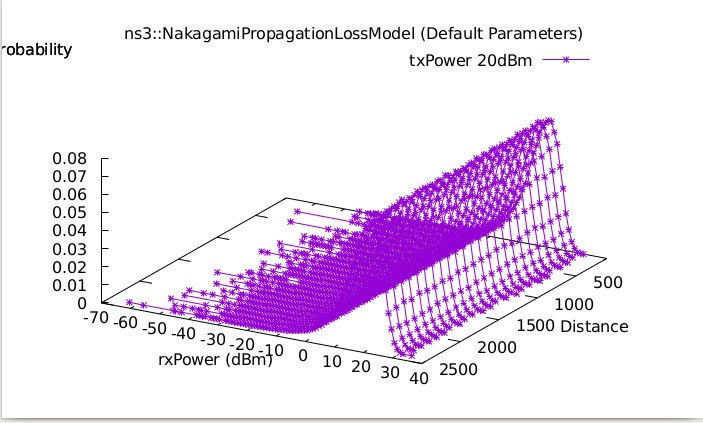
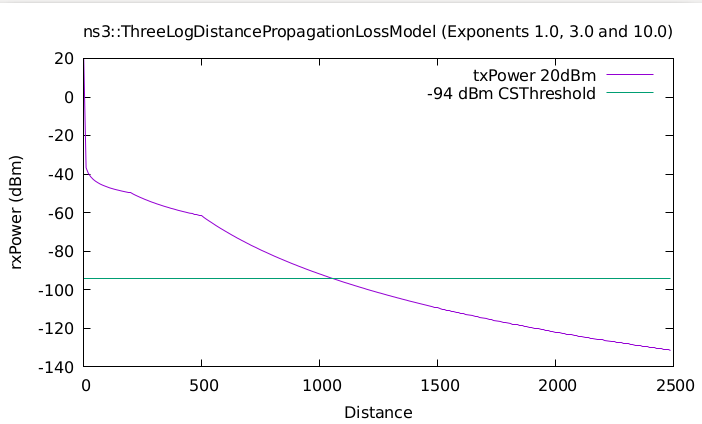
分析该例子中所包含的信道模型之间的差别，并尝试修改一些模型参数，重新编译、执行程序，并分析其影响。

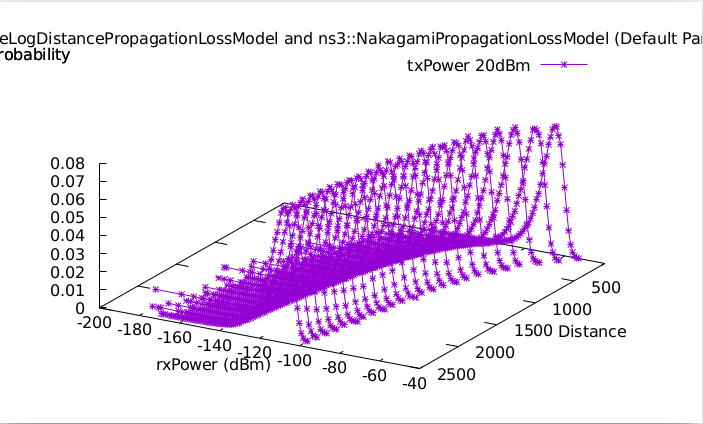
运行结果如下：











## 3.4 brite-generic-example.cc程序说明

该程序实现基于brite中的Barabasi和Waxman拓扑生成功能生成一个分层网络拓扑。该功能单元的使用要求brite模块生效，具体配置过程见NS-3-model-library.pdf 7.2.1节。

下面简单分析brite-generic-example.cc的代码。

int

main (int argc, char \*argv[])

{

LogComponentEnable ("UdpEchoClientApplication", LOG\_LEVEL\_ALL);

LogComponentEnable ("UdpEchoServerApplication", LOG\_LEVEL\_ALL);

LogComponentEnable ("BriteExample", LOG\_LEVEL\_ALL); //log等级设置为ALL，所有信息均输出。

// BRITE needs a configuration file to build its graph. By default, this

// example will use the TD\_ASBarabasi\_RTWaxman.conf file. There are many others

// which can be found in the BRITE/conf\_files directory

//这个例子中采用的配置文件是TD\_ASBarabasi\_RTWaxman.conf，它结合了Barabasi的AS拓扑和、、Waxman的RT拓扑生成算法，形成了层次化拓扑结构。

std::string confFile = "src/brite/examples/conf\_files/TD\_ASBarabasi\_RTWaxman.conf";

bool tracing = false; //追踪功能开关

bool nix = false; //nix是nix向量路由的选项开关

CommandLine cmd (\_\_FILE\_\_);

cmd.AddValue ("confFile", "BRITE conf file", confFile);

cmd.AddValue ("tracing", "Enable or disable ascii tracing", tracing);

cmd.AddValue ("nix", "Enable or disable nix-vector routing", nix);

cmd.Parse (argc,argv);//对命令行参数进行解析，可以通过命令行参数选择配置文件、是否追踪和是否选择nix向量录用。

// Invoke the BriteTopologyHelper and pass in a BRITE

// configuration file and a seed file. This will use

// BRITE to build a graph from which we can build the ns-3 topology

BriteTopologyHelper bth (confFile);//调用BriteTopologyHelper，其构造函数参数为配置文件名

bth.AssignStreams (3); //

PointToPointHelper p2p; //点对点通道对象

InternetStackHelper stack; //协议栈对象

if (nix)

{

Ipv4NixVectorHelper nixRouting;

stack.SetRoutingHelper (nixRouting);

} //如果选择nix，修改路由算法，否则，采用默认得全局路由

Ipv4AddressHelper address; //IPv4Helper对象

address.SetBase ("10.0.0.0", "255.255.255.252");//基地址和子网掩码

bth.BuildBriteTopology (stack); //建立拓扑，并安装指定的协议栈

bth.AssignIpv4Addresses (address); //分配IP地址

NS\_LOG\_INFO ("Number of AS created " << bth.GetNAs ());//创建的AS个数

//The BRITE topology generator generates a topology of routers. Here we create

//two subnetworks which we attach to router leaf nodes generated by BRITE

//Any NS3 topology may be used to attach to the BRITE leaf nodes but here we

//use just one node

NodeContainer client; //客户端对象

NodeContainer server; //服务器对象

client.Create (1); //创建1个客户端

stack.Install (client);//为客户端安装协议栈

//install client node on last leaf node of AS 0

int numLeafNodesInAsZero = bth.GetNLeafNodesForAs (0); //将客户端指定到AS0的最后一个叶子节点

client.Add (bth.GetLeafNodeForAs (0, numLeafNodesInAsZero - 1));

server.Create (1); //创建一个服务器

stack.Install (server);//为服务器安装协议栈

//install server node on last leaf node on AS 1

int numLeafNodesInAsOne = bth.GetNLeafNodesForAs (1); //将服务器指定倒AS1的最后一个节点

server.Add (bth.GetLeafNodeForAs (1, numLeafNodesInAsOne - 1));

p2p.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));//设定点对点通道通信速率

p2p.SetChannelAttribute ("Delay", StringValue ("2ms"));//设定点对点通道时延

NetDeviceContainer p2pClientDevices;//创建两个device

NetDeviceContainer p2pServerDevices;

p2pClientDevices = p2p.Install (client);//将两个device安装在client和server两个节点

p2pServerDevices = p2p.Install (server);

address.SetBase ("10.1.0.0", "255.255.0.0");

Ipv4InterfaceContainer clientInterfaces;

clientInterfaces = address.Assign (p2pClientDevices); //为client设定IP地址

address.SetBase ("10.2.0.0", "255.255.0.0");

Ipv4InterfaceContainer serverInterfaces;

serverInterfaces = address.Assign (p2pServerDevices);//为server设定IP地址

UdpEchoServerHelper echoServer (9); //9为端口号

ApplicationContainer serverApps = echoServer.Install (server.Get (0));//为server0安装echoserver类型的应用

serverApps.Start (Seconds (1.0));

serverApps.Stop (Seconds (5.0));//设定起止时间

UdpEchoClientHelper echoClient (serverInterfaces.GetAddress (0), 9);//指定客户端将访问的服务器地址和端口号

echoClient.SetAttribute ("MaxPackets", UintegerValue (1)); //数据包个数

echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.))); //时间间隔

echoClient.SetAttribute ("PacketSize", UintegerValue (1024));//数据包长

ApplicationContainer clientApps = echoClient.Install (client.Get (0));

clientApps.Start (Seconds (2.0));

clientApps.Stop (Seconds (5.0));//将应用安装到客户端

if (!nix)

{

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();//生成全局路由表

}

if (tracing)

{

AsciiTraceHelper ascii;

p2p.EnableAsciiAll (ascii.CreateFileStream ("briteLeaves.tr"));

}//创建追踪文件

// Run the simulator

Simulator::Stop (Seconds (6.0));

Simulator::Run ();

Simulator::Destroy ();

return 0;

}

运行该程序，会弹出一个图形界面，点击右下方的Simulate按钮进行仿真，可以在terminal里看到相应的log。尝试采用其他配置文件运行程序，并观察结果。