```
#include "scenario.h"

using namespace std;
using namespace ns3;

NS_LOG_COMPONENT_DEFINE("OpenGym");

void installTrafficGenerator(Ptr<ns3::Node> fromNode, Ptr<ns3::Node> toNode, int
port, string offeredLoad, double startTime);
void PopulateARPcache();
void recordHistory();

double envStepTime = 0.1;
double simulationTime = 10; //seconds
double current_time = 0.0;
bool verbose = false;
int end_delay = 0;
bool dry_run = false;

Ptr<FlowMonitor> monitor;
FlowMonitorHelper flowmon;
ofstream outfile ("scratch/linear-mesh/CW_data.csv", fstream::out);

uint32_t CW = 0;

// Our data structure for the scenario needs to be a vector of [history_length,
2]
uint32_t history_length = 20;
deque<float> history;

string type = "discrete";
bool non_zero_start = false;
Scenario *vanetSceanrio;

static void
CourseChange (std::string context, 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;
}

/*
Define observation space
*/
Ptr<OpenGymSpace> MyGetObservationSpace(void)
{
    float low = 0.0;
    float high = 10.0;
    std::vector<uint32_t> shape = {
        history_length,
    };
```

```
    std::string dtype = TypeNameGet<float>();
    Ptr<OpenGymBoxSpace> space = CreateObject<OpenGymBoxSpace>(low, high, shape,
dtype);
    if (verbose)
        NS_LOG_UNCOND("MyGetObservationSpace: " << space);
    return space;
}

/*
Define action space
*/
Ptr<OpenGymSpace> MyGetActionSpace(void)  //返回MyGetActionSpace指向OpenGymSpace对
象的指针
{
    float low = 0.0;
    float high = 10.0;
    std::vector<uint32_t> shape = {
        1,
    };
    std::string dtype = TypeNameGet<float>(); //这行代码的作用是将 float 的类型名称以
字符串的形式存储到 dtype 变量中。
    Ptr<OpenGymBoxSpace> space = CreateObject<OpenGymBoxSpace>(low, high, shape,
dtype);
    if (verbose)
        NS_LOG_UNCOND("MyGetActionSpace: " << space);
    return space;
}

/*
Define extra info. Optional
*/
uint64_t g_rxPktNum = 0;
uint64_t g_txPktNum = 0;

double jain_index(void)
{
    double flowThr;
    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>
(flowmon.GetClassifier());
    std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();

    double nominator;
    double denominator;
    double n=0;
    double station_id = 0;
    for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
stats.begin(); i != stats.end(); ++i)
    {
        flowThr = i->second.rxBytes;
        flowThr /= vanetSceanrio->getStationUptime(station_id, current_time);
        if(flowThr>0){
            nominator += flowThr;
            denominator += flowThr*flowThr;
            n++;
        }
        station_id++;
    }
    nominator *= nominator;
```

```
    denominator *= n;
    return nominator/denominator;
}

std::string MyGetExtraInfo(void)
{
    static float ticks = 0.0;
    static float lastValue = 0.0;
    float obs = g_rxPktNum - lastValue;
    lastValue = g_rxPktNum;
    ticks += envStepTime;

    float sentMbytes = obs * (1500 - 20 - 8 - 8) * 8.0 / 1024 / 1024;

    std::string myInfo = std::to_string(sentMbytes);
    myInfo = myInfo + "|" + to_string(CW) + "|";
    myInfo = myInfo + to_string(vanetSceanrio->getActiveStationCount(ticks)) +
"|";
    myInfo = myInfo + to_string(jain_index());

    if (verbose)
        NS_LOG_UNCOND("MyGetExtraInfo: " << myInfo);

    return myInfo;
}

/*
Execute received actions
*/
bool MyExecuteActions(Ptr<OpenGymDataContainer> action)
{
    if (verbose)
        NS_LOG_UNCOND("MyExecuteActions: " << action);

    Ptr<OpenGymBoxContainer<float>> box = DynamicCast<OpenGymBoxContainer<float>>
(action);
    std::vector<float> actionVector = box->GetData();

    if (type == "discrete")
    {
        CW = pow(2, 4+actionVector.at(0));
    }
    else if (type == "continuous")
    {
        CW = pow(2, actionVector.at(0) + 4);
    }
    else if (type == "direct_continuous")
    {
        CW = actionVector.at(0);
    }
    else
    {
        std::cout << "Unsupported agent type!" << endl;
        exit(0);
    }

    uint32_t min_cw = 16;
    uint32_t max_cw = 1024;
```

```
    CW = min(max_cw, max(CW, min_cw));
    outfile << current_time << "," << CW << endl;

    if(!dry_run){

 Config::Set("/$ns3::NodeListPriv/NodeList/*/$ns3::Node/DeviceList/*/$ns3::WifiNe
tDevice/Mac/$ns3::RegularWifiMac/BE_Txop/$ns3::QosTxop/MinCw",
UintegerValue(CW));

 Config::Set("/$ns3::NodeListPriv/NodeList/*/$ns3::Node/DeviceList/*/$ns3::WifiNe
tDevice/Mac/$ns3::RegularWifiMac/BE_Txop/$ns3::QosTxop/MaxCw",
UintegerValue(CW));
    }
    return true;
}

float MyGetReward(void)
{
    static float ticks = 0.0;
    static uint32_t last_packets = 0;
    static float last_reward = 0.0;
    ticks += envStepTime;

    float res = g_rxPktNum - last_packets;
    float reward = res * (1500 - 20 - 8 - 8) * 8.0 / 1024 / 1024 / (5 * 150 *
envStepTime) * 10;

    last_packets = g_rxPktNum;

    if (ticks <= 2 * envStepTime)
        return 0.0;

    if (verbose)
        NS_LOG_UNCOND("MyGetReward: " << reward);

    if(reward>1.0f || reward<0.0f)
        reward = last_reward;
    last_reward = reward;
    return last_reward;
}

/*
Collect observations
*/
Ptr<OpenGymDataContainer> MyGetObservation()
{
    recordHistory();

    std::vector<uint32_t> shape = {
        history_length,
    };
    Ptr<OpenGymBoxContainer<float>> box =
CreateObject<OpenGymBoxContainer<float>>(shape);

    for (uint32_t i = 0; i < history.size(); i++)
    {
        if (history[i] >= -100 && history[i] <= 100)
```

```
            box->AddValue(history[i]);
        else
            box->AddValue(0);
    }
    for (uint32_t i = history.size(); i < history_length; i++)
    {
        box->AddValue(0);
    }
    if (verbose)
        NS_LOG_UNCOND("MyGetObservation: " << box);
    return box;
}

bool MyGetGameOver(void)
{
    // bool isGameOver = (ns3::Simulator::Now().GetSeconds() > simulationTime +
end_delay + 1.0);
    return false;
}

void ScheduleNextStateRead(double envStepTime, Ptr<OpenGymInterface>
openGymInterface)
{
    Simulator::Schedule(Seconds(envStepTime), &ScheduleNextStateRead,
envStepTime, openGymInterface);
    openGymInterface->NotifyCurrentState();
}

void recordHistory()
{
    // Keep track of the observations
    // We will define them as the error rate of the last `envStepTime` seconds
    static uint32_t last_rx = 0;            // Previously received packets
    static uint32_t last_tx = 0;            // Previously transmitted packets
    static uint32_t calls = 0;              // Number of calls to this function
    calls++;
    current_time += envStepTime;    //+0.1

    float received = g_rxPktNum - last_rx;  // Received packets since the last
observation
    float sent = g_txPktNum - last_tx;      // Sent (...)
    float errs = sent - received;           // Errors (...)
    float ratio;

    ratio = errs / sent;
    history.push_front(ratio);

    // Remove the oldest observation if we have filled the history
    if (history.size() > history_length)
    {
        history.pop_back();
    }

    // Replace the last observation with the current one
    last_rx = g_rxPktNum;
    last_tx = g_txPktNum;

    if (calls < history_length && non_zero_start)
```

```
    {
        // Schedule the next observation if we are not at the end of the
simulation
        Simulator::Schedule(Seconds(envStepTime), &recordHistory);
    }
    else if (calls == history_length && non_zero_start)
    {
        g_rxPktNum = 0;
        g_txPktNum = 0;
        last_rx = 0;
        last_tx = 0;
    }
}

void packetReceived(Ptr<const Packet> packet)
{
    NS_LOG_DEBUG("Client received a packet of " << packet->GetSize() << "
bytes");
    g_rxPktNum++;
}

void packetSent(Ptr<const Packet> packet)
{
    g_txPktNum++;
}

void set_phy(int nRSU, int nVehicle, int guardInterval, NodeContainer &rsuNode,
NodeContainer &vehicularNode, YansWifiPhyHelper &phy)
{
    Ptr<MatrixPropagationLossModel> lossModel =
CreateObject<MatrixPropagationLossModel>();
    lossModel->SetDefaultLoss(50);

    rsuNode.Create(nRSU);
    vehicularNode.Create(nVehicle);

    YansWifiChannelHelper channel = YansWifiChannelHelper::Default();
    Ptr<YansWifiChannel> chan = channel.Create();
    chan->SetPropagationLossModel(lossModel);
    chan-
>SetPropagationDelayModel(CreateObject<ConstantSpeedPropagationDelayModel>());

    phy = YansWifiPhyHelper::Default();
    phy.SetChannel(chan);

    // Set guard interval
    phy.Set("GuardInterval", TimeValue(NanoSeconds(guardInterval)));
}

void set_nodes(int channelWidth, int rng, int32_t simSeed, NodeContainer rsuNode,
NodeContainer vehicularNode, YansWifiPhyHelper phy, WifiMacHelper mac, WifiHelper
wifi, NetDeviceContainer &rsuDevice)
{
    // Set the access point details
    //Ssid ssid = Ssid("ns3-80211ax");

    //mac.SetType("ns3::StaWifiMac",
                //"Ssid", SsidValue(ssid),
```

```
                //"ActiveProbing", BooleanValue(false),
                //"BE_MaxAmpduSize", UintegerValue(0));

   // NetDeviceContainer staDevice;
    //staDevice = wifi.Install(phy, mac, rsuNode);

    //mac.SetType("ns3::ApWifiMac",
                //"EnableBeaconJitter", BooleanValue(false),
                //"Ssid", SsidValue(ssid));
    NetDeviceContainer vehicularDevice;
    vehicularDevice = wifi.Install(phy, mac, vehicularNode);
    rsuDevice = wifi.Install(phy, mac, rsuNode);

    // Set channel width
    Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/ChannelWidth",
UintegerValue(channelWidth));

    // mobility.
    MobilityHelper mobility;  //给给定的节点添加移动模型
    mobility.SetPositionAllocator ("ns3::RandomDiscPositionAllocator",
                                   "X", StringValue ("100.0"),
                                   "Y", StringValue ("100.0"), //表示节点在以随机点为圆
心、以随机距离为半径的圆形区域内随机分布
                                   "Rho", StringValue
("ns3::UniformRandomVariable[Min=0|Max=30]"));
    mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
                               "Mode", StringValue ("Time"),
                               "Time", StringValue ("2s"),
                               "Speed", StringValue
("ns3::ConstantRandomVariable[Constant=1.0]"),
                               "Bounds", StringValue ("0|200|0|200"));


    mobility.Install(vehicularNode);  //为指定的节点安装移动模型

    Ptr<ListPositionAllocator> positionAlloc =
CreateObject<ListPositionAllocator>();//将一系列的向量坐标点保存到一个列表


    positionAlloc->Add(Vector(0.0, 0.0, 0.0));
    positionAlloc->Add(Vector(1.0, 0.0, 0.0));
    mobility.SetPositionAllocator(positionAlloc);
    mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel"); //不移动的模型
    mobility.Install(rsuNode);

    Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
                     MakeCallback (&CourseChange));
    /* Internet stack*/
    InternetStackHelper stack;   //安装网络协议
    stack.Install(vehicularNode);
    stack.Install(rsuNode);

    //Random
    if(simSeed!=-1)
        RngSeedManager::SetSeed(simSeed);  //设置随机种子
    RngSeedManager::SetRun(rng);

    Ipv4AddressHelper address;
```

```
    address.SetBase("192.168.1.0", "255.255.255.0"); //指定IP地址的基础部分，但是在Assign()方法中会根据网络拓扑和设备数量自动分配不同的IP地址给每个设备
    Ipv4InterfaceContainer staNodeInterface;
    Ipv4InterfaceContainer apNodeInterface;

    staNodeInterface = address.Assign(vehicularDevice);
    apNodeInterface = address.Assign(rsuDevice);

    if (!dry_run)   //是否使用反向退避算法
    {

 Config::Set("/$ns3::NodeListPriv/NodeList/*/$ns3::Node/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_Txop/$ns3::QosTxop/MinCw", UintegerValue(CW));  //处理数据和管理帧的数据包分段和重传。此类为数据和管理帧实现数据包分段和重传策略。它使用ns3::ChannelAccessManager辅助类来决定何时发送数据包。数据包存储在ns3::WifiMacQueue中，直到它们可以被发送。当前实施的策略使用一个简单的分段阈值：任何大于此阈值的数据包都被分成大小小于阈值的片段。重传策略也非常简单：重传每个数据包，直到成功传输或重传达到 SSRC 或 SLRC 阈值为止。



 Config::Set("/$ns3::NodeListPriv/NodeList/*/$ns3::Node/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_Txop/$ns3::QosTxop/MaxCw", UintegerValue(CW));
    }
    else
    {
        NS_LOG_UNCOND("Default CW");  // QoS 数据帧的数据包分段和重传以及 MSDU 聚合 (A-MSDU) 和块确认会话。

 Config::Set("/$ns3::NodeListPriv/NodeList/*/$ns3::Node/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_Txop/$ns3::QosTxop/MinCw", UintegerValue(16));

 Config::Set("/$ns3::NodeListPriv/NodeList/*/$ns3::Node/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_Txop/$ns3::QosTxop/MaxCw", UintegerValue(1024));
    }
}

//set_sim(tracing, dry_run, warmup"1", openGymPort, phy, rsuDevice, end_delay, monitor, flowmon);
void set_sim(bool tracing, bool dry_run, int warmup, uint32_t openGymPort, YansWifiPhyHelper phy, NetDeviceContainer rsuDevice, int end_delay, Ptr<FlowMonitor> &monitor, FlowMonitorHelper &flowmon)
{
    monitor = flowmon.InstallAll();
    monitor->SetAttribute("StartTime", TimeValue(Seconds(warmup)));   //在所有节点上安装流量监视器并将开始时间设置为指定的预热时间

    if (tracing)  //如果启用跟踪，则将pcap数据链路类型设置为DLT_IEEE802_11_RADIO，并启用 pcap 输出到容器中第一个设备的名为“cw”的文件rsuDevice
    {
        phy.SetPcapDataLinkType(WifiPhyHelper::DLT_IEEE802_11_RADIO);
        phy.EnablePcap("cw", rsuDevice.Get(0));
    }
```

```
    Ptr<OpenGymInterface> openGymInterface = CreateObject<OpenGymInterface>
(openGymPort);//创建opengyminterface具有指定端口号的对象并设置多个回调函数以用于与
openaigym环境交互
    openGymInterface->SetGetActionSpaceCb(MakeCallback(&MyGetActionSpace));
    openGymInterface-
>SetGetObservationSpaceCb(MakeCallback(&MyGetObservationSpace));
    openGymInterface->SetGetGameOverCb(MakeCallback(&MyGetGameOver));
    openGymInterface->SetGetObservationCb(MakeCallback(&MyGetObservation));
    openGymInterface->SetGetRewardCb(MakeCallback(&MyGetReward));
    openGymInterface->SetGetExtraInfoCb(MakeCallback(&MyGetExtraInfo));
    openGymInterface->SetExecuteActionsCb(MakeCallback(&MyExecuteActions));

    // if (!dry_run)
    // {
    if (non_zero_start)
    {
        Simulator::Schedule(Seconds(1.0), &recordHistory);
        Simulator::Schedule(Seconds(envStepTime * history_length + 1.0),
&ScheduleNextStateRead, envStepTime, openGymInterface);
    }
    else
        Simulator::Schedule(Seconds(1.0), &ScheduleNextStateRead, envStepTime,
openGymInterface);
    // }

    Simulator::Stop(Seconds(simulationTime + end_delay + 1.0 + envStepTime*
(history_length+1)));

    NS_LOG_UNCOND("Simulation started");
    Simulator::Run();
}

void signalHandler(int signum)
{
    cout << "Interrupt signal " << signum << " received.\n";
    exit(signum);
}



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

    std::cout<<"(############## Start ns-3 simulation "<<std::endl;
    int nRSU = 5;   //5个RSU
    int nVehicle=5;  //5辆车
    bool tracing = false;
    //bool useRts = false;
    //int mcs = 11;
    int channelWidth = 20;
    int guardInterval = 800; //设置无线物理层的保护间隔时间，即发送数据包前要等待的一段时
间，以避免数据包的前导码和数据部分重叠，导致数据包的丢失。
//guardInterval 参数指定了保护间隔的时间，以纳秒为单位，
    string offeredLoad = "150";
    int port = 1025;
    string outputCsv = "cw.csv";
    string scenario = "basic";
    dry_run = false;
```

```
    int rng = 1;
    int warmup = 1;

    uint32_t openGymPort = 5555;
    int32_t simSeed = -1;

    signal(SIGTERM, signalHandler);     //SIGTERM 15  以信号命名的信号处理函数signalHandler，SIGTERM信号编号••为 15。
    outfile << "SimulationTime,CW" << endl;  //C++ 中表示输出文件流类型的变量。输出文件流用于SimulationTime,CW写入文件

    CommandLine cmd;
    cmd.AddValue("openGymPort", "Specify port number. Default: 5555", openGymPort);
    cmd.AddValue("CW", "Value of Contention Window", CW);
    cmd.AddValue("historyLength", "Length of history window", history_length);
    cmd.AddValue("nRSU", "Number of wifi 802.11p RSU devices", nRSU);
    cmd.AddValue("nVehicle", "Number of wifi 802.11p vechicle nodes", nVehicle);
    cmd.AddValue("verbose", "Tell echo applications to log if true", verbose); //告诉echo应用程序记录是否为真
    cmd.AddValue("tracing", "Enable pcap tracing", tracing);
    cmd.AddValue("rng", "Number of RngRun", rng);   //模拟中使用的随机数生成器的数量
    cmd.AddValue("simTime", "Simulation time in seconds. Default: 10s", simulationTime);
    cmd.AddValue("envStepTime", "Step time in seconds. Default: 0.1s", envStepTime);
    cmd.AddValue("agentType", "Type of agent actions: discrete, continuous", type);   //智能体采取的行动的类型，离散的或是连续的
    cmd.AddValue("nonZeroStart", "Start only after history buffer is filled", non_zero_start); //历史记录缓冲区被填满之后才开始执行
    cmd.AddValue("scenario", "Scenario for analysis: basic, convergence, reaction", scenario);  //基本、收敛和反应场景
    cmd.AddValue("dryRun", "Execute scenario with BEB and no agent interaction", dry_run);  //使用BEB执行场景且无代理交互
    cmd.AddValue("seed", "Random seed", simSeed); //指定随机种子

    cmd.Parse(argc, argv);
    // history_length*=2;

    NS_LOG_UNCOND("Ns3Env parameters:");
    NS_LOG_UNCOND("--nRSU: " << nRSU);
    NS_LOG_UNCOND("--nVehicle: " << nVehicle);
    NS_LOG_UNCOND("--simulationTime: " << simulationTime);
    NS_LOG_UNCOND("--openGymPort: " << openGymPort);
    NS_LOG_UNCOND("--envStepTime: " << envStepTime);
    NS_LOG_UNCOND("--seed: " << simSeed);
    NS_LOG_UNCOND("--agentType: " << type);
    NS_LOG_UNCOND("--scenario: " << scenario);
    NS_LOG_UNCOND("--dryRun: " << dry_run);

    if (verbose)//false
    {
        LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO); //预先定义的日志组件，只输出信息级别以上的信息
        LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
    }
```

```
    //if (useRts)
    //{
        //Config::SetDefault("ns3::WifiRemoteStationManager::RtsCtsThreshold",
StringValue("0"));
    //}

    Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Mode", StringValue
("Time")); //设置随机游走2D模型模式为“TIME”，表示节点将在指定的时间间隔进行随机游走
    Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Time", StringValue
("2s")); //设置随机游走 2D 移动模型的时间间隔为 2 秒
    Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Speed", StringValue
("ns3::ConstantRandomVariable[Constant=1.0]")); //随机游走 2D 移动模型的速度为常数 1.0
    Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Bounds", StringValue
("0|200|0|200"));//设置随机2D移动模型的便捷是0到200的矩形，长宽都是200

    NodeContainer rsuNode;  //创造一个名为rsuNode的节点容器
    NodeContainer vehicularNode;   //创造一个vehicularNode的节点容器
    YansWifiPhyHelper phy;  //用于描述WiFi物理层的行为。它是ns-3中用于WiFi网络仿真的物理
层模型之一。
    set_phy(nRSU, nVehicle, guardInterval, rsuNode, vehicularNode, phy);

    NqosWaveMacHelper mac= NqosWaveMacHelper::Default (); //配置Wave网络的MAC层的帮
助程序
    Wifi80211pHelper wifi= Wifi80211pHelper::Default (); //配置Wave网络的物理层的帮助
程序

    std::string phyMode ("OfdmRate6MbpsBW10MHz");  //定义一个字符串类型的变量
phyMode，表示 Wi-Fi 设备的工作模式
    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                                  "DataMode",StringValue (phyMode),
                                  "ControlMode",StringValue (phyMode));


    //wifi.SetStandard(WIFI_PHY_STANDARD_80211ax_5GHZ);

    //std::ostringstream oss;
    //oss << "HeMcs" << mcs;
    //wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager", "DataMode",
StringValue(oss.str()),
                                 //"ControlMode", StringValue(oss.str()));

    //802.11ac PHY
    /*
  phy.Set ("ShortGuardEnabled", BooleanValue (0));
  wifi.SetStandard (WIFI_PHY_STANDARD_80211ac);
  wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
  "DataMode", StringValue ("VhtMcs8"),
  "ControlMode", StringValue ("VhtMcs8"));
 */
    //802.11n PHY
    //phy.Set ("ShortGuardEnabled", BooleanValue (1));
    //wifi.SetStandard (WIFI_PHY_STANDARD_80211n_5GHZ);
    //wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
    //                              "DataMode", StringValue ("HtMcs7"),
    //                              "ControlMode", StringValue ("HtMcs7"));

    NetDeviceContainer rsuDevice;
```

```
    set_nodes(channelWidth, rng, simSeed, rsuNode, vehicularNode, phy, mac, wifi,
rsuDevice);

    ScenarioFactory helper = ScenarioFactory(nRSU, rsuNode, vehicularNode, port,
offeredLoad, history_length);
    vanetSceanrio = helper.getScenario(scenario);

    // if (!dry_run)
    // {
    if (non_zero_start)  //false
        end_delay = envStepTime * history_length + 1.0;
    else
        end_delay = 0.0;
    // }

    vanetSceanrio->installScenario(simulationTime + end_delay + envStepTime,
envStepTime, MakeCallback(&packetReceived));

   //在这里，该函数将信号PhyTxBegin和回调函数packetSent进行了连接。信号PhyTxBegin是在wifi
设备的物理层发送封包开始的时候发出的信号。而packetSent是一个回调函数，它会在这个信号被发出时被
调用，执行一些指定的操作。这里使用了通配符*来匹配所有节点和设备，因此该连接将适用于所有的节点和
wifi设备。连接的对象是一个字符串，表示了属性路径，指定了要连接的信号和设备的物理层对象的属性。连
接过程中的回调函数是一个函数指针或一个类的成员函数指针，指向了当信号被触发时需要调用的函数。
    //
Config::ConnectWithoutContext("/NodeList/0/ApplicationList/*/$ns3::OnOffApplicati
on/Tx", MakeCallback(&packetSent));

 Config::ConnectWithoutContext("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/
PhyTxBegin", MakeCallback(&packetSent));

    vanetSceanrio->PopulateARPcache();
    Ipv4GlobalRoutingHelper::PopulateRoutingTables();
    //从网络拓扑中自动计算并填充 IPv4 路由表，使得网络中的各个节点能够正确地路由数据包。该方法
会遍历网络拓扑中的所有节点，将每个节点的路由表填充好，以确  保网络能够进行正确的转发。
//在 NS-3 中，该方法通常在创建完整个网络拓扑之后被调用，以便自动填充路由表，从而使网络能够进行正
确的通信。
//https://www.nsnam.org/docs/doxygen/da/d40/group__globalrouting.html#details

    set_sim(tracing, dry_run, warmup, openGymPort, phy, rsuDevice, end_delay,
monitor, flowmon);

    double flowThr;
    float res =  g_rxPktNum * (1500 - 20 - 8 - 8) * 8.0 / 1024 / 1024;
    printf("Sent mbytes: %.2f\tThroughput: %.3f", res, res/simulationTime);
    ofstream myfile;
    myfile.open(outputCsv, ios::app);

    /* Contents of CSV output file
    Timestamp, CW, nRSU, RngRun, SourceIP, DestinationIP, Throughput
    */
    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>
(flowmon.GetClassifier());
    std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();
    for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
stats.begin(); i != stats.end(); ++i)
    {
```

```
        std::cout<<"############## Output simulation result"<<std::endl;
        auto time = std::time(nullptr); //Get timestamp
        auto tm = *std::localtime(&time);
        Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(i->first);
        flowThr = i->second.rxBytes * 8.0 / simulationTime / 1000 / 1000;
        NS_LOG_UNCOND("Flow " << i->first << " (" << t.sourceAddress << " -> " <<
t.destinationAddress << ")\tThroughput: " << flowThr << " Mbps\tTime: " << i-
>second.timeLastRxPacket.GetSeconds() - i->second.timeFirstTxPacket.GetSeconds()
<< " s\tRx packets " << i->second.rxPackets);
        myfile << std::put_time(&tm, "%Y-%m-%d %H:%M") << "," << CW << "," <<
nRSU << "," << RngSeedManager::GetRun() << "," << t.sourceAddress << "," <<
t.destinationAddress << "," << flowThr;
        myfile << std::endl;
    }
    myfile.close();

    Simulator::Destroy();
    NS_LOG_UNCOND("Packets registered by handler: " << g_rxPktNum << " Packets"
<< endl);

    return 0;
}
```

# 使用 WifiNetDevice

## 创建 WifiNetDevice

- 决定使用哪个物理层框架，`SpectrumWifiPhy`或 `YansWifiPhy`这将影响要使用的通道和 Phy 类型。
- 配置频道：频道负责将信号从一个设备获取到同一 Wi-Fi 频道上的其他设备。WifiChannel 的主要配置是传播损耗模型和传播延迟模型。
- 配置 WifiPhy：WifiPhy 负责实际发送和接收来自 Channel 的无线信号。在这里，WifiPhy 根据接收到的信号强度和噪声来决定是否成功解码每一帧。因此，WifiPhy 的主要配置是错误率模型，它实际上是根据信号计算成功解码帧的概率的模型。
- Configure WifiMac：这一步跟架构和设备层面的关系比较大。用户配置 wifi 架构（即 ad-hoc 或 ap-sta）以及是否支持 QoS (802.11e)、HT (802.11n) 和/或 VHT (802.11ac) 和/或 HE (802.11ax) 功能.
- 创建 WifiDevice：在此步骤中，用户配置所需的 wifi 标准（例如**802.11b**、**802.11g**、**802.11a**、**802.11n**、**802.11ac**或**802.11ax**）和速率控制算法。
- 配置移动性：最后，（通常）需要移动性模型才能使用 WifiNetDevice；即使设备是静止的，传播损耗计算也需要它们的相对位置。

无线电信道带宽与物理层的信道带宽不完全一样，虽然它们在某些情况下可以是相等的，但是它们是不同的概念。

无线电信道带宽是指在无线电频谱中用于传输无线电信号的一定带宽范围，通常用Hz（赫兹）来表示。它是在物理层下的一个概念，表示信号在无线电频谱上占据的宽度，通常受到调制方式、信道选择、频率等因素的影响。

物理层的信道带宽是指在WLAN中物理层中用于传输数据和控制信息的一定带宽范围，通常用MHz（兆赫）来表示。它是在WLAN设备和网络中的一个属性，表示设备和网络用于传输数据和控制信息的带宽大小，通常受到无线电信道带宽、调制方式、码率等因素的影响。

因此，在车联网等应用中，无线电信道带宽和物理层的信道带宽都是需要考虑的因素，它们对无线通信的性能和传输速率都有重要的影响。

```
#ifndef SCENARIO_H
#define SCENARIO_H

#include <fstream>
#include <string>
#include <math.h>
#include <ctime>   //timestampi
#include <iomanip> // put_time
#include <deque>
#include <algorithm>

using namespace std;
using namespace ns3;

class Scenario
{
  protected:
    int nWifim;
    NodeContainer wifiStaNode;
    NodeContainer wifiApNode;
    int port;
    std::string offeredLoad;
    std::vector<double> start_times;
    std::vector<double> end_times;
    int history_length;

    void installTrafficGenerator(ns3::Ptr<ns3::Node> fromNode,
                                 ns3::Ptr<ns3::Node> toNode,
                                 int port,
                                 std::string offeredLoad,
                                 double startTime,
                                 double endTime,
                                 ns3::Callback<void, Ptr<const Packet>>
callback);

  public:
    Scenario(int nWifim, NodeContainer wifiStaNode, NodeContainer wifiApNode, int
port, std::string offeredLoad, int history_length);
    virtual void installScenario(double simulationTime, double envStepTime,
ns3::Callback<void, Ptr<const Packet>> callback) = 0;
    void PopulateARPcache();
    int getActiveStationCount(double time);
    float getStationUptime(int id, double time);
};



class BasicScenario : public Scenario
{
    using Scenario::Scenario; //此类BasicScenario使用Scenario类的构造函数而不必提供它自
己的构造函数

  public:
    void installScenario(double simulationTime, double envStepTime,
ns3::Callback<void, Ptr<const Packet>> callback) override;
}; //10+0.1*20+1.0+0.1
```

```
class ConvergenceScenario : public Scenario
{
    using Scenario::Scenario;

  public:
    void installScenario(double simulationTime, double envStepTime,
ns3::Callback<void, Ptr<const Packet>> callback) override;
};
//ScenarioFactory(nRSU, rsuNode, vehicularNode, port, offeredLoad "150",
history_length"20");
class ScenarioFactory
{
  private:
    int nWifim;
    NodeContainer wifiStaNode;
    NodeContainer wifiApNode;
    int port;
    int history_length;
    std::string offeredLoad;

  public:  //以下的成员变量是公共的
    ScenarioFactory(int nWifim, NodeContainer wifiStaNode, NodeContainer
wifiApNode, int port, std::string offeredLoad, int history_length) //: 声明一个公共
构造函数，ScenarioFactory它接受六个参数：nWifim, wifiStaNode, wifiApNode, port,
offeredLoad, 和history_length。
    {
        this->nWifim = nWifim;//在构造函数中，将nWifim参数赋值给类成员变量nWifim
        this->wifiStaNode = wifiStaNode;//在构造函数中，将wifiStaNode参数赋值给类成员变
量wifiStaNode
        this->wifiApNode = wifiApNode;
        this->port = port;
        this->offeredLoad = offeredLoad;
        this->history_length = history_length;//在构造函数中，将history_length参数赋
值给类成员变量history_length
    }

    Scenario *getScenario(std::string scenario)
    {
        Scenario *wifiScenario;//声明一个名为wifiScenariotype 的指针变量Scenario*。星
号*表示它是一个指针变量，并且Scenario是它可以指向的对象类型。
        if (scenario == "basic")
        {
            wifiScenario = new BasicScenario(this->nWifim, this->wifiStaNode,
this->wifiApNode, this->port, this->offeredLoad, this->history_length); //new是
C++ 中的关键字，用于在运行时在堆上动态分配内存。它用于创建类的对象并返回指向该对象的指针。
        }
        else if (scenario == "convergence")
        {
            wifiScenario = new ConvergenceScenario(this->nWifim, this-
>wifiStaNode, this->wifiApNode, this->port, this->offeredLoad, this-
>history_length);
        }
        else
        {
            std::cout << "Unsupported scenario" << endl;
            exit(0);
        }
        return wifiScenario;
```

```
    }
};


Scenario::Scenario(int nWifim, NodeContainer wifiStaNode, NodeContainer
wifiApNode, int port, std::string offeredLoad, int history_length)
{
    this->nWifim = nWifim;
    this->wifiStaNode = wifiStaNode;//rsu
    this->wifiApNode = wifiApNode;//vehicle
    this->port = port;
    this->offeredLoad = offeredLoad;
    this->history_length = history_length;
}

int Scenario::getActiveStationCount(double time)
{
    int res=0;
    for(uint i=0; i<start_times.size(); i++)
        if(start_times.at(i)<time && time<end_times.at(i))
            res++;
    return res;
}

float Scenario::getStationUptime(int id, double time)
{
    return time - start_times.at(id);
    // int res=0;
    // for(uint i=0; i<start_times.size(); i++)
    //     if(start_times.at(i)<time && time<end_times.at(i))
    //         res++;
    // return res;
}

//installTrafficGenerator(this->wifiStaNode.Get(i), this->wifiApNode.Get(0),
this->port++, this->offeredLoad, 0.0 , simulationTime + 2 + delay, callback);
void Scenario::installTrafficGenerator(Ptr<ns3::Node> fromNode, Ptr<ns3::Node>
toNode, int port, string offeredLoad, double startTime, double endTime,
                                       ns3::Callback<void, Ptr<const Packet>>
callback)
{
    start_times.push_back(startTime);
    end_times.push_back(endTime);

    Ptr<Ipv4> ipv4 = toNode->GetObject<Ipv4>();           // Get Ipv4 instance of
the node 获得汽车的IPV4地址
    Ipv4Address addr = ipv4->GetAddress(1, 0).GetLocal(); // Get
Ipv4InterfaceAddress of xth interface.检索分配给第二个网络接口的第一
//个 IPv4 地址。
    ApplicationContainer sourceApplications, sinkApplications; //创造两个不同的
ApplicationContainer

    uint8_t tosValue = 0x70; //AC_BE，8 位无符号整数。该变量初始化为值0x70，它表示加速转
发 (EF) PHB（每跳行为）组中名为“AF41”的保证转发 (AF)                                  //类
的差分服务 (DiffServ) 代码点 (DSCP) 值。
    //Add random fuzz to app start time
    double min = 0.0;
    double max = 1.0;
```

```
    Ptr<UniformRandomVariable> fuzz = CreateObject<UniformRandomVariable>();//创建一个指向UniformRandomVariable使用该CreateObject()
//方法的类的实例的指针
    fuzz->SetAttribute("Min", DoubleValue(min));  //指向的随机变量对象的最小值min，即0.0
    fuzz->SetAttribute("Max", DoubleValue(max));//指向的随机变量对象的最大值，1.0

    InetSocketAddress sinkSocket(addr, port);//此行使用addr和参数创建类的实例port。addr是一个Ipv4Address对象，表示接收套接字的 IP 地址，是
//port一个整数，表示套接字将在其上接收数据的端口号。
    sinkSocket.SetTos(tosValue);    //区分具有不同服务要求的数据包的优先级
    //OnOffHelper onOffHelper ("ns3::TcpSocketFactory", sinkSocket);
    OnOffHelper onOffHelper("ns3::UdpSocketFactory", sinkSocket); //此行使用UdpSocketFactory和参数创建类的实例sinkSocket。该类
OnOffHelper用于在 NS-3 中创建和配置 OnOff 应用程序。
    onOffHelper.SetConstantRate(DataRate(offeredLoad + "Mbps"), 1500 - 20 - 8 - 8);
    //此行设置 OnOff 应用程序的数据速率和数据包大小。该SetConstantRate()方法用于为 OnOff 应用程序设置恒定数据速率。第一个参数是DataRate表示提供负      //载的对象，第二个参数是应用程序的数据包大小（以字节为单位）。在这种情况下，数据包大小设置为 1464 字节 (1500 - 20 - 8 - 8)以说明 IP 标头（20      //字节）、UDP 标头（8 字节）以及以太网帧标头和帧尾（8 字节）。

    // onOffHelper.TraceConnectWithoutContext("Tx", MakeCallback(&packetSent));
    sourceApplications.Add(onOffHelper.Install(fromNode)); //fromNode

    //PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory", sinkSocket);
    // PacketSinkHelper packetSinkHelper ("ns3::UdpSocketFactory", sinkSocket);
    UdpServerHelper sink(port);
    sinkApplications = sink.Install(toNode);
    // sinkApplications.Add (packetSinkHelper.Install (toNode)); //toNode

    sinkApplications.Start(Seconds(startTime));//该行在参数指定的模拟时间启动接收器应用程序（即，将接收数据的应用程序）
    sinkApplications.Stop(Seconds(endTime));//此行在参数指定的仿真时间停止接收器应用程序endTime。该Stop()方法用于停止应用程序。

    Ptr<UdpServer> udpServer = DynamicCast<UdpServer>(sinkApplications.Get(0));//使用sinkApplications.Get(0)获取第一个接收数据的应用程序，然后使用DynamicCast将其转换为UdpServer对象。Ptr<UdpServer>表示udpServer是一个指向UdpServer对象的智能指针，这里的DynamicCast是一个RTTI（Run-Time Type Identification，运行时类型识别）操作符，用于将一个基类指针（或引用）转换为派生类指针（或引用）。
    udpServer->TraceConnectWithoutContext("Rx", callback);

    sourceApplications.Start(Seconds(startTime));
    sourceApplications.Stop(Seconds(endTime));
}



void Scenario::PopulateARPcache()
{
    Ptr<ArpCache> arp = CreateObject<ArpCache>();  //用于将第 3 层地址转换为第 2 层的缓存查找表。此实现执行从 IPv4 到 MAC 地址的查找
    arp->SetAliveTimeout(Seconds(3600 * 24 * 365)); //设置条目处于 ALIVE 状态的时间（除非刷新）

    //迭代所有节点
    for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
```

```
    {

        Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();//节点指针上调用该GetObject方法以检索与该节点关联的 Ipv4L3Protocol 对象,然后将检索到的 Ipv4L3Protocol 对象存储在ip指针中供以后使用
        NS_ASSERT(ip != 0);//测试给定条件是否为真。如果条件评估为假，则显示断言失败消息并退出程序。在这种情况下，如果ip为空（即未找到当前节点的 Ipv4L3Protocol 对象），则断言将失败并且程序将终止。

        ObjectVectorValue interfaces;//ObjectVectorValue 是一个类，表示一个对象向量的值，其中每个元素都可以是任何类型的对象。因此，这个变量可以
//存储任意数量的对象。
        ip->GetAttribute("InterfaceList", interfaces);
       //使用一个指针 ip 调用 GetAttribute 方法，并将其返回值存储在 interfaces 变量中。GetAttribute 方法的作用是获取指定属性的值，并将其存储在        //指定变量中。在这里，属性名是 "InterfaceList"，类型是 ObjectVectorValue，因此 interfaces 变量将包含 "InterfaceList" 属性的值。

        for (ObjectVectorValue::Iterator j = interfaces.Begin(); j != interfaces.End(); j++)
        {
            //遍历网络中的接口列表
            //获取指向与循环中当前接口关联的 Ipv4Interface 对象的指针。
            Ptr<Ipv4Interface> ipIface = (*j).second->GetObject<Ipv4Interface>();
            //检查指向 Ipv4Interface 对象的指针是否有效（非空）。
            NS_ASSERT(ipIface != 0);
            //此行获取指向与当前接口关联的 NetDevice 对象的指针。
            Ptr<NetDevice> device = ipIface->GetDevice();
            //此行检查指向 NetDevice 对象的指针是否有效（非空）。
            NS_ASSERT(device != 0);
            //此行获取当前 NetDevice 对象的 MAC 地址并将其转换为 Mac48Address 对象。
            Mac48Address addr = Mac48Address::ConvertFrom(device->GetAddress());

            //此行启动一个循环，循环遍历与当前 Ipv4Interface 对象关联的 IP 地址列表。
            for (uint32_t k = 0; k < ipIface->GetNAddresses(); k++)
            {
                //此行获取与循环中当前接口地址关联的 IP 地址。
                Ipv4Address ipAddr = ipIface->GetAddress(k).GetLocal();
                //此行检查 IP 地址是否为环回地址，如果是则跳至下一个地址。
                if (ipAddr == Ipv4Address::GetLoopback())
                    continue;
                //此行将一个新条目添加到当前 IP 地址的 ARP 缓存中。
                ArpCache::Entry *entry = arp->Add(ipAddr);
                //此行创建一个 IPv4 标头对象并将目标地址设置为当前 IP 地址
                Ipv4Header ipv4Hdr;
                ipv4Hdr.SetDestination(ipAddr);
                //此行创建一个大小为 100 字节的新数据包对象。
                Ptr<Packet> p = Create<Packet>(100);
                //此行将新的 ARP 缓存条目标记为等待回复并将数据包和 IPv4 标头与其相关联。
                entry->MarkWaitReply(ArpCache::Ipv4PayloadHeaderPair(p, ipv4Hdr));
                entry->MarkAlive(addr);
            }
        }
    }


    for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
```

```
    {
        Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();
        NS_ASSERT(ip != 0);
        ObjectVectorValue interfaces;
        ip->GetAttribute("InterfaceList", interfaces);

        for (ObjectVectorValue::Iterator j = interfaces.Begin(); j != 
interfaces.End(); j++)
        {
            Ptr<Ipv4Interface> ipIface = (*j).second->GetObject<Ipv4Interface>();
            ipIface->SetAttribute("ArpCache", PointerValue(arp));
        }
    }
}



void BasicScenario::installScenario(double simulationTime, double envStepTime, 
ns3::Callback<void, Ptr<const Packet>> callback)
{
    for (int i = 0; i < this->nWifim; ++i)
    {
        installTrafficGenerator(this->wifiStaNode.Get(i), this-
>wifiApNode.Get(0), this->port++, this->offeredLoad, 0.0, simulationTime + 2 + 
envStepTime*history_length, callback);
    }
}

void ConvergenceScenario::installScenario(double simulationTime, double 
envStepTime, ns3::Callback<void, Ptr<const Packet>> callback)
{
    float delta = simulationTime/(this->nWifim-4);
    float delay = history_length*envStepTime;
    if (this->nWifim > 5)
    {
        for (int i = 0; i < 5; ++i)
        {
            installTrafficGenerator(this->wifiStaNode.Get(i), this-
>wifiApNode.Get(0), this->port++, this->offeredLoad, 0.0 , simulationTime + 2 + 
delay, callback);
        }
        for (int i = 5; i < this->nWifim; ++i)
        {
            installTrafficGenerator(this->wifiStaNode.Get(i), this-
>wifiApNode.Get(0), this->port++, this->offeredLoad, delay+(i - 4) * delta, 
simulationTime + 2 + delay, callback);
        }
    }
    else
    {
        std::cout << "Not enough Wi-Fi stations to support the convergence 
scenario." << endl;
        exit(0);
    }
}
#endif
```