ENSC833:
Network Performance and Protocols
Spring 2022

Final Project
Performance Evaluation and Analysis of Wi-Fi 6 with $ns-3$

https://www.sfu.ca/~yza585/project.html

Team 6

JasonEpp	*ChuanJiang*	*YanyanZhang*
301447933	301217559	301439027
jea24@sfu.ca	*chuan_jiang@sfu.ca*	*yza585@sfu.ca*

April 14, 2022

Contents

Abstract

With the increase in multimedia content and gaming, the bandwidth used by a household has increased considerably. With HD video and gaming using up to 25Mbps per host [1] [2], and with services like cloud gaming demanding of high network performance with a tolerance of delay ranging from 300ms to 25ms[3], the improvement on the network devices and network protocols and firmware are always under the development. Because of these requirements, there is a need for high bandwidth, high performance wireless router systems for the home network to handle more and more traffics in many cases. Wi-Fi 6, also known as 802.11ax, is the newest implemented IEEE 802.11 standard wireless network protocol that has been used in devices in the market. Its maximum link speed is 9608Mbps [4] or 9.6Gbps [5]. Wi-Fi 6 is much faster than Wi-Fi 5, however speed wasn't necessary it's main goal. Wi-Fi 6 was created to make a wifi network perform better when a lot of devices are connected. Wi-Fi 6 has MU-MIMO (Multi-User, Multiple-Input Multiple-Output) technology and higher bandwidth channels, therefore shows much better performance [6]. Our goal for this project is to see how is Wi-Fi 6 implemented in $ns-3$ and also test some scenarios to see its performance. In our test scenarios, we will use $ns-3$ simulate a geographic region covered by Wi-Fi 6 with several game users and other stream users who share the same network to see how throughput and latency is affected.

1 Introduction

1.1 Introduction of Wi-Fi 6

Wi-Fi 6 is released in 2019, and is the latest wireless standard that is used in wireless devices and is the successor to the 802.11ac Wi-Fi standard which is known as Wi-Fi 5 [7]. Wi-Fi 6 is much faster than Wi-Fi 5, however speed was not necessary its main goal. Wi-Fi 6 was created to make a wireless network perform better when a lot of devices are connected.

There are some key capabilities that sets Wi-Fi 6 successor to Wi-Fi 5. Wi-Fi 6 supports Multi-User Multiple Input, Multiple Output (MU-MIMO), which allows more data to be transferred at one time, enabling Access Points (APs) to concurrently handle more devices. This is done by some advanced technologies. The first technology is 1024 Quadrature Amplitude Modulation mode (1024-QAM, MCS[10,11]) [6]. It can increase throughput for emerging, bandwidth intensive uses by encoding more data in the same amount of spectrum. The Second one is Orthogonal Frequency Division Multiple Access, also known as OFDMA. This technology can effectively share channels to increase network efficiency and lower latency for both uplink and downlink traffics in high demand environments. In addition, Wi-Fi 6 is more environment friendly as it has Target Wake time(TWT), which significantly improves network efficiency and device battery life, including IoT devices and is more resilient to overlapping APs' interference signal, which is done by Basic Service Set coloring (BSS coloring), distinguishing another network from its own and disregards their interfering distraction [6]. Figure 1 shows the main comparison between Wi-Fi 6 and Wi-Fi 5. Wi-Fi 6 has many application scenarios in reality. For example, 802.11ax supports multi-user high-speed concurrency, and can support ultra-high-definition video applications in user-intensive scenarios such as homes. So the residential Wi-Fi 6 server can provide reliable Internet access and connectivity of devices at home stadiums, and other public places [8].

TABLE 1: COMPARING WI-FI-5 AND WI-FI 6 STANDARDS

Parameter	Wi-Fi 5 (802.11ac)	Wi-Fi 6 (802.11ax)
Frequency	5 GHz	2.4 and 5.0 GHz
Bandwidths (channels)	20, 40, 80+80, 160 MHz	20, 40, 80+80, 160 MHz
Access	OFDM	OFDMA
Antennas	MU-MIMO (4 × 4)	MU-MIMO (8 × 8)
Modulation	256QAM	1024QAM
Maximum data rate	3.5 Gb/s	9.6 Gb/s
Maximum users/AP	4	8

Figure 1: networking wifi5 vs wifi6

1.2 Wi-Fi 6 Implementation in $ns-3$

1.2.1 Implemented Features Under Review

We use $ns-3$ simulator as a tool to evaluate network performance, as Wi-Fi 6 is being implemented as one of the standards supported since ns-3.34. With the latest release of ns-3.35, Wi-Fi 6 supports regarding data rates, configuration and some extension to transmission and receiving modules are in place for user to test. $ns-3$ nodes can contain a collection of NetDevice objects, much like an actual computer contains separate interface cards for Ethernet, Wi-Fi, Bluetooth, etc. By adding WifiNetDevice objects to $ns-3$ nodes, one can create models of 802.11-based infrastructure and ad hoc networks [9].

Although Wi-Fi 6 has been implemented and adopted in $ns-3$, and has a number of examples and papers and examples utilizing the Wi-Fi 6 implementation, since it is a relatively new technology the number of simulation papers are still limited compared to that of other technologies implemented in $ns-3$. There are still very few papers exploring the results of MCS and its self-adaptive function dealing with user-intense environment.

1.2.2 Scope and Limitations

The IEEE 802.11 standard [ieee80211] is a large specification, and not all aspects are covered by $ns-3$; the documentation of $ns-3$'s conformance by itself would lead to a very long document. This section attempts to summarize compliance with the standard and with behavior found in practice [9]. The physical layer and channel models operate on a per-packet basis, with no frequency-selective propagation nor interference effects when using the default YansWifiPhy model. Directional antennas are also not supported at this time. For additive white Gaussian noise (AWGN) scenarios, or wideband interference scenarios, performance is governed by the application of analytical models (based on modulation and factors such as channel width) to the received signal-to-noise ratio, where noise combines the effect of thermal noise and of interference from other Wi-Fi packets. Interference from other wireless technologies is only modeled when the SpectrumWifiPhy is used. The following details pertain to the physical layer and channel models:

1. 802.11ax MU-RTS/CTS is not yet supported; beamforming is not supported.
2. The wifi manager always selects the lowest basic rate for management frames.

1.3 Motivation and Scope

Wi-Fi 6 has been recently implemented with the ability to have higher bandwidth channels, faster modulation schemes, and higher resilience to interference. Wi-Fi-6 standard was proposed only in recent years, but it has occupied a large market share and its influence on routers is increasing rapidly. It is necessary to study the advantages and principles of Wi-Fi-6 protocol and simulate it in a specific environment. Although Wi-Fi 6 has been implemented and adopted in $ns-3$, and has a number of examples and papers and examples utilizing the Wi-Fi 6 implementation, since it is a relatively new technology the number of simulation papers are still limited compared to that of other technologies implemented in $ns-3$. There are still very few papers exploring the results of MCS and its self-adaptive function dealing with user-intense environment.

To summarize, the Wi-Fi ecosystem has undergone significant changes recently, which is worthy of a new study. Therefore, under this premise,our motivation is to assess the performance of Wi-Fi 6 for use in local area networks with $ns-3$ simulation environment. This project is to understand Wi-Fi 6, some main difference between Wi-Fi 6 and Wi-Fi 5. We constructed two test scenarios based on two different research interests. The first one is MCS evaluation, it will focus on the MCS implementation in $ns-3$ for Wi-Fi 6 performance and compare it with the expected value. In the mean time, The mechanism and performance of OFDMA would be observed and analysed. The second test case is about simulation performance in terms of throughput with some basic settings,and evaluation on frequency band performance. This one simulates point to point transmission and measures the Actual Throughput vs Ideal.

2 MCS Evaluation

2.1 MCS Explained and Motivation

The Modulation Coding Scheme (MCS) index is a metric based on several parameters of a Wi-Fi connection between two stations. As wireless devices are working in a radio frequency media, every single transmission will be reflected by the nature of how electromagnetic wave is propagating through the air. MCS index is a grouping of variety factors that will affect the performance of wireless devices in the form of data rate. It depends on the modulation type, the coding rate, the number of spatial streams, the channel width, and the guard interval. With the factory default ability, MCS index is also the guidance sets the data rate between the access points and clients. MCS is indexed from 0-11 in Wi-Fi 6 to organize different combinations in a fashion of increasing throughput. These combinations and their data rate are shown in [10].

The list of things that affects the data rate are listed below with explanation [11].

1. **Modulation Type** defines the phase and amplitude required for bit computing, from BPSK to QPSK and from 16-QAM to 256-QAM and now with 1024-QAM in new Wi Fi 6.
2. **Coding Rate** is referred as rate of bits transferred and Forward Error Correction. Minimizing the coding scheme would entail sending the data faster while losing robustness.
3. **Spatial Streams** represents the number of independent data streams are used. Higher values (e.g., 4 streams) increase data rates, but are susceptible to noise and interference.
4. **Channel Width** is the smaller bands within Wi-Fi frequency bands that are used by your wireless network to send and receive data.
5. **Guard Interval** is the waiting time or pause between each packet transmission. Wi-Fi 6 guard interval duration starting from 800ns to 3200ns.

$ns-3$ has recently implemented and released the basic Wi-Fi 6 standard in Wi-Fi module. Based on paper[5],[12] and [13], validation of the calculation of MCS implemented in 802.11ax standard is relatively simple but still can be considered as valuable work, as the index has increased to another level of complexity compared to Wi-Fi 5 and is affecting other key features of Wi-Fi 6. It will be a good practice for us to simulate a simple network and test out the throughput to be the starting point of this project.

2.2 MCS's Effects on Throughput Rate

The first experiment we want to do is to create a simple network that can produce traffic and with a period of simulation time, we can then calculate the throughput for the link. This is also a validation test of our network topology builder that we have to start with, since none of us had the experience before with $ns-3$.

The goals for this experiment are listed:

1. Test MCS implementation in $ns-3$ against Wi-Fi 6 IEEE 802.11ax standard with just the simplest network.
2. Learn how to use $ns-3$ simulator to build a simulation network and generate desired data for analysis.
3. Get familiar with coding interfaces for a simulator instead of graphical interface.

The first test is a script validation test. We started with the simplest case where we only have one server and one client. We placed an AP at a fixed position and placed a station within the AP's service range as shown in Figure 2. We want to generate traffic and run a period of time which is just enough to calculate the throughput. Since the MCS index of Wi-Fi 6 has both the OFDM section and OFDMA section which is for multi user and spatial reuse case, it is better to separate them into three sub tests as listed:

1. Test single user case with OFDM MCS Index.
2. Test multi user case with OFDM MCS Index.
3. Test multi user case with OFDMA MCS Index.

2.2.1 Test 1 SU Throughput Test, NO-OFDMA

Test 1 is where we test only the single client case. The algorithm we used is also implemented as simple as possible, we made a script that will step through all MCS settings with different channel band width and different guard interal period to execute a traffic simulation and calculate throughput of each setting. Wi-Fi 6 has three spacial stream settings and with the increasing number of spacial stream, the throughput will increase proportionally. Since this validation test is meant to help use learn how to use $ns-3$ as well as understanding the basic concept of WI-Fi module layers, we will only run the full validation test with numder of spatial stream set to 1.

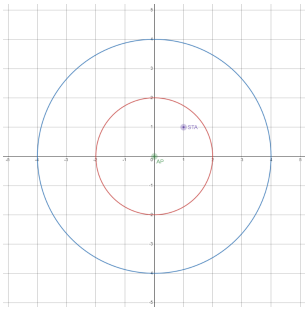


Figure 2: Topology Illustration

As shown on Figure 2, AP is sitting in the origin and we placed one station in side the AP's covered service range of distance (represented by the blue circle, and the station is randomly moving inside the red circle.

Pseudocode for Testing Throughput for Each MCS Setting:

Algorithm 1 Pseudocode for Testing Throughput for Each MCS Setting

```
Test 1 Topology setup and initialization;
Setup log file header, simulation interval;
LOOP    MCS = 0 − 11
  LOOP    Channel Bandwidth = 20 − 160 MHz
    LOOP    Guard Interval = 3200 − 800 ns
      phyModel selection; //Yans or Spectrum
      stream generation;
      Mobility model selection;
      node installation;
      IP address assignment;
      simulation scheduler;
      simulation run;
      simulation destroy;
      Collect data and calculation done;
```

To implement this test with $ns-3$, we started with the example "wifi-he-network.cc" provided by nsnam. As this is the first time we code in C++, it is a very time consuming process to get used to the simulator. We went through all "$ns-3$ Tutorial" eamples and still struggling with setting the parameters and compose a new topology.

The following are the modified parts for doing the desired test and also enabled user reading about the simulation process.

```
//include header files for monitor traffics and read out results of each simulation run
    from the console interface
#include "ns3/wifi-acknowledgment.h"
#include "ns3/rng-seed-manager.h"
#include "ns3/he-ru.h"
#include "ns3/flow-monitor.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/trace-helper.h"
#include <iostream>
#include <fstream>
```

```
//adding new varables that can be passed from commandline
  bool actprobing {false}; // Active Probing, normally set to false
  bool qos {false}; //Quality of Service Support
```

```
  bool bsrp {false}; //buffer status report poll
  bool mimo {false}; //MIMO enable
  int mcs {5}; // -1 indicates an unset value
  uint32_t payloadSize = 540; // must fit in the max TX duration when transmitting at
    MCS 0 over an RU of 26 tones
  std::size_t nStations {10};
  std::string dlAckSeqType {"NO-OFDMA"};
  std::string phyModel {"Spectrum"};
```

```
//adding new varables that can be helped from command --help
  cmd.AddValue ("actprobing", "UDP if set to 1, TCP otherwise", actprobing);
  cmd.AddValue ("qos", "Enable/disable Quality of Servic Support", qos);
  cmd.AddValue ("bsrp", "Enable/disable Buffer Status Report Poll", bsrp);
  cmd.AddValue ("mimo", "Enable/disable MIMO", mimo);
  cmd.AddValue ("mcs", "if set, limit testing to a specific MCS (0-11)", mcs);
  cmd.AddValue ("payloadSize", "The application payload size in bytes", payloadSize);
  cmd.AddValue ("nStations", "Number of non-AP HE stations", nStations);
  cmd.AddValue ("dlAckType", "Ack sequence type for DL OFDMA (NO-OFDMA, ACK-SU-FORMAT,
    MU-BAR, AGGR-MU-BAR)", dlAckSeqType);
  cmd.AddValue ("phyModel", "PHY model to use when OFDMA is disabled (Yans or Spectrum).
     If OFDMA is enabled then Spectrum is automatically selected", phyModel);
```

```
//print setting on screen inorder to log data for records
  std::cout << "UDP" << "\t" << "RTS" << "\t" << "EBA" << "\t" << "MCS" << "\t" << "ACK"
     << "\t\t" << "phyModel" << '\n';
  std::cout << udp << "\t" << useRts << "\t" << useExtendedBlockAck << "\t" << mcs << "\
    t" << dlAckSeqType << "\t" << phyModel << std::endl;
  std::cout << "Active Probing" << "\t" << "QosSupported" << "\t" << "BSRP" << "\t" << "
    MIMO" << '\n';
  std::cout << actprobing << "\t\t" << qos << "\t\t" << bsrp << "\t" << mimo << std::
    endl;
  std::cout << "Freq" << "\t" << "Dist" << "\t" << "SimTime" << "\t" << "nStations" << "
    \t" << "payloadsize" <<'\n';
  std::cout << frequency << "\t" << distance << "\t" << simulationTime << "\t" <<
    nStations << "\t\t" << payloadSize <<std::endl;
  std::cout << "If set, simulation fails if throughput is outside the range" <<'\n';
  std::cout << "minExpectedThroughput" << "\t---\t" << "maxExpectedThroughput" <<'\n';
  std::cout << minExpectedThroughput << "\t\t\t---\t" << maxExpectedThroughput <<std::
    endl;
```

```
//configure settings for some additional settings for Quality of service, multi user and
     more.
          if (mimo)
            {
            phy.Set ("Antennas", UintegerValue (3));
            phy.Set ("MaxSupportedTxSpatialStreams", UintegerValue (2));
            phy.Set ("MaxSupportedRxSpatialStreams", UintegerValue (2));
          }

          if (qos)
                    {
            mac.SetType ("ns3::StaWifiMac",
                   "Ssid", SsidValue (ssid),
                   "QosSupported", BooleanValue (qos),
                                "BE_BlockAckThreshold", UintegerValue (2),
                                "ActiveProbing", BooleanValue (actprobing));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            staDevices = wifi.Install (phy, mac, wifiStaNodes);

            mac.SetType ("ns3::ApWifiMac",
                   "Ssid", SsidValue (ssid),
                   "BeaconGeneration", BooleanValue (true),
                   "BeaconInterval", TimeValue (Seconds (0.25)),
                   "EnableBeaconJitter", BooleanValue (false),
                   "QosSupported", BooleanValue (qos));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            apDevice = wifi.Install (phy, mac, wifiApNode);
          }
          else
          {
            mac.SetType ("ns3::StaWifiMac",
                   "Ssid", SsidValue (ssid),
                   "ActiveProbing", BooleanValue (actprobing));
```

```
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            staDevices = wifi.Install (phy, mac, wifiStaNodes);

            mac.SetType ("ns3::ApWifiMac",
                                 "EnableBeaconJitter", BooleanValue (false),
                                 "Ssid", SsidValue (ssid));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            apDevice = wifi.Install (phy, mac, wifiApNode);
          }
```

```
//configure mobility to move inside the range
        MobilityHelper mobility;
        Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>
    ();
        positionAlloc->Add (Vector (0.0, 0.0, 0.0));

        double rad = 2.0;
        double angle = 360 / nSta;
        double x_pos = 0;
        double y_pos = 0;
        for (int i = 0; i < nSta; ++i ) {
        // adding devices to a radius for this test
        x_pos = rad*sin(angle*i*(3.14159/180));
        y_pos = rad*cos(angle*i*(3.14159/180));
        positionAlloc->Add(Vector(x_pos, y_pos, 0.0));
        NS_LOG_INFO("Setting STA position to " << Vector(x_pos, y_pos, 0.0));
        }
        mobility.SetPositionAllocator (positionAlloc);
```

```
//configure flowmonitor function to lod trace simulated and more.
              Ptr<FlowMonitor> flowMonitor;
              FlowMonitorHelper flowHelper;
              flowMonitor = flowHelper.InstallAll();
              phy.EnablePcapAll ("wifi-he-ofdma");
```

The following showed the results from one example run.

```
yanyan_zhang@DESKTOP-RQBUT6E:~/workspace/bake/source/ns-3.35$ ./waf --run "scratch/wifi-
    he-network-MCS --udp=1 --payloadSize=540 --mcs=5 --nStations=10" $CWD
Waf: Entering directory `/home/yanyan_zhang/workspace/bake/source/ns-3.35/build/debug'
[3053/3147] Compiling scratch/wifi-he-network-MCS.cc
[3068/3147] Compiling scratch/wifi-he-network4.cc
[3099/3147] Linking build/debug/scratch/wifi-he-network-MCS
[3100/3147] Linking build/debug/scratch/wifi-he-network4
Waf: Leaving directory `/home/yanyan_zhang/workspace/bake/source/ns-3.35/build/debug'
Build commands will be stored in build/debug/compile_commands.json
'build' finished successfully (25.877s)
UDP     RTS     EBA     MCS     ACK             phyModel
1       0       0       5       NO-OFDMA        Spectrum
Active Probing  QosSupported    BSRP    MIMO
0               0               0       0
Freq    Dist    SimTime nStations       payloadsize
5       1       100     10              540
If set, simulation fails if throughput is outside the range
minExpectedThroughput   ---     maxExpectedThroughput
0                       ---     0
MCS value               Channel width           GI                      Throughput
5                       20 MHz                  3200 ns                 38.0791 Mbit/s
5                       20 MHz                  1600 ns                 42.463 Mbit/s
5                       20 MHz                  800 ns                  44.1552 Mbit/s
5                       40 MHz                  3200 ns                 63.7822 Mbit/s
5                       40 MHz                  1600 ns                 69.6997 Mbit/s
5                       40 MHz                  800 ns                  81.4229 Mbit/s
5                       80 MHz                  3200 ns                 110.791 Mbit/s
5                       80 MHz                  1600 ns                 143.166 Mbit/s
5                       80 MHz                  800 ns                  150.195 Mbit/s
5                       160 MHz                 3200 ns                 176.084 Mbit/s
5                       160 MHz                 1600 ns                 234.888 Mbit/s
5                       160 MHz                 800 ns                  243.405 Mbit/s
```

As each of the simulation setting takes a long time to run and we have a huge index that we would like to compare with, we limited our simulation time for only 100 seconds between each simulation run and destroy for the above test results displayed.

We also ran a full test that go through the entire MCS settinngs for *spatial stream number* $= 1$. After we collected all the results, we analyzed the results plotted. Test setting are listed in Table 1.

Table 1: Parameters for Throughput Testing for MCS

Fixed Parameter	Value
Network radius	1 m
AP Transmission Power	16.0206 dBm
STA Transmission Power	16.0206 dBm
Frequency	5GHz
Number of STAs	1
Traffic Type	UDP
Application Layer Traffic Rate	100 Mbps
Access Category	BE
Beacon Period	0.1024s
Number of Antennas	1
Simulation Time	5s
Propagation Loss Model	Log Distance Propagation Loss Model

First, we look at the MCS effect on throughput.The results in Figure 3, Figure 4, Figure 5 are plotting the Throughput against channel bandwidth settings, which are showing as expected, and can be concluded as listed:

1. The higher the index, the faster data rate.
2. The wider the bandwidth, the faster data rate.
3. The shorter the guard interval, the faster the data rate.

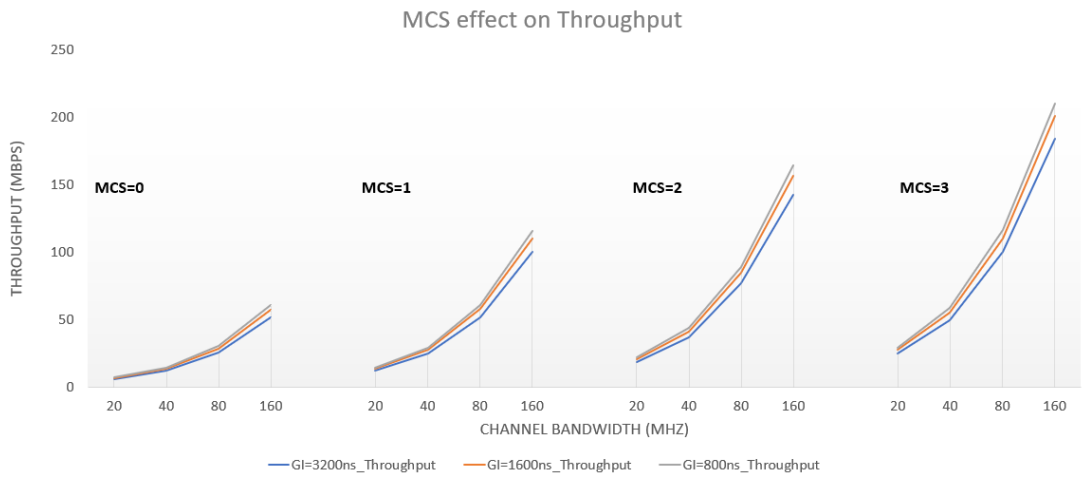


Figure 3: Throughput Plot for $MCS = 0$ to $MCS = 3$

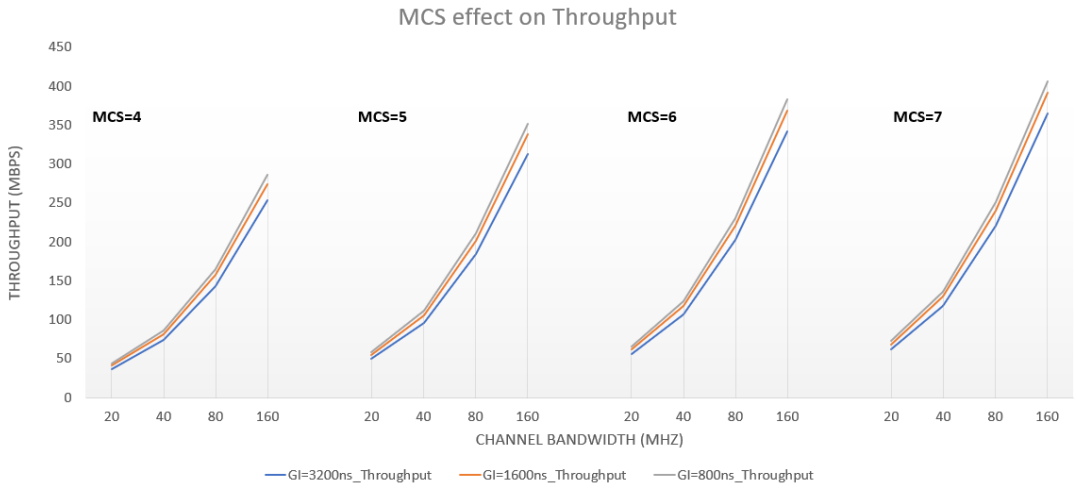


Figure 4: Throughput Plot for $MCS = 4$ to $MCS = 7$

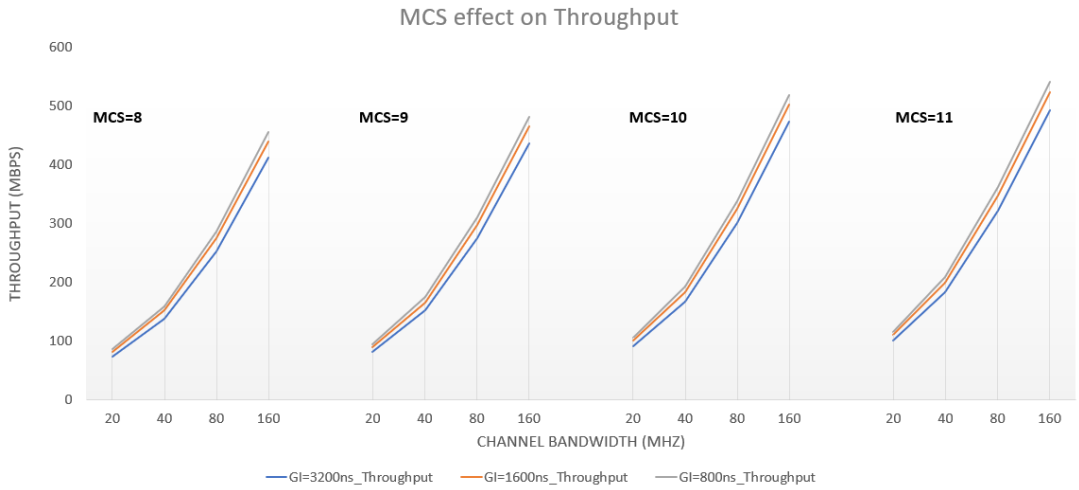


Figure 5: Throughput Plot for $MCS = 8$ to $MCS = 11$

We also compared with the calculated table[10] provided by semfionetworks.com. Our simulation results are in general slower than the expected. One sample of the comparison is shown in Figure 6. The possible route of cause could be the simulation time was too short that the traffic is not yet in the saturation for the calculation. As the total run time for this test is about 2 days, we decided to not going further to find out about the simulation error. Since the topology and test are relatively easy, the cause of the misalignment could also be related to $ns-3$'s implementation about Wi-Fi 6 regarding MCS.

MCS Index	Spatial Stream	Modulation	Coding	20MHz		
				0.8μs GI	1.6μs GI	3.2μs GI
0	1	BPSQ	1/2	8.6	8.1	7.3
1	1	QPSK	1/2	17.2	16.3	14.6
2	1	QPSK	3/4	25.8	24.4	21.9
3	1	16-QAM	1/2	34.4	32.5	29.3
4	1	16-QAM	3/4	51.6	48.8	43.9
5	1	64-QAM	2/3	68.8	65.0	58.5
6	1	64-QAM	3/4	77.4	73.1	65.8
7	1	64-QAM	5/6	86.0	81.3	73.1
8	1	256-QAM	3/4	103.2	97.5	87.8
9	1	256-QAM	5/6	114.7	108.3	97.5
10	1	1024-QAM	3/4	129.0	121.9	109.7
11	1	1024-QAM	5/6	143.4	135.4	121.9

(a) Calculated Data Rate

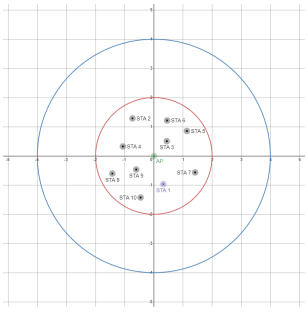
MCS	20MHz		
	800 ns	1600 ns	3200 ns
0	7.2	6.8	6.1
1	14.6	13.8	12.4
2	21.9	20.7	18.6
3	29.4	27.7	24.9
4	44.0	41.5	37.3
5	58.7	55.3	49.8
6	65.8	62.2	56.0
7	72.6	68.9	62.3
8	86.0	81.5	73.9
9	94.6	89.8	81.4
10	105.3	99.9	90.7
11	115.9	110.1	100.1

(b) Simulated Data Rate

Figure 6: MCS Theoretical VS Simulated Results, $Channel\ Bandwidth = 20MHz$

2.2.2 Test 2 MU Throughput Test, NO-OFDMA

Test 2 is to increase the number of stations for the same test, and we did two tests with $MCS = 5$ and $MCS = 11$. This time, we plotted the processed results of throughput as a results of changing guard interval period. As shown in Figure 8, there are chances that association requests collide and hence the throughput may be lower than expected. Overall, the data we collected are similar to the previous test case.



As shown on Figure 7, AP is sitting in the origin and we placed one station in side the AP's covered service range of distance (represented by the blue circle, and all the stations are randomly moving inside the red circle.

Figure 7: Topology Illustration when $nSta = 10$

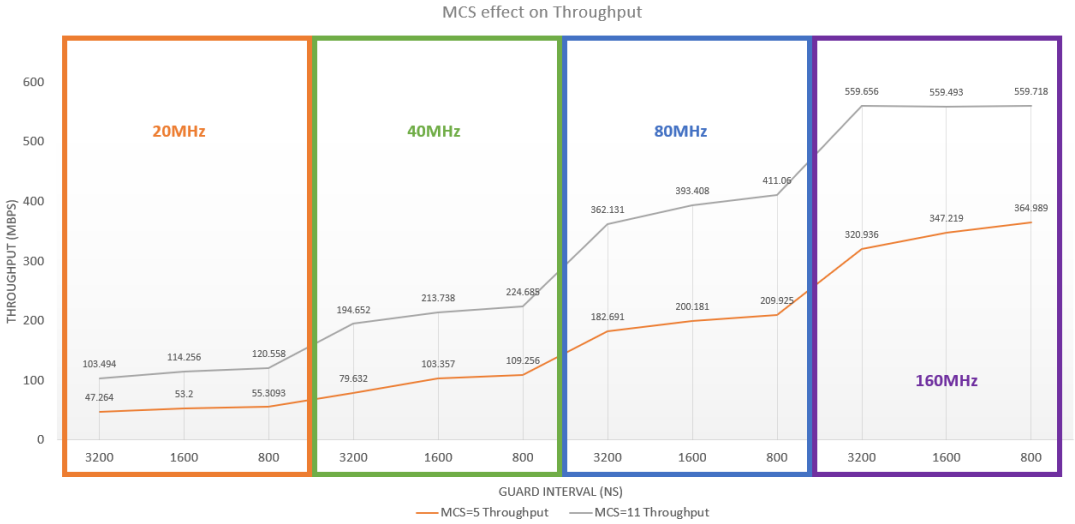


Figure 8: Throughput Plot for $MCS = 5$ and $MCS = 11$

Here we saw another fact about the MCS. As the data bandwidth increases, the throughput is jumping from one level to another. Their performance can be categorized with different class. This design of MCS index is due to the nature that wireless devices operate in an radio frequency environment as the RF media that our devices are working in is reflected in every single transmission. MCS helps the devices to evaluate the quality of the RF environment. Every transmitter device, whether it be an AP or client, it is going to make an internal decision on which MCS it shall use. It will also inform its AP to "discuss" about the data rate when connects to the internet.

By default, when we have more than one user in one network, Wi-Fi 6 devices are smart enough to change to multi user mode with OFDMA enabled. However, the above script is not capable of switching to OFDMA enabled simulation environment due to the lack of knowledge of us in how to configuring the topology and devices setup. As we started this project as a result of failing the other, the time left for us to implement the OFDMA selection function in $ns - 3$ seems impossible within the time frame. Therefore, we stopped going further in this direction of testing the MCS settings for OFDMA enabled.

2.3 The OFDMA Enable Validation with $ns - 3$

Wi-Fi 6, breaks with traditional previous iteration of Wi-Fi that was mostly aimed at increasing pair devices' throughput. Instead of focusing on speed, the idea of Wi-Fi 6 is to accommodate more and more

devices pair access point, and the metric we are trying to target is increasing area throughput. Some new technologies, especially Spatial Reuse, has given the possibility of a density populated network where APs are enclose proximity one of another. Spatial reuse involves letting devices transmit by tackling their transmission level, even making the co-existence between APs which are in range of others possible.

OFDMA is an enhancement over OFDM which was a single-user transmission. When a signal is sent or received, it is done with one device. In OFDMA, it allows multiple access which means simultaneous transmissions to/from multiple devices. As Wi-Fi 6 has changed the layer settings in many aspects, OFDMA implementation is not yet supported by official release, however, it is available in ns-3-dev and is currently under review. The paper[12], the authur has ran a basic simulation validation for the recently implemented OFDMA function in $ns-3$. Starting from here, we want to see if we can adopt their work and altered for our project use. The goal of this test is simple and easy, we want to see with our current understanding of Wi-Fi 6, whether or not could we enable the OFDMA option and do some tests with it.

We used the github package "Validation of the $ns-3$ 802.11ax OFDMA Implementation" as the $ns-3$ Wi-Fi 6 patch to see if we can enable OFDMA. This package is including all the necessary modifications for enabling the network devices to switch mode automatically. It is again a hard time for us to apply the patch because the development kit added parameters and functions that is only applicable to Wi-Fi 6 into the whole repository which can not be recognized by our package. Most of the case, when we exam the example code, we are facing the compiling errors instead of simulation setup errors. This code is branched from ns-3-dev kit so it should be adopted by ns-3-dev instead of ns-3.35. After switching to the other package, we renamed the package as "ofdma" and applied the patch. The wifi-ofdma-validation.cc model included in the patch can be run successfully with the only raised error case of no traffic flow. The script runs as expected as we have not given the simulator any parameters that defines the network.

```
yanyan_zhang@DESKTOP-RQBUT6E:~/workspace/bake/source/ofdma$ ./waf --run "scratch/wifi-
    ofdma-validation" $CWD
Waf: Entering directory `/home/yanyan_zhang/workspace/bake/source/ofdma/build/debug'
Waf: Leaving directory `/home/yanyan_zhang/workspace/bake/source/ofdma/build/debug'
Build commands will be stored in build/optimized/compile_commands.json
'build' finished successfully (1.333s)
aborted. cond="m_flows.empty ()", msg="No traffic flow specified!", +0.000000000s -1
    file=../../scratch/wifi-ofdma-validation.cc, line=770
terminate called without an active exception
Command ['/home/yanyan_zhang/workspace/bake/source/ofdma/build/optimized/scratch/wifi-
    ofdma-validation'] terminated with signal SIGIOT. Run it under a debugger to get
    more information (./waf --run <program> --gdb").
```

The next step is to simulate with some test settings transferred to run wifi-ofdma-validation.ccA.2.5. The patch package has provided a three python script to transfer simulation parameters for running the simulation. When going through the scripts, again we were facing compiling issues as the python scripts requires some packages that we don't have. We also fixed some problems with the transfer error as the scripts provided is coded with eariler version of python3 and we have to fix these errors before we can proceed to the next step.

We finally successfully coded a python script as attached in the A.2.1 that can run the three provided processing scripts. We got some simulation results but has no time to run the analysis and plotting them. The following is the data collected for future analysis.

```
yanyan_zhang@DESKTOP-RQBUT6E:~/workspace/bake/source/ofdma/output/ofdma-validation-
    results$ ls -R
.:
data  ofdma-validation-results.json

./data:
007f1f9d-fdbd-42bb-b5bb-e79339a66699  5a75a7ff-c852-456c-84e3-4ccd065775a1  b144db79-143
    a-4b76-9774-870112c4a9db
01aa6756-3a96-4651-ad2e-1ffad3d163f8  5b5f55aa-886e-4dc3-801b-c31360169654  b1f0ede9-7
    e36-4c75-8a1b-bf35fa94c641
01b44296-3e24-412d-aa5e-2035efcf51c5  5b7831ee-fecd-494d-9947-5bcfe7ab6c70  b265dd8d-
    a763-453d-b45e-e5c7dae7244a
0214abf9-1deb-47e3-ad35-7c1c2da44b96  5bcb722b-94b4-4886-ad67-e55d18890c60  b2aa32fc-
    ef54-4b84-94f3-2c44ba26e571
02bbaa34-7fc3-42c3-88f4-ee0d7d02f361  5c9464fd-0699-44be-b7b5-390130d6e875  b2b33c71-33
    b3-449b-90b8-3f23cca5c8ca
02e0e2af-61a1-4896-a7ae-6352fcece099  5cfb87f8-9320-4ba6-9152-5d1e970f9256  b394b30c-
    c719-46fc-baef-71ee6fad4911
0646f87c-40b7-4e31-8ed4-16ffa1e8d039  5e21fb84-a73a-484e-bf94-ec285c4c5609  b40c0f9a
    -6158-41f0-b92e-918b0549338e
```

```
0704ea51-d0b0-417f-9f85-a5645cbd6f4e  5f714c2e-02d5-4912-86db-8503b1050df7  b666b1e9-595
    d-4ae1-a6a3-e4c7ebb4c126
089b6548-297e-434b-a909-f605c55bcd93  5fa35900-1f35-456a-a4c0-56c5a9578deb  b7138f5d-
    f1fa-4821-b3f1-979084a8703f
...
```

Our contribution to this test is very limited to this script. All we did is to write a python script to use the provided processing script to run $ns-3$ simulator. However, this part of the project still helped us in getting to know more about the $ns-3$ simulator. We learned how to understand the parameter transfer and handling, we learned how to troubleshoot with the underlying code errors and we also learned the relationship between the simulator and the real life network. Due to the limited time we have, we only finished the simulator built with OFDMA option enabled. The simulation results will be submitted as a proof of a successful run.

3 SNR, RSSI, and Network effects on Throughput

In this section a number of aspects related to measurements of how Wi-Fi 6 might operate in the world were studied and explored. This was initially tested with effects of various parameters on throughput between two nodes. Initially this was tested on two node networks with low level packet protocol, and then work was done to enrich these tests into a bigger real world scenario. Many of these tests attempted to maximize parameters that would allow the highest possible throughput and determine how they would work in a larger network.

3.1 Effects of SNR, and other network parameters on Throughput

Initially the effects of SNR and RSSI were tested using the modified example script $my-wifi-managerexample.cc$. This script was utilized to find the nature of how a number of test parameters worked within the PHY layer of the Wi-Fi network model. As Wi-Fi 6 is a Link/PHY layer protocol, this was required to determine how individual Wi-Fi parameters affected the performance of the system, as well as provided a backdrop for some initial benchmark testing.

3.1.1 Test Method

In this set of tests, the 802.11ax parameters were varied based on a number of different test scenarios, and the effects of the system were noted to get an understanding of the PHY parameters required to get the desired performance out of the physical layer of the network. with this tests there were only minor modifications required of the script to allow for better plotting and data capture for the scenario, and much of the work came from performing a number of individual tests to determine the affects of Guard Intervals, RSSI, Frequency Band, and the number of Spatial Streams used to generate this data. In this example a Wifi-Manager was selected to be either 'Ideal', or 'Minestrel HT', as these we the only High Throughput (HT) managers available to do throughput testing in HT mode, with the other mode being the High Efficiency (HE) mode, which makes utilization of the OFDMA to allow for more efficient multiplexing of the system. For this after minor the minor modifications were made the various tests were completed. The test parameters used in each of these cases provided in results were executed with the following command parameters.

```
./waf --run 'my-wifi-manager-example --standard=802.11ax-2.4GHz --serverNss=4 --
    clientNss=4 --serverChannelWidth=40 --clientChannelWidth=40 --
    serverShortGuardInterval=800 clientShortGuardInterval=800'
```

For the above test there were actually six main tests utilized in this example. The above command was rerun with the following variations in parameters as seen in Table 2. In this script the main functionality was that of creating a Wi-Fi Manager, and set up the MAC and PHY layers (layer 4 and layer 5 of network stack) for a low level performance test of the Wi-Fi system from an Access Point (AP) to a station (STA).

Table 2: Variations of Parameters for Throughput Testing

Band (standard)	Spatial Streams (Nss)	Channel Width (MHz)	Guard Interval (ns)
802.11ax-2.4GHz	4	20	800
802.11ax-2.4GHz	4	40	800
802.11ax-5GHz	4	80	800
802.11ax-5GHz	4	160	800
802.11ax-6GHz	4	80	800
802.11ax-6GHz	4	160	800

In this section this gave us a genera idea of the capabilities, especially at the higher throughput end as the theoretical rate based on throughput was found to be much lower than that of the expectations for each band. It was noted though that the use of 4 spatial streams for once client is generally unrealistic as Wi-Fi host devices such as computers and phones only support 1-2 spatial streams per client [14]. The channel selection for each of these was chosen as the default and the maximum channel width for each band to be utilized. The Guard Interval (GI) was selected as the least Guard Interval. The purpose of the guard interval in Wi-Fi is of the collision avoidance protocol used in the Wi-Fi network, with a shorter guard interval ultimately having a slightly higher throughput, but it will generally be expected to have a lower collision avoidance performance in theory. In this example the simulation ran a $PacketSocketHelper$ application, and the data rate was selected at 10% greater than the calculated bandwidth of the channel. The simulation utilized a $TraceCallback$ function upon the reception of each packet, and added the number of bytes in a packet to a global variable used for the tracking of the amount of data received. In addition to this an event was scheduled every step for the duration of the simulation. At every event the interval time was determined, and divided by the amount of bytes received in that interval to obtain throughput, during this event, the Signal to Noise Ratio (SNR) was also calculated by finding the ratio of the set Receive Signal Strength Indicator (RSSI) and the noise floor set for the simulation. The throughput vs SNR were added to a gnuplot data set, then the RSSI was reduced to the next increment, and the next event was scheduled for the next time interval. This was repeated until the simulation was completed. In this simulation the duration of the simulation was 10 Seconds, as these were just simple throughput measurements regarding the PHY layer without utilization of a more complex network.

3.1.2 Test Results

These tests were run with many more parameters, and variations, though the six iterations of the test used in the following plots in Figures 9-14 were selected as they showed some interesting effects of the Wi-Fi protocol. First as expected the 2.4GHz band has much lower bandwidth than the 5GHz and 6GHz bands due to their channel widths. This demonstrates the expected results that the theoretical rate for the maximum channel width vs the default channel width should be double as the channel width is double. Another interesting result of this simulation is related to the MCS value as discussed previously. In the Figures shown it is possible to see the various MCS values as the SNR increases. This is more noticeable in the rate section, and this parameter is determined by a $RateChange$ callback, from a Wi-Fi Manager event connected to the Callback function, this event being the "$/NodeList/0/DeviceList/*$ /ns3::WifiNetDevice/RemoteStationManager/$ns3::IdealWifiManager/Rate$" which was used to find the new rate used for the ideal rate plot. Another point of interest was the fact that even though the bandwidth increased with the number of spatial streams and channel widths, The higher the frequency, and the higher the channel width resulted in poorer throughput efficiency or 'goodput' of the network. when operating at 2.4GHz band with only 20MHz channel width it was found that the idea rate was 575Mbps, with an achievable rate of 425Mbps. This indicates that at it's max MCS, with a fixed distance and propagation models, that Wi-Fi node was only able to receive 74% of the maximum number of packets the AP is able to send. Considering it is a wireless network this does not seem like poor results and wireless communication often relies on robust communication at the network layer due to difficulties with interference. When increasing the frequency and the bandwidth, it was noted that this got considerably worse with greater channel width and frequency, comparing Figure 14 where the selected rate was 4.8Gbps (half of the 9.6Gbps expected if all 8 spatial streams are utilized), but the actual number of received bytes in this example was 1.3Gbps which is only around 27% of the expected number of bytes for this iteration of Wi-Fi. This result was considerably lower than expected prior to performing this testing, though these data rates are still considered quite high for many high bandwidth applications.

Another interesting behavior to not though, as that as the channel width, and the frequency increased, the resilience to noise in the system also increased. In Figure 9 the data rate very quickly drops to zero as the SRN goes below 20dB, whereas in Figure 14 that the data rate remains quite high right up to a SNR of 10dB, and as the SNR increased and the selected rate goes down the ratio of data sent vs data received gets better, making the 5GHz and 6GHz generally much faster and much more resilient to noise.

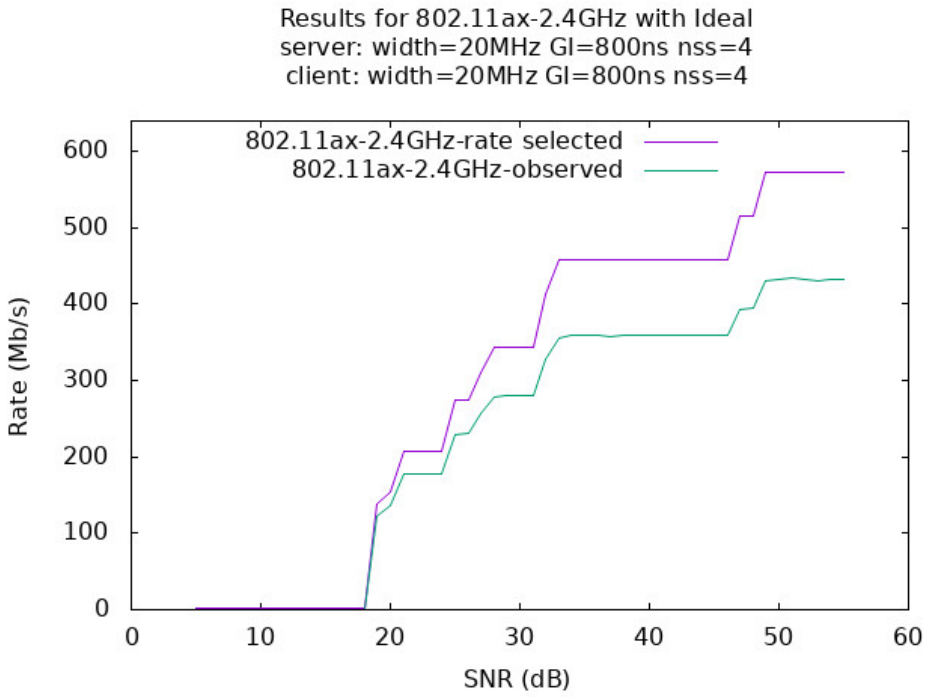


Figure 9: 2.4GHz results for parameter exploration with default channel width

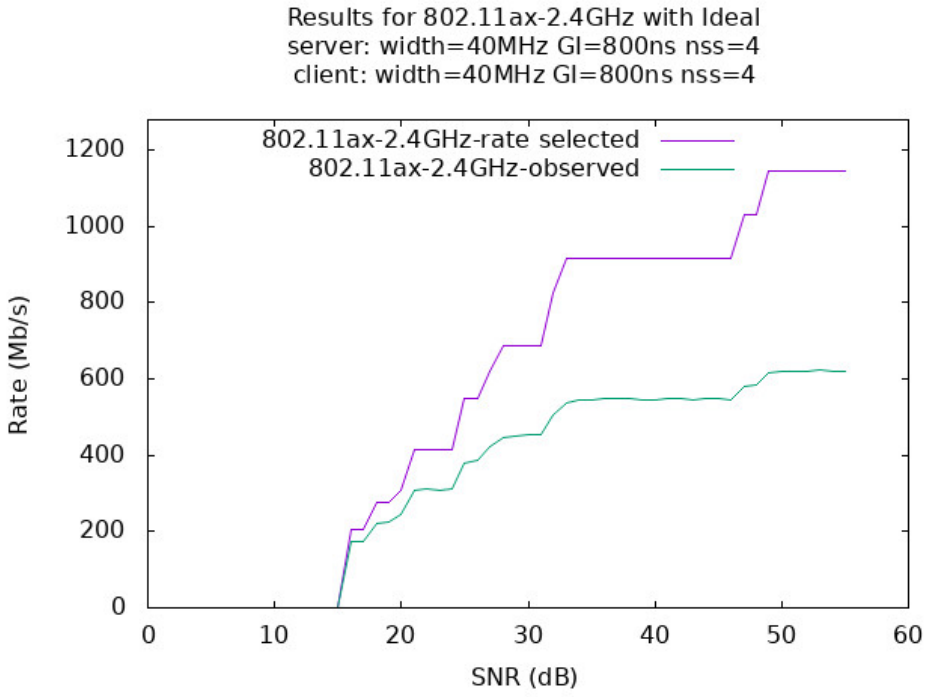


Figure 10: 2.4GHz results for parameter exploration with maximum channel width

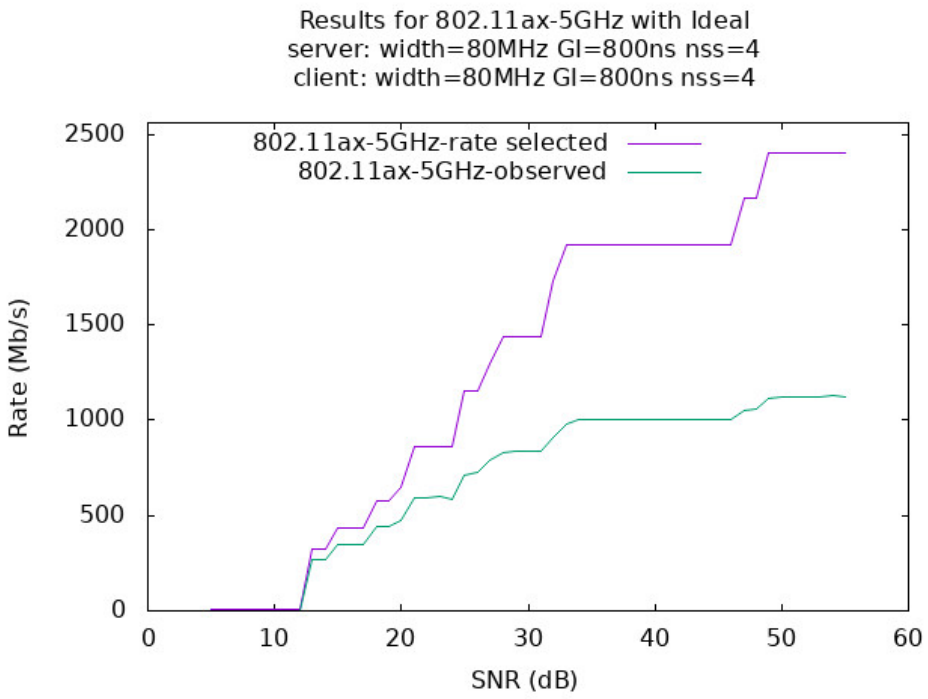


Figure 11: 5GHz results for default channel width

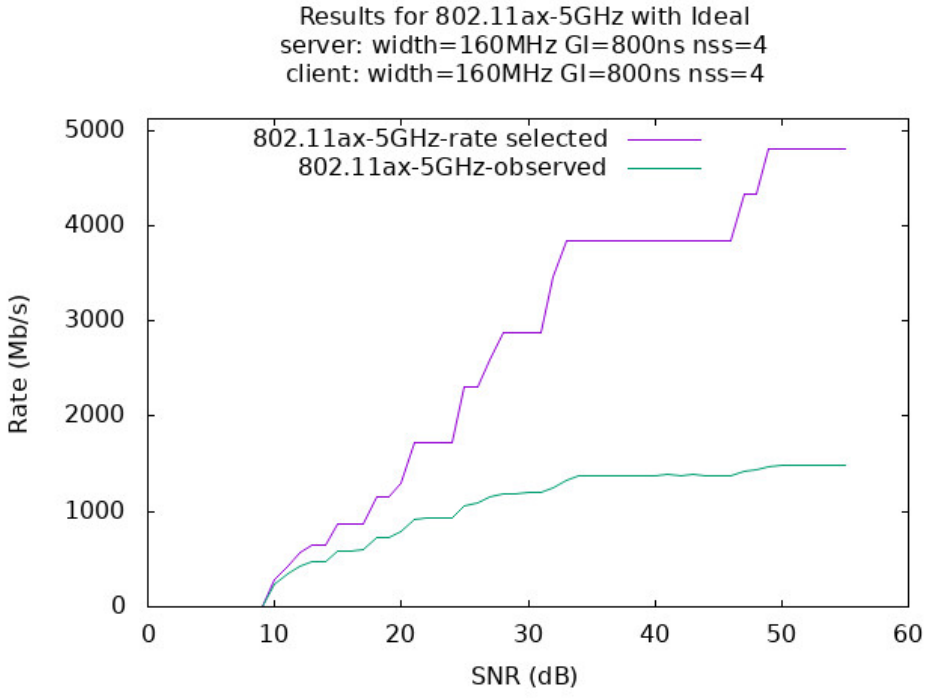


Figure 12: 5GHz results for maximum channel width

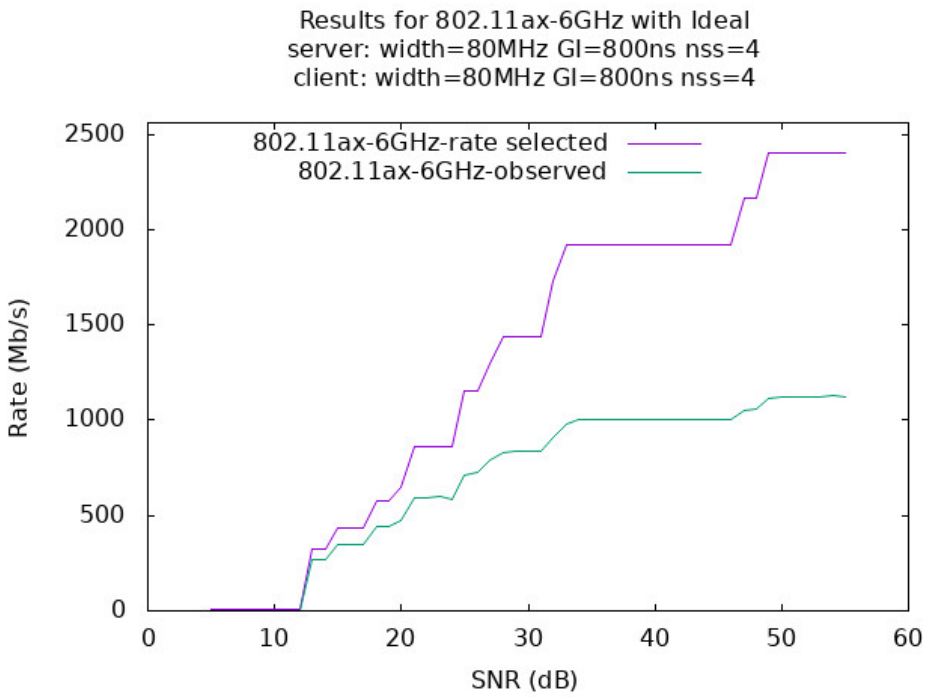


Figure 13: 6GHz results for default channel width

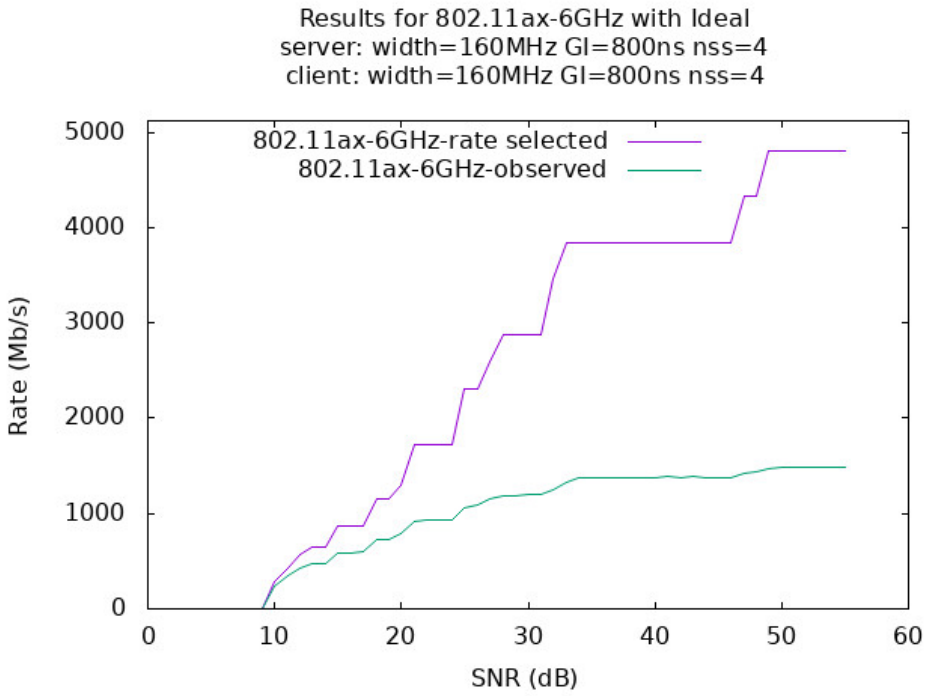


Figure 14: 6GHz results for maximum channel width

3.2 Effects of Throughput over Time

3.2.1 Test Method

The next iteration of the throughput related tests was that of monitoring throughput over time on a simulated Wi-Fi network. This part was developed in two scripts due to difficulties within the simulator. The first script written was *wifi_test.cc* which operated as the trivial example of this Wi-Fi test to get

plotting and data processing working. The second script that was developed under this set of test was $wifi - thoughput - test.cc$ which was a more detailed version of $wifi\_test$ that would set up a series of $UDPSocketClients$ on a CSMA network that was used to represent that of the Internet, and a series of $UDPSocketServer$ to represent clients streaming content. For the $wifi_test.cc$ script the Wi-fi parameters were selected to utilize the Wi-Fi standard 'WIFI_STANDARD_80211ax_5GHZ'. This simulation selected the default channel width of 80MHz for the Wi-Fi PHY layer. In this example the number of client spatial streams was reduced to 1 for a more realistic scenario, with 4 (the maximum for the simulation) Spatial streams used for the Access Point (AP), for simplicity this iteration of the simulation still utilized $ConstantSpeedPropagationModel$ for Wi-Fi propagation delay attributes, and $FixedRssLossModel$. In this initial example since the network was still a simple Node to Node network, the $PacketSocketHelper$ application was used for traffic generation. This used a simple scheme of generating slightly more data than the channel can handle to get an understanding of the throughput of the system. In this network the STA node was place 5m from the AP and it just used a $ConstantPositionMobilityModel$. This simulation utilized a similar callback structure to that of the $wifi - manager - example$ that had been previously explored. In this script the amount of data received was measure via a receiver trace callback that accumulated the number of bytes received per interval. The throughput was then calculated with scheduled events that tracked the amount of data received each interval. The data recorded at each interval was then added to a $GNUPlot$ container and output at the completion of the simulation.

Once the $wifi_test.cc$ simulation was working the next iteration involved creating a more realistic example with network congestion, based on a reasonable number of nodes. With this a network topology was created with a $PointToPoint$ network, representing the connection from the Wi-Fi Router to the Internet Service Provider (ISP), the ISP was represented by a CSMA network and the node representing the Wi-Fi router was configured as an Access Point (AP). From there a user selectable (or default value of 10) nodes were generated as Wi-Fi Station (STA) nodes, and CSMA nodes. the CSMA nodes were generated to represent a content server on the internet, and the STA nodes were to represent the clients streaming 25Mbps of data. The link between the ISP and AP was configured to 10Gbps using attributes, and each CSMA "Server" node was giving a link rate of 1Gbps to ensure that the devices outside teh local network had limited affect on the throughput and were not generating more data than their links could handle. As this network was larger and had a number of nodes. the $InternetStackHelper$ was used to install the stack on all of the nodes, and the $Ipv4InterfaceContainer$ was used to assign IP addresses to each of the nodes on the separate networks. Once the network was created and the IP addresses assigned, the $UdpServerHelper$, and $UdpClientHelper$ were used to generate point to point socket connections where data was streamed from the simulated content server to the device. The packet parameters were selected such that an expected data rate of 25Mbps per client server pair was used. To do this a loop was created that iterated through the Wi-Fi nodes, installed the 'Server' application, assigned a port number based on the base port of $4000 + (node - number)$ it then set up the client application and installed that on the corresponding CSMA node such that there were one server client pair for each of the STA nodes in the simulation. The Ipv4 Routing tables were then installed using the $Ipv4GlobalRoutingHelper$. Next for a 'fair' simulation, the mobility model was installed in each node. This was a simple $ConstantPositionMobilityModel$ where the locations of the STAs were set even around the AP at a fixed radius of 5m utilizing a loop. In this model all of the data tracking and plotting was kept the same as in the previous iteration of the test, as the goal was to see if having a number of nodes reduced the expected throughput per node, as this was the expected behaviour.

3.2.2 Test Results

The results of the first simple example worked as suspected, with a simple 2 node network. As seen in Figure 15 the rate at 5m distance which is well within Wi-Fi expected range, showed 160Mbps. When comparing this to the results of the previous section, it was found that with 4 Spatial Streams, and 160MHz channel width 1.3Gbps were seen, and therefore a throughput of 160Mbps, with only 1 Spatial Stream at the client, and the default channel width of 80MHz it is expected that the node would only get 1/8 the throughput achieved in the initial throughput simulations performed.

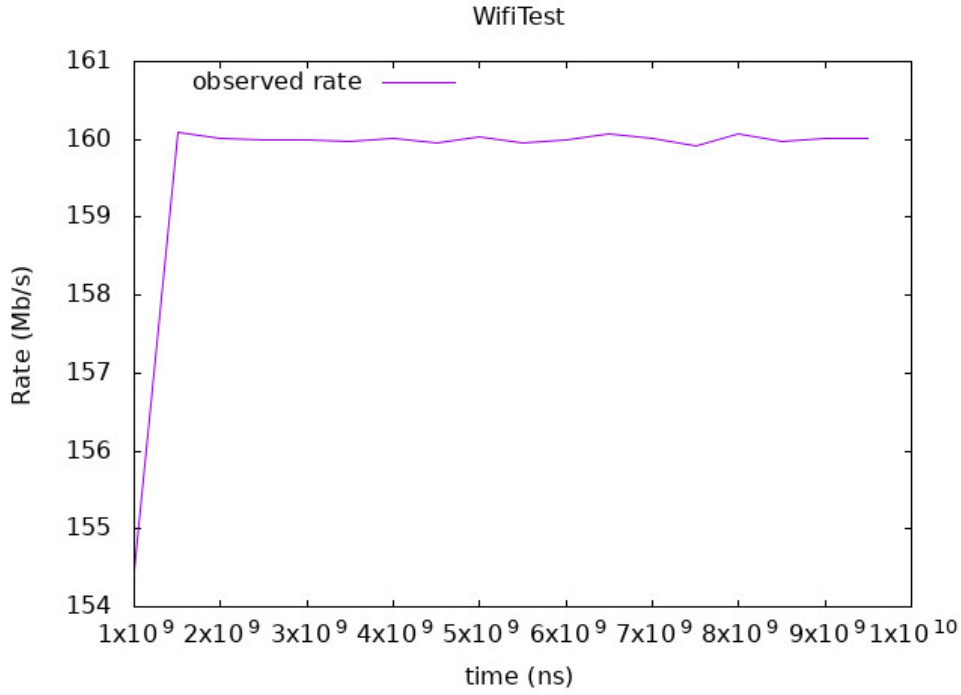


Figure 15: Wi-Fi 6 throughput vs time in a simple two node network

The results for the $wifi - throughput - test.cc$ simulation proved to be unsuccessful. In many scanarios using this type of network setup it appeared as soon as additional nodes were added, the network application would fail to operate, and there would be no packets received by the node callback, and no events seen in the Node receiver. This was tested further with different applications, including the $PacketSocketHelper$ though this was not expected to work on this network as this was a low layer protocol helper that did not work with larger network routing. This tested with a $UdpEchoServer/Client$ and was unsuccessful in this case as well. Unfortunately due to time constraints, the application addition to the network never provided any data though the network built and ran a simulation. At the conclusion of each simulation there were no packets seen, either in the callback, or in the pcap from the network. This simulation was developed to provide a comparison of result to that of the "Validation of the $ns-3$ 802.11ax OFDMA Implementation" by Magrin et al[15].

3.3 Home Network Wifi Simulation

3.3.1 Test Method

The final simulation developed was $wifi - sim - network.cc$. In this network, the network topology was similar to that of the $wifi - throughput - test.cc$, though the aim of this simulation was to provide a more realistic example of a home network to assess. This network utilized much of the low level PHY setup that had been used in the previous models, but with some minor differences for a more realistic simulation. One of the first differences was the selection of a $RangePropagationLossModel$ to represent a more realistic propagation loss in the network. The network topology used was similar to that of the previous network with CSMA nodes acting as a server somewhere on the internet, and STA nodes being used as hosts on the home network connected to the servers. With this many of the low level parameters had stayed the same as the previous example to allow for consistency between simulations. The next change that was made to increase the realism of the scenario is the nodes were allocated positions in a grid, but all of the STA nodes were given a $RandomWalkMobilityModel$ and the were bound by a $Rectangle$ object that was $10mX20m$ to represent that of a household. In this example, $UdpEchoClient/Server$ was employed to make the traffic bi directional, though the amount of traffic generated was selected such that it could provide a realistic representation of achievable throughput values for a home network. Due to the fact that there was some issue with the previous script, there was some simplification done to the data logging and the prots were generated using pcap files and Wireshark for simplicity and time related to testing and evaluation of the system.

3.3.2 Test Results

In this example the simulation was able to run successfully and the throughout could be seen for each node. This was considerably better that the similar topology that would not run the traffic application. With this though there were issues with the Wi-Fi manager, and even though the parameters were set for that of 802.11ax, the resulting simulation indicated that the packets were that of 802.11n, and the throughput reflected that of an 802.11n network. In Figures 16-25, it can be seen that the data rate becomes less constant over time as the network deals with various different traffic. From this the throughput seems constant enough on the network, though with it's current application the 802.11n network would net meet the criteria required for high bandwidth content applications on the network. Based on the difficulties encountered, the newly implemented 802.11ax standard in $ns-3$ has proven to be more difficult than expected when working with applications. Through considerable debug it could not be determined why the settings reverted back to the 802.11n standard, as the setting were selected and developed in earlier iteration of the series of tests performed, though on iteration the application failed to produce traffic, and in this example the traffic produced utilized PHY setting different to those set and functional in previous examples. This simulation demonstrates a simple example of the home network, but fails to assess the desired functionality due to some difficulties with the $ns-3$ simulator software.

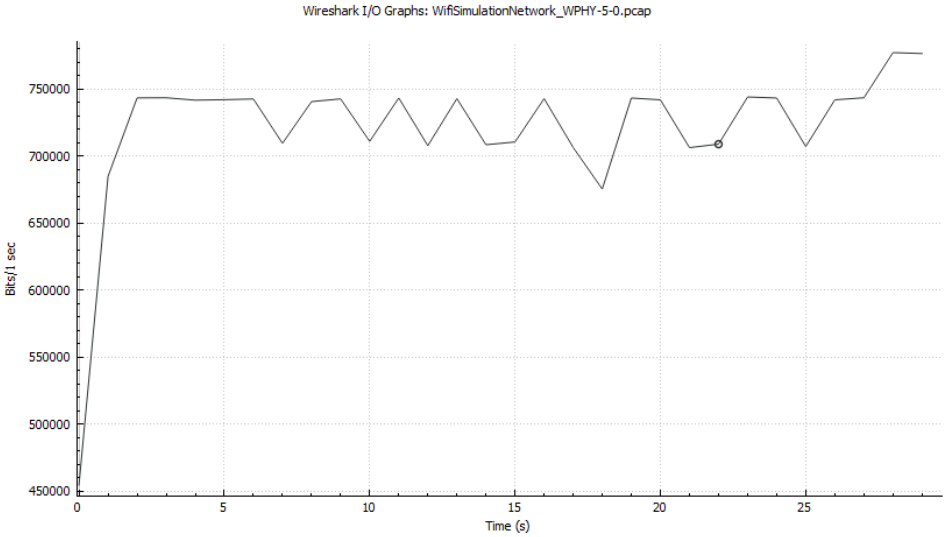


Figure 16: Throughput at station node 5 on home network

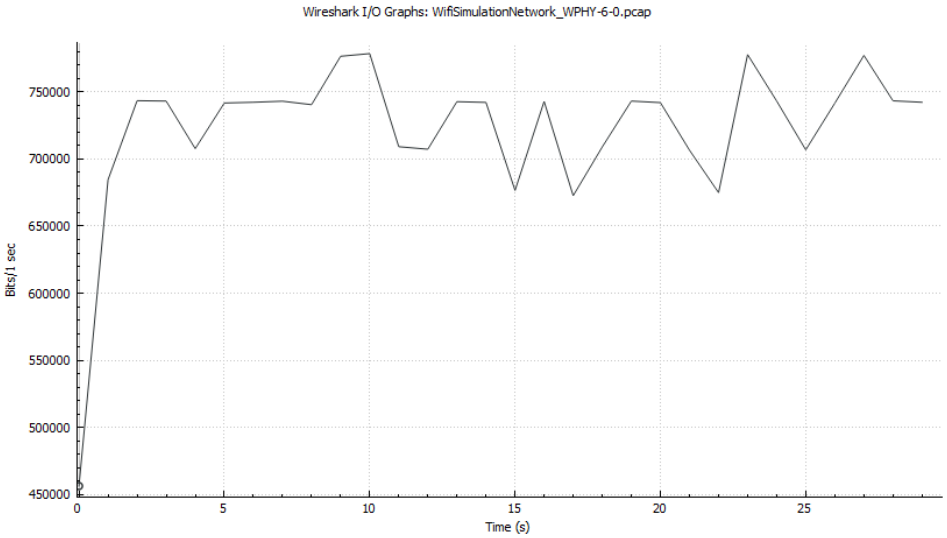


Figure 17: Throughput at station node 6 on home network

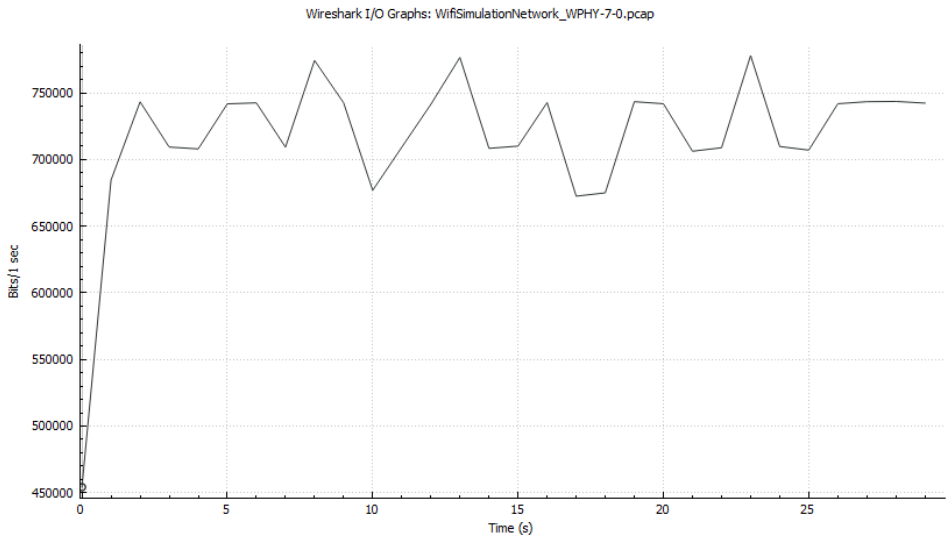


Figure 18: Throughput at station node 7 on home network

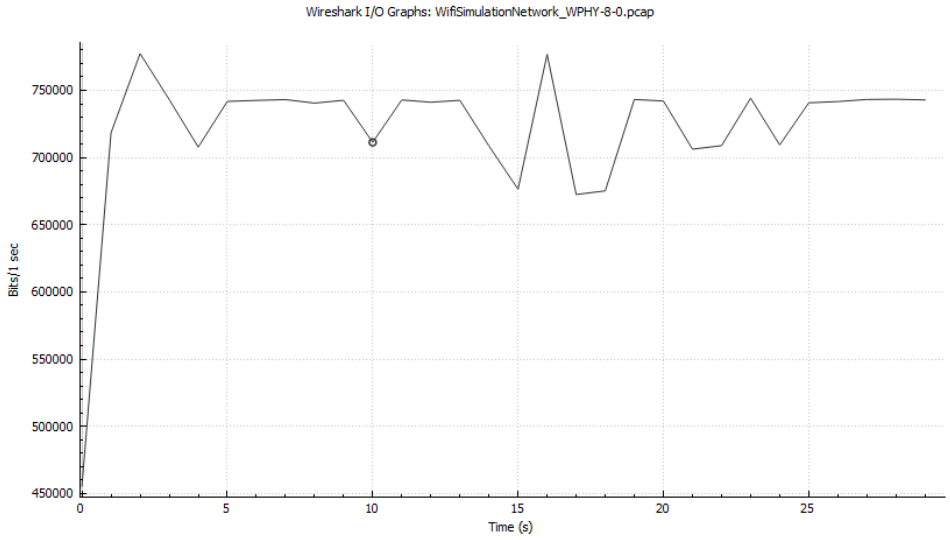


Figure 19: Throughput at station node 8 on home network

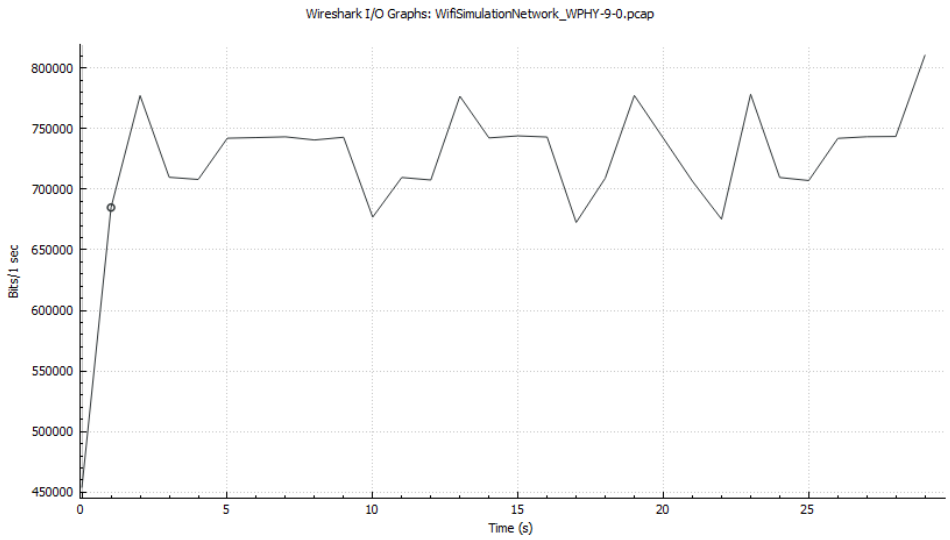


Figure 20: Throughput at station node 9 on home network

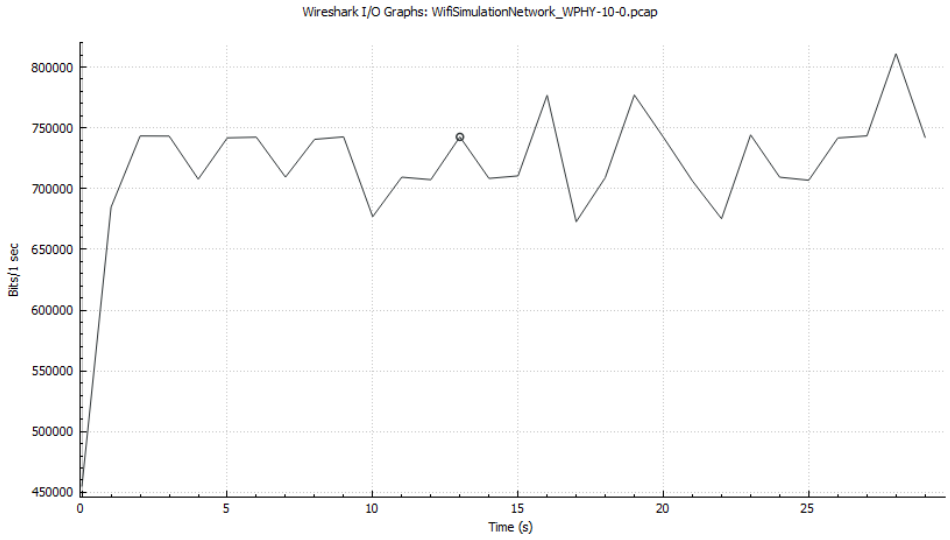


Figure 21: Throughput at station node 10 on home network

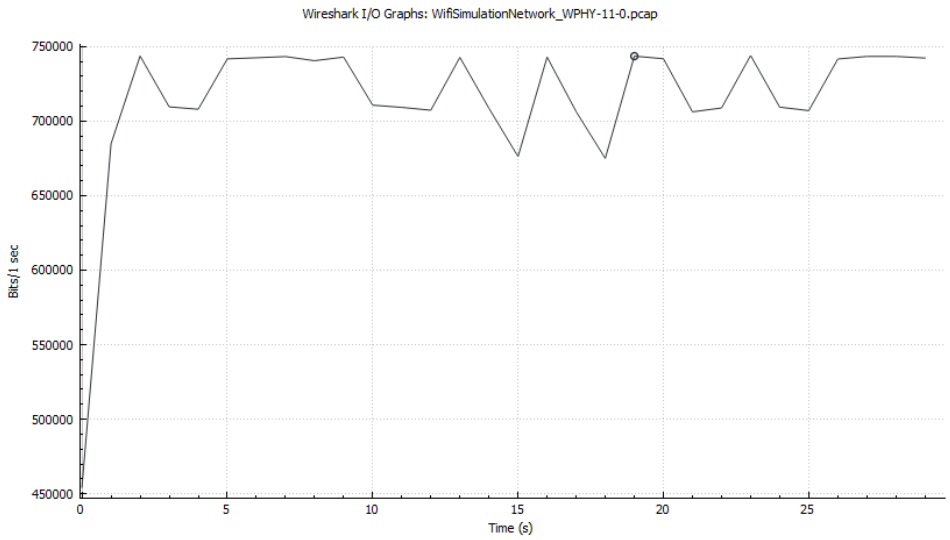


Figure 22: Throughput at station node 11 on home network

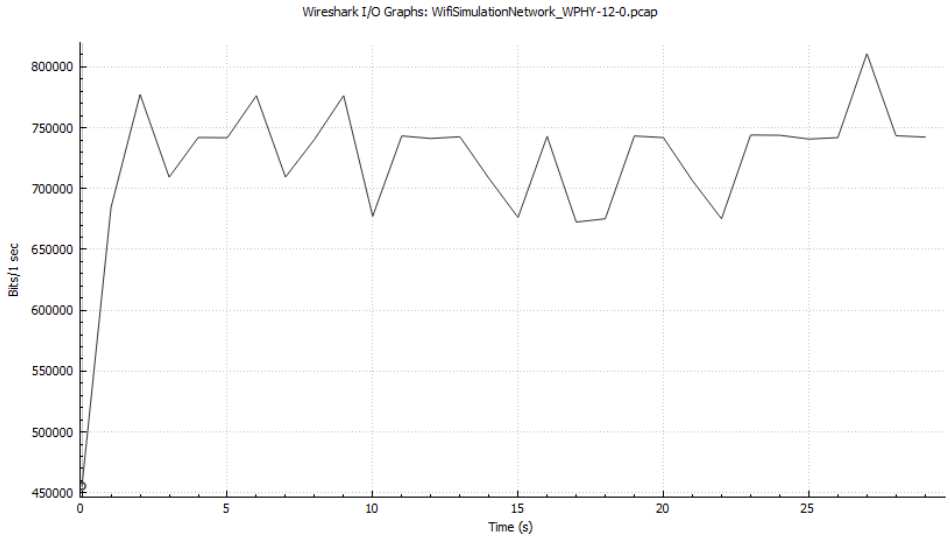


Figure 23: Throughput at station node 12 on home network

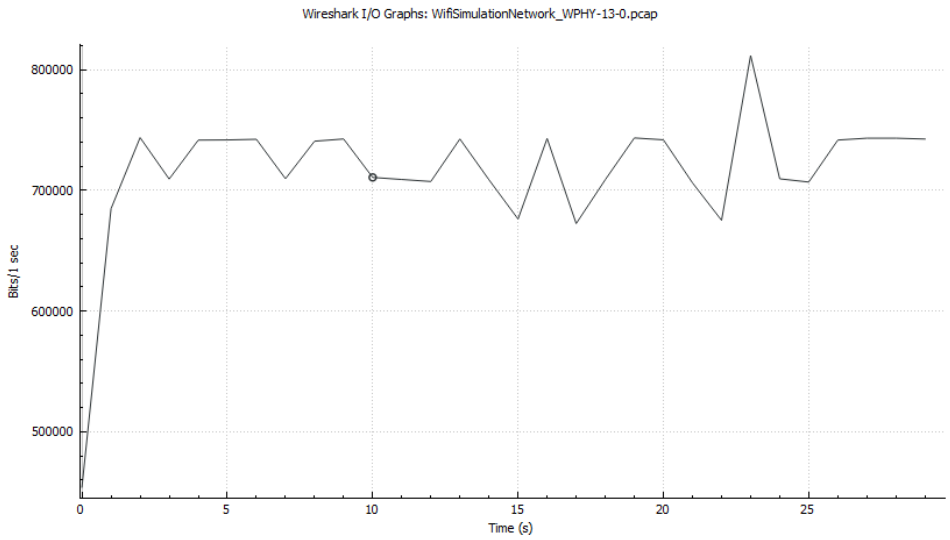


Figure 24: Throughput at station node 13 on home network

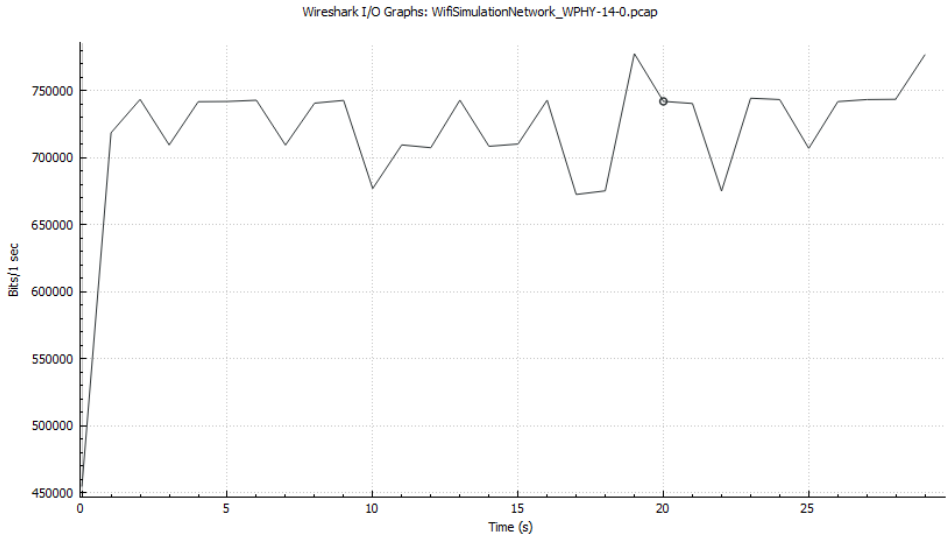


Figure 25: Throughput at station node 14 on home network

4 Discussion and Conclusion

4.1 Difficulties

In undertaking this project, a number of the low level parameters and details about the link and physical layer of the network were successfully simulated and assessed. This allowed for a reasonable intuition of the performance regarding Wi-Fi 6 improvements, as well as created a good platform to really study the details of link layer attributes, as well as the physical layer communications details of the Wi-Fi protocol, including knowledge of modulation patters, data rates, signaling schemes, multiplexing, as well as exposure to even more in depth details such as BSS coloring, MU-MIMO and how it utilizes spatial streams. But since this protocol is very new and has just been implemented currently by researchers in the field, the amount of support developed into $ns-3$ is limited. Due to these issues, the final network simulations, though successfully operational using some Wi-Fi 6 devices and a simulated network, we were unable to get all of the components and classes working together properly, with the networks not behaving as expected once multiple nodes had be implemented. In some of the sources it has been shown that this is possible but the debug time required and the time frame of the project did not allow for completion of the final test scenario, just leaving a realistic Wi-Fi scenario but, unable to test or tune to

the desired parameters for the scenario. With these simulations, there could be some future debug, and investigation as to the implementation, though it is suspect that some models that were used for the initial simulations do not support multiple 802.11ax nodes. Something that could be used for further debugging is to investigate the $SpectrumWifiPhyHelper$, as this is said to support 802.11ax whereas the $YansWifiPhyHelper$ may be lacking support for 802.11ax in the simulation developed. Among this, there are so many modules and classes available to $ns-3$ and with a newly adopted protocol, the implementation of components is done by researchers as required. This leads to the requirement of a deeper understanding of the inner workings of $ns-3$ to undertake such research on new technology.

4.2 Future work

The next logical step for this project is to simulate the same conditions using Latency Measurements to verify the improvement in Wi-Fi 6 for low latency applications. In addition, because 802.11ax has features to improve overlapping BSS (OBSS) operation in dense environments, including changes to deferral rules and CCA levels. It is possible to build a deployment simulation scenario with a number of (overlapping) BSS, each of which can be configured as 802.11ac or 802.11ax BSS. In such a scenario, the hypothesis that 802.11ax can lead to greater system performance can be tested[16].

Another Issue to consider is Investigating Spatial Stream Reuse for Multi-User operation in Wi-Fi 6. This would use similar scenario as second test case but focus on the Multi-User MIMO technology. The implementation of the first test case should be revised to include BSS Coloring capability for reduced interference between APs.

4.3 Conclusion

In this project, our goal is to understand and test the main improvements of Wi-Fi 6. We use the $ns-3$ network simulator to implement the network. We implemented $ns-3$ library and built-in functions to build network topology. Our simulation environment contains randomly moving APs, and many parameters are set to the maximum to test the maximum throughput and other data of the network. In order to implement a Wi-Fi 6 network test case in $ns-3$, we have faced many challenges. Firstly, it was very time consuming to implement our network topology. Secondly, because the example of $ns-3$ is based on C++, there is a lack of comment in the example, and there is no relevant tutorial on the network, it is not easy to read and understand the example of Wi-Fi 6 test of $ns-3$. At the same time, we often have to write testing case from scratch, and debugging and parameter adjustment becomes difficult. Since there are a few methods to extract traffic, and the methods used in each example are different, It is hard for us to integrate different sub-routines, resulting in incompatibility and inability to run. At the same time, the complexity of the code increases in the process of integration. Last but not least, The simulation setting takes a long time to run and we have a huge index that we would like to compare with, the trade off is that we limited our simulation time for only 100 seconds between each simulation run. When we run our first implementation, the result indicate that new technologies introduced in Wi-Fi 6 like OFDMA and 1024-QAM MCS settings can significantly increase the area throughput when dealing with more and more access points. In conclusion, although the task is hard and arduous, it can be considered that we have successfully achieved the predetermined goal.

References

[1] Christiansen, Peter and Parrish, Kevin, "how much speed do i need for online gaming." https://www.highspeedinternet.com/resources/how-much-speed-do-i-need-for-online-gaming. Accessed: 2022-04-07.

[2] zen.co, "the minimum internet speeds for streaming video in-2019." https://www.zen.co.uk/blog/posts/zen-blog/2019/04/18/the-minimum-internet-speeds-for-streaming-video-in-2019. Accessed: 2022-04-07.

[3] optimum, "What is latency." https://www.optimum.com/articles/internet/what-latency. Accessed: 2022-04-07.

[4] Lawrence, Nathan, "what is wifi 6." https://www.reviews.org/au/technology/what-is-wifi-6/. Accessed: 2022-04-07.

[5] E. Khorov, A. Kiryanov, A. Lyakhov, and G. Bianchi, "A tutorial on ieee 802.11ax high efficiency wlans," *IEEE Communications Surveys Tutorials*, vol. 21, pp. 197–216, Firstquarter 2019.

[6] Wi-Fi Alliance, "Wi-fi certified 6." https://www.wi-fi.org/discover-wi-fi/wi-fi-certified-6. Accessed: 2022-03-18.

[7] Irei, Alissa, "Wi-fi 6 explained: Speed, range, latency, frequency and security." https://www.techtarget.com/searchnetworking/feature/Wi-Fi-6-technology-explained-from-speed-to-security-and-more. Accessed: 2022-04-07.

[8] Raúl Katz, Juan Jung and Fernando Callorda, "The economic value of wi-fi®:a global view (2021 – 2025)." https://www.wi-fi.org/download.php?file=/sites/default/files/private/The_Economic_Value_of_Wi-Fi-A_Global_View_2021-2025_202109.pdf. Accessed: 2022-04-07.

[9] nsnam.org, "Wifi design." https://www.nsnam.org/docs/models/html/wifi-design.html#scope-and-limitations. Accessed: 2022-04-06.

[10] semfionetworks, "Mcs table (ht/vht/he)." https://mcsindex.net/?_ga=2.257012937.331301.1649982105-1945813884.1649704154. Accessed: 2022-03-18.

[11] Panagiotis Vouzis, "What is the mcs index?." https://netbeez.net/blog/what-is-mcs-index/. Accessed: 2022-04-11.

[12] D. Magrin, S. Avallone, S. Roy, and M. Zorzi, "Validation of the ns-3 802.11ax ofdma implementation," in *Proceedings of the Workshop on Ns-3*, pp. 1–8, Association for Computing Machinery, 2021.

[13] J. Sandoval and S. Cespedes, "Performance evaluation of ieee 802.11ax for residential networks," in *2021 IEEE Latin-American Conference on Communications (LATINCOM)*, pp. 1–7, Nov 2021.

[14] ARISTA, "Multi-user mimo in wifi6." https://www.arista.com/assets/data/pdf/Whitepapers/MU-MIMO-Whitepaper.pdf. Accessed: 2022-04-10.

[15] M. D., A. S., R. S., and Z. M., "Validation of the ns-3 802.11ax ofdma implementation," *Proceedings of the Workshop on Ns-3*, 2021.

[16] S. Deronne, S. C. T. Henderson, S. R. L. Lanante, M. S. M. Mehrnoush, and P. Monajemi, "ns-3 wi-fi 11ax project release final." https://depts.washington.edu/funlab/wp-content/uploads/2018/11/11ax-final-report.pdf, 2019.

A Code Listings

A.1 wifi-he-network-MCS.cc

```
/* -*-  Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2016 SEBASTIEN DERONNE
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Sebastien Deronne <sebastien.deronne@gmail.com>
 */

#include "ns3/command-line.h"
#include "ns3/config-store-module.h"
#include "ns3/config.h"
#include "ns3/uinteger.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include "ns3/string.h"
#include "ns3/enum.h"
#include "ns3/log.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/spectrum-wifi-helper.h"
#include "ns3/ssid.h"
#include "ns3/mobility-helper.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/udp-client-server-helper.h"
#include "ns3/packet-sink-helper.h"
#include "ns3/on-off-helper.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/packet-sink.h"
#include "ns3/yans-wifi-channel.h"
#include "ns3/multi-model-spectrum-channel.h"
#include "ns3/wifi-acknowledgment.h"
#include "ns3/rng-seed-manager.h"
#include "ns3/he-ru.h"
#include "ns3/flow-monitor.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/trace-helper.h"

#include <iostream>
#include <fstream>

// This is a simple example in order to show how to configure an IEEE 802.11ax Wi-Fi
    network.
//
// It outputs the UDP or TCP goodput for every HE MCS value, which depends on the MCS
    value (0 to 11),
// the channel width (20, 40, 80 or 160 MHz) and the guard interval (800ns, 1600ns or
    3200ns).
// The PHY bitrate is constant over all the simulation run. The user can also specify
    the distance between
// the access point and the station: the larger the distance the smaller the goodput.
//
// The simulation assumes a configurable number of stations in an infrastructure network
    :
//
//  STA     AP
//    *     *
//    |     |
```

```
//   n1     n2
//
// Packets in this simulation belong to BestEffort Access Class (AC_BE).
// By selecting an acknowledgment sequence for DL MU PPDUs, it is possible to aggregate
    a
// Round Robin scheduler to the AP, so that DL MU PPDUs are sent by the AP via DL OFDMA.

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("he-wifi-network");

int main (int argc, char *argv[])
{
  bool udp {false};
  bool useRts {false};
  bool useExtendedBlockAck {false};
  bool actprobing {false}; // Active Probing, normally set to false
  bool qos {false}; //Quality of Service Support
  bool bsrp {false}; //buffer status report poll
  bool mimo {false}; //MIMO enable

  double simulationTime {10}; //seconds
  double distance {1.0}; //meters
  double frequency {5}; //whether 2.4, 5 or 6 GHz
  double minExpectedThroughput {0};
  double maxExpectedThroughput {0};

  int mcs {5}; // -1 indicates an unset value
  uint32_t payloadSize = 540; // must fit in the max TX duration when transmitting at
    MCS 0 over an RU of 26 tones

  std::size_t nStations {10};
  std::string dlAckSeqType {"NO-OFDMA"};
  std::string phyModel {"Spectrum"};


  CommandLine cmd (__FILE__);

  cmd.AddValue ("udp", "UDP if set to 1, TCP otherwise", udp);
  cmd.AddValue ("useRts", "Enable/disable RTS/CTS", useRts);
  cmd.AddValue ("useExtendedBlockAck", "Enable/disable use of extended BACK",
    useExtendedBlockAck);
  cmd.AddValue ("actprobing", "UDP if set to 1, TCP otherwise", actprobing);
  cmd.AddValue ("qos", "Enable/disable Quality of Servic Support", qos);
  cmd.AddValue ("bsrp", "Enable/disable Buffer Status Report Poll", bsrp);
  cmd.AddValue ("mimo", "Enable/disable MIMO", mimo);

  cmd.AddValue ("simulationTime", "Simulation time in seconds", simulationTime);
  cmd.AddValue ("distance", "Distance in meters between the station and the access point
    ", distance);
  cmd.AddValue ("frequency", "Whether working in the 2.4, 5 or 6 GHz band (other values
    gets rejected)", frequency);
  cmd.AddValue ("minExpectedThroughput", "if set, simulation fails if the lowest
    throughput is below this value", minExpectedThroughput);
  cmd.AddValue ("maxExpectedThroughput", "if set, simulation fails if the highest
    throughput is above this value", maxExpectedThroughput);

  cmd.AddValue ("mcs", "if set, limit testing to a specific MCS (0-11)", mcs);
  cmd.AddValue ("payloadSize", "The application payload size in bytes", payloadSize);

  cmd.AddValue ("nStations", "Number of non-AP HE stations", nStations);
  cmd.AddValue ("dlAckType", "Ack sequence type for DL OFDMA (NO-OFDMA, ACK-SU-FORMAT,
    MU-BAR, AGGR-MU-BAR)", dlAckSeqType);
  cmd.AddValue ("phyModel", "PHY model to use when OFDMA is disabled (Yans or Spectrum).
     If OFDMA is enabled then Spectrum is automatically selected", phyModel);

  cmd.Parse (argc,argv);

  std::cout << "UDP" << "\t" << "RTS" << "\t" << "EBA" << "\t" << "MCS" << "\t" << "ACK"
     << "\t\t" << "phyModel" << '\n';
  std::cout << udp << "\t" << useRts << "\t" << useExtendedBlockAck << "\t" << mcs << "\
    t" << dlAckSeqType << "\t" << phyModel << std::endl;
```

```
std::cout << "Active Probing" << "\t" << "QosSupported" << "\t" << "BSRP" << "\t" << "
  MIMO" << '\n';
std::cout << actprobing << "\t\t" << qos << "\t\t" << bsrp << "\t" << mimo << std::
  endl;

std::cout << "Freq" << "\t" << "Dist" << "\t" << "SimTime" << "\t" << "nStations" << "
  \t" << "payloadsize" <<'\n';
std::cout << frequency << "\t" << distance << "\t" << simulationTime << "\t" <<
  nStations << "\t\t" << payloadSize <<std::endl;

std::cout << "If set, simulation fails if throughput is outside the range" <<'\n';
std::cout << "minExpectedThroughput" << "\t---\t" << "maxExpectedThroughput" <<'\n';
std::cout << minExpectedThroughput << "\t\t\t---\t" << maxExpectedThroughput <<std::
  endl;


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

if (dlAckSeqType == "ACK-SU-FORMAT")
  {
    Config::SetDefault ("ns3::WifiDefaultAckManager::DlMuAckSequenceType",
                        EnumValue (WifiAcknowledgment::DL_MU_BAR_BA_SEQUENCE));
  }
else if (dlAckSeqType == "MU-BAR")
  {
    Config::SetDefault ("ns3::WifiDefaultAckManager::DlMuAckSequenceType",
                        EnumValue (WifiAcknowledgment::DL_MU_TF_MU_BAR));
  }
else if (dlAckSeqType == "AGGR-MU-BAR")
  {
    Config::SetDefault ("ns3::WifiDefaultAckManager::DlMuAckSequenceType",
                        EnumValue (WifiAcknowledgment::DL_MU_AGGREGATE_TF));
  }
else if (dlAckSeqType != "NO-OFDMA")
  {
    NS_ABORT_MSG ("Invalid DL ack sequence type (must be NO-OFDMA, ACK-SU-FORMAT, MU-
  BAR or AGGR-MU-BAR)");
  }

if (phyModel != "Yans" && phyModel != "Spectrum")
  {
    NS_ABORT_MSG ("Invalid PHY model (must be Yans or Spectrum)");
  }
if (dlAckSeqType != "NO-OFDMA")
  {
    // SpectrumWifiPhy is required for OFDMA
    phyModel = "Spectrum";
  }

double prevThroughput [12];
for (uint32_t l = 0; l < 12; l++)
  {
    prevThroughput[l] = 0;
  }

std::cout << "MCS value" << "\t\t" << "Channel width" << "\t\t" << "GI" << "\t\t\t" <<
   "Throughput" << '\n';

int minMcs = 0;
int maxMcs = 11;
if (mcs >= 0 && mcs <= 11)
  {
    minMcs = mcs;
    maxMcs = mcs;
  }
for (int mcs = minMcs; mcs <= maxMcs; mcs++)
  {
    uint8_t index = 0;
    double previous = 0;
```

```
  uint8_t maxChannelWidth = frequency == 2.4 ? 40 : 160;
  for (int channelWidth = 20; channelWidth <= maxChannelWidth; ) //MHz
    {
      for (int gi = 3200; gi >= 800; ) //Nanoseconds
        {
          if (!udp)
            {
      Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue (payloadSize)
);
            }

          NodeContainer wifiStaNodes;
          wifiStaNodes.Create (nStations); // Create nStations=10 station node
objects
          NodeContainer wifiApNode;
          wifiApNode.Create (1); // Create 1 access point node object

          // Create a WifiMacHelper, which is reused across STA and AP
configurations
    WifiMacHelper mac;

    // Create a WifiHelper, which will use the above helpers to create
    // and install Wifi devices.  Configure a Wifi standard to use, which
    // will align various parameters in the Phy and Mac to standard defaults.
          WifiHelper wifi;

    // Declare NetDeviceContainers to hold the container returned by the helper
    NetDeviceContainer staDevices;
    NetDeviceContainer apDevice;

          if (frequency == 6)
            {
              wifi.SetStandard (WIFI_STANDARD_80211ax_6GHZ);
              Config::SetDefault ("ns3::LogDistancePropagationLossModel::
ReferenceLoss", DoubleValue (48));
            }
          else if (frequency == 5)
            {
              wifi.SetStandard (WIFI_STANDARD_80211ax_5GHZ);
            }
          else if (frequency == 2.4)
            {
              wifi.SetStandard (WIFI_STANDARD_80211ax_2_4GHZ);
              Config::SetDefault ("ns3::LogDistancePropagationLossModel::
ReferenceLoss", DoubleValue (40));
            }
          else
            {
              std::cout << "Wrong frequency value!" << std::endl;
              return 0;
            }

          std::ostringstream oss;
          oss << "HeMcs" << mcs;
          wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                "DataMode", StringValue (oss.str ()),
                                        "ControlMode", StringValue (oss.str ()));
          Ssid ssid = Ssid ("ns3-80211ax");

    // Create a channel helper and phy helper, and then create the channel
    // If OFDMA is enabled, then Spectrum WiFi models need  tobe used
          if (phyModel == "Spectrum")
            {
              /*
              * SingleModelSpectrumChannel cannot be used with 802.11ax because two
              * spectrum models are required: one with 78.125 kHz bands for HE PPDUs
              * and one with 312.5 kHz bands for, e.g., non-HT PPDUs (for more
details,
              * see issue #408 (CLOSED))
              */
              Ptr<MultiModelSpectrumChannel> spectrumChannel = CreateObject<
MultiModelSpectrumChannel> ();
              SpectrumWifiPhyHelper phy;
```

```
            phy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
            phy.SetChannel (spectrumChannel);

            if (dlAckSeqType != "NO-OFDMA")
              {
                mac.SetMultiUserScheduler ("ns3::RrMultiUserScheduler",
                                           "EnableUlOfdma", BooleanValue (false),
                                           "EnableBsrp", BooleanValue (bsrp));
              }

          if (mimo)
            {
            phy.Set ("Antennas", UintegerValue (3));
            phy.Set ("MaxSupportedTxSpatialStreams", UintegerValue (2));
            phy.Set ("MaxSupportedRxSpatialStreams", UintegerValue (2));
          }

          if (qos)
                    {
            mac.SetType ("ns3::StaWifiMac",
                   "Ssid", SsidValue (ssid),
                   "QosSupported", BooleanValue (qos),
                                "BE_BlockAckThreshold", UintegerValue (2),
                                "ActiveProbing", BooleanValue (actprobing));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            staDevices = wifi.Install (phy, mac, wifiStaNodes);

            mac.SetType ("ns3::ApWifiMac",
                   "Ssid", SsidValue (ssid),
                   "BeaconGeneration", BooleanValue (true),
                   "BeaconInterval", TimeValue (Seconds (2.5)),
                   "EnableBeaconJitter", BooleanValue (false),
                   "QosSupported", BooleanValue (qos));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            apDevice = wifi.Install (phy, mac, wifiApNode);
          }
          else
          {
            mac.SetType ("ns3::StaWifiMac",
                   "Ssid", SsidValue (ssid),
                   "ActiveProbing", BooleanValue (actprobing));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            staDevices = wifi.Install (phy, mac, wifiStaNodes);

            mac.SetType ("ns3::ApWifiMac",
                                 "EnableBeaconJitter", BooleanValue (false),
                                 "Ssid", SsidValue (ssid));
            phy.Set ("ChannelWidth", UintegerValue (channelWidth));
            apDevice = wifi.Install (phy, mac, wifiApNode);
          }
                }
              else
                {
                  YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
                  YansWifiPhyHelper phy;
          phy.SetChannel (channel.Create ());
                  phy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);

                  phy.Set ("ChannelWidth", UintegerValue (channelWidth));
          if (mimo)
            {
            phy.Set ("Antennas", UintegerValue (3));
            phy.Set ("MaxSupportedTxSpatialStreams", UintegerValue (2));
            phy.Set ("MaxSupportedRxSpatialStreams", UintegerValue (1));
          }
          mac.SetType ("ns3::StaWifiMac",
                 "Ssid", SsidValue (ssid),
                 "ActiveProbing", BooleanValue (false));
                  staDevices = wifi.Install (phy, mac, wifiStaNodes);

          phy.Set ("ChannelWidth", UintegerValue (channelWidth));
                  mac.SetType ("ns3::ApWifiMac",
                               "Ssid", SsidValue (ssid),
```

```
                "EnableBeaconJitter", BooleanValue (false));
                apDevice = wifi.Install (phy, mac, wifiApNode);
              }

            RngSeedManager::SetSeed (1);
            RngSeedManager::SetRun (1);
            int64_t streamNumber = 100;
            streamNumber += wifi.AssignStreams (apDevice, streamNumber);
            streamNumber += wifi.AssignStreams (staDevices, streamNumber);

            // Set guard interval and MPDU buffer size
            Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/HeConfiguration
/GuardInterval", TimeValue (NanoSeconds (gi)));
            Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/HeConfiguration
/MpduBufferSize", UintegerValue (useExtendedBlockAck ? 256 : 64));

            // mobility.
            MobilityHelper mobility;
            Ptr<ListPositionAllocator> positionAlloc = CreateObject<
ListPositionAllocator> ();

            positionAlloc->Add (Vector (0.0, 0.0, 0.0));
            positionAlloc->Add (Vector (distance, 0.0, 0.0));
            mobility.SetPositionAllocator (positionAlloc);

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

            mobility.Install (wifiApNode);
            mobility.Install (wifiStaNodes);

            /* Internet stack*/
            InternetStackHelper stack;
            stack.Install (wifiApNode);
            stack.Install (wifiStaNodes);

            Ipv4AddressHelper address;
            address.SetBase ("192.168.1.0", "255.255.255.0");
            Ipv4InterfaceContainer staNodeInterfaces;
            Ipv4InterfaceContainer apNodeInterface;

            staNodeInterfaces = address.Assign (staDevices);
            apNodeInterface = address.Assign (apDevice);

            /* Setting applications */
            ApplicationContainer serverApp;
            if (udp)
              {
                //UDP flow
                uint16_t port = 9;
                UdpServerHelper server (port);
                serverApp = server.Install (wifiStaNodes);
                serverApp.Start (Seconds (0.0));
                serverApp.Stop (Seconds (simulationTime + 1));

                for (std::size_t i = 0; i < nStations; i++)
                  {
                    UdpClientHelper client (staNodeInterfaces.GetAddress (i), port);
                    client.SetAttribute ("MaxPackets", UintegerValue (4294967295u));
                    client.SetAttribute ("Interval", TimeValue (Time ("0.00001"))); //
packets/s
                    client.SetAttribute ("PacketSize", UintegerValue (payloadSize));
                    ApplicationContainer clientApp = client.Install (wifiApNode.Get
(0));
                    clientApp.Start (Seconds (1));
                    clientApp.Stop (Seconds (simulationTime + 1));
                  }

              }
            else
              {
                //TCP flow
                uint16_t port = 50000;
                Address localAddress (InetSocketAddress (Ipv4Address::GetAny (), port)
```

```
);
                PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
localAddress);
                serverApp = packetSinkHelper.Install (wifiStaNodes);
                serverApp.Start (Seconds (0.0));
                serverApp.Stop (Seconds (simulationTime + 1));

                for (std::size_t i = 0; i < nStations; i++)
                  {
                    OnOffHelper onoff ("ns3::TcpSocketFactory", Ipv4Address::GetAny ()
);
                    onoff.SetAttribute ("OnTime",  StringValue ("ns3::
ConstantRandomVariable[Constant=1]"));
                    onoff.SetAttribute ("OffTime", StringValue ("ns3::
ConstantRandomVariable[Constant=0]"));
                    onoff.SetAttribute ("PacketSize", UintegerValue (payloadSize));
                    onoff.SetAttribute ("DataRate", DataRateValue (1000000000)); //bit
/s
                    AddressValue remoteAddress (InetSocketAddress (staNodeInterfaces.
GetAddress (i), port));
                    onoff.SetAttribute ("Remote", remoteAddress);
                    ApplicationContainer clientApp = onoff.Install (wifiApNode.Get (0)
);
                    clientApp.Start (Seconds (1.0));
                    clientApp.Stop (Seconds (simulationTime + 1));
                  }
              }

          Simulator::Schedule (Seconds (0), &Ipv4GlobalRoutingHelper::
PopulateRoutingTables);

    Ptr<FlowMonitor> flowMonitor;
          FlowMonitorHelper flowHelper;
          flowMonitor = flowHelper.InstallAll();
          // phy.EnablePcapAll ("wifi-he-ofdma");

          Simulator::Stop (Seconds (simulationTime + 1));

    // Output config store to txt format
    Config::SetDefault ("ns3::ConfigStore::Filename", StringValue ("output-
attributes1.txt"));
    Config::SetDefault ("ns3::ConfigStore::FileFormat", StringValue ("RawText"));
    Config::SetDefault ("ns3::ConfigStore::Mode", StringValue ("Save"));
    ConfigStore outputConfig;
    outputConfig.ConfigureDefaults ();
    outputConfig.ConfigureAttributes ();

          Simulator::Run ();

          // When multiple stations are used, there are chances that association
requests collide
          // and hence the throughput may be lower than expected. Therefore, we
relax the check
          // that the throughput cannot decrease by introducing a scaling factor (or
 tolerance)
          double tolerance = 0.10;
          uint64_t rxBytes = 0;
          if (udp)
            {
              for (uint32_t i = 0; i < serverApp.GetN (); i++)
                {
                  rxBytes += payloadSize * DynamicCast<UdpServer> (serverApp.Get (i)
)->GetReceived ();
                }
            }
          else
            {
              for (uint32_t i = 0; i < serverApp.GetN (); i++)
                {
                  rxBytes += DynamicCast<PacketSink> (serverApp.Get (i))->GetTotalRx
 ();
                }
            }
```

```
              double throughput = (rxBytes * 8) / (simulationTime * 1000000.0); //Mbit/s

              Simulator::Destroy ();

              std::cout << mcs << "\t\t\t" << channelWidth << " MHz\t\t\t" << gi << " ns
    \t\t\t" << throughput << " Mbit/s" << std::endl;

              //test first element
              if (mcs == 0 && channelWidth == 20 && gi == 3200)
                {
                  if (throughput * (1 + tolerance) < minExpectedThroughput)
                    {
                      NS_LOG_ERROR ("Obtained throughput " << throughput << " is not
    expected!");
                      exit (1);
                    }
                }
              //test last element
              if (mcs == 11 && channelWidth == 160 && gi == 800)
                {
                  if (maxExpectedThroughput > 0 && throughput > maxExpectedThroughput *
    (1 + tolerance))
                    {
                      NS_LOG_ERROR ("Obtained throughput " << throughput << " is not
    expected!");
                      exit (1);
                    }
                }
              //test previous throughput is smaller (for the same mcs)
              if (throughput * (1 + tolerance) > previous)
                {
                  previous = throughput;
                }
              else
                {
                  NS_LOG_ERROR ("Obtained throughput " << throughput << " is not
    expected!");
                  exit (1);
                }
              //test previous throughput is smaller (for the same channel width and GI)
              if (throughput * (1 + tolerance) > prevThroughput [index])
                {
                  prevThroughput [index] = throughput;
                }
              else
                {
                  NS_LOG_ERROR ("Obtained throughput " << throughput << " is not
    expected!");
                  exit (1);
                }
              index++;
              gi /= 2;
            }
          channelWidth *= 2;
        }
    }


  return 0;
}
```

A.2 ofdma-validation

A.2.1 ofdma-validation/ofdma-validation.py

```
# Standard imports
import copy
import seaborn as sns
import numpy as np
import pandas as pd
import xarray as xr
import sem
```

```
from itertools import product
from multiprocessing import Pool

# Load the code to manage results from simulations and the analytical model
from simulation import *
from model import *
from parsingfunctions import *

# Matplotlib tricks to get good quality images
import matplotlib
matplotlib.rc('lines', **{'linewidth': 2, 'mew': 2})
matplotlib.rc('font', **{'family': 'serif', 'serif': ['Computer Modern'], 'size': 13})
matplotlib.rc('text', usetex=True)
import matplotlib.pyplot as plt
from cycler import cycler

NSpath = '/home/yanyan_zhang/workspace/bake/source/ofdma'
PScript = 'wifi-ofdma-validation'
outdir = '/home/yanyan_zhang/workspace/bake/source/ofdma/output/ofdma-validation-results
    '
# results_folder = 'output/validation-results'

monochrome = cycler('linestyle', ['solid', 'dashed', '-.', 'dotted', (0, (3, 5, 1, 5, 1,
     5))]) + cycler('color', plt.rcParams['axes.prop_cycle'].by_key()['color'][:5])
plt.gca().set_prop_cycle(monochrome)
monochrome_markers = cycler('linestyle', ["None"]) * cycler('marker', ['x', 'o', 'd', '+
    ', 'P']) + cycler('color', plt.rcParams['axes.prop_cycle'].by_key()['color'][:5])
plt.gca().set_prop_cycle(monochrome_markers)

validation_params_model = {
    'nStations': list(range(1, 20)),
    'legacyFraction': [0],
    'frameSize': [1000],
    'channelWidth': [20],
    'Na': [256],
    'cwMin': [15, 127, 1023],
    'mcs': [5],
    'dl': ['su'],
    'ul': ['su'],
    'maxTxopDuration': [4800],
    'ackSeqType': ['AGGR-MU-BAR'],
    'scheduler': ['rr']
}

model_results = get_model_throughput(validation_params_model)
runs = 10
validation_params_sim = adapt_dictionary_for_sims(validation_params_model, step=2,
    simulationTime=5)
sim_results = get_simulation_metrics(validation_params_sim, runs, ns_path=NSpath, script
    =PScript, campaign_dir=outdir, overwrite=False, verbose=False)
plt.gca().set_prop_cycle(monochrome)
model_results.sel(metrics=['dl', 'ul']).sum('metrics').squeeze().plot.line(x='nStations'
    )
plt.gca().set_prop_cycle(monochrome_markers)
sim_results.sel(metrics=['dl', 'ul']).sum('metrics').mean('runs').squeeze().plot.line(x=
    'nStations')
plt.gca().set_prop_cycle(monochrome_markers)
plot_with_cis(sim_results.sel(metrics=['dl', 'ul']).sum('metrics'), runs, 'nStations')
plt.title("")
plt.xlabel("Number of STAs")
plt.ylabel("Throughput [Mbit/s]")
plt.xticks(validation_params_sim['nStations'])
plt.legend([r"Model, $CW_{\rm min} = 15$",
            r"Model, $CW_{\rm min} = 127$",
            r"Model, $CW_{\rm min} = 1023$",
            r"Sim, $CW_{\rm min} = 15$",
            r"Sim, $CW_{\rm min} = 127$",
            r"Sim, $CW_{\rm min} = 1023$"],
           framealpha=1)
plt.grid(True, axis='both', which='both')
plt.savefig('figures/suonly.pdf', bbox_inches='tight')
# plt.show()
```

```
validation_params_model = {
    'nStations': list(range(1, 10)),
    'legacyFraction': [0],
    'frameSize': [1000],
    'channelWidth': [20],
    'Na': [256],
    'cwMin': [15],
    'mcs': [5],
    'dl': ['mu'],
    'ul': ['su'],
    'maxTxopDuration': [4800],
    'ackSeqType': ['AGGR-MU-BAR'],
    'scheduler': ['rr', 'bellalta']
}
model_output = get_model_throughput(validation_params_model).squeeze()
plt.gca().set_prop_cycle(monochrome)
model_output.sel(metrics='dl').squeeze().plot.line(x='nStations')
runs = 10
validation_params_sim = adapt_dictionary_for_sims(validation_params_model, step=1,
    simulationTime=5)
sim_results = get_simulation_metrics(validation_params_sim, runs, ns_path=NSpath, script
    =PScript, campaign_dir=outdir, overwrite=False, verbose=False)
plt.gca().set_prop_cycle(monochrome_markers)
sim_results.sel(metrics='dl').mean('runs').squeeze().plot.line(x='nStations')
plot_with_cis(sim_results.sel(metrics='dl'), runs, 'nStations')
plt.title("")
plt.xlabel("Number of STAs")
plt.ylabel("Throughput [Mbit/s]")
plt.xticks(validation_params_sim['nStations'])
plt.legend([
    r"Model, $f_{bw}$",
    r"Model, $f_{hol}$",
    r"Sim, $f_{bw}$",
    r"Sim, $f_{hol}$",
            ], framealpha=1)
plt.grid(True, axis='both', which='both')
plt.savefig('figures/schedulers.pdf', bbox_inches='tight')
# plt.show()

validation_params_model = {
    'nStations': list(range(1, 10)),
    'legacyFraction': [0],
    'frameSize': [1000],
    'channelWidth': [20],
    'Na': [256],
    'cwMin': [15],
    'mcs': [5],
    'dl': ['mu'],
    'ul': ['None'],
    'maxTxopDuration': [4800],
    'ackSeqType': ['AGGR-MU-BAR'],
    'scheduler': ['rr', 'bellalta']
}
model_output = get_model_throughput(validation_params_model).squeeze()
plt.gca().set_prop_cycle(monochrome)
model_output.sel(metrics='hol').squeeze().plot.line(x='nStations')
runs = 2
validation_params_sim = adapt_dictionary_for_sims(validation_params_model, step=1,
    simulationTime=5)
sim_results = get_simulation_metrics(validation_params_sim, runs, ns_path=NSpath, script
    =PScript, campaign_dir=outdir, overwrite=False, verbose=False)
plt.gca().set_prop_cycle(monochrome_markers)
sim_results.sel(metrics='hol').mean('runs').squeeze().plot.line(x='nStations')
plt.title("")
plt.xlabel("Number of STAs")
plt.ylabel("Head-of-Line Delay [ms]")
plt.xticks(validation_params_sim['nStations'])
plt.legend([
    r"Model, $f_{bw}$",
    r"Model, $f_{hol}$",
    r"Sim, $f_{bw}$",
    r"Sim, $f_{hol}$",
    ], framealpha=1)
```

```
plt.grid(True, axis='both', which='both')
plt.savefig('figures/schedulers_hol.pdf', bbox_inches='tight')
# plt.show()

validation_params_model = {
    'nStations': list(range(1, 20)),
    'legacyFraction': [0],
    'frameSize': [1000],
    'channelWidth': [20],
    'Na': [256],
    'cwMin': [15],
    'mcs': [5],
    'dl': ['su', 'mu'],
    'ul': ['None'],
    'maxTxopDuration': [4800],
    'ackSeqType': ['ACK-SU-FORMAT', 'MU-BAR', 'AGGR-MU-BAR'],
    'scheduler': ['rr'],
}
model_output = get_model_throughput(validation_params_model)
plt.gca().set_prop_cycle(monochrome)
model_output.sel(metrics='dl', dl='su',
                 ackSeqType='ACK-SU-FORMAT').squeeze().plot.line(x='nStations')
model_output.sel(metrics='dl', dl='mu').squeeze().plot.line(x='nStations')
plt.gca().set_prop_cycle(monochrome_markers)
runs = 2
# SU
validation_params_sim = adapt_dictionary_for_sims(validation_params_model, step=4)
validation_params_sim['dl'] = ['su']
validation_params_sim['ackSeqType'] = ['NO-OFDMA']
validation_params_sim['dlFlowDataRate'] = [1]
validation_params_sim['ulFlowDataRate'] = [1]
sim_results = get_simulation_metrics(validation_params_sim, runs, ns_path=NSpath, script
    =PScript, campaign_dir=outdir, overwrite=False, verbose=False)
sim_results.sel(metrics='dl').mean('runs').squeeze().plot.line('x', x='nStations')
# MU
validation_params_sim['dl'] = ['mu']
validation_params_sim['ackSeqType'] = ['ACK-SU-FORMAT', 'MU-BAR', 'AGGR-MU-BAR']
sim_results = get_simulation_metrics(validation_params_sim, runs, ns_path=NSpath, script
    =PScript, campaign_dir=outdir, overwrite=False, verbose=False)
sim_results.sel(metrics='dl').mean('runs').squeeze().plot.line(x='nStations')
plt.title("")
plt.legend([
    r"Model, SU",
    r"Model, MU, ACK-SU-FORMAT",
    r"Model, MU, MU-BAR",
    r"Model, MU, AGGR-MU-BAR",
    r"Sim, SU",
    r"Sim, MU, ACK-SU-FORMAT",
    r"Sim, MU, MU-BAR",
    r"Sim, MU, AGGR-MU-BAR",
], framealpha=1)
plt.grid('both')
plt.xlabel("Number of STAs")
plt.ylabel("Throughput [Mbit/s]")
plt.xticks(validation_params_sim['nStations'])
plt.savefig('figures/acksequences.pdf', bbox_inches='tight')
# plt.show()
```

A.2.2 ofdma-validation/simulation.py

```
###############
# Simulation #
###############
import numpy as np
import copy
import sem
import matplotlib.pyplot as plt

def adapt_dictionary_for_sims(params, step=4, simulationTime=1, dataRate=100):
    validation_params_sim = copy.deepcopy(params)
    validation_params_sim['dlFlowDataRate'] = [dataRate]
    validation_params_sim['ulFlowDataRate'] = [dataRate]
```

```
    validation_params_sim['simulationTime'] = [simulationTime]
    params['nEhtStations'] = [0]
    validation_params_sim['nStations'] = list(range(params['nStations'][0], params['
    nStations'][-1]+1, step))
    return validation_params_sim

def plot_with_cis(results, runs, x, multiple_lines=False):
    results = results.squeeze()
    std = np.transpose(results.reduce(np.std, 'runs').data)
    avg = np.transpose(results.reduce(np.mean, 'runs').data)
    # ci = 1.96 * std / np.sqrt(runs)
    # ci = 2.32 * std / np.sqrt(runs)
    ci = 2.576 * std / np.sqrt(runs)
    # print(avg)
    # avg.plot.line('-', x=x)
    medianprops = dict(linestyle='None')
    boxprops = dict(linestyle='None')
    whiskerprops = dict(linewidth=1)
    ax = plt.gca()
    if len(avg.shape) > 1:
        for i in range(len(avg)):
            # p = ax.plot(results.coords[x].data, avg[i], ':')
            for x_idx, x_value in enumerate(results.coords[x].data):
                item = {}
                item["med"] = avg[i][x_idx]
                item["q1"] = avg[i][x_idx]
                item["q3"] = avg[i][x_idx]
                item["whislo"] = avg[i][x_idx] - ci[i][x_idx]
                item["whishi"] = avg[i][x_idx] + ci[i][x_idx]
                stats = [item]
                ax.bxp(stats,
                       positions=[x_value],
                       showfliers=False,
                       medianprops=medianprops,
                       boxprops=boxprops,
                       whiskerprops=whiskerprops)
                # ax.fill_between(results.coords[x].data, (avg[i]-ci[i]), (avg[i]+ci[i])
    , color=p[0].get_color(), alpha=.1)
    else:
        # p = ax.plot(results.coords[x].data, avg, ':')
        for x_idx, x_value in enumerate(results.coords[x].data):
            item = {}
            item["med"] = avg[x_idx]
            item["q1"] = avg[x_idx]
            item["q3"] = avg[x_idx]
            item["whislo"] = avg[x_idx] - ci[x_idx]
            item["whishi"] = avg[x_idx] + ci[x_idx]
            stats = [item]
            ax.bxp(stats,
                   positions=[x_value],
                   showfliers=False,
                   medianprops=medianprops,
                   boxprops=boxprops,
                   whiskerprops=whiskerprops)


def get_simulation_campaign(validation_params, runs, ns_path, script, campaign_dir,
    overwrite=False ):
    campaign = sem.CampaignManager.new(ns_path, script,
                                       campaign_dir,
                                       runner_type='ParallelRunner',
                                       overwrite=overwrite,
                                       optimized=False,
                                       check_repo=False,
                                       max_parallel_processes=8)
    params = copy.deepcopy(validation_params)
    params['verbose'] = [False]
    # params['printToFile'] = [True]
    campaign.run_missing_simulations(params, runs)
    return campaign

def get_metrics(result):
    lines = iter(result['output']['stdout'].splitlines())
```

```
    dl = ul = dllegacy = ullegacy = hol = dlmucomp = hetbcomp = 0
    # dls = uls = dlslegacy = ulslegacy = []
    dls = uls = []
    while (True):
        line = next(lines, None)
        if line is None:
            break
        if "Per-AC" in line:
            line = next(lines, None)
            line = next(lines, None)
            dls += [float(line.split(" ")[-1])]
            line = next(lines, None)
            line = next(lines, None)
            uls += [float(line.split(" ")[-1]) if line.split(" ")[-1] else 0]
            # line = next(lines, None)
            # line = next(lines, None)
            # dlslegacy += [float(line.split(" ")[-1])]
            # line = next(lines, None)
            # line = next(lines, None)
            # ulslegacy += [float(line.split(" ")[-1])]
        if "Throughput (Mbps) [DL]" in line:
            # Go down to total
            while (True):
                line = next(lines, None)
                if line is None:
                    break
                if "TOTAL:" in line:
                    dl = float(line.split(" ")[-1])
                    break
        if "Throughput (Mbps) [UL]" in line:
            # Go down to total
            while (True):
                line = next(lines, None)
                if line is None:
                    break
                if "TOTAL:" in line:
                    ul = float(line.split(" ")[-1])
                    break
        if "Throughput (Mbps) [DL] LEGACY" in line:
            # Go down to total
            while (True):
                line = next(lines, None)
                if line is None:
                    break
                if "TOTAL:" in line:
                    dllegacy = float(line.split(" ")[-1])
                    break
        if "Throughput (Mbps) [UL] LEGACY" in line:
            # Go down to total
            while (True):
                line = next(lines, None)
                if line is None:
                    break
                if "TOTAL:" in line:
                    ullegacy = float(line.split(" ")[-1])
                    break
        if "Pairwise Head-of-Line delay" in line:
            # Go down to total
            while (True):
                line = next(lines, None)
                if line is None:
                    break
                if "TOTAL:" in line:
                    count = float(line.split("[")[1].split("]")[0])
                    if (count == 0):
                        hol = 0
                        break
                    hol = float(line.split("<")[1].split(">")[0])
                    break
        if "DL MU PPDU completeness" in line:
            # Go down to total
            line = next(lines, None)
            line = next(lines, None)
```

```
            dlmucomp = 0
            break
        if "HE TB PPDU completeness" in line:
            # Go down to total
            line = next(next(lines, None))
            line = next(lines, None)
            hetbcomp = 0
            break
    return [dl, ul, dllegacy, ullegacy, hol, dlmucomp, hetbcomp]

def print_detailed_simulation_output (ns_path, script, campaign_dir, validation_params,
    runs, overwrite=False, verbose=False):

    campaign = sem.CampaignManager.new(ns_path, script,
                                       campaign_dir,
                                       runner_type='ParallelRunner',
                                       overwrite=overwrite,
                                       check_repo=False)
    params = copy.deepcopy(validation_params)
    params['verbose'] = [True]
    # params['printToFile'] = [True]
    campaign.run_missing_simulations(params, runs)

    example_result = next(campaign.db.get_complete_results(params))

    print(example_result['output']['stdout'])

    print(sem.utils.get_command_from_result(script, example_result, debug =False))

    for line in example_result['output']['stdout'].splitlines():
        if "Starting statistics" in line:
            start = int(line.split(" ")[-1])
        if "Stopping statistics" in line:
            stop = int(line.split(" ")[-1])

    print("Transmission times for devices: ")
    for line in example_result['output']['WifiPhyStateLog.txt'].splitlines():
        cur_time, context, time, duration, state = line.split(" ")
        if state == "TX" and float(time) > start and float(time) < stop:
            print("Time %s: device %s transmitted for %s" % (time, context, duration))

    print("Simulation throughput: %s" % get_metrics(example_result))
    return example_result


def get_simulation_metrics(validation_params, runs, ns_path, script, campaign_dir,
    overwrite=False, verbose=False):

    campaign = sem.CampaignManager.new(ns_path, script, campaign_dir,
                                       runner_type='ParallelRunner',
                                       overwrite=overwrite,
                                       check_repo=False,
                                       max_parallel_processes=8)

    params = copy.deepcopy(validation_params)
    params['verbose'] = verbose
    if runs == None:
        campaign.run_missing_simulations(params)
    else:
        campaign.run_missing_simulations(params, runs)

    if verbose:
        for result in campaign.db.get_results(params):
            print(sem.utils.get_command_from_result(script, result, debug=True))
    throughput_results = campaign.get_results_as_xarray( params, get_metrics, ['dl', 'ul
    ', 'dllegacy', 'ullegacy', 'hol', 'dlmucomp', 'hetbcomp'], runs).squeeze()

    return throughput_results
```

A.2.3 ofdma-validation/parsingfunctions.py

```
import numpy as np
```

```
from io import StringIO
import itertools
import xarray
import pandas as pd
import seaborn as sns
sns.set_style("whitegrid")
import copy
import sem
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib import cm
import ast

########################
# Per-Flow Statistics #
########################

def parse_per_flow_statistics(result):
    flowFiles = list(filter(lambda item: "flowStats" in item[0], result['output'].items
    ()))
    flow_counter = 1
    flows = []
    for flowFilename, flowContents in flowFiles:
        flowData = flowContents.splitlines()[0].split(" ")
        if len(flowContents.splitlines()) > 1:
            flowLatencies = flowContents.splitlines()[1].split(" ")[:-1]
        else:
            flowLatencies = []
        if len(flowLatencies) <= 1:
            flowLatencies = []
        flow = {
            "flowId": flow_counter,
            "bss": int(flowData[0]),
            "staId": int(flowData[1]),
            "direction": flowData[2],
            "ac": flowData[3],
            "transport": flowData[4],
            "throughput": float(flowData[5]),
            "expectedDataRate": float(flowData[6]),
            "actualDataRate": float(flowData[7]),
            "packetLossRate": float(flowData[8]),
            "droppedPackets": int(flowData[9]),
            "txBytes": int(flowData[10]),
            "txPackets": int(flowData[11]),
            "rxBytes": int(flowData[12]),
            "rxPackets": int(flowData[13]),
            "latencySamples": [float(i) for i in flowLatencies]
            }
        flows.append(flow)
        flow_counter += 1
    return flows

def get_flow_info(result):
    flows = parse_per_flow_statistics(result)
    if (len(flows) == 0):
        print(sem.utils.get_command_from_result('wifi-multi-ap', result))
        return [float('nan') for i in range(30)]
    return [list(f.values())[:-1] for f in flows]

def get_flow_latencies(result):
    flows = parse_per_flow_statistics(result)
    return [[flow['flowId'], flow['bss'], flow['ac'], flow['direction'], flow['transport
    '], item] for flow in flows for item in flow['latencySamples']]

def parse_pairwise_per_ac_statistics(result):
    parsed_statistics = []
    for f in result['output'].keys():
        if "pairwise" in f:
            contents = result['output'][f]
            expired, rejected, failed = [float(i) for i in contents.splitlines()[0].
    split(" ")]
            l2Latency = [float(i) for i in contents.splitlines()[1].split(" ")[:-1]]
            pairwiseHol = [float(i) for i in contents.splitlines()[2].split(" ")[:-1]]
```

```
            ampduSize = [float(i) for i in contents.splitlines()[3].split(" ")[:-1]]
            ampduRatio = [float(i) for i in contents.splitlines()[4].split(" ")[:-1]]
            stats = {
                'bss': int(f.split("-")[1]),
                'direction': str.upper(f.split("-")[2]),
                'staId': int(f.split("-")[3]),
                'ac': str.upper(f.split("-")[4]),
                'expired': [expired],
                'rejected': [rejected],
                'failed': [failed],
                'l2Latency': l2Latency,
                'pairwiseHol': pairwiseHol,
                'ampduSize': ampduSize,
                'ampduRatio': ampduRatio
                }
            parsed_statistics += [stats]
    return parsed_statistics

def get_pairwise_samples(result, quantity):
    parsed_statistics = parse_pairwise_per_ac_statistics(result)
    return [[stat['bss'], stat['direction'], stat['staId'], stat['ac'], item] for stat
    in parsed_statistics for item in stat[quantity]]

def parse_per_ac_statistics(result):
    parsed_statistics = []
    for f in result['output'].keys():
        if "perac" in f:
            contents = result['output'][f]
            droppedByQueueDisc = [float(i) for i in contents.splitlines()[0].split(" ")]
            txopDuration = [float(i) for i in contents.splitlines()[1].split(" ")[:-1]]
            queueDiscSojournTime = [float(i) for i in contents.splitlines()[2].split(" "
    )[:-1]]
            aggregateHol = [float(i) for i in contents.splitlines()[3].split(" ")[:-1]]
            stats = {
                'bss': int(f.split("-")[1]),
                'staId': int(f.split("-")[2]),
                'ac': str.upper(f.split("-")[3]),
                'droppedByQueueDisc': [droppedByQueueDisc],
                'txopDuration': txopDuration,
                'queueDiscSojournTime': queueDiscSojournTime,
                'aggregateHol': aggregateHol,
                }
            parsed_statistics += [stats]
    return parsed_statistics

def get_per_ac_samples(result, quantity):
    parsed_statistics = parse_per_ac_statistics(result)
    return [[stat['bss'], stat['staId'], stat['ac'], item] for stat in parsed_statistics
     for item in stat[quantity]]

def parse_per_bss_statistics(result, stat_name):
    stat = []
    for f in result['output'].keys():
        if stat_name in f:
            contents = result['output'][f]
            bss = int(f.split("-")[-1])
            if len(contents.splitlines()) > 0:
                stat.append([bss, [float(i) for i in contents.splitlines()[0].split(" ")
    [:-1]]])
    return stat

def get_aggregate_samples(result, quantity):
    parsed_statistics = parse_per_bss_statistics(result, quantity)
    return [[bss[0], item] for bss in parsed_statistics for item in bss[1]]

def get_phy_states(result):
    output = []
    for line in result['output']['WifiPhyStateLog.txt'].splitlines():
        cur_time, context, time, duration, state = line.split(" ")
        if state == "TX":
            output += [[int(context), int(time), int(duration), state]]
    return output
```

```
# Other plotting functions
##########################

def plot_phy_state (result, filename=None):
    plotstatemap = {
        "RX": False,
        "TX": True,
        "IDLE": False,
        "CCA_BUSY": False,
        "SWITCHING": False,
        "SLEEP": False,
        "OFF": False,
    }
    statecolormap = {
        "RX": 'r',
        "TX": 'b',
        "IDLE": 'grey',
        "CCA_BUSY": 'grey',
        "SWITCHING": 'grey',
        "SLEEP": 'grey',
        "OFF": 'grey',
    }
    preambletextmap = {
        "HE_TB": 'TB',
        "HE_SU": 'SU',
        "HE_MU": 'MU',
        "LONG": '',
    }
    statetextmap = {
        "RX": '',
        "TX": '',
        "IDLE": '',
        "CCA_BUSY": '',
        "SWITCHING": '',
        "SLEEP": '',
        "OFF": '',
    }
    psdutextmap = {
        'CTL_ACK': "A",
        'CTL_BACKRESP': "BA",
        'CTL_BACKREQ': "BACKREQ",
        'QOSDATA_NULL': "QoSNull",
        'QOSDATA': "",
        'CTL_TRIGGER': "TF",
        'MGT_BEACON': "Beacon",
        'MGT_ASSOCIATION_REQUEST': "ARq",
        'MGT_ASSOCIATION_RESPONSE': "ARe",
        'MGT_ACTION': "MA",
        "ANN": "ANN",
        "AQR": "AQR",
        "AQRP": "AQRP",
    }
    for line in result['output']['WifiPhyStateLog.txt'].splitlines():
        cur_time, context, time, duration, state = line.split(" ")
        cur_time = float(cur_time)
        context = float(context)
        time = float(time)
        duration = float(duration)
        if plotstatemap[state]:
            plt.plot([time, time+duration], [context, context],
                     statecolormap[state], linewidth=2)
            if statetextmap.get(state):
                plt.text(time+duration/2, context + 0.05,
                         statetextmap[state],
                         ha='center', fontsize=10, clip_on=True)
    allmpdutypes = set()
    allpreambles = set()
    for line in result['output']['PsduForwardedDown.txt'].splitlines():
        cur_time, context, preamble, *mpdu_types = line.strip().split(" ")
        cur_time = float(cur_time)
        context = float(context)
        contextoffset = 0
        allpreambles.add(preamble)
```

```
            plt.text(cur_time, context - 0.4,
                        preambletextmap.get(preamble, None),
                        ha='left', fontsize=10, clip_on=True)
            plt.scatter(cur_time, context, color='b', marker='x')
            for mpdu_type in mpdu_types:
                allmpdutypes.add(mpdu_type)
                if psdutextmap.get(mpdu_type):
                    plt.text(cur_time, context - 0.6 - contextoffset,
                                psdutextmap[mpdu_type],
                                ha='left', fontsize=10, clip_on=True)
                    contextoffset += 0.15
    # Plot TXOPs
    for line in result['output']['txops.txt'].splitlines():
        bss, nodeId, context, time, duration = line.strip().split(" ")
        bss = int(bss)
        nodeId = int(nodeId)
        context = int(context)
        time = int(time)
        duration = int(duration)
        plt.plot ([time, time+duration],
                    [context, context],
                    'grey',
                    linewidth=20,
                    solid_capstyle='butt',
                    alpha=0.3)
    print(allmpdutypes)
    print(allpreambles)
    nDevicesPerBss = result['params']['nEhtStations'] + result['params']['nHeStations']
    + 1
    nBsss = result['params']['nBss']
    apCount = 1
    staCount = 1
    for sta in range(int(nDevicesPerBss * nBsss)):
        staType = "AP"+str(apCount) if sta % nDevicesPerBss == 0 else "STA"+str(staCount
    )
        if sta % nDevicesPerBss == 0:
            apCount += 1
            staCount = 1
        else:
            staCount += 1
        plt.text(0.3*duration, sta, staType, fontsize=12, va='center')
    plt.title("Visualization of status of devices")
    plt.xlabel("Time [ns]")
    plt.ylim(bottom=-1, top=nDevicesPerBss*nBsss+1)
    plt.gca().spines['left'].set_visible(False)
    plt.yticks (list(range(15)), labels=["" for _ in range(15)])
    plt.grid(axis='x', which='both')
    if filename:
        plt.savefig(filename, bbox_inches='tight')
        plt.close()

def plot_device_locations (result, filename=None):
    colors=list(itertools.islice(iter(cm.rainbow(np.linspace(0,1,result['params']['nBss'
    ]))), result['params']['nBss']))
    plt.figure()
    for line in result['output']['locations.txt'].splitlines():
        bss, dev, x, y = line.split(" ")
        plt.text(float(x), float(y), dev, color=colors[int(bss)])
    plt.ylim(-20, 20)
    plt.xlim(-20, 20)
    plt.xlabel("X [m]")
    plt.ylabel("Y [m]")
    plt.title("Simulated network topology")
    if filename:
        plt.savefig(filename, bbox_inches='tight')
        plt.close()
    else:
        plt.show()

def plot_cw_and_bo (result):
    bo = pd.DataFrame(data=[[int(i) for i in entry.split(" ")] for entry in result['
    output']['BackoffLog.txt'].splitlines()],
                      columns=['time', 'device', 'value'])
```

```
    cw = pd.DataFrame(data=[[int(i) for i in entry.split(" ")] for entry in result['
    output']['CWLog.txt'].splitlines()],
                      columns=['time', 'device', 'oldCw', 'value'])
    cw = cw.drop(columns='oldCw')
    cw = cw.assign(quantity='cw')
    bo = bo.assign(quantity='bo')
    cwandbo = pd.concat([cw, bo], ignore_index=True)
    sns.relplot(data=cwandbo,
                x='time',
                y='value',
                row='device',
                style='quantity',
                kind='line')
```

A.2.4 ofdma-validation/model.py

```
#########
# Model #
#########
import xarray as xr
import pandas as pd
from tqdm import tqdm
from multiprocessing import Pool
import copy
import math
from itertools import product
import numpy as np
import scipy.optimize  # Used to solve the fixed-point problem
import scipy
from functools import reduce
import operator
import random

def dict_product(d):
    keys = d.keys()
    for element in product(*d.values()):
        yield dict(zip(keys, element))

def get_ru_number_bellalta(channel_width, N):
    """
    Map linking the number of users to the width of the assigned RUs, according
    to the model policy. This function tries to fit as many users in each
    transmission opportunity.
    """
    if channel_width == 20e6 and N >= 9:
            return 9
    elif channel_width == 40e6 and N >= 18:
            return 18
    elif channel_width == 80e6 and N >= 37:
            return 37
    return N


def get_ru_number_roundrobin(channel_width, N):
    """
    Map linking the number of users to the width of the assigned RUs, according
    to the Round Robin policy. This function tries to assign the largest size
    of RUs such that all users will be served within two DL TX opportunities.
    """
    if channel_width == 20e6:
        if (N >= 9):
            return 9
        elif (N >= 4):
            return 4
        elif (N >= 2):
            return 2
        elif (N >= 1):
            return 1
    elif channel_width == 40e6:
        if (N >= 18):
            return 18
        elif (N >= 8):
```

```
            return 8
        elif (N >= 4):
            return 4
        elif (N >= 2):
            return 2
        elif (N >= 1):
            return 1
    elif channel_width == 80e6:
        if (N >= 37):
            return 37
        elif (N >= 16):
            return 16
        elif (N >= 8):
            return 8
        elif (N >= 4):
            return 4
        elif (N >= 2):
            return 2
        elif (N >= 1):
            return 1
    elif channel_width == 160e6:
        if (N >= 74):
            return 74
        elif (N >= 32):
            return 32
        elif (N >= 16):
            return 16
        elif (N >= 8):
            return 8
        elif (N >= 4):
            return 4
        elif (N >= 2):
            return 2
        elif (N >= 1):
            return 1
    if (N == 0):
        return 1
    raise Exception("Unrecognized channel + N combination")


def get_data_subcarriers(channel_width, N_ru):
    """
    Map linking RU width with number of data subcarriers.
    """
    if channel_width == 20e6:
        if (N_ru <= 1):
            return 234
        elif (N_ru <= 2):
            return 102
        elif (N_ru <= 4):
            return 48
        elif (N_ru <= 9):
            return 24
    elif channel_width == 40e6:
        if (N_ru <= 1):
            return 468
        elif (N_ru <= 2):
            return 234
        elif (N_ru <= 4):
            return 102
        elif (N_ru <= 8):
            return 48
        elif (N_ru <= 18):
            return 24
    elif channel_width == 80e6:
        if (N_ru <= 1):
            return 980
        elif (N_ru <= 2):
            return 468
        elif (N_ru <= 4):
            return 234
        elif (N_ru <= 8):
            return 102
```

```
        elif (N_ru <= 16):
            return 48
        elif (N_ru <= 37):
            return 24
    elif channel_width == 160e6:
        if (N_ru <= 1):
            return 2*980
        elif (N_ru <= 2):
            return 980
        elif (N_ru <= 4):
            return 468
        elif (N_ru <= 8):
            return 234
        elif (N_ru <= 16):
            return 102
        elif (N_ru <= 32):
            return 48
        elif (N_ru <= 74):
            return 24


def get_rate_and_coding_for_modulation(mcs):
    """
    Return the bits/symbol and the coding rate associated to a certain MCS.
    """
    bits_per_symbol = [1, 2, 2, 4, 4, 6, 6, 6, 8, 8, 10, 10]
    code_rate = [1/2, 1/2, 3/4, 1/2, 3/4, 2/3, 3/4, 5/6, 3/4, 5/6, 3/4, 5/6]
    return [bits_per_symbol[mcs], code_rate[mcs]]


def compute_model_throughput(params, print_stuff=False):

    if params['mcs'] == 'ideal':
        mcs = 11
    else:
        mcs = params['mcs']
    Y_m, Y_c = get_rate_and_coding_for_modulation(mcs)
    NHe = params['nStations'] # Number of user stations
    L_D = params['frameSize'] * 8 # Frame size in bits
    N_a = params['Na']  # Number of aggregated frames

    if (params['dl'] == 'None'):  # Disable AP traffic
        CW_min_ap = 15e1000  # Minimum contention window
    else:
        CW_min_ap = params['cwMin']  # Minimum contention window

    if params['ul'] == 'None':
        CW_min_he_sta = 15e1000  # Disable STA traffic
    else:
        CW_min_he_sta = params['cwMin']  # STAs contend just like the AP

    # Number of backoff stages
    m_ap = m_he_sta = 6
    # OFDM symbol duration (12.8 symbol + 3.2 GI)
    # sigma = 16e-6
    # OFDM symbol duration (12.8us symbol + 800ns GI)
    sigma = 12.8e-6 + 800e-9
    # OFDM symbol duration (12.8us symbol + 1600ns GI)
    sigma_1600 = 12.8e-6 + 1.6e-6
    # Duration of an empty backoff slot
    T_e = 9e-6
    # Channel width
    B = params['channelWidth']*1e6
    # Function determining how we get the number of Resource Units to employ.
    if params['scheduler'] == 'rr':
        ru_number_function = get_ru_number_roundrobin
    elif params['scheduler'] == 'bellalta':
        ru_number_function = get_ru_number_bellalta
    else:
        raise Exception("Unknown scheduler: %s" % params['scheduler'])
    # Get the number of resource units to employ in MU transmissions
    V_u = max(1, min(NHe, ru_number_function(B, NHe)))
    # Number of data sub-carriers in the bandwidth for a single RU
```

```
Y_sc = get_data_subcarriers(B, ru_number_function(B, NHe))
# Number of data sub-carriers in the whole channel, for SU transmissions
Y_sc_su = get_data_subcarriers(B, ru_number_function(B, 1))
# We assume a single Spatial Stream
V_s = 1
# Bits per OFDM symbol in the current setup, in MU transmissions
r = V_s * Y_m * Y_c * Y_sc
# Bits per OFDM symbol for SU transmissions
r_su = V_s * Y_m * Y_c * Y_sc_su
# Bits/OFDM symbol in 6Mbps legacy mode
r_legacy_6 = 1 * 48 * 1/2
# Bits/OFDM symbol in 24Mbps legacy mode
r_legacy_24 = 4 * 48 * 1/2
# Legacy OFDM symbol duration
sigma_legacy = 4e-6
# Whether to employ MU or SU traffic at the AP
alpha = 0 if params['dl'] == 'mu' else 1
beta = 0
if params['dl'] == 'mu' and (params['ul'] == 'su' or params['ul'] == 'None'):
    beta = 1
elif params['dl'] == 'mu' and params['ul'] == 'mu':
    beta = 0.5
# TXOP duration, given in microseconds -> Convert to seconds
maxTxopDuration = params['maxTxopDuration'] * 1e-6
# PHY header lengths for different kinds of packets
T_PHY_HE_TB = 97e-6
T_PHY_HE_MU = 48e-6
T_PHY_legacy = 20e-6
T_PHY_HE_SU = 44e-6
L_MD = 4*8  # MPDU Delimiter
L_MH = 26*8  # MAC header length
L_UDP = 8*36  # UDP header length
# Length of an MPDU: UDP header + MAC header + APP Payload
L_MPDU = L_UDP + L_MH + L_D
# MAC trailer
L_MT = 4*8
# See MpduAggregator for padding formula
L_PAD = ((4 - ((L_MPDU/8) % 4)) % 4) * 8
# The following is the length if we were to instruct each STA
# L_MU_BAR_TF = (8 + V_u * 9 + 2) * 8
# However, since we are aggregating this TF, we only need information about
# a single user (i.e., the destination STA)
L_MU_BAR_TF = (8 + V_u * 9 + 2) * 8
# L_trigger_BASIC is the length of the TF MPDU instructing devices on how
# to perform simultaneous transmissions.
# 8 Header + 6 User info * nUsers + 2 Padding
# 16 as header, 4 as padding
L_trigger_BASIC = 8 * (16 + (8 + 6 * V_u + 2) + 4)  # In bits
# Length of the TF sent by the AP to poll for BSRs
L_trigger_BSRP = 224 + 48 * V_u
# This is the length of the BSR report as a response to a BSRP
# There are 8 QOS_NULL frames, with 2 bytes of padding
# We subtract 2 * 8 since the last aggregated frame has no padding.
L_BSR = (8 * (L_MD + 30 * 8 + 2 * 8) - 2 * 8)
# Length of the Multi-User Block ACK, sent in response to a MU DL
# transmission.
L_BACK_MU = 56 * 8
# Length of Single-User ACKs
L_BACK = 56 * 8
# Short Inter-Frame Spacing
T_SIFS = 16e-6
# Arbitration Inter-Frame Spacing
T_AIFS = 34e-6
# Length of Multi-User A-MPDUs. These include MU_BAR_TF, the Trigger Frame
# used to coordinate the simultaneous UL MU BACK
# TODO If aggregated, do not add BAR
L_MU_BAR = L_MD + L_MU_BAR_TF
# Length of Single-User A-MPDUs, the same as above but without the
# aggregation of a MU_BAR_TF.
# Here we subtract L_PAD since there is no padding in the last MPDU

T_trigger_basic = (T_PHY_legacy + np.ceil((L_trigger_BASIC) / r_legacy_6) *
                   sigma_legacy)
```

```
T_MU_BAR = (T_PHY_legacy + np.ceil((L_MU_BAR) / r_legacy_6) *
                    sigma_legacy)
T_BACK = (T_PHY_legacy + np.ceil((L_BACK) / r_legacy_24) * sigma_legacy)
# 16e-6 is the PHY HEADER duration
T_BSR = (T_PHY_HE_TB + 16e-6 + np.ceil((L_BSR) / r) * sigma)
# T_BACK_MU = (T_PHY_HE_TB + np.ceil((L_BACK_MU) / r) * sigma)
T_BACK_MU = 112e-6
# T_BAR = (T_PHY_HE_MU + 16e-6 + np.ceil(L_MU_BAR / r) * sigma)
T_BAR = 32e-6
# T_MU_BAR = 172e-6



# Computation of the length of a MU DL PPDU
def get_L_MU_DL_AMPDU(n_a_mu_dl):
    if params['ackSeqType'] == 'ACK-SU-FORMAT':
        return (n_a_mu_dl * (L_MD + L_MPDU + L_MT + L_PAD) - L_PAD)
    elif params['ackSeqType'] == 'MU-BAR':
        return (n_a_mu_dl * (L_MD + L_MPDU + L_MT + L_PAD) - L_PAD)
    elif params['ackSeqType'] == 'AGGR-MU-BAR':
        return (n_a_mu_dl * (L_MD + L_MPDU + L_MT + L_PAD) + (L_MD + L_MU_BAR_TF))

def get_T_mu_d_D(n_a_mu_dl):
    return T_PHY_HE_MU + 16e-6 + np.ceil(get_L_MU_DL_AMPDU(n_a_mu_dl) / r) * sigma

def get_T_mu_d(n_a_mu, V_u):
    T_mu_d_D = get_T_mu_d_D(n_a_mu)
    if params['ackSeqType'] == 'ACK-SU-FORMAT':
        return (T_mu_d_D + T_SIFS + T_BACK + (V_u - 1) * (T_SIFS + T_BAR + T_SIFS +
T_BACK) + T_AIFS)
    elif params['ackSeqType'] == 'MU-BAR':
        return (T_mu_d_D + T_SIFS + T_MU_BAR + T_SIFS + T_BACK_MU + T_AIFS)
    elif params['ackSeqType'] == 'AGGR-MU-BAR':
        return (T_mu_d_D + T_SIFS + T_BACK_MU + T_AIFS)

# Build the longest MU PPDU that will fit in the TXOP
n_a_mu_dl = 1
if maxTxopDuration != 0:
    while True:
        if (n_a_mu_dl > N_a or get_T_mu_d(n_a_mu_dl, V_u) > maxTxopDuration):
            n_a_mu_dl -= 1
            break
        n_a_mu_dl += 1
else:
    n_a_mu_dl = N_a
L_MU_DL_AMPDU = get_L_MU_DL_AMPDU(n_a_mu_dl)
T_mu_d_D = get_T_mu_d_D(n_a_mu_dl)




# Computation of the length of a MU DL PPDU
def get_L_MU_UL_AMPDU(n_a_mu_ul):
    return (n_a_mu_ul * (L_MD + L_MPDU + L_MT + L_PAD) - L_PAD)

# Computation of the length of a MU DL PPDU
def get_T_mu_u_D(L_MU_UL_AMPDU):
    L_MU_UL_AMPDU = get_L_MU_UL_AMPDU(n_a_mu_ul)
    return T_PHY_HE_TB + 16e-6 + np.ceil(L_MU_UL_AMPDU / r) * sigma_1600

def get_T_mu_u(n_a_mu_ul):
    T_mu_u_D = get_T_mu_u_D(get_L_MU_UL_AMPDU(n_a_mu_ul))
    return (T_trigger_basic + T_SIFS + T_mu_u_D + T_SIFS + T_BACK + T_SIFS)

# Build the longest MU PPDU that will fit in the TXOP
n_a_mu_ul = 1
if maxTxopDuration != 0:
    while True:
        if (n_a_mu_ul > N_a or get_T_mu_u(n_a_mu_ul) > maxTxopDuration):
            n_a_mu_ul -= 1
            break
```

```
            n_a_mu_ul += 1
    else:
        n_a_mu_ul = N_a
    L_MU_UL_AMPDU = get_L_MU_UL_AMPDU(n_a_mu_ul)
    T_mu_u_D = get_T_mu_u_D(L_MU_UL_AMPDU)




    # Computation of the length of a SU PPDU
    def get_L_SU_AMPDU(n_a_su):
        return n_a_su * (L_MD + L_MPDU + L_MT + L_PAD) - L_PAD

    def get_T_su_d(n_a_su):
        L_SU_AMPDU = get_L_SU_AMPDU(n_a_su)
        return T_PHY_HE_SU + np.ceil(L_SU_AMPDU / r_su) * sigma

    # Build the longest SU PPDU that will fit in the TXOP
    n_a_su = 1
    if maxTxopDuration != 0:
        while True:
            if (n_a_su >= N_a or get_T_su_d(n_a_su) > maxTxopDuration):
                n_a_su -= 1
                break
            n_a_su += 1
        n_a_su -= 1
    else:
        n_a_su = N_a
    L_SU_AMPDU = get_L_SU_AMPDU(n_a_su)
    T_su_D = get_T_su_d(n_a_su)




    ######################
    # Core of the model #
    ######################

    def core_function(x):

        # print(x)

        tau_ap, tau_he_sta, p_c_ap, p_c_he_sta = x

        n_he_sta = NHe

        def E(CW_min, m, p_c):
            return (((1 - p_c - p_c * (2 * p_c) ** m) / (1 - 2 * p_c)) *
                    ((CW_min + 1) / 2) - 1/2)

        p_c_ap = 1 - (1 - tau_he_sta)**n_he_sta

        if n_he_sta > 0:
            p_c_he_sta = 1 - ((1 - tau_ap) *
                              (1 - tau_he_sta)**(n_he_sta-1))
        else:
            p_c_he_sta = 0

        tau_ap = 1 / (E(CW_min_ap, m_ap, p_c_ap) + 1)

        if n_he_sta > 0:
            tau_he_sta = 1 / (E(CW_min_he_sta, m_he_sta, p_c_he_sta) + 1)
        else:
            tau_he_sta = 0

        outcome = [tau_ap, tau_he_sta, p_c_ap, p_c_he_sta]

        return np.array(outcome)
```

```
# Use Fixed Point Iteration to solve the problem
initial_guess = np.zeros(4)
tau_ap, tau_he_sta, p_c_ap, p_c_he_sta = scipy.optimize.fixed_point(core_function,
initial_guess)

E_ap = (((1 - p_c_ap - p_c_ap * (2 * p_c_ap) ** m_ap) / (1 - 2 * p_c_ap)) * ((
CW_min_ap + 1) / 2) - 1/2)
E_he_sta = (((1 - p_c_ap - p_c_ap * (2 * p_c_ap) ** m_ap) / (1 - 2 * p_c_ap)) * ((
CW_min_ap + 1) / 2) - 1/2)

#############
# Formulas #
#############

# TODO Convert this
T_trigger_basic = (T_PHY_legacy + np.ceil((L_trigger_BASIC) / r_legacy_6) *
                   sigma_legacy)
T_BACK = (T_PHY_legacy + np.ceil((L_BACK) / r_legacy_24) * sigma_legacy)
# 16e-6 is the PHY HEADER duration
T_BSR = (T_PHY_HE_TB + 16e-6 + np.ceil((L_BSR) / r) * sigma)

# Computation of transmission times
T_su = (T_su_D + T_SIFS + T_BACK + T_SIFS)
T_mu_d = get_T_mu_d(n_a_mu_dl, V_u)
T_mu_u = maxTxopDuration

# This is a collision between two SU transmissions (a STA in UL SU and
# another STA in UL SU)
T_c_su = T_su_D + T_SIFS
# This is a collision between a DL MU transmissions and an UL SU
# transmission (an AP and a STA)
T_c_mu_dl = T_mu_d
T_c_mu_ul_sta = T_mu_u
T_c_mu_ul_ap = 96e-6 + T_SIFS


#################################################################
# Probabilities of having a certain outcome at a given slot #
#################################################################

n_he_sta = int(NHe)

# Probability of picking a HE device for a DL transmission
p_he_device = NHe / (params['nStations'])

# Probability the AP initiates a DL SU transmission, and there is no
# collision.
a1 = (tau_ap *
      p_he_device *
      alpha *
      (1 - p_c_ap))

# Probability exactly a single non-backed-off STA transmits, and there is no
collision
a2 = (n_he_sta *
      tau_he_sta *
      (1 - p_c_he_sta))

# Probability the AP initiates a MU DL transmission, no collision
a3 = ((1 - alpha) *
      beta *
      tau_ap *
      p_he_device *
      (1 - p_c_ap))

# Probability the AP initiates a MU UL transmission, no collision
a4 = ((1 - alpha) *
      (1 - beta) *
      tau_ap *
      p_he_device *
      (1 - p_c_ap))
```

```
# Probability no one tries to transmit in a given slot
b1 = ((1 - tau_ap) *
      (1 - tau_he_sta)**n_he_sta)

p_he_tx = (1 - (1 - tau_he_sta)**n_he_sta)
p_atleast_one_sta = p_he_tx

# Probability of a collision for a DL SU transmission by the AP
c1 = (alpha *
      tau_ap *
      p_c_ap)

# Probability of a collision for a DL MU transmission by the AP
c2 = ((1 - alpha) *
      beta *
      tau_ap *
      p_c_ap)

# Probability of a collision for an UL MU transmission by the AP
c3 = ((1 - alpha) *
      (1 - beta) *
      tau_ap *
      p_c_ap)

# Probability of a collision between stations
c4 = 1 - a1 - a2 - a3 - a4 - b1 - c1 - c2 - c3

#########################
# UL and DL throughput #
#########################

T_a1 = T_su
T_a2 = T_su
T_a3 = T_mu_d
T_a4 = T_mu_u
T_c1 = T_c_su
T_c2_sta = T_c_su
T_c2_ap = T_c_mu_dl
T_c3_sta = T_c_su
T_c3_ap = T_c_mu_ul_ap
T_c4 = T_c_su

all_cases_sta = (b1 * T_e +
             a1 * (T_a1 + T_e) +
             a2 * (T_a2 + T_e) +
             a3 * (T_a3 + T_e) +
             a4 * (T_a4 + T_e) +
             c1 * (T_c1 + T_e) +
             c2 * (T_c2_sta + T_e) +
             c3 * (T_c3_sta + T_e) +
             c4 * (T_c4 + T_e))

all_cases_ap = (b1 * T_e +
             a1 * (T_a1 + T_e) +
             a2 * (T_a2 + T_e) +
             a3 * (T_a3 + T_e) +
             a4 * (T_a4 + T_e) +
             c1 * (T_c1 + T_e) +
             c2 * (T_c2_ap + T_e) +
             c3 * (T_c3_ap + T_e) +
             c4 * (T_c4 + T_e))

# Throughput for HE devices
S_d = ((a1 * n_a_su * L_D + a3 * V_u * n_a_mu_dl * L_D) / all_cases_ap)
S_u = ((a2 * n_a_su * L_D + a4 * V_u * n_a_mu_ul * L_D) / all_cases_sta)

dl_throughput = S_d / 1e6
ul_throughput = S_u / 1e6

#################
# Head-of-line #
#################
```

```
    if params['dl'] == 'su':
        hol = NHe * (T_su + E_ap * T_e) * 1000
    else:
        hol = NHe / V_u * (T_mu_d + E_ap * T_e) * 1000

    if print_stuff:
        print("\n--- Relevant parameters ---\n")
        print("Transmission_rate: %s Mb/s" % (r/sigma/1e6))
        print("DL_MU_A-MPDU_size: %s bytes" % (L_MU_DL_AMPDU / 8))
        print("UL_MU_A-MPDU_size: %s bytes" % (L_MU_UL_AMPDU / 8))
        print("SU_A-MPDU_size: %s bytes" % (L_SU_AMPDU / 8))
        print("SU_Aggregation: %s" % n_a_su)
        print("DL_MU_Aggregation: %s" % n_a_mu_dl)
        print("UL_MU_Aggregation: %s" % n_a_mu_ul)
        print("alpha: %s" % alpha)
        print("beta: %s" % beta)
        print("V_u: %s" % V_u)

        print("\n--- Time quantities ---\n")
        print("T_su_D: %1.1f us" % (T_su_D * 1e6))
        print("T_mu_d_D: %1.1f us" % (T_mu_d_D * 1e6))
        print("T_mu_u_D: %1.1f us" % (T_mu_u_D * 1e6))
        print("T_trigger_basic: %1.1f us" % (T_trigger_basic * 1e6))
        print("L_trigger_basic: %s bytes" % (L_trigger_BASIC/8))
        print("L_BSR: %s bytes" % (L_BSR/8))
        print("T_BSR: %1.1f us" % (T_BSR * 1e6))
        print("T_BACK: %1.1f us" % (T_BACK * 1e6))
        print("T_BACK_MU: %1.1f us" % (T_BACK_MU * 1e6))
        print("T_BAR: %1.1f us" % (T_BAR * 1e6))
        print("T_MU_BAR: %1.1f us" % (T_MU_BAR * 1e6))

        print("\n--- Exchange durations ---\n")
        print("MU_DL: %1.1f us" % (T_mu_d * 1e6))
        print("MU_UL: %1.1f us" % (T_mu_u * 1e6))

        print("\n--- Channel access probabilities ---\n")
        print("n_he_sta: %s" % n_he_sta)
        print("n_he_sta_mu: %s" % n_he_sta_mu)
        print("tau_ap: %s" % tau_ap)
        print("tau_he_sta: %s" % tau_he_sta)
        print("tau_he_sta_mu: %s" % tau_he_sta_mu)
        print("p_c_ap: %s" % p_c_ap)
        print("p_c_he_sta: %s" % p_c_he_sta)
        print("p_c_he_sta_mu: %s" % p_c_he_sta_mu)
        print("E_ap: %s" % E_ap)
        print("E_he_sta: %s" % E_he_sta)
        print("E_he_sta_mu: %s" % E_he_sta_mu)
        print("Successful_DL_SU_probability_a1: %s" % a1)
        print("Successful_UL_SU_probability_a2: %s" % a2)
        print("Successful_DL_MU_probability_a3: %s" % a3)
        print("Successful_UL_MU_probability_a4: %s" % a4)
        print("Collided_DL_SU_probability_c1: %s" % c1)
        print("Collided_DL_MU_probability_c2: %s" % c2)
        print("Collided_UL_MU_probability_c3: %s" % c3)
        print("Collided_UL_STAs_probability_c4: %s" % c4)
        print("Empty_slot_probability_b1: %s" % b1)
        print("Sum_of_probabilities: %s" % (a1 + a2 + a3 + a4 + a5 + a6 + a7 + b1 + c1 +
     c2 + c3 + c4))

        print("T_a1: %s" % T_a1)
        print("T_a2: %s" % T_a2)
        print("T_a3: %s" % T_a3)
        print("T_a4: %s" % T_a4)
        print("T_a5: %s" % T_a5)
        print("T_a6: %s" % T_a6)
        print("T_a7: %s" % T_a7)

        print("T_c1: %s" % T_c1)
        print("T_c2_sta: %s" % T_c2_sta)
        print("T_c2_ap: %s" % T_c2_ap)
        print("T_c3_sta: %s" % T_c3_sta)
        print("T_c3_ap: %s" % T_c3_ap)
        print("T_c4: %s" % T_c4)
```

```
        print("\n--- Throughput ---\n")
        print("TOT_throughput: %s" % (dl_throughput+ul_throughput))
        print("DL_throughput: %s" % dl_throughput)
        print("UL_throughput: %s" % ul_throughput)
        print("HoL: %s" % hol)

    return [dl_throughput, ul_throughput,
            hol]


def print_detailed_model_output(params):
    compute_model_throughput(list(dict_product(params))[0], True)

def get_model_throughput(validation_params):
    params_with_metrics = copy.deepcopy(validation_params)
    params_with_metrics['metrics'] = ['dl', 'ul', 'hol']
    model_output = xr.DataArray(np.zeros([len(i) for i in
                                          params_with_metrics.values()]),
                                          list(zip(list(params_with_metrics.keys()),
                                                   params_with_metrics.values())))
    with Pool() as pool:
        param_list = list(dict_product(validation_params))
        for param_comb, result in zip(param_list,
                                      pool.imap(compute_model_throughput,
                                                tqdm(param_list,
                                                     total=len(param_list),
                                                     desc="Running model",
                                                     unit="Parameter combination"))):
            model_output.loc[param_comb] = result
    return model_output
```

A.2.5 wifi-ofdma-validation.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2019
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Authors: Stefano Avallone <stavallo@unina.it>
 *          Sebastien Deronne <sebastien.deronne@gmail.com>
 * Modified by: Davide Magrin <magrinda@uw.edu>
 */

#include "ns3/command-line.h"
#include "ns3/config.h"
#include "ns3/string.h"
#include "ns3/pointer.h"
#include "ns3/log.h"
#include "ns3/spectrum-wifi-helper.h"
#include "ns3/ssid.h"
#include "ns3/mobility-helper.h"
#include "ns3/wifi-net-device.h"
#include "ns3/sta-wifi-mac.h"
#include "ns3/ap-wifi-mac.h"
#include "ns3/qos-txop.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/application-container.h"
#include "ns3/packet-sink-helper.h"
```

```
#include "ns3/packet-sink.h"
#include "ns3/on-off-helper.h"
#include "ns3/v4ping-helper.h"
#include "ns3/multi-model-spectrum-channel.h"
#include "ns3/wifi-mac-queue.h"
#include "ns3/wifi-psdu.h"
#include "ns3/ctrl-headers.h"
#include "ns3/traffic-control-helper.h"
#include "ns3/traffic-control-layer.h"
#include "ns3/he-configuration.h"
#include "ns3/wifi-acknowledgment.h"
#include <vector>
#include <map>
#include <unordered_map>
#include <set>
#include <cmath>
#include <iomanip>
#include <sstream>
#include <numeric>
#include <fstream>
#include <algorithm>
#include <type_traits>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("WifiOfdmaExample");

/**
 * \brief Example to test DL/UL OFDMA
 *
 * This example creates an 11ax BSS (the BSS under test) and a configurable
 * number of overlapping BSSes made of stations of a given technology (11n,
 * 11ac or 11ax).
 *
 * Usage: ./waf --run "wifi-ofdma [options]"
 *
 * The available options can be listed by typing:
 *
 * ./waf --run "wifi-ofdma --help"
 *
 * It is mandatory to pass a text file (via the trafficFile option) specifying
 * the traffic flows to generate. Empty lines or lines beginning with '#'
 * in such text file are ignored. The syntax of a line is:
 *
 * <START_STA_ID> [<END_STA_ID>] <AC> <L4PROTO> <DIR> <PKTSIZE> IDT|CBR <VALUE>
 *
 * where:
 *
 * - START_STA_ID is the ID of the non-AP STA which is the sender/receiver
 *   (depending on the traffic direction) of this flow. IDs start at 1.
 * - if END_STA_ID (>= START_STA_ID) is specified, flows will be generated
 *   for every station having ID included in START_STA_ID..END_STA_ID.
 * - AC is the Access Category ('BE', 'BK', 'VI' or 'VO').
 * - L4PROTO is the transport layer protocol ('UDP' or 'TCP').
 * - DIR can be 'DL' for a single downlink flow, 'UL' for a single uplink flow
 *   and 'DL+UL' for two symmetric flows (downlink and uplink).
 * - PKTSIZE is the packet size in bytes.
 * - if 'IDT' is specified, VALUE represents the inter-departure time (in
 *   milliseconds) between two consecutive packets. If 'CBR' is specified,
 *   VALUE represents the data rate (in Mbps).
 */
class WifiOfdmaExample
{
public:
  // Helper class to store min, max, avg and count statistics
  template <typename T>
  struct Stats
  {
    Stats (uint16_t val = 0) {}   // to allow init to 0 (in PrintStatsWithTotal)
    void AddSample (T value);
    void RemoveSample (T value);
    Stats<T>& operator+= (const Stats<T>& rhs);  // merge samples
```

```
  std::multiset<T> m_samples;
};

struct Flow
{
  enum
  {
    DOWNLINK,
    UPLINK
  } m_direction;
  enum
  {
    TCP,
    UDP
  } m_l4Proto;
  AcIndex m_ac;
  uint16_t m_stationId;     // starting at 1
  uint16_t m_dstPort;
  uint32_t m_payloadSize;   // bytes
  double m_dataRate;        // b/s
  /* Measured end-to-end metrics */
  Stats<double> m_latency;
  uint64_t m_txBytes {0};
  uint64_t m_txPackets {0};
  uint64_t m_packetsRejectedBySocket{0};
  uint64_t m_rxBytes {0};
  uint64_t m_rxPackets {0};
  uint64_t m_prevRxBytes {0};
  /*
   * For UDP flows, map packet's UID to the time it was transmitted. For TCP flows,
   * map the total amount of bytes transmitted at the time a packet was sent to
   * the time the packet was transmitted.
   */
  std::unordered_map<uint64_t, Time> m_inFlightPackets;
};

/**
 * Create an example instance.
 */
WifiOfdmaExample ();
virtual ~WifiOfdmaExample ();
/**
 * Parse the options provided through command line.
 */
void Config (int argc, char *argv[]);
/**
 * Read the specs of the traffic flows to generate.
 */
void GenerateTrafficFlows (void);
/**
 * Setup nodes, devices and internet stacks.
 */
void Setup (void);
/**
 * Run simulation.
 */
void Run (void);
/**
 * Print results.
 */
void PrintResults (std::ostream& os);
/**
 * Make the current station associate with the AP.
 */
void StartAssociation (void);
/**
 * Make the AP establish a BA agreement with the current station.
 */
void EstablishBaAgreement (Mac48Address bssid);
/**
 * Start a client application.
 */
void StartClient (OnOffHelper client, std::size_t i, Ptr<Node> node);
```

```
/**
 * Start an OBSS client application.
 */
void StartObssClient (OnOffHelper client, uint16_t bss, Ptr<Node> node);
/**
 * Delay the start of traffic generation.
 */
void DelayStart (void);
/**
 * Start generating traffic.
 */
void StartTraffic (void);
/**
 * Start collecting statistics.
 */
void StartStatistics (void);
/**
 * Stop collecting statistics.
 */
void StopStatistics (void);
/**
 * Report that an MPDU was not correctly received.
 */
void NotifyTxFailed (Ptr<const WifiMacQueueItem> mpdu);
/**
 * Report that the lifetime of an MSDU expired.
 */
void NotifyMsduExpired (Ptr<const WifiMacQueueItem> item);
/**
 * Report that an MSDU was dropped before enqueue into EDCA queue.
 */
void NotifyMsduRejected (Ptr<const WifiMacQueueItem> item);
/**
 * Report that a packet was transmitted by the i-th application.
 */
void NotifyAppTx (std::size_t i, Ptr<const Packet> packet);
/**
 * Report that a packet was received by the i-th application.
 */
void NotifyAppRx (std::size_t i, Ptr<const Packet> packet, const Address &address);
/**
 * Report that a packet was enqueued in an EDCA queue.
 */
void NotifyEdcaEnqueue (Ptr<const WifiMacQueueItem> item);
/**
 * Report that the MAC layer is forwarding up a received packet.
 */
void NotifyMacForwardUp (Ptr<const Packet> p);
/**
 * Report that an MSDU was dequeued from the EDCA queue.
 */
void NotifyMsduDequeuedFromEdcaQueue (Time maxDelay, Ptr<const WifiMacQueueItem> item)
  ;
/**
 * Report that PSDUs were forwarded down to the PHY.
 */
void NotifyPsduForwardedDown (WifiConstPsduMap psduMap, WifiTxVector txVector, double
  txPowerW);
/**
 * Report the TXOP duration when a TXOP ends.
 */
void NotifyTxopDuration (uint32_t nodeId, AcIndex ac, Time startTime, Time duration);
/**
 * Report the sojourn time of a packet dequeued from a qdisc
 */
void NotifySojournTime (std::size_t nodeId, AcIndex ac, Time sojournTime);
/**
 * Report that a packet has been dropped after dequeue
 */
void NotifyDropAfterDequeue (std::size_t nodeId, AcIndex ac, Ptr<const QueueDiscItem>
  item, const char* reason);
/**
 * Store the per-flow amount of bytes received so far
```

```
   */
  void StoreCumulativeRxBytes (void);
  /**
   * Parse context strings of the form "/NodeList/x/DeviceList/y/" to extract the NodeId
   */
  uint32_t ContextToNodeId (const std::string &context);

  void NotifyStateChange (std::string context, Time oldt, Time duration, WifiPhyState
    newState);

  void NotifyCwChange (uint32_t oldCw, uint32_t newCw);
  void NotifyBackoffChange (uint32_t backoff);

  static const std::map<AcIndex, std::string> m_aciToString;
  static uint16_t m_nEcdfSamples;       // by default, minimum, median and maximum

private:
  template <typename T>
  void ReadParamList (std::string s, std::vector<T>& vec);

  template <typename T, typename FUNC>
  void PrintStatsWithTotal (std::vector<std::map<AcIndex, T>> v, FUNC select,
                            std::ostream& os, std::string s, std::string s0 = "");
  template <typename T, typename FUNC>
  void PrintStats (std::vector<std::map<AcIndex, T>> v, FUNC select,
                   std::ostream& os, std::string s, std::string s0 = "");

  /* Parameters */
  // double m_simulationTime{20}; // seconds
  double m_simulationTime{60}; // seconds
  uint32_t m_startInterval{10};
  int m_na{5};
  std::string m_dlTraffic{"mu"};
  std::string m_ulTraffic{"mu"};
  double m_ulFlowDataRate{1}; // If -1, full queues
  double m_dlFlowDataRate{1}; // If -1, full queues
  // duration of the interval in which apps can start sending packets (ms)
  uint16_t m_nHeStations{4};
  uint16_t m_nVhtStations{0};
  double m_legacyFraction{0};
  uint16_t m_nStations; // number of non-AP stations in the BSS under test
  uint16_t m_nObss{0}; // number of overlapping BSSs
  uint16_t m_nStationsPerObss{0}; // number of non-AP stations per each OBSS
  WifiStandard m_obssStandard;
  double m_radius{0}; // meters
  bool m_enableDlOfdma{true};
  bool m_forceDlOfdma{true};
  bool m_enableUlOfdma{false};
  bool m_enableTxopSharing{false};
  bool m_enableBsrp{false};
  bool m_useCentral26TonesRus {false};
  uint32_t m_ulPsduSize{2000}; // bytes
  uint16_t m_channelWidth{20}; // channel bandwidth (MHz)
  uint8_t m_channelNumber{36};
  WifiPhyBand m_band {WIFI_PHY_BAND_UNSPECIFIED};
  uint16_t m_guardInterval{800}; // GI in nanoseconds
  bool m_enableObssPd{false};
  double m_obssPdThresholdBss{-99.0};
  double m_obssPdThresholdMinBss{-82.0};
  double m_obssPdThresholdMaxBss{-62.0};
  bool m_enableThresholdPreambleDetection{false};
  uint32_t m_obssDlPayload{1400}; // bytes
  uint32_t m_obssUlPayload{1400}; // bytes
  double m_obssDlAggregateRate{50.0}; // Mbps
  double m_obssUlAggregateRate{50.0}; // Mbps
  double m_powerSta{17.0}; // dBm
  double m_powerAp{17.0}; // dBm
  double m_psdLimitation {10.0};        // dBm/MHz
  double m_ccaThresholdSta{-62}; // dBm
  double m_ccaThresholdAp{-62}; // dBi
  double m_txGain{0.0}; // dBi
  double m_rxGain{0.0}; // dBi
  double m_rxSensitivity{-91.0};
```

```
uint16_t m_maxNRus{18}; // max number of RUs per MU PPDU
std::string m_heRate{"ideal"}; // "ideal" or MCS value
uint16_t m_beMaxAmsduSize{0}; // maximum A-MSDU size for BE
uint16_t m_bkMaxAmsduSize{0}; // maximum A-MSDU size for BK
uint16_t m_viMaxAmsduSize{0}; // maximum A-MSDU size for VI
uint16_t m_voMaxAmsduSize{0}; // maximum A-MSDU size for VO
uint32_t m_beMaxAmpduSize{8388607u}; // maximum A-MPDU size for BE
uint32_t m_bkMaxAmpduSize{8388607u}; // maximum A-MPDU size for BK
uint32_t m_viMaxAmpduSize{8388607u}; // maximum A-MPDU size for VI
uint32_t m_voMaxAmpduSize{8388607u}; // maximum A-MPDU size for VO
uint32_t m_obssMaxAmpduSize{65535}; // maximum A-MPDU size for overlapping BSSes
double m_beTxopLimit{4800}; // microseconds
double m_bkTxopLimit{4800}; // microseconds
double m_viTxopLimit{4800}; // microseconds
double m_voTxopLimit{4800}; // microseconds
double m_obssTxopLimit{2528}; // microseconds
uint32_t m_macQueueSize{10000}; // packets
uint32_t m_beMsduLifetime{20000}; // milliseconds
uint32_t m_bkMsduLifetime{500}; // milliseconds
uint32_t m_viMsduLifetime{500}; // milliseconds
uint32_t m_voMsduLifetime{100}; // milliseconds
bool m_useExplicitBar{false}; // use explicit BAR after missed Block Ack
// uint16_t m_muBeAifsn{3};
// uint16_t m_muBkAifsn{0};
// uint16_t m_muViAifsn{0};
// uint16_t m_muVoAifsn{0};
uint16_t m_BeCwMin{15};
// uint16_t m_muBeCwMin{15};
// uint16_t m_muBkCwMin{15};
// uint16_t m_muViCwMin{15};
// uint16_t m_muVoCwMin{15};
// uint16_t m_muBeCwMax{8191};
// uint16_t m_muBkCwMax{8191};
// uint16_t m_muViCwMax{8191};
// uint16_t m_muVoCwMax{8191};
// uint32_t m_beMuEdcaTimer{0}; // microseconds
// uint32_t m_bkMuEdcaTimer{0}; // microseconds
// uint32_t m_viMuEdcaTimer{0}; // microseconds
// uint32_t m_voMuEdcaTimer{0}; // microseconds
bool m_enableRts{false};
std::string m_dlAckSeqType{"ACK-SU-FORMAT"};
uint16_t m_baBufferSize{256};
std::string m_queueDisc{"default"};
bool m_enablePcap{false};
double m_warmup{0.5}; // duration of the warmup period (seconds)
uint32_t m_tcpSegmentSize{1500}; // TCP maximum segment size (0 = use default)
uint32_t m_tcpInitialCwnd{0}; // TCP initial congestion window size (segments, 0 = use
   default)
uint32_t m_tcpMinRto{500}; // TCP minimum retransmit timeout (milliseconds, 0 = use
  default)
std::string m_trafficFile; // name of file describing traffic flows to generate
bool m_verbose{false};
uint16_t m_nIntervals{20}; // number of intervals in which the simulation time is
  divided
uint16_t m_elapsedIntervals{0};
std::string m_scheduler = "rr";

int m_frameSize{1500}; // Size of APP layer packets in bytes
std::string m_ssidPrefix{"network-"};
std::vector<double> m_apDistances, m_apThetas;
NodeContainer m_apNodes;
NodeContainer m_staNodes;
std::vector<NodeContainer> m_obssStaNodes;
NetDeviceContainer m_apDevices;
NetDeviceContainer m_staDevices;
std::vector<NetDeviceContainer> m_obssStaDevices;
Ipv4InterfaceContainer m_apInterfaces;
Ipv4InterfaceContainer m_staInterfaces;
std::vector<Ipv4InterfaceContainer> m_obssStaInterfaces;
uint16_t m_currentSta {0};         // index of the current station
ApplicationContainer m_sinkApps;
std::vector<ApplicationContainer> m_obssSinkApps;
std::vector<Ptr<Application>> m_clientApps;
```

```
std::vector<ApplicationContainer> m_obssClientApps;
std::vector<Flow> m_flows;

std::vector<uint64_t> m_obssDlRxStart, m_obssDlRxStop;
std::vector<uint64_t> m_obssUlRxStart, m_obssUlRxStop;
Stats<double> m_dlMuCompleteness; // ratio of actual amount of bytes to max amount of
  bytes
                                  // (given its duration) carried in a DL MU PPDU
Stats<double> m_dlMuPpduDuration; // TX duration (ms) of DL MU PPDUs
uint64_t m_nBasicTriggerFramesSent {0};
uint64_t m_nFailedBasicTriggerFrames {0};  // no station responded (with a QoS Data or
   BAR frame)
uint64_t m_nBsrpTriggerFramesSent {0};
uint64_t m_nFailedBsrpTriggerFrames {0};  // no station responded with a QoS Null
  frame
Stats<double> m_heTbCompleteness; // ratio of sum of durations of A-MPDUs carrying QoS
   Data/BAR frames in
                                  // an HE TB PPDU to sum of durations granted by
  Basic Trigger Frame
Time m_tfUlLength {Seconds (0)};  // TX duration coded in UL Length subfield of
  Trigger Frame
Stats<double> m_heTbPpduDuration; // TX duration (ms) coded in UL Length subfield of
  Basic TFs
Time m_overallTimeGrantedByTf {Seconds (0)}; // m_tfUlLength times the number of
  addressed stations
Time m_durationOfResponsesToLastBasicTf {Seconds (0)}; // sum of the durations of the
  HE TB PPDUs (excluding
                                                       // QoS Null frames) in response
   to the last Basic TF
uint64_t m_countOfNullResponsesToLastTf {0}; // count of QoS Null frames sent in
  response to the last TF
TriggerFrameType m_lastTfType;   // type of the last Trigger Frame sent (Basic or Bsrp
  )

// Metrics that can be measured (sender side) for each (AP,STA) pair (DL) or
// (STA,AP) pair (UL) and for each Access Category
struct PairwisePerAcStats
{
  uint64_t expired {0};
  uint64_t rejected {0};          // dropped before enqueue into EDCA queue
  uint64_t failed {0};
  Stats<double> l2Latency;
  Time lastTxTime {Seconds (0)};
  Stats<double> pairwiseHol;      // pairwise Head-of-Line delay *DL only*
  Stats<uint32_t> ampduSize;      // size (bytes) of MPDUs sent to/received from each
  STAs
  Stats<double> ampduRatio;       // ratio of the duration of the A-MPDU sent to (for
  DL) or received
                                  // from (for UL) a STA to the duration of the DL MU
  PPDU or HE TB PPDU
};
std::vector<std::map<AcIndex, PairwisePerAcStats>> m_dlPerStaAcStats;   // A vector
  element per station (DL)
std::vector<std::map<AcIndex, PairwisePerAcStats>> m_ulPerStaAcStats;   // A vector
  element per station (UL)

// Metrics that can be measured (sender side) for each Access Category
// and are independent of the receiver
struct PerAcStats
{
  Stats<double> txopDuration;
  Stats<double> queueDiscSojournTime;
  uint32_t droppedByQueueDisc {0};
  Time lastTxTime {Seconds (0)};
  Stats<double> aggregateHoL;     // aggregate Head-of-Line delay
};
std::vector<std::map<AcIndex, PerAcStats>> m_perAcStats;  // first vector element for
  the AP, then one per STA

struct InFlightPacketInfo
{
  Mac48Address m_srcAddress;
  Mac48Address m_dstAddress;
```

```
    AcIndex m_ac;
    Ptr<const Packet> m_ptrToPacket;
    Time m_edcaEnqueueTime {Seconds (0)};  // time the packet was enqueued into an EDCA
    queue
  };
  std::unordered_map<uint64_t /* UID */, std::list<InFlightPacketInfo>>
    m_inFlightPacketMap;

  std::vector<uint64_t> m_nSolicitingBasicTriggerFrames;

  // Function object to compute the hash of a MAC address
  struct MacAddressHash
  {
    std::size_t operator()(const Mac48Address& address) const;
  };

  std::unordered_map <Mac48Address, uint32_t, MacAddressHash> m_staMacAddressToNodeId;

  /**
   * Return the ID of the node containing the device having the given address.
   */
  uint32_t MacAddressToNodeId (Mac48Address address);
  /**
   * Return an iterator to the PairwisePerAcStats object corresponding to the given MPDU
     header.
   */
  std::pair<std::map<AcIndex, PairwisePerAcStats>::iterator, bool>
    GetPairwisePerAcStats (const WifiMacHeader& hdr, AcIndex = AC_UNDEF);
  /**
   * Return the OBSS id and the index of station having the given id.
   */
  std::pair<uint16_t, std::size_t> StationIdToBssIndexPair (std::size_t id);
};

const std::map<AcIndex, std::string> WifiOfdmaExample::m_aciToString = { {AC_BE, "BE"},
    {AC_BK, "BK"},
                                                                          {AC_VI, "VI"},
    {AC_VO, "VO"}, };

uint16_t WifiOfdmaExample::m_nEcdfSamples = 3;       // by default, minimum, median and
    maximum

template <typename T>
void
WifiOfdmaExample::Stats<T>::AddSample (T value)
{
  m_samples.insert (value);
}

template <typename T>
void
WifiOfdmaExample::Stats<T>::RemoveSample (T value)
{
  auto it = m_samples.find (value);
  if (it != m_samples.end ())
    {
      m_samples.erase (it);
    }
}

template <typename T>
WifiOfdmaExample::Stats<T>&
WifiOfdmaExample::Stats<T>::operator+= (const Stats<T>& rhs)
{
  this->m_samples.insert (rhs.m_samples.begin (), rhs.m_samples.end ());
  return *this;
}

template <typename T>
std::ostream& operator<< (std::ostream& os, const WifiOfdmaExample::Stats<T> &stats)
{
//   os << std::fixed << std::setprecision (3)
//      << "(" << stats.m_min << ", " << stats.m_avg << ", " << stats.m_max << ", " <<
```

```
    stats.m_count << ") ";
  os << std::fixed << std::setprecision (3) << "(";
  uint16_t m = WifiOfdmaExample::m_nEcdfSamples - 1;   // number of sub-intervals of
    [0,1]
  std::size_t pos = 0, count = stats.m_samples.size ();
  double sum = 0;

  if (count > 0)
    {
      for (auto it = stats.m_samples.begin (); it != stats.m_samples.end(); ++it)
        {
          sum += *it;
        }
    }

  if (count > 0)
    {
      auto it = stats.m_samples.begin ();
      for (uint8_t i = 0; i <= m; i++)
        {
          std::advance (it, pos);
          os << *it << (i < m ? ", " : "");
          pos = (i + 1) * (count - 1) / m - i * (count - 1) / m;
        }
    }
  os << ")[" << count << "]"
     << "<" << sum / double (count) << ">";
  return os;
}

std::ostream &
operator<< (std::ostream &os, const WifiOfdmaExample::Flow &flow)
{
  os << "{staId=" << flow.m_stationId
     << (flow.m_direction == WifiOfdmaExample::Flow::DOWNLINK ? " DL " : " UL ")
     << "AC=" << WifiOfdmaExample::m_aciToString.at (flow.m_ac)
     << (flow.m_l4Proto == WifiOfdmaExample::Flow::TCP ? " TCP " : " UDP ") << flow.
    m_payloadSize
     << "B " << std::defaultfloat << flow.m_dataRate << "bps}";
  return os;
}

std::size_t
WifiOfdmaExample::MacAddressHash::operator() (const Mac48Address &address) const
{
  uint8_t buffer[6];
  address.CopyTo (buffer);
  std::string s (buffer, buffer + 6);
  return std::hash<std::string>{}(s);
}

std::pair<uint16_t, std::size_t>
WifiOfdmaExample::StationIdToBssIndexPair (std::size_t id)
{
  NS_ABORT_MSG_IF (id < m_nStations, "Station is part of the BSS under test");
  return std::make_pair ((id - m_nStations) / m_nStationsPerObss + 1,
                         (id - m_nStations) % m_nStationsPerObss);
}

template <typename T>
void
WifiOfdmaExample::ReadParamList (std::string s, std::vector<T> &vec)
{
  std::size_t start = 0, end;
  do
    {
      end = s.find_first_of (",", start);
      std::stringstream ss (s.substr (start, end));
      T tmp;
      NS_ABORT_MSG_IF (!(ss >> tmp), "Unexpected value " << ss.str ());
      vec.push_back (tmp);
      if (end != std::string::npos)
        {
```

```
          start = end + 1;
        }
    }
  while (end != std::string::npos);
}

WifiOfdmaExample::WifiOfdmaExample ()
{
}

WifiOfdmaExample::~WifiOfdmaExample ()
{
  m_dlPerStaAcStats.clear ();
  m_ulPerStaAcStats.clear ();
  m_perAcStats.clear ();
  m_inFlightPacketMap.clear ();
}

void
WifiOfdmaExample::Config (int argc, char *argv[])
{
  NS_LOG_FUNCTION (this);
  std::string simScheduler ("map");
  std::string obssStandard = "11ax";
  std::string psdLimitRegulator = "FCC";
  std::string apDistance, apTheta;

  CommandLine cmd;
  cmd.AddValue ("verbose", "Whether to print additional output", m_verbose);
  cmd.AddValue ("frameSize", "Size of APP frames", m_frameSize);
  cmd.AddValue ("nStations", "Number of non-AP stations in the BSS under test",
    m_nStations);
  cmd.AddValue ("legacyFraction", "Fraction of stations using 802.11ac",
    m_legacyFraction);
  cmd.AddValue ("channelWidth", "Channel bandwidth (20, 40, 80, 160)", m_channelWidth);
  cmd.AddValue ("dl", "Whether the AP should generate traffic (toward all STAs)",
    m_dlTraffic);
  cmd.AddValue ("ul", "Whether all STAs should generate traffic (toward the AP)",
    m_ulTraffic);
  cmd.AddValue ("mcs", "'ideal' or the constant MCS value to transmit HE PPDUs",
    m_heRate);
  cmd.AddValue ("Na", "Maximum allowed number of aggregated MPDUs", m_na);
  cmd.AddValue ("cwMin", "Value of the QosTxop CWmin parameter", m_BeCwMin);
  // cmd.AddValue ("muCwMin", "Value of the QosTxop CWmin parameter", m_muBeCwMin);
  // cmd.AddValue ("muEdcaTimer", "Duration of the MU EDCA Parameter Set",
    m_beMuEdcaTimer);
  cmd.AddValue ("dlFlowDataRate", "Data rate of each flow", m_dlFlowDataRate);
  cmd.AddValue ("ulFlowDataRate", "Data rate of each flow", m_ulFlowDataRate);
  cmd.AddValue ("ackSeqType", "ACK Sequence to employ for DL tx", m_dlAckSeqType);
  cmd.AddValue ("maxTxopDuration", "TXOP duration for BE in microseconds", m_beTxopLimit
    );
  cmd.AddValue ("scheduler", "Scheduler to employ [rr, my]", m_scheduler);
  cmd.AddValue ("simulationTime", "Time to simulate", m_simulationTime);
  cmd.Parse (argc, argv);

  // if (m_muBeCwMin == 0)
  //   {
  //     // m_muBeAifsn = 0;
  //     // m_muBeCwMin = 32767;
  //     // m_muBeCwMax = 32767;
  //   }
  // else
  //   {
  //     m_muBeCwMax = (64 * (m_muBeCwMin + 1)) - 1;
  //   }

  m_nVhtStations = (int) (m_nStations * m_legacyFraction);
  m_nHeStations = (int) (m_nStations - m_nVhtStations);

  if (m_dlTraffic != "mu")
    {
      m_dlAckSeqType = "NO-OFDMA";
    }
```

```
m_enableDlOfdma = (m_dlAckSeqType != "NO-OFDMA");
m_enableUlOfdma = m_ulTraffic == "mu";
// m_enableBsrp = m_ulTraffic == "mu";
m_enableBsrp = false;
m_forceDlOfdma = m_dlTraffic == "mu";
if (m_dlTraffic == "mu")
  {
    m_maxNRus = std::max (1, (int) m_nHeStations);
  }
else
  {
    m_maxNRus = 1;
  }
// m_ulPsduSize = 200 + (m_frameSize + 72) * m_na;
m_ulPsduSize = 2500;
m_beMaxAmpduSize = 200 + (m_frameSize + 72) * m_na;

// if (m_beMuEdcaTimer > 0)
//   {
//     m_bkMuEdcaTimer = m_beMuEdcaTimer;
//     m_viMuEdcaTimer = m_beMuEdcaTimer;
//     m_voMuEdcaTimer = m_beMuEdcaTimer;
//   }

ObjectFactory factory;

if (simScheduler == "map")
  {
    factory.SetTypeId ("ns3::MapScheduler");
  }
else if (simScheduler == "list")
  {
    factory.SetTypeId ("ns3::ListScheduler");
  }
else if (simScheduler == "heap")
  {
    factory.SetTypeId ("ns3::HeapScheduler");
  }
else if (simScheduler == "cal")
  {
    factory.SetTypeId ("ns3::CalendarScheduler");
  }
else
  {
    NS_ABORT_MSG ("Unknown simulator scheduler: " << simScheduler);
  }
Simulator::SetScheduler (factory);

NS_ABORT_MSG_IF (m_nVhtStations > 0 && m_heRate != "ideal",
                 "Cannot use constant rate manager when both VHT and HE stations are
  present");

if (m_maxNRus == 0)
  {
    // we do not want to use OFDMA
    m_dlAckSeqType = "NO-OFDMA";
  }

m_enableDlOfdma = (m_dlAckSeqType != "NO-OFDMA");

switch (m_channelWidth)
  {
  case 20:
    m_channelNumber = 36;
    break;
  case 40:
    m_channelNumber = 38;
    break;
  case 80:
    m_channelNumber = 42;
    break;
  case 160:
    m_channelNumber = 50;
```

```
      break;
    default:
      NS_ABORT_MSG ("Invalid channel bandwidth (must be 20, 40, 80 or 160)");
    }

  if (obssStandard == "11ax")
    {
      m_obssStandard = WIFI_STANDARD_80211ax_5GHZ;
    }
  else if (obssStandard == "11ac")
    {
      m_obssStandard = WIFI_STANDARD_80211ac;
    }
  else if (obssStandard == "11n")
    {
      m_obssStandard = WIFI_STANDARD_80211n_5GHZ;
    }
  else
    {
      NS_ABORT_MSG ("Invalid standard for OBSS (choose among 11ax, 11ac, 11n)");
    }

  if (m_obssStandard != WIFI_STANDARD_80211ax_5GHZ)
    {
      m_enableObssPd = false;
    }

  if (psdLimitRegulator == "FCC")
    {
      m_psdLimitation = 11.0; // 11 dBm/MHz, FCC limit for 5150 to 5350 MHz range for
    STAs
    }
  else if (psdLimitRegulator == "ETSI")
    {
      m_psdLimitation = 10.0; // 10 dBm/MHz, ETSI limit for 5150 to 5350 MHz range
    }
  else
    {
      NS_ABORT_MSG ("Invalid regulation body for PSD limitation (choose among FCC, ETSI)
    ");
    }

  m_nStations = m_nHeStations + m_nVhtStations;

  GenerateTrafficFlows ();
  NS_ABORT_MSG_IF (m_flows.empty (), "No traffic flow specified!");

  if (!apDistance.empty ())
    {
      ReadParamList (apDistance, m_apDistances);
    }
  NS_ABORT_MSG_IF (m_apDistances.size () != m_nObss,
                   "The size of the AP distance vector ("
                       << m_apDistances.size ()
                       << ") does not match the number of overlapping BSSes (" <<
    m_nObss << ")");

  if (!apTheta.empty ())
    {
      ReadParamList (apTheta, m_apThetas);
    }
  NS_ABORT_MSG_IF (m_apThetas.size () != m_nObss,
                   "The size of the AP theta vector ("
                       << m_apThetas.size () << ") does not match the number of
    overlapping BSSes ("
                       << m_nObss << ")");

  std::cout << "Channel bw = " << m_channelWidth << " MHz" << std::endl
            << (m_heRate == "ideal" ? "Ideal rate manager" : "HE MCS = " + m_heRate) <<
    std::endl
            << "Number of stations = " << m_nStations << std::endl
            << "EDCA queue max size = " << m_macQueueSize << " MSDUs" << std::endl
            << "MSDU lifetime = " << m_beMsduLifetime << " ms [BE]  " <<
```

```
    m_bkMsduLifetime
            << " ms [BK]  " << m_viMsduLifetime << " ms [VI]  " << m_voMsduLifetime << "
     ms [VO]  "
            << std::endl
            << "BA buffer size = " << m_baBufferSize << std::endl;
  if (m_enableDlOfdma)
    {
      std::cout << "Ack sequence = " << m_dlAckSeqType << std::endl;
    }
  else
    {
      std::cout << "No OFDMA" << std::endl;
    }
  std::cout << std::endl;
}

void
WifiOfdmaExample::GenerateTrafficFlows ()
{
  uint16_t dstPort = 7000;
  uint16_t endStaId = m_nStations;

  for (uint16_t staId = 1; staId <= endStaId; staId++)
    {
      if (m_dlTraffic != "None")
        {
          Flow flow;
          flow.m_ac = AC_BE;
          flow.m_l4Proto = Flow::UDP;
          flow.m_payloadSize = m_frameSize;
          flow.m_stationId = staId;
          flow.m_dataRate = m_dlFlowDataRate * 8 * 1e6;
          flow.m_direction = Flow::DOWNLINK;
          flow.m_dstPort = dstPort++;
          NS_LOG_DEBUG ("Adding flow " << flow);
          m_flows.push_back (flow);
        }
      if (m_ulTraffic != "None")
        {
          Flow flow;
          flow.m_ac = AC_BE;
          flow.m_l4Proto = Flow::UDP;
          flow.m_payloadSize = m_frameSize;
          flow.m_stationId = staId;
          flow.m_dataRate = m_ulFlowDataRate * 8 * 1e6;
          flow.m_direction = Flow::UPLINK;
          flow.m_dstPort = dstPort++;
          NS_LOG_DEBUG ("Adding flow " << flow);
          m_flows.push_back (flow);
        }
    }
}

void
WifiOfdmaExample::NotifyCwChange (uint32_t oldCw, uint32_t newCw)
{
  NS_LOG_FUNCTION ("Call to NotifyCwChange" << unsigned (oldCw) << unsigned (newCw));

  std::string filename = "CwLog.txt";
  std::ofstream outFile;
  outFile.open (filename.c_str (), std::ios_base::out | std::ios_base::app);
  if (!outFile.is_open ())
    {
      NS_LOG_ERROR ("Can't open file " << filename);
      return;
    }

  uint32_t c = Simulator::GetContext ();

  outFile << Simulator::Now ().GetSeconds () << " " << unsigned (c) << " " << unsigned (
    oldCw)
          << " " << unsigned (newCw) << std::endl;
}
```

```
void
WifiOfdmaExample::NotifyBackoffChange (uint32_t backoff)
{
  NS_LOG_FUNCTION ("Call to NotifyBackoffChange" << unsigned(backoff));

  std::string filename = "BackoffLog.txt";
  std::ofstream outFile;
  outFile.open (filename.c_str (), std::ios_base::out | std::ios_base::app);
  if (!outFile.is_open ())
    {
      NS_LOG_ERROR ("Can't open file " << filename);
      return;
    }

  uint32_t c = Simulator::GetContext();

  outFile << Simulator::Now ().GetSeconds () << " " << unsigned (c) << " " << unsigned (
    backoff)
          << std::endl;
}

void
WifiOfdmaExample::NotifyStateChange (std::string context, Time oldt, Time duration,
                                     WifiPhyState newState)
{
  NS_LOG_FUNCTION ("Call to NotifyStateChange" << context << oldt << duration <<
    newState);

  std::string filename = "WifiPhyStateLog.txt";
  std::ofstream outFile;
  outFile.open (filename.c_str (), std::ios_base::out | std::ios_base::app);
  if (!outFile.is_open ())
    {
      NS_LOG_ERROR ("Can't open file " << filename);
      return;
    }

  uint32_t c = ContextToNodeId (context);

  outFile << Simulator::Now ().GetSeconds () << " " << c << " " << oldt.GetMicroSeconds
    () << " "
          << duration.GetMicroSeconds () << " " << newState << std::endl;
}

void
WifiOfdmaExample::Setup (void)
{
  NS_LOG_FUNCTION (this);

  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
                      m_enableRts ? StringValue ("0") : StringValue ("999999"));
  Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (999999));
  Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (999999));
  Config::SetDefault ("ns3::HeConfiguration::GuardInterval", TimeValue (NanoSeconds (
    m_guardInterval)));
  Config::SetDefault ("ns3::WifiPhy::PowerDensityLimit", DoubleValue (m_psdLimitation));
  Config::SetDefault ("ns3::ArpCache::AliveTimeout", TimeValue (Seconds (3600 * 24)));
    // ARP cache entries expire after one day
  Config::SetDefault ("ns3::WifiMacQueue::MaxSize", QueueSizeValue (QueueSize (PACKETS,
    m_macQueueSize)));
  Config::SetDefault ("ns3::HeConfiguration::MpduBufferSize", UintegerValue (
    m_baBufferSize));
  Config::SetDefault ("ns3::QosTxop::UseExplicitBarAfterMissedBlockAck", BooleanValue (
    m_useExplicitBar));
  // Config::SetDefault ("ns3::HeConfiguration::MuBeAifsn", UintegerValue (m_muBeAifsn))
    ;
  // Config::SetDefault ("ns3::HeConfiguration::MuBkAifsn", UintegerValue (m_muBkAifsn))
    ;
  // Config::SetDefault ("ns3::HeConfiguration::MuViAifsn", UintegerValue (m_muViAifsn))
    ;
  // Config::SetDefault ("ns3::HeConfiguration::MuVoAifsn", UintegerValue (m_muVoAifsn))
    ;
```

```
// Config::SetDefault ("ns3::HeConfiguration::MuBeCwMin", UintegerValue (m_muBeCwMin))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuBkCwMin", UintegerValue (m_muBkCwMin))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuViCwMin", UintegerValue (m_muViCwMin))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuVoCwMin", UintegerValue (m_muVoCwMin))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuBeCwMax", UintegerValue (m_muBeCwMax))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuBkCwMax", UintegerValue (m_muBkCwMax))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuViCwMax", UintegerValue (m_muViCwMax))
  ;
// Config::SetDefault ("ns3::HeConfiguration::MuVoCwMax", UintegerValue (m_muVoCwMax))
  ;
// Config::SetDefault ("ns3::HeConfiguration::BeMuEdcaTimer", TimeValue (MicroSeconds
  (m_beMuEdcaTimer)));
// Config::SetDefault ("ns3::HeConfiguration::BkMuEdcaTimer", TimeValue (MicroSeconds
  (m_bkMuEdcaTimer)));
// Config::SetDefault ("ns3::HeConfiguration::ViMuEdcaTimer", TimeValue (MicroSeconds
  (m_viMuEdcaTimer)));
// Config::SetDefault ("ns3::HeConfiguration::VoMuEdcaTimer", TimeValue (MicroSeconds
  (m_voMuEdcaTimer)));

// Try to minimize the chances of collision among flows
Config::SetDefault ("ns3::FqCoDelQueueDisc::Perturbation", UintegerValue (9973));
Config::SetDefault ("ns3::FqCoDelQueueDisc::EnableSetAssociativeHash", BooleanValue (
  true));

if (m_tcpSegmentSize != 0)
  {
    Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue (m_tcpSegmentSize
  ));
  }
if (m_tcpInitialCwnd != 0)
  {
    Config::SetDefault ("ns3::TcpSocket::InitialCwnd", UintegerValue (m_tcpInitialCwnd
  ));
  }
if (m_tcpMinRto != 0)
  {
    Config::SetDefault ("ns3::TcpSocketBase::MinRto", TimeValue (MilliSeconds (
  m_tcpMinRto)));
  }


Ptr<MultiModelSpectrumChannel> spectrumChannel = CreateObject<
  MultiModelSpectrumChannel> ();
Ptr<FriisPropagationLossModel> lossModel = CreateObject<FriisPropagationLossModel> ();
spectrumChannel->AddPropagationLossModel (lossModel);
Ptr<ConstantSpeedPropagationDelayModel> delayModel = CreateObject<
  ConstantSpeedPropagationDelayModel> ();
spectrumChannel->SetPropagationDelayModel (delayModel);

for (uint16_t bss = 0; bss < m_nObss + 1; bss++)
  {
    NodeContainer apNode, heStaNodes, vhtStaNodes, obssStaNodes;

    apNode.Create (1);
    m_apNodes.Add (apNode); // AP of the BSS under test is Node 0

    if (bss == 0)
      {
        // BSS under test
        heStaNodes.Create (m_nHeStations);
        m_staNodes.Add (heStaNodes);
        vhtStaNodes.Create (m_nVhtStations);
        m_staNodes.Add (vhtStaNodes);
      }
    else
      {
        // Overlapping BSS
```

```
        obssStaNodes.Create (m_nStationsPerObss);
        m_obssStaNodes.push_back (obssStaNodes);
      }

    SpectrumWifiPhyHelper phy;
    phy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
    phy.SetErrorRateModel ("ns3::NistErrorRateModel");
    phy.SetChannel (spectrumChannel);
    phy.Set ("ChannelNumber", UintegerValue (m_channelNumber));
    phy.Set ("ChannelWidth", UintegerValue (m_channelWidth));
    if (m_enableThresholdPreambleDetection)
      {
        phy.SetPreambleDetectionModel ("ns3::ThresholdPreambleDetectionModel");
      }
    phy.Set ("TxGain", DoubleValue (m_txGain));
    phy.Set ("RxGain", DoubleValue (m_rxGain));
    phy.Set ("RxSensitivity", DoubleValue (m_rxSensitivity));

    WifiHelper wifi;
    if (bss == 0 && m_verbose)
      {
        wifi.EnableLogComponents ();
      }
    if (bss > 0 || m_heRate == "ideal")
      {
        wifi.SetRemoteStationManager ("ns3::IdealWifiManager");
      }
    else
      {
        NS_ASSERT (m_nVhtStations == 0);
        uint8_t mcs = std::stoul (m_heRate);
        std::string ctrlRate =
            mcs == 0 ? "OfdmRate6Mbps" : (mcs < 3 ? "OfdmRate12Mbps" : "OfdmRate24Mbps
");
        wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode",
                                      StringValue ("HeMcs" + m_heRate), "ControlMode",
                                      StringValue (ctrlRate));
      }

    if (m_dlAckSeqType == "ACK-SU-FORMAT")
      {
        Config::SetDefault ("ns3::WifiDefaultAckManager::DlMuAckSequenceType",
                            EnumValue (WifiAcknowledgment::DL_MU_BAR_BA_SEQUENCE));
      }
    else if (m_dlAckSeqType == "MU-BAR")
      {
        Config::SetDefault ("ns3::WifiDefaultAckManager::DlMuAckSequenceType",
                            EnumValue (WifiAcknowledgment::DL_MU_TF_MU_BAR));
      }
    else if (m_dlAckSeqType == "AGGR-MU-BAR")
      {
        Config::SetDefault ("ns3::WifiDefaultAckManager::DlMuAckSequenceType",
                            EnumValue (WifiAcknowledgment::DL_MU_AGGREGATE_TF));
      }
    else if (m_dlAckSeqType != "NO-OFDMA")
      {
        NS_ABORT_MSG ("Invalid DL ack sequence type (must be NO-OFDMA, ACK-SU-FORMAT,
MU-BAR or AGGR-MU-BAR)");
      }

    // Select the appropriate OfdmaManager based on the command line argument
    WifiMacHelper mac;
    // New configuration
      if (bss == 0 && m_enableDlOfdma)    // OFDMA only allowed for BSS under test
      {
        std::string schedulerString;
        if (m_scheduler == "rr")
          {
            schedulerString = "Standard";
          }
        else if (m_scheduler == "bellalta")
          {
            schedulerString = "Bellalta";
```

```
          }
        mac.SetMultiUserScheduler ("ns3::RrMultiUserScheduler",
                                   "NStations", UintegerValue (m_maxNRus),
                                   "ForceDlOfdma", BooleanValue (m_forceDlOfdma),
                                   "EnableUlOfdma", BooleanValue (m_enableUlOfdma),
                                   "EnableTxopSharing", BooleanValue (
m_enableTxopSharing),
                                   "EnableBsrp", BooleanValue (m_enableBsrp),
                                   "UlPsduSize", UintegerValue (m_ulPsduSize),
                                   "UseCentral26TonesRus", BooleanValue (
m_useCentral26TonesRus),
                                   "SchedulerLogic", StringValue (schedulerString));
      }
    // if (m_scheduler == "rr" && bss == 0 && m_enableDlOfdma)
    //   {
    //     wifi.SetOfdmaManager (
    //         "ns3::RrOfdmaManager", "NStations", UintegerValue (m_maxNRus), "
ForceDlOfdma",
    //         BooleanValue (m_forceDlOfdma), "EnableUlOfdma", BooleanValue (
m_enableUlOfdma),
    //         "EnableTxopSharing", BooleanValue (m_enableTxopSharing), "EnableBsrp",
    //         BooleanValue (m_enableBsrp), "UlPsduSize", UintegerValue (m_ulPsduSize)
,
    //         "SchedulerLogic", StringValue("Standard"));
    //   }
    // else if (m_scheduler == "bellalta" && bss == 0 && m_enableDlOfdma)
    //   {
    //     wifi.SetOfdmaManager (
    //         "ns3::RrOfdmaManager", "NStations", UintegerValue (m_maxNRus), "
ForceDlOfdma",
    //         BooleanValue (m_forceDlOfdma), "EnableUlOfdma", BooleanValue (
m_enableUlOfdma),
    //         "EnableTxopSharing", BooleanValue (m_enableTxopSharing), "EnableBsrp",
    //         BooleanValue (m_enableBsrp), "UlPsduSize", UintegerValue (m_ulPsduSize)
,
    //         "SchedulerLogic", StringValue("Bellalta"));
    //   }
    // else if (bss == 0 && m_enableDlOfdma)
    //   {
    //     NS_ABORT_MSG ("Unrecognized scheduler: " << m_scheduler);
    //   }

    if (m_enableObssPd)
      {
        wifi.SetObssPdAlgorithm ("ns3::ConstantObssPdAlgorithm", "ObssPdLevelMin",
                                 DoubleValue (m_obssPdThresholdMinBss), "
ObssPdLevelMax",
                                 DoubleValue (m_obssPdThresholdMaxBss), "ObssPdLevel",
                                 DoubleValue (m_obssPdThresholdBss));
      }

    mac.SetType ("ns3::StaWifiMac",
                 "Ssid", SsidValue (Ssid ("non-existing-ssid")));  // prevent stations
from automatically associating
    phy.Set ("TxPowerStart", DoubleValue (m_powerSta));
    phy.Set ("TxPowerEnd", DoubleValue (m_powerSta));
    phy.Set ("CcaEdThreshold", DoubleValue (m_ccaThresholdSta));

    NetDeviceContainer staDevices, apDevice;

    if (bss == 0)
      {
        wifi.SetStandard (WIFI_STANDARD_80211ax_5GHZ);
        staDevices.Add (wifi.Install (phy, mac, heStaNodes));

        wifi.SetStandard (WIFI_STANDARD_80211ac);
        staDevices.Add (wifi.Install (phy, mac, vhtStaNodes));

        m_staDevices.Add (staDevices);
      }
    else
      {
        wifi.SetStandard (m_obssStandard);
```

```
          staDevices = wifi.Install (phy, mac, obssStaNodes);
          m_obssStaDevices.push_back (staDevices);
        }

      wifi.SetStandard (bss == 0 ? WIFI_STANDARD_80211ax_5GHZ : m_obssStandard);
      mac.SetType ("ns3::ApWifiMac",
                   "Ssid", SsidValue (Ssid (m_ssidPrefix + std::to_string (bss))));
      phy.Set ("TxPowerStart", DoubleValue (m_powerAp));
      phy.Set ("TxPowerEnd", DoubleValue (m_powerAp));
      phy.Set ("CcaEdThreshold", DoubleValue (m_ccaThresholdAp));

      apDevice = wifi.Install (phy, mac, apNode);
      m_apDevices.Add (apDevice);

      // The below statements may be simplified in a future HeConfigurationHelper
      if ((m_enableObssPd))
        {
          Ptr<HeConfiguration> heConfiguration =
              DynamicCast<WifiNetDevice> (apDevice.Get (0))->GetHeConfiguration ();
          heConfiguration->SetAttribute ("BssColor", UintegerValue (bss + 1));
        }

      if (m_enablePcap)
        {
          phy.EnablePcap ("STA_pcap", staDevices);
          phy.EnablePcap ("AP_pcap", apDevice);
        }
    }

  int64_t streamNumber = 100;

  // Assign fixed streams to random variables in use
  WifiHelper wifi;
  streamNumber += wifi.AssignStreams (m_apDevices, streamNumber);
  streamNumber += wifi.AssignStreams (m_staDevices, streamNumber);
  for (const auto& devices : m_obssStaDevices)
    {
      streamNumber += wifi.AssignStreams (devices, streamNumber);
    }

  // Configure max A-MSDU size and max A-MPDU size on the AP of the BSS under test
  Ptr<WifiNetDevice> dev = DynamicCast<WifiNetDevice> (m_apDevices.Get (0));
  dev->GetMac ()->SetAttribute ("BE_MaxAmsduSize", UintegerValue (m_beMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("BK_MaxAmsduSize", UintegerValue (m_bkMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("VI_MaxAmsduSize", UintegerValue (m_viMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("VO_MaxAmsduSize", UintegerValue (m_voMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("BE_MaxAmpduSize", UintegerValue (m_beMaxAmpduSize));
  dev->GetMac ()->SetAttribute ("BK_MaxAmpduSize", UintegerValue (m_bkMaxAmpduSize));
  dev->GetMac ()->SetAttribute ("VI_MaxAmpduSize", UintegerValue (m_viMaxAmpduSize));
  dev->GetMac ()->SetAttribute ("VO_MaxAmpduSize", UintegerValue (m_voMaxAmpduSize));
  // XXX Set large beacon generation interval at the AP
  dev->GetMac ()->SetAttribute ("BeaconInterval", TimeValue (NanoSeconds (102400000 *
    20)));
  m_band = dev->GetPhy ()->GetPhyBand ();
  // Configure TXOP Limit and MSDU lifetime on the AP
  PointerValue ptr;
  dev->GetMac ()->GetAttribute ("BE_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_beTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_beMsduLifetime))
    ;
  ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (MilliSeconds
     (m_beMsduLifetime));
  ptr.Get<QosTxop> ()->SetMinCw (m_BeCwMin);
  dev->GetMac ()->GetAttribute ("BK_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_bkTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_bkMsduLifetime))
    ;
  ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (MilliSeconds
     (m_bkMsduLifetime));
  dev->GetMac ()->GetAttribute ("VI_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_viTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_viMsduLifetime))
    ;
```

```
ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (MilliSeconds
   (m_bkMsduLifetime));
dev->GetMac ()->GetAttribute ("VO_Txop", ptr);
ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_voTxopLimit));
ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_voMsduLifetime))
  ;
ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (MilliSeconds
   (m_bkMsduLifetime));

    // Configure max A-MSDU size, max A-MPDU size, TXOP Limit and MSDU lifetime on the
   stations
    // of the BSS under test
    for (uint32_t i = 0; i < m_staNodes.GetN (); i++)
{
  dev = DynamicCast<WifiNetDevice> (m_staDevices.Get (i));
  dev->GetMac ()->SetAttribute ("BE_MaxAmsduSize", UintegerValue (m_beMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("BK_MaxAmsduSize", UintegerValue (m_bkMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("VI_MaxAmsduSize", UintegerValue (m_viMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("VO_MaxAmsduSize", UintegerValue (m_voMaxAmsduSize));
  dev->GetMac ()->SetAttribute ("BE_MaxAmpduSize", UintegerValue (m_beMaxAmpduSize));
  dev->GetMac ()->SetAttribute ("BK_MaxAmpduSize", UintegerValue (m_bkMaxAmpduSize));
  dev->GetMac ()->SetAttribute ("VI_MaxAmpduSize", UintegerValue (m_viMaxAmpduSize));
  dev->GetMac ()->SetAttribute ("VO_MaxAmpduSize", UintegerValue (m_voMaxAmpduSize));
  dev->GetMac ()->GetAttribute ("BE_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_beTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_beMsduLifetime
  ));
  ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (
  MilliSeconds (m_beMsduLifetime));
  ptr.Get<QosTxop> ()->SetMinCw (m_BeCwMin);
  dev->GetMac ()->GetAttribute ("BK_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_bkTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_bkMsduLifetime
  ));
  ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (
  MilliSeconds (m_beMsduLifetime));
  dev->GetMac ()->GetAttribute ("VI_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_viTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_viMsduLifetime
  ));
  ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (
  MilliSeconds (m_beMsduLifetime));
  dev->GetMac ()->GetAttribute ("VO_Txop", ptr);
  ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_voTxopLimit));
  ptr.Get<QosTxop> ()->GetWifiMacQueue ()->SetMaxDelay (MilliSeconds (m_voMsduLifetime
  ));
  ptr.Get<QosTxop> ()->GetBaManager ()->GetRetransmitQueue ()->SetMaxDelay (
  MilliSeconds (m_beMsduLifetime));
  }

// Configure max A-MPDU size and TXOP Limit for APs and stations of the overlapping
  BSSes
for (uint16_t bss = 0; bss < m_nObss; bss++)
  {
    dev = DynamicCast<WifiNetDevice> (m_apDevices.Get (bss + 1));
    dev->GetMac ()->SetAttribute ("BE_MaxAmpduSize", UintegerValue (m_obssMaxAmpduSize
  ));
    dev->GetMac ()->GetAttribute ("BE_Txop", ptr);
    ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_obssTxopLimit));

    for (std::size_t i = 0; i < m_nStationsPerObss; i++)
      {
        dev = DynamicCast<WifiNetDevice> (m_obssStaDevices.at (bss).Get (i));
        dev->GetMac ()->SetAttribute ("BE_MaxAmpduSize", UintegerValue (
  m_obssMaxAmpduSize));
        dev->GetMac ()->GetAttribute ("BE_Txop", ptr);
        ptr.Get<QosTxop> ()->SetTxopLimit (MicroSeconds (m_obssTxopLimit));
      }
  }

// Setting mobility model
for (uint16_t bss = 0; bss < m_nObss + 1; bss++)
  {
```

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

    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
    Vector apPos (0.0, 0.0, 0.0); // initialized to the position of the AP of the BSS
  under test

    if (bss > 0)
      {
        // position of the AP of an overlapping BSS
        apPos = Vector (
            m_apDistances.at (bss - 1) * std::cos (m_apThetas.at (bss - 1) * M_PI /
  180),
            m_apDistances.at (bss - 1) * std::sin (m_apThetas.at (bss - 1) * M_PI /
  180), 0.0);
      }
    positionAlloc->Add (apPos);
    mobility.SetPositionAllocator (positionAlloc);
    mobility.Install (m_apNodes.Get (bss));

    Ptr<UniformDiscPositionAllocator> staPositionAlloc =
      CreateObjectWithAttributes<UniformDiscPositionAllocator> ("rho", DoubleValue (
  m_radius),
                                                                "X", DoubleValue (
  apPos.x),
                                                                "Y", DoubleValue (
  apPos.y));
    streamNumber += staPositionAlloc->AssignStreams (streamNumber);
    mobility.SetPositionAllocator (staPositionAlloc);

    if (bss == 0)
      {
        mobility.Install (m_staNodes);
      }
    else
      {
        mobility.Install (m_obssStaNodes.at (bss - 1));
      }
  }

/* Internet stack */
InternetStackHelper stack;
stack.Install (m_apNodes);
stack.Install (m_staNodes);
for (auto &nodes : m_obssStaNodes)
  {
    stack.Install (nodes);
  }

Ipv4AddressHelper address;
address.SetBase ("10.1.0.0", "255.255.0.0");
m_staInterfaces = address.Assign (m_staDevices);
m_apInterfaces = address.Assign (m_apDevices.Get (0));

for (uint16_t bss = 0; bss < m_nObss; bss++)
  {
    address.SetBase (std::string ("10." + std::to_string (bss + 2) + ".0.0").c_str (),
                     "255.255.0.0");
    m_apInterfaces.Add (address.Assign (m_apDevices.Get (bss + 1)));
    m_obssStaInterfaces.push_back (address.Assign (m_obssStaDevices.at (bss)));
  }

/* Traffic Control layer */
TrafficControlHelper tch;
if (m_queueDisc.compare ("default") != 0)
  {
    // Uninstall the root queue disc on all the devices
    tch.Uninstall (m_apDevices);
    tch.Uninstall (m_staDevices);
    for (auto &devices : m_obssStaDevices)
      {
        tch.Uninstall (devices);
      }
```

```
  }

m_clientApps.resize (m_flows.size ());
m_obssClientApps.resize (m_nObss);

/* Install applications (receiver side) */
for (auto &flow : m_flows)
  {
    std::string socketType =
        (flow.m_l4Proto == Flow::TCP ? "ns3::TcpSocketFactory" : "ns3::
  UdpSocketFactory");
    PacketSinkHelper packetSinkHelper (
        socketType, InetSocketAddress (Ipv4Address::GetAny (), flow.m_dstPort));
    if (flow.m_direction == Flow::DOWNLINK)
      {
        m_sinkApps.Add (packetSinkHelper.Install (m_staNodes.Get (flow.m_stationId -
  1)));
      }
    else
      {
        m_sinkApps.Add (packetSinkHelper.Install (m_apNodes.Get (0)));
      }
  }

m_sinkApps.Stop (
    Seconds (m_warmup + m_simulationTime + 100)); // let the servers be active for a
  long time

m_obssSinkApps.resize (m_nObss);

if (m_obssDlAggregateRate > 0)
  {
    // install receivers on stations of overlapping BSSes
    PacketSinkHelper packetSinkHelper ("ns3::UdpSocketFactory",
                                       InetSocketAddress (Ipv4Address::GetAny (),
  7000));
    for (uint16_t bss = 0; bss < m_nObss; bss++)
      {
        m_obssSinkApps.at (bss).Add (packetSinkHelper.Install (m_obssStaNodes.at (bss)
  ));
      }
  }

if (m_obssUlAggregateRate > 0)
  {
    // install receivers on the APs of overlapping BSSes
    PacketSinkHelper packetSinkHelper ("ns3::UdpSocketFactory",
                                       InetSocketAddress (Ipv4Address::GetAny (),
  7000));
    for (uint16_t bss = 0; bss < m_nObss; bss++)
      {
        m_obssSinkApps.at (bss).Add (packetSinkHelper.Install (m_apNodes.Get (bss + 1)
  ));
      }
  }

for (auto &obssSinkApps : m_obssSinkApps)
  {
    obssSinkApps.Stop (
        Seconds (m_warmup + m_simulationTime + 100)); // let the servers be active for
   a long time
  }

/* Init statistics */
m_obssDlRxStart.assign (m_nObss, 0.0);
m_obssUlRxStart.assign (m_nObss, 0.0);
m_obssDlRxStop.assign (m_nObss, 0.0);
m_obssUlRxStop.assign (m_nObss, 0.0);

// per-AC and pairwise per-AC statistics
m_perAcStats.resize (m_nStations + 1);
m_dlPerStaAcStats.resize (m_nStations);
m_ulPerStaAcStats.resize (m_nStations);
```

```
  m_nSolicitingBasicTriggerFrames.assign (m_nStations, 0);

  for (auto &flow : m_flows)
    {
      if (flow.m_direction == Flow::DOWNLINK || flow.m_l4Proto == Flow::TCP)
        {
          if (flow.m_direction == Flow::DOWNLINK)
            {
              m_dlPerStaAcStats[flow.m_stationId - 1][flow.m_ac] = PairwisePerAcStats ()
    ;
              m_perAcStats[0][flow.m_ac] = PerAcStats ();
            }
          else
            {
              // TCP uplink flow -> BE downlink flow comprising TCP acks
              m_dlPerStaAcStats[flow.m_stationId - 1][AC_BE] = PairwisePerAcStats ();
              m_perAcStats[0][AC_BE] = PerAcStats ();
            }
        }
      if (flow.m_direction == Flow::UPLINK || flow.m_l4Proto == Flow::TCP)
        {
          if (flow.m_direction == Flow::UPLINK)
            {
              m_ulPerStaAcStats[flow.m_stationId - 1][flow.m_ac] = PairwisePerAcStats ()
    ;
              m_perAcStats[flow.m_stationId][flow.m_ac] = PerAcStats ();
            }
          else
            {
              // TCP downlink flow -> BE uplink flow comprising TCP acks
              m_ulPerStaAcStats[flow.m_stationId - 1][AC_BE] = PairwisePerAcStats ();
              m_perAcStats[flow.m_stationId][AC_BE] = PerAcStats ();
            }
        }
    }

  Config::ConnectWithoutContext ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3
    ::StaWifiMac/Assoc",
                                 MakeCallback (&WifiOfdmaExample::EstablishBaAgreement,
    this));

  // populate m_staMacAddressToNodeId map
  for (auto it = m_staDevices.Begin (); it != m_staDevices.End (); it++)
    {
      m_staMacAddressToNodeId[Mac48Address::ConvertFrom ((*it)->GetAddress ())] =
          (*it)->GetNode ()->GetId ();
    }
}

void
WifiOfdmaExample::Run (void)
{
  NS_LOG_FUNCTION (this);

  // Start the setup phase by having the first station associate with the AP
  Simulator::ScheduleNow (&WifiOfdmaExample::StartAssociation, this);

  Simulator::Run ();
}

template <typename T, typename FUNC>
void
WifiOfdmaExample::PrintStatsWithTotal (std::vector<std::map<AcIndex, T>> v, FUNC select,
                                       std::ostream &os, std::string s, std::string s0)
{
  std::size_t i;
  auto ac = m_aciToString.begin ();
  decltype (select (std::declval<T> ())) sum{0}, totalSum{0}, ap{0};

  auto print = [&] (const std::map<AcIndex, T> &x, std::string str = "") {
    auto it = x.find (ac->first);
    if (it != x.end ())
      {
```

```
        os << (str.empty () ? s + std::to_string (i) : s0) << ": " << select (it->second
    ) << " ";
        sum += select (it->second);
      }
    i++;
  };

  os << "-------------------------------" << std::endl;
  for (; ac != m_aciToString.end (); ac++)
    {
      // check if at least one station exchanged traffic of this AC
      if (std::find_if (v.begin (), v.end (), [&ac] (const std::map<AcIndex, T> &map) {
            return map.find (ac->first) != map.end ();
          }) == v.end ())
        {
          continue;
        }

      os << "[" << ac->second << "] ";
      auto it = v.begin ();
      if (!s0.empty ())
        {
          i = 0;
          print (*it, "AP");
          it++;
          ap = sum;
          sum = 0;
        }
      i = 1;
      std::for_each (it, v.end (), print);
      os << std::endl;
      if (!s0.empty ())
        {
          os << "[" << ac->second << "] TotalDL: " << ap << std::endl
             << "[" << ac->second << "] TotalUL: " << sum << std::endl;
        }
      else
        {
          os << "[" << ac->second << "] Total: " << sum << std::endl;
        }
      totalSum += sum;
      totalSum += ap;
      sum = 0;
      ap = 0;
    }
  os << "TOTAL: " << totalSum << std::endl << std::endl;
}

template <typename T, typename FUNC>
void
WifiOfdmaExample::PrintStats (std::vector<std::map<AcIndex, T>> v, FUNC select, std::
    ostream &os,
                              std::string s, std::string s0)
{
  std::size_t i;
  auto ac = m_aciToString.begin ();

  auto print = [&] (const std::map<AcIndex, T> &x, std::string str = "") {
    auto it = x.find (ac->first);
    if (it != x.end ())
      {
        os << (str.empty () ? s + std::to_string (i) : s0) << ": " << select (it->second
    ) << " ";
      }
    i++;
  };

  os << "-------------------------------" << std::endl;
  for (; ac != m_aciToString.end (); ac++)
    {
      // check if at least one station exchanged traffic of this AC
      if (std::find_if (v.begin (), v.end (), [&ac] (const std::map<AcIndex, T> &map) {
            return map.find (ac->first) != map.end ();
```

```
          }) == v.end ())
        {
          continue;
        }

      os << "[" << ac->second << "] ";
      auto it = v.begin ();
      if (!s0.empty ())
        {
          i = 0;
          print (*it, "AP");
          it++;
        }
      i = 1;
      std::for_each (it, v.end (), print);
      os << std::endl;
    }
  os << std::endl;
}

void
WifiOfdmaExample::PrintResults (std::ostream &os)
{
  NS_LOG_FUNCTION (this);

  os << "PER-FLOW statistics" << std::endl << "**********************" << std::endl;
  for (std::size_t i = 0; i < m_flows.size (); i++)
    {
      os << "FLOW " << m_flows[i] << std::endl
         << std::fixed << std::setprecision (3)
         << "Throughput: " << (m_flows[i].m_rxBytes * 8.) / (m_simulationTime * 1e6) <<
    std::endl
         << "Expected data rate: " << m_flows[i].m_dataRate / 1e6 << std::endl
         << "Actual data rate: " << (m_flows[i].m_txBytes * 8.) / (m_simulationTime * 1
    e6)
         << std::endl
         << "Packet loss rate: "
         << static_cast<double> (m_flows[i].m_txPackets + m_flows[i].
    m_packetsRejectedBySocket -
                                 m_flows[i].m_rxPackets) /
                (m_flows[i].m_txPackets + m_flows[i].m_packetsRejectedBySocket)
         << std::endl
         << "Dropped packets (app layer): " << m_flows[i].m_packetsRejectedBySocket <<
    std::endl
         << "Latency: " << m_flows[i].m_latency << std::endl
         << std::endl;
    }

  for (uint16_t bss = 0; bss < m_nObss; bss++)
    {
      os << "OBSS " << bss + 1 << std::endl;
      if (m_obssDlAggregateRate > 0)
        {
          os << "DL Throughput: "
             << ((m_obssDlRxStop[bss] - m_obssDlRxStart[bss]) * 8.) / (m_simulationTime
    * 1e6)
             << std::endl;
        }
      if (m_obssUlAggregateRate > 0)
        {
          os << "UL Throughput: "
             << ((m_obssUlRxStop[bss] - m_obssUlRxStart[bss]) * 8.) / (m_simulationTime
    * 1e6)
             << std::endl;
        }
      os << std::endl;
    }

  /* Per-AC statistics from per-flow statistics */
  std::vector<std::map<AcIndex, double>> dlTput (m_nStations), ulTput (m_nStations);
  std::vector<std::map<AcIndex, Stats<double>>> dlLatency (m_nStations), ulLatency (
    m_nStations);
  std::vector<std::map<AcIndex, uint64_t>> dlSentPkts (m_nStations), ulSentPkts (
```

```
  m_nStations),
    dlRecvPkts (m_nStations), ulRecvPkts (m_nStations);

for (std::size_t i = 0; i < m_flows.size (); i++)
  {
    std::vector<std::map<AcIndex, double>>::iterator tputVecIt;
    std::vector<std::map<AcIndex, Stats<double>>>::iterator latencyVecIt;
    std::vector<std::map<AcIndex, uint64_t>>::iterator sentVecIt, recvVecIt;

    if (m_flows[i].m_direction == Flow::DOWNLINK)
      {
        tputVecIt = std::next (dlTput.begin (), m_flows[i].m_stationId - 1);
        latencyVecIt = std::next (dlLatency.begin (), m_flows[i].m_stationId - 1);
        sentVecIt = std::next (dlSentPkts.begin (), m_flows[i].m_stationId - 1);
        recvVecIt = std::next (dlRecvPkts.begin (), m_flows[i].m_stationId - 1);
      }
    else
      {
        tputVecIt = std::next (ulTput.begin (), m_flows[i].m_stationId - 1);
        latencyVecIt = std::next (ulLatency.begin (), m_flows[i].m_stationId - 1);
        sentVecIt = std::next (ulSentPkts.begin (), m_flows[i].m_stationId - 1);
        recvVecIt = std::next (ulRecvPkts.begin (), m_flows[i].m_stationId - 1);
      }

    auto mapIt = tputVecIt->insert ({m_flows[i].m_ac, 0.0}).first;
    mapIt->second += (m_flows[i].m_rxBytes * 8.) / (m_simulationTime * 1e6);

    auto mapIt2 = latencyVecIt->insert ({m_flows[i].m_ac, 0}).first;
    mapIt2->second += m_flows[i].m_latency;

    auto mapIt3 = sentVecIt->insert ({m_flows[i].m_ac, 0.0}).first;
    mapIt3->second += (m_flows[i].m_txPackets + m_flows[i].m_packetsRejectedBySocket);

    mapIt3 = recvVecIt->insert ({m_flows[i].m_ac, 0.0}).first;
    mapIt3->second += m_flows[i].m_rxPackets;
  }

os << "PAIRWISE PER-AC statistics" << std::endl << "************************" << std::
  endl;

os << "Throughput (Mbps) [DL]" << std::endl;
PrintStatsWithTotal (dlTput, [](const double& t) { return t; }, os, "To_STA_");
os << "Throughput (Mbps) [UL]" << std::endl;
PrintStatsWithTotal (ulTput, [](const double& t) { return t; }, os, "From_STA_");

os << "E2e latency (ms) [DL]" << std::endl;
PrintStatsWithTotal (dlLatency, [](const Stats<double>& t) { return t; }, os, "To_STA_
  ");
os << "E2e latency (ms) [UL]" << std::endl;
PrintStatsWithTotal (ulLatency, [](const Stats<double>& t) { return t; }, os, "
  From_STA_");

os << "Transmitted packets [DL]" << std::endl;
PrintStatsWithTotal (dlSentPkts, [](const uint64_t& t) { return t; }, os, "To_STA_");
os << "Transmitted packets [UL]" << std::endl;
PrintStatsWithTotal (ulSentPkts, [](const uint64_t& t) { return t; }, os, "From_STA_")
  ;

os << "Received packets [DL]" << std::endl;
PrintStatsWithTotal (dlRecvPkts, [](const uint64_t& t) { return t; }, os, "To_STA_");
os << "Received packets [UL]" << std::endl;
PrintStatsWithTotal (ulRecvPkts, [](const uint64_t& t) { return t; }, os, "From_STA_")
  ;

/* EXPIRED MSDUs */
os << "MSDUs expired at the AP [DL]" << std::endl;
PrintStatsWithTotal (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  expired; },
                     os, "To_STA_");
os << "MSDUs expired at the STAs [UL]" << std::endl;
PrintStatsWithTotal (m_ulPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  expired; },
                     os, "From_STA_");
```

```
/* REJECTED MSDUs */
os << "MSDUs rejected at the AP [DL]" << std::endl;
PrintStatsWithTotal (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  rejected; },
                     os, "To_STA_");
os << "MSDUs rejected at the STAs [UL]" << std::endl;
PrintStatsWithTotal (m_ulPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  rejected; },
                     os, "From_STA_");

/* FAILED MPDUs */
os << "TX failures at the AP [DL]" << std::endl;
PrintStatsWithTotal (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  failed; },
                     os, "To_STA_");
os << "TX failures at the STAs [UL]" << std::endl;
PrintStatsWithTotal (m_ulPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  failed; },
                     os, "From_STA_");

/* A-MPDU SIZE */
os << "A-MPDU size (min/avg/max/count) [DL]" << std::endl;
PrintStats (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.ampduSize;
  },
            os, "To_STA_");
os << "A-MPDU size (min/avg/max/count) [UL]" << std::endl;
PrintStats (m_ulPerStaAcStats, [](const PairwisePerAcStats& s) { return s.ampduSize;
  },
            os, "From_STA_");

/* A-MPDU DURATION TO DL/UL MU PPDU DURATION RATIO */
os << "A-MPDU duration to DL MU PPDU duration ratio (min/avg/max/count) [DL]" << std::
  endl;
PrintStats (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.ampduRatio;
  },
            os, "To_STA_");
os << "A-MPDU duration to Basic TF's UL Length ratio (min/avg/max/count) [UL]" << std
  ::endl;
PrintStats (m_ulPerStaAcStats, [](const PairwisePerAcStats& s) { return s.ampduRatio;
  },
            os, "From_STA_");

/* L2 LATENCY */
os << "L2 latency (percentiles[count]) [DL]" << std::endl;
PrintStatsWithTotal (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  l2Latency; },
                     os, "To_STA_");
os << "L2 latency (percentiles[count]) [UL]" << std::endl;
PrintStatsWithTotal (m_ulPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  l2Latency; },
                     os, "From_STA_");

/* HEAD-OF-LINE DELAY */
os << "Pairwise Head-of-Line delay (percentiles[count]) at the AP [DL]" << std::endl;
PrintStatsWithTotal (m_dlPerStaAcStats, [](const PairwisePerAcStats& s) { return s.
  pairwiseHol; },
                     os, "To_STA_");

os << "PER-AC statistics" << std::endl << "************************" << std::endl;

os << "Aggregate Head-of-Line delay (percentiles[count])" << std::endl;
PrintStats (m_perAcStats, [](const PerAcStats& s) { return s.aggregateHoL; },
            os, "STA_", "AP");

/* QUEUE DISC SOJOURN TIME AND PACKET DROPPED  */
os << "Sojourn time in the queue disc associated with each AC" << std::endl;
PrintStatsWithTotal (m_perAcStats, [](const PerAcStats& s) { return s.
  queueDiscSojournTime; },
                     os, "STA_", "AP");

os << "Packets dropped by the queue disc associated with each AC" << std::endl;
PrintStatsWithTotal (m_perAcStats, [](const PerAcStats& s) { return s.
```

```
    droppedByQueueDisc; },
                      os, "STA_", "AP");

  /* TXOP DURATION */
  os << "TXOP duration (percentiles[count])" << std::endl;
  PrintStats (m_perAcStats, [](const PerAcStats& s) { return s.txopDuration; },
              os, "STA_", "AP");

  os << "PER-STA statistics" << std::endl << "**********************" << std::endl;

  os << "DL MU PPDU completeness (percentiles[count])" << std::endl
     << "-------------------------------" << std::endl
     << m_dlMuCompleteness << std::endl
     << std::endl;

  os << "HE TB PPDU completeness (percentiles[count])" << std::endl
     << "-------------------------------" << std::endl
     << m_heTbCompleteness << std::endl
     << std::endl;

  os << "DL MU PPDU duration (percentiles[count])" << std::endl
     << "-------------------------------" << std::endl
     << m_dlMuPpduDuration << std::endl
     << std::endl;

  os << "HE TB PPDU duration (percentiles[count])" << std::endl
     << "-------------------------------" << std::endl
     << m_heTbPpduDuration << std::endl
     << std::endl;

  os << "Basic Trigger Frames (failed/sent)" << std::endl
     << "-------------------------------" << std::endl
     << "(" << m_nFailedBasicTriggerFrames << ", " << m_nBasicTriggerFramesSent << ")"
    << std::endl
     << std::endl;

  os << "BSRP Trigger Frames (failed/sent)" << std::endl
     << "-------------------------------" << std::endl
     << "(" << m_nFailedBsrpTriggerFrames << ", " << m_nBsrpTriggerFramesSent << ")" <<
    std::endl
     << std::endl;

  os << "Unresponded Basic TFs ratio" << std::endl << "-------------------------------"
    << std::endl;
  for (uint16_t i = 0; i < m_nStations; i++)
    {
      uint64_t heTbPPduCount = 0; // for i-th STA
      for (auto &acStats : m_ulPerStaAcStats.at (i))
        {
          heTbPPduCount += acStats.second.ampduRatio.m_samples.size ();
        }
      os << "STA_" << i + 1 << ": "
         << static_cast<double> (m_nSolicitingBasicTriggerFrames.at (i) - heTbPPduCount)
     /
              m_nSolicitingBasicTriggerFrames.at (i)
         << " ";
    }
  os << std::endl << std::endl;
}

void
WifiOfdmaExample::StartAssociation (void)
{
  NS_LOG_FUNCTION (this << m_currentSta);
  NS_ASSERT (m_currentSta < m_nStations + m_nObss * m_nStationsPerObss);

  Ptr<WifiNetDevice> dev;
  uint16_t bss = 0;

  if (m_currentSta < m_nStations)
    {
      dev = DynamicCast<WifiNetDevice> (m_staDevices.Get (m_currentSta));
    }
```

```
  else
    {
      auto pair = StationIdToBssIndexPair (m_currentSta);
      bss = pair.first;
      dev = DynamicCast<WifiNetDevice> (m_obssStaDevices.at (bss - 1).Get (pair.second))
    ;
    }

  NS_ASSERT (dev != 0);
  // this will lead the station to associate with the AP
  dev->GetMac ()->SetSsid (Ssid (m_ssidPrefix + std::to_string (bss)));
}

void
WifiOfdmaExample::EstablishBaAgreement (Mac48Address bssid)
{
  NS_LOG_FUNCTION (this << bssid << m_currentSta);

  // Now that the current station is associated with the AP, let's trigger the creation
  // of an entry in the ARP cache (of both the AP and the STA) and the establishment of
  // Block Ack agreements between the AP and the STA (and viceversa) for the relevant
  // TIDs. This is done by having the AP send 4 ICMP Echo Requests to the STA
  uint16_t pingInterval = 50; // milliseconds

  std::map<AcIndex, uint8_t> acTos = {{AC_BE, 0x00 /* CS0 */},
                                      {AC_BK, 0x28 /* AF11 */},
                                      {AC_VI, 0xb8 /* EF */},
                                      {AC_VO, 0xc0 /* CS7 */}};
  Ipv4Address staAddress;
  Ptr<Node> apNode;
  uint16_t bss = 0;

  if (m_currentSta < m_nStations)
    {
      staAddress = m_staInterfaces.GetAddress (m_currentSta);
      apNode = m_apNodes.Get (0);
    }
  else
    {
      auto pair = StationIdToBssIndexPair (m_currentSta);
      bss = pair.first;
      staAddress = m_obssStaInterfaces.at (bss - 1).GetAddress (pair.second);
      apNode = m_apNodes.Get (bss);
    }

  V4PingHelper ping (staAddress);
  void (*addPingApp)(V4PingHelper, Ptr<Node>);  // function pointer
  addPingApp = [](V4PingHelper helper, Ptr<Node> node)
               {
                 ApplicationContainer apps = helper.Install (node);
                 apps.Stop (MilliSeconds (10)); // send just one Echo Request
               };

  if (m_verbose)
    {
      ping.SetAttribute ("Verbose", BooleanValue (true));
    }
  for (auto &ac : acTos)
    {
      // ping.SetAttribute ("Tos", UintegerValue (ac.second));
      Simulator::Schedule (MilliSeconds (pingInterval * static_cast<uint16_t> (ac.first)
    ),
                           addPingApp, ping, apNode);
    }

  // set the duration of the "On" interval to zero for all client applications so that,
  // once started, they will stay quiet (i.e., no packet is transmitted) and wake at the
  // end of every "Off" interval to check if the "On" period has become non-zero.
  // The start of all the client applications is aligned to an integer multiple of
  // pingInterval (also referred to as a "slot"),

  // convert pingInterval to time units
  int64_t slotUnits = pingInterval * std::pow (10, (Time::GetResolution () - Time::MS) *
```

```
    3);
// the start time is 4 pingIntervals from now and aligned to the next slot boundary
int64_t startTime =
    ((Simulator::Now ().GetTimeStep () + 4 * slotUnits) / slotUnits + 1) * slotUnits;
Time startDelay = TimeStep (startTime) - Simulator::Now ();
NS_ASSERT (startDelay.IsStrictlyPositive ());

std::size_t i = 0;
for (auto &flow : m_flows)
  {
    // Install a client application for each flow involving the current station.
    // In case of TCP traffic, this will trigger the establishment of a TCP connection
  .
    if (flow.m_stationId == m_currentSta + 1)
      {
        std::string socketType =
            (flow.m_l4Proto == Flow::TCP ? "ns3::TcpSocketFactory" : "ns3::
  UdpSocketFactory");
        OnOffHelper client (socketType, Ipv4Address::GetAny ());
        client.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[
  Constant=0]"));
        client.SetAttribute ("OffTime",
                             StringValue ("ns3::ConstantRandomVariable[Constant=" +
                                          std::to_string (pingInterval / 1000.) + "]")
  );
        client.SetAttribute ("DataRate", DataRateValue (DataRate (flow.m_dataRate)));
        client.SetAttribute ("PacketSize", UintegerValue (flow.m_payloadSize));

        Ipv4Address destAddress;
        Ptr<Node> srcNode;
        if (flow.m_direction == Flow::DOWNLINK)
          {
            destAddress = m_staInterfaces.GetAddress (m_currentSta);
            srcNode = m_apNodes.Get (0);
          }
        else
          {
            destAddress = m_apInterfaces.GetAddress (0);
            srcNode = m_staNodes.Get (m_currentSta);
          }
        InetSocketAddress dest (destAddress, flow.m_dstPort);
        dest.SetTos (acTos.at (flow.m_ac));
        client.SetAttribute ("Remote", AddressValue (dest));

        Simulator::Schedule (startDelay, &WifiOfdmaExample::StartClient, this, client,
   i,
                             srcNode);
      }
    i++;
  }

// Install client applications in the OBSSes
if (m_currentSta >= m_nStations && m_obssDlAggregateRate > 0)
  {
    OnOffHelper client ("ns3::UdpSocketFactory", Ipv4Address::GetAny ());
    client.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant
  =0]"));
    client.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant
  ="
                                                 + std::to_string (pingInterval /
  1000.) + "]"));
    client.SetAttribute ("DataRate", DataRateValue (DataRate (m_obssDlAggregateRate *
  1e6 / m_nStationsPerObss)));
    client.SetAttribute ("PacketSize", UintegerValue (m_obssDlPayload));
    client.SetAttribute ("Remote", AddressValue (InetSocketAddress (staAddress, 7000))
  );

    Simulator::Schedule (startDelay, &WifiOfdmaExample::StartObssClient, this, client,
   bss,
                         apNode);
  }

if (m_currentSta >= m_nStations && m_obssUlAggregateRate > 0)
```

```
    {
      OnOffHelper client ("ns3::UdpSocketFactory", Ipv4Address::GetAny ());
      client.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant
    =0]"));
      client.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant
    ="
                                                   + std::to_string (pingInterval /
    1000.) + "]"));
      client.SetAttribute ("DataRate", DataRateValue (DataRate (m_obssUlAggregateRate *
    1e6 / m_nStationsPerObss)));
      client.SetAttribute ("PacketSize", UintegerValue (m_obssUlPayload));
      client.SetAttribute ("Remote", AddressValue (InetSocketAddress (m_apInterfaces.
    GetAddress (bss),
                                                                       7000)));

      auto pair = StationIdToBssIndexPair (m_currentSta);

      Simulator::Schedule (startDelay, &WifiOfdmaExample::StartObssClient, this, client,
     bss,
                           m_obssStaNodes.at (bss - 1).Get (pair.second));
    }

  // continue with the next station, if any is remaining
  if (++m_currentSta < m_nStations + m_nObss * m_nStationsPerObss)
    {
      Simulator::Schedule (startDelay + MilliSeconds (pingInterval),
                           &WifiOfdmaExample::StartAssociation, this);
    }
  else
    {
      // call DelayStart (which changes the duration of the last "Off" period) 1 ms
      // before a slot boundary
      Simulator::Schedule (startDelay + MilliSeconds (pingInterval - 1),
                           &WifiOfdmaExample::DelayStart, this);
    }
}

void
WifiOfdmaExample::StartClient (OnOffHelper client, std::size_t i, Ptr<Node> node)
{
  NS_LOG_FUNCTION (this << node->GetId ());

  m_clientApps[i] = client.Install (node).Get (0);
  m_clientApps[i]->SetStopTime (
      Seconds (m_warmup + m_simulationTime + 100)); // let clients be active for a long
    time
}

void
WifiOfdmaExample::StartObssClient (OnOffHelper client, uint16_t bss, Ptr<Node> node)
{
  NS_LOG_FUNCTION (this << node->GetId ());

  Ptr<Application> app = client.Install (node).Get (0);
  app->SetStopTime (
      Seconds (m_warmup + m_simulationTime + 100)); // let clients be active for a long
    time
  m_obssClientApps.at (bss - 1).Add (app);
}

void
WifiOfdmaExample::DelayStart (void)
{
  NS_LOG_FUNCTION (this);

  // OnOffApplication clients send the first packet an inter-departure time after the
  // beginning of the first "On" period. Compute the maximum IDT among all the flows
  double maxIdt = 0.; // seconds
  for (auto &flow : m_flows)
    {
      maxIdt = std::max (maxIdt, (flow.m_payloadSize * 8 / flow.m_dataRate));
    }
  if (m_nStationsPerObss > 0 && m_obssDlAggregateRate > 0)
```

```
  {
    maxIdt = std::max (
        maxIdt, (m_obssDlPayload * 8 / (m_obssDlAggregateRate * 1e6 /
  m_nStationsPerObss)));
  }
if (m_nStationsPerObss > 0 && m_obssUlAggregateRate > 0)
  {
    maxIdt = std::max (
        maxIdt, (m_obssUlPayload * 8 / (m_obssUlAggregateRate * 1e6 /
  m_nStationsPerObss)));
  }

// for each client application 'c', the last "Off" period lasts maxIdt + startDelay(c)
   - IDT(c)
// so that every client application transmits the first packet after maxIdt +
  startDelay(c)
Ptr<UniformRandomVariable> startDelayRV;
startDelayRV = CreateObjectWithAttributes<UniformRandomVariable> ("Min", DoubleValue
  (0.0), "Max",
                                                                    DoubleValue (
  m_startInterval));
UintegerValue pktSize;
DataRateValue cbrRate;
bool found;
double offTime;

for (auto &clientApp : m_clientApps)
  {
    found = clientApp->GetAttributeFailSafe ("PacketSize", pktSize);
    NS_ABORT_IF (!found);
    found = clientApp->GetAttributeFailSafe ("DataRate", cbrRate);
    NS_ABORT_IF (!found);
    offTime = maxIdt + (startDelayRV->GetValue () / 1000.) -
              (pktSize.Get () * 8. / cbrRate.Get ().GetBitRate ()) + 50e-9;
    // we add 50 nanoseconds to offTime to prevent it from being zero, which
    // happens if m_startInterval is zero and all the flows have the same IDT.
    // In such a case, at the next slot boundary both the "On" and the "Off"
    // periods would be zero and this would cause an infinite loop
    NS_ASSERT (offTime > 0);

    std::stringstream ss;
    ss << "ns3::ConstantRandomVariable[Constant=" << std::fixed << std::setprecision
  (10)
       << offTime << "]";
    clientApp->SetAttribute ("OffTime", StringValue (ss.str ()));
  }

for (auto &obssClientApp : m_obssClientApps)
  {
    for (auto appIt = obssClientApp.Begin (); appIt != obssClientApp.End (); appIt++)
      {
        found = (*appIt)->GetAttributeFailSafe ("PacketSize", pktSize);
        NS_ABORT_IF (!found);
        found = (*appIt)->GetAttributeFailSafe ("DataRate", cbrRate);
        NS_ABORT_IF (!found);
        offTime = maxIdt + (startDelayRV->GetValue () / 1000.) -
                  (pktSize.Get () * 8. / cbrRate.Get ().GetBitRate ()) + 50e-9;
        NS_ASSERT (offTime > 0);

        std::stringstream ss;
        ss << "ns3::ConstantRandomVariable[Constant=" << std::fixed << std::
  setprecision (10)
           << offTime << "]";
        (*appIt)->SetAttribute ("OffTime", StringValue (ss.str ()));
      }
  }

// StartTraffic sets the "Off" time to zero and the "On" time to the simulation
// duration. Therefore, it must be called after the next slot boundary (when
// the last "Off" period must be scheduled) and before the last "Off" period
// expires. The next slot boundary is in one millisecond and the minimum
// duration of the last "Off" period is 50 nanoseconds.
Simulator::Schedule (MilliSeconds (1) + NanoSeconds (30), &WifiOfdmaExample::
```

```
    StartTraffic, this);
}

void
WifiOfdmaExample::StartTraffic (void)
{
  NS_LOG_FUNCTION (this);

  std::string onTime = std::to_string (m_warmup + m_simulationTime);

  for (auto &clientApp : m_clientApps)
    {
      clientApp->SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[
    Constant="
                                                      + onTime + "]"));
      clientApp->SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[
    Constant=0]"));
    }

  for (auto &obssClientApp : m_obssClientApps)
    {
      for (auto appIt = obssClientApp.Begin (); appIt != obssClientApp.End (); appIt++)
        {
          (*appIt)->SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[
    Constant="
                                                         + onTime + "]"));
          (*appIt)->SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[
    Constant=0]"));
        }
    }

  Simulator::Schedule (Seconds (m_warmup), &WifiOfdmaExample::StartStatistics, this);
}

void
WifiOfdmaExample::StartStatistics (void)
{
  NS_LOG_FUNCTION (this);

  std::cout << "Starting statistics at " << Simulator::Now ().GetMicroSeconds() << std::
    endl;
  /* Connect traces on all the stations (including the AP) and for all the ACs */
  NetDeviceContainer devices = m_staDevices;
  devices.Add (m_apDevices.Get (0));

  if (m_verbose)
    {
      Config::Connect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/State/State",
                       MakeCallback (&WifiOfdmaExample::NotifyStateChange, this));
    }

  for (auto devIt = devices.Begin (); devIt != devices.End (); devIt++)
    {
      Ptr<WifiNetDevice> dev = DynamicCast<WifiNetDevice> (*devIt);
      Ptr<QosTxop> qosTxop;

      for (auto &ac : m_aciToString)
        {
          // get the EDCA on the station for the given AC
          PointerValue ptr;
          dev->GetMac ()->GetAttribute (std::string (ac.second + "_Txop"), ptr);
          qosTxop = ptr.Get<QosTxop> ();
          // trace expired MSDUs
          qosTxop->GetWifiMacQueue ()->TraceConnectWithoutContext ("Expired",
            MakeCallback (&WifiOfdmaExample::NotifyMsduExpired, this));
          qosTxop->GetBaManager ()->GetRetransmitQueue ()->TraceConnectWithoutContext ("
    Expired",
            MakeCallback (&WifiOfdmaExample::NotifyMsduExpired, this));
          // trace rejected MSDUs
          qosTxop->GetWifiMacQueue ()->TraceConnectWithoutContext ("DropBeforeEnqueue",
            MakeCallback (&WifiOfdmaExample::NotifyMsduRejected, this));
          // trace packets enqueued in the EDCA queue
          qosTxop->GetWifiMacQueue ()->TraceConnectWithoutContext ("Enqueue",
```

```
          MakeCallback (&WifiOfdmaExample::NotifyEdcaEnqueue, this));
        // trace MSDUs dequeued from the EDCA queue
        qosTxop->GetWifiMacQueue ()->TraceConnectWithoutContext ("Dequeue",
          MakeCallback (&WifiOfdmaExample::NotifyMsduDequeuedFromEdcaQueue, this)
          .Bind (qosTxop->GetWifiMacQueue ()->GetMaxDelay ()));
        // trace TXOP duration
        qosTxop->TraceConnectWithoutContext ("TxopTrace",
          MakeCallback (&WifiOfdmaExample::NotifyTxopDuration, this)
          .TwoBind (MacAddressToNodeId (Mac48Address::ConvertFrom (dev->GetAddress ())
), ac.first));
      }

    // trace TX failures
    DynamicCast<RegularWifiMac> (dev->GetMac ())->TraceConnectWithoutContext ("
NAckedMpdu", MakeCallback (&WifiOfdmaExample::NotifyTxFailed, this));
    // trace PSDUs forwarded down to the PHY
    dev->GetPhy ()->TraceConnectWithoutContext ("PhyTxPsduBegin", MakeCallback (&
WifiOfdmaExample::NotifyPsduForwardedDown, this));
    // trace packets forwarded up by the MAC layer
    dev->GetMac ()->TraceConnectWithoutContext ("MacRx", MakeCallback (&
WifiOfdmaExample::NotifyMacForwardUp, this));
  }

for (std::size_t i = 0; i < m_flows.size (); i++)
  {
    Ptr<OnOffApplication> sender = DynamicCast<OnOffApplication> (m_clientApps[i]);
    NS_ABORT_MSG_IF (sender == 0, "Not an OnOffApplication?");
    sender->TraceConnectWithoutContext ("Tx", MakeCallback (&WifiOfdmaExample::
NotifyAppTx, this)
                                            .Bind (i));
    // m_flows[i].m_packetsRejectedBySocket = sender->GetTotalPacketsFailed ();

    Ptr<PacketSink> sink = DynamicCast<PacketSink> (m_sinkApps.Get (i));
    NS_ABORT_MSG_IF (sink == 0, "Not a PacketSink?");
    sink->TraceConnectWithoutContext ("Rx", MakeCallback (&WifiOfdmaExample::
NotifyAppRx, this)
                                        .Bind (i));
    m_flows[i].m_prevRxBytes = sink->GetTotalRx ();
  }

for (std::size_t i = 0; i <= m_nStations; i++)
  {
    Ptr<Node> node = (i == 0 ? m_apNodes.Get (0) : m_staNodes.Get (i - 1));
    Ptr<NetDevice> device = (i == 0 ? m_apDevices.Get (0) : m_staDevices.Get (i - 1));
    Ptr<TrafficControlLayer> tc = node->GetObject<TrafficControlLayer> ();
    NS_ABORT_MSG_IF (tc == 0, "No TrafficControlLayer object aggregated to node");

    Ptr<QueueDisc> qdisc = tc->GetRootQueueDiscOnDevice (device);
    if (qdisc != 0 && qdisc->GetNQueueDiscClasses () == 4)
      {
        Ptr<QueueDisc> child;
        for (auto &acStats : m_perAcStats.at (i))
          {
            child = qdisc->GetQueueDiscClass (static_cast<std::size_t> (acStats.first)
)
                        ->GetQueueDisc ();
            acStats.second.droppedByQueueDisc = child->GetStats ().
nTotalDroppedPackets;
            child->TraceConnectWithoutContext ("SojournTime",
                                               MakeCallback (&WifiOfdmaExample::
NotifySojournTime, this)
                                               .TwoBind (i, acStats.first));
            child->TraceConnectWithoutContext ("DropAfterDequeue",
                                               MakeCallback (&WifiOfdmaExample::
NotifyDropAfterDequeue, this)
                                               .TwoBind (i, acStats.first));
          }
      }
  }

if (m_obssDlAggregateRate > 0)
  {
    for (uint16_t bss = 0; bss < m_nObss; bss++)
```

```
        {
          for (std::size_t i = 0; i < m_nStationsPerObss; i++)
            {
              m_obssDlRxStart.at (bss) +=
                  DynamicCast<PacketSink> (m_obssSinkApps.at (bss).Get (i))->GetTotalRx
    ();
            }
        }
    }

  if (m_obssUlAggregateRate > 0)
    {
      for (uint16_t bss = 0; bss < m_nObss; bss++)
        {
          uint32_t last = m_obssSinkApps.at (bss).GetN () - 1;
          m_obssUlRxStart.at (bss) =
              DynamicCast<PacketSink> (m_obssSinkApps.at (bss).Get (last))->GetTotalRx
    ();
        }
    }

  Simulator::Schedule (Seconds (m_simulationTime / m_nIntervals),
                       &WifiOfdmaExample::StoreCumulativeRxBytes, this);
  Simulator::Schedule (Seconds (m_simulationTime), &WifiOfdmaExample::StopStatistics,
    this);
}

void
WifiOfdmaExample::StoreCumulativeRxBytes (void)
{
#if 0
  Ptr<WifiMacQueue> queue = DynamicCast<RegularWifiMac> (DynamicCast<WifiNetDevice> (
    m_apDevices.Get (0))->GetMac ())->GetQosTxop (AC_BE)->GetWifiMacQueue ();
  for (auto it = queue->QueuedPacketsBegin (); it != queue->QueuedPacketsEnd (); it++)
    {
      std::cout << "STA " << it->first.first << " : " << it->second << " MSDUs" << std::
    endl;
    }
#endif
  std::map<AcIndex, double> dlTput, ulTput;
  std::map<AcIndex, double>::iterator mapIt;

  for (std::size_t i = 0; i < m_flows.size (); i++)
    {
      Ptr<PacketSink> sink = DynamicCast<PacketSink> (m_sinkApps.Get (i));
      NS_ABORT_MSG_IF (sink == 0, "Not a PacketSink?");

      if (m_flows[i].m_direction == Flow::DOWNLINK)
        {
          mapIt = dlTput.insert ({m_flows[i].m_ac, 0.0}).first;
        }
      else
        {
          mapIt = ulTput.insert ({m_flows[i].m_ac, 0.0}).first;
        }
      mapIt->second += (sink->GetTotalRx () - m_flows[i].m_prevRxBytes) * 8. /
                       (m_simulationTime / m_nIntervals * 1e6);
      m_flows[i].m_prevRxBytes = sink->GetTotalRx ();
    }

  std::cout << "Per-AC throughput in the last time interval (time: " << Simulator::Now
    () << ")"
            << std::endl << "DOWNLINK" << std::endl;
  for (auto& acMap : dlTput)
    {
      std::cout << "[" << m_aciToString.at (acMap.first) << "] " << acMap.second << std
    ::endl;
    }
  std::cout << "UPLINK" << std::endl;
  for (auto& acMap : ulTput)
    {
      std::cout << "[" << m_aciToString.at (acMap.first) << "] " << acMap.second << std
    ::endl;
```

```
    }
  std::cout << std::endl;

  if (++m_elapsedIntervals < m_nIntervals)
    {
      Simulator::Schedule (Seconds (m_simulationTime / m_nIntervals),
                           &WifiOfdmaExample::StoreCumulativeRxBytes, this);
    }
}

void
WifiOfdmaExample::StopStatistics (void)
{
  NS_LOG_FUNCTION (this);

  std::cout << "Stopping statistics at " << Simulator::Now ().GetMicroSeconds () << std
    ::endl;

  if (m_verbose)
    {
      Config::Disconnect ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/State/State"
    ,
                          MakeCallback (&WifiOfdmaExample::NotifyStateChange, this));
    }

  /* Disconnect traces on all the stations (including the AP) and for all the ACs */
  NetDeviceContainer devices = m_staDevices;
  devices.Add (m_apDevices.Get (0));

  for (auto devIt = devices.Begin (); devIt != devices.End (); devIt++)
    {
      Ptr<WifiNetDevice> dev = DynamicCast<WifiNetDevice> (*devIt);
      Ptr<QosTxop> qosTxop;

      for (auto &ac : m_aciToString)
        {
          // get the EDCA on the station for the given AC
          PointerValue ptr;
          dev->GetMac ()->GetAttribute (std::string (ac.second + "_Txop"), ptr);
          qosTxop = ptr.Get<QosTxop> ();
          // stop tracing expired MSDUs
          qosTxop->GetWifiMacQueue ()->TraceDisconnectWithoutContext ("Expired",
            MakeCallback (&WifiOfdmaExample::NotifyMsduExpired, this));
          qosTxop->GetBaManager ()->GetRetransmitQueue ()->TraceDisconnectWithoutContext
     ("Expired",
            MakeCallback (&WifiOfdmaExample::NotifyMsduExpired, this));
          // stop tracing rejected MSDUs
          qosTxop->GetWifiMacQueue ()->TraceDisconnectWithoutContext ("DropBeforeEnqueue
    ",
            MakeCallback (&WifiOfdmaExample::NotifyMsduRejected, this));
          // stop tracing packets enqueued in the EDCA queue
          qosTxop->GetWifiMacQueue ()->TraceDisconnectWithoutContext ("Enqueue",
            MakeCallback (&WifiOfdmaExample::NotifyEdcaEnqueue, this));
          // stop tracing MSDUs dequeued from the EDCA queue
          qosTxop->GetWifiMacQueue ()->TraceDisconnectWithoutContext ("Dequeue",
            MakeCallback (&WifiOfdmaExample::NotifyMsduDequeuedFromEdcaQueue, this)
            .Bind (qosTxop->GetWifiMacQueue ()->GetMaxDelay ()));
          // stop tracing TXOP duration
          qosTxop->TraceDisconnectWithoutContext ("TxopTrace",
            MakeCallback (&WifiOfdmaExample::NotifyTxopDuration, this)
            .TwoBind (MacAddressToNodeId (Mac48Address::ConvertFrom (dev->GetAddress ())
    ), ac.first));
        }

      // stop tracing TX failures
      DynamicCast<RegularWifiMac> (dev->GetMac ())->TraceDisconnectWithoutContext ("
    NAckedMpdu", MakeCallback (&WifiOfdmaExample::NotifyTxFailed, this));
      // stop tracing PSDUs forwarded down to the PHY
      dev->GetPhy ()->TraceDisconnectWithoutContext ("PhyTxPsduBegin", MakeCallback (&
    WifiOfdmaExample::NotifyPsduForwardedDown, this));
      // stop tracing packets forwarded up by the MAC layer
      dev->GetMac ()->TraceDisconnectWithoutContext ("MacRx", MakeCallback (&
    WifiOfdmaExample::NotifyMacForwardUp, this));
```

```
  }

for (std::size_t i = 0; i < m_flows.size (); i++)
  {
    Ptr<OnOffApplication> sender = DynamicCast<OnOffApplication> (m_clientApps[i]);
    NS_ABORT_MSG_IF (sender == 0, "Not an OnOffApplication?");
    sender->TraceDisconnectWithoutContext ("Tx", MakeCallback (&WifiOfdmaExample::
  NotifyAppTx, this)
                                                    .Bind (i));
    // m_flows[i].m_packetsRejectedBySocket = sender->GetTotalPacketsFailed ()
    //                                        - m_flows[i].m_packetsRejectedBySocket;

    Ptr<PacketSink> sink = DynamicCast<PacketSink> (m_sinkApps.Get (i));
    NS_ABORT_MSG_IF (sink == 0, "Not a PacketSink?");
    sink->TraceDisconnectWithoutContext ("Rx", MakeCallback (&WifiOfdmaExample::
  NotifyAppRx, this)
                                                  .Bind (i));
  }

for (std::size_t i = 0; i <= m_nStations; i++)
  {
    Ptr<Node> node = (i == 0 ? m_apNodes.Get (0) : m_staNodes.Get (i - 1));
    Ptr<NetDevice> device = (i == 0 ? m_apDevices.Get (0) : m_staDevices.Get (i - 1));
    Ptr<TrafficControlLayer> tc = node->GetObject<TrafficControlLayer> ();
    NS_ABORT_MSG_IF (tc == 0, "No TrafficControlLayer object aggregated to node");

    Ptr<QueueDisc> qdisc = tc->GetRootQueueDiscOnDevice (device);
    if (qdisc != 0 && qdisc->GetNQueueDiscClasses () == 4)
      {
        Ptr<QueueDisc> child;
        for (auto &acStats : m_perAcStats.at (i))
          {
            child = qdisc->GetQueueDiscClass (static_cast<std::size_t> (acStats.first)
  )
                        ->GetQueueDisc ();
            acStats.second.droppedByQueueDisc =
                child->GetStats ().nTotalDroppedPackets - acStats.second.
  droppedByQueueDisc;
          }
      }
  }

if (m_obssDlAggregateRate > 0)
  {
    for (uint16_t bss = 0; bss < m_nObss; bss++)
      {
        for (std::size_t i = 0; i < m_nStationsPerObss; i++)
          {
            m_obssDlRxStop.at (bss) +=
                DynamicCast<PacketSink> (m_obssSinkApps.at (bss).Get (i))->GetTotalRx
  ();
          }
      }
  }

if (m_obssUlAggregateRate > 0)
  {
    for (uint16_t bss = 0; bss < m_nObss; bss++)
      {
        uint32_t last = m_obssSinkApps.at (bss).GetN () - 1;
        m_obssUlRxStop.at (bss) =
            DynamicCast<PacketSink> (m_obssSinkApps.at (bss).Get (last))->GetTotalRx
  ();
      }
  }

// (Brutally) stop client applications
for (auto &clientApp : m_clientApps)
  {
    clientApp->Dispose ();
  }
for (auto &obssClientApps : m_obssClientApps)
  {
```

```
      for (auto appIt = obssClientApps.Begin (); appIt != obssClientApps.End (); appIt
    ++)
        {
          (*appIt)->Dispose ();
        }
    }

  // we are done
  Simulator::Stop ();
}

uint32_t
WifiOfdmaExample::MacAddressToNodeId (Mac48Address address)
{
  if (m_apDevices.Get (0)->GetAddress () == address)
    {
      return m_apNodes.Get (0)->GetId ();
    }

  auto it = m_staMacAddressToNodeId.find (address);
  if (it != m_staMacAddressToNodeId.end ())
    {
      return it->second;
    }

  NS_ABORT_MSG ("Found no node having MAC address " << address);
}

std::pair<std::map<AcIndex, WifiOfdmaExample::PairwisePerAcStats>::iterator, bool>
WifiOfdmaExample::GetPairwisePerAcStats (const WifiMacHeader &hdr, AcIndex ac)
{
  std::map<AcIndex, PairwisePerAcStats>::iterator mapIt;

  if (hdr.IsQosData () && !hdr.GetAddr1 ().IsGroup ())
    {
      ac = QosUtilsMapTidToAc (hdr.GetQosTid ());
    }
  else if (!hdr.IsBlockAckReq () || ac == AC_UNDEF)
    {
      // we need to count BARs transmnitted in HE TB PPDUs
      return std::make_pair (mapIt, false);
    }

  uint32_t srcNodeId = MacAddressToNodeId (hdr.GetAddr2 ());
  uint32_t dstNodeId = MacAddressToNodeId (hdr.GetAddr1 ());
  std::vector<std::map<AcIndex, PairwisePerAcStats>>::iterator vecIt;

  if (srcNodeId == 0)
    {
      // downlink
      vecIt = std::next (m_dlPerStaAcStats.begin (), dstNodeId - 1);
    }
  else
    {
      // uplink
      vecIt = std::next (m_ulPerStaAcStats.begin (), srcNodeId - 1);
    }

  mapIt = vecIt->find (ac);

  return std::make_pair (mapIt, mapIt != vecIt->end ());
}

void
WifiOfdmaExample::NotifyTxFailed (Ptr<const WifiMacQueueItem> mpdu)
{
  if (mpdu->GetHeader ().GetAddr1 ().IsGroup ())
    {
      return;
    }

  auto itPair = GetPairwisePerAcStats (mpdu->GetHeader ());
```

```
  if (itPair.second)
    {
      itPair.first->second.failed++;
    }
}

void
WifiOfdmaExample::NotifyMsduExpired (Ptr<const WifiMacQueueItem> item)
{
  if (item->GetHeader ().GetAddr1 ().IsGroup ())
    {
      return;
    }

  auto itPair = GetPairwisePerAcStats (item->GetHeader ());

  if (itPair.second)
    {
      itPair.first->second.expired++;
    }
}

void
WifiOfdmaExample::NotifyMsduRejected (Ptr<const WifiMacQueueItem> item)
{
  if (item->GetHeader ().GetAddr1 ().IsGroup ())
    {
      return;
    }

  auto itPair = GetPairwisePerAcStats (item->GetHeader ());

  if (itPair.second)
    {
      itPair.first->second.rejected++;
    }
}

void
WifiOfdmaExample::NotifyMsduDequeuedFromEdcaQueue (Time maxDelay, Ptr<const
    WifiMacQueueItem> item)
{
  if (!item->GetHeader ().IsQosData () || item->GetHeader ().GetAddr1 ().IsGroup () ||
      Simulator::Now () > item->GetTimeStamp () + maxDelay)
    {
      // the frame is not a unicast QoS data frame or the MSDU lifetime is higher than
    the
      // max queue delay, hence the MSDU has been discarded. Do nothing in this case.
      return;
    }

  uint32_t srcNodeId = MacAddressToNodeId (item->GetHeader ().GetAddr2 ());

  // update the pairwise HoL delay if this packet is being sent by the AP
  if (srcNodeId == 0)
    {
      auto itPair = GetPairwisePerAcStats (item->GetHeader ());

      if (itPair.second)
        {
          // a new HoL sample is stored if this is not the first MSDU being dequeued
          // and if this MSDU is not dequeued to be aggregated to a previously dequeued
          // MSDU (which would give a null sample)
          if (itPair.first->second.lastTxTime.IsStrictlyPositive () &&
              Simulator::Now () > itPair.first->second.lastTxTime)
            {
              double newHolSample =
                  (Simulator::Now () - Max (itPair.first->second.lastTxTime, item->
    GetTimeStamp ()))
                      .ToDouble (Time::MS);
              itPair.first->second.pairwiseHol.AddSample (newHolSample);
            }
          itPair.first->second.lastTxTime = Simulator::Now ();
```

```
        }
    }

  // update the aggregate HoL delay
  auto mapIt =
      m_perAcStats.at (srcNodeId).find (QosUtilsMapTidToAc (item->GetHeader ().GetQosTid
     ()));

  if (mapIt == m_perAcStats.at (srcNodeId).end ())
    {
      return;
    }

  // a new HoL sample is stored if this is not the first MSDU being dequeued
  // and if this MSDU is not dequeued to be aggregated to a previously dequeued
  // MSDU (which would give a null sample)
  if (mapIt->second.lastTxTime.IsStrictlyPositive () &&
      Simulator::Now () > mapIt->second.lastTxTime)
    {
      double newHolSample =
          (Simulator::Now () - Max (mapIt->second.lastTxTime, item->GetTimeStamp ()))
              .ToDouble (Time::MS);

      mapIt->second.aggregateHoL.AddSample (newHolSample);
    }
  mapIt->second.lastTxTime = Simulator::Now ();
}

void
WifiOfdmaExample::NotifyPsduForwardedDown (WifiConstPsduMap psduMap, WifiTxVector
    txVector, double txPowerW)
{
  Ptr<WifiNetDevice> dev = DynamicCast<WifiNetDevice> (m_apDevices.Get (0));
  Mac48Address apAddress = dev->GetMac ()->GetAddress ();

  if (psduMap.size () == 1 && psduMap.begin ()->second->GetAddr1 () == apAddress)
    {
      // Uplink frame
      const WifiMacHeader &hdr = psduMap.begin ()->second->GetHeader (0);

      if (txVector.GetPreambleType () == WIFI_PREAMBLE_HE_TB)
        {
          // HE TB PPDU
          if (hdr.HasData () || hdr.IsBlockAckReq ())
            {
              Time txDuration = WifiPhy::CalculateTxDuration (psduMap.begin ()->second->
    GetSize (),
                                                              txVector, m_band,
                                                              psduMap.begin ()->first);
              m_durationOfResponsesToLastBasicTf += txDuration;
              // double currRatio = txDuration.GetSeconds () / m_tfUlLength.GetSeconds
    ();

              AcIndex ac = AC_UNDEF;
              if (hdr.IsBlockAckReq ())
                {
                  CtrlBAckRequestHeader baReqHdr;
                  psduMap.begin ()->second->GetPayload (0)->PeekHeader (baReqHdr);
                  ac = QosUtilsMapTidToAc (baReqHdr.GetTidInfo ());
                }

              auto itPair = GetPairwisePerAcStats (hdr, ac);
              if (itPair.second)
                {
                  // itPair.first->second.ampduRatio.AddSample (currRatio);
                }
            }
          else if (hdr.GetType () == WIFI_MAC_QOSDATA_NULL)
            {
              m_countOfNullResponsesToLastTf++;
            }
        }
```

```
    if (hdr.HasData ())
      {
        auto itPair = GetPairwisePerAcStats (hdr);
        if (itPair.second)
          {
            itPair.first->second.ampduSize.AddSample (psduMap.begin ()->second->
  GetSize ());
          }
      }
  }
// Downlink frame
else if (psduMap.begin ()->second->GetHeader (0).IsQosData ())
  {
    for (auto &psdu : psduMap)
      {
        auto itPair = GetPairwisePerAcStats (psdu.second->GetHeader (0));
        if (itPair.second)
          {
            itPair.first->second.ampduSize.AddSample (psdu.second->GetSize ());
          }
      }

    Time dlMuPpduDuration = WifiPhy::CalculateTxDuration (psduMap, txVector, m_band);
    m_dlMuPpduDuration.AddSample (dlMuPpduDuration.ToDouble (Time::MS));

    // DL MU PPDU
    if (txVector.GetPreambleType () == WIFI_PREAMBLE_HE_MU)
      {
        Time dlMuPpduDuration = WifiPhy::CalculateTxDuration (psduMap, txVector,
  m_band);
        m_dlMuPpduDuration.AddSample (dlMuPpduDuration.ToDouble (Time::MS));
        Time psduDurationSum = Seconds (0);

        for (auto &staIdPsdu : psduMap)
          {
            // double currRatio = 0.0;

            if (staIdPsdu.second->GetSize () > 0)
              {
                // Time txDuration = WifiPhy::CalculateTxDuration (staIdPsdu.second->
  GetSize (),
                //                                                 txVector, m_band,
                //                                                 staIdPsdu.first);
                // psduDurationSum += txDuration;
                // currRatio = txDuration.GetSeconds () / dlMuPpduDuration.GetSeconds
  ();
              }

            auto itPair = GetPairwisePerAcStats (staIdPsdu.second->GetHeader (0));
            if (itPair.second)
              {
                // itPair.first->second.ampduRatio.AddSample (currRatio);
              }
          }
        m_dlMuCompleteness.AddSample (psduDurationSum.GetSeconds () /
                                      (dlMuPpduDuration.GetSeconds () * psduMap.size
  ()));
      }
  }
else if (psduMap.size () == 1 && psduMap.begin ()->second->GetHeader (0).IsTrigger ())
  {
    CtrlTriggerHeader trigger;
    psduMap.begin ()->second->GetPayload (0)->PeekHeader (trigger);

    // if (m_tfUlLength.IsStrictlyPositive ())
    //   {
    //     // This is not the first Trigger Frame being sent
    //     if (m_lastTfType == BASIC_TRIGGER)
    //       {
    //         if (m_durationOfResponsesToLastBasicTf.IsZero ())
    //           {
    //             // no station responded to the previous TF
    //             m_nFailedBasicTriggerFrames++;
```

```
      //           }
      //         else
      //           {
      //             // double currRatio = m_durationOfResponsesToLastBasicTf.GetSeconds
     () /
      //             //                    m_overallTimeGrantedByTf.GetSeconds ();
      //             // m_heTbCompleteness.AddSample (currRatio);
      //           }
      //       }
      //     else if (m_lastTfType == BSRP_TRIGGER)
      //       {
      //         if (m_countOfNullResponsesToLastTf == 0)
      //           {
      //             // no station responded to the previous TF
      //             m_nFailedBsrpTriggerFrames++;
      //           }
      //       }
      //   }

      // reset counters
      m_countOfNullResponsesToLastTf = 0;
      m_durationOfResponsesToLastBasicTf = Seconds (0);

      WifiTxVector heTbTxVector = trigger.GetHeTbTxVector (trigger.begin ()->GetAid12 ()
    );
      // m_tfUlLength = WifiPhy::ConvertLSigLengthToHeTbPpduDuration (trigger.
    GetUlLength (), heTbTxVector, m_band);
      // m_overallTimeGrantedByTf = m_tfUlLength * trigger.GetNUserInfoFields ();

      if (trigger.IsBasic ())
        {
          m_lastTfType = BASIC_TRIGGER;
          m_nBasicTriggerFramesSent++;
          // m_heTbPpduDuration.AddSample (m_tfUlLength.ToDouble (Time::MS));

          dev = DynamicCast<WifiNetDevice> (m_apDevices.Get (0));
          Ptr<ApWifiMac> mac = DynamicCast<ApWifiMac> (dev->GetMac ());

          for (auto &userInfo : trigger)
            {
              Mac48Address address = mac->GetStaList ().at (userInfo.GetAid12 ());
              std::size_t index = MacAddressToNodeId (address) - 1;
              NS_ASSERT (index < m_nSolicitingBasicTriggerFrames.size ());
              m_nSolicitingBasicTriggerFrames[index]++;
            }
        }
      else if (trigger.IsBsrp ())
        {
          m_lastTfType = BSRP_TRIGGER;
          m_nBsrpTriggerFramesSent++;
        }
    }
}

void
WifiOfdmaExample::NotifyTxopDuration (uint32_t nodeId, AcIndex ac, Time startTime, Time
    duration)
{
  auto mapIt = m_perAcStats.at (nodeId).find (ac);

  if (mapIt != m_perAcStats.at (nodeId).end ())
    {
      mapIt->second.txopDuration.AddSample (duration.ToDouble (Time::MS));
    }
}

void
WifiOfdmaExample::NotifySojournTime (std::size_t nodeId, AcIndex ac, Time sojournTime)
{
  auto mapIt = m_perAcStats.at (nodeId).find (ac);

  if (mapIt != m_perAcStats.at (nodeId).end ())
    {
```

```
      mapIt->second.queueDiscSojournTime.AddSample (sojournTime.ToDouble (Time::MS));
    }
}

void
WifiOfdmaExample::NotifyDropAfterDequeue (std::size_t nodeId, AcIndex ac,
                                          Ptr<const QueueDiscItem> item, const char *
    reason)
{
  auto mapIt = m_perAcStats.at (nodeId).find (ac);

  if (mapIt != m_perAcStats.at (nodeId).end ())
    {
      double sample = (Simulator::Now () - item->GetTimeStamp ()).ToDouble (Time::MS);
      mapIt->second.queueDiscSojournTime.RemoveSample (sample);
    }
}

void
WifiOfdmaExample::NotifyAppTx (std::size_t i, Ptr<const Packet> packet)
{
  m_flows[i].m_txBytes += packet->GetSize ();
  m_flows[i].m_txPackets++;
  bool inserted;

  if (m_flows[i].m_l4Proto == Flow::UDP)
    {
      inserted =
          m_flows[i].m_inFlightPackets.insert ({packet->GetUid (), Simulator::Now ()}).
    second;
      NS_ABORT_MSG_IF (!inserted, "Duplicate UID " << packet->GetUid () << " with UDP?")
    ;
    }
  else
    {
      inserted =
          m_flows[i].m_inFlightPackets.insert ({m_flows[i].m_txBytes, Simulator::Now ()
    }).second;
      NS_ABORT_MSG_IF (!inserted,
                       "Duplicate total bytes sent " << m_flows[i].m_txBytes << " with
    TCP?");
    }
}

void
WifiOfdmaExample::NotifyAppRx (std::size_t i, Ptr<const Packet> packet, const Address &
    address)
{
  m_flows[i].m_rxBytes += packet->GetSize ();
  m_flows[i].m_rxPackets++;

  if (m_flows[i].m_l4Proto == Flow::UDP)
    {
      // there must be a unique packet with the same UID as the received packet
      auto it = m_flows[i].m_inFlightPackets.find (packet->GetUid ());
      if (it == m_flows[i].m_inFlightPackets.end ())
        {
          NS_LOG_WARN ("Packet with UID " << packet->GetUid () << " not found");
          return;
        }

      m_flows[i].m_latency.AddSample ((Simulator::Now () - it->second).ToDouble (Time::
    MS));
      m_flows[i].m_inFlightPackets.erase (it);
    }
  else
    {
      uint64_t totalBytesReceived = DynamicCast<PacketSink> (m_sinkApps.Get (i))->
    GetTotalRx ();
      for (auto it = m_flows[i].m_inFlightPackets.begin (); it != m_flows[i].
    m_inFlightPackets.end (); )
        {
          if (it->first <= totalBytesReceived)
```

```
            {
              m_flows[i].m_latency.AddSample ((Simulator::Now () - it->second).ToDouble
    (Time::MS));
              it = m_flows[i].m_inFlightPackets.erase (it);
            }
          else
            {
              it++;
            }
        }
    }
}

void
WifiOfdmaExample::NotifyEdcaEnqueue (Ptr<const WifiMacQueueItem> item)
{
  if (!item->GetHeader ().IsQosData ())
    {
      return;
    }
  // init a map entry if the packet's UID is not present
  auto mapIt =
      m_inFlightPacketMap.insert ({item->GetPacket ()->GetUid (), std::list<
    InFlightPacketInfo> ()})
          .first;

  InFlightPacketInfo info;
  info.m_srcAddress = item->GetHeader ().GetAddr2 ();
  info.m_dstAddress = item->GetHeader ().GetAddr1 ();
  info.m_ac = QosUtilsMapTidToAc (item->GetHeader ().GetQosTid ());
  info.m_ptrToPacket = item->GetPacket ();
  info.m_edcaEnqueueTime = Simulator::Now ();

  mapIt->second.insert (mapIt->second.end (), info);
}

void
WifiOfdmaExample::NotifyMacForwardUp (Ptr<const Packet> p)
{
  auto mapIt = m_inFlightPacketMap.find (p->GetUid ());
  if (mapIt == m_inFlightPacketMap.end ())
    {
      NS_LOG_WARN ("No packet with UID " << p->GetUid () << " is currently in flight");
      return;
    }

  auto listIt = std::find_if (mapIt->second.begin (), mapIt->second.end (),
                              [&p](const InFlightPacketInfo& info)
                                { return info.m_ptrToPacket == p; });

  if (listIt == mapIt->second.end ())
    {
      NS_LOG_WARN ("Forwarding up a packet that has not been enqueued?");
      return;
    }

  if (listIt->m_dstAddress.IsGroup ())
    {
      return;
    }

  std::vector<std::map<AcIndex, PairwisePerAcStats>>::iterator vecIt;

  if (MacAddressToNodeId (listIt->m_srcAddress) == 0)
    {
      // downlink
      vecIt = std::next (m_dlPerStaAcStats.begin (), MacAddressToNodeId (listIt->
    m_dstAddress) - 1);
    }
  else
    {
      // uplink
      vecIt = std::next (m_ulPerStaAcStats.begin (), MacAddressToNodeId (listIt->
```

```
    m_srcAddress) - 1);
    }

  NS_ABORT_MSG_IF (vecIt->find (listIt->m_ac) == vecIt->end (),
                   "AC " << listIt->m_ac << " not found");

  NS_ABORT_MSG_IF (listIt->m_edcaEnqueueTime.IsZero (), "Unknown EDCA enqueue time for
    the packet");

  vecIt->at (listIt->m_ac)
      .l2Latency.AddSample ((Simulator::Now () - listIt->m_edcaEnqueueTime).ToDouble (
    Time::MS));
  mapIt->second.erase (listIt);
}

uint32_t
WifiOfdmaExample::ContextToNodeId (const std::string &context)
{
  std::string sub = context.substr (10);
  uint32_t pos = sub.find ("/Device");
  uint32_t nodeId = atoi (sub.substr (0, pos).c_str ());
  return nodeId;
}

int
main (int argc, char *argv[])
{
  WifiOfdmaExample example;
  example.Config (argc, argv);
  example.Setup ();
  example.Run ();
  example.PrintResults (std::cout);

  Simulator::Destroy ();

  return 0;
}
```

A.3 my-wifi-manager-example.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2016 University of Washington
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Authors: Tom Henderson <tomhend@u.washington.edu>
 *          Matias Richart <mrichart@fing.edu.uy>
 *          Sebastien Deronne <sebastien.deronne@gmail.com>
 */

// Test the operation of a wifi manager as the SNR is varied, and create
// a gnuplot output file for plotting.
//
// The test consists of a device acting as server and a device as client generating
    traffic.
//
// The output consists of a plot of the rate observed and selected at the client device.
//
// By default, the 802.11a standard using IdealWifiManager is plotted. Several command
    line
```

```
// arguments can change the following options:
// --wifiManager (Aarf, Aarfcd, Amrr, Arf, Cara, Ideal, Minstrel, MinstrelHt, Onoe, Rraa
    , ThompsonSampling)
// --standard (802.11a, 802.11b, 802.11g, 802.11p-10MHz, 802.11p-5MHz, 802.11n-5GHz,
    802.11n-2.4GHz, 802.11ac, 802.11ax-6GHz, 802.11ax-5GHz, 802.11ax-2.4GHz)
// --serverShortGuardInterval and --clientShortGuardInterval (for 802.11n/ac)
// --serverNss and --clientNss (for 802.11n/ac)
// --serverChannelWidth and --clientChannelWidth (for 802.11n/ac)
// --broadcast instead of unicast (default is unicast)
// --rtsThreshold (by default, value of 99999 disables it)

#include "ns3/log.h"
#include "ns3/config.h"
#include "ns3/uinteger.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include "ns3/gnuplot.h"
#include "ns3/command-line.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/ssid.h"
#include "ns3/propagation-loss-model.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/rng-seed-manager.h"
#include "ns3/mobility-helper.h"
#include "ns3/wifi-net-device.h"
#include "ns3/packet-socket-helper.h"
#include "ns3/packet-socket-client.h"
#include "ns3/packet-socket-server.h"
#include "ns3/ht-configuration.h"
#include "ns3/he-configuration.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("WifiManagerExample");

// 290K @ 20 MHz
const double NOISE_DBM_Hz = -174.0;
double noiseDbm = NOISE_DBM_Hz;

double g_intervalBytes = 0;
uint64_t g_intervalRate = 0;

void
PacketRx (Ptr<const Packet> pkt, const Address &addr)
{
  g_intervalBytes += pkt->GetSize ();
}

void
RateChange (uint64_t oldVal, uint64_t newVal)
{
  NS_LOG_DEBUG ("Change from " << oldVal << " to " << newVal);
  g_intervalRate = newVal;
}

/// Step structure
struct Step
{
  double stepSize; ///< step size in dBm
  double stepTime; ///< step size in seconds
};

/// StandardInfo structure
struct StandardInfo
{
  StandardInfo ()
  {
    m_name = "none";
  }
  /**
   * Constructor
   *
   * \param name reference name
```

```
   * \param standard wifi standard
   * \param width channel width
   * \param snrLow SNR low
   * \param snrHigh SNR high
   * \param xMin x minimum
   * \param xMax x maximum
   * \param yMax y maximum
   */
  StandardInfo (std::string name, WifiStandard standard, uint16_t width, double snrLow,
    double snrHigh, double xMin, double xMax, double yMax)
    : m_name (name),
      m_standard (standard),
      m_width (width),
      m_snrLow (snrLow),
      m_snrHigh (snrHigh),
      m_xMin (xMin),
      m_xMax (xMax),
      m_yMax (yMax)
  {
  }
  std::string m_name; ///< name
  WifiStandard m_standard; ///< standard
  uint16_t m_width; ///< channel width
  double m_snrLow; ///< lowest SNR
  double m_snrHigh; ///< highest SNR
  double m_xMin;  ///< X minimum
  double m_xMax;  ///< X maximum
  double m_yMax;  ///< Y maximum
};

void
ChangeSignalAndReportRate (Ptr<FixedRssLossModel> rssModel, Step step, double rss,
    Gnuplot2dDataset& rateDataset, Gnuplot2dDataset& actualDataset)
{
  NS_LOG_FUNCTION (rssModel << step.stepSize << step.stepTime << rss);
  double snr = rss - noiseDbm;
  rateDataset.Add (snr, g_intervalRate / 1e6);
  // Calculate received rate since last interval
  double currentRate = ((g_intervalBytes * 8) / step.stepTime) / 1e6; // Mb/s
  actualDataset.Add (snr, currentRate);
  rssModel->SetRss (rss - step.stepSize);
  NS_LOG_INFO ("At time " << Simulator::Now ().As (Time::S) << "; selected rate " << (
    g_intervalRate / 1e6) << "; observed rate " << currentRate << "; setting new power
    to " << rss - step.stepSize);
  g_intervalBytes = 0;
  Simulator::Schedule (Seconds (step.stepTime), &ChangeSignalAndReportRate, rssModel,
    step, (rss - step.stepSize), rateDataset, actualDataset);
}

int main (int argc, char *argv[])
{
  std::vector <StandardInfo> serverStandards;
  std::vector <StandardInfo> clientStandards;
  uint32_t steps;
  uint32_t rtsThreshold = 999999;  // disabled even for large A-MPDU
  uint32_t maxAmpduSize = 65535;
  double stepSize = 1; // dBm
  double stepTime = 1; // seconds
  uint32_t packetSize = 1024; // bytes
  bool broadcast = 0;
  int ap1_x = 0;
  int ap1_y = 0;
  int sta1_x = 5;
  int sta1_y = 0;
  uint16_t serverNss = 1;
  uint16_t clientNss = 1;
  uint16_t serverShortGuardInterval = 800;
  uint16_t clientShortGuardInterval = 800;
  uint16_t serverChannelWidth = 0;  // use default for standard and band
  uint16_t clientChannelWidth = 0;  // use default for standard and band
  std::string wifiManager ("Ideal");
  std::string standard ("802.11a");
  StandardInfo serverSelectedStandard;
```

```
StandardInfo clientSelectedStandard;
bool infrastructure = false;
uint32_t maxSlrc = 7;
uint32_t maxSsrc = 7;

CommandLine cmd (__FILE__);
cmd.AddValue ("maxSsrc", "The maximum number of retransmission attempts for a RTS
  packet", maxSsrc);
cmd.AddValue ("maxSlrc", "The maximum number of retransmission attempts for a Data
  packet", maxSlrc);
cmd.AddValue ("rtsThreshold", "RTS threshold", rtsThreshold);
cmd.AddValue ("maxAmpduSize", "Max A-MPDU size", maxAmpduSize);
cmd.AddValue ("stepSize", "Power between steps (dBm)", stepSize);
cmd.AddValue ("stepTime", "Time on each step (seconds)", stepTime);
cmd.AddValue ("broadcast", "Send broadcast instead of unicast", broadcast);
cmd.AddValue ("serverChannelWidth", "Set channel width of the server (valid only for
  802.11n or ac)", serverChannelWidth);
cmd.AddValue ("clientChannelWidth", "Set channel width of the client (valid only for
  802.11n or ac)", clientChannelWidth);
cmd.AddValue ("serverNss", "Set nss of the server (valid only for 802.11n or ac)",
  serverNss);
cmd.AddValue ("clientNss", "Set nss of the client (valid only for 802.11n or ac)",
  clientNss);
cmd.AddValue ("serverShortGuardInterval", "Set short guard interval of the server
  (802.11n/ac/ax) in nanoseconds", serverShortGuardInterval);
cmd.AddValue ("clientShortGuardInterval", "Set short guard interval of the client
  (802.11n/ac/ax) in nanoseconds", clientShortGuardInterval);
cmd.AddValue ("standard", "Set standard (802.11a, 802.11b, 802.11g, 802.11p-10MHz,
  802.11p-5MHz, 802.11n-5GHz, 802.11n-2.4GHz, 802.11ac, 802.11ax-6GHz, 802.11ax-5GHz,
  802.11ax-2.4GHz)", standard);
cmd.AddValue ("wifiManager", "Set wifi rate manager (Aarf, Aarfcd, Amrr, Arf, Cara,
  Ideal, Minstrel, MinstrelHt, Onoe, Rraa, ThompsonSampling)", wifiManager);
cmd.AddValue ("infrastructure", "Use infrastructure instead of adhoc", infrastructure)
  ;
cmd.Parse (argc,argv);

// Print out some explanation of what this program does
std::cout << std::endl << "This program demonstrates and plots the operation of
  different " << std::endl;
std::cout << "Wi-Fi rate controls on different station configurations," << std::endl;
std::cout << "by stepping down the received signal strength across a wide range" <<
  std::endl;
std::cout << "and observing the adjustment of the rate." << std::endl;
std::cout << "Run 'wifi-manager-example --PrintHelp' to show program options." << std
  ::endl << std::endl;

if (infrastructure == false)
  {
    NS_ABORT_MSG_IF (serverNss != clientNss, "In ad hoc mode, we assume sender and
  receiver are similarly configured");
  }

if (standard == "802.11b")
  {
    if (serverChannelWidth == 0)
      {
        serverChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211b,
  WIFI_PHY_BAND_2_4GHZ);
      }
    NS_ABORT_MSG_IF (serverChannelWidth != 22 && serverChannelWidth != 22, "Invalid
  channel width for standard " << standard);
    NS_ABORT_MSG_IF (serverNss != 1, "Invalid nss for standard " << standard);
    if (clientChannelWidth == 0)
      {
        clientChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211b,
  WIFI_PHY_BAND_2_4GHZ);
      }
    NS_ABORT_MSG_IF (clientChannelWidth != 22 && clientChannelWidth != 22, "Invalid
  channel width for standard " << standard);
    NS_ABORT_MSG_IF (clientNss != 1, "Invalid nss for standard " << standard);
  }
else if (standard == "802.11a" || standard == "802.11g")
  {
```

```
    if (serverChannelWidth == 0)
      {
        serverChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211g,
  WIFI_PHY_BAND_2_4GHZ);
      }
    NS_ABORT_MSG_IF (serverChannelWidth != 20, "Invalid channel width for standard "
  << standard);
    NS_ABORT_MSG_IF (serverNss != 1, "Invalid nss for standard " << standard);
    if (clientChannelWidth == 0)
      {
        clientChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211g,
  WIFI_PHY_BAND_2_4GHZ);
      }
    NS_ABORT_MSG_IF (clientChannelWidth != 20, "Invalid channel width for standard "
  << standard);
    NS_ABORT_MSG_IF (clientNss != 1, "Invalid nss for standard " << standard);
  }
else if (standard == "802.11n-5GHz" || standard == "802.11n-2.4GHz")
  {
    WifiPhyBand band = (standard == "802.11n-2.4GHz" ? WIFI_PHY_BAND_2_4GHZ :
  WIFI_PHY_BAND_5GHZ);
    if (serverChannelWidth == 0)
      {
        serverChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211n, band);
      }
    NS_ABORT_MSG_IF (serverChannelWidth != 20 && serverChannelWidth != 40, "Invalid
  channel width for standard " << standard);
    NS_ABORT_MSG_IF (serverNss == 0 || serverNss > 4, "Invalid nss " << serverNss << "
   for standard " << standard);
    if (clientChannelWidth == 0)
      {
        clientChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211n, band);
      }
    NS_ABORT_MSG_IF (clientChannelWidth != 20 && clientChannelWidth != 40, "Invalid
  channel width for standard " << standard);
    NS_ABORT_MSG_IF (clientNss == 0 || clientNss > 4, "Invalid nss " << clientNss << "
   for standard " << standard);
  }
else if (standard == "802.11ac")
  {
    if (serverChannelWidth == 0)
      {
        serverChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211ac,
  WIFI_PHY_BAND_5GHZ);
      }
    NS_ABORT_MSG_IF (serverChannelWidth != 20 && serverChannelWidth != 40 &&
  serverChannelWidth != 80 && serverChannelWidth != 160, "Invalid channel width for
  standard " << standard);
    NS_ABORT_MSG_IF (serverNss == 0 || serverNss > 4, "Invalid nss " << serverNss << "
   for standard " << standard);
    if (clientChannelWidth == 0)
      {
        clientChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211ac,
  WIFI_PHY_BAND_5GHZ);
      }
    NS_ABORT_MSG_IF (clientChannelWidth != 20 && clientChannelWidth != 40 &&
  clientChannelWidth != 80 && clientChannelWidth != 160, "Invalid channel width for
  standard " << standard);
    NS_ABORT_MSG_IF (clientNss == 0 || clientNss > 4, "Invalid nss " << clientNss << "
   for standard " << standard);
  }
else if (standard == "802.11ax-6GHz" ||standard == "802.11ax-5GHz" || standard == "
  802.11ax-2.4GHz")
  {
    WifiPhyBand band = (standard == "802.11ax-2.4GHz" ? WIFI_PHY_BAND_2_4GHZ :
  standard == "802.11ax-6GHz" ? WIFI_PHY_BAND_6GHZ : WIFI_PHY_BAND_5GHZ);
    if (serverChannelWidth == 0)
      {
        serverChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211ax, band);
      }
    NS_ABORT_MSG_IF (serverChannelWidth != 20 && serverChannelWidth != 40 &&
  serverChannelWidth != 80 && serverChannelWidth != 160, "Invalid channel width for
  standard " << standard);
```

```
    NS_ABORT_MSG_IF (serverNss == 0 || serverNss > 4, "Invalid nss " << serverNss << "
   for standard " << standard);
    if (clientChannelWidth == 0)
      {
        clientChannelWidth = GetDefaultChannelWidth (WIFI_PHY_STANDARD_80211ax, band);
      }
    NS_ABORT_MSG_IF (clientChannelWidth != 20 && clientChannelWidth != 40 &&
  clientChannelWidth != 80 && clientChannelWidth != 160, "Invalid channel width for
  standard " << standard);
    NS_ABORT_MSG_IF (clientNss == 0 || clientNss > 4, "Invalid nss " << clientNss << "
   for standard " << standard);
  }

// As channel width increases, scale up plot's yRange value
uint32_t channelRateFactor = std::max (clientChannelWidth, serverChannelWidth) / 20;
channelRateFactor = channelRateFactor * std::max (clientNss, serverNss);

// The first number is channel width, second is minimum SNR, third is maximum
// SNR, fourth and fifth provide xrange axis limits, and sixth the yaxis
// maximum
serverStandards.push_back (StandardInfo ("802.11a", WIFI_STANDARD_80211a, 20, 3, 27,
  0, 30, 60));
serverStandards.push_back (StandardInfo ("802.11b", WIFI_STANDARD_80211b, 22, -5, 11,
  -6, 15, 15));
serverStandards.push_back (StandardInfo ("802.11g", WIFI_STANDARD_80211g, 20, -5, 27,
  -6, 30, 60));
serverStandards.push_back (StandardInfo ("802.11n-5GHz", WIFI_STANDARD_80211n_5GHZ,
  serverChannelWidth, 3, 30, 0, 35, 80 * channelRateFactor));
serverStandards.push_back (StandardInfo ("802.11n-2.4GHz", WIFI_STANDARD_80211n_2_4GHZ
  , serverChannelWidth, 3, 30, 0, 35, 80 * channelRateFactor));
serverStandards.push_back (StandardInfo ("802.11ac", WIFI_STANDARD_80211ac,
  serverChannelWidth, 5, 50, 0, 55, 120 * channelRateFactor));
serverStandards.push_back (StandardInfo ("802.11p-10MHz", WIFI_STANDARD_80211p, 10, 3,
   27, 0, 30, 60));
serverStandards.push_back (StandardInfo ("802.11p-5MHz", WIFI_STANDARD_80211p, 5, 3,
  27, 0, 30, 60));
serverStandards.push_back (StandardInfo ("802.11ax-6GHz", WIFI_STANDARD_80211ax_6GHZ,
  serverChannelWidth, 5, 55, 0, 60, 120 * channelRateFactor));
serverStandards.push_back (StandardInfo ("802.11ax-5GHz", WIFI_STANDARD_80211ax_5GHZ,
  serverChannelWidth, 5, 55, 0, 60, 120 * channelRateFactor));
serverStandards.push_back (StandardInfo ("802.11ax-2.4GHz",
  WIFI_STANDARD_80211ax_2_4GHZ, serverChannelWidth, 5, 55, 0, 60, 120 *
  channelRateFactor));

clientStandards.push_back (StandardInfo ("802.11a", WIFI_STANDARD_80211a, 20, 3, 27,
  0, 30, 60));
clientStandards.push_back (StandardInfo ("802.11b", WIFI_STANDARD_80211b, 22, -5, 11,
  -6, 15, 15));
clientStandards.push_back (StandardInfo ("802.11g", WIFI_STANDARD_80211g, 20, -5, 27,
  -6, 30, 60));
clientStandards.push_back (StandardInfo ("802.11n-5GHz", WIFI_STANDARD_80211n_5GHZ,
  clientChannelWidth, 3, 30, 0, 35, 80 * channelRateFactor));
clientStandards.push_back (StandardInfo ("802.11n-2.4GHz", WIFI_STANDARD_80211n_2_4GHZ
  , clientChannelWidth, 3, 30, 0, 35, 80 * channelRateFactor));
clientStandards.push_back (StandardInfo ("802.11ac", WIFI_STANDARD_80211ac,
  clientChannelWidth, 5, 50, 0, 55, 120 * channelRateFactor));
clientStandards.push_back (StandardInfo ("802.11p-10MHz", WIFI_STANDARD_80211p, 10, 3,
   27, 0, 30, 60));
clientStandards.push_back (StandardInfo ("802.11p-5MHz", WIFI_STANDARD_80211p, 5, 3,
  27, 0, 30, 60));
clientStandards.push_back (StandardInfo ("802.11ax-6GHz", WIFI_STANDARD_80211ax_6GHZ,
  clientChannelWidth, 5, 55, 0, 60, 160 * channelRateFactor));
clientStandards.push_back (StandardInfo ("802.11ax-5GHz", WIFI_STANDARD_80211ax_5GHZ,
  clientChannelWidth, 5, 55, 0, 60, 160 * channelRateFactor));
clientStandards.push_back (StandardInfo ("802.11ax-2.4GHz",
  WIFI_STANDARD_80211ax_2_4GHZ, clientChannelWidth, 5, 55, 0, 60, 160 *
  channelRateFactor));

for (std::vector<StandardInfo>::size_type i = 0; i != serverStandards.size (); i++)
  {
    if (standard == serverStandards[i].m_name)
      {
        serverSelectedStandard = serverStandards[i];
```

```
      }
  }
for (std::vector<StandardInfo>::size_type i = 0; i != clientStandards.size (); i++)
  {
    if (standard == clientStandards[i].m_name)
      {
        clientSelectedStandard = clientStandards[i];
      }
  }

NS_ABORT_MSG_IF (serverSelectedStandard.m_name == "none", "Standard " << standard << "
   not found");
NS_ABORT_MSG_IF (clientSelectedStandard.m_name == "none", "Standard " << standard << "
   not found");
std::cout << "Testing " << serverSelectedStandard.m_name << " with " << wifiManager <<
   " ..." << std::endl;
NS_ABORT_MSG_IF (clientSelectedStandard.m_snrLow >= clientSelectedStandard.m_snrHigh,
  "SNR values in wrong order");
steps = static_cast<uint32_t> (std::abs (static_cast<double> (clientSelectedStandard.
  m_snrHigh - clientSelectedStandard.m_snrLow ) / stepSize) + 1);
NS_LOG_DEBUG ("Using " << steps << " steps for SNR range " << clientSelectedStandard.
  m_snrLow << ":" << clientSelectedStandard.m_snrHigh);
Ptr<Node> clientNode = CreateObject<Node> ();
Ptr<Node> serverNode = CreateObject<Node> ();

std::string plotName = "wifi-manager-example-";
std::string dataName = "wifi-manager-example-";
plotName += wifiManager;
dataName += wifiManager;
plotName += "-";
dataName += "-";
plotName += standard;
dataName += standard;
if (standard == "802.11n-5GHz"
    || standard == "802.11n-2.4GHz"
    || standard == "802.11ac"
    || standard == "802.11ax-6GHz"
    || standard == "802.11ax-5GHz"
    || standard == "802.11ax-2.4GHz")
  {
    plotName += "-server_";
    dataName += "-server_";
    std::ostringstream oss;
    oss << serverChannelWidth << "MHz_" << serverShortGuardInterval << "ns_" <<
  serverNss << "SS";
    plotName += oss.str ();
    dataName += oss.str ();
    plotName += "-client_";
    dataName += "-client_";
    oss.str ("");
    oss << clientChannelWidth << "MHz_" << clientShortGuardInterval << "ns_" <<
  clientNss << "SS";
    plotName += oss.str ();
    dataName += oss.str ();
  }
plotName += ".jpeg";
dataName += ".plt";
std::ofstream outfile (dataName.c_str ());
Gnuplot gnuplot = Gnuplot (plotName);

Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (maxSlrc))
  ;
Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (maxSsrc))
  ;
Config::SetDefault ("ns3::MinstrelWifiManager::PrintStats", BooleanValue (true));
Config::SetDefault ("ns3::MinstrelWifiManager::PrintSamples", BooleanValue (true));
Config::SetDefault ("ns3::MinstrelHtWifiManager::PrintStats", BooleanValue (true));

WifiHelper wifi;
wifi.SetStandard (serverSelectedStandard.m_standard);
YansWifiPhyHelper wifiPhy;

Ptr<YansWifiChannel> wifiChannel = CreateObject<YansWifiChannel> ();
```

```
// TODO <RandomPropagationDelayModel> <--another delay model that can be used
Ptr<ConstantSpeedPropagationDelayModel> delayModel = CreateObject<
  ConstantSpeedPropagationDelayModel> ();
wifiChannel->SetPropagationDelayModel (delayModel);
// TODO <RangePropagationLossModel>, <NakagamiPropagationLossModel>, <
  LogDistancePropagationLossModel>, <ThreeLogDistancePropagationLossModel>, <
  TwoRayGroundPropagationLossModel>, <FriisPropagationLossModel>, <
  RandomPropagationLossModel>
Ptr<FixedRssLossModel> rssLossModel = CreateObject<FixedRssLossModel> ();
wifiChannel->SetPropagationLossModel (rssLossModel);
wifiPhy.SetChannel (wifiChannel);

wifi.SetRemoteStationManager ("ns3::" + wifiManager + "WifiManager", "RtsCtsThreshold"
  , UintegerValue (rtsThreshold));

NetDeviceContainer serverDevice;
NetDeviceContainer clientDevice;

WifiMacHelper wifiMac;
if (infrastructure)
  {
    Ssid ssid = Ssid ("ns-3-ssid");
    wifiMac.SetType ("ns3::StaWifiMac",
                     "Ssid", SsidValue (ssid));
    wifiPhy.Set ("ChannelWidth", UintegerValue (serverSelectedStandard.m_width));
    serverDevice = wifi.Install (wifiPhy, wifiMac, serverNode);
    wifiMac.SetType ("ns3::ApWifiMac",
                     "Ssid", SsidValue (ssid));
    wifiPhy.Set ("ChannelWidth", UintegerValue (clientSelectedStandard.m_width));
    clientDevice = wifi.Install (wifiPhy, wifiMac, clientNode);
  }
else
  {
    wifiMac.SetType ("ns3::AdhocWifiMac");
    wifiPhy.Set ("ChannelWidth", UintegerValue (serverSelectedStandard.m_width));
    serverDevice = wifi.Install (wifiPhy, wifiMac, serverNode);
    wifiPhy.Set ("ChannelWidth", UintegerValue (clientSelectedStandard.m_width));
    clientDevice = wifi.Install (wifiPhy, wifiMac, clientNode);
  }

RngSeedManager::SetSeed (1);
RngSeedManager::SetRun (2);
wifi.AssignStreams (serverDevice, 100);
wifi.AssignStreams (clientDevice, 100);

Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/BE_MaxAmpduSize",
  UintegerValue (maxAmpduSize));

Config::ConnectWithoutContext ("/NodeList/0/DeviceList/*/$ns3::WifiNetDevice/
  RemoteStationManager/$ns3::" + wifiManager + "WifiManager/Rate", MakeCallback (&
  RateChange));

// Configure the mobility.
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
//Initial position of AP and STA
positionAlloc->Add (Vector (ap1_x, ap1_y, 0.0));
NS_LOG_INFO ("Setting initial AP position to " << Vector (ap1_x, ap1_y, 0.0));
positionAlloc->Add (Vector (sta1_x, sta1_y, 0.0));
NS_LOG_INFO ("Setting initial STA position to " << Vector (sta1_x, sta1_y, 0.0));
mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (clientNode);
mobility.Install (serverNode);

Gnuplot2dDataset rateDataset (clientSelectedStandard.m_name + std::string ("-rate
  selected"));
Gnuplot2dDataset actualDataset (clientSelectedStandard.m_name + std::string ("-
  observed"));
Step step;
step.stepSize = stepSize;
step.stepTime = stepTime;
```

```
// Perform post-install configuration from defaults for channel width,
// guard interval, and nss, if necessary
// Obtain pointer to the WifiPhy
Ptr<NetDevice> ndClient = clientDevice.Get (0);
Ptr<NetDevice> ndServer = serverDevice.Get (0);
Ptr<WifiNetDevice> wndClient = ndClient->GetObject<WifiNetDevice> ();
Ptr<WifiNetDevice> wndServer = ndServer->GetObject<WifiNetDevice> ();
Ptr<WifiPhy> wifiPhyPtrClient = wndClient->GetPhy ();
Ptr<WifiPhy> wifiPhyPtrServer = wndServer->GetPhy ();
uint8_t t_clientNss = static_cast<uint8_t> (clientNss);
uint8_t t_serverNss = static_cast<uint8_t> (serverNss);
wifiPhyPtrClient->SetNumberOfAntennas (t_clientNss);
wifiPhyPtrClient->SetMaxSupportedTxSpatialStreams (t_clientNss);
wifiPhyPtrClient->SetMaxSupportedRxSpatialStreams (t_clientNss);
wifiPhyPtrServer->SetNumberOfAntennas (t_serverNss);
wifiPhyPtrServer->SetMaxSupportedTxSpatialStreams (t_serverNss);
wifiPhyPtrServer->SetMaxSupportedRxSpatialStreams (t_serverNss);
// Only set the guard interval for HT and VHT modes
if (serverSelectedStandard.m_name == "802.11n-5GHz"
    || serverSelectedStandard.m_name == "802.11n-2.4GHz"
    || serverSelectedStandard.m_name == "802.11ac")
  {
    Ptr<HtConfiguration> clientHtConfiguration = wndClient->GetHtConfiguration ();
    clientHtConfiguration->SetShortGuardIntervalSupported (clientShortGuardInterval ==
   400);
    Ptr<HtConfiguration> serverHtConfiguration = wndServer->GetHtConfiguration ();
    serverHtConfiguration->SetShortGuardIntervalSupported (serverShortGuardInterval ==
   400);
  }
else if (serverSelectedStandard.m_name == "802.11ax-6GHz"
         || serverSelectedStandard.m_name == "802.11ax-5GHz"
         || serverSelectedStandard.m_name == "802.11ax-2.4GHz")
  {
    wndServer->GetHeConfiguration ()->SetGuardInterval (NanoSeconds (
  serverShortGuardInterval));
    wndClient->GetHeConfiguration ()->SetGuardInterval (NanoSeconds (
  clientShortGuardInterval));
  }
NS_LOG_DEBUG ("Channel width " << wifiPhyPtrClient->GetChannelWidth () << " noiseDbm "
   << noiseDbm);
NS_LOG_DEBUG ("NSS " << wifiPhyPtrClient->GetMaxSupportedTxSpatialStreams ());

// Configure signal and noise, and schedule first iteration
noiseDbm += 10 * log10 (clientSelectedStandard.m_width * 1000000);
double rssCurrent = (clientSelectedStandard.m_snrHigh + noiseDbm);
rssLossModel->SetRss (rssCurrent);
NS_LOG_INFO ("Setting initial Rss to " << rssCurrent);
//Move the STA by stepsSize meters every stepTime seconds
Simulator::Schedule (Seconds (0.5 + stepTime), &ChangeSignalAndReportRate,
  rssLossModel, step, rssCurrent, rateDataset, actualDataset);

PacketSocketHelper packetSocketHelper;
packetSocketHelper.Install (serverNode);
packetSocketHelper.Install (clientNode);

PacketSocketAddress socketAddr;
socketAddr.SetSingleDevice (serverDevice.Get (0)->GetIfIndex ());
if (broadcast)
  {
    socketAddr.SetPhysicalAddress (serverDevice.Get (0)->GetBroadcast ());
  }
else
  {
    socketAddr.SetPhysicalAddress (serverDevice.Get (0)->GetAddress ());
  }
// Arbitrary protocol type.
// Note: PacketSocket doesn't have any L4 multiplexing or demultiplexing
//       The only mux/demux is based on the protocol field
socketAddr.SetProtocol (1);

Ptr<PacketSocketClient> client = CreateObject<PacketSocketClient> ();
client->SetRemote (socketAddr);
client->SetStartTime (Seconds (0.5));  // allow simulation warmup
```

```
client->SetAttribute ("MaxPackets", UintegerValue (0));  // unlimited
client->SetAttribute ("PacketSize", UintegerValue (packetSize));

// Set a maximum rate 10% above the yMax specified for the selected standard
double rate = clientSelectedStandard.m_yMax * 1e6 * 1.10;
double clientInterval = static_cast<double> (packetSize) * 8 / rate;
NS_LOG_DEBUG ("Setting interval to " << clientInterval << " sec for rate of " << rate
  << " bits/sec");

client->SetAttribute ("Interval", TimeValue (Seconds (clientInterval)));
clientNode->AddApplication (client);

Ptr<PacketSocketServer> server = CreateObject<PacketSocketServer> ();
server->SetLocal (socketAddr);
server->TraceConnectWithoutContext ("Rx", MakeCallback (&PacketRx));
serverNode->AddApplication (server);

Simulator::Stop (Seconds ((steps + 1) * stepTime));
Simulator::Run ();
Simulator::Destroy ();

gnuplot.AddDataset (rateDataset);
gnuplot.AddDataset (actualDataset);

std::ostringstream xMinStr, xMaxStr, yMaxStr;
std::string xRangeStr ("set xrange [");
xMinStr << clientSelectedStandard.m_xMin;
xRangeStr.append (xMinStr.str ());
xRangeStr.append (":");
xMaxStr << clientSelectedStandard.m_xMax;
xRangeStr.append (xMaxStr.str ());
xRangeStr.append ("]");
std::string yRangeStr ("set yrange [0:");
yMaxStr << clientSelectedStandard.m_yMax;
yRangeStr.append (yMaxStr.str ());
yRangeStr.append ("]");

std::string title ("Results for ");
title.append (standard);
title.append (" with ");
title.append (wifiManager);
title.append ("\\n");
if (standard == "802.11n-5GHz"
    || standard == "802.11n-2.4GHz"
    || standard == "802.11ac"
    || standard == "802.11ax-6GHz"
    || standard == "802.11ax-5GHz"
    || standard == "802.11ax-2.4GHz")
  {
    std::ostringstream serverGiStrStr;
    std::ostringstream serverWidthStrStr;
    std::ostringstream serverNssStrStr;
    title.append ("server: width=");
    serverWidthStrStr << serverSelectedStandard.m_width;
    title.append (serverWidthStrStr.str ());
    title.append ("MHz");
    title.append (" GI=");
    serverGiStrStr << serverShortGuardInterval;
    title.append (serverGiStrStr.str ());
    title.append ("ns");
    title.append (" nss=");
    serverNssStrStr << serverNss;
    title.append (serverNssStrStr.str ());
    title.append ("\\n");
    std::ostringstream clientGiStrStr;
    std::ostringstream clientWidthStrStr;
    std::ostringstream clientNssStrStr;
    title.append ("client: width=");
    clientWidthStrStr << clientSelectedStandard.m_width;
    title.append (clientWidthStrStr.str ());
    title.append ("MHz");
    title.append (" GI=");
    clientGiStrStr << clientShortGuardInterval;
```

```
      title.append (clientGiStrStr.str ());
      title.append ("ns");
      title.append (" nss=");
      clientNssStrStr << clientNss;
      title.append (clientNssStrStr.str ());
    }
  gnuplot.SetTerminal ("jpeg enh ");
  gnuplot.SetLegend ("SNR (dB)", "Rate (Mb/s)");
  gnuplot.SetTitle (title);
  gnuplot.SetExtra (xRangeStr);
  gnuplot.AppendExtra (yRangeStr);
  gnuplot.AppendExtra ("set key top left");
  gnuplot.GenerateOutput (outfile);
  outfile.close ();

  return 0;
}
```

A.4 wifi_test.cc

```
#include "ns3/command-line.h"
#include "ns3/config.h"
#include "ns3/double.h"
#include "ns3/uinteger.h"
#include "ns3/boolean.h"
#include "ns3/string.h"
#include "ns3/log.h"
#include "ns3/ssid.h"
#include "ns3/callback.h"
#include "ns3/gnuplot.h"
#include "ns3/wifi-standards.h"
#include "ns3/wifi-helper.h"
#include "ns3/wifi-phy.h"
#include "ns3/wifi-net-device.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/yans-wifi-phy.h"
#include "ns3/mobility-helper.h"
#include "ns3/mobility-model.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/propagation-loss-model.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/packet-socket-helper.h"
#include "ns3/packet-socket-client.h"
#include "ns3/packet-socket-server.h"
#include "ns3/wifi-mac-header.h"
#include "ns3/csma-helper.h"
#include "ns3/rng-seed-manager.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("WifiSimpleTest");

double g_intervalBytes = 0;

void Monitor (std::string context, Ptr<const Packet> pkt, uint16_t channel, WifiTxVector
     tx, MpduInfo mpdu, SignalNoiseDbm snr, uint16_t staid)
//Ptr<const Packet>, uint16_t /* frequency (MHz) */, WifiTxVector, MpduInfo,
    SignalNoiseDbm, uint16_t /* STA-ID*/
{
  std::cout << context << std::endl;
  std::cout << "\tChannel :" << channel << " Tx: " << tx.GetMode() << " \tSingal= " <<
    snr.signal <<  "\tNoise= " << snr.noise << std::endl;
  std::cout << "\tTime: " << Simulator::Now() << std::endl;

  WifiMacHeader hdr;
  if (pkt->PeekHeader(hdr))
  {
    std::cout << "\tAddr1: " << hdr.GetAddr1() << "\t" << hdr.GetAddr2() << std::endl;
    std::cout << "\tSeqNo. " << hdr.GetSequenceNumber() << "\t" << (hdr.IsData() ? "Data
    " : "Not Data") << std::endl;
  }
}
```

```
void PacketRx(Ptr<const Packet> pkt, const Address &addr){
  //NS_LOG_FUNCTION(addr);
  g_intervalBytes += pkt->GetSize();
}

void AddNodeReportRate ( double step_time,  Gnuplot2dDataset& rateDataset) {
  NS_LOG_FUNCTION (step_time);
  NS_LOG_INFO ("At time " << Simulator::Now().As (Time::S) << " scheduling next
    increment");
  double currentRate = ((g_intervalBytes * 8) / step_time) / 1e6; // Mb/s
  rateDataset.Add (Simulator::Now().GetDouble(), currentRate );
  g_intervalBytes = 0;
  Simulator::Schedule (Seconds(step_time), &AddNodeReportRate, step_time, rateDataset);
}

int main (int argc, char *argv[]) {
  //cmd line arguements
  uint32_t rtsThreshold = 999999;  // disabled even for large A-MPDU
  //std::string phyMode ("DsssRate1Mbps");
  std::string wifiManager ("MinstrelHt");
  uint32_t packetSize = 1024;
  double stepTime = 0.5;
  //uint8_t steps = 20;
  bool verbose = false;
  uint32_t nCsma = 3;
  uint32_t nWifi = 2;
  uint32_t nPackets = 100;
  bool tracing = true;
  uint32_t maxSlrc = 7;
  uint32_t maxSsrc = 7;
  uint16_t serverNss = 4; // Number of spatial streams
  uint16_t clientNss = 1;
  //uint16_t serverShortGuardInterval = 800;
  //uint16_t clientShortGuardInterval = 800;

  CommandLine cmd (__FILE__);
  //cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
  cmd.AddValue ("wifiManager", "Set wifi rate manager (Aarf, Aarfcd, Amrr, Arf, Cara,
    Ideal, Minstrel, MinstrelHt, Onoe, Rraa, ThompsonSampling)", wifiManager);
  cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", verbose);
  cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
  cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
  cmd.AddValue ("nPackets", "Number of packets to echo", nPackets);
  cmd.AddValue ("tracing", "Enable pcap tracing", tracing);
  cmd.Parse (argc, argv);

  /*****************************************/
  /* WIFI Standards and setting configuration */
  /*****************************************/
  NS_LOG_INFO("Configuring WiFi Parameters");
  WifiHelper wifi;
  if (verbose)
  {
    wifi.EnableLogComponents ();  // Turn on all Wifi logging
  }

  Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (maxSlrc))
    ;
  Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (maxSsrc))
    ;
  Config::SetDefault ("ns3::MinstrelWifiManager::PrintStats", BooleanValue (true));
  Config::SetDefault ("ns3::MinstrelWifiManager::PrintSamples", BooleanValue (true));
  Config::SetDefault ("ns3::MinstrelHtWifiManager::PrintStats", BooleanValue (true));
  // set the wifi standard
  wifi.SetStandard (WIFI_STANDARD_80211ax_5GHZ);
  YansWifiPhyHelper wifiPhy;
  // get the maximum channel width
  uint16_t ch_width = GetDefaultChannelWidth(WIFI_PHY_STANDARD_80211ax,
    WIFI_PHY_BAND_5GHZ);
  uint16_t max_ch_width = GetMaximumChannelWidth(WIFI_PHY_STANDARD_80211ax);

  uint32_t channelRateFactor = ch_width / 20;
```

```
  channelRateFactor = channelRateFactor * std::max (clientNss, serverNss);

  //Channel
  Ptr<YansWifiChannel> wifiChannel = CreateObject<YansWifiChannel> ();
  // Add propagation delay model to channel
  // TODO could use <ConstantSpeedPropagationDelayModel>, <RandomPropagationDelayModel>
  Ptr<ConstantSpeedPropagationDelayModel> delayModel = CreateObject<
    ConstantSpeedPropagationDelayModel> ();
  wifiChannel->SetPropagationDelayModel (delayModel);
  // Add Recieve Signal Strength (RSS) Loss Model
  // TODO decide Model: <FixedRssLossModel>, <RangePropagationLossModel>,
  // <NakagamiPropagationLossModel>, <LogDistancePropagationLossModel>,
  // <ThreeLogDistancePropagationLossModel>, <TwoRayGroundPropagationLossModel>,
  // <FriisPropagationLossModel>, <RandomPropagationLossModel>
  Ptr<FixedRssLossModel> rssLossModel = CreateObject<FixedRssLossModel> ();
  wifiChannel->SetPropagationLossModel (rssLossModel);
  // PHY
  wifiPhy.SetChannel(wifiChannel);
  //std::cout << "Default Channel Width: " << ch_width << std::endl;
  //std::cout << "Maximum Channel Width: " << max_ch_width << std::endl;
  wifi.SetRemoteStationManager("ns3::" + wifiManager + "WifiManager", "RtsCtsThreshold",
     UintegerValue (rtsThreshold));



  /**************************************/
  /* Node and Network Topology creation */
  /**************************************/
  // Nodes to be used
  NS_LOG_INFO("Creating Client Server Nodes");
  //Ptr<Node> clientNode = CreateObject<Node> ();
  //Ptr<Node> serverNode = CreateObject<Node> ();
#if 0
  // TODO nodes as above or below? probably below, can modify at line 96, 103
  NodeContainer p2pNodes;
  p2pNodes.Create (2);

  NodeContainer wifi_ap_node= p2pNodes.Get (0);
  //wifi_ap_node.Create(1);

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

  CsmaHelper csma;
  csma.SetChannelAttribute ("DataRate", StringValue ("10Gbps"));
  csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));
#endif

  NodeContainer wifi_ap_node;
  wifi_ap_node.Create(1);
  NodeContainer wifi_dev_node;
  wifi_dev_node.Create(1);//nWifi);
  // Devices
  NetDeviceContainer serverDevice;
  NetDeviceContainer clientDevice;
  //NetDeviceContainer clientDevice2;

  // MAC setup associated with nodes
  // Client MAC setup
  WifiMacHelper wifiMac;
  Ssid ssid = Ssid ("LocalNetwork");
  wifiMac.SetType("ns3::ApWifiMac", "Ssid", SsidValue (ssid)); // TODO verify AP vs Sta
  // TODO Lets look into Channel width parameters (max_ch_width)
  wifiPhy.Set ("ChannelWidth", UintegerValue (ch_width));
  serverDevice = wifi.Install(wifiPhy, wifiMac, wifi_ap_node.Get(0)); //assocate the
    various setting
  // AP MAC setup
  ssid = Ssid ("LocalNetwork");
  wifiMac.SetType("ns3::StaWifiMac", "Ssid", SsidValue (ssid));
  // TODO Lets look into Channel width parameters (max_ch_width)
  wifiPhy.Set ("ChannelWidth", UintegerValue (ch_width));
  clientDevice = wifi.Install(wifiPhy, wifiMac, wifi_dev_node.Get(0));
```

```
//clientDevice2 = wifi.Install(wifiPhy, wifiMac, wifi_dev_node.Get(1));
RngSeedManager::SetSeed (1);
RngSeedManager::SetRun (2);
wifi.AssignStreams (serverDevice, 100);
wifi.AssignStreams (clientDevice, 100);
//wifi.AssignStreams (clientDevice2, 100);
/**************************/
/* Plotting data settings */
/**************************/
std::string plotNameThru = "wifi6-network-test-throughput.jpeg";
std::string dataNameThru = "wifi6-network-test-throughput.plt";
std::ofstream outfile (dataNameThru.c_str ());
//std::string plotNameLat = "wifi6-network-test-latency.jpeg";
//std::string dataNameLat = "wifi6-network-test-throughput.plt";
Gnuplot2dDataset rateData("observed rate");

/****************/
/* PHY settings */
/****************/
// Perform post-install configuration from defaults for channel width,
// guard interval, and nss, if necessary
// Obtain pointer to the WifiPhy
Ptr<NetDevice> ndClient = clientDevice.Get(0);
//Ptr<NetDevice> ndClient2 = clientDevice2.Get(0);
Ptr<NetDevice> ndServer = serverDevice.Get(0);
Ptr<WifiNetDevice> wndClient = ndClient->GetObject<WifiNetDevice>();
//Ptr<WifiNetDevice> wndClient2 = ndClient2->GetObject<WifiNetDevice>();
Ptr<WifiNetDevice> wndServer = ndServer->GetObject<WifiNetDevice>();
Ptr<WifiPhy> wifiPhyPtrClient = wndClient->GetPhy();
//Ptr<WifiPhy> wifiPhyPtrClient2 = wndClient2->GetPhy();
Ptr<WifiPhy> wifiPhyPtrServer = wndServer->GetPhy();
uint8_t t_clientNss = static_cast<uint8_t>(1); //1-4 are acceptable
uint8_t t_serverNss = static_cast<uint8_t>(1); //4 streams (and antennas for server)
wifiPhyPtrClient->SetNumberOfAntennas(t_clientNss);
wifiPhyPtrClient->SetMaxSupportedTxSpatialStreams(t_clientNss);
wifiPhyPtrClient->SetMaxSupportedRxSpatialStreams(t_clientNss);
//wifiPhyPtrClient2->SetNumberOfAntennas(t_clientNss);
//wifiPhyPtrClient2->SetMaxSupportedTxSpatialStreams(t_clientNss);
//wifiPhyPtrClient2->SetMaxSupportedRxSpatialStreams(t_clientNss);
wifiPhyPtrServer->SetNumberOfAntennas(t_serverNss);
wifiPhyPtrServer->SetMaxSupportedTxSpatialStreams(t_serverNss);
wifiPhyPtrServer->SetMaxSupportedRxSpatialStreams(t_serverNss);
// Only set the guard interval for HT and VHT modes
//wndServer->GetHeConfiguration()->SetGuardInterval(NanoSeconds (
  serverShortGuardInterval));
//wndClient->GetHeConfiguration()->SetGuardInterval(NanoSeconds (
  clientShortGuardInterval));
NS_LOG_DEBUG ("Channel width " << wifiPhyPtrClient->GetChannelWidth());
NS_LOG_DEBUG ("NSS " << wifiPhyPtrClient->GetMaxSupportedTxSpatialStreams());
rssLossModel->SetRss (-20);
/**************************************/
/* Traffic creation for data collection */
/**************************************/
//int step = 0
//start schedule events
Simulator::Schedule( Seconds (0.5+stepTime), &AddNodeReportRate, stepTime, rateData);
  // TODO need to get function arguements set up and nodes sorted

PacketSocketHelper PacketSocketHelper;
PacketSocketHelper.Install(wifi_ap_node.Get(0));
PacketSocketHelper.Install(wifi_dev_node.Get(0));
//PacketSocketHelper.Install(wifi_dev_node.Get(1));

PacketSocketAddress sockAddr;
sockAddr.SetSingleDevice(serverDevice.Get(0)->GetIfIndex());
sockAddr.SetPhysicalAddress(serverDevice.Get(0)->GetAddress());
sockAddr.SetProtocol(1); // TODO what are the other options, this is arbitrary
  protocol?

// Client data attributs
Ptr<PacketSocketClient> client = CreateObject<PacketSocketClient> ();
client->SetRemote(sockAddr);
client->SetStartTime(Seconds (0.5));
```

```
  client->SetAttribute("MaxPackets", UintegerValue(0)); //unlimited packets
  client->SetAttribute("PacketSize", UintegerValue(packetSize));
  //Ptr<PacketSocketClient> client2 = CreateObject<PacketSocketClient> ();
  //client2->SetRemote(sockAddr);
  //client2->SetStartTime(Seconds (1.0));
  //client2->SetAttribute("MaxPackets", UintegerValue(0)); //unlimited packets
  //client2->SetAttribute("PacketSize", UintegerValue(packetSize));
  // TODO Need to set local client address and add rx callback if this is where we are
    receiveing throughtput data

  double rate = max_ch_width*1e6; //maximum rate for channel
  double interval = 8*static_cast<double>(packetSize)/rate;
  NS_LOG_INFO("Setting interval to " << interval << " sec for rate of " << rate << "
    bits/sec");

  client->SetAttribute ("Interval", TimeValue (Seconds (interval)));
  //client2->SetAttribute ("Interval", TimeValue (Seconds (interval)));
  wifi_dev_node.Get(0)->AddApplication (client);
  //wifi_dev_node.Get(1)->AddApplication (client2);
  Ptr<PacketSocketServer> server = CreateObject<PacketSocketServer>();
  server->SetLocal(sockAddr);
  server->TraceConnectWithoutContext("Rx", MakeCallback(&PacketRx)); // TODO need to
    determine what we do when we recieve packet
  wifi_ap_node.Get(0)->AddApplication (server);
  // TODO Need to keep adding remote clients (scheduler function? and sending data to
    them)

  Simulator::Stop(Seconds (10.0)); // simulation total run time

  //Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/MonitorSnifferRx",
     MakeCallback(&Monitor)); //Monitor callback
  //Config::ConnectWithoutContext ("/NodeList/0/DeviceList/*/$ns3::WifiNetDevice/
    RemoteStationManager/$ns3::" + wifiManager + "WifiManager/Rate", MakeCallback (&
    RateChange));

  wifiPhy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
  wifiPhy.EnablePcapAll("WifiTest");

  // Configure the mobility.
  MobilityHelper mobility;
  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
  //Initial position of AP and STA
  positionAlloc->Add (Vector (0.0, 0.0, 0.0));
  NS_LOG_INFO ("Setting initial AP position to " << Vector (0.0, 0.0, 0.0));
  positionAlloc->Add (Vector (0.0, 5.0, 0.0));
  NS_LOG_INFO ("Setting initial STA position to " << Vector (0.0, 0.5, 0.0));
  //positionAlloc->Add (Vector (0.5, 0.0, 0.0));
  //NS_LOG_INFO ("Setting initial STA position to " << Vector (0.5, 0.0, 0.0));
  mobility.SetPositionAllocator (positionAlloc);
  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install(wifi_dev_node.Get(0));
  //mobility.Install(wifi_dev_node.Get(1));
  mobility.Install (wifi_ap_node.Get(0));

#if 0
  /**************************************/
  /* Tracing info creation for pcap collection */
  /**************************************/
  if (tracing)
    {
      wifiPhy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
    // DLT_IEEE802_11       = PcapHelper::DLT_IEEE802_11,       /**< IEEE 802.11
    Wireless LAN headers on packets */
      // DLT_PRISM_HEADER     = PcapHelper::DLT_PRISM_HEADER,     /**< Include Prism
    monitor mode information */
      // DLT_IEEE802_11_RADIO = PcapHelper::DLT_IEEE802_11_RADIO  /**< Include Radiotap
    link layer information */
      pointToPoint.EnablePcapAll ("WifiSimpleTest");
      phy.EnablePcap ("WifiSimpleTest", apDevices.Get (0));
      csma.EnablePcap ("WifiSimpleTest", csmaDevices.Get (0), true);
    }

  std::ostringstream oss;
```

```
  oss << "/NodeList/" << wifiStaNodes.Get (nWifi - 1)->GetId () << "/$ns3::MobilityModel
    /CourseChange";

  Config::Connect (oss.str (), MakeCallback (&CourseChange));
#endif
  /***************************************/
  /* Simulation Scheduler and execution */
  /***************************************/
  //Simulator::Schedule( Seconds (0.5+stepTime), &AddNodeReportRate, /*args*/);
  //Simulator::Stop (Seconds ((steps + 1) * stepTime));
  Simulator::Run ();
  Simulator::Destroy ();

  /*****************************************************/
  /* Plot output settings after simulation completion */
  /*****************************************************/
  Gnuplot gnuplot = Gnuplot (plotNameThru);
  gnuplot.AddDataset (rateData);
  gnuplot.SetTerminal ("jpeg enh ");
  gnuplot.SetLegend ("time (ns)", "Rate (Mb/s)");
  gnuplot.SetTitle ("WifiTest");
  //gnuplot.SetExtra (xRangeStr);
  //gnuplot.AppendExtra (yRangeStr);
  gnuplot.AppendExtra ("set key top left");
  gnuplot.GenerateOutput (outfile);
  outfile.close ();

  return 0;

}
```

A.5 wifi-throughput-test.cc

```
#include "ns3/command-line.h"
#include "ns3/config.h"
#include "ns3/double.h"
#include "ns3/uinteger.h"
#include "ns3/boolean.h"
#include "ns3/string.h"
#include "ns3/log.h"
#include "ns3/ssid.h"
#include "ns3/callback.h"
#include "ns3/gnuplot.h"
#include "ns3/wifi-standards.h"
#include "ns3/wifi-helper.h"
#include "ns3/wifi-phy.h"
#include "ns3/wifi-net-device.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/yans-wifi-phy.h"
#include "ns3/mobility-helper.h"
#include "ns3/mobility-model.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/propagation-loss-model.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/packet-socket-helper.h"
#include "ns3/packet-socket-client.h"
#include "ns3/packet-socket-server.h"
#include "ns3/wifi-mac-header.h"
#include "ns3/csma-helper.h"
#include "ns3/rng-seed-manager.h"
#include "ns3/point-to-point-helper.h"
#include "ns3/rectangle.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/address.h"
#include "ns3/udp-client-server-helper.h"
#include <math.h>

// Default Network Topology
// Nodes 0-1 are Point to point
// CSMA nodes 2-i+1
// Wifi nodes i+2-2i+1
```

```
//
//   Wifi 10.1.3.0
//    -i WiFi nodes-  AP
//  *  ...  *    *    *
//  |       |    |    |    10.1.1.0      -i CSMA devices
// n2i     ni+2  ni+1 n0 -------------- n1   n2   n3 ... ni
//                       point-to-point  |    |    |      |
//                                       ==================
//                                         LAN 10.1.2.0

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("WifiNetworkThroughputTest");

double g_intervalBytes = 0;

void Monitor (std::string context, Ptr<const Packet> pkt, uint16_t channel, WifiTxVector
     tx, MpduInfo mpdu, SignalNoiseDbm snr, uint16_t staid)
//Ptr<const Packet>, uint16_t /* frequency (MHz) */, WifiTxVector, MpduInfo,
    SignalNoiseDbm, uint16_t /* STA-ID*/
{
  std::cout << context << std::endl;
  std::cout << "\tChannel :" << channel << " Tx: " << tx.GetMode() << " \tSingal= " <<
    snr.signal <<  "\tNoise= " << snr.noise << std::endl;
  std::cout << "\tTime: " << Simulator::Now() << std::endl;

  WifiMacHeader hdr;
  if (pkt->PeekHeader(hdr))
  {
    std::cout << "\tAddr1: " << hdr.GetAddr1() << "\t" << hdr.GetAddr2() << std::endl;
    std::cout << "\tSeqNo. " << hdr.GetSequenceNumber() << "\t" << (hdr.IsData() ? "Data
    " : "Not Data") << std::endl;
  }
}

void PacketRx(Ptr<const Packet> pkt, const Address &addr){
  //NS_LOG_FUNCTION(addr);
  g_intervalBytes += pkt->GetSize();
}

void AddNodeReportRate ( double step_time,  Gnuplot2dDataset& rateDataset) {
  NS_LOG_FUNCTION (step_time);
  NS_LOG_INFO ("At time " << Simulator::Now().As (Time::S) << " scheduling next
    increment");
  double currentRate = ((g_intervalBytes * 8) / step_time) / 1e6; // Mb/s
  rateDataset.Add (Simulator::Now().GetDouble(), currentRate );
  g_intervalBytes = 0;
  Simulator::Schedule (Seconds(step_time), &AddNodeReportRate, step_time, rateDataset);
}

int main (int argc, char *argv[]) {
  //cmd line arguements
  uint32_t rtsThreshold = 999999;  // disabled even for large A-MPDU
  //std::string phyMode ("DsssRate1Mbps");
  std::string wifiManager ("MinstrelHt");
  uint32_t packetSize = 1024;
  double stepTime = 0.5;
  //uint8_t steps = 20;
  bool verbose = false;
  uint32_t nCsma = 2;
  uint32_t nWifi = 2;
  uint32_t nPackets = 100;
  bool tracing = true;
  uint32_t maxSlrc = 7;
  uint32_t maxSsrc = 7;
  uint16_t serverNss = 4; // Number of spatial streams
  uint16_t clientNss = 1;
  //uint16_t serverShortGuardInterval = 800;
  //uint16_t clientShortGuardInterval = 800;

  CommandLine cmd (__FILE__);
  //cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
  cmd.AddValue ("wifiManager", "Set wifi rate manager (Aarf, Aarfcd, Amrr, Arf, Cara,
```

```
    Ideal, Minstrel, MinstrelHt, Onoe, Rraa, ThompsonSampling)", wifiManager);
  cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", verbose);
  cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
  cmd.AddValue ("nPackets", "Number of packets to echo", nPackets);
  cmd.AddValue ("tracing", "Enable pcap tracing", tracing);
  cmd.Parse (argc, argv);
  nCsma = nWifi; //need a souce and sink pair for every wifi device
  /*****************************************/
  /* WIFI Standards and setting configuration */
  /*****************************************/
  NS_LOG_INFO("Configuring WiFi Parameters");
  WifiHelper wifi;
  if (verbose)
  {
    wifi.EnableLogComponents ();  // Turn on all Wifi logging
  }

  Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (maxSlrc))
    ;
  Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (maxSsrc))
    ;
  Config::SetDefault ("ns3::MinstrelWifiManager::PrintStats", BooleanValue (true));
  Config::SetDefault ("ns3::MinstrelWifiManager::PrintSamples", BooleanValue (true));
  Config::SetDefault ("ns3::MinstrelHtWifiManager::PrintStats", BooleanValue (true));
  // set the wifi standard
  wifi.SetStandard (WIFI_STANDARD_80211ax_5GHZ);
  YansWifiPhyHelper wifiPhy;
  // get the maximum channel width
  uint16_t ch_width = GetDefaultChannelWidth(WIFI_PHY_STANDARD_80211ax,
    WIFI_PHY_BAND_5GHZ);
  uint16_t max_ch_width = GetMaximumChannelWidth(WIFI_PHY_STANDARD_80211ax);

  uint32_t channelRateFactor = ch_width / 20;
  channelRateFactor = channelRateFactor * std::max (clientNss, serverNss);

  //Channel
  Ptr<YansWifiChannel> wifiChannel = CreateObject<YansWifiChannel> ();
  // Add propagation delay model to channel
  // TODO could use <ConstantSpeedPropagationDelayModel>, <RandomPropagationDelayModel>
  Ptr<ConstantSpeedPropagationDelayModel> delayModel = CreateObject<
    ConstantSpeedPropagationDelayModel> ();
  wifiChannel->SetPropagationDelayModel (delayModel);
  // Add Recieve Signal Strength (RSS) Loss Model
  // TODO decide Model: <FixedRssLossModel>, <RangePropagationLossModel>,
  // <NakagamiPropagationLossModel>, <LogDistancePropagationLossModel>,
  // <ThreeLogDistancePropagationLossModel>, <TwoRayGroundPropagationLossModel>,
  // <FriisPropagationLossModel>, <RandomPropagationLossModel>
  Ptr<FixedRssLossModel> rssLossModel = CreateObject<FixedRssLossModel> ();
  wifiChannel->SetPropagationLossModel (rssLossModel);
  // PHY
  wifiPhy.SetChannel(wifiChannel);
  //std::cout << "Default Channel Width: " << ch_width << std::endl;
  //std::cout << "Maximum Channel Width: " << max_ch_width << std::endl;
  wifi.SetRemoteStationManager("ns3::" + wifiManager + "WifiManager", "RtsCtsThreshold",
     UintegerValue (rtsThreshold));

  /*************************************/
  /* Node and Network Topology creation */
  /*************************************/
  // Nodes to be used
  NS_LOG_INFO("Creating Client Server Nodes");
//Connection from AP to 'internet'
  NodeContainer p2pNodes;
  p2pNodes.Create (2);

  //Pont to point conenction from our LAN to outside CSMA network
  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("10Gbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
  // AP and ISP connection to some arbitrary network
  NetDeviceContainer p2pDevices;
  p2pDevices = pointToPoint.Install (p2pNodes);
```

```
//Connection from 'ISP' to servers
NodeContainer csmaNodes;
//CSMA node connected to Wifi Router
csmaNodes.Add (p2pNodes.Get (1)); //P2P node 1 reepresents ISP
// nodes past ISP
csmaNodes.Create (nCsma);

CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("1Gbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

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

// creation of Local Area Nodes
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi); // Stations ie local hosts
NodeContainer wifiApNode = p2pNodes.Get (0); //Access Point
// MAC setup associated with nodes
// AP MAC and PHY setup
WifiMacHelper wifiMac;
Ssid ssid = Ssid ("LocalNetwork");
wifiMac.SetType("ns3::ApWifiMac", "Ssid", SsidValue (ssid));
wifiPhy.Set ("ChannelWidth", UintegerValue (ch_width));
NetDeviceContainer apDevice;
apDevice = wifi.Install(wifiPhy, wifiMac, wifiApNode); //assocate the various setting

// STA MAC and PHY setup
wifiMac.SetType("ns3::StaWifiMac", "Ssid", SsidValue (ssid));
// TODO Lets look into Channel width parameters (max_ch_width)
wifiPhy.Set ("ChannelWidth", UintegerValue (ch_width));
NetDeviceContainer staDevices;
staDevices = wifi.Install(wifiPhy, wifiMac, wifiStaNodes);
/****************/
/* PHY settings */
/****************/
// Perform post-install configuration from defaults for channel width,
// guard interval, and nss, if necessary
// Obtain pointer to the WifiPhy
int nDev = staDevices.GetN();
for (int i = 0; i < nDev; ++i ){
  // set PHY parameters for each STA
  Ptr<NetDevice> ndClient = staDevices.Get(i);
  Ptr<WifiNetDevice> wndClient = ndClient->GetObject<WifiNetDevice>();
  Ptr<WifiPhy> wifiPhyPtrClient = wndClient->GetPhy();
  uint8_t t_clientNss = static_cast<uint8_t>(1); //1-4 are acceptable
  wifiPhyPtrClient->SetNumberOfAntennas(t_clientNss);
  wifiPhyPtrClient->SetMaxSupportedTxSpatialStreams(t_clientNss);
  wifiPhyPtrClient->SetMaxSupportedRxSpatialStreams(t_clientNss);
  NS_LOG_DEBUG ("Channel width " << wifiPhyPtrClient->GetChannelWidth());

}
Ptr<NetDevice> ndAP = apDevice.Get(0);
Ptr<WifiNetDevice> wndAP = ndAP->GetObject<WifiNetDevice>();
Ptr<WifiPhy> wifiPhyPtrAP = wndAP->GetPhy();
uint8_t t_APNss = static_cast<uint8_t>(4); //4 streams (and antennas for server)
wifiPhyPtrAP->SetNumberOfAntennas(t_APNss);
wifiPhyPtrAP->SetMaxSupportedTxSpatialStreams(t_APNss);
wifiPhyPtrAP->SetMaxSupportedRxSpatialStreams(t_APNss);
// Only set the guard interval for HT and VHT modes
//wndServer->GetHeConfiguration()->SetGuardInterval(NanoSeconds (
  serverShortGuardInterval));
//wndClient->GetHeConfiguration()->SetGuardInterval(NanoSeconds (
  clientShortGuardInterval));



/*************************/
/* Plotting data settings */
/*************************/
std::string plotNameThru = "wifi6-network-test-throughput.jpeg";
std::string dataNameThru = "wifi6-network-test-throughput.plt";
std::ofstream outfile (dataNameThru.c_str ());
```

```
//std::string plotNameLat = "wifi6-network-test-latency.jpeg";
//std::string dataNameLat = "wifi6-network-test-throughput.plt";
Gnuplot2dDataset rateData("observed rate");
/**************************************/
/* Traffic creation for data collection */
/**************************************/
//int step = 0
//start schedule events
Simulator::Schedule( Seconds (0.5+stepTime), &AddNodeReportRate, stepTime, rateData);
  // TODO need to get function arguements set up and nodes sorted

double rate = max_ch_width*1e6; //maximum rate for channel
double interval = 8*static_cast<double>(packetSize)/rate;
NS_LOG_INFO("Setting interval to " << interval << " sec for rate of " << rate << "
  bits/sec");


InternetStackHelper stack;
stack.Install (csmaNodes);
stack.Install (wifiApNode);
stack.Install (wifiStaNodes);

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");
Ipv4InterfaceContainer staInterfaces;
Ipv4InterfaceContainer apInterface;
staInterfaces = address.Assign (staDevices);
apInterface = address.Assign (apDevice);

Address serverAddress;
uint16_t base_port = 4000;
for (int i = 0; i < nDev; ++i ){
  serverAddress = Address(staInterfaces.GetAddress(i));
  UdpServerHelper server(base_port+i);

  ApplicationContainer apps = server.Install(wifiStaNodes.Get(i));
  apps.Start (Seconds (0.5));
  apps.Stop (Seconds (30.0));

  uint32_t MaxPacketSize = 1024;
  Time interPacketInterval = Seconds (0.00004); //~25Mbps
  uint32_t maxPacketCount = 0; // unlimited?
  UdpClientHelper client (serverAddress, (base_port+i));
  client.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
  client.SetAttribute ("Interval", TimeValue (interPacketInterval));
  client.SetAttribute ("PacketSize", UintegerValue (MaxPacketSize));
  apps = client.Install(csmaNodes.Get(i+1));
  apps.Start(Seconds(1.0));
  apps.Stop(Seconds(30.0));

}
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
Simulator::Stop(Seconds (30.0)); // simulation total run time

//Config::Connect("/NodeList/*/ApplicationList/*/$ns3::UdpServer/Rx", MakeCallback (&
  PacketRx) ); //Monitor callback

wifiPhy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
wifiPhy.EnablePcapAll("WifiTest");

// Configure the mobility.
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
positionAlloc->Add (Vector (0.0, 0.0, 0.0));
```

```
  NS_LOG_INFO ("Setting initial AP position to " << Vector (0.0, 0.0, 0.0));
  double rad = 3.0;
  double angle = 360 / nDev;
  double x_pos = 0;
  double y_pos = 0;
  for (int i = 0; i < nDev; ++i ) {
    // adding devices to a radius for this test
    x_pos = rad*sin(angle*i*(3.14159/180));
    y_pos = rad*cos(angle*i*(3.14159/180));
    positionAlloc->Add(Vector(x_pos, y_pos, 0.0));
    NS_LOG_INFO("Setting STA position to " << Vector(x_pos, y_pos, 0.0));
  }
  mobility.SetPositionAllocator (positionAlloc);
  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (wifiApNode);
  mobility.Install(wifiStaNodes);

  wifiPhy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);

  /************************************/
  /* Simulation Scheduler and execution */
  /************************************/

  Simulator::Run ();
  Simulator::Destroy ();

  /*****************************************************/
  /* Plot output settings after simulation completion */
  /*****************************************************/
  Gnuplot gnuplot = Gnuplot (plotNameThru);
  gnuplot.AddDataset (rateData);
  gnuplot.SetTerminal ("jpeg enh ");
  gnuplot.SetLegend ("time (ns)", "Rate (Mb/s)");
  gnuplot.SetTitle ("WifiTest");
  gnuplot.AppendExtra ("set key top left");
  gnuplot.GenerateOutput (outfile);
  outfile.close ();

  return 0;

}
```

A.6 wifi-sim-network.cc

```
#include "ns3/command-line.h"
#include "ns3/config.h"
#include "ns3/double.h"
#include "ns3/uinteger.h"
#include "ns3/boolean.h"
#include "ns3/string.h"
#include "ns3/log.h"
#include "ns3/ssid.h"
#include "ns3/callback.h"
#include "ns3/gnuplot.h"
#include "ns3/wifi-standards.h"
#include "ns3/wifi-helper.h"
#include "ns3/wifi-phy.h"
#include "ns3/wifi-net-device.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/yans-wifi-phy.h"
#include "ns3/mobility-helper.h"
#include "ns3/mobility-model.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/propagation-loss-model.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/packet-socket-helper.h"
#include "ns3/packet-socket-client.h"
#include "ns3/packet-socket-server.h"
#include "ns3/wifi-mac-header.h"
#include "ns3/csma-helper.h"
#include "ns3/rng-seed-manager.h"
#include "ns3/point-to-point-helper.h"
```

```
#include "ns3/rectangle.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/udp-echo-helper.h"
#include "ns3/ipv4-global-routing-helper.h"

// Default Network Topology
//
//   Wifi 10.1.3.0
//    -i WiFi nodes-  AP
//  *  ...  *    *    *
//  |       |    |    |    10.1.1.0      -j CSMA devices
// nj+i     nj+2  nj+1 n0 -------------- n1   n2   n3 ... nj
//                       point-to-point  |    |    |      |
//                                       ==================
//                                         LAN 10.1.2.0

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("WifiSimulationNetwork");

double g_intervalBytes = 0;

void Monitor (std::string context, Ptr<const Packet> pkt, uint16_t channel, WifiTxVector
     tx, MpduInfo mpdu, SignalNoiseDbm snr, uint16_t staid)
//Ptr<const Packet>, uint16_t /* frequency (MHz) */, WifiTxVector, MpduInfo,
    SignalNoiseDbm, uint16_t /* STA-ID*/
{
  std::cout << context << std::endl;
  std::cout << "\tChannel :" << channel << " Tx: " << tx.GetMode() << " \tSingal= " <<
    snr.signal <<  "\tNoise= " << snr.noise << "\tCH Width " << tx.GetChannelWidth() <<
    std::endl;
}

void RateChange(){
  std::cout << "Rate Change Occured" << std::endl;
}

void PacketRx(Ptr<const Packet> pkt, const Address &addr){
  //NS_LOG_FUNCTION(addr);
  g_intervalBytes += pkt->GetSize();
}

void AddNodeReportRate ( double step_time,  Gnuplot2dDataset& rateDataset) {
  NS_LOG_FUNCTION (step_time);
  NS_LOG_INFO ("At time " << Simulator::Now().As (Time::S) << " scheduling next
    increment");
  double currentRate = ((g_intervalBytes * 8) / step_time) / 1e6; // Mb/s
  rateDataset.Add (Simulator::Now().GetDouble(), currentRate );
  g_intervalBytes = 0;
  Simulator::Schedule (Seconds(step_time), &AddNodeReportRate, step_time, rateDataset);
}

int main (int argc, char *argv[]) {
  //cmd line arguements
  uint32_t rtsThreshold = 999999;  // disabled even for large A-MPDU
  //std::string phyMode ("DsssRate1Mbps");
  std::string wifiManager("Ideal"); //("MinstrelHt");
  //uint32_t packetSize = 1024;
  //double stepTime = 0.5;
  //uint8_t steps = 20;
  bool verbose = false;
  uint32_t nCsma = 3;
  uint32_t nWifi = 10;
  uint32_t nPackets = 100;
  bool tracing = true;
  uint32_t maxSlrc = 7;
  uint32_t maxSsrc = 7;
  uint16_t serverNss = 1;
  uint16_t clientNss = 1;

  CommandLine cmd (__FILE__);
  cmd.AddValue ("wifiManager", "Set wifi rate manager (Aarf, Aarfcd, Amrr, Arf, Cara,
    Ideal, Minstrel, MinstrelHt, Onoe, Rraa, ThompsonSampling)", wifiManager);
```

```
cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", verbose);
cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
cmd.AddValue ("nPackets", "Number of packets to echo", nPackets);
cmd.AddValue ("tracing", "Enable pcap tracing", tracing);
cmd.Parse (argc, argv);

/*****************************************/
/* WIFI Standards and setting configuration */
/*****************************************/
NS_LOG_INFO("Configuring WiFi Parameters");
WifiHelper wifi;
if (verbose)
{
  wifi.EnableLogComponents ();  // Turn on all Wifi logging
}

Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (maxSlrc))
  ;
Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (maxSsrc))
  ;
Config::SetDefault ("ns3::MinstrelWifiManager::PrintStats", BooleanValue (true));
Config::SetDefault ("ns3::MinstrelWifiManager::PrintSamples", BooleanValue (true));
Config::SetDefault ("ns3::MinstrelHtWifiManager::PrintStats", BooleanValue (true));
// set the wifi standard
wifi.SetStandard (WIFI_STANDARD_80211ax_5GHZ);
YansWifiPhyHelper wifiPhy;
// get the maximum channel width
uint16_t ch_width = GetDefaultChannelWidth(WIFI_PHY_STANDARD_80211ax,
  WIFI_PHY_BAND_5GHZ);
//uint16_t max_ch_width = GetMaximumChannelWidth(WIFI_PHY_STANDARD_80211ax);

uint32_t channelRateFactor = ch_width / 20;
channelRateFactor = channelRateFactor * std::max (clientNss, serverNss);

//Channel
Ptr<YansWifiChannel> wifiChannel = CreateObject<YansWifiChannel> ();
// Add propagation delay model to channel
// TODO could use <ConstantSpeedPropagationDelayModel>, <RandomPropagationDelayModel>
Ptr<ConstantSpeedPropagationDelayModel> delayModel = CreateObject<
  ConstantSpeedPropagationDelayModel> ();
wifiChannel->SetPropagationDelayModel (delayModel);
// Add Recieve Signal Strength (RSS) Loss Model
// TODO decide Model: <FixedRssLossModel>, <RangePropagationLossModel>,
// <NakagamiPropagationLossModel>, <LogDistancePropagationLossModel>,
// <ThreeLogDistancePropagationLossModel>, <TwoRayGroundPropagationLossModel>,
// <FriisPropagationLossModel>, <RandomPropagationLossModel>
Ptr<RangePropagationLossModel> rssLossModel = CreateObject<RangePropagationLossModel>
  ();
wifiChannel->SetPropagationLossModel (rssLossModel);
// PHY
wifiPhy.SetChannel(wifiChannel);
//std::cout << "Default Channel Width: " << ch_width << std::endl;
//std::cout << "Maximum Channel Width: " << max_ch_width << std::endl;
wifi.SetRemoteStationManager("ns3::" + wifiManager + "WifiManager", "RtsCtsThreshold",
   UintegerValue (rtsThreshold));

/*************************************/
/* Node and Network Topology creation */
/*************************************/
// Nodes to be used
NS_LOG_INFO("Creating Client Server Nodes");
//Connection from AP to 'internet'
NodeContainer p2pNodes;
p2pNodes.Create (2);

//Pont to point conenction from our LAN to outside CSMA network
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("10Gbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
// AP and ISP connection to some arbitrary network
NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

```
//Connection from 'ISP' to servers
NodeContainer csmaNodes;
//CSMA node connected to Wifi Router
csmaNodes.Add (p2pNodes.Get (1)); //P2P node 1 reepresents ISP
// nodes past ISP
csmaNodes.Create (nCsma);

CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("1Gbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

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

//TODO create mobile and stationary STA nodes for better simulation
// creation of Local Area Nodes
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi); // Stations ie local hosts
NodeContainer wifiApNode = p2pNodes.Get (0); //Access Point

// MAC setup associated with nodes
// AP MAC and PHY setup
WifiMacHelper wifiMac;
Ssid ssid = Ssid ("LocalNetwork");
wifiMac.SetType("ns3::ApWifiMac", "Ssid", SsidValue (ssid));
wifiPhy.Set ("ChannelWidth", UintegerValue (ch_width));
NetDeviceContainer apDevice;
apDevice = wifi.Install(wifiPhy, wifiMac, wifiApNode); //assocate the various setting

// STA MAC and PHY setup
wifiMac.SetType("ns3::StaWifiMac", "Ssid", SsidValue (ssid));
// TODO Lets look into Channel width parameters (max_ch_width)
wifiPhy.Set ("ChannelWidth", UintegerValue (ch_width));
NetDeviceContainer staDevices;
staDevices = wifi.Install(wifiPhy, wifiMac, wifiStaNodes);

RngSeedManager::SetSeed (1);
RngSeedManager::SetRun (2);
wifi.AssignStreams (staDevices, 100);
wifi.AssignStreams (apDevice, 100);


/****************/
/* PHY settings */
/****************/
// Perform post-install configuration from defaults for channel width,
// guard interval, and nss, if necessary
// Obtain pointer to the WifiPhy
int nDev = staDevices.GetN();
for (int i = 0; i < nDev; ++i ){
  // set PHY parameters for each STA
  Ptr<NetDevice> ndClient = staDevices.Get(i);
  Ptr<WifiNetDevice> wndClient = ndClient->GetObject<WifiNetDevice>();
  Ptr<WifiPhy> wifiPhyPtrClient = wndClient->GetPhy();
  uint8_t t_clientNss = static_cast<uint8_t>(1); //1-4 are acceptable
  wifiPhyPtrClient->SetNumberOfAntennas(t_clientNss);
  wifiPhyPtrClient->SetMaxSupportedTxSpatialStreams(t_clientNss);
  wifiPhyPtrClient->SetMaxSupportedRxSpatialStreams(t_clientNss);
  NS_LOG_DEBUG ("Channel width " << wifiPhyPtrClient->GetChannelWidth());

}
Ptr<NetDevice> ndAP = apDevice.Get(0);
Ptr<WifiNetDevice> wndAP = ndAP->GetObject<WifiNetDevice>();
Ptr<WifiPhy> wifiPhyPtrAP = wndAP->GetPhy();
uint8_t t_APNss = static_cast<uint8_t>(4); //4 streams (and antennas for server)
wifiPhyPtrAP->SetNumberOfAntennas(t_APNss);
wifiPhyPtrAP->SetMaxSupportedTxSpatialStreams(t_APNss);
wifiPhyPtrAP->SetMaxSupportedRxSpatialStreams(t_APNss);


/**************************************/
/* Traffic creation for data collection */
```

```
/**************************************/
//int step = 0
//start schedule events
//Simulator::Schedule( Seconds (0.5+stepTime), &AddNodeReportRate, stepTime, rateData)
  ; // TODO need to get function arguements set up and nodes sorted

wifiPhy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11);
wifiPhy.EnablePcapAll("WifiSimulationNetwork_WPHY");
//pointToPoint.EnablePcapAll ("WifiSimulationNetwork_P2P");
//csma.EnablePcapAll ("WifiSimulationNetwork_CSMA", csmaDevices.Get (0), true);

// Configure the mobility.
MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                               "MinX", DoubleValue (0.0),
                               "MinY", DoubleValue (0.0),
                               "DeltaX", DoubleValue (1.0),
                               "DeltaY", DoubleValue (1.0),
                               "GridWidth", UintegerValue (3),
                               "LayoutType", StringValue ("RowFirst"));
// bound it to the area of a home
mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
                           "Bounds", RectangleValue (Rectangle (-5, 5, -10, 10)));
mobility.Install (wifiStaNodes);

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

InternetStackHelper stack;
stack.Install (csmaNodes);
stack.Install (wifiApNode);
stack.Install (wifiStaNodes);

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 (apDevice);

UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
serverApps.Start (Seconds (0.5));
serverApps.Stop (Seconds (30.0));

UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (10000000));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (4096));

ApplicationContainer clientApps = echoClient.Install (wifiStaNodes);//.Get (nWifi - 1)
  );
clientApps.Start (Seconds (0.5));
clientApps.Stop (Seconds (30.0));

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();


//Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/MonitorSnifferRx",
   MakeCallback(&Monitor)); //Monitor callback
//throws error
//Config::ConnectWithoutContext ("/NodeList/0/DeviceList/*/$ns3::WifiNetDevice/
  RemoteStationManager/$ns3::" + wifiManager + "WifiManager/Rate", MakeCallback (&
  RateChange));
```

```
  Simulator::Stop(Seconds (30.0)); // simulation total run time

  /************************************/
  /* Simulation Scheduler and execution */
  /************************************/
  //Simulator::Schedule( Seconds (0.5+stepTime), &AddNodeReportRate, /*args*/);
  //Simulator::Stop (Seconds ((steps + 1) * stepTime));
  Simulator::Run ();
  Simulator::Destroy ();



  return 0;

}
```

B Previous Project Work

The original project goal was to simulate Mulit-path TCP utilizing a $ns-3$ library that had yet to be adopted into releases. Due to the lack of inclusion in more recent versions, and the inability to get the $ns-3$ patch files to run without any issues, this project was not pursued any further. The following includes an interim report of the work conducted on the previous project idea.

ENSC833:
Network Performance and Protocols
Spring 2022

Interim Report

Jason Epp
Yanyan Zhang
Rowen Jiang
March 13, 2022

I Project Introduction

Multipath TCP (MPTCP) is a protocol that utilizes multiple network interfaces, such as Wi-Fi, and Mobile networks simultaneously to balance the load on each interface creating a reliable connection. This is also beneficial for smooth handover from one network to another when moving between Wi-Fi and mobile connections by creating a reliable connection to one network before closing the other. With the additional bandwidth and MPTCP should be able to provide reliable low latency, high bandwidth video streaming to users by balancing loads over different networks and having redundant data links in case of interruptions of service. This project implements a MPTCP with ns-3 and evaluated the performance of real-time video transmission in terms of latency, bandwidth, and reliability. The expectation is that MPTCP should have a better performance compared to TCP in video streaming. This project is intended to show illustrations of the performance improvement.

Our work has been divided into two parts:

1. Get the source code built successfully and run example code bug-free.
2. Compose a test environment and list out simulation parameters as desired.

II Project Code Candidates

Original Code candidate:

1. Github repo: `https://github.com/mkheirkhah/mptcp`
2. Support page: `https://groups.google.com/g/mptcp/c/YYSWVnpzXS8`

This was the original repository that had been commonly referenced for Multi-Path TCP. This appeared to be well referenced by our other references selected for the project, and had a number of starts a forks which looked promising. It appears that there was no further work done past 2015 which became an area of concern. In the initial attempts at building it was found that this would not successfully build in Ubuntu 18.04, or 20.04, though it appears to have been working in Ubuntu 16.04 according to the README, as stated by the repository, but was based on work done in 2015, so it was likely too far out of support for more recent versions of ns-3.

Alternative Patch repository:

1. Github Repository: `https://github.com/teto/ns-3-dev-git/tree/mptcp_options`
2. NSNAM Wiki Page: `https://www.nsnam.org/wiki/GSOC2015MpTcpImplementation`

This was generally looking to be poorly supported regardless of being reviewed for a candidate, not much time spent working with this repository, as it had also not been updated since 2015. Therefor this wasn't considered a good candidate for the project.

Selected candidate for patch:

1. Github repo:
   `https://github.com/Kashif-Nadeem/ns-3-mptcp`

2. Alternative Github repo (forked from nsnam):
   `https://github.com/Kashif-Nadeem/ns-3-dev-git`

3. Official Code Review Page:
   `https://codereview.appspot.com/369810043/#ps90001`

4. NSAM Currentl Development Wiki Listing:
   `https://www.nsnam.org/wiki/Current_Development#Multipath_TCP_for_ns-3.29_and_ns-3.8`

5. ns-3-reviews Google group:
   `https://groups.google.com/g/ns-3-reviews/c/wtgm2E5yIU8/m/emHDjv9OBAAJ`

6. mptcp example network:
   `https://groups.google.com/g/ns-3-reviews/c/wtgm2E5yIU8/m/emHDjv9OBAAJ`

7. mptcp textbook chapter:
   `https://www.springerprofessional.de/en/an-ns-3-mptcp-implementation/16532910`

This candidate was ultimately selected as the candidate we would use to test Multipath TCP in ns-3, it appeared to be an official candidate for the the ns-3 mptcp implementation, it was currently going through code review, though it hadn't been updated for over a year. There was a google support group created for those reviewing and testing the code, many people reported positive results with the example, though there were still some bugs within some topology. And this was referenced within a text where the author had published a chapter in relation to the project. This was able to build successfully with a little work on the part of the group, some things required included ensuring the correct openssl, and libssl-dev packages were installed, and that python 2.7 was installed in the Linux environment. Once it was building further work was done to try and see how the code works or track down an example. An example script that had been used in the testing is available in the links above.

III Build ns-3-mptcp

We tried several versions of ns-3 combined with different versions of python and finally get mptcp model build. We started with ns-3.35 and python 3.8 and python 2.7 as this was installed after the first assignment. The repository supplied a code pack with ns-3.29, so we also tried ns-3.29 with python 2.7 and python 3.8. We can only get it to built with the provided code which is with ns-3.29 combined with python 2.7. All the other combination would give fetal error in the building progress.

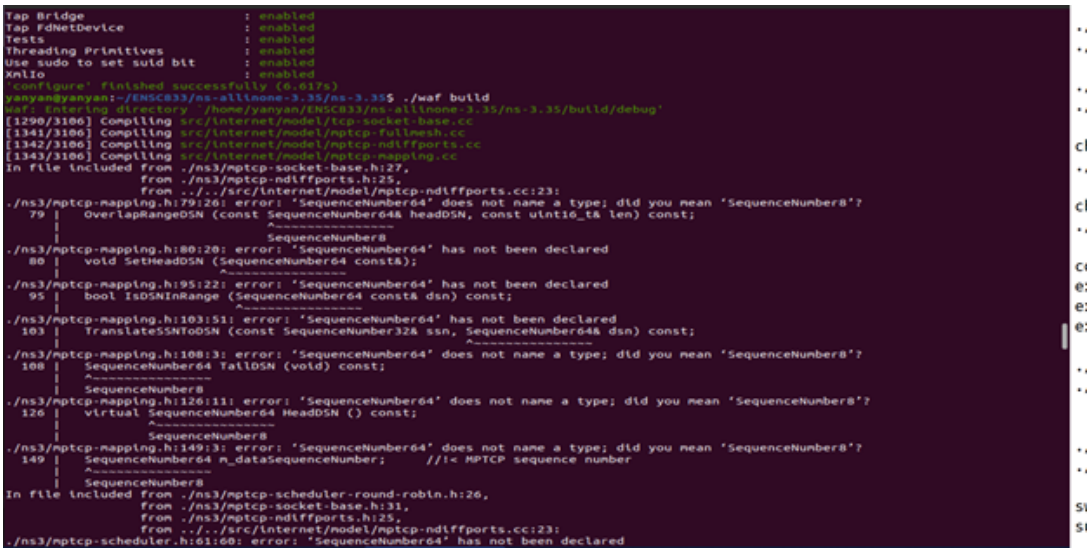


Figure 26: Example of Build Fail with version combination: ns-3.35 with python 2.7

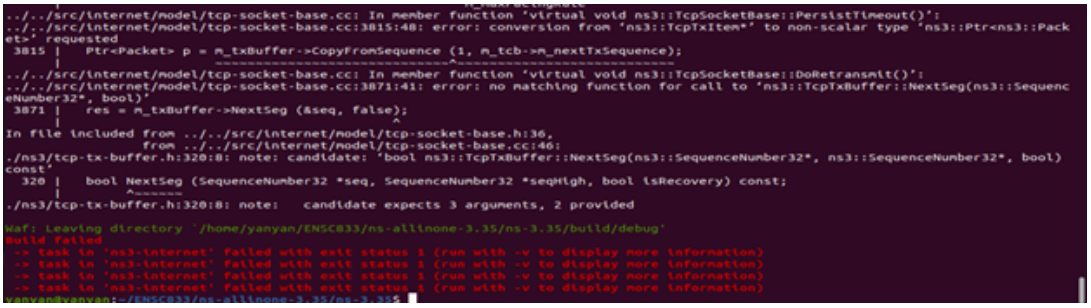


Figure 27: Example of Build Fail with version combination: ns-3.35 with python 2.7 (2)

IV Project Debugging

Run Example code

It was when testing this that it was ultimately unsuccessful in implementing this project any further.When working with this project there were a number of small obstacles to overcome with regards to these repositories. For the first repository it was found that it was difficult to build and that nobody in the group was successful. But then some alternative projects were found once reviewing the literature. This is when the version from Kashif Nadeem was selected, as it seemed like the most viable candidate. Initially when trying to build this repository there were some difficulties one of the original difficulties was that by default ns-3 considers warnings as errors as configured in the compiler. To fix this issue during build the following command had to be run to complete the build.

```
CXXFLAGS="-Wall" ./waf configure build
```

Once this was discovered, and all of the required openssl libraries were downloaded from the apt repository, the project would build successfully.

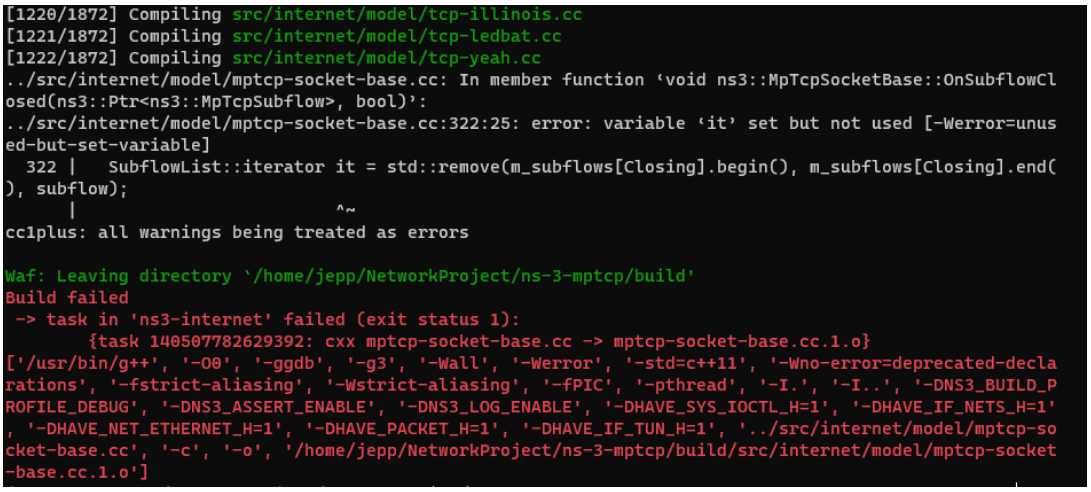


Figure 28: Build failure associated with selected code project

Once the project was building confidence grew in the ability to use the project, and the next stage of familiarization and exploration were undertaken by going through the code reviews, and the implementation. An example file was available as part of the code reviews, so next step was to get the example running. Many people had reported positively that the example worked, but that specifically a leaf spline topology was causing issues. This was considered to be low risk as long as we did not try to

utilize that topology within our testing. Once the example was tested it was noted that nobody could successfully run the code.

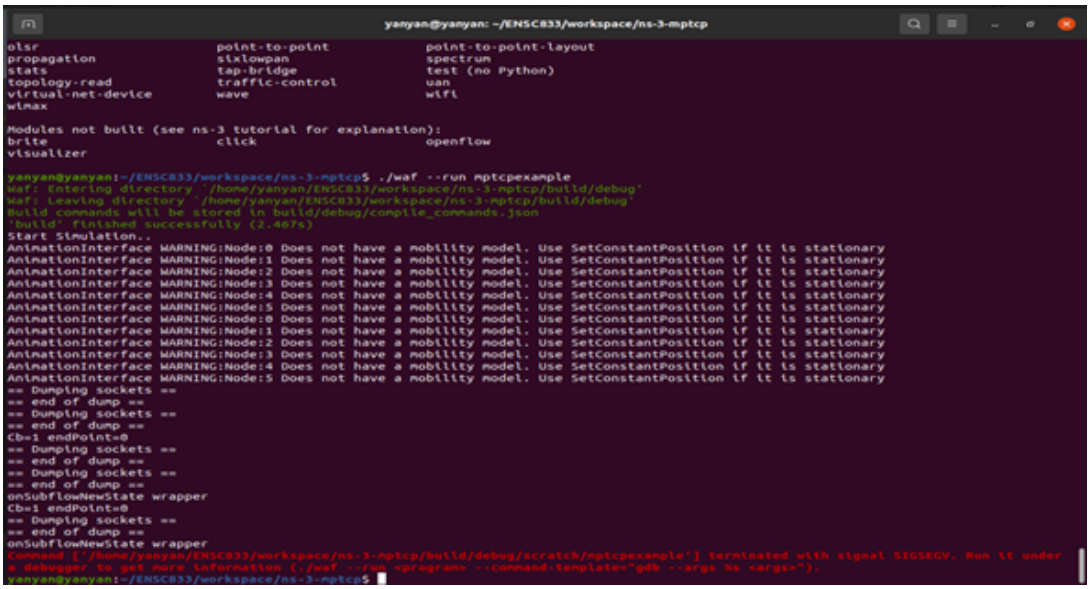


Figure 29: Run result with mptcp-example.cc

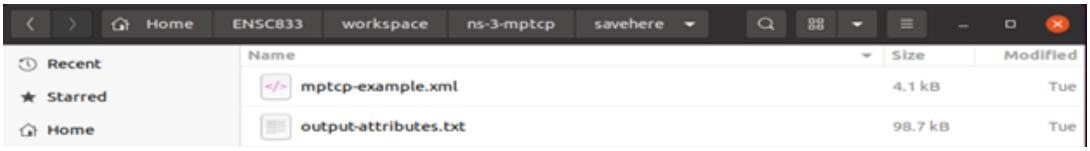


Figure 30: Two files were generated after Run

Figure 31: Detail of the log information

Deep Debugging

Through some debugging and creative print outputs, the location of the failure was debugged. With this it was found that the system was failing during the run command which is where all of the simulation was executed, so further investigation into the changes was required. The first step was to run again in gdb to see if further details could be seen. gdb debugging was executed with the following command

```
./waf --run "mptcp-example" --command-template="gdb %s"
```

From this we could see further that there was a segmentation fault during the running of the simulation, and it appeared to be associated with TCP headers and likely due to the the MPTCP changes. See Figure 32 for initial failure mode output. Once the failure was seen a backtrace was also performed in gdb to insect the call stack and see the nature of the fail (See Figure 33)

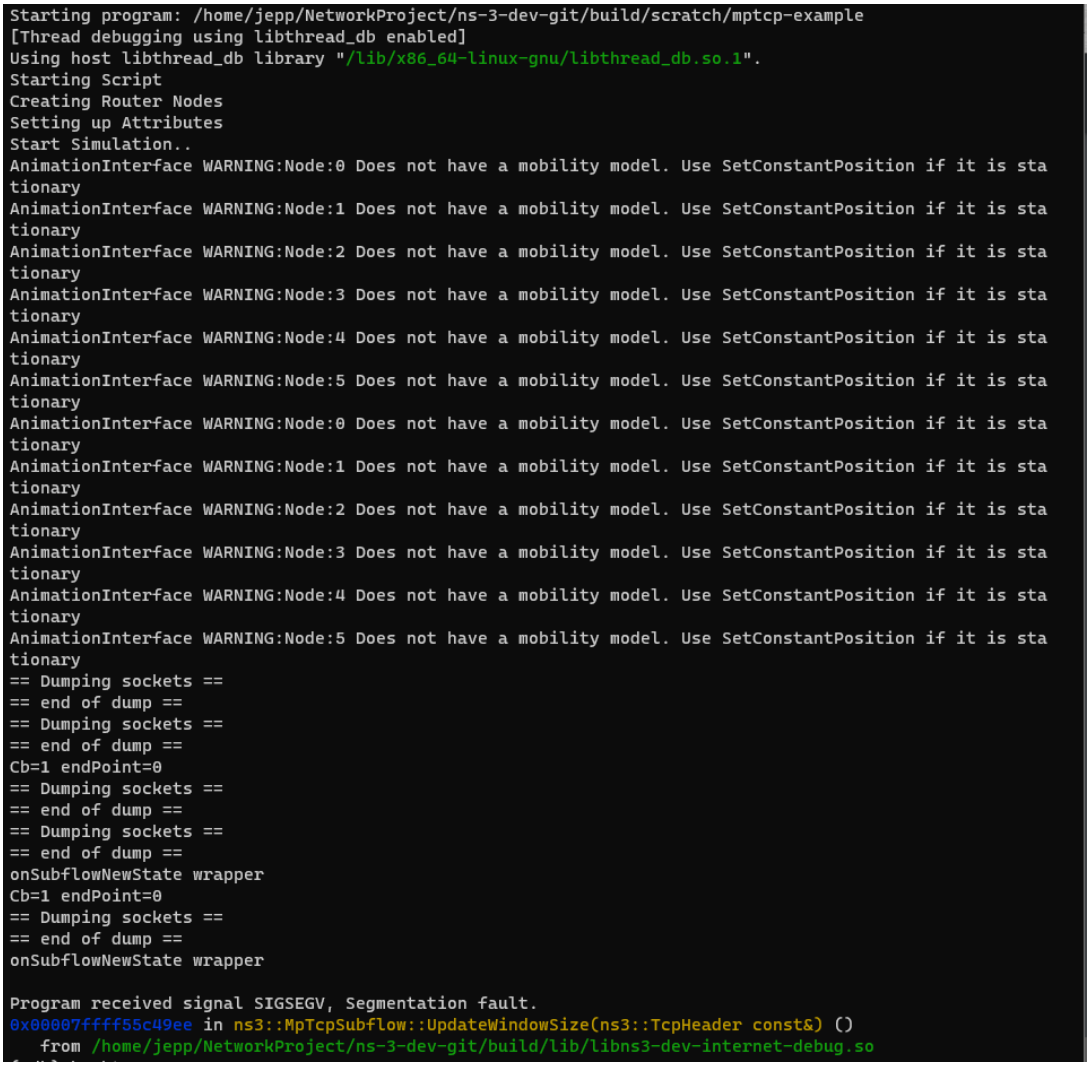


Figure 32: Output of example script, with failure as seen in gdb

```
(gdb) backtrace
#0  0x00007ffff55c49ee in ns3::MpTcpSubflow::UpdateWindowSize(ns3::TcpHeader const&) ()
   from /home/jepp/NetworkProject/ns-3-dev-git/build/lib/libns3-dev-internet-debug.so
#1  0x00007ffff54d1d40 in ns3::TcpSocketBase::DoForwardUp(ns3::Ptr<ns3::Packet>, ns3::Address const&,
ns3::Address const&) ()
   from /home/jepp/NetworkProject/ns-3-dev-git/build/lib/libns3-dev-internet-debug.so
#2  0x00007ffff54cf38d in ns3::TcpSocketBase::ForwardUp(ns3::Ptr<ns3::Packet>, ns3::Ipv4Header, unsign
ed short, ns3::Ptr<ns3::Ipv4Interface>) ()
   from /home/jepp/NetworkProject/ns-3-dev-git/build/lib/libns3-dev-internet-debug.so
#3  0x00007ffff55c0d3a in ns3::MpTcpSubflow::ForwardUp(ns3::Ptr<ns3::Packet>, ns3::Ipv4Header, unsigne
d short, ns3::Ptr<ns3::Ipv4Interface>) ()
   from /home/jepp/NetworkProject/ns-3-dev-git/build/lib/libns3-dev-internet-debug.so
#4  0x00007ffff552956a in ns3::MemPtrCallbackImpl<ns3::Ptr<ns3::TcpSocketBase>, void (ns3::TcpSocketBa
se::*)(ns3::Ptr<ns3::Packet>, ns3::Ipv4Header, unsigned short, ns3::Ptr<ns3::Ipv4Interface>), void, ns
3::Ptr<ns3::Packet>, ns3::Ipv4Header, unsigned short, ns3::Ptr<ns3::Ipv4Interface>, ns3::empty, ns3::e
mpty, ns3::empty, ns3::empty, ns3::empty>::operator()(ns3::Ptr<ns3::Packet>, ns3::Ipv4Header, unsigned
 short, ns3::Ptr<ns3::Ipv4Interface>) ()
   from /home/jepp/NetworkProject/ns-3-dev-git/build/lib/libns3-dev-internet-debug.so
```

Figure 33: Backtrace of failure in gdb

Further debugging was performed with more print outputs in the script, as well as some tests were done between Ubuntu 18.04 native install, and Ubuntu 20.04 in a WSL container in Windows, as well as a Ubuntu 18.04 container within windows for comparison between the behaviour of Ubuntu 18.04. Within some of these changes the failure mode was generally the same but some further details were visible when using the Ubuntu 20.04 distribution. With this it was found that it was convenient to use the Ubuntu 20.04 WSL and that the failure performed the same but there was a little more information available when using Ubuntu 20.04. When debugging it was found the failure happened in the exact same spot between both Linux distributions. To debug further some code was added in the project and it was found that it always experienced a segmentation fault after the second printout before third printout. See Figures 34, 35 or code changes and corresponding output. Also it was noted on the 20.04 distribution that it looked like the segmentation fault was occurring during a deconstructor call to MpTcpSocketBase class. this made little sense as this should not happen until the base class goes out of context but the window size was not yet being printed in to the terminal as seen in Figure 35. It was at this point that it became difficult to debug further, and it appeared that debug symbols were not available in gdb when running the script, not allowing us to have further information into the failure mode. Some various methods to configure the simulator configuration as debug were attempted but yielded no better results. At this point it is being decided that due to the difficult nature of this failure and no response yet from the repository creator that changing the project may be the best course of action.

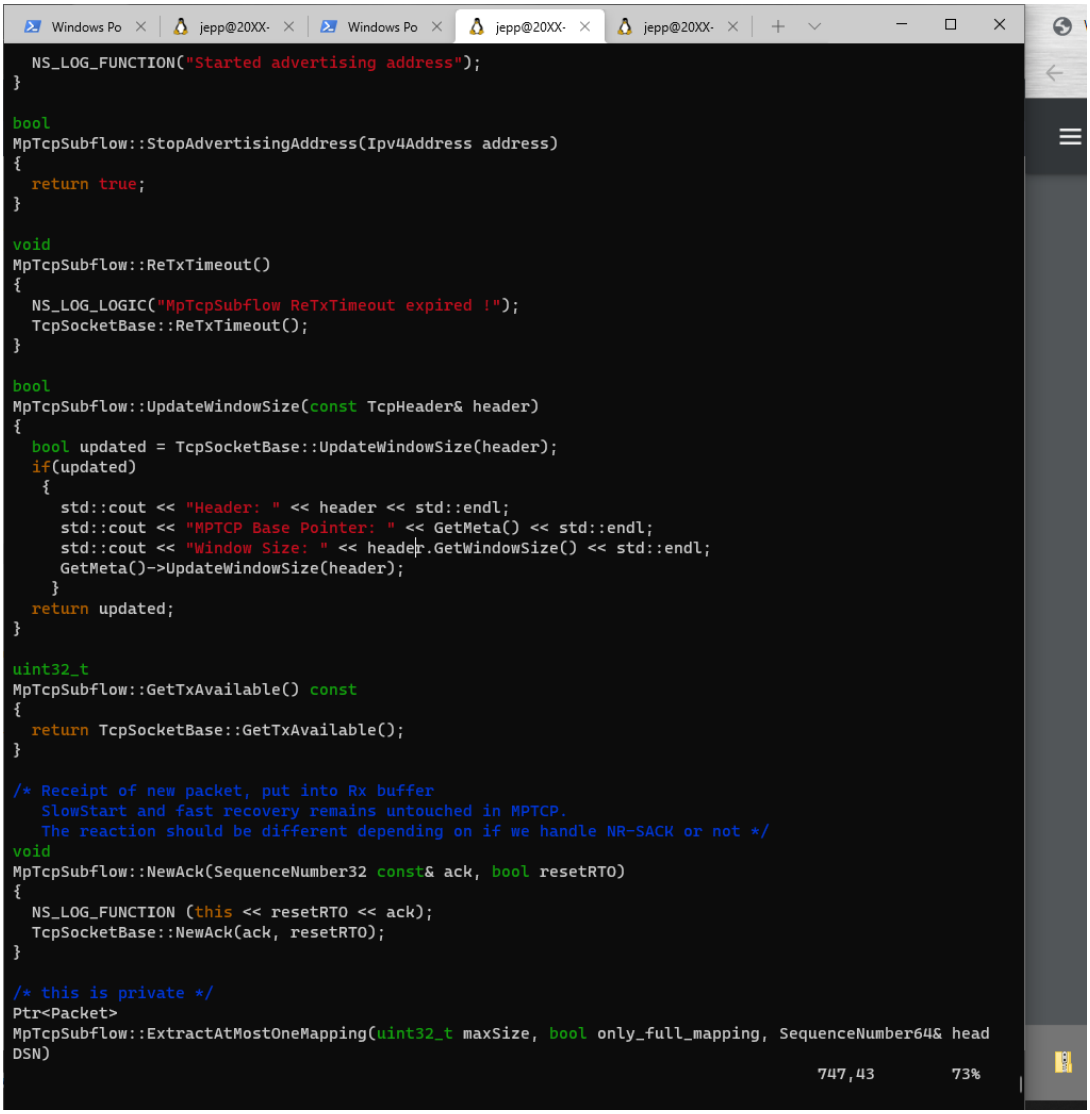


Figure 34: Debug outputs added to the code to pinpoint the failure

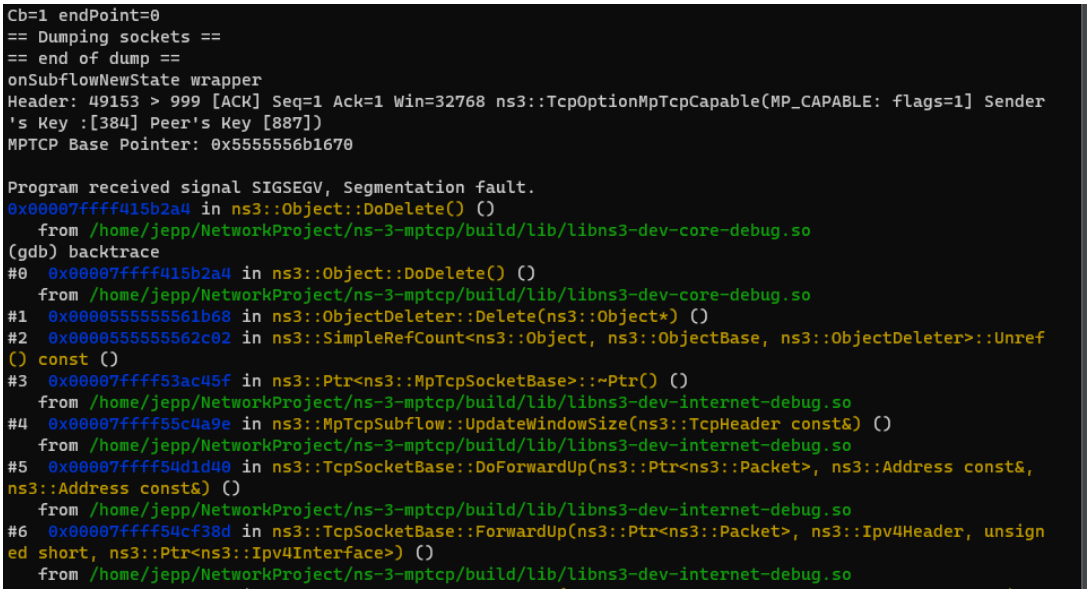


Figure 35: Debug outputs added to the code to pinpoint the failure

V Project Status

Since we started the project, we have spent time on finding the references, supported source code, and consult as much documents as we could to get the first stage of out project done. We have seen that future work are doable but will be of the interest of debugging the code build instead of simulation and get the learning experience for this course.