```
#include <fstream>

#include <iostream>

#include <chrono>

#include <thread>

#include "ns3/core-module.h"

#include "ns3/dsr-module.h"

#include "ns3/applications-module.h"

#include "ns3/yans-wifi-helper.h"

#include "ns3/internet-module.h"

#include "ns3/mobility-module.h"

#include "ns3/aodv-module.h"

#include "ns3/olsr-module.h"

#include "ns3/network-module.h"

#include "ns3/dsdv-module.h"

#include "ns3/rtt-estimator.h"

#include "ns3/node.h"

#include "ns3/log.h"

using std::chrono::high_resolution_clock;
using std::chrono::microseconds;
using std::chrono::duration_cast;

using namespace std::chrono;

using namespace ns3;
using namespace dsr;

NS_LOG_COMPONENT_DEFINE("vanet-compare");

clock_t t;

class Routing_Definition {
  public: Routing_Definition();
  void Run(int number_Sinks, double txp, std::string CSV_Name);

  std::string CommandSetup(int argc, char ** argv);

  private: Ptr < Socket > SetupPacketReceive(Ipv4Address addr, Ptr < Node >
node);
```

```
  // Start measuring time
  static auto start = chrono::high_resolution_clock::now();
  void ReceivePacket(Ptr < Socket > socket);

  void CheckThroughput();

  // Stop measuring time and calculate the elapsed time
  auto end = std::chrono::high_resolution_clock::now();
  auto elapsed = std::chrono::duration_cast < std::chrono::nanoseconds > (end -
begin);

  //NS_LOG_INFO ("rtt for the above packet was",duration," microseconds");

  uint32_t port;
  uint32_t total_bytes;
  uint32_t Received_Packs;
  Time roundTrip;

  std::string CSV_named;
  int m_number_Sinks;
  std::string m_protocolName;
  double m_txp;
  bool Mobility_Trace;
  uint32_t m_protocol;
};

Routing_Definition::Routing_Definition(): port(9),
  total_bytes(0),
  Received_Packs(0),
  CSV_named("vanet.output.csv"),
  Mobility_Trace(true),
  m_protocol(1) {}

/*
Define observation space
*/
Ptr < OpenGymSpace > MyGetObservationSpace(void) {
  auto elapsed;

  Ptr < OpenGymBoxSpace > space = CreateObject < OpenGymBoxSpace > (elapsed);
  NS_LOG_UNCOND("MyGetObservationSpace: " << space);
  return space;
}

/*
Collect observations
*/
Ptr < OpenGymDataContainer > MyGetObservation(void) {
  auto elapsed
  box -> AddValue(elapsed);
}

NS_LOG_UNCOND("MyGetObservation: " << box);
return box;
}
```

```
static inline std::string PrintReceivedPacket(Ptr < Socket > socket, Ptr < Packet
> packet, Address senderAddress) {
  std::ostringstream oss;

  oss << Simulator::Now().GetSeconds() << " " << socket -> GetNode() -> GetId();

  if (InetSocketAddress::IsMatchingType(senderAddress)) {
    InetSocketAddress addr = InetSocketAddress::ConvertFrom(senderAddress);
    oss << " received one packet from " << addr.GetIpv4();
  } else {
    oss << " received one packet!";
  }

  return oss.str();
}

void Routing_Definition::ReceivePacket(Ptr < Socket > socket) {
  Ptr < Packet > packet;
  Address senderAddress;
  while ((packet = socket -> RecvFrom(senderAddress))) {
    total_bytes += packet -> GetSize();
    Received_Packs += 1;
    NS_LOG_UNCOND(PrintReceivedPacket(socket, packet, senderAddress));
  }
}

void Routing_Definition::CheckThroughput() {
  double kbs = (total_bytes * 8.0) / 1000;
  total_bytes = 0;

  std::ofstream out(CSV_named.c_str(), std::ios::app);

  out << (Simulator::Now()).GetSeconds() << "," <<
    kbs << "," <<
    Received_Packs << "," <<
    m_number_Sinks << "," <<
    m_protocolName << "," <<
    m_txp << "" <<
    std::endl;

  out.close();
  Received_Packs = 0;
  Simulator::Schedule(Seconds(1.0), & Routing_Definition::CheckThroughput, this);
}

Ptr < Socket > Routing_Definition::SetupPacketReceive(Ipv4Address addr, Ptr <
Node > node) {
  TypeId tid = TypeId::LookupByName("ns3::UdpSocketFactory");
  Ptr < Socket > sink = Socket::CreateSocket(node, tid);
  InetSocketAddress local = InetSocketAddress(addr, port);
  sink -> Bind(local);
  sink -> SetRecvCallback(MakeCallback( & Routing_Definition::ReceivePacket,
this));
  return sink;
}

std::string Routing_Definition::CommandSetup(int argc, char ** argv) {
  CommandLine cmd;
  cmd.AddValue("CSV_Name", "The name of the CSV output file name", CSV_named);
  cmd.AddValue("traceMobility", "Enable mobility tracing", Mobility_Trace);
  cmd.AddValue("protocol", "1=OLSR;2=AODV;3=DSDV;4=DSR", m_protocol);
  cmd.Parse(argc, argv);
  return CSV_named;
}

int main(int argc, char * argv[]) {
  Routing_Definition experiment;
  std::string CSV_Name = experiment.CommandSetup(argc, argv);

  //blank out the last output file and write the column headers
  std::ofstream out(CSV_Name.c_str());
  out << "SimulationSecond," <<
    "ReceiveRate," <<
    "Received_Packs," <<
    "NumberOfSinks," <<
    "RoutingProtocol," <<
    "TransmissionPower" <<
    std::endl;
  out.close();

  int number_Sinks = 10;
  double txp = 7.5;

  experiment.Run(number_Sinks, txp, CSV_Name);
}

void
Routing_Definition::Run(int number_Sinks, double txp, std::string CSV_Name) {
  Packet::EnablePrinting();
  m_number_Sinks = number_Sinks;
  m_txp = txp;
  CSV_named = CSV_Name;

  int nWifis = 50;

  double TotalTime = 200.0;
  std::string rate("2048bps");
  std::string phyMode("DsssRate11Mbps");
  std::string tr_name("vanet");
  int nodeSpeed = 20; //in m/s
  int nodePause = 1; //in s
  m_protocolName = "protocol";

  Config::SetDefault("ns3::OnOffApplication::PacketSize", StringValue("64"));
  Config::SetDefault("ns3::OnOffApplication::DataRate", StringValue(rate));

  //Set Non-unicastMode rate to unicast mode
  Config::SetDefault("ns3::WifiRemoteStationManager::NonUnicastMode",
StringValue(phyMode));

  NodeContainer adhocNodes;
  adhocNodes.Create(nWifis);

  // setting up wifi phy and channel using helpers
  WifiHelper wifi;
  wifi.SetStandard(WIFI_PHY_STANDARD_80211b);
  YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default();
  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannel.AddPropagationLoss("ns3::FriisPropagationLossModel");
  wifiPhy.SetChannel(wifiChannel.Create());

  // Add a mac and disable rate control
  WifiMacHelper wifiMac;
  wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
    "DataMode", StringValue(phyMode),
    "ControlMode", StringValue(phyMode));

  wifiPhy.Set("TxPowerStart", DoubleValue(txp));
  wifiPhy.Set("TxPowerEnd", DoubleValue(txp));

  wifiMac.SetType("ns3::AdhocWifiMac");
  NetDeviceContainer adhocDevices = wifi.Install(wifiPhy, wifiMac, adhocNodes);

  MobilityHelper mobilityAdhoc;
  int64_t streamIndex = 0; // used to get consistent mobility across scenarios

  ObjectFactory pos;
  pos.SetTypeId("ns3::RandomRectanglePositionAllocator");
  pos.Set("X", StringValue("ns3::UniformRandomVariable[Min=0.0|Max=300.0]"));
  pos.Set("Y", StringValue("ns3::UniformRandomVariable[Min=0.0|Max=1500.0]"));

  Ptr < PositionAllocator > taPositionAlloc = pos.Create() -> GetObject <
PositionAllocator > ();
  streamIndex += taPositionAlloc -> AssignStreams(streamIndex);

  std::stringstream ssSpeed;
  ssSpeed << "ns3::UniformRandomVariable[Min=0.0|Max=" << nodeSpeed << "]";
  std::stringstream ssPause;
  ssPause << "ns3::ConstantRandomVariable[Constant=" << nodePause << "]";
  mobilityAdhoc.SetMobilityModel("ns3::RandomWaypointMobilityModel",
    "Speed", StringValue(ssSpeed.str()),
    "Pause", StringValue(ssPause.str()),
    "PositionAllocator", PointerValue(taPositionAlloc));
  mobilityAdhoc.SetPositionAllocator(taPositionAlloc);
  mobilityAdhoc.Install(adhocNodes);
  streamIndex += mobilityAdhoc.AssignStreams(adhocNodes, streamIndex);
 NS_UNUSED(streamIndex); // From this point, streamIndex is unused

  AodvHelper aodv;
  OlsrHelper olsr;
  DsdvHelper dsdv;
  DsrHelper dsr;
  DsrMainHelper dsrMain;
  Ipv4ListRoutingHelper list;
  InternetStackHelper internet;

  switch (m_protocol) {
  case 1:
    list.Add(olsr, 100);
    m_protocolName = "OLSR";
    break;
  case 2:
    list.Add(aodv, 100);
    m_protocolName = "AODV";
    break;
  case 3:
    list.Add(dsdv, 100);
    m_protocolName = "DSDV";
    break;
  case 4:
    m_protocolName = "DSR";
    break;
  default:
    NS_FATAL_ERROR("No such protocol:" << m_protocol);
  }

  if (m_protocol < 4) {
    internet.SetRoutingHelper(list);
    internet.Install(adhocNodes);
  } else if (m_protocol == 4) {
    internet.Install(adhocNodes);
    dsrMain.Install(dsr, adhocNodes);
  }

  NS_LOG_INFO("assigning ip address");

  Ipv4AddressHelper addressAdhoc;
  addressAdhoc.SetBase("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer adhocInterfaces;
  adhocInterfaces = addressAdhoc.Assign(adhocDevices);

  OnOffHelper onoff1("ns3::UdpSocketFactory", Address());
  onoff1.SetAttribute("OnTime",
StringValue("ns3::ConstantRandomVariable[Constant=1.0]"));
  onoff1.SetAttribute("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0.0]"));

  for (int i = 0; i < number_Sinks; i++) {
    Ptr < Socket > sink = SetupPacketReceive(adhocInterfaces.GetAddress(i),
adhocNodes.Get(i));

    AddressValue remoteAddress(InetSocketAddress(adhocInterfaces.GetAddress(i),
port));
    onoff1.SetAttribute("Remote", remoteAddress);
    Ptr < UniformRandomVariable >
      var = CreateObject < UniformRandomVariable > ();
    ApplicationContainer temp = onoff1.Install(adhocNodes.Get(i + number_Sinks));
    temp.Start(Seconds(var -> GetValue(100.0, 101.0)));
    temp.Stop(Seconds(TotalTime));

  }

  std::stringstream ss;
  ss << nWifis;
  std::string nodes = ss.str();

  std::stringstream ss2;
  ss2 << nodeSpeed;
  std::string sNodeSpeed = ss2.str();
  std::stringstream ss3;
  ss3 << nodePause;
  std::string sNodePause = ss3.str();

  std::stringstream ss4;
  ss4 << rate;
  std::string sRate = ss4.str();

  AsciiTraceHelper ascii;
  MobilityHelper::EnableAsciiAll(ascii.CreateFileStream(tr_name + ".mob"));

  NS_LOG_INFO("Run Simulation.");

  CheckThroughput();

  // OpenGym Env
  Ptr < OpenGymInterface > openGymInterface = CreateObject < OpenGymInterface >
(openGymPort);
  //openGymInterface->SetGetActionSpaceCb( MakeCallback (&MyGetActionSpace) );
  openGymInterface -> SetGetObservationSpaceCb(MakeCallback( &
GetObservationSpace));
  //openGymInterface->SetGetGameOverCb( MakeCallback (&MyGetGameOver) );
  openGymInterface -> SetGetObservationCb(MakeCallback( & GetObservation));
  openGymInterface->SetGetRewardCb( MakeCallback (&MyGetReward) );
  openGymInterface->SetExecuteActionsCb( MakeCallback (&MyExecuteActions) );

  Simulator::Stop(Seconds(TotalTime));
  Simulator::Run();
  openGymInterface -> NotifySimulationEnd();
  Simulator::Destroy();
}
```