TECHNICAL UNIVERSITY OF CRETE
ELECTRICAL AND COMPUTER
ENGINEERING DEPARTMENT
TELECOMMUNICATIONS DIVISION

Network Modeling and Quality of Service (QoS) in NS-3

by

Antonios Andreadakis

A THESIS SUBMITTED IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS
FOR THE DIPLOMA THESIS OF
ELECTRICAL AND COMPUTER
ENGINEERING

October 2023

THESIS COMMITTEE

Professor Aggelos Bletsas, *Thesis Supervisor*

Professor George N. Karystinos

Associate Prof. Vasileios V. Samoladas

Abstract

This thesis offers a concrete guide on how to setup and program communication network simulations in network simulator 3 (NS-3). First, the architecture of the simulator is described and its basic components. Then, a detailed guide is offered based on three example networks, with increasing complexity. The examples include both wired and wireless networking scenarios, using throughput (in bps) as the quality of service (QoS) metric. Subsequently, a case study of energy efficiency maximization in the user association problem for heterogeneous LTE networks is performed, comparing three state-of-the-art algorithms. Empirical cumulative distribution functions for spectral efficiency (in bps/Hz) and energy efficiency (in bits/Hz/Joule) are obtained and compared. NS-3 is capable of simulating complex scenarios, with however, a steep learning curve.

Thesis Supervisor: Professor Aggelos Bletsas

Acknowledgements

At first, I would like to express my deeply appreciation and gratitude for my supervisor Prof. A. Bletsas whose guidance, encouragement, patience and support were outstanding throughout this study. Thanks to you I have enriched my knowledge in many aspects.

Of course, I want to offer my special thanks to my friends and colleagues for all the memories we shared.

Also, I am deeply grateful for the continuous and priceless support, forbearance and love of my family for all those years during my studies.

Table of Contents

List of figures

Chapter 1: Introduction & Problem Description

Nowadays is the information age. People own many devices of different technologies that offer interconnection to other people and devices. The basic tool offered in wireless communications is mobile phone and is a reference point serving a wide range of services. Today's smartphones can be used in both voice and video calls, access the internet, either music or video or games entertainment, data exchange etc. Rapid growth of data traffic and improvement of capacity in order to serve the demands of users, contribute to global energy consumption, so it is necessary to focus in optimizing the energy efficiency in a thrifty and environmental approach.

A cellular network is a telecommunications network that provides wireless communication services to mobile devices, such as smartphones, tablets, and laptops. It is a complex system of interconnected cells or base stations that cover a specific geographic area. Cells are responsible for transmitting and receiving signals to and from mobile devices within their coverage areas. Cellular networks play a vital role in modern telecommunications, enabling people to stay connected while on the move and facilitating a wide range of applications, from voice calls and text messaging to mobile internet access, mobile apps, and the Internet of Things (IoT).

5^{th} and 6^{th} generation (5G/6G) mobile communications are discussed. In fact, the attempt to combine heterogeneous technologies (GPRS, EDGE, HSPA, LTE, WiFi etc), so that mobiles use the most efficient solution to achieve it. A Heterogeneous Network (HetNet) is a type of network architecture consisted of multiple types of wireless access technologies and different types of network nodes. In a HetNet, various network elements, such as macrocells, microcells, picocells, and femtocells are deployed together providing wireless coverage and capacity in a more efficient and flexible manner.

Any new technology for commercial systems must first be tested in simulated environments to determine and correct any problems that arise. Most common simulation environment is Network Simulator 3. A discrete simulator of facts accepted by the academic and research community. In this thesis, versions 3.31 (June 2020), 3.33 (January 2021) and 3.35 (October 2021) along with python 3 scripting language are used. In this context, present work tests utilization of NS-3 in research for a modern version of the classic user association problem i.e., associating users to base stations, while maximizing energy efficiency in downlink direction. That is, how users can be optimally allocated while saving energy and maintaining good performance in the network.

Six chapters complete this work. Chapter [2] introduces the Network Simulator 3 along with some of its applications and provide a short guide of installation. In Chapter [3], there is a detailed scripting guide and anyone following those instructions succeeds to setup a simple or advanced script, as well as three examples of increasing complexity. Chapter [4] describes the modules used and the implementation. The results occurred and analysis of them is introduced in Chapter [5]. At last, Chapter [6] consists of the conclusions and future work.

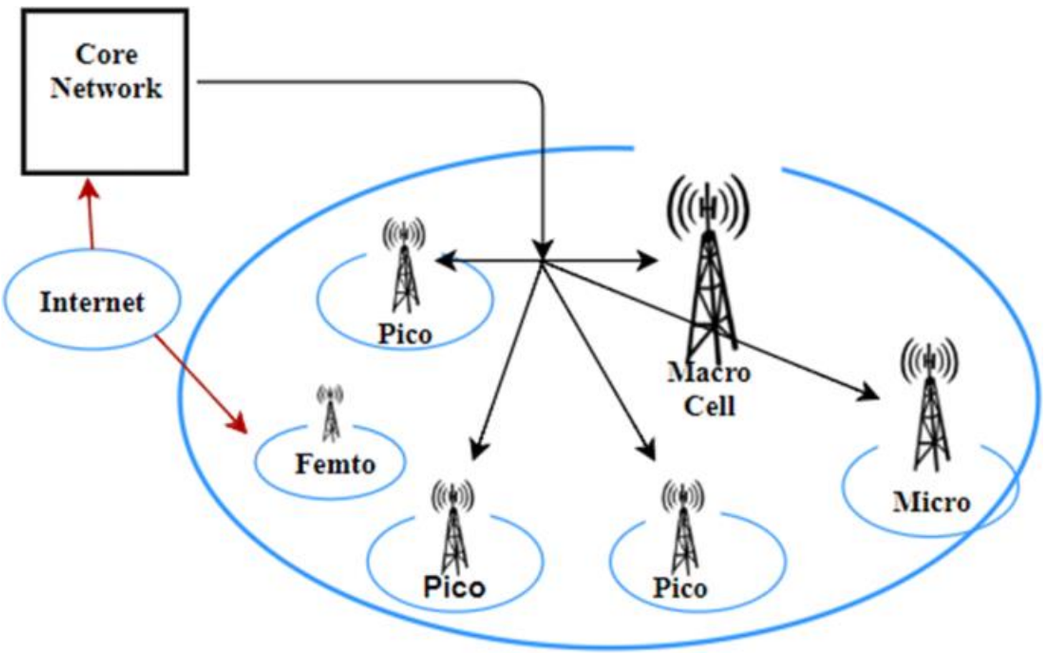

Figure 1.1 - Graphical representation of a hetnet, Analysis of Acquired Indoor LTE-A Data from an Actual HetNet Cellular Deployment | SpringerLink [1]

Chapter 2: Overview of Network Simulator 3

Introduction of Network Simulator 3 and ways to program a simulation script, using its modules is described. A brief explanation of the architecture and its modules if offered in the main document, as well as an installation guide in the Appendix.

2.1 Introduction to NS-3

Network Simulator 3 is a Linux based framework, built as a system of libraries that work together between them or any external library. It has minimal prerequisites for most basic installation, such as C++ compiler, python3 support, CMake build system and at least one of make, ninja or Xcode build systems. Only flaw is the lack of a window app offering debugging, execution etc., which is common in other simulators and user works with the command line for every attempt to execute or debug the script.

- The Network Simulator is a free software using GNU GPLv2 license and is used for research and educational purposes. Early version of network simulator known as ns-1, roots back in 1995 and was developed in Lawrence Berkley National Laboratory. It was known as LBNL Network Simulator and comes from the old simulator named REAL of S. Keshav. Its core was written in C++ and simulations were written in Tcl scripts. It is no longer used.
- Then, a newer version of network simulator was introduced in 1996. The core remained in C++, but the Tcl was replaced by Object Tcl of MIT. No longer supported and is not acceptable for scientific publication since 2009.
- At last, new simulator does not have compatibility with ns-2. The development began in 2006 exclusively in C++ and later a framework was added for Python bindings and usage of <waf build system>. First version 3.1 was published in 2008 and the project continued to update every 3 months. Latest version is 3.40 (September 2023).

A wide range of known networks and models is implemented in NS-3. Its core supports IP and non-IP network protocols. Most research focus on wireless and IP simulations including models like Wi-Fi, WiMAX, LTE and a variety of static or dynamic routing protocols. Simulation time fluctuates from seconds to days depending on the needs of the simulation and availability of resources, mostly the processing power.

It includes a real-time scheduler that facilitates the interaction of “simulations in the magnifying glass” use cases with real systems. For example, users can both transmit and receive packets from ns-3 to real network devices. Also works as a virtual machine interface network. Provides functionality which gives opportunity at the results to interact with other open-source tools for further analysis, like Wireshark, NetAnim, gnuPlot etc.

Many users focus on implemented models for their simulations and export results. Existing models might not match with their purpose or be missing some functionality required. So, modifying the existing models or developing one from scratch is a solution. In our case, a detailed

search took place for the appropriate models and understand the classes and usage, along with solving the problems occurred during the implementation of our thesis.

2.2 Applications of NS-3

Specific tool is mostly used for educational and research purposes, so it is an important choice for students to work with in the context of preparing a diploma thesis and many other fields of telecommunications, networking and energy. Plenty of support forums for solving questions and troubleshooting exist. Manual is very detailed and users can get familiar with it. A brief reference of some models used in Ns-3 follows.

2.2.1 Wi-Fi Module

WifiNetDevice models a wireless network interface controller based on the IEEE 802.11 standard [ieee80211]. In brief ns-3 provides models for aspects of 802.11:

- basic 802.11 DCF with infrastructure and adhoc modes
- 802.11a, 802.11b, 802.11g, 802.11n (both 2.4 and 5 GHz bands), 802.11ac and 802.11ax (both 2.4 and 5 GHz bands) physical layers
- MSDU aggregation and MPDU aggregation extensions of 802.11n, and both can be combined together (two level aggregation)
- QoS-based EDCA and queueing extensions of 802.11e
- the ability to use different propagation loss models and propagation delay models, please see the chapter on Propagation for more detail various rate control algorithms including Aarf, Arf, Cara, Onoe, Rraa, ConstantRate, and Minstrel
- 802.11s (mesh), described in another chapter
- 802.11p and WAVE (vehicular), described in another chapter of manual

The set of 802.11 models provided in NS-3 attempts to describe an accurate MAC-level implementation of the 802.11 specification and a packet-level abstraction of the PHY-level for different PHYs, corresponding to 802.11a/b/e/g/n/ac/ax specifications.

In NS-3, nodes can have multiple WifiNetDevices on separate channels, and the WifiNetDevice is able to coexist with other device types. With the use of the SpectrumWifiPhy framework, one can also build scenarios involving cross-channel interference or multiple wireless technologies on a single channel. The source code for the WifiNetDevice and its models lives in the directory src/wifi. Implementation is modular and provides roughly three sublayers of models:

- the PHY layer models
- the so-called MAC low models: they model functions such as medium access (DCF and EDCA), RTS/CTS and ACK. In ns-3, the lower-level MAC is further subdivided into a MAC low and MAC middle sub layering, with channel access grouped into the MAC middle.

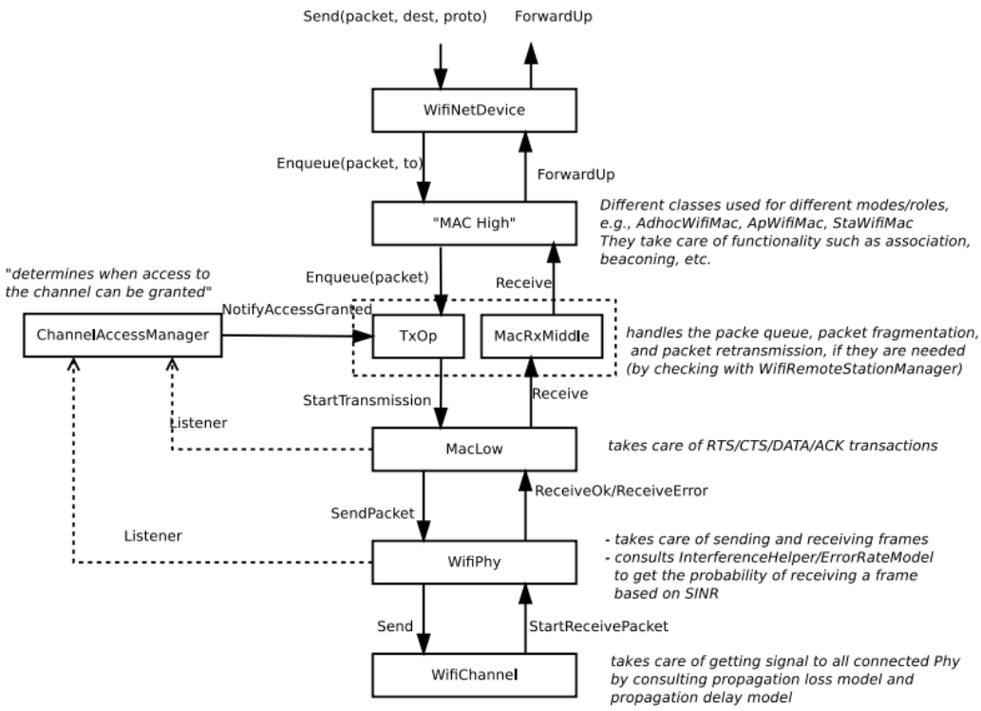

Figure 2.2.1 - Overview of WiFi model [2]

- the so-called MAC high models implement non-time-critical processes in Wifi such as the MAC-level beacon generation, probing, and association state machines, and a set of Rate control algorithms. In literature, this sublayer is sometimes called the upper MAC and consists of more software-oriented implementations vs. time-critical hardware implementations.

2.2.2 WiMAX Module

By adding WimaxNetDevice objects to ns3 nodes, one can create models of 802.16-based networks. Below, we list some more details about what the ns-3 WiMAX models cover but, in summary, the most important features of the ns-3 model are:

- a scalable and realistic physical layer and channel model
- a packet classifier for the IP convergence sublayer
- efficient uplink and downlink schedulers
- support for Multicast and Broadcast Service (MBS), and
- packet tracing functionality

The source code for the WiMAX models lives in the directory src/wimax. There have been two academic papers published on this model [3] & [4]

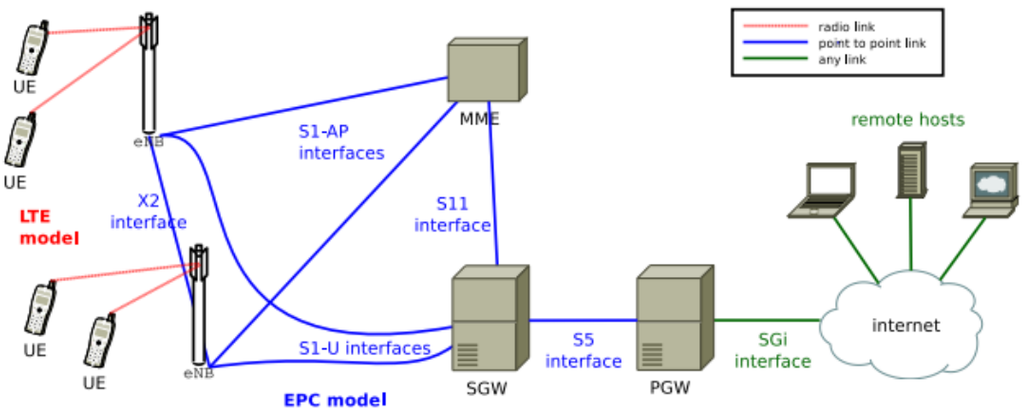

Figure 2.2.3 - Overview of LTE-EPC model [2]

2.2.3 LTE & EPC Module

Term LTE stands for Long Term Evolution and is the standard for wireless data transmission. The prototype name initially was LENA before being incorporated into Ns-3. It was developed by Ubuquisys (now is a department of Cisco) and the Center of Technology and Telecommunications of Catalonia (CTTC). Ubuquisys is a company specialized in developing smart antennas, while CTTC is a research center for communications technology.

System gives the opportunity to control the macro, micro and femto cells, so that their best interoperability is achieved. Those operate as self-organized networks. LENA can be used for design and estimation of efficiency in up/downlink schedulers, load balance, mobility management etc. An overview [2] of the LTE-EPC simulation model is depicted in the Figure 2.2.3 - Overview of LTE-EPC model. There are 2 basic components of LENA:

- LTE model which includes LTE Radio stack Protocol (RLC, PDCP, RRC, PHY, MAC). Entities exist in each UE and each evolvedNodeB (eNB) nodes.
- EPC model includes core network interfaces, protocols and entities. Entities exist in SGW, PGW and MME nodes and partially within eNB nodes. With Service Gateway (SGW) network mobility function is ensured and it is responsible for routing and forward network functions. The Packet Data Network Gateway (PDN GW) permits network connection and offers QoS. Mobility Management Entity (MME) manages mobility and provide security service to different data authentication application in LTE. With EPC, high packet data rate with low latency is achieved. It also provides ability to optimize packet flow with variety of operational scenarios.

Chapter 3: NS-3 scripting guide

A communication is established with at least one transmitter and one receiver along with the communication channel. Communication channel serves as the medium between transmitter and receiver and can be either wired or wireless, depending on the application.

Usually, transmitter and receiver are shown as nodes, placed in plane and forming a line segment indicating the transmission medium. In reality it cannot be a straight line, as it can be influenced by phenomena such as diffraction / refraction / reflection /scattering and other parameters that create curvature in the communication path. Communication channel is used by transmitter and sends the signal containing the information to receiver.

Additionally, it is necessary to determine transmission/connection type, achieved with protocols like Udp or Tcp, as well as other parameters such as transmission speed (data rate), packet size in bytes, number of packets to be transmitted and delay in the transmission due to link type (wired or wireless) of communication channel.

Here's a description of how to program a simulation in NS-3:

1. Install NS-3:

   First, you need to install NS-3 on your computer. You can download it from the official ns-3 website and follow the installation instructions.

2. Create a New Simulation Program:

   Create a new C++ of Python file with a ".cc" / ".cpp" or ".py" extension respectively for your simulation program.

3. Include Necessary Libraries:

   On top of the file, include the necessary NS-3 libraries, such as:

C++	Python
#include "ns3/core-module.h"	import ns.core
#include "ns3/network-module.h"	import ns.network

4. Define the Network Topology:

   Define network devices, nodes, routing rules, and other network parameters you want to simulate.

5. Set Simulation Parameters:

   Specify simulation settings, such as duration, time step, events, startup, and termination behavior.

6. Program Events:

   Create events that will occur during the simulation, such as packet transmission, topology changes, etc.

7. Run the Simulation:

   Execute simulation program and monitor the results. You can run the simulation from the command line or within the NS-3 environment.

8. Analysis and Results:

   Study the simulation results and analyze the performance of your network.

This is a basic overview of how to program a simulation in ns-3. NS-3 simulations can be quite complex, depending on your specific research or testing goals and program can be extended to include more advanced features and analysis as needed.

By default, NS-3 has 7 example scripts in C++ and initial three of them are written in python as well, placed in folder ‘examples/tutorial’. Always keep in mind that any new scripts must be placed in folder ‘scratch’. Just open any editor in Linux, create a new file, save it in the directory just mentioned and start scripting. Compile and run for each script are done using the command ‘./waf --pyrun scratch/fileName.py’. Otherwise, ‘.cc’ files with ‘./waf --run scratch/fileName’. For command line arguments: “./waf --pyrun "folder_name/fileName.py --argument=value"”

3.1 A hello world wired NS-3 script

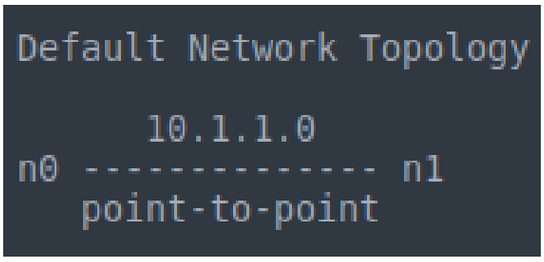

Figure 3.1.1 - Script topology 1

```
import ns.applications
import ns.core
import ns.internet
import ns.network
import ns.csma
import ns.point_to_point
import ns.mobility
import ns.netanim
import ns.wimax
import ns.wave
import sys
```

Figure 3.1.2 - Import modules

```
#Enable logging components that are built into the
#Echo Client and Echo Server applications:
#This  will  result in the application printing out
#messages as packets are sent and received
#during the simulation
ns.core.LogComponentEnable("UdpEchoClientApplication",
    ns.core.LOG_LEVEL_INFO)
ns.core.LogComponentEnable("UdpEchoServerApplication",
    ns.core.LOG_LEVEL_INFO)
```

Figure 3.1.3 - Enable logging info

Regarding the above and first script of "tutorials" [2], familiarization begins in creating a first script in python. Each library from ns-3 is called as 'ns.name'. Including correct libraries requires a few things to consider like topology, application and the purpose of simulation. Then, parameters are defined. A basic starting point in which a wired connection is constructed with two nodes is shown in Figure 3.1.1.

Node n0 is connected to node n1 and communicate to each other by sending a Udp packet of size 1024 bytes. Topology is shown in Figure 3.1.1 - Script topology 1 and represents the two nodes in a point-to-point (p2p) connection. Figure 3.1.2 - Import modules, informs about the needed libraries.

In Figure 3.1.3 - Enable logging info, logging information is provided during the simulation using '*ns.core*' library and as a parameter accepts the application's name which is later setup. Lots of levels of logging verbosity/detail are available. Logging is always useful in debugging.

```
#Create ns-3 Node objects:
nodes = ns.network.NodeContainer()
nodes.Create(2)
```

Figure 3.1.4 - Create nodes

```
#Construct a point-to-point link
#instantiate a PointToPoint object on the stack
pointToPoint = ns.point_to_point.PointToPointHelper()
#Set attributes
pointToPoint.SetDeviceAttribute("DataRate",
    ns.core.StringValue("13Mbps"))
pointToPoint.SetChannelAttribute("Delay",
    ns.core.StringValue("5ms"))

#We need to have a list of all of the NetDevice
#objects that are created:
devices1 = pointToPoint.Install(nodes)
```

Figure 3.1.5 - Connect nodes and define link properties

Nodes in simulation, represent computers or any other device about to connect to each other. Protocol stacks, applications and peripheral cards can be added. Node objects are created by instantiating the *NodeContainer* class of "*ns.network*" library as shown in Figure 3.1.4 - Create nodes, which contains useful modules like Node Container (large number of nodes), Network Devices (*NetDevices*), Sockets APIs etc.

First line declares a NodeContainer called "*nodes*" and in second line "*Create*" method is called on "*nodes*" object and asks the container to create two nodes. Nodes cannot communicate, so next step is vital. Wired link which connects the nodes is relevant to library '*ns.point_to_point'* and is linking exactly two devices over a PointToPointChannel.

Method *"ns.point_to_point.PointToPointHelper()*" instantiates a *PointToPointHelper* object on the stack. Its parameters (Attributes) for "*DataRate*" and "*Delay*" define the communication conditions, such as link speed and propagation delay as seen in Figure 3.1.5 - Connect nodes and define link properties. Values are set to 13 Mbps and 5 ms respectively. Link is installed on nodes and stored in variable "*devices*", which is a *NetDeviceContainer* attribute. So far, devices and channel have been configured.

```
#topology to define TCP/UDP/IP etc
#Nodes and devices are configured,
#so we need a protocol stack to install on our nodes:
stack = ns.internet.InternetStackHelper()
stack.Install(nodes)

#Associate devices on our nodes with IP addresses:
address1 = ns.internet.Ipv4AddressHelper()
address1.SetBase(ns.network.Ipv4Address("10.1.1.0"),
    ns.network.Ipv4Mask("255.255.255.0"))

#Perform the actual address assignment:
interfaces1 = address1.Assign(devices1)
```

Figure 3.1.6 - Addressing

```
#now we need to generate traffic
#Set UDP echo server application:
#we require the port number as a parameter to the constructor
echoServer1 = ns.applications.UdpEchoServerHelper(9)

serverApps1 = echoServer1.Install(nodes.Get(1))
serverApps1.Start(ns.core.Seconds(1.0))
serverApps1.Stop(ns.core.Seconds(100.0))
```

Figure 3.1.7 - Application server

Next, internet model is set. It is a useful functionality for IP-based objects with many related protocols (IPv4, IPv6, ARP, UDP, TCP etc). Internet protocol stack requires the "*ns.internet*" library. All of the internet model is installed on nodes and devices corresponding to nodes for configuration, as shown in Figure 3.1.6 - Addressing. *InternetStackHelper* is a topology helper just like *PointToPointHelper* is for point-to-point net devices. "*Install"* method will place an Internet Stack on each of the nodes in the *NodeContainer*.

Devices are associated to nodes with IP addresses. A topology helper *Ipv4AddressHelper* manages the allocation of IP addresses and "*SetBase*" method sets IP address and network mask. By default, allocated addresses will start at one and increase monotonically. "*Assign*" method performs the actual address assignment.

Library '*ns.applications*' contains every available function for the application layer (internet models). Two specializations are called, the *UdpEchoServerApplication* and *UdpEchoClientApplication*. Likewise in previous calls, a helper object to configure and manage the underlying objects.

Randomly a node is chosen as server (n1), a port number is defined to establish communication from the instance "*ns.applications.UdpEchoServerHelper*" and application is installed on it, when "*Install"* method is called. Also, time interval is set as shown in Figure 3.1.7 - Application server.

```
echoClient1 = ns.applications.UdpEchoClientHelper(interfaces1.GetAddress(1), 9)
#set max number of packets we allow  to send
echoClient1.SetAttribute("MaxPackets", ns.core.UintegerValue(1))
#how long to wait between packets
echoClient1.SetAttribute("Interval",
    ns.core.TimeValue(ns.core.Seconds(1.0)))
#send a packet of size 1024-bytes
echoClient1.SetAttribute("PacketSize",
    ns.core.UintegerValue(1024))

clientApps1 = echoClient1.Install(nodes.Get(0))
clientApps1.Start(ns.core.Seconds(2.0))
clientApps1.Stop(ns.core.Seconds(100.0))
```

Figure 3.1.8 - Application client

```
#Set mobility:
mobility = ns.mobility.MobilityHelper()
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                               "MinX", ns.core.DoubleValue(10.0),
                               "MinY", ns.core.DoubleValue (10.0),
                               "DeltaX", ns.core.DoubleValue(5.0),
                               "DeltaY", ns.core.DoubleValue(5.0),
                               "GridWidth", ns.core.UintegerValue(3),
                               "LayoutType", ns.core.StringValue("RowFirst"))
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel")
mobility.Install(nodes)
```

Figure 3.1.9 - Mobility and position on nodes

Application client is set as seen in Figure 3.1.8 - Application client, similar to previous case with the *UdpEchoClientApplication* object managed by *UdpEchoClientHelper*. A few parameters are set in next lines, attributes for remote address and remote port number (same as server). Recall that an *Ipv4InterfaceContainer* was used to keep track of the IP addresses assigned to devices. The zeroth interface will correspond to IP address of the zeroth node and first interface corresponds to first node etc.

Number of packets "*MaxPackets*" for transmission equals to 1, while "*Interval*" represents delay between packets and is set to 1 second. Size of packets "*PacketSize*" is 1024 bytes and duration time of the client application accepts arguments similar to echo server. Furthermore, "*ns.mobility*" module is installed on nodes. Mobility model objects track the evolution of position with respect to a cartesian coordinate system. Typically aggregated to an "*ns3::Node*" object. The initial position of objects is set with *PositionAllocator*. Similar types of objects will lay out the position of a notional canvas.

Once simulation starts, *PositionAllocator* may no longer be required or picks future mobility "waypoints" for such mobility models (in case of moving nodes). Coordinate system includes structures of *Rectangle*, *Box* and *Waypoint*, while *MobilityModel* has "*ConstantPosition"*, "*RandomWalk2D"* etc. *PositionAllocator* is mostly used for "*ListPositionAllocator*" in which positions occur as sequential list, but also contains random or uniform allocations or in a grid way. In Figure 3.1.9 - Mobility and position on nodes the "*GridPositionAllocator"* is called and module *ConstantPositionMobilityModel* sets the nodes as stationary.

```
ns.core.Simulator.Run()
ns.core.Simulator.Destroy()
```

Figure 3.1.10 - Run and Clear Resources

```
toni@ubuntu:~/Desktop/workspace/ns-allinone-3.31/ns-3.31$ ./waf --pyrun scratch/first.py
Waf: Entering directory `/home/toni/Desktop/workspace/ns-allinone-3.31/ns-3.31/build'
Waf: Leaving directory `/home/toni/Desktop/workspace/ns-allinone-3.31/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.481s)
```

Figure 3.1.11 - Execute script 1

Simulation is executed when global function "*ns.core.Simulator.Run()*" is called. Previously with methods "*Start*" and "*Stop*", events were scheduled at 1, 2 and 100 seconds respectively. System crosses through the list of scheduled events and executes them. Fist event at 1.0 second will enable the echo server application (may schedule other events), event begins at 2.0 seconds starting the echo client application (might also enable other events) and start event of echo client application begins data transfer phase of the simulation by sending a packet to the server.

A chain of events is triggered automatically behind the scenes, according to various timing parameters set in simulation. Since only one packet is sent, the chain of events triggered by client, echo request will taper off and simulation is set idle. Remaining events become the stop events for server and client. During their execution, no further events are processed and the simulation is completed. Clean up resources is accomplished with method "*ns.core.Simulator.Destroy()*". Figure 3.1.10 - Run and Clear Resources displays those function calls. Finally, command: "./waf --pyrun scratch/first.py" is executed as seen in Figure 3.1.11 - Execute script 1. The output is:

Sent 1024 bytes to 10.1.1.2

Received 1024 bytes from 10.1.1.

Received 1024 bytes from 10.1.1.2

Communication between the two nodes was successful. A packet of 1024 bytes was sent to echo server on 10.1.1.2 IP address and confirmation message below shows that the packet was received from client of IP address 10.1.1.1. Then, echo client receives its packet back from the server.

3.2 Second NS-3 script

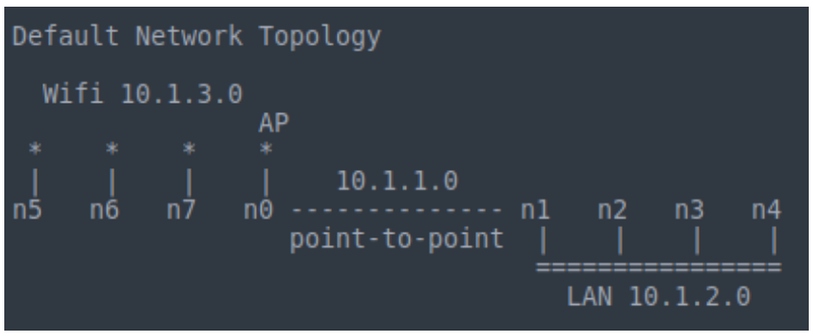

Figure 3.2.1 - Example topology 2

```
import ns.core
import ns.network
import ns.point_to_point
import ns.applications
import ns.wifi
import ns.mobility
import ns.netanim
import ns.csma
import ns.wimax
import ns.wave
import ns.internet
import ns.flow_monitor
import sys
```

Figure 3.2.2 - Import modules

A modified example of ‘tutorials’ folder of Ns-3 [2]. Represents a topology of wireless and wired connection communicating with each other, offering information about signal strength, noise power, frequency, etc and calculates the total throughput.

Carrier Sense Multiple Access (CSMA) is a bus network of devices and channels. As seen in point-to-point (p2p), CSMA topology helper object also constructs point-to-point topologies. Wifi topology helpers are presented in this example as well. Figure 3.2.1 - Example topology 2 represents a combination of wired and wireless topology communicating to each other. Lan 10.1.2.0 is a bus network on the right side. A simple network in the spirit of Ethernet. On the left side, there is a wireless network. Figure 3.2.2 - Import modules, informs about the necessary libraries.

```
cmd = ns.core.CommandLine()
cmd.nCsma = 3
cmd.verbose = "True"
cmd.nWifi = 3
cmd.tracing = "False"

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

cmd.Parse(sys.argv)

nCsma = int(cmd.nCsma)
verbose = cmd.verbose
nWifi = int(cmd.nWifi)
tracing = cmd.tracing
```

Figure 3.2.3 - Command line arguments

```
# The underlying restriction of 18 is due to the grid position
# allocator's configuration; the grid layout will exceed the
# bounding box if more than 18 nodes are provided.
if nWifi > 18:
    print ("nWifi should be 18 or less; otherwise grid layout exceeds the bounding box")
    sys.exit(1)

if verbose == "True":
    ns.core.LogComponentEnable("UdpEchoClientApplication",
        ns.core.LOG_LEVEL_INFO)
    ns.core.LogComponentEnable("UdpEchoServerApplication",
        ns.core.LOG_LEVEL_INFO)
```

Figure 3.2.4 - Logging components

A verbose flag determines whether the logging components are enabled and defaults to true (components enabled). Via command line, number of devices on the CSMA network can be changed (*nCsma* parameter), as well as the verbose flag, number of devices on the Wifi network and trace files during testing, as shown in Figure 3.2.3 - Command line arguments and Figure 3.2.4 - Logging components. “If” statement is a simple error detection about the capacity and effectiveness of the application.

```
p2pNodes = ns.network.NodeContainer()
p2pNodes.Create(2)

pointToPoint = ns.point_to_point.PointToPointHelper()
pointToPoint.SetDeviceAttribute("DataRate",
    ns.core.StringValue("5Mbps"))
pointToPoint.SetChannelAttribute("Delay",
    ns.core.StringValue("2ms"))

p2pDevices = pointToPoint.Install(p2pNodes)
```

Figure 3.2.5 - Create p2p nodes, set connection parameters and install devices on them

```
csmaNodes = ns.network.NodeContainer()
csmaNodes.Add(p2pNodes.Get(1))
csmaNodes.Create(nCsma)

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

csmaDevices = csma.Install(csmaNodes)
```

Figure 3.2.6 - CSMA nodes

Node objects are created by instantiating the *NodeContainer* class of "*ns.network*" library as shown in Figure 3.2.5 - Create p2p nodes, set connection parameters and install devices on them. First line declares a "*NodeContainer*" called "*p2pNodes*" and in second line "*Create*" method is called on "*p2pnodes*" object and asks the container to create two nodes. Nodes cannot communicate, so next step is vital. Wired link which connects the nodes is relevant to library '*ns.point_to_point'* and is linking exactly two devices over a PointToPointChannel.

"ns.point_to_point.PoinitToPointHelper()" instantiates a *PointToPointHelper* object on the stack. Its parameters (Attributes) for "*DataRate*" and "*Delay*" define the communication conditions, such as link speed and propagation delay. Values are set to 5 Mbps and 2 ms respectively. Link is installed on the nodes and stored in variable "*p2pDevices*", which is a "*NetDeviceContainer*" attribute. So far, devices and channel have been configured for the point-to-point connection.

Another "*NodeContainer*" declared in Figure 3.2.6 - CSMA nodes is responsible for nodes being part of the bus network. Node indexed to one is picked with method "*Get*" from the *p2p* node container and is added to container of CSMA nodes to be treated as a CSMA device. CSMA nodes are created when "*Create*" is called right below. After that, an instance of *CsmaHelper* is created and configure its attributes likewise the point-to-point. Linking is achieved with "*Install*" method on the CSMA nodes.

```
wifiStaNodes = ns.network.NodeContainer()
wifiStaNodes.Create(nWifi)
wifiApNode = p2pNodes.Get(0)

channel = ns.wifi.YansWifiChannelHelper.Default()
phy = ns.wifi.YansWifiPhyHelper.Default()
phy.SetChannel(channel.Create())

wifi = ns.wifi.WifiHelper()
wifi.SetRemoteStationManager("ns3::AarfWifiManager")

mac = ns.wifi.WifiMacHelper()
ssid = ns.wifi.Ssid ("ns-3-ssid")
```

Figure 3.2.7 - Wifi setup

```
mac.SetType ("ns3::StaWifiMac", "Ssid", ns.wifi.SsidValue(ssid),
    "ActiveProbing", ns.core.BooleanValue(False))
staDevices = wifi.Install(phy, mac, wifiStaNodes)

mac.SetType("ns3::ApWifiMac","Ssid", ns.wifi.SsidValue (ssid))
apDevices = wifi.Install(phy, mac, wifiApNode)
```

Figure 3.2.8 - Ssid setup

```
mobility = ns.mobility.MobilityHelper()
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                               "MinX", ns.core.DoubleValue(0.0),
                               "MinY", ns.core.DoubleValue (0.0),
                               "DeltaX", ns.core.DoubleValue(5.0),
                               "DeltaY", ns.core.DoubleValue(10.0),
                               "GridWidth", ns.core.UintegerValue(3),
                               "LayoutType", ns.core.StringValue("RowFirst"))

mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
    "Bounds", ns.mobility.RectangleValue(ns.mobility.Rectangle (-50, 50, -50, 50)))
mobility.Install(wifiStaNodes)

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

Figure 3.2.9 - Mobility and position on nodes

Next, wifi devices are constructed as well as the interconnection channel between wifi nodes as observed in Figure 3.2.7 - Wifi setup. At first, configuration of the physical layer is accomplished with channel helpers with *YansWifiChannelHelper* and *YansWifiPhyHelper* respectively (named after Yet another network simulator). For simplicity, the default configuration and channel models are used. Once these objects are created, channel object is associated with physical layer to ensure that all physical layer objects of *YansWifiPhyHelper* share the same channel (wireless medium), communicate and interfere, achieved by "*channel.Create()*" call. A non-QoS MAC is chosen and MAC layer is constructed by "*WifiMacHelper*" object.

"*SetRemoteStationManager"* method defines the rate control algorithm. A set identifier (SSID) object is created and used in MAC layer. Configuration of the MAC type is associated to SSID as to ensure base stations do not perform active probing. Similarly, MAC layer is configured for "*ns3::StaWifiMac*" and "*ns::ApWifiMac*" at Figure 3.2.8 - Ssid setup and installed on respective user and access point nodes.

```
stack = ns.internet.InternetStackHelper()
stack.Install(csmaNodes)
stack.Install(wifiApNode)
stack.Install(wifiStaNodes)

address = ns.internet.Ipv4AddressHelper()
address.SetBase(ns.network.Ipv4Address("10.1.1.0"),
    ns.network.Ipv4Mask("255.255.255.0"))
p2pInterfaces = address.Assign(p2pDevices)

address.SetBase(ns.network.Ipv4Address("10.1.2.0"),
    ns.network.Ipv4Mask("255.255.255.0"))
csmaInterfaces = address.Assign(csmaDevices)

address.SetBase(ns.network.Ipv4Address("10.1.3.0"),
    ns.network.Ipv4Mask("255.255.255.0"))
address.Assign(staDevices)
address.Assign(apDevices)
```

Figure 3.2.10 - Internet stack and IP addressing configuration

Furthermore, "*ns.mobility*" module is installed on nodes. Mobility model objects track the evolution of position with respect to a cartesian coordinate system. Typically aggregated to an "*ns3::Node*" object. Initial position of objects is set with "*PositionAllocator*". Similar types of objects will lay out the position of a notional canvas.

Once simulation starts, "*PositionAllocator*" may no longer be required or picks future mobility "waypoints" for such mobility models (in case of moving nodes). Coordinate system includes structures of "*Rectangle*", "*Box*" and "*Waypoint*", while "*MobilityModel*" has "*ConstantPosition"*, "*RandomWalk2D"* etc. "*PositionAllocator*" is mostly used for "*ListPositionAllocator*" in which positions occur as sequential list, but also contains random or uniform allocations or in a grid way.

In Figure 3.2.9 - Mobility and position on nodes, the "*GridPositionAllocator*" sets a two-dimensional grid to place the STA nodes, "*RandomWalk2dMobilityModel*" enables node movement in a random direction and speed around a bounding box. Mobility model is installed on STA nodes.

Access point remains in a fixed position using "*ConstantPositionMobilityModel*" module. Next, internet model is set. A useful functionality for IP-based objects with many related protocols (IPv4, IPv6, ARP, UDP, TCP etc). Internet protocol stack requires "*ns.internet*" library. Internet model is installed on devices corresponding to nodes for configuration, as shown in Figure 3.2.10 - Internet stack and IP addressing configuration. InternetStackHelper is a topology helper just like *PointToPointHelper* is for point-to-point net devices. "*Install"* method will place an Internet Stack on each of the nodes in the *NodeContainer*.

Devices are associated to nodes with IP addresses. A topology helper *Ipv4AddressHelper* manages the allocation of IP addresses and "*SetBase*" method sets IP address and network mask. By default, allocated addresses will start at one and increase monotonically. "*Assign*" method performs the actual address assignment.

```
echoServer = ns.applications.UdpEchoServerHelper(9)

serverApps = echoServer.Install(csmaNodes.Get(nCsma))
serverApps.Start(ns.core.Seconds(1.0))
serverApps.Stop(ns.core.Seconds(10.0))
```

Figure 3.2.11 - Application UdpEchoServer

```
echoClient = ns.applications.UdpEchoClientHelper(csmaInterfaces.GetAddress(nCsma), 9)
echoClient.SetAttribute("MaxPackets", ns.core.UintegerValue(5))
echoClient.SetAttribute("Interval", ns.core.TimeValue(ns.core.Seconds (1.0)))
echoClient.SetAttribute("PacketSize", ns.core.UintegerValue(1024))

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

Figure 3.2.12 - Application UdpEchoClient

Library '*ns.applications*' contains every available function for the application layer (internet models). Two specializations of this class are called, *UdpEchoServerApplication* and *UdpEchoClientApplication*. Likewise in previous calls, a helper object is used to configure and manage the underlying objects.

"*ns.applications.UdpEchoServerHelper*" establishes communication. The "rightmost" node of point-to-point nodes is chosen and application is installed with "*Install*" method. Also, time interval to operate is set as shown in Figure 3.2.11 - Application UdpEchoServer.

Application client is configured as seen in Figure 3.1.8 - Application client, similar to previous case with *UdpEchoClientApplication* object managed by *UdpEchoClientHelper*. In first line two attributes are set, remote address and remote port number (same as server). Recall that an *Ipv4InterfaceContainer* keeps track of the IP addresses assigned to devices. The zeroth interface corresponds to IP address of the zeroth node and first interface corresponds to first node etc.

All parameters are configured shown in Figure 3.2.12 - Application UdpEchoClient. "*MaxPackets*" to be transmitted equals to 1, "*Interval*" for the delay between packets is set to 1 second, "*PacketSize*" the size of packets is 1024 bytes and the duration time of the client application is similar to echo server.

```
ns.internet.Ipv4GlobalRoutingHelper.PopulateRoutingTables()

flowmonHelper = ns.flow_monitor.FlowMonitorHelper()
monitor = flowmonHelper.InstallAll()
xml_filename = "third.xml"
anim = ns.netanim.AnimationInterface(xml_filename)

ns.core.Simulator.Stop(ns.core.Seconds(10.0))

if tracing == "True":
    pointToPoint.EnablePcapAll ("third")
    phy.EnablePcap ("third", apDevices.Get (0))
    csma.EnablePcap ("third", csmaDevices.Get (0), True)
```

Figure 3.2.13 - Routing Tables, stop simulation, enable trace files

```
ns.core.Simulator.Run()

flowmon_filename = "third.flowmon"
monitor.SerializeToXmlFile(flowmon_filename,
        ns.core.BooleanValue(True), ns.core.BooleanValue(True))
monitor.CheckForLostPackets()
flows = monitor.GetFlowStats()

throughput = []
for flow_id, flow_stats in flows:
    if flow_stats.timeLastRxPacket.GetSeconds() - flow_stats.timeFirstTxPacket.GetSeconds() != 0:
        throughput.append(calculate_throughput(flow_stats))

total = sum(throughput)
print(f'\tTotal throughput:\t{total:.2f} Mbps')

ns.core.Simulator.Destroy()
```

Figure 3.2.14 - Run simulation and clear resources

```
At time 5.01058s client received 1024 bytes from 10.1.2.4 port 9
At time 6s client sent 1024 bytes to 10.1.2.4 port 9
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:2 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:3 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:4 Does not have a mobility model. Use SetConstantPosition if it is stationary
At time 6.00529s server received 1024 bytes from 10.1.3.3 port 49153
At time 6.00529s server sent 1024 bytes to 10.1.3.3 port 49153
AnimationInterface WARNING:Node:4 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:4 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:4 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:1 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:2 Does not have a mobility model. Use SetConstantPosition if it is stationary
AnimationInterface WARNING:Node:3 Does not have a mobility model. Use SetConstantPosition if it is stationary
At time 6.01058s client received 1024 bytes from 10.1.2.4 port 9
        Total throughput:       0.02 Mbps
toni@ubuntu:~/Desktop/workspace/ns-allinone-3.31/ns-3.31$
```

Figure 3.2.15 - Script output 2

An internetwork is built and internetwork routing is configured with "*PopulateRoutingTables()*" method as seen in Figure 3.2.13 - Routing Tables, stop simulation, enable trace files. "*Stop*" method ensures proper functionality, because beacons are generated forever since a non-QoS MAC was chosen on Ssid setup and simulation would be executed infinitely.

Also, enough tracing is created to cover all three networks producing pcap files and can be later processed in cooperation with any other tool (Wireshark). Global function "*ns.core.Simulator.Run()*" render simulation executable. Method "*ns.core.Simulator.Destroy()*" releases resources. Figure 3.2.14 - Run simulation and clear resources displays both function calls. Finally, the script is executed: "./waf --pyrun scratch/sample.py" as seen in Figure 3.2.15 - Script output 2 and displays the total throughput during simulation (function is found in Appendix).

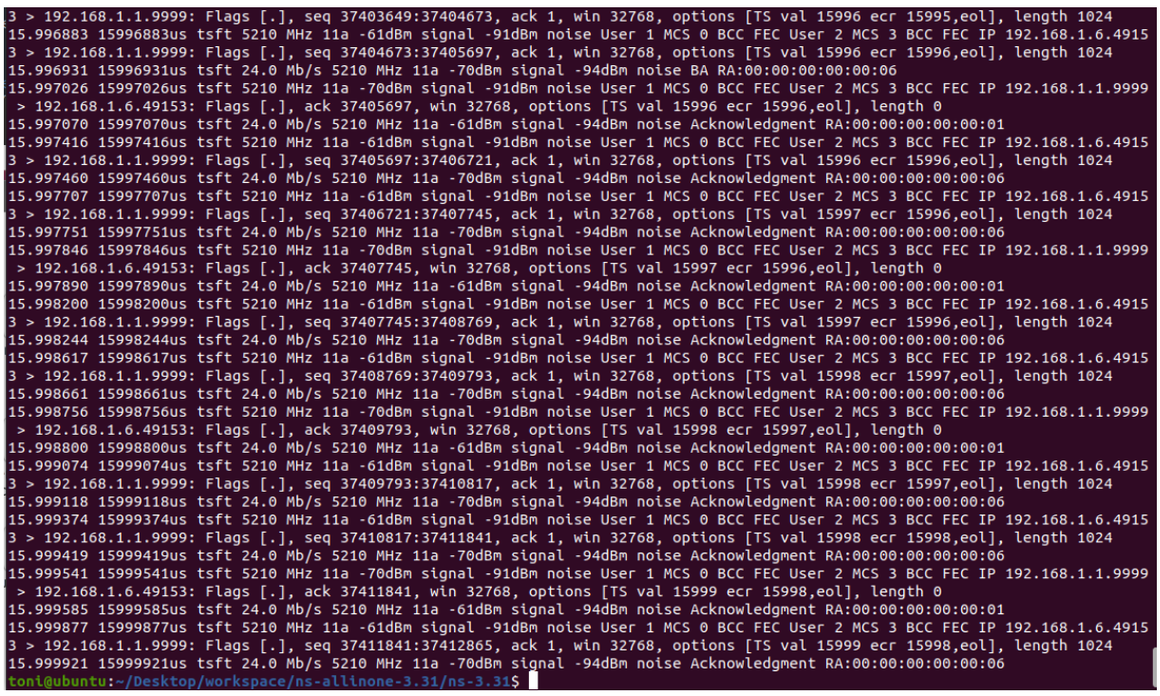

Figure 3.2.16 – Pcap file output

Also, Figure 3.2.15 - Script output 2 informs about a client sending and receiving packets on port number 9, a few milliseconds later server sends and receives packet on port number 49153 (default in NS-3) and so on during the simulation time. At the end the total throughput is calculated and displayed.

Directory contains a few trace files created from the simulation as well as the "*.xml*" file providing information about data transfer in each flow. Different trace files correspond to a separate part of the topology. Pcap files are processed via command "*tcpdump -nn -tt -r Total-1-0.pcap*". As seen in Figure 3.2.16 – Pcap file output, contains enough information about the connection providing signal strength, noise power, frequency etc.

3.3 An advanced Ns-3 script

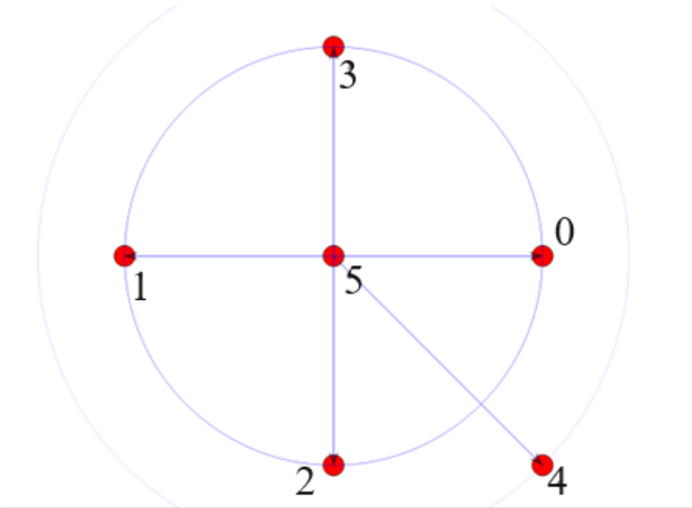

Figure 3.3.1 - Example topology 3

```
import ns.core
import ns.network
import ns.applications
import ns.wifi
import ns.mobility
import ns.internet
import ns.flow_monitor
import random
import ns.netanim
import ns.csma
import ns.wimax
import ns.wave
```

Figure 3.3.2 - Import modules

A representation of a complex application for voice service using WiFi module with different data rates, packet size and port numbers with different cases of Udp and Tcp socket connections. Datagrams are sent to a unique access point. Each different case shows a user being served on the same base station. Tracing is enabled, as well as monitoring with ".xml" and ".pcap" files produced for further analysis with Wireshark and examination with NetAnim.

Each node can communicate with all others in different time intervals. Positions are not defined with a specific model as shown later. Figure 3.3.1 - Example topology 3 is a visual representation of the topology and Figure 3.3.2 - Import modules represents the necessary libraries for simulation.

```
def main(argv):
    simulationTime = 15 #seconds
    distance = 10.0 #meters
    simulationTime = float(simulationTime)
    distance = float(distance)

    #Configuration arguments
    bandwidth = 40
    mcs = 3
```

Figure 3.3.3 - Create function and initialize parameters

```
channel = ns.wifi.YansWifiChannelHelper.Default()
phy = ns.wifi.YansWifiPhyHelper.Default()
wifi = ns.wifi.WifiHelper()
mac = ns.wifi.WifiMacHelper()

phy.SetPcapDataLinkType (ns.wifi.WifiPhyHelper.DLT_IEEE802_11_RADIO)
phy.SetChannel (channel.Create ())

wifi.SetStandard (ns.wifi.WIFI_PHY_STANDARD_80211ac)

wifiStaNode = ns.network.NodeContainer ()
wifiStaNode.Create (5)
wifiApNode = ns.network.NodeContainer ()
wifiApNode.Create (1)
```

Figure 3.3.4 - Set Wifi

As a starting point a function is created in which all modules are called. Simulation time, distance, bandwidth and value '*mcs*' (parameter for data rate) are initialized as seen in Figure 3.3.3 - Create function and initialize parameters. Then, wifi devices are constructed and interconnection channel between wifi nodes as observed in Figure 3.3.4 - Set Wifi. At first, configuration of the physical layer is accomplished with channel helpers with *YansWifiChannelHelper* and *YansWifiPhyHelper* respectively.

For simplicity, default configuration and channel models are used. Once objects are created, channel object is associated with physical layer ensuring all physical layer objects of *YansWifiPhyHelper* share the same channel (wireless medium) and can communicate and interfere. It is accomplished with "*channel.Create()*" call. MAC layer is set with a QoS-Supported choice and MAC layer is constructed by "*WifiMacHelper*" object.

"*SetRemoteStationManager*" method defines rate control algorithm. Constant rate is chosen (explained in documentation of ns-3 [2]). A set identifier (SSID) object is created and retrieved in MAC layer. Configuration of MAC type with SSID confirms base stations do not perform active probing. Similarly, MAC layer is configured for "*ns3::StaWifiMac*" and "*ns::ApWifiMac*" and installed on respective nodes along with the bandwidth configuration, according to Figure 3.3.5 - Wifi configurations.

```
DataRate = "VhtMcs"+str(mcs)

wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
                             "DataMode", ns.core.StringValue(DataRate),
                             "ControlMode", ns.core.StringValue(DataRate))

ssid = ns.wifi.Ssid ("wifi-80211ac")

mac.SetType("ns3::StaWifiMac", "QosSupported", ns.core.BooleanValue (True),
                               "Ssid", ns.wifi.SsidValue (ssid))

staDevice = wifi.Install (phy, mac, wifiStaNode)

mac.SetType("ns3::ApWifiMac", "QosSupported", ns.core.BooleanValue (True),
                              "Ssid", ns.wifi.SsidValue (ssid))

apDevice = wifi.Install (phy, mac, wifiApNode)

# Set channel width
ns.core.Config.Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/ChannelWidth",
                   ns.core.UintegerValue (bandwidth))
```

Figure 3.3.5 - Wifi configurations

```
# mobility
mobility = ns.mobility.MobilityHelper ()
positionAlloc = ns.mobility.ListPositionAllocator ()

positionAlloc.Add (ns.core.Vector3D (0.0, 0.0, 0.0))
positionAlloc.Add (ns.core.Vector3D (distance, 0.0, 0.0))
positionAlloc.Add (ns.core.Vector3D (-distance, 0.0, 0.0))
positionAlloc.Add (ns.core.Vector3D (0, distance, 0.0))
positionAlloc.Add (ns.core.Vector3D (0, -distance, 0.0))
positionAlloc.Add (ns.core.Vector3D (distance, distance, 0.0))
mobility.SetPositionAllocator (positionAlloc)

mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel")

mobility.Install (wifiApNode)
mobility.Install (wifiStaNode)
```

Figure 3.3.6 - Mobility and position on nodes

Furthermore, "*ns.mobility*" module is installed on nodes. Mobility model objects track the evolution of position with respect to a cartesian coordinate system. Typically aggregated to an "*ns3::Node*" object. Initial position of objects is set with *PositionAllocator*. Similar types of objects will lay out the position of a notional canvas.

Once simulation starts, "*PositionAllocator*" may no longer be required or picks future mobility "waypoints" for such mobility models (in case of moving nodes). Coordinate system includes structures of "*Rectangle*", "*Box*" and "*Waypoint*", while "*MobilityModel*" has "*ConstantPosition"*, "*RandomWalk2D*" etc. "*PositionAllocator*" is mostly used for the "*ListPositionAllocator*" in which positions occur as sequential list, but also contains random or uniform allocations or in a grid way. Allocation of nodes is represented in Figure 3.3.6 - Mobility and position on nodes. Placed directly through '*distance*' parameter (defined and initialized on top). All nodes remain in a fixed position with module "*ConstantPositionMobilityModel*".

```
# Internet stack
stack = ns.internet.InternetStackHelper ()
stack.Install (wifiApNode)
stack.Install (wifiStaNode)

address = ns.internet.Ipv4AddressHelper ()

address.SetBase (ns.network.Ipv4Address("192.168.1.0"),
    ns.network.Ipv4Mask ("255.255.255.0"))
staNodeInterface = address.Assign (staDevice)
apNodeInterface = address.Assign (apDevice)
ns.core.Config.SetDefault ("ns3::TcpSocket::SegmentSize",
    ns.core.UintegerValue (1024))
```

Figure 3.3.7 - Internet stack & Addressing

```
# Setting applications
serverApp = ns.network.ApplicationContainer ()
sinkApp = ns.network.ApplicationContainer ()

viPort = 9999
voPort_1 = 9998
voPort_2 = 9997
voPort_3 = 9996
voPort_4 = 9995
bePort = 9994
bkPort = 9993

ipv4 = wifiStaNode.Get (0).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
print("Sta IP Addr:", adr)
#print(adr)
sinkSocket = ns.network.InetSocketAddress (adr, viPort)
sinkSocket.SetTos (0x4)
onOffHelper = ns.applications.OnOffHelper("ns3::TcpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate",
    ns.core.StringValue ("20Mbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (1024))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (1.001 + random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiApNode.Get(0)))

packetSinkHelper = ns.applications.PacketSinkHelper("ns3::TcpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiStaNode.Get (0)))
```

Figure 3.3.8 – First node app

Next, internet model is set. A useful functionality for IP-based objects with many related protocols (IPv4, IPv6, ARP, UDP, TCP etc). Internet protocol stack requires "*ns.internet*" library. Internet model is installed on devices corresponding to nodes for configuration, as shown in Figure 3.3.7 - Internet stack & Addressing. "*InternetStackHelper*" is a topology helper just like "*PointToPointHelper*" is for point-to-point net devices. "*Install*" method will place an Internet Stack on each node of "*NodeContainer*". Devices are associated to nodes with IP addresses. A topology helper "*Ipv4AddressHelper*" manages allocation of IP addresses and the "*SetBase*" method sets IP address and network mask. By default, allocated addresses will start at one and increase monotonically. "*Assign*" method performs the actual address assignment.

A container stores each different application. At first, user '*wifiStaNode.Get(0)*' is configured specifying socket Tcp connection. Each node's IP address can be obtained in a tricky way. Direct access is not available, so the "*internet.Ipv4*" is called as instance and object is obtained. IP address of the particular node is extracted and given as input in '*sinkSocket*'. Also, from "*applications*" the "*OnOffHelper*" determines application type (TCP).

Parameters 'on' and 'off' time accept values 1 and 0 respectively. "*ConstantRandomVariable*" indicates a Constant Bit Rate with data rate of 20 Mbps and packet size at 1024 bytes. Start time value ranges from 1.001 to 1.101 in a uniform distribution. Specific node cited as a source receiving data and is configured as "*PacketSinkHelper*". Last three lines of Figure 3.3.8 – First node app indicate a downlink direction. Sender in this case is the base station '*wifiApNode*'.

```
#Voice - 1
ipv4 = wifiStaNode.Get (1).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
print("Voice 1 IP Addr:", adr)
sinkSocket = ns.network.InetSocketAddress (adr, voPort_1)
sinkSocket.SetTos (0x6)
onOffHelper=ns.applications.OnOffHelper ("ns3::UdpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate",
    ns.core.StringValue ("64Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (160))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (5.001)))

serverApp.Add (onOffHelper.Install(wifiApNode.Get(0)))

packetSinkHelper = ns.applications.PacketSinkHelper("ns3::UdpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiStaNode.Get (1)))
```

Figure 3.3.9 – Second node app 1

```
ipv4 = wifiApNode.Get (0).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
sinkSocket = ns.network.InetSocketAddress (adr, voPort_1)
sinkSocket.SetTos (0x6)
onOffHelper=ns.applications.OnOffHelper ("ns3::UdpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate",
    ns.core.StringValue ("64Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (160))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (5.001+random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiStaNode.Get(1)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::UdpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiApNode.Get (0)))
```

Figure 3.3.10 - Second node app 2

For each of the following cases of setting application on pair nodes, there will be reference once and figures are shown subsequently. Similar to previous, each next "*wifiStaNode*" is obtained as receiver, with Udp or Tcp socket, setting parameters "*DataRate*" to 64, 161 and 250 Kbps respectively, "*PacketSize*" to 160, 501 and 750 bytes respectively and "*StartTime*" at 5.001 ranges from 5.001 to 5.101 in a uniform distribution and 10.001 ranged from 10.001 to 10.101 in a uniform distribution as seen in Figure 3.3.9 – Second node app 1, Figure 3.3.11 - Third node app 1, Figure 3.3.13 – Fourth node app 1 and Figure 3.3.15 - Fifth node app 1.

Figure 3.3.10 - Second node app 2, Figure 3.3.12 - Third node app 2, Figure 3.3.14 - Fourth node app 2 and Figure 3.3.16 - Fifth node app 2 reflect the uplink and are structured similarly. Both the two applications provide us the uplink and downlink. As mentioned, next two pages 35 and 36 follow the same structure for different parameters in "*DataRate*", "*PacketSize*" and "*StartTime*", while the last part on page 37 includes file creation for further analysis and script execution. Page 38 contains figure of NetAnim and explanation of figures in pages 37 and 38.

```
#Voice - 2
ipv4 = wifiStaNode.Get (2).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
print("Voice 2 IP Addr:", adr)
sinkSocket = ns.network.InetSocketAddress (adr, voPort_2)
sinkSocket.SetTos (0x6)
onOffHelper=ns.applications.OnOffHelper ("ns3::UdpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute("DataRate",    ns.core.StringValue ("64Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (160))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (5.001)))

serverApp.Add (onOffHelper.Install(wifiApNode.Get(0)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::UdpSocketFactory",
    sinkSocket)
```

Figure 3.3.11 - Third node app 1

```
ipv4 = wifiApNode.Get (0).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
sinkSocket = ns.network.InetSocketAddress (adr, voPort_2)
sinkSocket.SetTos (0x6)
onOffHelper=ns.applications.OnOffHelper ("ns3::UdpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate", ns.core.StringValue ("64Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (160))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (5.001+random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiStaNode.Get(2)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::UdpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiApNode.Get (0)))
```

Figure 3.3.12 - Third node app 2

```
#BestEffort
ipv4 = wifiStaNode.Get (3).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
print("Best Effort IP Addr:", adr)
sinkSocket = ns.network.InetSocketAddress (adr, bePort)
sinkSocket.SetTos (0x0)
onOffHelper=ns.applications.OnOffHelper ("ns3::TcpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate", ns.core.StringValue ("161Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (501))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (10.001+random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiApNode.Get(0)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::TcpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiStaNode.Get (3)))
```

Figure 3.3.13 – Fourth node app 1

```
ipv4 = wifiApNode.Get (0).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
sinkSocket = ns.network.InetSocketAddress (adr, bePort)
sinkSocket.SetTos (0x0)
onOffHelper=ns.applications.OnOffHelper ("ns3::TcpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate", ns.core.StringValue ("161Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (501))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (10.001+random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiStaNode.Get(3)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::TcpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiApNode.Get (0)))
```

Figure 3.3.14 - Fourth node app 2

```
#BackGroun
ipv4 = wifiStaNode.Get (4).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
print("Background IP Addr:", adr)
sinkSocket = ns.network.InetSocketAddress (adr, bkPort)
sinkSocket.SetTos (0x1)
onOffHelper=ns.applications.OnOffHelper ("ns3::TcpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate", ns.core.StringValue ("250Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (750))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (10.001+random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiApNode.Get(0)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::TcpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiStaNode.Get (4)))
```

Figure 3.3.15 - Fifth node app 1

```
ipv4 = wifiApNode.Get (0).GetObject(ns.internet.Ipv4.GetTypeId ())
adr = ipv4.GetAddress (1,0).GetLocal ()
sinkSocket = ns.network.InetSocketAddress (adr, bkPort)
sinkSocket.SetTos (0x1)
onOffHelper=ns.applications.OnOffHelper ("ns3::TcpSocketFactory", sinkSocket)
onOffHelper.SetAttribute ("OnTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=1]"))
onOffHelper.SetAttribute ("OffTime",
    ns.core.StringValue ("ns3::ConstantRandomVariable[Constant=0]"))
onOffHelper.SetAttribute ("DataRate", ns.core.StringValue ("250Kbps"))
onOffHelper.SetAttribute ("PacketSize", ns.core.UintegerValue (750))
onOffHelper.SetAttribute ("StartTime" ,
    ns.core.TimeValue (ns.core.Seconds (10.001+random.uniform(0,0.1))))

serverApp.Add (onOffHelper.Install(wifiStaNode.Get(4)))

packetSinkHelper = ns.applications.PacketSinkHelper ("ns3::TcpSocketFactory",
    sinkSocket)
sinkApp.Add (packetSinkHelper.Install (wifiApNode.Get (0)))
```

Figure 3.3.16 - Fifth node app 2

```
ns.internet.Ipv4GlobalRoutingHelper.PopulateRoutingTables ()

phy.EnablePcap( "AP.pcap", apDevice.Get (0))
phy.EnablePcapAll("Total")

flowmonHelper = ns.flow_monitor.FlowMonitorHelper()
monitor = flowmonHelper.InstallAll()
xml_filename = "sample.xml"
anim = ns.netanim.AnimationInterface(xml_filename)

ns.core.Simulator.Stop (ns.core.Seconds (simulationTime+1))
ns.core.Simulator.Run ()
flowmon_filename = "sample.flowmon"
monitor.SerializeToXmlFile(flowmon_filename,
    ns.core.BooleanValue(True), ns.core.BooleanValue(True))
monitor.CheckForLostPackets()
ns.core.Simulator.Destroy ()

return 0
```

Figure 3.3.17 - Routing Tables, enable tracing and monitoring, Run simulation and clear resources

```
if __name__ == '__main__':
    import sys
    sys.exit (main (sys.argv))
```

Figure 3.3.18 - Execute script

```
Waf: Leaving directory `/home/toni/Desktop/workspace/ns-allinone-3.31/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.071s)
Sta IP Addr: 192.168.1.1
Voice 1 IP Addr: 192.168.1.2
Voice 2 IP Addr: 192.168.1.3
Best Effort IP Addr: 192.168.1.4
Background IP Addr: 192.168.1.5
Max Packets per trace file exceeded
toni@ubuntu:~/Desktop/workspace/ns-allinone-3.31/ns-3.31$
```

Figure 3.3.19 - Script output 3

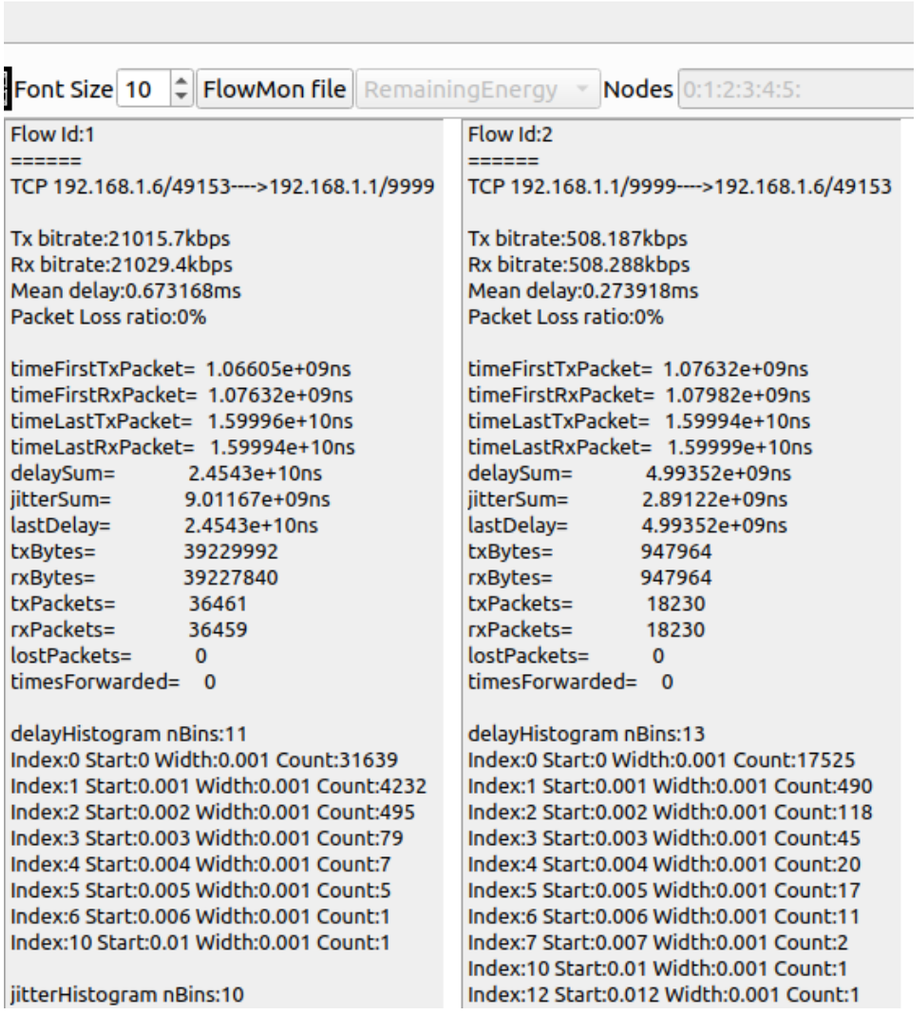

Figure 3.3.20 - Flow monitor in NetAnim

Internetwork is built and internetwork routing is enabled with "*ns.internet.Ipv4GlobalRoutingHelper.PopulateRoutingTables()*" as shown in Figure 3.3.17. Also, trace file is enabled for a user, while base station provides "*.pcap*" files for all incoming connections.

"*flow_monitor*" module is used for visualization of topology with NetAnim and also investigate further for lost packets, data rate, throughput, delay etc. "*Stop*" method is necessary, cause simulation would be executed infinitely. Also, enough tracing is created to cover all three networks producing pcap files. Those can be later processed using any other tool (Wireshark). Global function "*ns.core.Simulator.Run()*" render simulation executable. Method "*ns.core.Simulator.Destroy()*" releases resources. Figure 3.3.18 - Execute script, represents main function call in a python way. Finally, script is executed with command: "./waf --pyrun scratch/sample.py" as seen in Figure 3.3.19 - Script output 3 and "*.pcap*" files are present along with "*.xml*" and "*.flowmon*" files within directory and are available for further processing with Wireshark or visualized with NetAnim respectively as in Figure 3.3.20 - Flow monitor in NetAnim. A minimal part is shown as an example. Each flow is represented separately providing insights about the throughput, delay, packet loss ratio, data rate etc.

Chapter 4: Case Study in User Association

4.1 Description of the modules - Theory

Long Term Evolution module is used in 4th and later generation networks. LTE's high data rates, low latency, scalability, efficient spectrum utilization, QoS support, interoperability, security, voice and data integration, energy efficiency, and backward compatibility make it a compelling choice for cellular HetNets. These advantages enable LTE to meet the diverse communication needs of users and devices in a heterogeneous network environment. Following subjects are included in references [2].

Main objective of the Evolved Packet Core (EPC) model is to provide means for simulation of end-to-end IP connectivity over the LTE model. To this aim, it supports interconnection of multiple UEs to the Internet, via a radio access network of multiple eNBs connected to core network, as shown in Figure 2.2.3 - Overview of LTE-EPC model. Following design choices have been made for EPC model:

1. Packet Data Network (PDN) type supported is both IPv4 and IPv6. In other words, the end-to-end connections between UEs and remote hosts can be IPv4 and IPv6. However, networks between the core network elements (MME, SGWs and PGWs) are IPv4-only.
2. SGW and PGW functional entities are implemented in different nodes, which are hence referred to as SGW node and PGW node, respectively.
3. MME functional entities are implemented as a network node, which is hence referred to as the MME node.
4. The scenarios with inter-SGW mobility are not of interest. But several SGW nodes may be present in simulations scenarios.
5. A requirement for EPC model is its convenience to simulate the end-to-end performance of realistic applications. Hence, it should be possible to use with EPC model any regular ns-3 application working on top of TCP or UDP.
6. Another requirement is the possibility of simulating network topologies with presence of multiple eNBs, some of which might be equipped with a backhaul connection with limited capabilities. In order to simulate such scenarios, the user data plane protocols being between the eNBs and the SGW should be modeled accurately.
7. It should be possible for a single UE to use different applications with different QoS profiles. Hence, multiple EPS bearers should be supported for each UE. This includes the necessary classification of TCP/UDP traffic over IP done at the UE in the uplink and at the PGW in the downlink.

Channel and Propagation

For channel modeling purposes, LTE module uses the SpectrumChannel interface provided by the spectrum module. Two implementations of such interface are available:
SingleModelSpectrumChannel and MultiModelSpectrumChannel. LTE module requires MultiModelSpectrumChannel in order to work properly. Support of different frequency and

bandwidth configurations is necessary. All the propagation models supported by Multi Model Spectrum Channel can be used within LTE module.

Fading Model

With respect to the mathematical channel propagation model, the one provided by the rayleighchan function of Matlab is suggested, since it provides a well-accepted channel modelization both in time and frequency domain. For more information, the reader is referred to [mathworks]. The simulator provides a matlab script (src/lte/model/fading-traces/fading-trace-generator.m) for generating traces based on the format used by the simulator.

In detail, the channel object created with the rayleighchan function is used for filtering a discrete-time impulse signal in order to obtain the channel impulse response. The filtering is repeated for different TTI, thus yielding subsequent time-correlated channel responses (one per TTI). The channel response is then processed with the pwelch function for obtaining its power spectral density values, which are then saved in a file with the proper format compatible with the simulator model.

Antennas

Being based on the SpectrumPhy, the LTE PHY model supports antenna modeling via the ns-3 AntennaModel class. Hence, any model based on it can be associated with any eNB or UE instance. For example, CosineAntennaModel associated with an eNB device allows to model one sector of a macro base station. By default, the IsotropicAntennaModel is used for both eNBs and UEs.

Resource Allocation

We now briefly describe how resource allocation is handled in LTE, clarifying how it is modeled in the simulator. Scheduler is in charge of generating specific structures called Data Control Indication (DCI) which are then transmitted by the PHY of eNB to connected UEs, in order to inform them of the resource allocation on a per subframe basis. In downlink direction, the scheduler has to fill some specific fields of the DCI structure with all the information, such as: the Modulation and Coding Scheme (MCS) to be used, the MAC Transport Block (TB) size, and the allocation bitmap identifying which RBs will contain the data transmitted by an eNB to each user.

Mapping of resources to physical RBs, a localized mapping approach is adopted (see [Sesia2009], Section 9.2.2.1); hence in a given subframe each RB is always allocated to the same user in both slots. Allocation bitmap can be coded in different formats; in this implementation, Allocation Type 0 is considered defined in [TS36213], according to which the RBs are grouped in Resource Block Groups (RBG) of different size determined as a function of Transmission Bandwidth Configuration in use. For certain bandwidth values not all the RBs are usable, since the group size is not a common divisor of the group. This is the case when the bandwidth is equal to 25 RBs, which results in a RBG size of 2 RBs, and therefore 1 RB will result not addressable. In uplink the format of the DCIs is different, since only adjacent RBs can be used because of the SC-FDMA modulation. As a consequence, all RBs can be allocated by the eNB regardless of the bandwidth configuration.

Round Robin Scheduler

Round Robin (RR) scheduler is probably the simplest scheduler found in the literature. It works by dividing the available resources among the active flows, i.e., those logical channels which have a non-empty RLC queue. If number of RBGs is greater than the number of active flows, all flows can be allocated in the same subframe.

Otherwise, if number of active flows is greater than number of RBGs, not all the flows can be scheduled in a given subframe; then, next subframe starts the allocation from the last flow that was not allocated. The MCS to be adopted for each user is done according to the received wideband CQIs.

NakagamiPropagationLossModel

Specific propagation loss model implements the Nakagami-m fast fading model, which accounts for the variations in signal strength due to multipath fading. It does not account path loss due to distance traveled by the signal, hence for typical simulation usage it is recommended to consider using it in combination with other models that consider this aspect. The Nakagami-m distribution is applied to the power level. Probability density function is defined as:

$$\mathrm{p}(\mathrm{x};\ \mathrm{m}, \omega) = \frac{2m^m}{\Gamma(m)\omega^m} x^{2m-1} e^{-\frac{m}{\omega}x^2} \qquad (4.1)$$

with m the fading depth parameter and ω the average received power. It is implemented by either a GammaRandomVariable or ErlangRandomVariable random variable. For m = 1 the Nakagami-m distribution equals to Rayleigh distribution flat fading. The equation's form is:

$$\mathrm{p}(\mathrm{x};\ 1, \omega) = \frac{2}{\Gamma(1)\omega^1} x^{2-1} e^{-\frac{1}{\omega}x^2}$$

and we know that Γ(1) equals to 0! = 1, so:

$$\mathrm{p}(\mathrm{x};\ 1, \omega) = \frac{2}{\omega} x e^{-\frac{x^2}{\omega}} \qquad (4.2)$$

In that case the ω parameter is not affecting us and could be ignored or considered to be equal to 1.

LogDistancePropagationLossModel

Implements a log distance propagation model. The reception power is calculated with a so-called log-distance propagation model:

$$L = L_0 + 10\beta\, log\left(\frac{d}{d_0}\right) \qquad (4.3)$$

where: β is the path loss distance exponent, d_0 is reference distance (m), L_0 is path loss at reference distance (dB), d is negative value "-distance" (m) and L is path loss (dB). When the path loss is requested at a distance smaller than the reference distance, the tx power is returned.

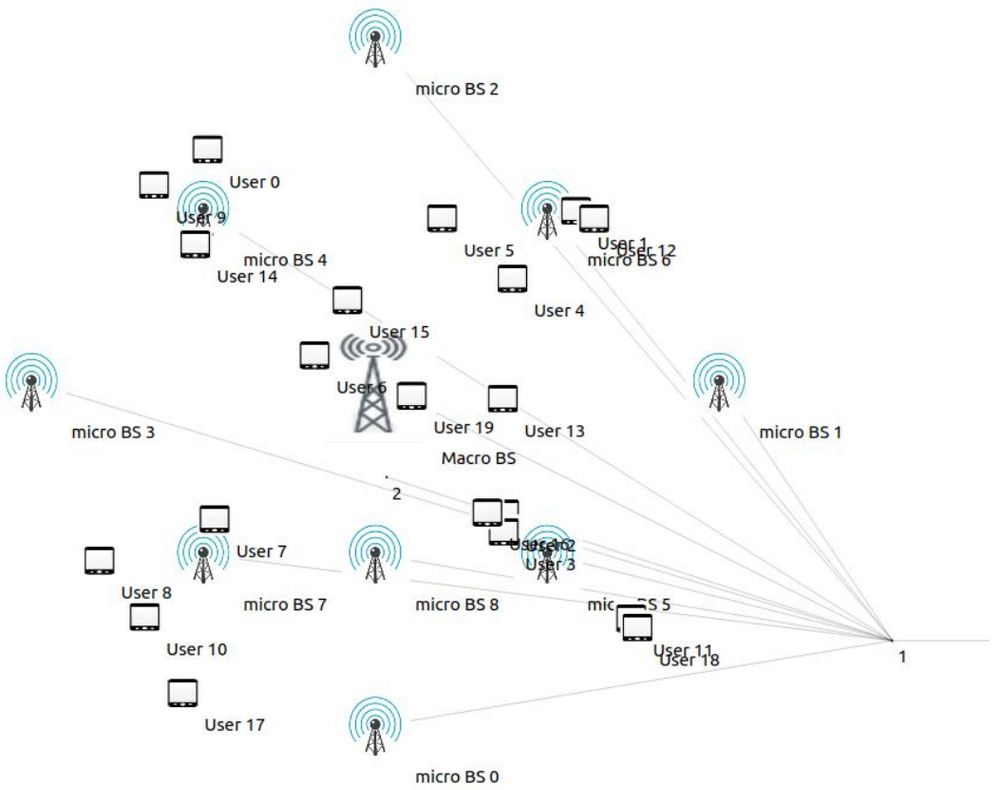

Figure 4.2.1 - Network Topology

4.2 Implementation

Following the structure of the cited work [5] and considering all the parameters and models for the implementation, we built the script in Ns-3 using python 3. A two-tier heterogeneous network is considered, consisting of B base stations. The B−1 are micro base stations (mBS) and one is the macro base station (MBS).

Base stations have fixed positions considering a circle centered at (0,0) and radius at 200 m. Users referred as terminals or user equipment (UE) are placed also at fixed positions in a random way using same circle of radius 200 m. UEs are located inside the coverage area created by base stations.

Micro base stations are blue colored antenna and macro base station is slightly different sized with grey color as seen in Figure 4.2.1 - Network Topology. UEs are represented as mobile phones, while the EPC is the doted numbers {1,2} for SGW, PGW and MME respectively (mentioned in 4.1). Macro base station is placed at (0,0) and micro base stations are placed considering the center (0,0) and at distance r = 200 m successively in sub-multiples of ‘r’. UEs are also placed in a cyclic manner considering the same distance r, but with a random way. Depending on the number of UEs, random angles are created for a specific number of UEs N = {10, 15, 20} and is used in the x-axis, while the y-axis uses random values between 0 and 1.

In NS-3 the transmission power is set by assuming that a base station - evolved Node B (eNB) maintains a constant transmit power regardless of the specific operational context. This simplification is common in many simulation environments to reduce complexity of modeling wireless communication systems. So, the equation of calculating the power is:

$$P_{all}(u, u_1) = (u-1)P_m + u_1 P_M \qquad (4.4)$$

Where u indicates the total number of active (transmitting) micro eNBs, u_1 the state of macro eNB (active or not), P_m is the transmission power of micro base stations and P_M is the transmission power of macro base stations.

The average noise power is denoted as σ^2 and calculated as: $\sigma^2 = -174 + 10\,log(B) + NF$ for a given bandwidth B and Noise Figure. In our case bandwidth is set to 100 Resource Blocks (explanation of RB in previous section - Resource Allocation) and corresponds to 20 MHz and Noise Figure is set zero. The frequency in which the eNBs and UEs operate is set at 36000 Earfcn [6] (Band 33 TDD) which corresponds to 1.9 GHz.

Channel conditions between eNB and UE are modeled using modules Nakagami along with LogDistance (see previous section). Nakagami with m parameters equal to 1 denotes the small-scale fading coefficient, following a circular symmetric complex Gaussian with unit variance. Log Distance's path-loss exponent is β and distance between eNB b and UE n is denoted by d, while d_0 is the reference distance and set to 1 m. In this case we have a Rayleigh distribution for flat fading.

Main point was to properly associate UEs to eNBs. Running the matlab project of the reference in which the user association problem was studied, the truth tables for each case were extracted and used. Firstly, all components are created using the appropriate libraries and function calls. The eNBs are created in ascending order and placed in defined positions. Then, a list of the created eNBs is kept and based on the truth table we assign (attach method) UEs to eNBs. Thus, we ensure that each unique UE is attached to one eNB while an eNB can hold many UEs.

The application was set as Udp server – client, configuring the number of packets to be transferred, the packet size and the delay, as well as the simulation time to 2 seconds. Insights to the actual throughput during the simulation, the packet loss and other useful metrics are obtained. The achievable rate measured in bits/seconds/Hz for downlink between user n and base station b is:

$$R_{b,n} = \frac{1}{m} log_2 \left(1 + \frac{u_b h_{b,n} P_b}{\sigma^2 + \sum_{j \epsilon B \setminus b} u_j h_{j,n} P_j} \right) \qquad (4.5)$$

Where user $n \in \{1, \ldots, N\}$, base station $b \in \{1, \ldots, B\}$, m denotes the number of UEs served by eNB b. Numerator of "log()" function corresponds to received Power (assuming fading) at UE of a specific set eNB – UE and the denominator corresponds to noise occurring on UE due to rest eNBs. This division is actually the Sinr of eNB – UE pair.

Then, energy efficiency of a heterogeneous network measured in bits/Hz/Joule is:

$$\eta = \sum_{n \in N} \frac{\sum_{b \in B} R_{b,n} x_{b,n}}{P_{all}(u, u_1)} \qquad (4.6)$$

Where the variable $R_{b,n}$ is calculated right above (4.5), $x_{b,n}$ denotes the association of user n to eNB b and P_{all} is total power also calculated (4.4).

```
dev_pool = []
# Loop through each BS in the array
# and keep the id of each BS
for i in range(num_enb):
    dev = microDevs.Get(i)
    dev_pool.append(dev)

# Based on the truth table assign
# each UE to corresponding eNB
lines = arr_to_use
for r, line in enumerate(lines):
    # Loop through each UE for the current BS
    for j, element in enumerate(line):
        # if we are in the first row
        # we check if user is asssociated
        # with macro base station
        if (r == 0 and element != 0):
            # Associate UE (UE j) with macro BS
            lteHelper.Attach(ueDevices.Get(j), macroDev.Get(0))
        elif (r != 0 and element != 0):
            # Associate UE (UE j) with micro BS (BS i - always one less
            lteHelper.Attach(ueDevices.Get(j), dev_pool[r-1])
```

Figure 4.2.2 - User association with attach method

Above figure reflects the method of attaching UEs to eNBs based on the truth table mentioned earlier.

There were 3 different algorithms used. The proposed approach is a distributed solver with correctness and convergence guarantees. The IE algorithm [7] utilizes a distributed protocol for the sub-gradient method. Finally, the SC algorithm [8] is a matrix-based EE algorithm with low complexity.

Each algorithm was run in a single scenario for N = {10, 15, 20} UEs, with B = 10 eNBs and for 200 independent experiments. In each experiment, the position of UEs randomly changes, as well as the small-scale fading. Scripts had a long running time and high complexity, so the optimized profile of Ns-3 was preferred. Each scenario was a 5-hour long simulation after optimization of NS-3.

At last, results were gathered and provided the charts in Energy Efficiency and Spectral Efficiency respectively. Simulation results occur by calculating the cumulative distribution function (described in Appendix) of Energy Efficiency (EE) and Spectral Efficiency. EE in networking, refers to the ability of a network or its components to deliver data and services while minimizing energy consumption. It's an essential consideration in today's world, where energy consumption and environmental impact are significant concerns.

CHAPTER 5: Analysis of Results

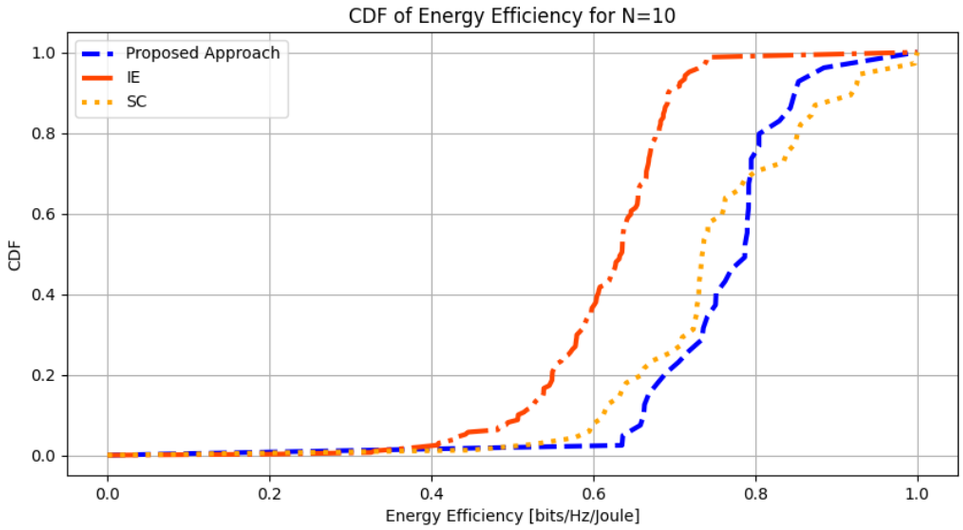

Figure 5.1.1 - Cdf of Energy Efficiency with 10 BS and 10 Users

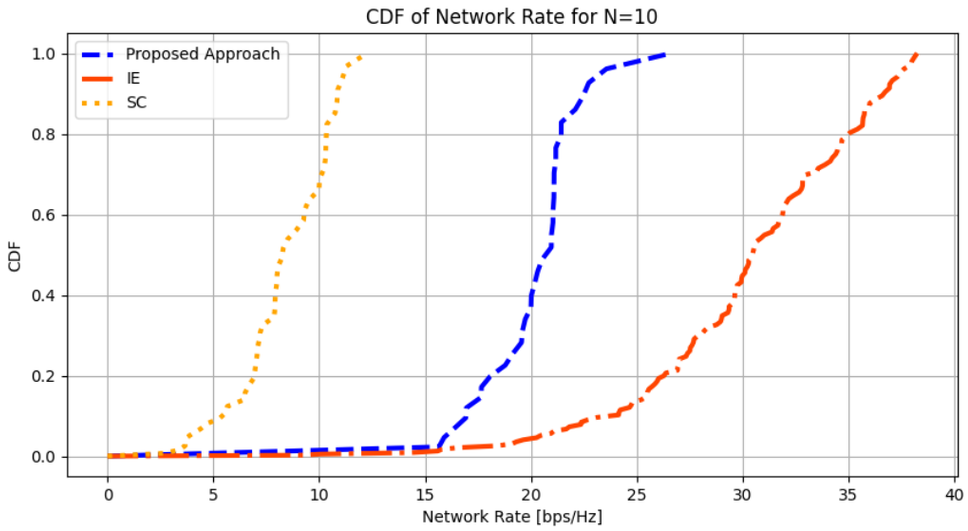

Figure 5.1.2 - Cdf of Spectral Efficiency with 10 BS and 10 Users

Algorithms introduced in previous chapter are SC (matrix-based), IE (distributed for sub-gradient) and Proposed Approach (a distributed solver based on belief propagation message passing between base stations) in color orange, red and blue respectively.

In present case 10 base stations and 10 users interacted. Figure 5.1.1 - Cdf of Energy Efficiency with 10 BS and 10 Users displays IE's curve steeper than SC's and Proposed approach at value 0.8, meaning IE's probability its values being lower or equal than 0.8 is higher, while SC's probability is lower and Proposed Approach values are not as likely less than 0.8.

The chart in Figure 5.1.2 - Cdf of Spectral Efficiency with 10 BS and 10 Users represents IE with sufficient spectral efficiency and SC with the worst network rate Proposed Approach is located between them.

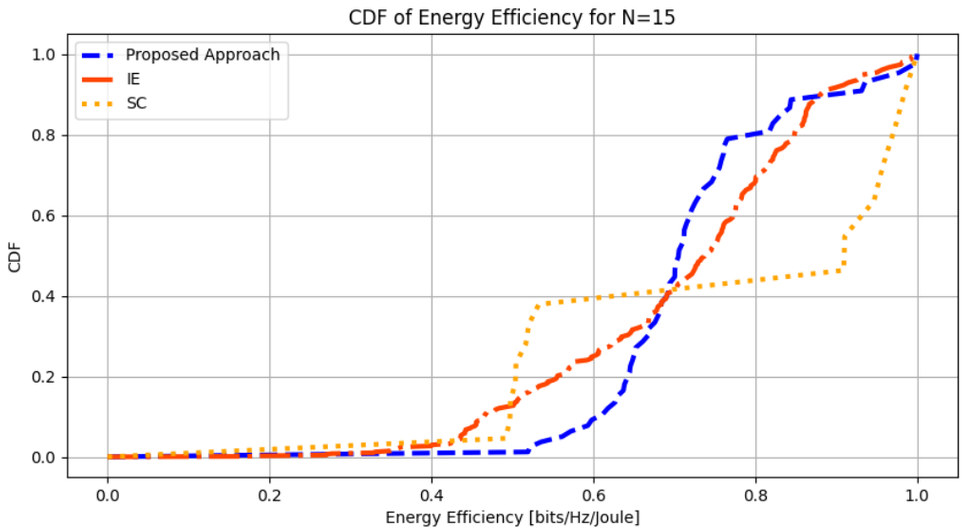

Figure 5.1.3 - Cdf of Energy Efficiency with 10 BS and 15 Users

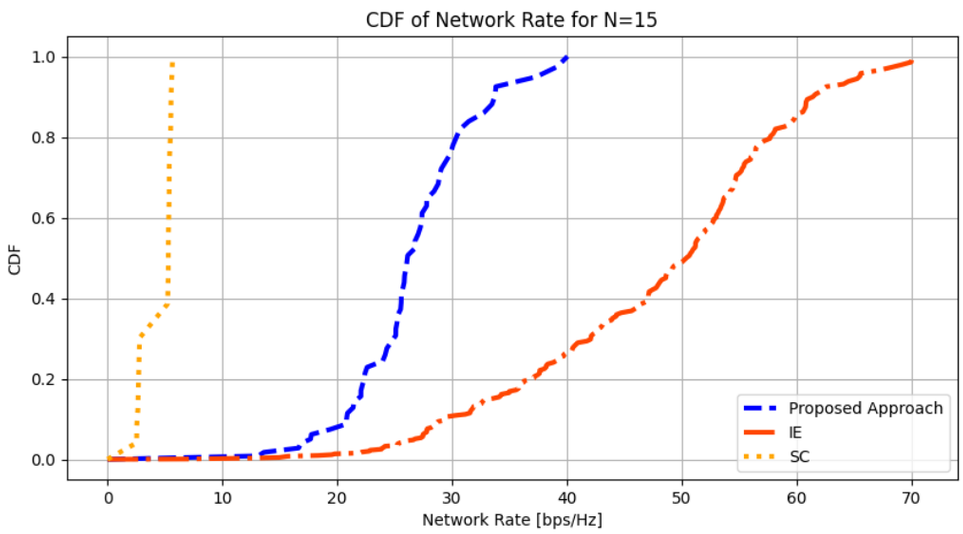

Figure 5.1.4 - Cdf of Spectral Efficiency with 10 BS and 15 Users

Case second is about 10 base stations and 15 users. As mentioned three algorithms were compared. In Figure 5.1.3 - Cdf of Energy Efficiency with 10 BS and 15 Users SC's probability values less or equal than 0.7 exceed IE's and in turn Proposed approach. Behavior is differentiated above value 0.7. Proposed approach probability's values lower than 0.8 surpasses both SC and IE whose probability remain relatively constant between them. Figure 5.1.4 - Cdf of Spectral Efficiency with 10 BS and 15 Users reveals once again IE with optimal spectral efficiency, followed by proposed approach and SC as in the first case.

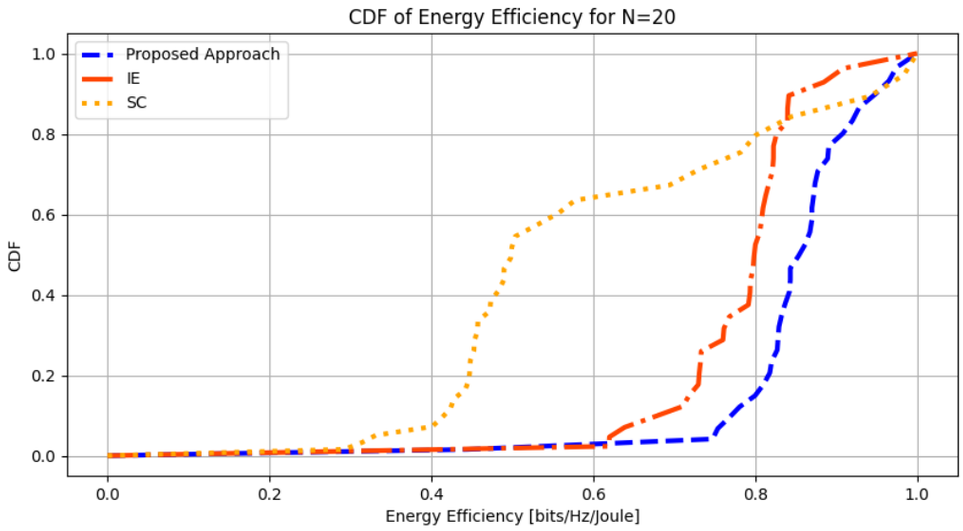

Figure 5.1.5 - Cdf of Energy Efficiency with 10 BS and 20 Users

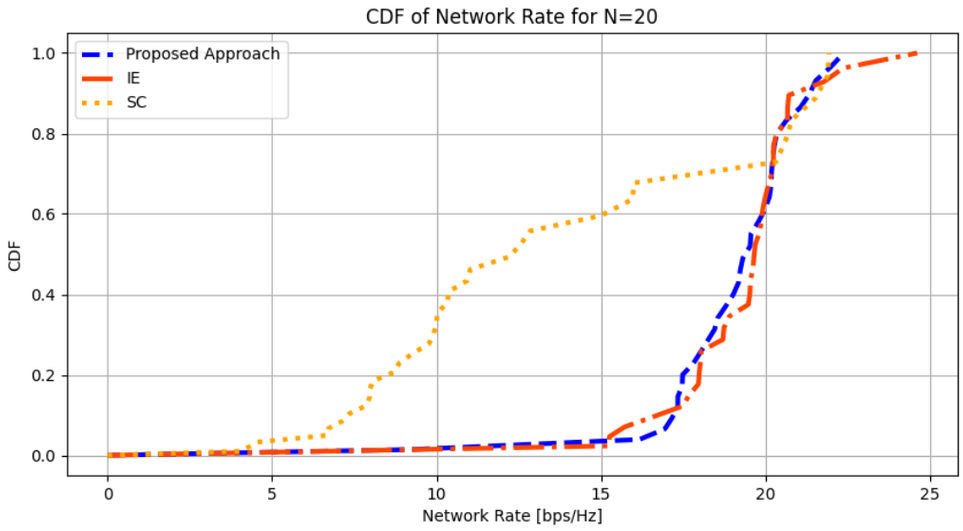

Figure 5.1.6 - Cdf of Spectral Efficiency with 10 BS and 20 Users

Lastly, with 10 base stations and 20 users comparison between algorithms provide remarkable results. Probability of SC values lower or equal to 0.8 hyper passed both IE and Proposed. Figure 5.1.5 - Cdf of Energy Efficiency with 10 BS and 20 Users displays Proposed Approach with the least probability. Chart in Figure 5.1.6 - Cdf of Spectral Efficiency with 10 BS and 20 Users differs to previous cases with IE and Proposed Approach being identical, while SC still remains worst.

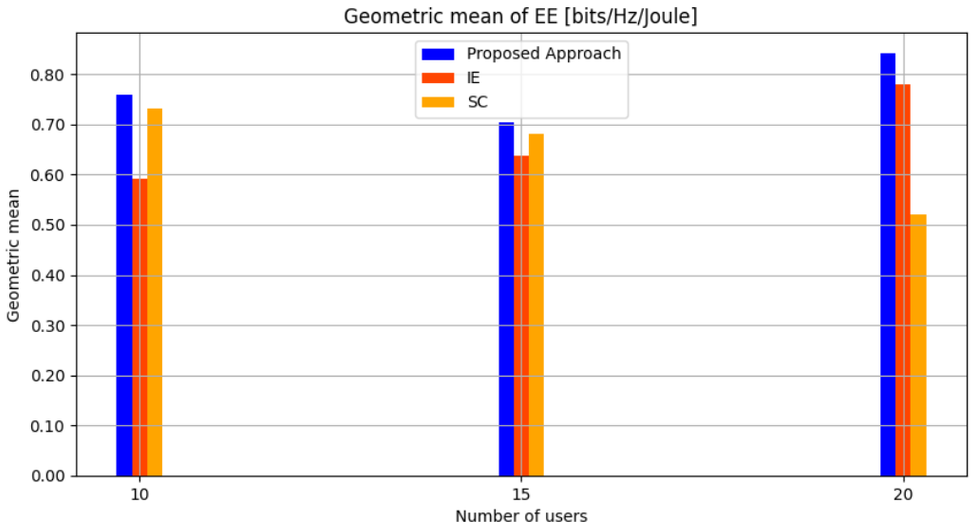

Figure 5.1.7 - Geometric Mean of Energy Efficiency

Geometric mean of energy efficiency in networking is a mathematical measure used to assess the overall energy efficiency of a network or a set of network devices, typically in the context of data communication and information technology. Provides a way of representing the energy efficiency of various network components or systems as a single, aggregated value. The geometric mean is a specific type of average calculated by multiplying a set of values together and taking the n^{th} root of the product, where n is the number of values. In the context of energy efficiency, it is applied to account for the multiplicative nature of power consumption.

Performance of the proposed approach algorithm is compared to the state-of-the-art algorithms IE and SC. Proposed method's geometric mean outperforms both IE and SC for all N values as seen in Figure 5.1.7 - Geometric Mean of Energy Efficiency.

Qualitatively the results are similar to reference study, especially in 10 and 15 UEs, with a slight difference at 20 users. There is a considerable variation quantitatively in results. Major difference is the calculation of P_{all} (4.4). As mentioned, in NS-3 the transmission power is constant and does not consider operational context. So, this quantity is considered in calculating Energy Efficiency at equation (4.6). It is common sense that as a denominator is reduced the result is increased.

Also, this variation, as parameters remain constant, is due to the simulation environment and hardware or operating system. Most significant is the different environments. NS-3 is highly accurate in calculations due to the models used and is specially designed for network communication, wireless networks, protocols and mobility modeling, offering fine-grained network modeling and simulation with detailed protocol models, unlike Matlab which is a more general tool with wide range of applications such as signal processing etc. Different hardware or software architectures can affect simulations due to available amount and speed of RAM required in large amount of data to be loaded and processed. Simulations that require large-scale data processing can benefit from more powerful CPUs.

Chapter 6: Conclusions

In summary, present work introduced network simulator 3, offering a competent user guide in python, showing its architecture and basic elements. Scenarios of increasing difficulty and complexity are provided for both wireless and wired networks. Also, implements an application example of energy efficiency maximization in heterogeneous LTE networks, studying the assignment of users to base stations and comparing three state-of-the-art algorithms. Measurements of occurred simulation were compared against empirical cumulative distribution functions for spectral and energy efficiency (in bps/Hz and bit/Hz/Joule respectively). Overall, NS-3 can simulate complex scenarios, but at the expense of learning time.

As a future work we could study the influence of movement from users or increase the number of users making a more complex heterogeneous network for a more realistic scenario or even use real time devices, since NS-3 provides this possibility, with immediate practical application.

Appendix

Operating System Requirements And Installation Guide

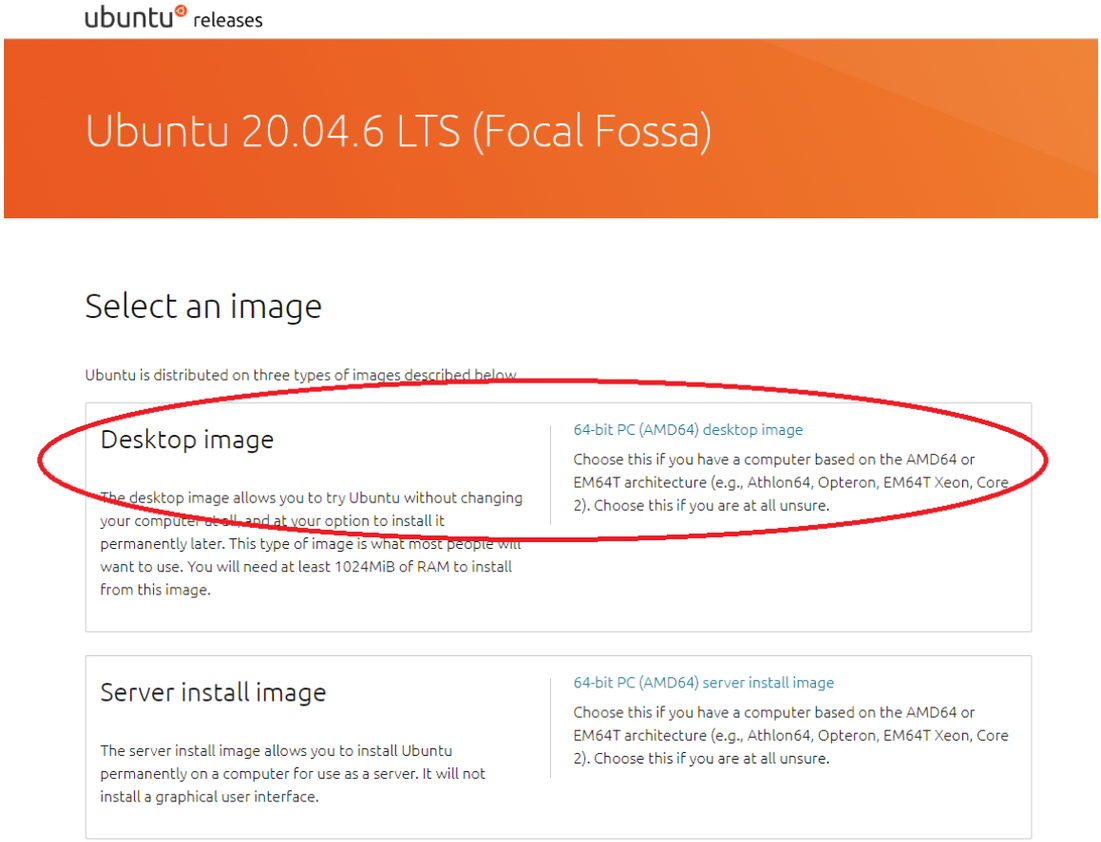

Figure 7.1 - Download Ubuntu software [11]

For Windows operating system, we can use a Virtual Machine and install Ubuntu/Debian Linux distributions (as we did in our case). Following the guidance of installation is easy, just download any iso version of Linux and proceed as mentioned [9]. Another option is to use Windows as Linux with WSL [10] (I do not recommend it as it is complex and anyone without much knowledge and experience might be confused).

Otherwise, if we have Linux as primarily operating system it is fine. Then, there are a few things we need to consider as mentioned below. Just be careful with older versions of OS, the supported packages might not match with newer versions of NS-3. So, the versions must be compatible with each other. You may download the Ubuntu OS as shown above.

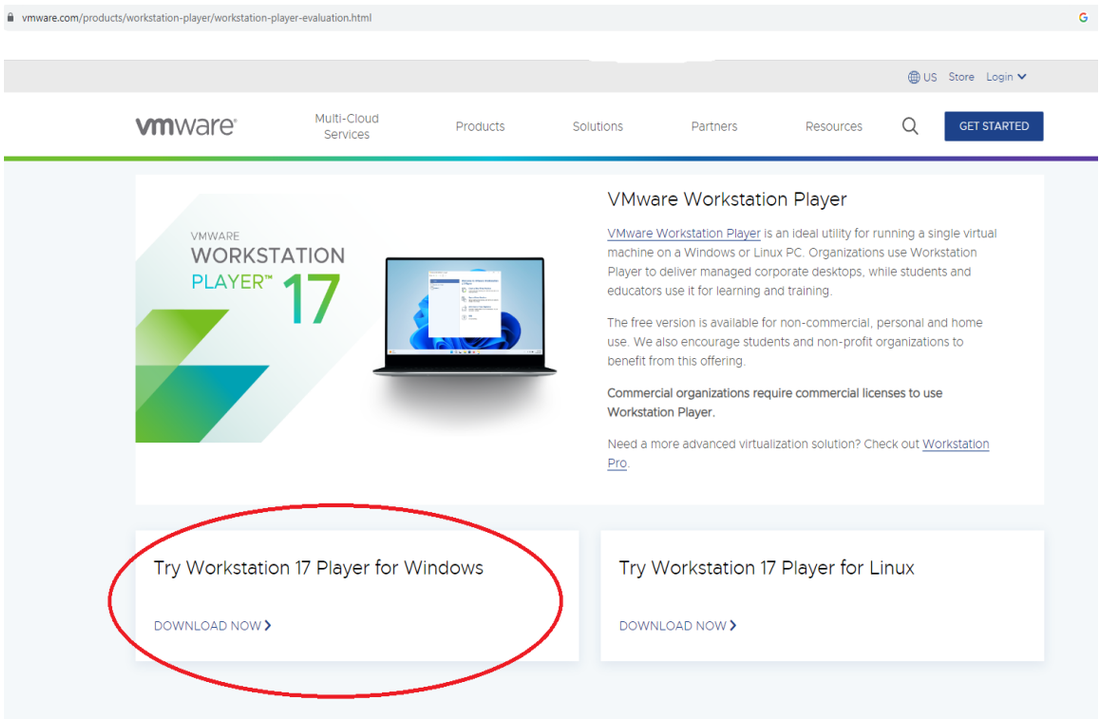

Figure 7.2 - Download VMware [12]

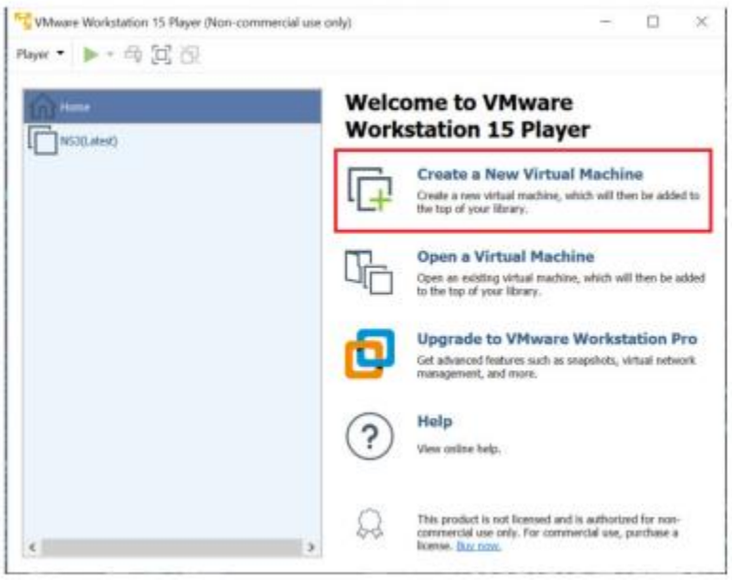

Figure 7.3 - Create new virtual machine [13]

Download and install the Virtual machine. Then open the VM.

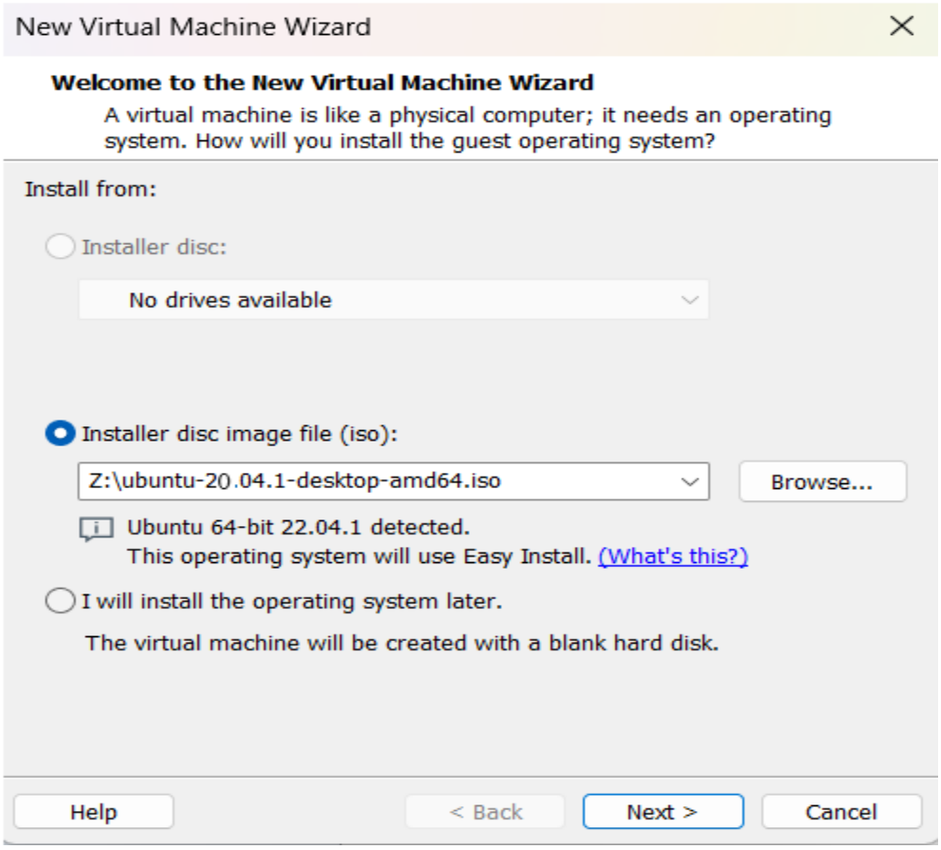

Figure 7.4 - Select Operating System [9]

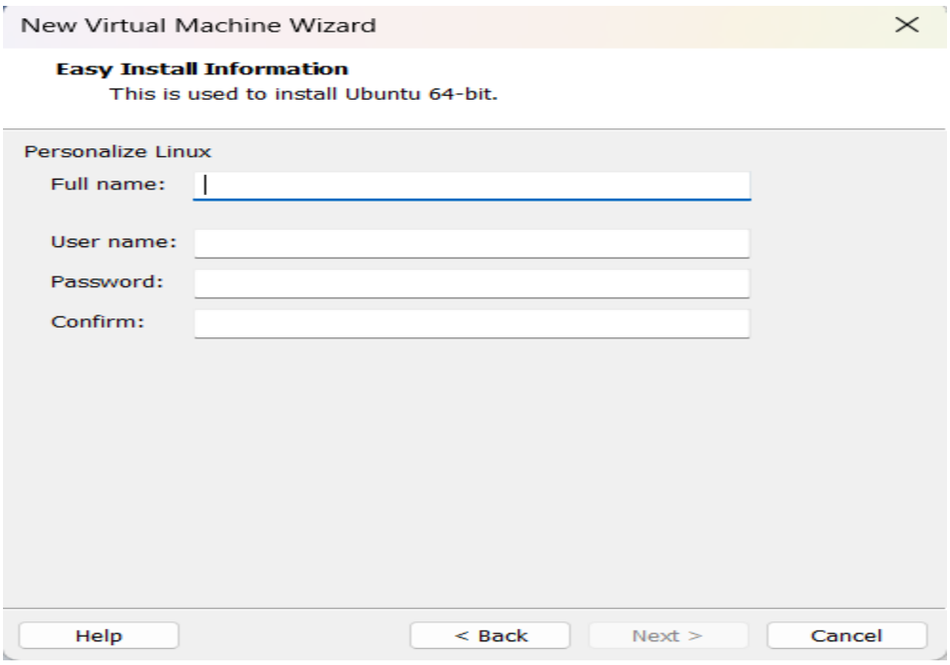

Figure 7.5 – Enter Account Details [9]

Create a new virtual machine and browse the ISO file we downloaded earlier. Enter details and proceed.

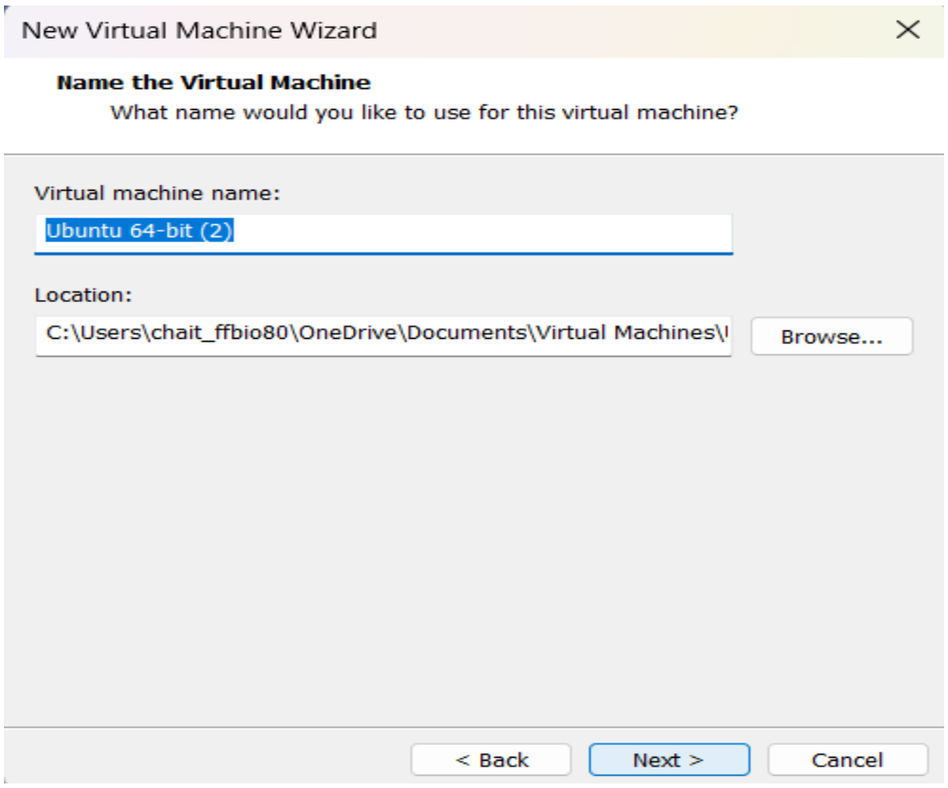

Figure 7.6 - Name VM [9]

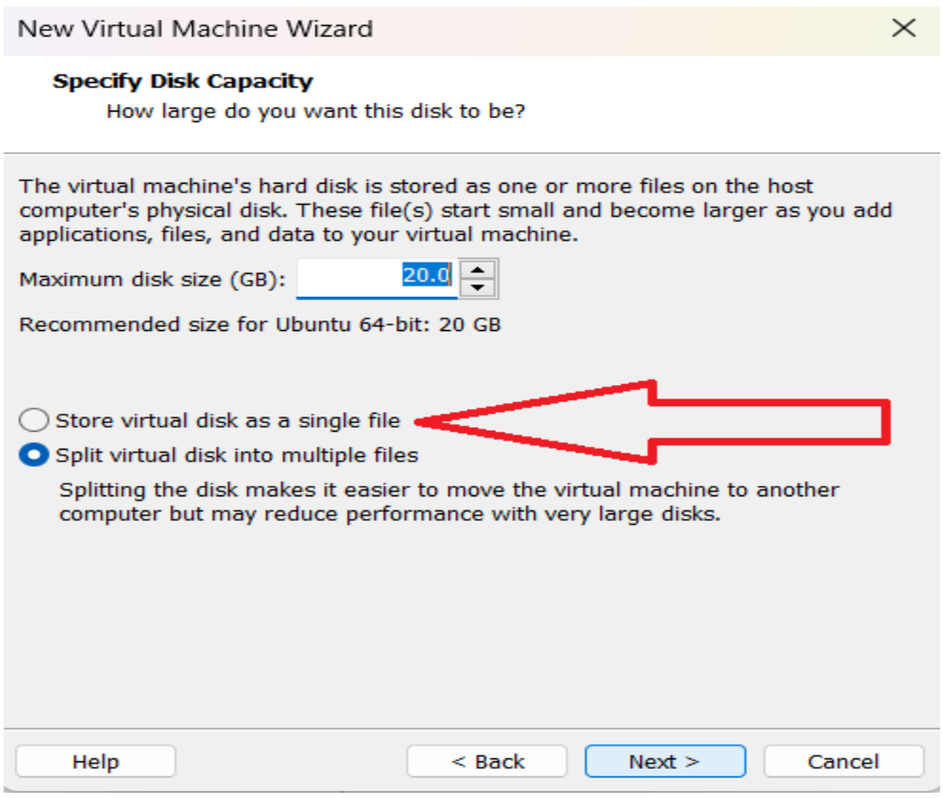

Figure 7.7 - Select disk space [9]

Name the machine and Specify disk capacity.

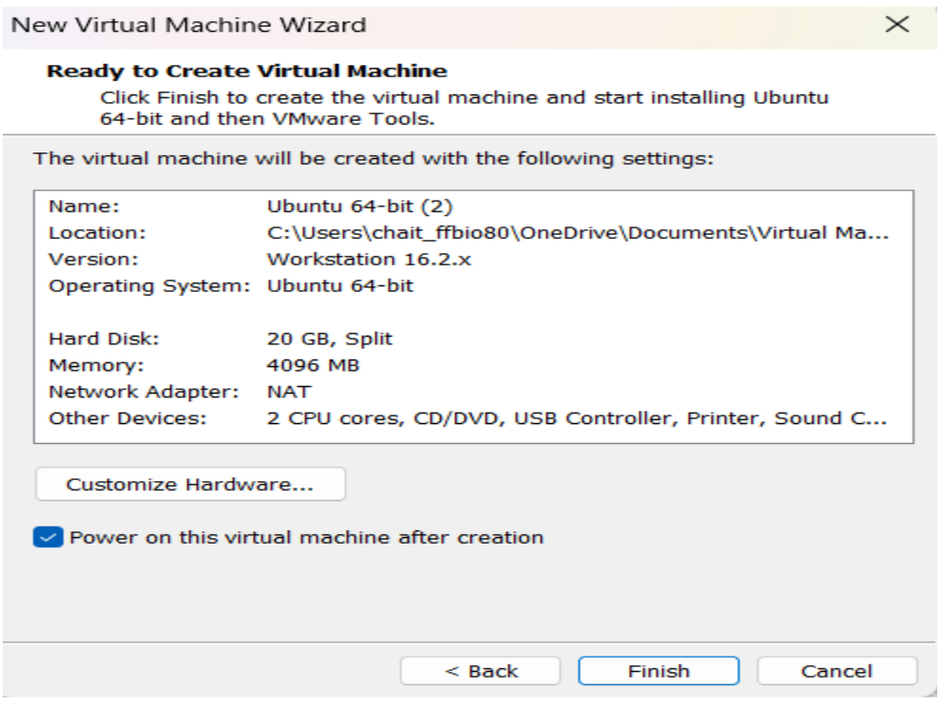

Figure 7.8 – Finish VM setup [9]

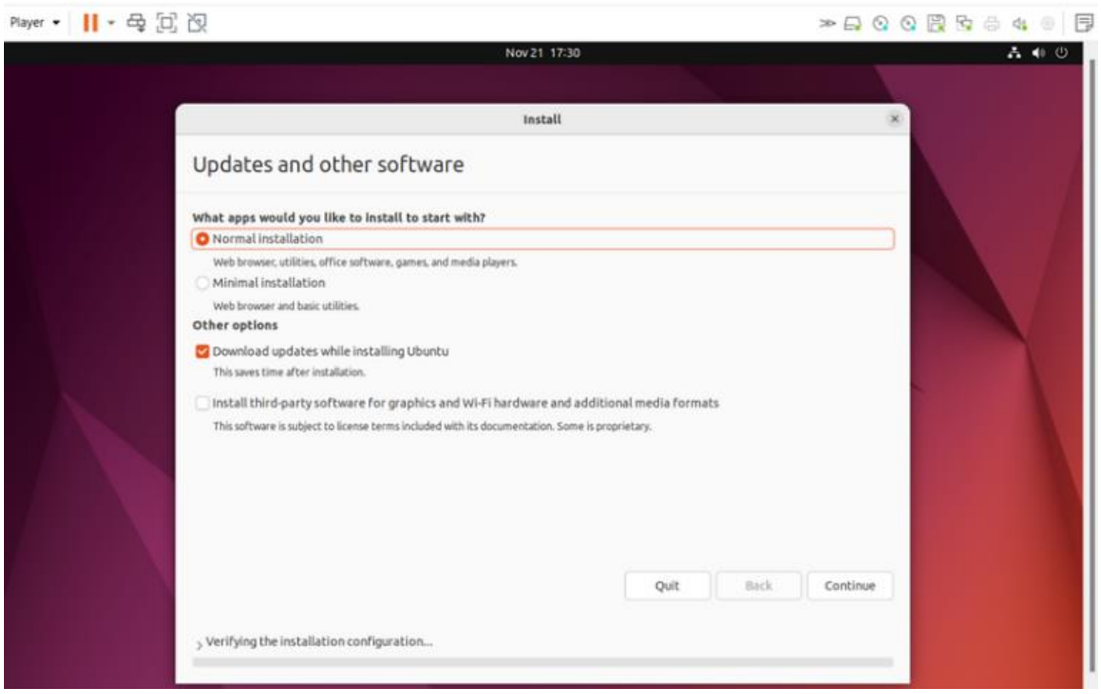

Figure 7.9 - Install Ubuntu [9]

If necessary customize the hardware and proceed. The procedure is known as in every other installation of operating system.

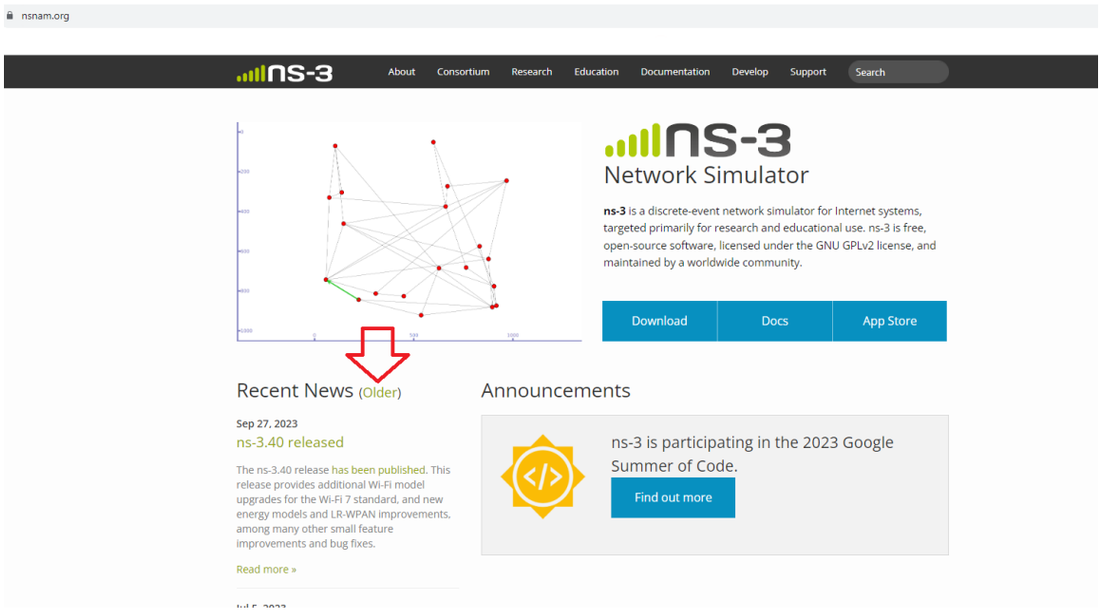

Figure 7.10 - Visit official website of NS-3 [2]

After the whole setup is complete and we have the OS installed, open the browser (Firefox). Visit the official website of ns-3 [2] and download any version of choice. Follow the rest of installation [13]. At this point, we need to install a few libraries and packages using the terminal (command line):

sudo apt-get install g++ python3 python3-dev pkg-config sqlite3 python3- setuptools git qt5-default mercurial gir1.2-goocanvas-2.0 python-gi python-gi-cairo python3-gi python3-gi-cairo python3-pygraphviz gir1.2- gtk-3.0 ipython3 openmpi-bin openmpi-common openmpi-doc libopenmpi-dev autoconf cvs bzr unrar gdb valgrind uncrustify doxygen graphviz imagemagick texlive texlive-extra-utils texlive-latex-extra texlivefont-utils dvipng latexmk python3-sphinx dia gsl-bin libgsl-dev libgsl23 libgslcblas0 tcpdump sqlite sqlite3 libsqlite3-dev libxml2 libxml2-dev cmake libc6-dev libc6-dev-i386 libclang-6.0-dev llvm-6.0-dev automake python3-pip libgtk-3-dev synaptic vtun lxc uml-utilities

Then we need to run: sudo pip3 install cxxfilt. This is critical for ns-3 to work properly and differs from version to version of ns-3 (closely look at the documentation of the version we are about to use).

Finally, we extract the ns-3 zip file in a desired folder/workspace. After that, we enter the folder/workspace and there are some sub folders along with some files. We build the project with the "build.py" file using the command './build.py --enable-examples --enable-tests'. Then, we enter in the directory with the same name 'cd <name of ns3 folder>'. In that folder we can either run './test.py' or './waf configure' because we must configure some parameters before start executing any other scripts. First case './test.py' requires some time but can also run test scripts and provide useful info about the proper installation of ns-3.

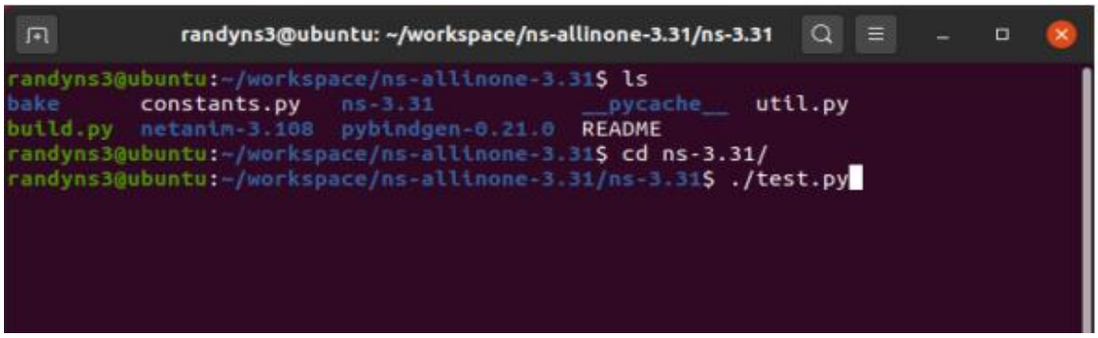

Figure 7.11 - Test the build and installation [13]

If we are not familiar with the simulator or if we create simple scripts, we can have the default profile of execution. In case we have complex functionalities in our script, then we must use the optimized profile for faster execution as: './waf configure --build-profile=optimized'.
We definitely have to install an editor for scripting, in our case we used "Sublime Text". You may use any of your preference.

Build NetAnim for visual representation of topologies requires the qmake package in the following process:

a. Go to NetAnim directory pasting these commands in the terminal subsequently:
   I. cd
   II. cd workspace
   III. cd <ns folder name>
   IV. cd <netanim folder name>
b. Clean make files using the command: "make clean"
c. Make NetAnim using the commands: "qmake NetAnim.pro" and then "make"
d. Test the NetAnim installation by pasting the following command in the terminal, while within the netanim directory: "./NetAnim"

If NetAnim opens, congratulations! NetAnim is now installed.

Function calculating throughput in script 3.2:

```
def calculate_throughput(stats):
    time_diff = (stats.timeLastRxPacket.GetSeconds() - stats.timeFirstTxPacket.GetSeconds())
    # Calculate throughput for the given flow stats
    if time_diff != 0:
        return (stats.rxBytes * 8) / (time_diff) / (1024 * 1024)
    return 0.0
```

Description of CDF mentioned in 4.2

The CDF (Cumulative Distribution Function) is a concept used in probability and statistics. It is a function that provides information about the probability distribution of a random variable. In particular, the CDF of a random variable X, denoted as F(x), gives the probability that X will take on a value less than or equal to x.

Table 4.2: Simulation parameters

Description	**Value in NS-3**	**Corresponding value**
Micro eNB transmit power	21.14 dBm	21.14 dBm
Macro eNB transmit power	37.99 dBm	37.99 dBm
Thermal noise σ^2	-101 dBm	-101 dBm
Bandwidth	100 RB	20 MHz
Carrier frequency	36000 Earfcn	1.9 GHz
Path loss exponent β	4	4

Bibliography

[1] H. Gao, J. S. Bawa, R. Paranjape, Analysis of Acquired Indoor LTE-A Data from an Actual HetNet Cellular Deployment. Wireless Pers Commun, vol. 114, issue 1, pages 545–563, 2020.

[2] Nsnam. Retrieved 10 11, 2023, https://www.nsnam.org/docs/release/3.31/models/ns-3-model-library.pdf.

[3] M.A. Ismail, G. Piro, L.A. Grieco, T. Turletti. An improved IEEE 802.16 WiMAX module for the ns-3 simulator, SIMUTools 2010 Conference, pages 1-10, March 2010.

[4] J. Farooq and T. Turletti, “An IEEE 802.16 WiMAX module for the NS-3 Simulator,” SIMUTools 2009 Conference, March 2009, Rome, Italy.

[5] R. Chatzigeorgiou, and A. Bletsas, "Inference-based, Energy Efficient User Association with Convergence Guarantees", IEEE International Conference on Communications (ICC), May-June 2023, Rome, Italy.

[6] I. Poole, *LTE Frequency Bands & Spectrum Allocations*. Retrieved 10 11, 2023 http://www.radio-electronics.com/info/cellulartelecomms/lte-long-term-evolution/lte-frequency-spectrum.php.

[7] D. Liu, L. Wang, Y. Chen, T. Zhang, K. K. Chai, and M. Elkashlan, “Distributed Energy Efficient Fair User Association in Massive MIMO Enabled HetNets,” IEEE Commun. Lett., vol. 19, no. 10, pp. 1770–1773, Jul. 2015.

[8] G. Lee and H. Kim, “Green Small Cell Operation of Ultra-Dense Networks Using Device Assistance,” Energies, vol. 9, no. 12, pp. 1–19, Dec. 2016.

[9] GeeksforGeeks. Retrieved 10 11, 2023, https://www.geeksforgeeks.org/how-to-install-ubuntu-on-windows-using-vmware/.

[10] S. Carlos, HOW TO INSTALL WSL ON WINDOWS *IEEE Software*, 10, 2019, https://works.bepress.com/smith-carlos/5.

[11] ubuntu. Retrieved 10 11, 2023, https://releases.ubuntu.com/focal/.

[12] vmw. *vmware*. Retrieved 10 11, 2023, https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html.

[13] A. S. Randy. Retrieved 10 11, 2023, https://engineering.fresnostate.edu/research/bulldogmote/documents/NS3%20Tutorial%20Installation%20%20Randy.pdf.