Thz_Demo.cc file Source Code

```
#include "ns3/antenna-module.h"
#include "ns3/applications-module.h"
#include "ns3/config-store.h"
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/mobility-module.h"
#include "ns3/network-module.h"
#include "ns3/thz-channel.h"
#include "ns3/thz-dir-antenna.h"
#include "ns3/thz-directional-antenna-helper.h"
#include "ns3/thz-mac-macro-ap-helper.h"
#include "ns3/thz-mac-macro-ap.h"
#include "ns3/thz-mac-macro-client-helper.h"
#include "ns3/thz-mac-macro-client.h"
#include "ns3/thz-mac-macro-helper.h"
#include "ns3/thz-mac-macro.h"
#include "ns3/thz-phy-macro-helper.h"
#include "ns3/thz-phy-macro.h"
#include "ns3/thz-spectrum-waveform.h"
#include "ns3/thz-udp-client-server-helper.h"
#include "ns3/thz-udp-client.h"
#include "ns3/thz-udp-server.h"
#include "ns3/thz-udp-trace-client.h"
#include "ns3/traffic-generator-helper.h"
#include "ns3/traffic-generator.h"
#include "ns3/netanim-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/node.h"
#include "ns3/net-device.h"
#include <fstream>
#include <sstream>
#include "ns3/queue.h"
#include <cmath>
#include <iostream>
#include <vector>
#include <sys/stat.h>
#include "ns3/ipv4-flow-classifier.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/flow-monitor-helper.h"

#define _BPSK 1

#define _QPSK 2

#define _8PSK 3
```

```
#define _16QAM 4

#define _64QAM 5
using namespace ns3;

/* This example file is for the macroscale scenario of the THz-band communication
networks, i.e.,

 * with transmission distance larger than several meters. A centralized network architecture is

 * implemented. A high speed turning directional antenna is used in the base station
(Servernodes),

 * while all clients (Clientnodes) point the directional antennas towards the receiver.

 *

 * Important parameters:

 *  - configuration: sets the frequency window used, the number of sectors and modulation
used

 *  - handshake_ways: use a 0-, 1-, 2- or 3-way handshake. (0: CSMA, 1: ADAPT-1, 2:
CSMA/CA, 3: ADAPT-3)

 *  - nodeNum: number of client nodes

 *  - interArrivalTime: average time between two packets arriving at client's queue

 *

 * Output: TXT file with an entry for each packet in the format:

 *    (client_id, packet_size, packet_delay, success, discard)

 * Note that throughput and discard rate metrics have to be computed in postprocessing from
this TXT

 * file. A MATLAB script is provided at last part of this pdf.

 */



NS_LOG_COMPONENT_DEFINE("MacroCentral");
const double BOLTZMANN_CONSTANT = 1.380649e-23; // [J/K] Boltzmann constant

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

```
{
    /* ---------------------------------- PARAMETERS SET UP ---------------------------------- */
    Time::SetResolution(Time::PS); // Picoseconds
    int mcs, sectors;

    double beamwidth, maxGain, radius, noiseFloor, carrierSenseTh, txPower, sinrTh,
basicRate, dataRate, bandwidth, centralFreq, bit_energy, noiseTotal;

    double csth_BPSK, csth_QPSK, csth_8PSK, csth_16QAM, csth_64QAM;

    Time prop_delay;


    int configuration = 20;     // Configuration (1, 20-29)

    int seedNum = 1;            // Seed number

    int nodeNum = 50;           // Number of client nodes

    int handshake_ways = 3;     // Protocol (0: CSMA, 1: ADAPT-1, 2: CSMA/CA, 3: ADAPT-
3)

    int packetSize = 65000;     // [bytes] Packet size

    int interArrivalTime = 200; // [us] Mean inter-arrival time

    double simDuration = 0.001;  // [s] Simulation duration

    int boSlots = 5;            // Number of slots in the random backoff

    int rtsLim = 5;             // RTS retry limit

    double temperature = 300;   // [K] Temperature

    double noiseFigure = 7;     // [dB] Noise figure

    bool use_whiteList = true;  // Flag to use white list

    bool use_adaptMCS = true;   // Flag to use adaptive MCS


    CommandLine cmd;

    cmd.AddValue("seedNum", "Seed number", seedNum);

    cmd.AddValue("nodeNum", "Number of Clients", nodeNum);

    cmd.AddValue("way", "Chose handshake ways", handshake_ways);

    cmd.AddValue("packetSize", "Packet size in bytes", packetSize);
```

```
    cmd.AddValue("interArrivalTime", "Mean time between the arrival of packets.
Exponantial distribution", interArrivalTime);

    cmd.Parse(argc, argv);


    /* ---------------------------------- ENABLE LOGS --------------------------------------- */

     //LogComponentEnable("THzSpectrumValueFactory", LOG_LEVEL_ALL);

     //LogComponentEnable("THzSpectrumPropagationLoss", LOG_LEVEL_ALL);

     //LogComponentEnable("THzDirectionalAntenna", LOG_LEVEL_ALL);

     //LogComponentEnable("THzNetDevice", LOG_LEVEL_ALL);

     //LogComponentEnable("THzMacMacro", LOG_LEVEL_ALL);

     //LogComponentEnable("THzPhyMacro", LOG_LEVEL_ALL);

     //LogComponentEnable("THzChannel", LOG_LEVEL_ALL);

     //LogComponentEnable ("THzUdpClient", LOG_LEVEL_ALL);

     //LogComponentEnable ("THzUdpServer", LOG_LEVEL_ALL);


    /* --------------------------------- CONFIGURATION PARAMETERS --------------------------
*/

    // Config 1: True THz window (90 GHz wide at fc = 1.0345 THz). Don't use Adaptive
MCS - data rates are for 802.15.3d window

    if (configuration == 1)

    {

        txPower = 0;             // [dBm] Transmit power

        bandwidth = 90e9;        // [Hz] Bandwidth

        centralFreq = 1.0345e12; // [Hz] Central frequency

        radius = 2.7;            // [m] Radius

        dataRate = 1.8e11;       // [bps] Data rate

        basicRate = 1.8e11;      // [bps] Basic rate
```

```
        bit_energy = 10.6;       // [dB] Eb/N0

        beamwidth = 6;           // [deg] Beamwidth

        maxGain = 30.59;         // [dBi] Maximum gain



        sinrTh = bit_energy + 10 * log10(dataRate / bandwidth);                    // [dB] SINR_th = 
Eb/N0*R/B

        noiseFloor = 10 * log10(BOLTZMANN_CONSTANT * temperature * bandwidth) + 30; 
// [dBm] Noise floor = kTB

        noiseTotal = noiseFloor + noiseFigure;                                   // [dBm] Total noise floor

        carrierSenseTh = noiseFloor + sinrTh;                                    // [dBm] Received power 
threshold


        use_whiteList = false;

        use_adaptMCS = false;



        Config::SetDefault("ns3::THzSpectrumValueFactory::TotalBandWidth", 
DoubleValue(bandwidth));

        Config::SetDefault("ns3::THzSpectrumValueFactory::NumSample", DoubleValue(32));

        Config::SetDefault("ns3::THzSpectrumValueFactory::CentralFrequency", 
DoubleValue(centralFreq));

        Config::SetDefault("ns3::THzSpectrumValueFactory::SubBandWidth", 
DoubleValue(9e8));

        Config::SetDefault("ns3::THzSpectrumValueFactory::NumSubBand", 
DoubleValue(100));

    }



    // Configs 20-29: 69.12 GHz window at fc = 287. GHz

    // Config 20 and 29 reproduce results in "ADAPT: An Adaptive Directional Antenna 
Protocol for medium access control in Terahertz communication networks"

    else
```

```
{
    txPower = 20;
    bandwidth = 69.12e9;
    centralFreq = 287.28e9;

    if (configuration == 20)
    {
        mcs = _8PSK;
        sectors = 30;
        radius = 18;
    }
    else if (configuration == 21)
    {
        mcs = _64QAM;
        sectors = 45;
        radius = 16.7;
    }
    else if (configuration == 22)
    {
        mcs = _QPSK;
        sectors = 30;
        radius = 34;
    }
    else if (configuration == 23)
    {
```

```
    mcs = _16QAM;
    sectors = 45;
    radius = 35;
}
else if (configuration == 24)
{
    mcs = _64QAM;
    sectors = 60;
    radius = 30;
}
else if (configuration == 25)
{
    mcs = _BPSK;
    sectors = 30;
    radius = 48;
}
else if (configuration == 26)
{
    mcs = _8PSK;
    sectors = 45;
    radius = 40;
}
else if (configuration == 27)
{
    mcs = _16QAM;
```

```
    sectors = 60;
    radius = 64;
}
else if (configuration == 28)
{
    mcs = _QPSK;
    sectors = 15;
    radius = 8.4;
}
else // (configuration == 29)
{
    mcs = _64QAM;
    sectors = 30;
    radius = 7.5;
}

noiseFloor = 10 * log10(BOLTZMANN_CONSTANT * temperature * bandwidth) + 30;
noiseTotal = noiseFloor + noiseFigure;

// BPSK
dataRate = 52.4e9;
bit_energy = 10.6;
double sinrTh_BPSK = bit_energy + 10 * log10(dataRate / bandwidth);
csth_BPSK = noiseTotal + sinrTh_BPSK;
```

```
// QPSK
dataRate = 105.28e9;
bit_energy = 10.6;
double sinrTh_QPSK = bit_energy + 10 * log10(dataRate / bandwidth);
csth_QPSK = noiseTotal + sinrTh_QPSK;

// 8-PSK
dataRate = 157.44e9;
bit_energy = 14;
double sinrTh_8PSK = bit_energy + 10 * log10(dataRate / bandwidth);
csth_8PSK = noiseTotal + sinrTh_8PSK;

// 16-QAM
dataRate = 210.24e9;
bit_energy = 14.4;
double sinrTh_16QAM = bit_energy + 10 * log10(dataRate / bandwidth);
csth_16QAM = noiseTotal + sinrTh_16QAM;

// 64-QAM
dataRate = 315.52e9;
bit_energy = 18.8;
double sinrTh_64QAM = bit_energy + 10 * log10(dataRate / bandwidth);
csth_64QAM = noiseTotal + sinrTh_64QAM;

// Modulation Coding Scheme
```

```
if (mcs == _BPSK) // BPSK
{
    dataRate = 52.4e9;
    bit_energy = 10.6;
    sinrTh = sinrTh_BPSK;
    carrierSenseTh = csth_BPSK;
}
else if (mcs == _QPSK) // QPSK
{
    dataRate = 105.28e9;
    bit_energy = 10.6;
    sinrTh = sinrTh_QPSK;
    carrierSenseTh = csth_QPSK;
}
else if (mcs == _8PSK) // 8-PSK
{
    dataRate = 157.44e9;
    bit_energy = 14;
    sinrTh = sinrTh_8PSK;
    carrierSenseTh = csth_8PSK;
}
else if (mcs == _16QAM) // 16-QAM
{
    dataRate = 210.24e9;
    bit_energy = 14.4;
```

```
            sinrTh = sinrTh_16QAM;

            carrierSenseTh = csth_16QAM;

        }

        else // (mcs == _64QAM) // 64-QAM

        {

            dataRate = 315.52e9;

            bit_energy = 18.8;

            sinrTh = sinrTh_64QAM;

            carrierSenseTh = csth_64QAM;

        }


        basicRate = dataRate;

        beamwidth = 360 / sectors;

        maxGain = 20 * log10(sectors) - 4.971498726941338;


        Config::SetDefault("ns3::THzSpectrumValueFactory::TotalBandWidth",
    DoubleValue(bandwidth));

        Config::SetDefault("ns3::THzSpectrumValueFactory::NumSample", DoubleValue(32));

        Config::SetDefault("ns3::THzSpectrumValueFactory::CentralFrequency",
    DoubleValue(centralFreq));

        Config::SetDefault("ns3::THzSpectrumValueFactory::SubBandWidth",
    DoubleValue(2.16e9));

        Config::SetDefault("ns3::THzSpectrumValueFactory::NumSubBand",
    DoubleValue(32));

    }


    prop_delay = PicoSeconds(radius * 3336); // Propagation delay is 3336 ps/m (1/c s/m)
```

```
    std::string outputFile = "result_" + std::to_string(handshake_ways) + "way_" +
std::to_string(nodeNum) + "n_" + std::to_string(interArrivalTime) + "us_" +
std::to_string(seedNum) + ".txt";


    RngSeedManager seed;

    seed.SetSeed(seedNum);


    uint8_t SNodes = 1;

    uint16_t CNodes = nodeNum;

    NodeContainer Servernodes;

    Servernodes.Create(SNodes);

    NodeContainer Clientnodes;

    Clientnodes.Create(CNodes);

    NodeContainer nodes;

    nodes.Add(Servernodes);

    nodes.Add(Clientnodes);


    /* --------------------------------- MOBILITY ------------------------------------------ */

    MobilityHelper mobility;

    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>();

    positionAlloc->Add(Vector(0.0, 0.0, 0.0));

    mobility.SetPositionAllocator(positionAlloc);

    mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");

    mobility.Install(Servernodes);
```

```
  mobility.SetPositionAllocator("ns3::UniformDiscPositionAllocator",
                              "X", DoubleValue(0.0),
                              "Y", DoubleValue(0.0),
                              "rho", DoubleValue(radius));
   mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
  mobility.Install(Clientnodes);


  /* --------------------------------- SET ATTRIBUTES AND CONNECT ALL --------------------
*/
  NetDeviceContainer serverDevices;
  NetDeviceContainer clientDevices;


  // CHANNEL
  Ptr<THzChannel> thzChan = CreateObjectWithAttributes<THzChannel>("NoiseFloor",
DoubleValue(noiseTotal));




  AnimationInterface anim("thz_macro.xml");
  anim.SetMobilityPollInterval(MicroSeconds(250));
  anim.EnablePacketMetadata(true);



 uint32_t serverImageId = anim.AddResource("server.png");
  uint32_t clientImageId = anim.AddResource("client.png");
```

```
    for (uint16_t i = 0; i < Servernodes.GetN(); ++i) {
    anim.UpdateNodeImage(Servernodes.Get(i)->GetId(), serverImageId);
}

for (uint16_t i = 0; i < Clientnodes.GetN(); ++i) {
    anim.UpdateNodeImage(Clientnodes.Get(i)->GetId(), clientImageId);
}

    // PHY
    THzPhyMacroHelper thzPhy = THzPhyMacroHelper::Default();
    thzPhy.Set("CsPowerTh", DoubleValue(carrierSenseTh));
    thzPhy.Set("TxPower", DoubleValue(txPower));
    thzPhy.Set("SinrTh", DoubleValue(sinrTh));
    thzPhy.Set("BasicRate", DoubleValue(basicRate));
    thzPhy.Set("DataRate", DoubleValue(dataRate));




    if (handshake_ways == 1 || handshake_ways == 3) // ADAPT-1 or ADAPT-3
    {
        // MAC AP
        THzMacMacroApHelper thzMacAp = THzMacMacroApHelper::Default();
        thzMacAp.Set("CS_BPSK", DoubleValue(csth_BPSK));
        thzMacAp.Set("CS_QPSK", DoubleValue(csth_QPSK));
```

```
thzMacAp.Set("CS_8PSK", DoubleValue(csth_8PSK));
thzMacAp.Set("CS_16QAM", DoubleValue(csth_16QAM));
thzMacAp.Set("CS_64QAM", DoubleValue(csth_64QAM));

thzMacAp.Set("UseWhiteList", BooleanValue(use_whiteList));
thzMacAp.Set("UseAdaptMCS", BooleanValue(use_adaptMCS));
thzMacAp.Set("OutputFile", StringValue(outputFile));
thzMacAp.Set("BoSlots", UintegerValue(boSlots));
thzMacAp.Set("PacketSize", UintegerValue(packetSize));
thzMacAp.Set("PropDelay", TimeValue(prop_delay));
thzMacAp.Set("HandshakeWays", UintegerValue(handshake_ways));

// MAC CLIENT
THzMacMacroClientHelper thzMacClient = THzMacMacroClientHelper::Default();
thzMacClient.Set("OutputFile", StringValue(outputFile));
thzMacClient.Set("BoSlots", UintegerValue(boSlots));
thzMacClient.Set("PacketSize", UintegerValue(packetSize));
thzMacClient.Set("RtsRetryLimit", UintegerValue(rtsLim));
thzMacClient.Set("DataRate", DoubleValue(dataRate));
thzMacClient.Set("PropDelay", TimeValue(prop_delay));
thzMacClient.Set("HandshakeWays", UintegerValue(handshake_ways));

// Directional Antenna
THzDirectionalAntennaHelper thzDirAntenna =
THzDirectionalAntennaHelper::Default();
```

```
        thzDirAntenna.Set("MaxGain", DoubleValue(maxGain));

        thzDirAntenna.Set("BeamWidth", DoubleValue(beamwidth));



        // Connect all layers in a NetDevice

        THzHelper thz;

        serverDevices = thz.Install(Servernodes, thzChan, thzPhy, thzMacAp, thzDirAntenna);

        clientDevices = thz.Install(Clientnodes, thzChan, thzPhy, thzMacClient,
thzDirAntenna);

    }

    else // CSMA (0-way) or CSMA/CA (2-way)

    {

        double turningSpeed = 0;

        if (configuration == 20)

        {

            turningSpeed = 9000; // Tsector is aprox 3704 ns, enough for 1 DATA packet of 65000
B

        }

        else if (configuration == 29)

        {

            turningSpeed = 19000; // Tsector is aprox 1754 ns

        }

        // For other configurations, calculate and set the turning speed that makes Tsector just
enough to transmit one packet



        // MAC (same MAC for AP and Client nodes)

        THzMacMacroHelper thzMac = THzMacMacroHelper::Default();

        thzMac.Set("TurnSpeed", DoubleValue(turningSpeed));
```

```
        thzMac.Set("MaxGain", DoubleValue(maxGain));
        thzMac.Set("NumSectors", UintegerValue(sectors));
        thzMac.Set("DataRate", DoubleValue(dataRate));
        thzMac.Set("BasicRate", DoubleValue(basicRate));
        thzMac.Set("Radius", DoubleValue(radius));
        thzMac.Set("Nodes", UintegerValue(nodeNum));
        thzMac.Set("PacketSize", UintegerValue(packetSize));
        thzMac.Set("Tia", UintegerValue(interArrivalTime));
        thzMac.Set("HandshakeWays", UintegerValue(handshake_ways));
        thzMac.Set("OutputFile", StringValue(outputFile));

        // Directional Antenna
        THzDirectionalAntennaHelper thzDirAntenna =
THzDirectionalAntennaHelper::Default();
        thzDirAntenna.Set("TurningSpeed", DoubleValue(turningSpeed));
        thzDirAntenna.Set("MaxGain", DoubleValue(maxGain));
        thzDirAntenna.Set("BeamWidth", DoubleValue(beamwidth));

        // Connect all layers in a NetDevice
        THzHelper thz;
        serverDevices = thz.Install(Servernodes, thzChan, thzPhy, thzMac, thzDirAntenna);
        clientDevices = thz.Install(Clientnodes, thzChan, thzPhy, thzMac, thzDirAntenna);
    }
    // Group all devices
    NetDeviceContainer devices = NetDeviceContainer(serverDevices, clientDevices);
```

```
/* --------------------------------- PRINT IN CONSOLE ---------------------------------- */
std::printf("Time resolution set to: %d\n", Time::GetResolution());
std::printf("seedNum = %d\n", seed.GetSeed());
std::printf("config = %d\n", configuration);
std::printf("nodeNum = %d\n", Clientnodes.GetN());
std::printf("Tia = %d\n", interArrivalTime);
std::printf("Configuration = %d\n", configuration);
std::printf("NoiseFloor = %f\n", noiseTotal);
std::printf("carrierSenseTh = %f\n", carrierSenseTh);
std::printf("txPower = %f\n", txPower);
std::printf("SinrTh = %f\n", sinrTh);
std::printf("BasicRate = %f\n", basicRate);
std::printf("DataRate = %f\n", dataRate);
std::printf("Radius = %f\n", radius);
std::printf("Beamwidth = %f\n", beamwidth);
std::printf("MaxGain = %f\n", maxGain);
std::printf("Use white list = %d\n", use_whiteList);
std::printf("Use adaptive MCS = %d\n", use_adaptMCS);
std::printf("Handshake ways: %d way\n", handshake_ways);

/* --------------------------------- SETUP NETWORK LAYER ------------------------------- */
InternetStackHelper internet;
internet.Install(nodes);
```

```
Ipv4AddressHelper ipv4;
ipv4.SetBase("10.1.2.0", "255.255.254.0");
Ipv4InterfaceContainer iface = ipv4.Assign(devices);


  FlowMonitorHelper flowmon;
 Ptr<FlowMonitor> monitor = flowmon.InstallAll();


/* --------------------------------- POPULATE ARP CACHE -------------------------------- */
Ptr<ArpCache> arp = CreateObject<ArpCache>();
arp->SetAliveTimeout(Seconds(3600.0));
for (uint16_t i = 0; i < nodes.GetN(); i++)
{
    Ptr<Ipv4L3Protocol> ip = nodes.Get(i)->GetObject<Ipv4L3Protocol>();
    NS_ASSERT(ip);
    int ninter = (int)ip->GetNInterfaces();
    for (int j = 0; j < ninter; j++)
    {
        Ptr<Ipv4Interface> ipIface = ip->GetInterface(j);
        NS_ASSERT(ipIface);
        Ptr<NetDevice> device = ipIface->GetDevice();
        NS_ASSERT(device);
        Mac48Address addr = Mac48Address::ConvertFrom(device->GetAddress());
        for (uint32_t k = 0; k < ipIface->GetNAddresses(); k++)
        {
            Ipv4Address ipAddr = ipIface->GetAddress(k).GetLocal();
```

```
            if (ipAddr == Ipv4Address::GetLoopback())
            {
                continue;
            }
            ArpCache::Entry* entry = arp->Add(ipAddr);

            Ipv4Header ipHeader;
            Ptr<Packet> packet = Create<Packet>();
            packet->AddHeader(ipHeader);

            entry->MarkWaitReply(ArpCache::Ipv4PayloadHeaderPair(packet, ipHeader));
            entry->MarkAlive(addr);
        }
    }
}
for (uint16_t i = 0; i < nodes.GetN(); i++)
{
    Ptr<Ipv4L3Protocol> ip = nodes.Get(i)->GetObject<Ipv4L3Protocol>();
    NS_ASSERT(ip);
    int ninter = (int)ip->GetNInterfaces();
    for (int j = 0; j < ninter; j++)
    {
        Ptr<Ipv4Interface> ipIface = ip->GetInterface(j);
        ipIface->SetArpCache(arp);
    }
```

```
}

for (uint16_t i = 0; i < nodes.GetN(); i++)
{
    anim.UpdateNodeDescription(nodes.Get(i), "THzNode");
    anim.UpdateNodeSize(nodes.Get(i)->GetId(), 2.0, 2.0);
}

/* --------------------------------- START SIMULATION ---------------------------------- */
THzUdpServerHelper Server(9);
ApplicationContainer Apps = Server.Install(Servernodes);
Apps.Start(Seconds(0.0));
Apps.Stop(Seconds(10.0));

THzUdpClientHelper Client(iface.GetAddress(0), 9);
Client.SetAttribute("PacketSize", UintegerValue(packetSize));
Client.SetAttribute("Mean", DoubleValue(interArrivalTime));
Apps = Client.Install(Clientnodes);
Apps.Start(MicroSeconds(15));
Apps.Stop(Seconds(10.0));

//Simulator::Stop(Seconds(0.1));
ConfigStore config;
config.ConfigureDefaults();
config.ConfigureAttributes();
```

```
    Simulator::Stop(Seconds(simDuration+0.001));

    Simulator::Run();


    Simulator::Destroy();


   std::ifstream traceFile("result_3way_50n_200us_1.txt");

    if (!traceFile.is_open()) {

        std::cerr << "Error opening trace file." << std::endl;

        return 1;

    }


    // Variables to store throughput information

    double totalBytesSent = 0.0;

    double totalBytesReceived = 0.0;


    // Loop through each line in the trace file

    std::string line;

    while (std::getline(traceFile, line)) {

        // Parse the line to extract relevant information

        // For simplicity, let's assume a comma-separated format: client_id, packet_size,
packet_delay, success, discard

        // You may need to adjust this based on your actual trace file format

        int clientId, packetSize, packetDelay, success, discard;

        std::istringstream iss(line);

        if (!(iss >> clientId >> packetSize >> packetDelay >> success >> discard)) {
```

```
            break; // Error parsing line
        }

        // Update throughput based on success status
        if (clientId > 0) { // Ignore server node (assuming client_id > 0)
            totalBytesSent += packetSize;
            if (success == 1 && discard == 0) {
                totalBytesReceived += packetSize;
            }
        }
    }

    // Calculate throughput in bits per second (bps) for client nodes
    double throughput = (totalBytesReceived * 8) / simDuration; // simDuration in seconds

    // Output throughput for client nodes
    std::cout << "Total Bytes Sent: " << totalBytesSent << std::endl;
    std::cout << "Total Bytes Received: " << totalBytesReceived << std::endl;
    std::cout << "Throughput (bps): " << throughput << std::endl;

    // Close the trace file
    traceFile.close();
    return 0;
}
//-----------------------------------------------------------------------------------------------------
```

Metrics.py file for output calculation

```
#! /usr/bin/env python3

import numpy as np

import matplotlib.pyplot as plt

# Parameters

handshake_ways = 3  # 0, 1, 2, or 3-way handshake (0: CSMA, 1: ADAPT-1, 2: CSMA/CA,
3: ADAPT-3)

nodeNum = 50        # Number of client nodes

Tia = 200           # [us] Mean inter-arrival time

bandwidth = 90e9;


# Load simulation results

filename = f"result_3way_50n_200us_1.txt"

data = np.loadtxt(filename, dtype=int)



# Compute metrics

data = data.T

file_path="/home/ybab/ns-allinone-3.39/ns-
3.39/contrib/thz/results/result_3way_50n_200us_1.txt"

with open(file_path,'r') as file:

    da=[list(map(int, line.split())) for line in file]



#da = da[da[:, 1].argsort()]

da = sorted(da, key=lambda x: x[1])

# Calculate delay vs. number of nodes

delays = np.zeros(nodeNum)
```

```
for n in range(1, nodeNum + 1):
    # Select only data from node n and successful packets
    succ_data = np.array([row for row in da if row[0] == n and row[3] == 1])
    delays[n - 1] = np.mean(succ_data[:, 2])
print(delays)

# Plot delay vs. number of nodes
plt.plot(range(1, nodeNum + 1), delays, marker='o')
plt.xlabel('Number of Nodes')
plt.ylabel('Average Packet Delay (us)')
plt.title('Average Packet Delay vs Number of Nodes')
plt.grid(True)
plt.show()

jitters = np.zeros(nodeNum)
for n in range(1, nodeNum + 1):
    # Select only data from node n and successful packets
    succ_data = np.array([row for row in da if row[0] == n and row[3] == 1])
    jitters[n - 1] = np.std(succ_data[:, 2])

# Plot jitter vs. number of nodes
plt.figure()  # Create a new figure for the jitter plot
plt.plot(range(1, nodeNum + 1), jitters, marker='o', color='r')
plt.xlabel('Number of Nodes')
plt.ylabel('Packet Jitter (us)')
```

```
plt.title('Packet Jitter vs Number of Nodes')

plt.grid(True)

plt.show()



even_latencies = np.zeros(nodeNum)

for n in range(1, nodeNum + 1):

    # Select only data from node n and successful packets

    succ_data = np.array([row for row in da if row[0] == n and row[3] == 1])

    sorted_delays = np.sort(succ_data[:, 2])

    even_latencies[n - 1] = np.mean(sorted_delays[1::2])  # Take every second element for
even latency


# Plot even latency vs. number of nodes

plt.figure()  # Create a new figure for the even latency plot

plt.plot(range(1, nodeNum + 1), even_latencies, marker='o', color='g')

plt.xlabel('Number of Nodes')

plt.ylabel('Even Latency (us)')

plt.title('Even Latency vs Number of Nodes')

plt.grid(True)

plt.show()



pdr_values = np.zeros(nodeNum)

for n in range(1, nodeNum + 1):
```

```
    total_packets = sum(row[0] == n for row in da)
    successful_packets = sum((row[0] == n) and (row[3] == 1) for row in da)
    pdr_values[n - 1] = successful_packets / total_packets if total_packets > 0 else 0

# Plot PDR vs. number of nodes
plt.figure()  # Create a new figure for the PDR plot
plt.plot(range(1, nodeNum + 1), pdr_values, marker='o', color='b')
plt.xlabel('Number of Nodes')
plt.ylabel('Packet Delivery Ratio (PDR)')
plt.title('PDR vs Number of Nodes')
plt.grid(True)
plt.show()



qos_values = np.zeros(nodeNum)
for n in range(1, nodeNum + 1):
    total_packets = sum(row[0] == n for row in da)
    successful_packets = sum((row[0] == n) and (row[3] == 1) for row in da)

    avg_latency = np.mean([row[2] for row in da if row[0] == n and row[3] == 1])
    packet_jitter = np.std([row[2] for row in da if row[0] == n and row[3] == 1])
    pdr = successful_packets / total_packets if total_packets > 0 else 0

    # Define your QoS metric based on latency, jitter, and PDR
    qos_values[n - 1] = 1 / (avg_latency + packet_jitter) * pdr
```

```
# Plot QoS vs. number of nodes
plt.figure()  # Create a new figure for the QoS plot
plt.plot(range(1, nodeNum + 1), qos_values, marker='o', color='purple')
plt.xlabel('Number of Nodes')
plt.ylabel('Quality of Service (QoS)')
plt.title('QoS vs Number of Nodes')
plt.grid(True)
plt.show()



# Throughput calculation
throughput_node = np.zeros(nodeNum)  # [Gbps] Throughput of each node
for n in range(1, nodeNum + 1):
    # Select only data from node n and successful packets
    succ_data = data[(data[:, 0] == n) & (data[:, 3] == 1)]

    if succ_data.shape[0] > 0:
        throughput_node[n - 1] = np.mean(succ_data[:, 1] * 8 / succ_data[:, 2])
    else:
        throughput_node[n - 1] = 0

throughput = np.mean(throughput_node)  # [Gbps] Average throughput


# Discard rate
```

```
discard_rate = np.sum(data[:, 4]) / data.shape[0]


# [us] Average packet time

succ_data = data[data[:, 3] == 1]

average_packet_time = np.mean(succ_data[-round(len(succ_data) / 10):, 2]) * 1e-3 * 1e3

spectral_efficiency = throughput*10 / (bandwidth * 1e-9)


# Print out results

print(f"Throughput = {throughput*10} Tbps")

print(f"Spectral Efficiency = {spectral_efficiency} bps/Hz")

print(f"Discard rate = {discard_rate}")

print(f"Average packet time = {average_packet_time:.2f} ps")


# Print out latency results

average_latency = np.mean(delays)  # [us] Average latency

print(f'Aver#! /usr/bin/env python3


import numpy as np

import matplotlib.pyplot as plt


# Parameters

handshake_ways = 3  # 0, 1, 2, or 3-way handshake (0: CSMA, 1: ADAPT-1, 2: CSMA/CA,
3: ADAPT-3)

nodeNum = 50        # Number of client nodes

Tia = 200           # [us] Mean inter-arrival time

bandwidth = 90e9;
```

```
# Load simulation results

filename = f"result_3way_50n_200us_1.txt"

data = np.loadtxt(filename, dtype=int)



# Compute metrics

data = data.T

file_path="/home/ybab/ns-allinone-3.39/ns-3.39/contrib/thz/results/result_3way_50n_200us_1.txt"

with open(file_path,'r') as file:

    da=[list(map(int, line.split())) for line in file]



#da = da[da[:, 1].argsort()]

da = sorted(da, key=lambda x: x[1])

# Calculate delay vs. number of nodes

delays = np.zeros(nodeNum)

for n in range(1, nodeNum + 1):

    # Select only data from node n and successful packets

    succ_data = np.array([row for row in da if row[0] == n and row[3] == 1])

    delays[n - 1] = np.mean(succ_data[:, 2])

print(delays)



# Plot delay vs. number of nodes

plt.plot(range(1, nodeNum + 1), delays, marker='o')

plt.xlabel('Number of Nodes')
```

```
plt.ylabel('Average Packet Delay (us)')
plt.title('Average Packet Delay vs Number of Nodes')
plt.grid(True)
plt.show()

jitters = np.zeros(nodeNum)
for n in range(1, nodeNum + 1):
    # Select only data from node n and successful packets
    succ_data = np.array([row for row in da if row[0] == n and row[3] == 1])
    jitters[n - 1] = np.std(succ_data[:, 2])

# Plot jitter vs. number of nodes
plt.figure()  # Create a new figure for the jitter plot
plt.plot(range(1, nodeNum + 1), jitters, marker='o', color='r')
plt.xlabel('Number of Nodes')
plt.ylabel('Packet Jitter (us)')
plt.title('Packet Jitter vs Number of Nodes')
plt.grid(True)
plt.show()


even_latencies = np.zeros(nodeNum)
for n in range(1, nodeNum + 1):
    # Select only data from node n and successful packets
    succ_data = np.array([row for row in da if row[0] == n and row[3] == 1])
```

```
    sorted_delays = np.sort(succ_data[:, 2])

    even_latencies[n - 1] = np.mean(sorted_delays[1::2])  # Take every second element for
even latency


# Plot even latency vs. number of nodes

plt.figure()  # Create a new figure for the even latency plot

plt.plot(range(1, nodeNum + 1), even_latencies, marker='o', color='g')

plt.xlabel('Number of Nodes')

plt.ylabel('Even Latency (us)')

plt.title('Even Latency vs Number of Nodes')

plt.grid(True)

plt.show()



pdr_values = np.zeros(nodeNum)

for n in range(1, nodeNum + 1):

    total_packets = sum(row[0] == n for row in da)

    successful_packets = sum((row[0] == n) and (row[3] == 1) for row in da)

    pdr_values[n - 1] = successful_packets / total_packets if total_packets > 0 else 0


# Plot PDR vs. number of nodes

plt.figure()  # Create a new figure for the PDR plot

plt.plot(range(1, nodeNum + 1), pdr_values, marker='o', color='b')

plt.xlabel('Number of Nodes')

plt.ylabel('Packet Delivery Ratio (PDR)')

plt.title('PDR vs Number of Nodes')
```

```
plt.grid(True)
plt.show()



qos_values = np.zeros(nodeNum)
for n in range(1, nodeNum + 1):
    total_packets = sum(row[0] == n for row in da)
    successful_packets = sum((row[0] == n) and (row[3] == 1) for row in da)

    avg_latency = np.mean([row[2] for row in da if row[0] == n and row[3] == 1])
    packet_jitter = np.std([row[2] for row in da if row[0] == n and row[3] == 1])
    pdr = successful_packets / total_packets if total_packets > 0 else 0

    # Define your QoS metric based on latency, jitter, and PDR
    qos_values[n - 1] = 1 / (avg_latency + packet_jitter) * pdr

# Plot QoS vs. number of nodes
plt.figure()  # Create a new figure for the QoS plot
plt.plot(range(1, nodeNum + 1), qos_values, marker='o', color='purple')
plt.xlabel('Number of Nodes')
plt.ylabel('Quality of Service (QoS)')
plt.title('QoS vs Number of Nodes')
plt.grid(True)
plt.show()
```

```
# Throughput calculation
throughput_node = np.zeros(nodeNum)  # [Gbps] Throughput of each node
for n in range(1, nodeNum + 1):
    # Select only data from node n and successful packets
    succ_data = data[(data[:, 0] == n) & (data[:, 3] == 1)]

    if succ_data.shape[0] > 0:
        throughput_node[n - 1] = np.mean(succ_data[:, 1] * 8 / succ_data[:, 2])
    else:
        throughput_node[n - 1] = 0

throughput = np.mean(throughput_node)  # [Gbps] Average throughput

# Discard rate
discard_rate = np.sum(data[:, 4]) / data.shape[0]

# [us] Average packet time
succ_data = data[data[:, 3] == 1]
average_packet_time = np.mean(succ_data[-round(len(succ_data) / 10):, 2]) * 1e-3 * 1e3
spectral_efficiency = throughput*10 / (bandwidth * 1e-9)

# Print out results
print(f"Throughput = {throughput*10} Tbps")
print(f"Spectral Efficiency = {spectral_efficiency} bps/Hz")
```

```
print(f"Discard rate = {discard_rate}")

print(f"Average packet time = {average_packet_time:.2f} ps")


# Print out latency results

average_latency = np.mean(delays)  # [us] Average latency

print(f'Average latency = {average_latency} us')
age latency = {average_latency} us')
```