2. 构建拓扑

2.1 构建总线网络拓扑

之前在 1. 一些配置模块 中，我们设计了P2P网络，在本章中将引入一个CSAM拓扑助手。
(/example/tutorial/second.cc)

```
// Default Network Topology
//
//       10.1.1.0
// n0 -------------- n1   n2   n3   n4
//    point-to-point  |    |    |    |
//                    ================
//                      LAN 10.1.2.0

//主程序的解释
bool verbose = true;
uint32_t nCsma = 3;  //额外节点的数量，因为nCsma必须有一个主节点和一个额外节点，这里有3个。

CommandLine cmd;   //设置cmd的输入：开启两个变量
cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

cmd.Parse (argc, argv);

if (verbose)
  {
    LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
  }

nCsma = nCsma == 0 ? 1 : nCsma; //设置为3个




//接下来就是创建节点
NodeContainer p2pNodes;   //创建p2pNode的容器
p2pNodes.Create (2);

NodeContainer csmaNodes;  //创建csmaNode的容器
csmaNodes.Add (p2pNodes.Get (1)); //这个点是同时拥有点对点和CSMA网络设备的节点
csmaNodes.Create (nCsma);



//接下来就是设计channel 和 device

/*P2P的channel和device*/
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
```

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

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);


/*Csma的channel和device*/
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);




//创建Node，Device，Channel之后，我们还没有协议栈，因此要用InternetStack Helper去安装堆栈：
InternetStackHelper stack;
stack.Install (p2pNodes.Get (0));
stack.Install (csmaNodes);
//Note：注意不要重复的覆盖



//分配IP地址
//p2p Network 的IP地址
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);

//csma Network 的IP地址
address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);



//建立应用程序
首先是server端：建立在csma网络的最后一个节点上
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));


然后是创建客户端：注意创建的点在
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9); //socket
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
```

```
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (p2pNodes.Get (0));  //安装在
p2p 的第一个点
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

2.1.1 ns-3对路由的选择：

我们这里介绍一个全局路由来帮助您：

基本上，每个节点的行为都像一个OSPF的路由器，在幕后立刻与所有路由器进行神奇的通信。每个节点生成链接广告，并将其传达给全局路由管理器，管理器用全局信息为每个节点构建路由表：

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

2.1.2 CSMA网络的追踪：

CSMA是一个多点对点网络。这意味着共享媒体可以有多个端点。每个端点都有一个与之关联的网络设备。如果想在这样的网络中搜集跟踪信息，基本有两种基本选择：

1. 为网络中的每个设备创建一个跟踪文件，并仅存储该网络设备发射或消耗的数据包。
2. 采用其中一个设备“滥交模式”。这个设备“嗅探”所有数据包的网络，并将其存储在单个pcap文件中。

2.1.3 对于模拟器的运行：

```
  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

2.1.4 构建模型

```
# cp examples/tutorial/second.cc scratch/mysecond.cc
# ./waf --run mysecond
```

2.1.5 设置日志级别

```
# export NS_LOG=""
# ./waf --run scratch/mysecond
```

2.1.6 输出结果

```
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.0078s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.0078s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.01761s client received 1024 bytes from 10.1.2.4 port 9
```

2.1.7 查看追踪目录（在顶级目录）

```
second-0-0.pcap  second-1-0.pcap  second-2-0.pcap
<name>-<node>-<device>.pcap
```

（1）用tcpdump去查看最左边p2p的跟踪文件

```
# tcpdump -nn -tt -r second-0-0.pcap

reading from file second-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.017607 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

```
# tcpdump -nn -tt -r second-1-0.pcap

reading from file second-1-0.pcap, link-type PPP (PPP)
2.003686 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.013921 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

```
# tcpdump -nn -tt -r second-2-0.pcap  //Node-2-device-0 是其中一个额外节点，下面都是额外节点Node2嗅探到的

reading from file second-2-0.pcap, link-type EN10MB (Ethernet)
2.007698 ARP, Request who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
//10.1.2.1 发送ARP探测包问（广播）谁是10.1.2.4，请回话

2.007710 ARP, Reply 10.1.2.4 is-at 00:00:00:00:00:06, length 50
//嗅探到10.1.2.4的回复包，我的MAC地址是00:00:00:00:00:06

2.007803 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
//10.1.1.1知道了地址然后发生传送

2.013815 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4, length 50
//返回的时候10.1.2.1问谁是10.1.2.4，给我MAC地址

2.013828 ARP, Reply 10.1.2.1 is-at 00:00:00:00:00:03, length 50
//10.1.2.1的MAC地址是00:00:00:00:00:03

2.013921 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
//知道路由进行传送
```

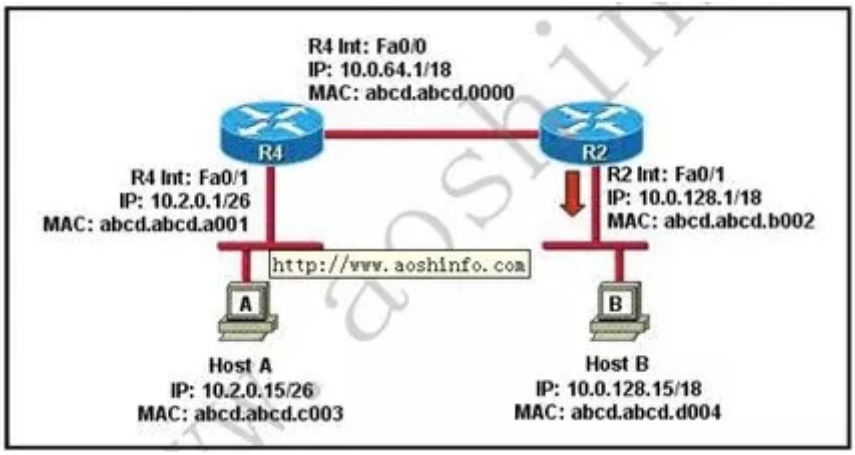
当nCsma多的时候，可以用GetID的方法去寻找节点的ID

```
pointToPoint.EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
csma.EnablePcap ("second", csmaNodes.Get (nCsma)->GetId (), 0, false);
csma.EnablePcap ("second", csmaNodes.Get (nCsma-1)->GetId (), 0, false);
```

也可以动态修改节点的数量

```
# ./waf --run "scratch/mysecond --nCsma=100"
```

这里我还科普了一下什么是ARP协议



这个怎么说呢？
1. 首先我画个图吧，就上面这个图。
这个图能说明：
(1) 每个路由器都有多个IP地址和多个MAC，每个IP对应一个MAC。
(2) 每个路由器都知道自己的IP地址和MAC地址
(3) 每个路由器都要尽量知道对方的IP地址。(路由advertisement)
(4) 就算知道对方的IP，也需要对方的MAC，因为查找是通过MAC包装确认的，所以要知道目的，不知道目的MAC是不行的。

所以就出现了ARP。

2.2 模型、属性和现实

这一段落说的是： 现实有更多的参数需要设置，但是模拟只是理想值，擅于发现现实和模型的属性差距，是模拟中比较重要的环节。

2.3 构建无线网络拓扑（WiFi网络）

示例脚本在/example/tutorial 的second.cc：

2.3.1 建设网络拓扑

```
// Default Network Topology
//
//   Wifi 10.1.3.0
//                 AP
//  *    *    *    *
//  |    |    |    |    10.1.1.0
// n5   n6   n7   n0 -------------- n1   n2   n3   n4
//                   point-to-point  |    |    |    |
//                                   ================
//
```

1. 首先添加一些模块

```
#include "ns3/core-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-module.h"
```

2. 然后是命名空间和日志

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("ThirdScriptExample");
```

3. 然后是设置日志开关和nCSMA，nWiFi个数

```
bool verbose = true;
uint32_t nCsma = 3;
uint32_t nWifi = 3;

CommandLine cmd;
cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

cmd.Parse (argc,argv);

if (verbose)
  {
    LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
  }
```

4. 首先是p2p节点的创建，p2p的channel，p2p的网卡device

```
NodeContainer p2pNodes;
p2pNodes.Create (2);

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

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

然后是CSMA网络的节点，CSMA的channel，CSMA的网卡device

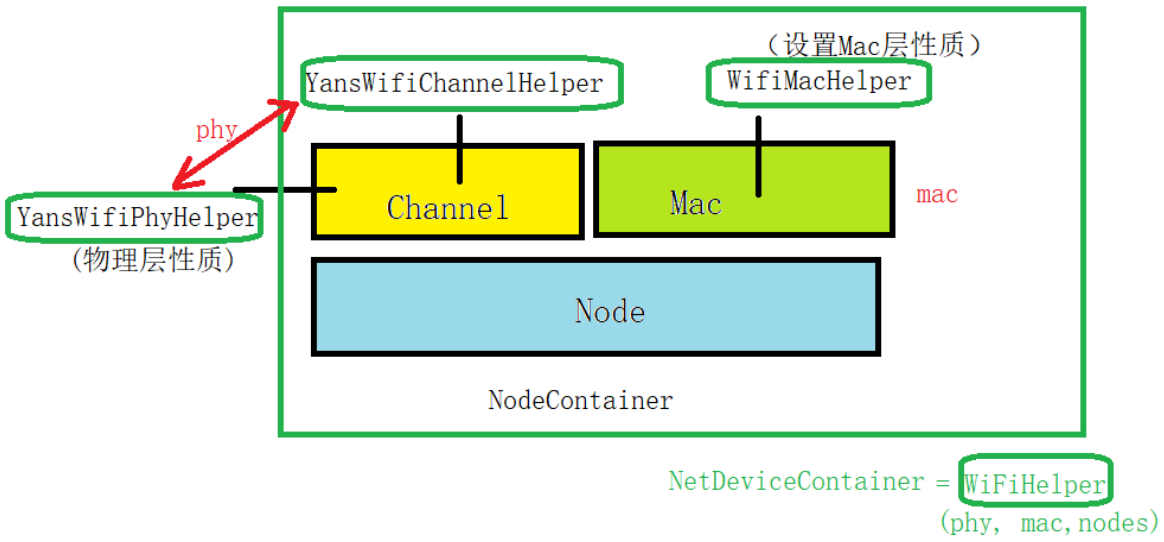
```
NodeContainer csmaNodes;
csmaNodes.Add (p2pNodes.Get (1));
csmaNodes.Create (nCsma);

CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);
```

最后是WiFi的节点，WiFi的channel，WiFi的网卡device

wifi的channel，一共需要四个helper去做： YansWifiChannelHelper, YansWifiPhyHelper,WifiMacHelper, WifiHelper:

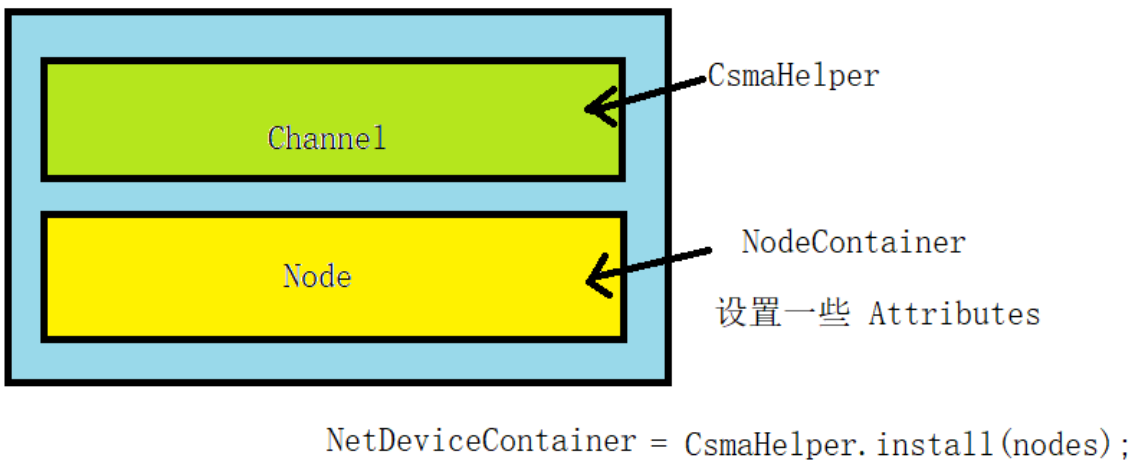
```
//1.NodeContainer
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi);
NodeContainer wifiApNode = p2pNodes.Get (0);  //选一个基站


//2. phy物理层的设置
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper phy = YansWifiPhyHelper::Default (); //默认的物理层性质
phy.SetChannel (channel.Create ());  //组合(完成了黄色的一步)


//3. mac层的设置
3. WifiMacHelper mac;     //配置Mac层参数
Ssid ssid = Ssid ("ns-3-ssid"); //创建一个802.11的服务标识符SSID对象，这个对象用于设置MAc层实现的“Ssid 属性”的值。

Notes: 这里解释一下什么是Ssid?  就是WiFi的名字

mac.SetType ("ns3::StaWifiMac", //代表创建的类型是“StaWifiMac”
  "Ssid", SsidValue (ssid), //ssid性质的确定
```

```
  "ActiveProbing", BooleanValue (false)); //主动进行中的性质设置成false，代表创建的mac不会发送探针请求，并且电台将监听AP信标。



//4. 然后就是用WiFiHelper连接起来
WifiHelper wifi;
NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes);  //看好这是设定的WiFi节点而不是AP


//5. 我们也要为AP设置mac协议，并且组装
mac.SetType ("ns3::ApWifiMac",
             "Ssid", SsidValue (ssid));

NetDeviceContainer apDevices;
apDevices = wifi.Install (phy, mac, wifiApNode);  //这里才是设置AP节点
```



到此还没有设定移动模型，但是发现和CSMA网络的设定是有区别的，说明IEEE802.11 和CSMA是有区别，尤其是在Mac层上：

1. CSMA在MAC层上无需指明嗅探，或者名称。 但是Wifi需要：需要指明是否需要嗅探的过程，并且指明Ssid。



2.3.2 构建移动模型

我们将staWifiNode设置成移动模型，但是AP是静止的：

```
MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator", //位置分配器
  "MinX", DoubleValue (0.0),
  "MinY", DoubleValue (0.0),
  "DeltaX", DoubleValue (5.0),
  "DeltaY", DoubleValue (10.0),
  "GridWidth", UintegerValue (3),
  "LayoutType", StringValue ("RowFirst"));


// 这些代码告诉移动辅助器使用二维网格来放置STA节点
```

```
/* 然后我们将告诉节点如何移动，采用的是RandomWalk的MobilityModel模型
安排我们的Node在初始网格中，然后让node以随机的移动方向和随机的速度进行移动，在这个二维网格内*/

mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
  "Bounds", RectangleValue (Rectangle (-50, 50, -50, 50)));  //设置移动模型和边界

mobility.Install (wifiStaNodes); //移动模型的安装
```

```
//AP 安装的是静止模型
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNode);
```

然后就是正常网络层设置了.....

```
//设置Ip地址:
Ipv4AddressHelper address;

address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);

address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);

address.SetBase ("10.1.3.0", "255.255.255.0");
address.Assign (staDevices);
address.Assign (apDevices);
```

然后就是应用层的设置了.....，同样也是安装echo服务器

```
//服务器安装在AP上
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma)); //
服务安装的参数是Node，这里的服务器我们设置在CSMA的最后一个节点上
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));

//客户端安装在sta的最后一个节点上
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps =echoClient.Install (wifiStaNodes.Get (nWifi -
1));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables (); //我们建立了一个互联网络，所以要建
立互联网络的路由，像我们如果建立单个的网络，那就不需要这步，就像first.cc就没有这句话
```

```
//Beacon信标帧，由AP以一定时间间隔发出的，用来告诉StaNode自己的网络是存在的
//因为这个原因，那么模拟器就永远不会停止，这个时候我们要手动设置模拟器的停止时间
Simulator::Stop (Seconds (10.0));
```

设置Pcap的追踪

```
pointToPoint.EnablePcapAll ("third");
phy.EnablePcap ("third", apDevices.Get (0)); //在AP上设置追踪
csma.EnablePcap ("third", csmaDevices.Get (0), true); //在Csma的第一个点上设置追踪

//这个追踪都是谁发出来的？都是物理层的Helper对象生成的组件可以设置EnablePcap
```

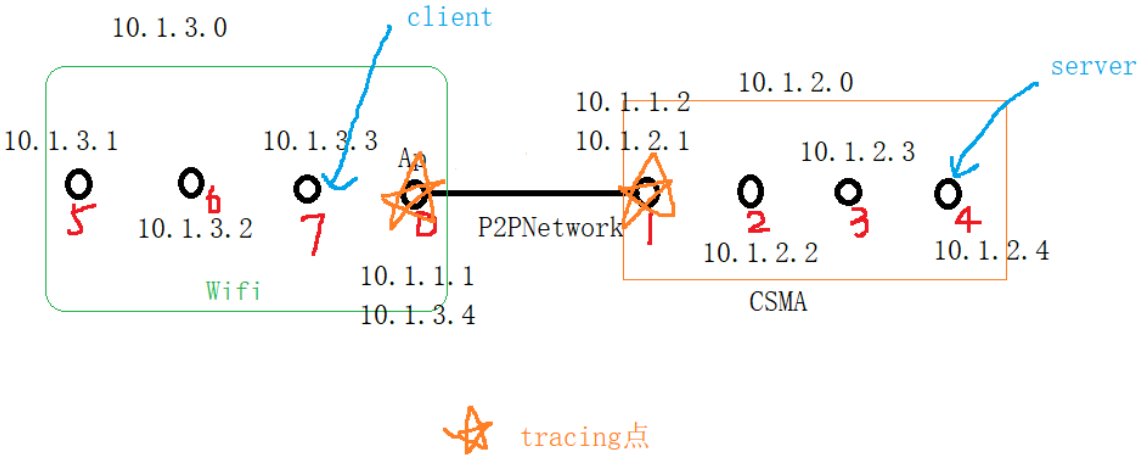
```
  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

构建脚本：

```
$ cp examples/tutorial/third.cc scratch/mythird.cc
$ ./waf --run 'scratch/mythird --tracing=1'
```

2.3.3 分析输出

```
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.01624s server received 1024 bytes from 10.1.3.3 port 49153
At time +2.01624s server sent 1024 bytes to 10.1.3.3 port 49153
At time +2.02849s client received 1024 bytes from 10.1.2.4 port 9
```



通过顶级目录中的pcap去查看跟踪，发现有四个跟踪点，这个比较好理解

```
third-0-0.pcap(Node0 的p2p网络设备)  third-0-1.pcap (Node0 的WiFi网络设备)

third-1-0.pcap(Node1 的p2p网络设备)   third-1-1.pcap (Node1 的CSMA网络设备)
```

```
$ tcpdump -nn -tt -r third-0-1.pcap #查看一下捕捉文件

#这里要看懂，可能要去复习一下WiFi传输的细节，网页中展示了四个.pcap 的追踪文件。我发现了如下的不同：
#1. CSMA用的是ARP协议，但是P2P却能直接发现这个网络的传送，WiFi网络是用Beacon嗅探的方式
#2. Wifi 网络的嗅探过程是先 Beacon ->Assoc Request -> 然后是Acknowledgment -> CF-End -> Successful 可以注意一下时间节点。

reading from file third-0-1.pcap, link-type IEEE802_11_RADIO (802.11 plus radiotap header)

0.033119 33119us tsft 6.0 Mb/s 5210 MHz 11a Beacon (ns-3-ssid) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS #嗅探包

0.120504 120504us tsft 6.0 Mb/s 5210 MHz 11a -62dBm signal -94dBm noise Assoc Request (ns-3-ssid) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]

0.120520 120520us tsft 6.0 Mb/s 5210 MHz 11a Acknowledgment RA:00:00:00:00:00:08

0.120632 120632us tsft 6.0 Mb/s 5210 MHz 11a -62dBm signal -94dBm noise CF-End RA:ff:ff:ff:ff:ff:ff
```

```
0.120666 120666us tsft 6.0 Mb/s 5210 MHz 11a Assoc Response AID(1) :: Successful
...
```

2.3.4 关于打印mobility信息的方法

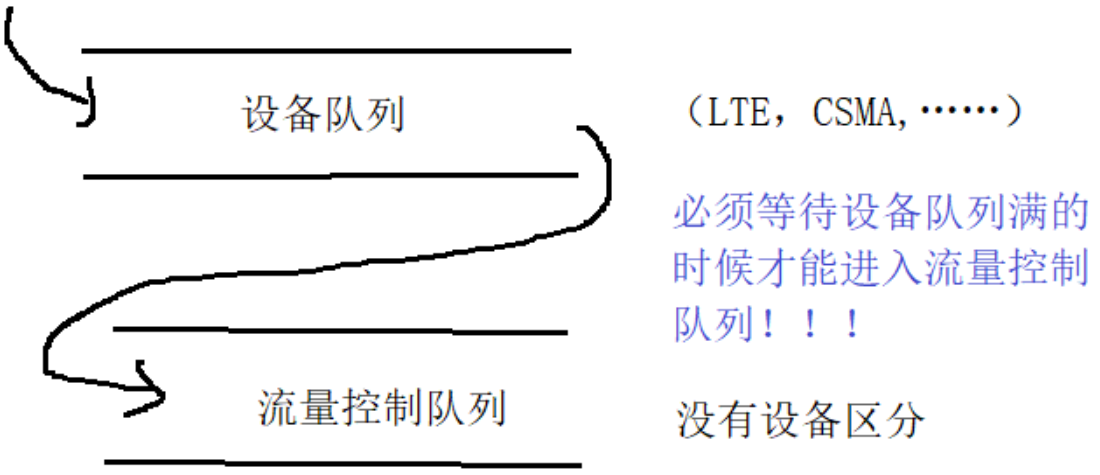
```
//设置追踪源(tracing source)和接受器(tracing sink)就行了，并且我们也提供方式去连接二者。现在我们就要将source和trace event连接。我们自己写一个接收器，去展示tracing source的一些信息。看起来很难，实际还是很简单的。

//1. 首先在程序中加入这个接收器
void
CourseChange (std::string context, Ptr<const MobilityModel> model)// context 代表日志，model指针表示对应的model对象
{
  Vector position = model->GetPosition (); //向量类型接受移动的位置
  NS_LOG_UNCOND (context <<
    " x = " << position.x << ", y = " << position.y); //往日志里写
}


//2.绑定连接器和接收器： 上述只能写在日志里，但是我们希望每次安装client的StaNode节点移动的时候，这个function都会自动的被调用。我们使用Connet这个函数去实现功能，在Simulator::Run之前写这两行代码：

std::ostringstream oss; //输出流
oss <<
  "/NodeList/" << wifiStaNodes.Get (nWifi - 1)->GetId () <<
  "/$ns3::MobilityModel/CourseChange";  //单纯的打印信息

Config::Connect (oss.str (), MakeCallback (&CourseChange)); //每一个tracing source更改时间都会回调CourseChange函数。在调用函数之前打印oss的信息。
```

2.4 ns-3中的队列

从架构上来说，ns3具有两个队列：1. “流量控制层”，在这里活动队列管理 和 服务质量 的优先级是通过排队纪律，和设备无关（软件队列）。2. 队列层通常可以通过NetDevice找到，不同的队列（LTE，WiFI）对这些队列有不同的实现（硬件队列）。「二者都来自NetDevice，只是有些NetDevice不支持流量控制层」

在实际过程中，可能更加复杂，比如地址解析协议也有一个小队列，Linux中的WiFi有四层排队。

注意： 当设备队列已满的时候，流量控制才有效。



支持流控制、流量控制层能力的NetDevices，他们使用Queue对象来存储其数据包：

```
点对点
CSMA
Wi-Fi
SimpleNetDevice
```

NetDevice的队列大小对排队规则有很大的影响。但是ns3中的队列都是最简单的FiFO。然而，可以通过启用BQ（字节队列限制）来动态调整队列的大小。

（1）ns-3中可用的排队模式

对于流量控制层而言：

```
在流量控制层有以下几个选项：
PFifoFastQueueDisc：默认最大大小为1000个数据包
FifoQueueDisc：默认最大大小为1000个数据包
RedQueueDisc：默认最大大小为25个数据包
CoDelQueueDisc：默认最大大小为1500千字节
FqCoDelQueueDisc：默认最大大小为10240个数据包
PieQueueDisc：默认最大大小为25个数据包
MqQueueDisc：无限制
TbfQueueDisc：默认最大大小为1000个数据包
```

一般来说，NetDevice都默认会安装PFifoFastQueueDisc队列纪律「具有优先级的FIFO」

对于设备的队列而言：

```
PointToPointNetDevice：默认（或者由helper）是安装默认大小为DropTail队列（100个数据包）
CsmaNetDevice：默认安装默认大小为DropTail队列（100个数据包）
WiFiNetDevice：默认配置为非QoS站点安装的默认大小为100个数据包的DropTail队列，为QoS站点安装四个默认大小的DropTail队列（100个数据包）。
SimpleNetDevice：默认配置安装大小为100个的DropTail
LTENetDevice：发生在RLC层，默认缓冲区为10*1024字节
UanNetDevice：MAC层有一个默认的10个数据包队列
```

(2) 从默认值更改

对于流量控制层的修改：BQL

```
InternetStackHelper stack;
stack.Install (nodes);

TrafficControlHelper tch; //流量控制助手
tch.SetRootQueueDisc ("ns3::CoDelQueueDisc", "MaxSize", StringValue ("1000p"));
tch.SetQueueLimits ("ns3::DynamicQueueLimits", "HoldTime", StringValue ("4ms"));
tch.Install (devices);
```

对于设备队列的修改：

```
NodeContainer nodes;
nodes.Create (2);

PointToPointHelper p2p;
p2p.SetQueue ("ns3::DropTailQueue", "MaxSize", StringValue ("50p")); //设备队列的修改

NetDeviceContainer devices = p2p.Install (nodes);
```

对于整体流程的修改：

```
NodeContainer nodes;
nodes.Create (2);

PointToPointHelper p2p;
p2p.SetQueue ("ns3::DropTailQueue", "MaxSize", StringValue ("50p"));

NetDeviceContainer devices = p2p.Install (nodes);

InternetStackHelper stack;
stack.Install (nodes);

TrafficControlHelper tch;
tch.SetRootQueueDisc ("ns3::CoDelQueueDisc", "MaxSize", StringValue ("1000p"));
tch.Install (devices);
```