UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Energy Consumption in Wireless Mesh and Wireless Ad-hoc Networks

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Lluís Garcia Romero

Advisor: Luis J. De la Cruz Llopis

Barcelona, September 2015

Abstract

Wireless networks without infrastructure, as the Ad-hoc or Mesh networks, are nowadays being studied by many researchers, because of their enormous potential in different environments. In this TFG, we make a study by simulation of the performance and the energy consumption of the devices involved on a Mesh in different configurations. As a simulator, it is used NS-3, because it is a modern program with open code and modifiable. At the beginning, we thought it was well prepared for this purpose but the truth is that some resources on the energy module are missing. Also, since it is an open source program developed by volunteers and collaborators, in some cases the documentation is really poor or inexistent.

This lecture can be useful and interesting for those who want to start with NS3 programming or the energy consumption in simulations. After understanding the program and its functionalities, it will be exposed how the energy behaves in Ad-hoc networks and which problems could they have.

Resum

Les xarxes inalàmbriques sense infraestructura, entre les quals trobem les xarxes Ad-hoc i les xarxes tipus Mesh, són protagonistes actualment de nombrosos treballs d'investigació, degut al seu enorme potencial en múltiples entorns. En aquest TFG es realitza un estudi mitjançant simulació de les prestacions i el consum d'energia dels dispositius que formin part d'una Mesh en diverses configuracions. Com a simulador, s'ha usat NS-3, ja que és un programa modern, de codi obert i modificable. En un principi vàrem pensar que estaria ben preparat per a aquest propòsit, tot i que la realitat és que hi manquen alguns recursos en el bloc del consum d'energia. A més, al ser un programa obert i desenvolupat en gran part per voluntaris i col·laboradors, en alguns casos la documentació és molt escassa o nul·la.

Aquesta lectura pot ser útil i interessant per a aquells que desitgin introduir-se en la utilització de NS3, o en el consum d'energia en simulacions. Després d'entendre el funcionament del programa, s'exposarà com es comporta l'energia en les xarxes finalment Ad-hoc i quines dificultats poden presentar.

Resumen

Las redes inalámbricas sin infraestructura, entre las que se encuentran las redes Ad-hoc y las redes tipo Mesh, son objeto en la actualidad de numerosos trabajos de investigación, dado su enorme potencial en múltiples entornos. En éste TFG se realiza un estudio mediante simulación de las prestaciones y el consumo de energía de dispositivos que formen parte de una Mesh en diversas configuraciones. Como simulador, se ha utilizado NS-3, ya que es un programa moderno, de código abierto y modificable. En un principio pensamos que estaría bien preparado para éste propósito, aunque la realidad es que carece de algunos recursos en el bloque del consumo de energía. Además, al ser un programa abierto y desarrollado en gran parte por voluntarios o colaboradores, en algunos casos la documentación es muy escasa o nula.

Ésta lectura puede ser útil e interesante para aquellos que deseen introducirse en la utilización de NS3, o en el consumo de energía en simulaciones. Después de entender el funcionamiento de dicho programa, se expondrá cómo se comporta la energía en redes finalmente Ad-hoc y qué dificultades pueden presentar.

This work is dedicated to my family and closest persons in my life, who always gave me strength to keep forward in my studies.

Acknowledgements

I would like to thank the support given by my project supervisor (Luis J. De la Cruz) in the learning process through this work, and with the decisions we made to get over the difficulties found during this project. I also want to thank Marco Miozzo from CTTC (UPC) for his diligent help on some tasks with NS3 when we requested him.

Revision history and approval record

Revision	Date	Purpose
0	22/09/2015	Document creation
1	29/09/2015	First revision
2	05/10/2015	Second revision
3	10/10/2015	Third revision
4	15/10/2015	Fourth revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Lluís Garcia Romero	lluisgar16@hotmail.com
Luis J. De la Cruz Llopis	luis.delacruz@entel.upc.edu

Written by: Lluís Garcia		Reviewed and approved by:	
Date	14/10/2015	Date	15/10/2015
Name	Lluís Garcia	Name	Luis J. De la Cruz
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

1. Introduction

The mobile communications are clearly a great part of our lives nowadays. The number of users is increasing every year and the current infrastructure can be saturated in some situations. This work is based on some previous thesis which study the possibility to implement a mesh topology to deal with zones with a high density of users. Following this line, this project consists on studying the energy consumption on this networks and explain their viability.

We will use an open source program called NS3, explained forward, that will allow us to simulate and analyse the behaviour of this type of networks, but first, we will need to understand the program functionalities and the programming methods.

The initial plan was to develop this work in 6 months as the calendar fixed for the semester, but as explained in this report, some difficulties appeared that made us decide to postpone the deliver to the extraordinary date. All the work finally done can be found at the Gantt diagram on Fig.1.

The energy model in NS3 has a very poor documentation on how to use or configure it, so the understanding of this point of the project is what made us delay all the other tasks. Besides, the mesh models on NS3 are not supported to be used on the energy simulations of the program.

At this point we had to decide a new perspective for the project. We finally understood the energy model, but couldn't work with mesh networks. So we decided to use the most similar models that we had available: The Ad-hoc networks. This is the reason because this networks are more detail explained in this final report.

This document has a three part structure, the informative part for the networks and technologies used, the program functionality explanation and the simulations, and finally the results and conclusions.

The first part of this report, the chapter 2, consist on a state of the art chapter explaining the current Infrastructure, Ad-hoc wireless LANs and the Mesh networks to place the reader in the context. The next chapter is an introduction to the NS3 programming system and the functionality of its simulations. This part can be a very simple but effective tutorial on how to use NS3, and allows us to understand the simulations and results exposed on the chapter 4. On this chapter, we show how we studied the energy consumption in different detailed scenarios, and take conclusions for the project. This conclusions are explained on chapter 6, but the 5th one is somewhat apart from the main work. It is a budget study on what would be the cost to implement the experiments with real devices and real traffic. Finally, there is a Bibliography supporting the information exposed.

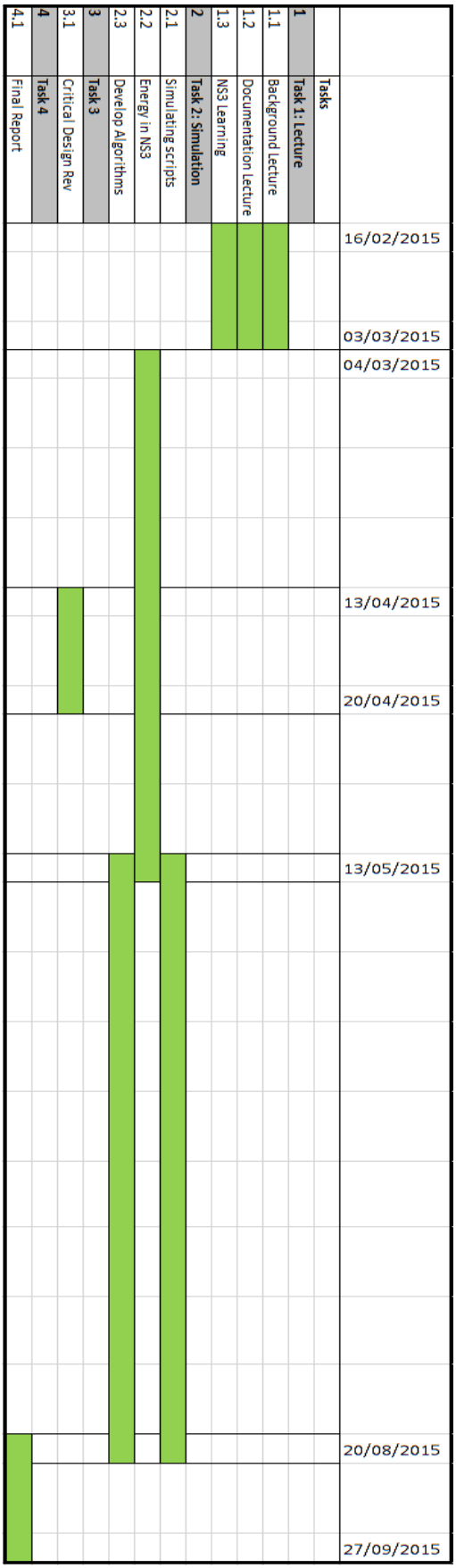


Fig. 1 Gantt Diagram

2. State of the art

2.1 - Background on Wireless LAN Networks

2.1.1 - Main features

A Wireless Local Area Network, or WLAN, is a network that allows to interconnect several devices using a wireless distribution method within a limited area. This gives the clients the ability to move around the area of coverage and to maintain the connection to the network. Modern WLANs are based on IEEE 802.11 standards, and are commonly known as the Wi-Fi brand name. The great perk that represents being able to connect to the network, usually with outside Internet connectivity, and without wire restrictions, makes this type of networks the most commonly used in particular homes or public buildings and spaces.

All devices that connect to a WLAN network, are called stations or STAs, and each are classified in 2 types: Access Points (APs), and Basic Stations (STs). APs are not always present in all WLANs but are usually wireless routers which forward packets to STs and could have interconnection with another network. STs are usually mobile phones or laptops which send and receive traffic data from other STs or an outside network. A set of STAs which communicate with each other is called a Basic Service Set, or BSS, and there are also 2 types of BSSs: The infrastructure BSS, and the independent BSS or ad-hoc network, which contains no APs and where STs have routing functionalities. A set of connected BSSs is called an Extended Service Set, or ESS, and the connection between access points in an ESS is called a Distribution System, or DS. [1]

2.1.2 - Infrastructure WLANs

Most Wi-Fi networks are deployed in infrastructure mode. As described before, in infrastructure mode, the BSS contains at least one station acting as AP and forwarding traffic data to STs. All STAs communicate with each other through the AP which controls the entire flux of traffic, and usually has an outside connection, so the network offers to clients an Internet connection.

On figure 2, an ESS of Infrastructure WLANs is graphically described.

2.1.3 - Ad-hoc WLANs

Networks in ad-hoc mode, are independent, which means the STs don't have the supervision of an AP and communicate with each other using routing functionalities, as seen in figure 3. This type of networks are self-configuring and dynamic because all STs are free to move so the routing capabilities are also adaptive. Ad-hoc networks require minimal configuration and are easily extended and quickly formed. This decentralized nature makes these networks suitable for a variety of applications. Nevertheless, STs usually have high mobility, causing that links are frequently broken and established, and not only are dynamically changing, but changing so fast it can be highly challenging to manage. Protocols will be required to do so. [2] [3]

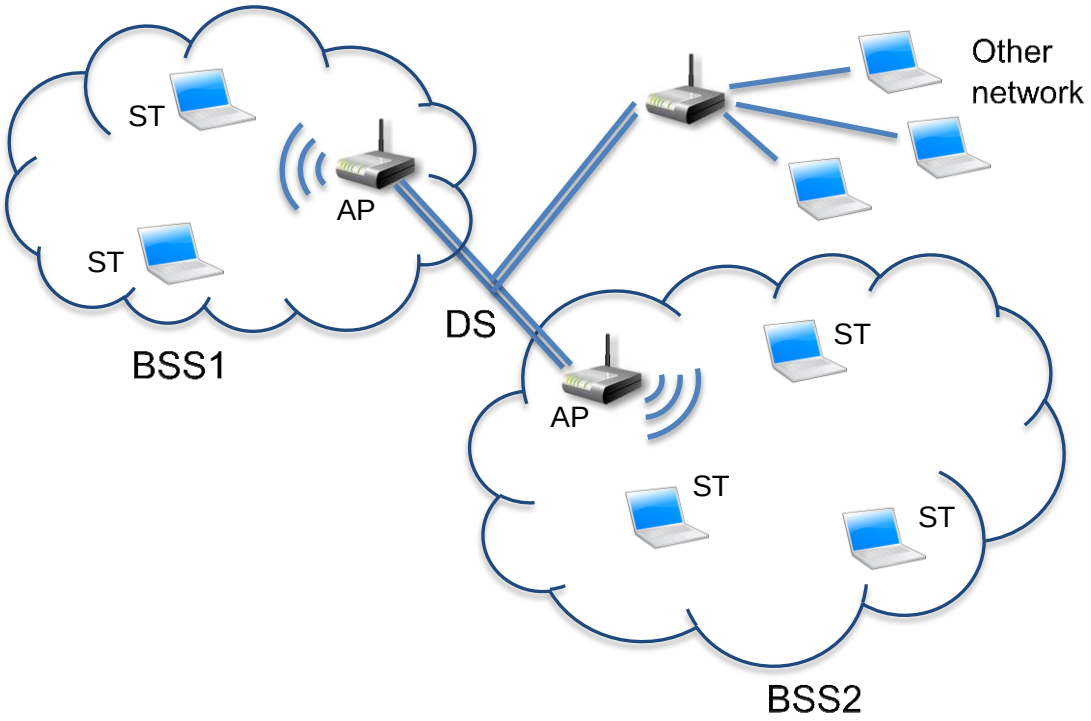


Fig.2 ESS of Infrastructure WLANs

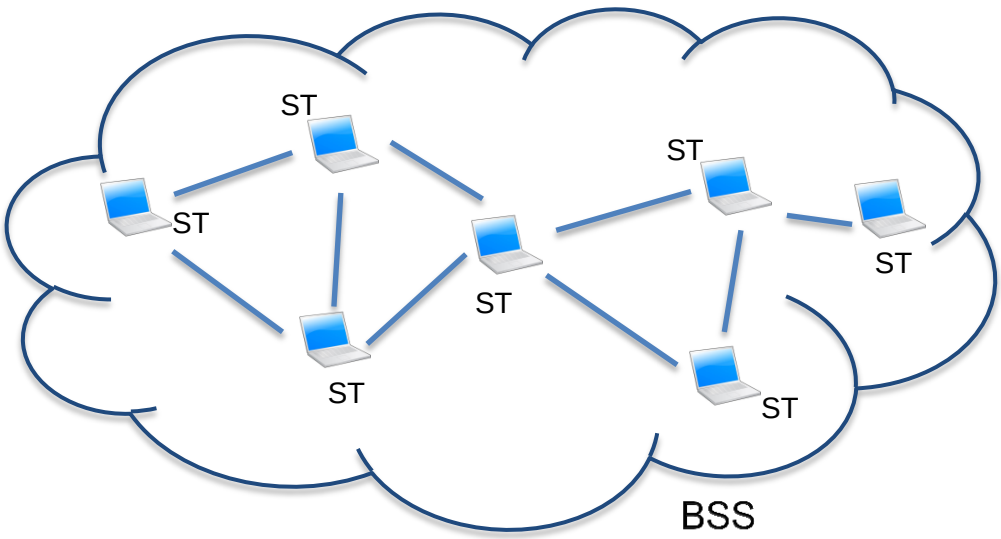


Fig.3 BSS of an ad-hoc WLAN

2.1.4 - Routing protocols

There are several types of protocols that can control an ad-hoc WLAN, but in this project we are going to work with only one of them, as described below:

2.1.4.1 - Proactive Routing

These type of protocols maintains a list of destinations and their routes by periodically distributing routing tables through the network, and allowing the multi-hop functionality by knowing each station the next-hop way to every other node. This type of protocols require a huge amount of data to be sent continuously and it grows exponentially with the number of nodes in the network. This protocols also have a slow reaction to link or node failures and restructuring.

In particular we will use the Optimized Link State Routing (OLSRv1) [4]. Using this protocol, the stations send repeatedly a HELLO frame to every neighbour node. This frames contain a list of the neighbours of the source, allowing the receiver to update its routing table to reach all two-hop-distance stations. Also periodically, each node broadcasts a Topology Control frame that informs about its nearby topology (the one obtained through the HELLO frames), and allows all the network nodes to calculate each routing entry in the table for all possible destinations. OLSR is a Distance-Vector protocol based on calculating the direction and distance to any link in a network. Direction refers to the next hop and interface where to forward traffic, and distance is a measure of the cost to reach the destination, usually the number of hops. The least cost route between two nodes is the path with minimum distance. Each node maintains a vector, or table, of minimum distance to every node.

2.1.4.2 - Reactive Routing

These protocols find a route on demand when a station wants to transmit by flooding the network with Route Request packets. This requires also a huge amount of data to be sent to the network but only when a station needs to transmit, and this means that there is a high latency time in route finding. A reactive protocol can also be used in another way, by just sending a packet to every link, and then forwarding it to every each other link except for the one it arrived on. This is called a Flooding Routing, it requires no routes to be known, but implies that every packet will be nearly broadcasted.

2.1.4.3 - Hybrid Routing

As the name suggests, it is a combination of the advantages of proactive and reactive routing. Hybrid protocols establish some proactive routes initially, and serves the demand from the additionally activated nodes through reactive flooding.

2.2 - Background on Wireless Mesh Networks

2.2.1 - Main features

Wireless Mesh Networks or, as we are forwardly referring to, WMNs, are self-forming multi-hop networks able to provide wireless connectivity between mesh clients through multiple mesh routers. Usually the mesh routers are fixed devices with more than one radio interface, and with an outside connection to provide Internet access to mesh clients. On the other hand, mesh clients are usually mobile devices not as powerful as routers, which could also provide routing and forwarding capabilities. The main objective of client meshing is to dynamically extend the coverage of the mesh backbone.

WMNs are fast and simple, because of its self-configuring capabilities, which allow the extending of the coverage without considerable effort. The nodes on a WMN detect their neighbours, node failures, poor channel conditions, and dynamically establish a next-hop network connectivity all by themselves.

All this functionality requires some protocols implemented at least on the link layer, and most of the research in mesh networking are based on IEEE 802.11, so we are focusing on the standardized IEEE 802.11s mesh networking technology. [5] [6] [7]

2.2.2 - IEEE 802.11s-based WMNs

The IEEE 802.11s mesh networking is an extension of the IEEE 802.11 that supports multi-hop mesh networking. It defines a path selection and forwarding mechanism at the link layer instead of the traditional layer 3 routing. There are also included mesh discovery, peering management, congestion control, beaconing and synchronization, etc.

The IEEE 802.11 Mesh architecture consists of 3 types of components, described below and on figure 4:

- The mesh stations or STAs, which include mesh functionalities to allow multi-hop communication. They actively participate in the generation of the mesh network establishing wireless links between their neighbours, receiving and forwarding traffic.
- The mesh Gate is a logical component that allows mesh networks interconnection, and also the integration with other non-mesh WLANs.
- The mesh Portal another logical component that allows interconnection with other non-IEEE 802.11 network technologies.

2.2.3 - Routing protocols and functionalities

2.2.3.1 - Mesh Discovery

The mesh discovery consists of a scanning process that provides the mesh profile information of the stations that periodically send Beacon frames. This mesh profile contains the configuration attributes of a mesh network such as the mesh ID, the path selection protocol, the path selection metric, the congestion control mode, the synchronization method, and the authentication protocol. All STAs must use the same mesh profile and must be synchronized on the functionality they are using.

2.2.3.2 - Mesh Peering Management Protocol

This protocol uses its own frames to open, manage and close links between STA neighbours in mesh networks. A peer link is established only if both stations have sent Peering Open requests and successfully received each confirm response.

2.2.3.3 - Mesh Path Selection and Forwarding

The path selection and forwarding mechanism is the main mesh functionality to provide multi-hop communications. The path selection protocol used is the Hybrid Wireless Mesh Protocol, which combines an on-demand path selection, with a proactive tree building mode.

As the reactive routing on WLANs, in on-demand mode, when a mesh station demands a path to another, a Path Request frame is broadcasted and propagated through the network. The destination station then responses with a unicast Path Reply frame, something similar to the ARP protocol.

In proactive mode, one station or more are configured as roots and the stations build a proactive tree to them. The roots periodically broadcast Path Requests frames, so the STAs receiving this frames update their paths to the root, and continue propagating the original Path Request frames. With this method, all STAs know at any time the distance to the root and the next-hop to it.

The tree to the root can also be built in another different way. In this mechanism the root propagates frames containing path metrics to the root, and if a STA wants to establish a path to it, it sends a unicast Path Request frame. Then the root replies with a Path Reply to create the path.

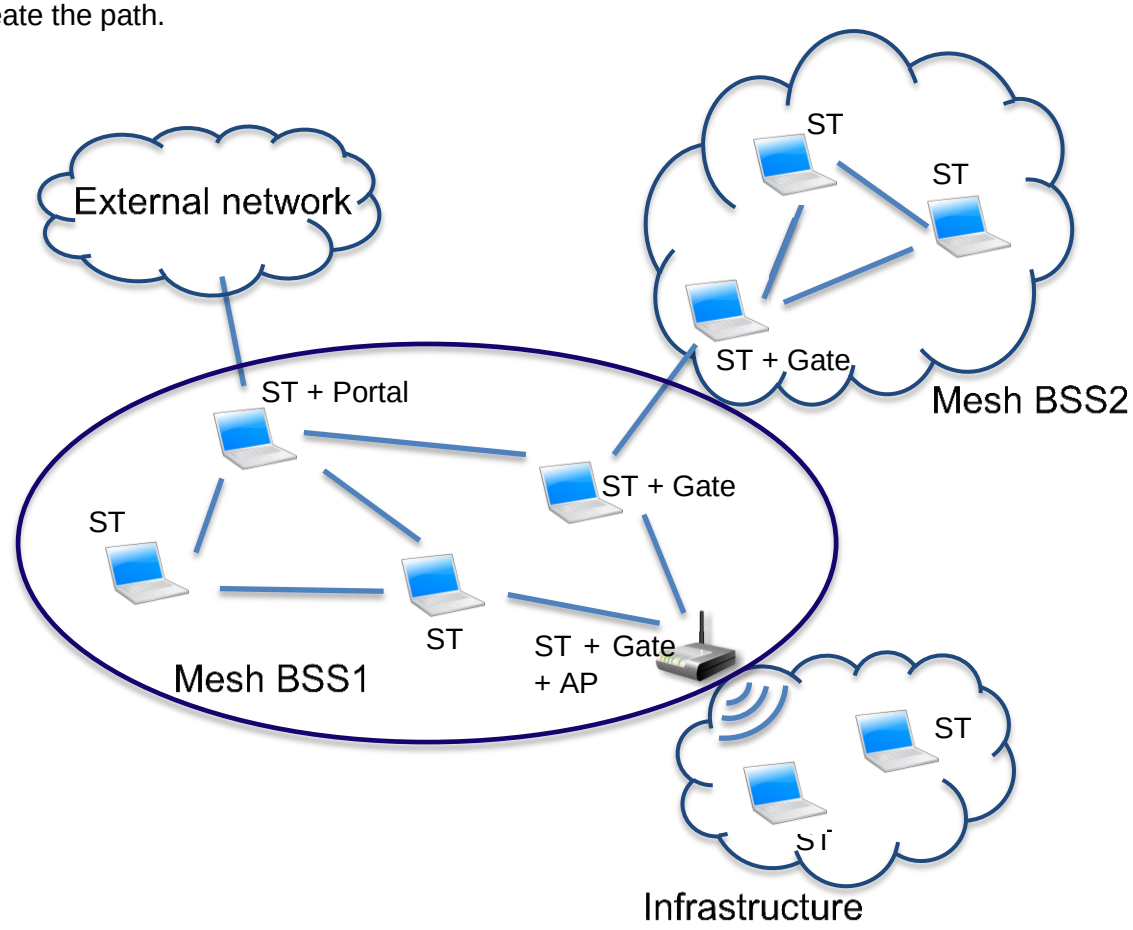


Fig.4 Mesh components architecture

3. Simulation and project development:

3.1 - About NS3

NS3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. NS3 is free software, licensed under the GNU GPLv2 license, and is publicly available on the web page [8] for research, development, and use. It is aligned with the simulation needs of modern networking research and allow a deep study of nearly all kind of networks using the commonly known C++ language.

NS3 contains a solid simulation core that adapts to every need of every network use the users may want to analyse. The simulation models are sufficiently realistic to allow NS3 to be used as a real-time network emulator. It also supports a real-time scheduler that allow users to emit and receive NS3-generated packets on real network devices, and can serve as an interconnection framework between virtual machines.

NS3 is continuously changing and adapting to new models, networks, and devices because the modern technology is also evolving. Every three months, a new stable version of NS3 is shipped with new models, and this development is done by a large community of users and developers. All the releases and documentation can be found at the same web page mentioned before.

3.2 - First simulations and program learning

The full code of all the scripts used in this project can be found on annexes.

For the first studio and to get familiar with the program and its functionality, we started doing an infrastructure Wifi simulation just as it is described in figure 5.

```
// Default Network Topology
//
//    Wifi 10.1.1.0
//                 AP
//   *     *     *
//   |     |     |
//  n0    n1    n2
```

Fig.5 First scenario

We will configure in this network an AP station that will forward packets from one station to another (n2 and n1), and the two STs sending a few packets to check the correct functionality. This packets will be generated and sent by a UDP application provided by the module “ns3/applications-module.h”.

After a few program configuration details, first of all we want to create the nodes we are going to use, and the channel in which we want those packets to be sent:

```
uint32_t nWifi = 2;
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi);
NodeContainer wifiApNode;
wifiApNode.Create(1);
```

This lines will create the three nodes needed using the NodeContainer class provided by Ns3 core.

```
YansWifiChannelHelper channel = YansWifiChannelHelper::Default();
YansWifiPhyHelper phy = YansWifiPhyHelper::Default();
phy.SetChannel (channel.Create());
```

For the first simulations we just want a default Wifi channel.

The next step is at the node level. We must assign to every node a MAC address, and for our simulations, also an IPv4 address and interface. We will give a virtual position to every node too:

```
NqosWifiMacHelper mac = NqosWifiMacHelper::Default();

NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes);
NetDeviceContainer apDevices;
apDevices = sifi.Install (phy, mac, wifiApNode);
```

We also set a default mac addressing and install it to our created nodes. This operation will create a device for each node called staDevices or apDevices if the node were a ST or the AP.

```
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                    "MinX", DoubleValue (0.0),
                                    "MinY", DoubleValue (0.0),
                                    "DeltaX", DoubleValue (5.0),
                                    "DeltaY", DoubleValue (0.0),
                                    "GridWidth", UintegerValue (1),
                               "LayoutType", StringValue("RowFirst"));

mobility.SetMobilityModel ("ns3::ConstantPositionModel");
mobility.Install(wifiStaNodes);
mobility.Install(wifiApNode);
```

We want at first the nodes to be fixed just to make the example as simple as possible, so we set the mobility model to a constant position model. In some scenarios we will experiment with that configuring a random mobility model. The DeltaX will give an inter-node distance of 5 meters, and the GridWith, set at 1, indicates the number of rows (Because it is set as a RowFirst layout) in the grid. This means that we will only have one row with three nodes at distance 5.

```
Ipv4AddressHelper address;
address.SetBase (“10.1.1.0”, “255.255.255.0”);
Ipv4InterfaceContainer wifiInterfaces = address.Assign(staDevices);
address.Assign(apDevices);
```

The Ip network address is the 10.1.1.0, and the mask given is just as seen above, the 255.255.255.0. So the two stations will have the addresses 10.1.1.1 (n0), 10.1.1.2 (n1), and 10.1.1.3 (AP).

Almost everything is configured now, we just need to install the applications working on each station which will send or receive packets:

```
UdpEchoServerHelper echoServer (9);
ApplicationContainer serverApps =
                  echoServer.Install(wifistaNodes.Get(0));
serverApps.Start(Seconds (0.0));
serverApps.Stop(Seconds (3.0));

UdpEchoClientHelper echoClient (wifiInterfaces.GetAddress (0), 9);
echoClient.SetAttribute( “MaxPackets”, UintegerValue (1));
echoClient.SetAttribute (“Interval”, TimeValue (Seconds (1.0)));
echoClient.SetAttribute (“PacketSize”, UintegerValue (1000));

ApplicationContainer clientApps =
                  echoClient.Install(wifiStaNodes.Get(1));
clientApps.Start(Seconds (1.0));
clientApps.Stop(Seconds (3.0));
```

We are using an echo application over UDP. The server is installed on the node 0 (n0 at the figure) and the client on the node 1 (n1). We set a start and stop time of each application in seconds, and the server app to be listening at port 9. The client will send only 2 packets with 1kByte length and with an interval of 1 second, so at 1.0s of the simulation time, the client app will start and instantly send one packet, which will be echoed if received successfully by the server. At time 2.0s the client node will send another packet, and at 3.0s the two applications will be stopped. To see all of this happening we can tell Ns3 to output those packets in a pcap format readable by a Protocol Analyser such as Wireshark:

```
phy.EnablePcap(“wifi_node1”, apDevices.Get (0));
```

Finally we just need the simulator to run and simulate the script:

```
Simulator::Run();
Simulator::Destroy();
return(0);
```

3.3 – The energy module in NS3

Working on the previous wifi simple script, we added the energy functionality to understand all the parameters and results involved during the simulation. To do so, there are a few lines added that were not explained before:

```
/** Energy Model **/
/*************************************************************************/
/* energy source */
BasicEnergySourceHelper basicSourceHelper;
// configure energy source
basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue
(5.0));
// install source
EnergySourceContainer sources = basicSourceHelper.Install
(wifiStaNodes);
/* device energy model */
WifiRadioEnergyModelHelper radioEnergyHelper;
// configure radio energy model
radioEnergyHelper.Set ("IdleCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("CcaBusyCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("TxCurrentA", DoubleValue (1.0));
radioEnergyHelper.Set ("RxCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("SwitchingCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("SleepCurrentA", DoubleValue (0));
DeviceEnergyModelContainer deviceModels = radioEnergyHelper.Install
(staDevices, sources);

/*************************************************************************/
```

At first we will need to configure the energy model, with all its parameters, and to install it on our virtual devices. The class BasicEnergySourceHelper provides us with a single Install method which we use on the station nodes (wifiStaNodes) after we set an initial energy double value (5.0 in this case). This function returns an EnergySourceContainer will all the energy sources installed.
After that, we have to configure the radio energy model. There are some attributes that refer to the current that will consume the energy of the device. After deep research, we have not been able to know how the 'Idle' and 'Ccabusy' current affect to the energy

consumption, so we will just use the 'TxCurrentA' or 'RxCurrentA' which mean the amperes used to receive and transmit packets. Actually the consumption in transmission is much greater than in reception, so we will just set the 'TxCurrentA' to 1 Ampere as a reference value to compare experiments and the other attributes to 0. The 'SwitchingCurrentA' refers to the current consumed when changing between transmitting and receiving states, and the 'SleepCurrentA' is the current when the device is in sleep mode or just not receiving or transmitting. At this point is where we had the main problems during this project. As it is seen, the WifiRadioEnergyModelHelper class is used in this case, and also has an Install method which requires two arguments: a NetDeviceContainer with only WifiNetDevices, and the EnergySourceContainer mentioned before. It seems logic that only wifi devices can be configured with a WifiRadioEnergyModelHelper, and if we take a look to the source code of this Install method, on the first lines we can notice it:

```
if (deviceName.compare ("ns3::WifiNetDevice") != 0)
{
  NS_FATAL_ERROR ("NetDevice type is not WifiNetDevice!");
}
```

The purpose of this project was to use the energy model on a mesh network, so we made a long search over the NS3 API to find out some way to implement it. But the radio energy model can only be configured on wifi devices for the moment and with the NS3 source code. There was only one way to do it, just to implement ourselves a MeshRadioEnergyModel, but we decided that it was out of the project priorities.

The basic energy sources used, contain a RemainingEnergy trace which informs us that some energy has been consumed. So we will want to connect this trace to some functions processing the information:

```
Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource>
    (sources.Get(1));
basicSourcePtr -> TraceConnectWithoutContext("RemainingEnergy",
    MakeCallback(&RemainingEnergy));

for(uint32_t i=0; i<nWifi; i++)
{ Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource>
    (sources.Get(i));
  basicSourcePtr -> TraceConnectWithoutContext("RemainingEnergy",
    MakeCallback(&TotalEnergy));
}
```

The first pointer sets the callback to a RemainingEnergy function in the source number 1, corresponding to the node n1, or the client in the udp echo-client service installed.

And the second part sets a callback to a TotalEnergy function in all the sources, so this function will compute the total energy consumed in the network by all its devices.

The functions mentioned are set before the main:

```
void RemainingEnergy (double oldValue, double newValue)
{ std::cout << "At time " << Simulator::Now().GetSeconds() << "s Current
        remaining energy = " << newValue << "J" << std::endl;
}

void TotalEnergy (double oldValue, double newValue)
{ total_network_energy += (oldValue – newValue);
}
```

In the remaining energy function, we just print the new energy remaining value, so we will see on the output, every moment that the device consumed energy, and which amount of energy actually has consumed. But the total energy just cumulates the energy consumed by all the devices and we just need to print this double value at the end of the simulation to see the result.

4. Results

4.1 – First scenario: wifi communication results

After seeing the functionality of NS3 and once the simulation has run, the next step is to analyse the first results obtained. We configured a simple wifi network topology with three nodes acting as UDP server, UDP client, and an AP. We have activated the UDP logs, to trace what is actually happening during this simulation and the result is the following:

```
Node 0 is at (0, 0)
Node 1 is at (5, 0)
AP is at (10, 0)
At time 1s client sent 1000 bytes to 10.1.1.1 port 9
At time 1.0099s server received 1000 bytes from 10.1.1.2 port 49153
At time 1.0099s server sent 1000 bytes to 10.1.1.2 port 49153
At time 1.0168s client received 1000 bytes from 10.1.1.1 port 9
```

We can see that our configuration has been successful here. The three nodes are located in a row and spaced five meters between them, and the client (address 10.1.1.2 in the simulation) has sent the packet at 1 second to the server (address 10.1.1.1). As we installed the UDP echo application, this packet has been returned to the client and the simulation is completed.

Now that we know everything is working, we can add the energy module and see the result:

```
Node 0 is at (0, 0)
Node 1 is at (5, 0)
AP is at (10, 0)
At time 0.000362016s Current remaining energy = 4.99971 J
At time 0.000600048s Current remaining energy = 4.99958 J
At time 1.00611s Current remaining energy = 4.99924 J
At time 1.00679s Current remaining energy = 4.99911 J
At time 1.00827s Current remaining energy = 4.99478 J
At time 1.01336s Current remaining energy = 4.99444 J
At time 1.01686s Current remaining energy = 4.99431 J
Total energy consumed in network = 0.011376 J
```

We configured the energy model to consume battery every time some data is being transmitted, so this energy traces must correspond to packets sent by the node 0, or the server, where we connected the callback. There aren't only the single UDP packets sent in the network, and this is the reason why there are more consumption tracings.

We can actually see which are all those packets in figure 6, because we activated the Pcap tracing.

Filter: wlan.addr==00:00:00:00:00:01 Expressi

No.	Time	Source	Destination	Protocol
6	0.000528	00:00:00_00:00:01	00:00:00_00:00:03	802.11
7	0.000684		00:00:00_00:00:01	(802.11
8	0.000883	00:00:00_00:00:03	00:00:00_00:00:01	802.11
22	1.006219	00:00:00_00:00:01	00:00:00_00:00:02	ARP
23	1.006391		00:00:00_00:00:01	(802.11
24	1.006600	00:00:00_00:00:01	00:00:00_00:00:02	ARP
26	1.008138	10.1.1.2	10.1.1.1	UDP
28	1.009766	10.1.1.2	10.1.1.1	UDP
30	1.012766	00:00:00_00:00:01	Broadcast	ARP
31	1.012938		00:00:00_00:00:01	(802.11
32	1.013084	00:00:00_00:00:01	Broadcast	ARP
33	1.013230	00:00:00_00:00:02	00:00:00_00:00:01	ARP
35	1.013553	00:00:00_00:00:02	00:00:00_00:00:01	ARP
37	1.013647	10.1.1.1	10.1.1.2	UDP
38	1.015151		00:00:00_00:00:01	(802.11
39	1.016665	10.1.1.1	10.1.1.2	UDP

Fig.6 Pcap tracing on Wifi network

This is all the traffic in the network related to the server MAC address. There are four types of packets being transmitted during the entire simulation. The first one sent at time 0.000528s is an Association Request sent to the AP. The times shown here doesn't correspond exactly with the traced simulation consumption, but we can see the relation between a packet shown in this pcap and its energy tracing. The packets without an apparent source are just acknowledgements to received packets. The important packets are the coloured ones. The entire simulation can also be seen here. The packets sent between the client and the server appear twice, because in an infrastructure WLAN they have to be sent to the AP and then forwarded to the actual destination. Taking this into account, the picture shows how the client (10.1.1.2) sends one UDP packet to the server (10.1.1.1) at time 1.008138s after resolving the ARP (broadcasted packets from other nodes doesn't pass the filter for a specific MAC address so the ARP request doesn't appear here). Then the server, as soon as it resolves an ARP request, sends back the packet to the origin of the communication.

4.2 – Second scenario: Ad-hoc node saturation

We know now how the program works and it is the time to start with the main work of the project studying now the behaviour of the energy consumption in an ad-hoc WLAN within different scenarios. The first one is designed to increase the traffic destined to a single node which will be acting as a server, and all the other nodes will be UDP clients. The topology will be as shown on figure 7.

Fig. 7 Second scenario

We will use now, of course, some ad-hoc nodes instead of the devices used previously:

```
NodeContainer wifiadhocnodes;
wifiadhocnodes.Create (nAdhoc);

WifiHelper wifi; wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

// Set it to adhoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac,
                                           wifiadhocnodes);
```

The position of the nodes in this case will be also different, according to this:

```
if(scenario == 1)
{  MobilityHelper mobility;
   Ptr<ListPositionAllocator> positionAlloc =
CreateObject<ListPositionAllocator> ();
   while(1)
   { positionAlloc->Add (Vector (0.0, 0.0, 0.0));
     if(nAdhoc==2) break;
     positionAlloc->Add (Vector (0.0, 200.0, 0.0));
     if(nAdhoc==3) break;
     positionAlloc->Add (Vector (0.0, -200.0, 0.0));
     if(nAdhoc==4) break;
     positionAlloc->Add (Vector (0.0, 400.0, 0.0));
     if(nAdhoc==5) break;
     positionAlloc->Add (Vector (0.0, -400.0, 0.0));
     if(nAdhoc==6) break;
     positionAlloc->Add (Vector (0.0, 800.0, 0.0));
     if(nAdhoc==7) break;
     positionAlloc->Add (Vector (0.0, -800.0, 0.0));
     break;
   }
   positionAlloc->Add (Vector (200.0, 0.0, 0.0));
   mobility.SetPositionAllocator (positionAlloc);
   mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
   mobility.Install (wifiadhocnodes);
}
```

We will do the simulations with two to seven ad-hoc nodes starting from two, placed on (0,0,0) and (200,0,0), and adding 2 more nodes to the ‘y’ axis as seen in figure 7.

The energy model is still the same, but now installed to the new nodes. Notice that the server node will now be the last of the nodes in the network, as it is the last to be assigned, so when we install the UDP application must check it correctly. But before, we talked about OLSR protocol on ad-hoc communications, so we will configure it for our simulations:

```
// Enable OLSR
OlsrHelper olsr;
Ipv4StaticRoutingHelper staticRouting;

Ipv4ListRoutingHelper list;
list.Add (staticRouting, 0);
list.Add (olsr, 10);

InternetStackHelper internet;
internet.SetRoutingHelper (list);
internet.Install (wifiadhocnodes);
```

The OLSR protocol will allow nodes to set tables of neighbours and next-hop destinations. But we should assure it converges before sending traffic. So the actual simulation will start at 30 seconds. We will set a total time to 500 seconds and see the result.

```
double stoptime = 500; // seconds

UdpServerHelper server (9);
ApplicationContainer serverApps = server.Install (wifiadhocnodes.Get
                                                  (nAdhoc-1));
serverApps.Start (Seconds (30.0));
serverApps.Stop (Seconds (stoptime));

UdpClientHelper client (i.GetAddress (nAdhoc-1), 9);
client.SetAttribute ("MaxPackets", UintegerValue (numPackets));
client.SetAttribute ("Interval", TimeValue (Seconds (interval)));
client.SetAttribute ("PacketSize", UintegerValue (packetSize));

for (uint32_t i = 0; i<nSrc; i++)
{ ApplicationContainer clientApps = client.Install (wifiadhocnodes.Get
                                                    (i));
   clientApps.Start (Seconds (30.0));
   clientApps.Stop (Seconds (stoptime));
}
```

After all the configuration, we connected some traces to check and compare the total packets generated, transmitted, received by the server, and also the energy consumption in the network. We are using the UDPserver-client application now, without echo, in order to keep the server free from responding every single UDP packet received. We have configured the simulation to keep every node sending up to 10000 packets between intervals of 0.75 to 0.01 seconds.

To do all the simulations in the ad-hoc scenarios, we implemented a modification to the NS3 source code, to include an exponentially distributed packet inter-arrival time, so the values provided in the experiments are just the mean value of that time in each test. The explanation and demonstration of this point is included in annex [4].

The first stage of this scenario is the simplest possible. Just two nodes acting as client and server and sending this 10000 packets.

Fig. 8 Second scenario with 2 nodes

The result is not surprising in this case, shown on figure 9 and 10.

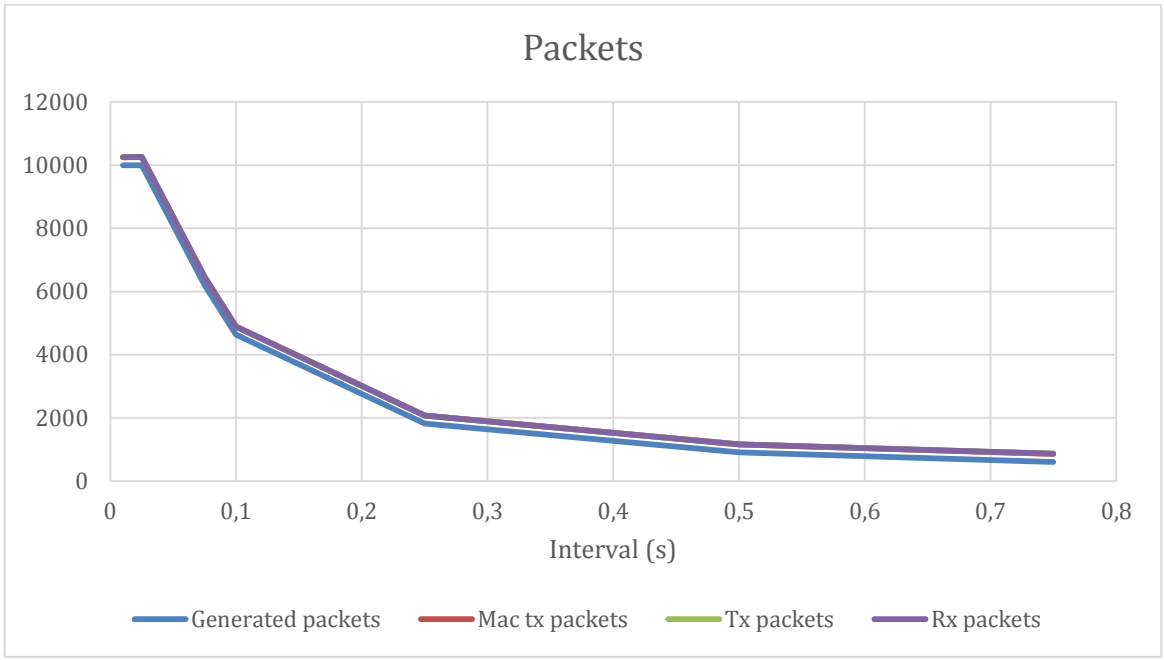


Fig.9 Packets analysed with 2 nodes

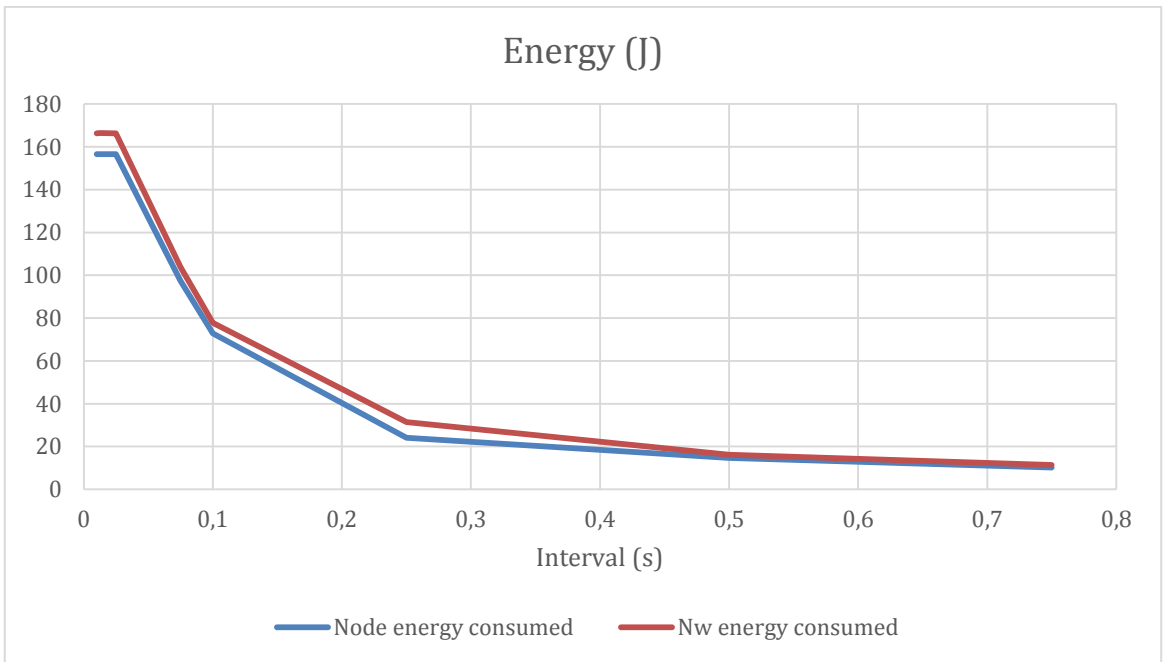


Fig.10 Energy results with 2 nodes

The first chart shows the packets generated, transmitted to physical layer, MAC transmitted, and received by the server node. The difference between MAC transmitted and transmitted is that when a packet is received at physical layer, it sends the MacTx trace, and when its transmission begins, we will capture the PhyTxBegin trace. When a packet is transmitted but collides and needs retransmission, the MacTx trace will appear once, but the PhyTxBegin will be sent as many times as the packet is being transmitted and retransmitted, so we can see if there are too many collisions in the channel.

The number of packets either generated or transmitted increase exponentially the lower the interval is, and when the interval is lower than approximately 0.03 seconds, the client is capable of generating the 10000 packets in the 500 seconds of the simulation duration. The energy consumed by the client or the sender is fully proportional to the packets transmitted. It must be kept in mind that we are considering consumption only when transmitting packets. In this case, it is expected that the total network energy consumed will be almost the same as the single node energy consumed (the energy consumed by the client), because the server has to send only Acknowledgements.

If we increase the number of client nodes, according to figure 11, we can see that with a low interval between packets, the network will start having trouble managing the flooding, but the energy remains following the same curve as the transmitted packets, seen on figures 12 and 13.

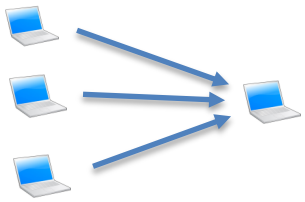


Fig.11 Second scenario with 4 nodes

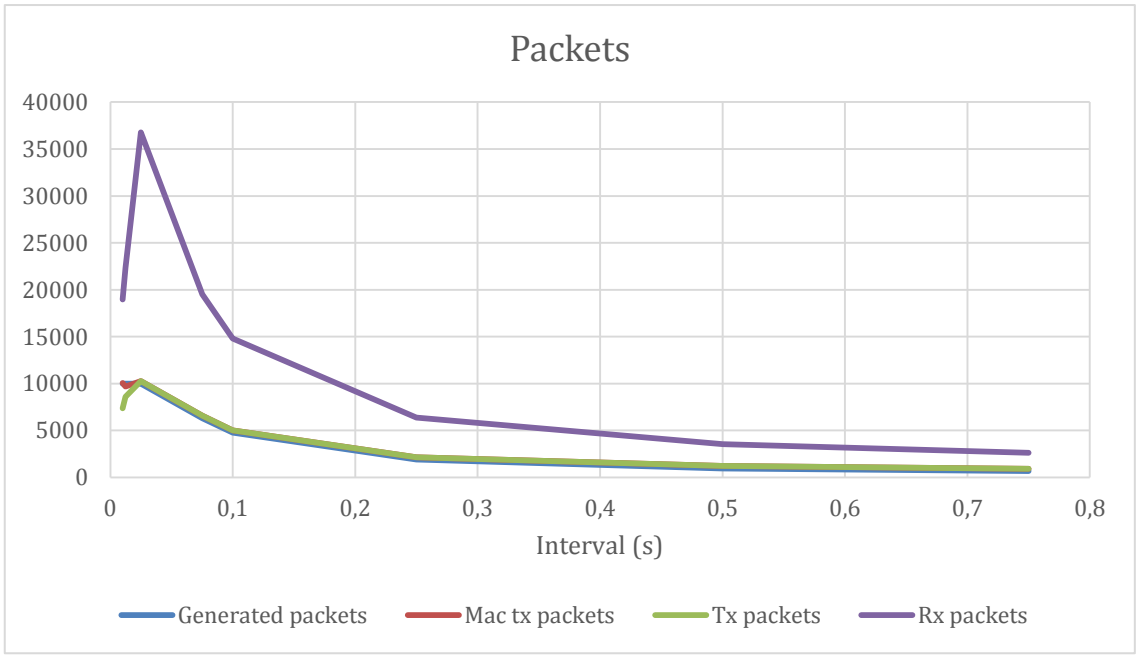


Fig.12 Packets analysed with 4 nodes

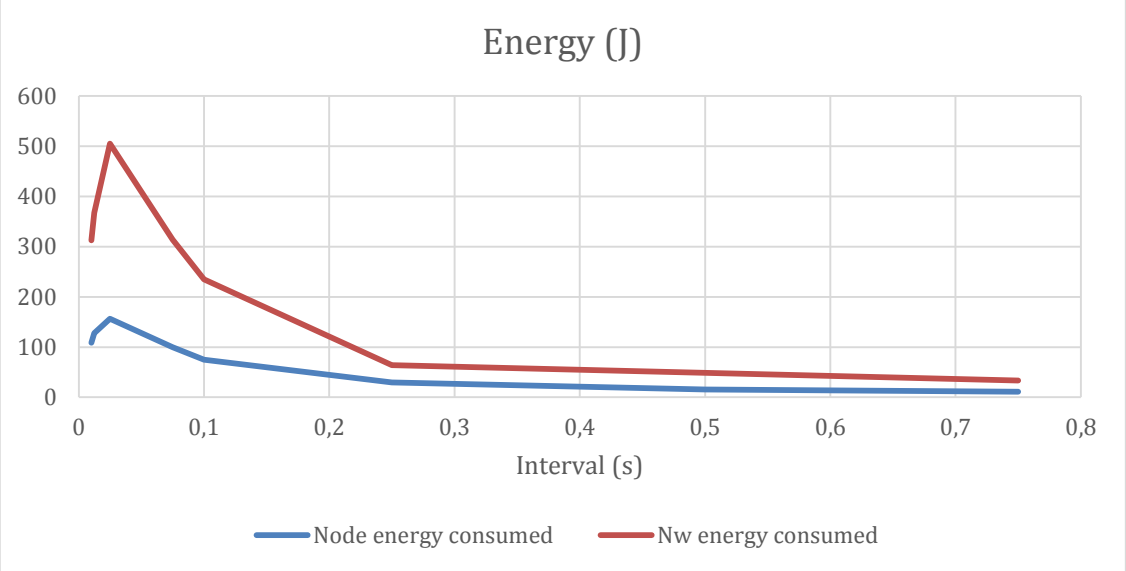


Fig.13 Energy results with 4 nodes

We can see here what was commented before about the MacTx and Transmitted packets. However, what it happening is reversed as expected. The MacTx packets are higher than the transmitted packets. At the physical layer, actually arrive the 10000 packets generated, but at low intervals, the transmitted packets drop to almost only 7000. The reason this is happening is because in this wireless networks, it is used a CSMA/CA mechanism, not just collision detection but collision avoidance. This means that all the packets are being generated, but before transmitting and colliding, the nodes wait until the transmission is possible.

The energy consumed by the first node is again proportional to the packets transmitted, obvious as we are considering consumption when a packet is transmitted, and the total energy consumed is similar to the packets received by the server. That is because the server receives the packets transmitted by all other nodes, so it is consistent that the two measures are related.

If we increase now the clients up to 6 nodes, the flooding is of course even worse as the graphic on figure 14 exposes.

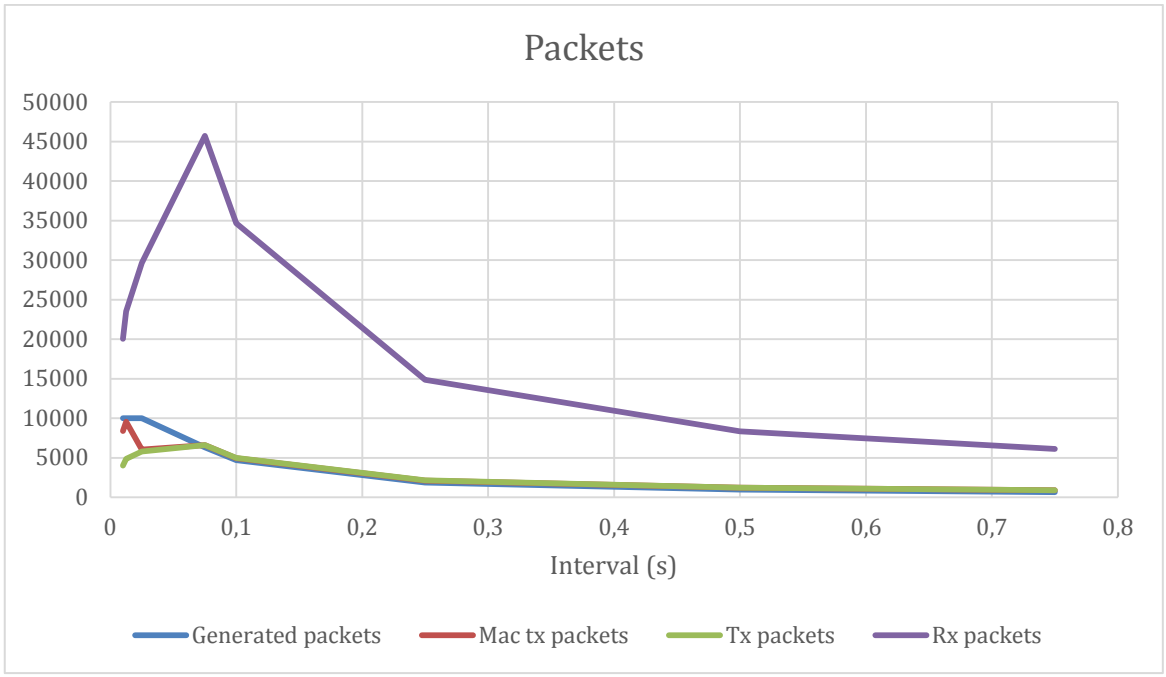


Fig.14 Packets analysed with 7 nodes

The transmitted packets here drop at a higher interval because there are a lot more nodes sending traffic and the saturation of the network appear earlier. The energy in this case behaves exactly like in the previous simulations, drawing the same curve as the transmitted packets and the received by the sender. The collision avoidance prevents the nodes in an ad-hoc network to increase exponentially the packets transmitted, and in direct consequence, the energy consumed by them. This functionality can prevent a node to consume all its battery and disconnect from the network, but in cases of very high traffic, a node may seem to have lost connection because of the high number of collisions suffered. It may not be able to send any packets if the network is constantly colliding.

4.3 – Third scenario: Constant traffic in increasing network

On this experiment we are trying to see what happens when the same amount of traffic goes through different sizes of networks. So we started simulating a small 4-node network sending all traffic to the last node, and increasing the number of nodes exponentially to 25. The topology in this case is disposed in a square, placing the nodes every 500 meters as figure 15 shows. The interval time between packets on each node will also increase from 0.3 seconds in each node (a total of 10 packets / second on the network) to 2.4 seconds (see figure 16), taking into account that the last node is only acting as a server:

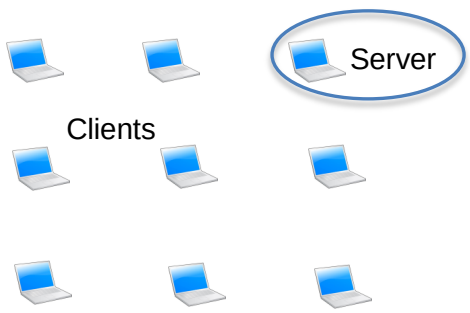


Fig.15: 9-node topology sending traffic to last one

Simulation	Adhoc nodes	Interval	Packets/s
1	4	0,3	3
2	9	0,8	1,25
3	16	1,5	0,6667
4	25	2,4	0,41667

Fig.16 Traffic distribution on third scenario

The entire simulation lasts again 500 seconds, and the results for the generated and received packets are shown in figure 17.

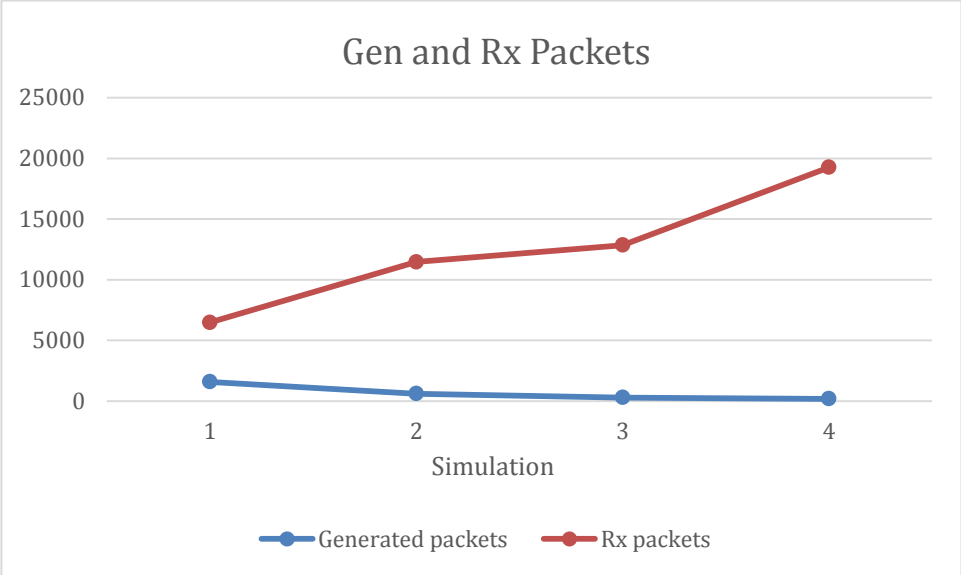


Fig. 17 Traffic distribution on third scenario

The generated packets obviously is decreasing over the simulations, because the interval between packets are configured this way, and we are only considering a single node in this parameter. The total received packets, even we are having always a total of 10 packets/second in the network, is increasing on each simulation. This is because all nodes are sending routing OLSR packets to keep contact and if the number of nodes increases, even maintaining the total traffic, the packets received also increase.

To make a deeper study in this scenario we implemented a python script that parses an output trace configured on the NS3 code:

```
AsciiTraceHelper ascii;
wifiPhy.EnableAsciiAll (ascii.CreateFileStream ("wifi-adhoc.tr"));
wifiPhy.EnablePcap ("wifi-adhoc", devices);
// Trace routing tables
Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper>
    ("wifi-simple-adhoc.routes", std::ios::out);
olsr.PrintRoutingTableAllEvery (Seconds (2), routingStream);
Ptr<OutputStreamWrapper> neighborStream = Create<OutputStreamWrapper>
    ("wifi-adhoc.neighbors", std::ios::out);
olsr.PrintNeighborCacheAllEvery (Seconds (2), neighborStream);
```

The python script can be found on annex 3.

The output file wifi-adhoc.tr will contain the information of every single packet sent through the network, which we will use in this case to check the retries done by each node. This will let us know the 'stability' of this exponential node increase. The wifi-adhoc.neighbors will show the routing tables of the nodes every time they are updated. Here we checked that actually not all nodes are sending traffic directly, but are forwarding packets through the network. In fact, the maximum distance in which the packets are sent directly is approximately 600 meters, and the distance between nodes we configured is set intentionally to 500 meters, so every node has only a maximum of 4 neighbours.

The number of retries is shown in figure 18.

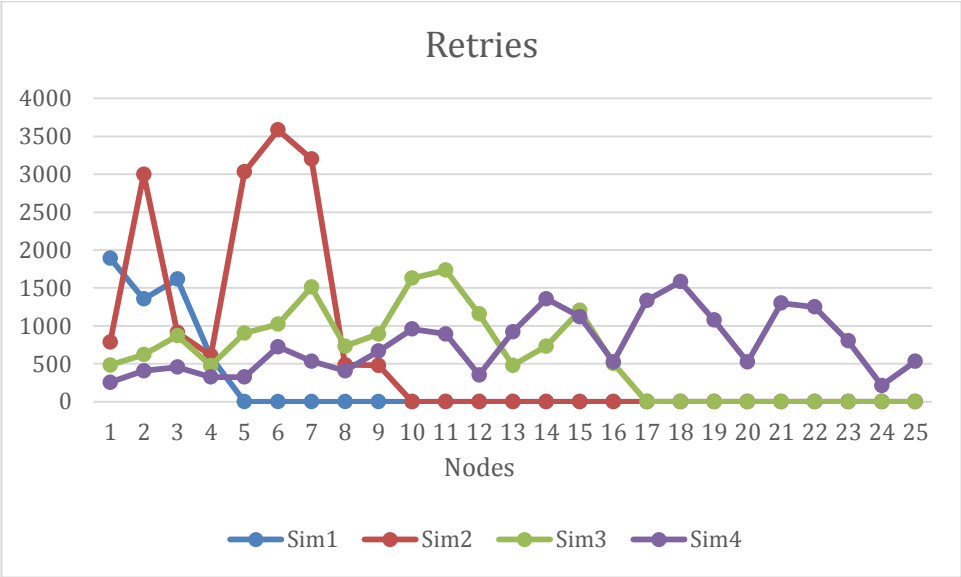


Fig. 18 Retries on third scenario

We can see that there is a greater number of retries in the first simulations than on the other two. We could expect that if we increase the number of nodes, and the total traffic of the network is also increasing, as seen on the last graphic, the number of retries will also increase, but that is not what happens here. Especially on the simulation 2, with 9 nodes on the network, the number of retries is very high. There are some nodes that are only sending traffic, and don't have to forward packets because they are not on the 'shortest path' of any other node. That is the reason that some nodes have less retries than the average. And vice versa, there are nodes that have to forward more packets and collide more times in the network, so they have more retries than the others. On the first simulations we are concentrating the forwarding tasks to less nodes that send more UDP

packets than on the last two experiments. The OLSR packets are far smaller than the UDP ones, configured to 1kbyte. This is the reason why on the second simulation we have greater retries than on the others, which distribute the traffic to more nodes.

4.4 – Last scenario: Mobility implication

In the last experiment, we are setting the 5x5 topology in which we ended the last scenario and will check how the mobility of the nodes affects the traffic and the energy consumption. We will set again the last node as server and the others as clients and will run the simulation on both cases with static position and random mobility. We have reduced the distance between nodes to 250 meters to avoid network splitting: The maximum distance between reachable nodes is near 600m, so if we set the inter-distance to 500m there is a possibility that a node from the perimeter walks outside the reachable area of its neighbours.

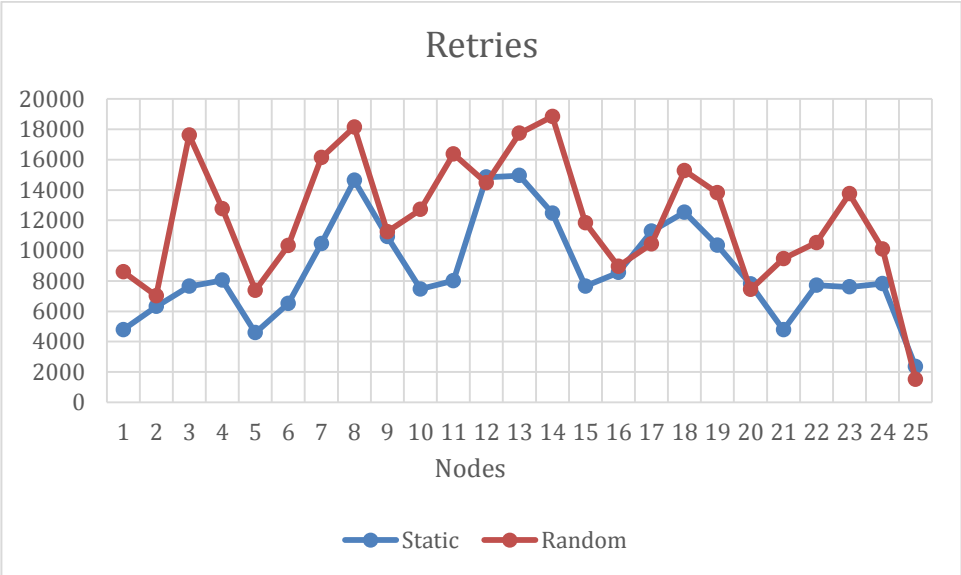


Fig. 19 Mobility vs fixed position retries

The retries on each node are almost always greater in the mobility simulation, and follows the same form as the static plot. But on the random experiment appear more retries peaks than on the first one. There are more nodes affected by the forwarding tasks in this case, the nodes 3, 8, 11, 14, 17 and 23, all of this taking in mind that almost all nodes are having more collisions.

We checked the position before and after the simulation on figure 20, to see if a higher mobility has something to do with the greater retries.

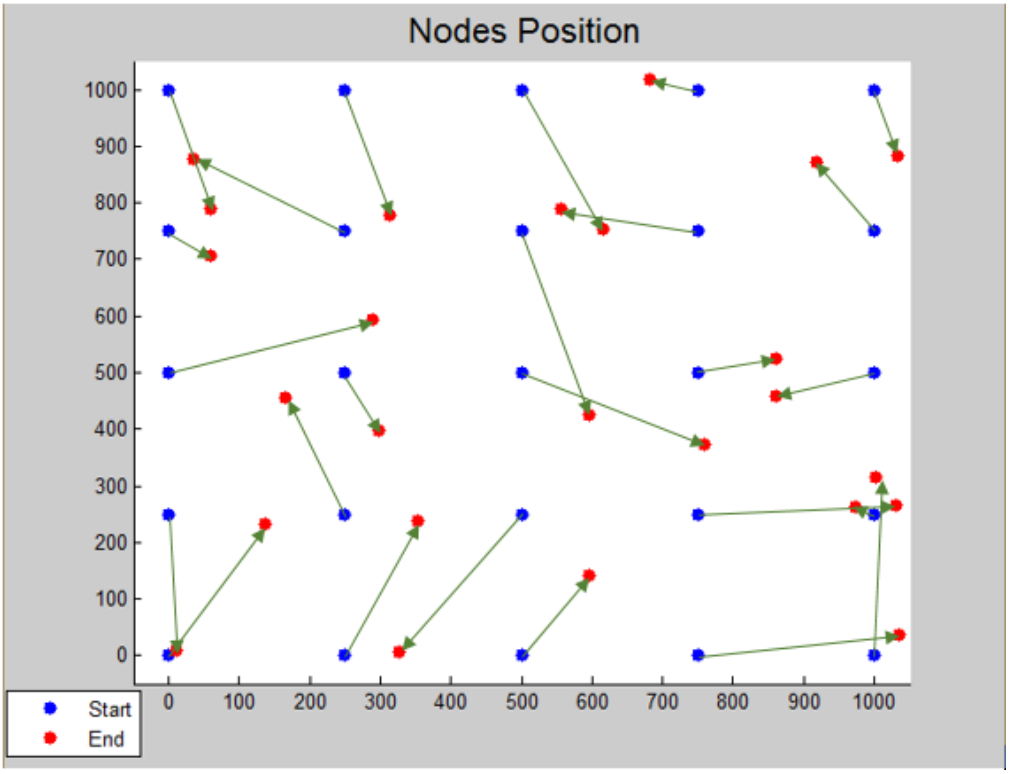


Fig. 20 Mobility diagram

The figure shows only the relative distance travelled between the beginning and the end of the simulation. Obviously they didn't go through a straight line because the mobility is set to random and every update of the position can go to any direction. By calculating the module of the vectors printed on the image, we can see the relative distance travelled by each node (see figure 21).

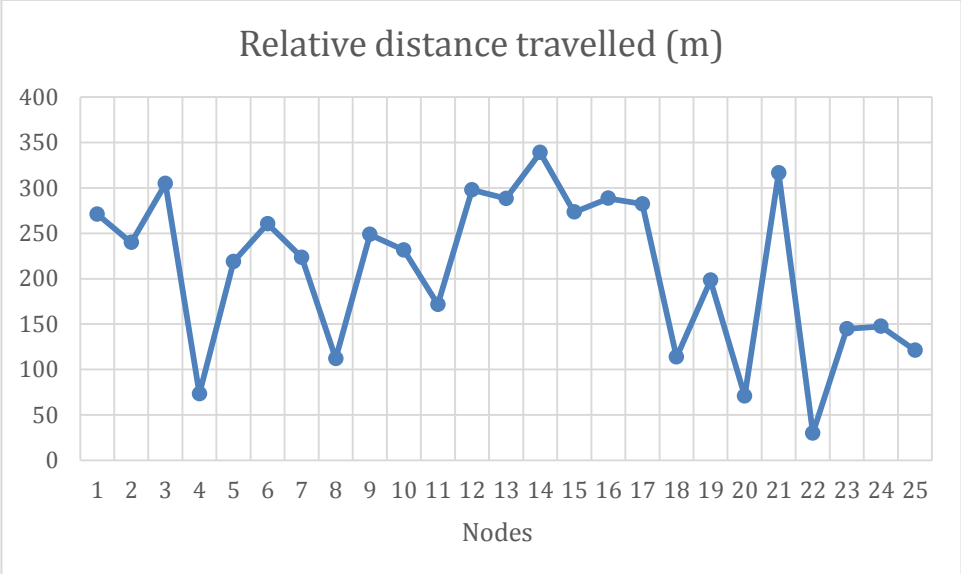


Fig. 21 Relative distance travelled on mobility scenario

The nodes that travelled the most are 2, 6, 12 – 17 and 21. If we compare this information with the number of retries in the mobility situation, we can conclude that the relative mobility doesn't affect directly to the collisions found during the simulation because in this case, the node 8 is one of the nodes that has travelled less, but suffered one of the greater number of retries, and in the inverse situation, node 21 has travelled a lot, but has a low number of retries.

5. Budget

To calculate the total budget for developing this project, at first we must consider the salary of a Wireless Engineer. The current mean salary for this type of employees is about 23000€ a year. If the duration of this project will be of 6 months, the amount of money destined to the first salary will be:

```
Wireless Engineer (6 months): 11500 €
```
[9]

We may also need a software engineer to program some application on the mesh devices that allows us to send the simulated traffic with a specified packet length, inter-arrival time, and all the parameters used on this project, and then we should be able to analyze the traffic received by all the nodes. This engineer would work for approximately 3 months while on the other 3, we would be doing the tests.

```
Software Engineer (3 months): 5750 €
```
[9]

The next thing we should need is to develop this simulations on real devices and do a full study if some investors are interested in our work.

```
Mesh/Ad-hoc Devices (25 units): 1680 € (~70 € each)
PC: ~500€
Ethernet cables, USB cables, etc. (total): ~30 €
Desks and office material (total): ~600€
```
[10]
[11]

Another requirement should be an office in Barcelona to analyze the results and write the reports. We would rent a place to work, not an entire office, because we just need a desk and the computer, so we would need a total of 9 rent months, 6 for the first engineer and 3 for the software one. The two engineers will work at the same time but just looking at the numbers, we will need a 9 month renting, as the first 3 count double.

```
Office renting 12 m² (9 months): 981€ €
```
[12]

At last, to do the tests with the devices, we will need 25 people using the developed app and walking through the streets of Barcelona with the mesh or ad-hoc devices. This will require an approximate 3 day compensation to 25 people responding to an offer. This will be calculated as the minimum salary in Spain per month, divided by 30 days a month and multiplied by days of work.

```
Volunteer Salary (25 person 3 days): ~2000€
```
[13]

```
Total approximate budget: 23041 €
```

This is an affordable budget for a medium or big technological company that wants to invest on this type of research for future implementations.

6. Conclusions and future development:

This project consisted on study the energy consumption of mesh/ad-hoc communication devices in different scenarios. The first conclusion that we hit is that actually NS3 is not developed to introduce an energy model to any kind of network, just on Wifi ones. This will make an obstacle into a mesh energy research, as it has been to this project. NS3 is prepared to adapt own code to the source because it is a collaborative open source program, so if in the future this research is wanted to be applied on mesh networks, an energy model for mesh devices must be developed.

Now looking through this work on the ad-hoc networks, we can conclude that the energy consumed is just a direct consequence of the number of packets sent and received. If we can control this parameter, we can also assume that the energy consumed will be controlled. As it has been seen, ad-hoc networks work through a CSMA/CA mechanism that does exactly this work, and assure us that the energy will not be totally consumed even if we are in a highly flooded network.

In another scenario, we have seen that the number of the nodes in the network affects the traffic, even if we are maintaining the same application packet rate. The routing packets sent in this case grow with the number of total nodes in the network, so a bigger network, implies a greater flooding. But we have also concluded that the number of collisions or retries in each node, is worse in a small network. This happens because if a mesh or ad-hoc network increases, the possible routes between a sender and a receiver also increases.

In the last scenario, we added mobility to the nodes, and observed that there are more collisions than if the nodes stay fixed. After studying the movement of the nodes in the simulation, there seems to be no relation between the number of retries suffered by a node, and the distance travelled. The mobility implies worse conditions for communication, and this has been reflected in the results.

To extend this work, some real experiments should be carried out on real devices to give full conclusions. We included the total approximate budget to be checked if this kind of research is of the interest of the technological companies. We think that any middle or big company can afford this research if it is on their technologic interests.

After this project, it can be easier for someone who is beginning to work with NS3 and energy simulation to learn this functionality reading this work rather than searching over the Internet as we have been doing. So in the future, if someone has to take forward the project, he should learn the NS3 functionality through this Thesis, and then follow the two possible steps that have already been explained: Testing the simulations on real devices, and/or extend the working to the mesh networks.

Bibliography:

[1] https://en.wikipedia.org/wiki/Wireless_LAN

[2] https://en.wikipedia.org/wiki/Wireless_ad_hoc_network

[3] http://www.howtogeek.com/180649/htg-explains-whats-the-difference-between-ad-hoc-and-infrastructure-mode/

[4] T. Clausen, P. Jacquet, “RFC 3626: Optimized Link State Routing Protocol (OLSR)”, October 2003.

[5] I. F. Akyildiz, X. Wang, and W. Wang, “Wireless mesh networks: a survey”, Computer Networks, vol. 47, no.4, pp. 445 – 487, 2005

[6] Andrés Marcelo Vázquez Rodas, “Contribution to the Improvement of the Performance of Wireless Mesh Networks Providing Real Time Services”, Ph.D. Dissertation, Network Engineering Department, UPC, February 2015.

[7] Andrés Vázquez Rodas, Luis J. de la Cruz Llopis, “A centrality-based topology control protocol for wireless mesh networks”, Ad Hoc Networks 24, pp. 34-54, Elsevier, 2015.

[8] www.nsnam.org

[9] http://www.payscale.com/

[10] http://www.cetronic.es/sqlcommerce/disenos/plantilla1/seccion/Catalogo.jsp?idIdioma=&idTienda=93&cPath=1358&gclid=CLncirSOl8gCFUnlwgodFG8BRw

[11] http://www.ofiraso.es/es/empresas/oficina-general/mesas.html?gclid=CK_oz5784ccCFWv3wgodtOcCiQ

[12] http://www.idealista.com/inmueble/29106130/

[13] http://www.datosmacro.com/smi/espana

Appendices

Annex 1: Wifi script

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/internet-module.h"
#include "ns3/energy-module.h"
#include "ns3/config-store-module.h"
#define TOTAL_ENERGY
#define NODE_ENERGY

// Default Network Topology
//
//   Wifi 10.1.1.0
//                AP
//     *    *    *
//     |    |    |
//    n2   n1   n0
//

#ifdef TOTAL_ENERGY
  double total_energy = 0.0;
#endif

using namespace ns3;

/// Trace function for remaining energy at node.

#ifdef NODE_ENERGY
void
RemainingEnergy (double oldValue, double newValue)
{
  std::cout << "At time " << Simulator::Now ().GetSeconds ()
                << "s Current remaining energy = " << newValue << " J" << std::endl;
}
#endif

#ifdef TOTAL_ENERGY
void
TotalEnergy (double oldValue, double newValue)
{
  total_energy += (oldValue - newValue);
}
#endif

int
main (int argc, char *argv[])
{
  bool verbose = true;
  uint32_t nWifi = 2;

  CommandLine cmd;
  cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
  cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

  if (verbose)
    {
      LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
      LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
    }
```

```
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi);
NodeContainer wifiApNode;
wifiApNode.Create(1);

YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
phy.SetChannel (channel.Create ());

WifiHelper wifi = WifiHelper::Default ();
wifi.SetRemoteStationManager ("ns3::AarfWifiManager");

NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();

Ssid ssid = Ssid ("ns-3-ssid");
mac.SetType ("ns3::StaWifiMac",
             "Ssid", SsidValue (ssid),
             "ActiveProbing", BooleanValue (false));

NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes);

mac.SetType ("ns3::ApWifiMac",
             "Ssid", SsidValue (ssid));

NetDeviceContainer apDevices;
apDevices = wifi.Install (phy, mac, wifiApNode);

MobilityHelper mobility;

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

mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
                           "Bounds", RectangleValue (Rectangle (-50, 50, -50, 50)));

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

/** Energy Model **/
/***************************************************************************/
/* energy source */
BasicEnergySourceHelper basicSourceHelper;
// configure energy source
basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue (5.0));
// install source
EnergySourceContainer sources = basicSourceHelper.Install (wifiStaNodes);
/* device energy model */
WifiRadioEnergyModelHelper radioEnergyHelper;
// configure radio energy model
radioEnergyHelper.Set ("IdleCurrentA", DoubleValue (1.0));
radioEnergyHelper.Set ("CcaBusyCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("TxCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("RxCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("SwitchingCurrentA", DoubleValue (0));
radioEnergyHelper.Set ("SleepCurrentA", DoubleValue (0));
DeviceEnergyModelContainer deviceModels = radioEnergyHelper.Install (staDevices, sources);
/***************************************************************************/
```

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

  Ipv4AddressHelper address;
  address.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer wifiInterfaces = address.Assign (staDevices);
  address.Assign (apDevices);

  UdpEchoServerHelper echoServer (9);
  ApplicationContainer serverApps = echoServer.Install (wifiStaNodes.Get (0));
  serverApps.Start (Seconds (0.0));
  serverApps.Stop (Seconds (3.0));



  UdpEchoClientHelper echoClient (wifiInterfaces.GetAddress (0), 9);
  echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
  echoClient.SetAttribute ("PacketSize", UintegerValue (976));

  ApplicationContainer clientApps = echoClient.Install (wifiStaNodes.Get (1));
  clientApps.Start (Seconds (1.0));
  clientApps.Stop (Seconds (3.0));

  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  Simulator::Stop (Seconds (3.0));

/*phy.EnablePcap ("wifi", apDevices.Get (0));
phy.EnablePcap ("wifi", staDevices.Get(0));
phy.EnablePcap ("wifi", staDevices.Get(1));*/

#ifdef NODE_ENERGY
  Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource> (sources.Get (1));
  basicSourcePtr->TraceConnectWithoutContext ("RemainingEnergy", MakeCallback (&RemainingEnergy));
#endif

#ifdef TOTAL_ENERGY
uint32_t i;

  for( i=0 ; i<nWifi ; i++)
  {

    Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource> (sources.Get (i));
    basicSourcePtr->TraceConnectWithoutContext ("RemainingEnergy", MakeCallback(&TotalEnergy));
  }
#endif

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

#ifdef TOTAL_ENERGY
  std::cout <<  "Total energy consumed in network = " << total_energy << " J" << std::endl;
#endif

  return 0;
}
```

Annex 2: Wifi ad-hoc script

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/energy-module.h"
#include "ns3/applications-module.h"
#include "ns3/olsr-helper.h"
#include "ns3/ipv4-static-routing-helper.h"
#include "ns3/ipv4-list-routing-helper.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

double total_network_energy = 0.0;
double total_node_energy = 0.0;
using namespace ns3;
uint32_t packets_generated = 0;
uint32_t packets_mac_tx = 0;
uint32_t packets_transmitted = 0;
uint32_t packets_received = 0;

NS_LOG_COMPONENT_DEFINE ("WifiSimpleAdhoc");

void PacketGenerated(std::string context,double intTime,uint32_t packetLength) {
  //std::cout << Simulator::Now ().GetSeconds () << "  " << "Packet Generated  " << intTime << "  "
<< packetLength << std::endl;
  packets_generated++;
//std::cout << intTime << "     " << packetLength << std::endl;
}

void PacketTransmitted(std::string context,const Ptr<const Packet> p) {
  //std::cout << Simulator::Now ().GetSeconds () << "\t" << "Packet Transmitted" << std::endl;

   packets_transmitted++;
}

void PacketReceived(std::string context,const Ptr<const Packet> p) {
  //std::cout << Simulator::Now ().GetSeconds () << "\t" << "Packet Received" << std::endl;
  packets_received++;
}

void PacketDroppedTx(std::string context,const Ptr<const Packet> p) {
  //std::cout << Simulator::Now ().GetSeconds () << "\t" << "Packet Dropped MacTx" << std::endl;
}

void PacketDroppedRx(std::string context,const Ptr<const Packet> p) {
  //std::cout << Simulator::Now ().GetSeconds () << "\t" << "Packet Dropped MacRx" << std::endl;
}

void PacketMacTx(std::string context,const Ptr<const Packet> p) {
  //std::cout << Simulator::Now ().GetSeconds () << "\t" << "Packet MacTx" << std::endl;
  packets_mac_tx++;
}

void PacketTxDrop(std::string context, const Ptr<const Packet> p){
  //std::cout << Simulator::Now().GetSeconds () << "\t" << "Packet Tx Drop" << std::endl;
}
```

```
void
RemainingEnergy (double oldValue, double newValue)
{
  total_node_energy += (oldValue - newValue);
  //std::cout << "At time " << Simulator::Now ().GetSeconds () << "s Current remaining energy = "
<< newValue << " J" << std::endl;
  std::cout << "At time " << Simulator::Now ().GetSeconds () << "s Total energy consumed by sender
= " << total_node_energy << "J" << std::endl;
}

void
TotalEnergy (double oldValue, double newValue)
{
  total_network_energy += (oldValue - newValue);
}

void
Position (std::string context, Ptr<const MobilityModel> model)
{
  Vector position = model->GetPosition ();
  NS_LOG_UNCOND (context << " x = " << position.x << ", y = " << position.y);
}

void
CourseChange (std::string context, Ptr<const MobilityModel> model)
{
  Vector position = model->GetPosition ();
  NS_LOG_UNCOND (context << " x = " << position.x << ", y = " << position.y);
}

int main (int argc, char *argv[])
{
  std::string phyMode ("DsssRate11Mbps");
  double distance = 250;  // m
  uint32_t packetSize = 1000; // bytes
  uint32_t numPackets = 10000;
  double interval = 1; // seconds
  uint32_t nAdhoc = 4; // number of nodes
  uint32_t nSrc = 1; // number of sources sending traffic
  uint32_t scenario = 2; // index of current scenario
  bool verbose = false;
  bool total_energy = true;
  bool node_energy  = false;
  bool static_position = true;
  bool random_mobility = false;
  bool enable_pcap = false;

  CommandLine cmd;

  cmd.AddValue ("scenario", "changes the scenario of simulation", scenario);
  cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
  cmd.AddValue ("packetSize", "size of application packet sent", packetSize);
  cmd.AddValue ("numPackets", "number of packets generated", numPackets);
  cmd.AddValue ("interval", "interval (seconds) between packets", interval);
  cmd.AddValue ("nAdhoc", "number of adhoc nodes in the network", nAdhoc);
  cmd.AddValue ("nSrc", "number of adhoc nodes sending traffic", nSrc);
  cmd.AddValue ("distance", "distance between nodes", distance);
  cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", verbose);
  cmd.AddValue ("total_energy", "enables the energy consumed by network tracing", total_energy);
  cmd.AddValue  ("node_energy"  ,  "enables  the  energy  consumed  by  one  single  node  tracing",
node_energy);
  cmd.AddValue ("static_position", "enables the tracing of static nodes", static_position);
  cmd.AddValue ("random_mobility", "enables the tracing of random mobility", random_mobility);
  cmd.AddValue ("enable_pcap", "enables pcap tracing", enable_pcap);

  cmd.Parse (argc, argv);
```

```
  // Convert to time object
  Time interPacketInterval = Seconds (interval);

  // disable fragmentation for frames below 2200 bytes
  Config::SetDefault      ("ns3::WifiRemoteStationManager::FragmentationThreshold",      StringValue
("2200"));
  // turn off RTS/CTS for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("2200"));
  // Fix non-unicast data rate to be the same as that of unicast
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
                      StringValue (phyMode));
  // Set the speed of nodes moving randomly through the simulation space
  Config::SetDefault ("ns3::RandomWalk2dMobilityModel::Speed",
                      StringValue ("ns3::ConstantRandomVariable[Constant=100.0]"));


  NodeContainer wifiadhocnodes;
  wifiadhocnodes.Create (nAdhoc);

  // The below set of helpers will help us to put together the wifi NICs we want
  WifiHelper wifi;
  if (verbose)
    {
      LogComponentEnable ("UdpClient", LOG_LEVEL_INFO);
      //LogComponentEnable ("UdpServer", LOG_LEVEL_INFO);
      //wifi.EnableLogComponents ();  // Turn on all Wifi logging
    }
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

  YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
  // This is one parameter that matters when using FixedRssLossModel
  // set it to zero; otherwise, gain will be added
  wifiPhy.Set ("RxGain", DoubleValue (0) );
  // ns-3 supports RadioTap and Prism tracing extensions for 802.11b
  wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel");;
  wifiPhy.SetChannel (wifiChannel.Create ());

  // Add a non-QoS upper mac, and disable rate control
  NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                                "DataMode",StringValue (phyMode),
                                "ControlMode",StringValue (phyMode));
  // Set it to adhoc mode
  wifiMac.SetType ("ns3::AdhocWifiMac");
  NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, wifiadhocnodes);

if(scenario == 1)
{  MobilityHelper mobility;
   Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
   while(1)
   { positionAlloc->Add (Vector (0.0, 0.0, 0.0));
     if(nAdhoc==2) break;
     positionAlloc->Add (Vector (0.0, 200.0, 0.0));
     if(nAdhoc==3) break;
     positionAlloc->Add (Vector (0.0, -200.0, 0.0));
     if(nAdhoc==4) break;
     positionAlloc->Add (Vector (0.0, 400.0, 0.0));
     if(nAdhoc==5) break;
     positionAlloc->Add (Vector (0.0, -400.0, 0.0));
     if(nAdhoc==6) break;
     positionAlloc->Add (Vector (0.0, 800.0, 0.0));
     if(nAdhoc==7) break;
```

```
      positionAlloc->Add (Vector (0.0, -800.0, 0.0));
      break;
    }
    positionAlloc->Add (Vector (200.0, 0.0, 0.0));
    mobility.SetPositionAllocator (positionAlloc);
    mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
    mobility.Install (wifiadhocnodes);
}


if(scenario == 2)
{  MobilityHelper mobility;
    mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
  "MinX", DoubleValue (0.0),
  "MinY", DoubleValue (0.0),
  "DeltaX", DoubleValue (distance),
  "DeltaY", DoubleValue (distance),
  "GridWidth", UintegerValue (5),
  "LayoutType", StringValue ("ColumnFirst"));
    mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
    mobility.Install (wifiadhocnodes);
}

if(scenario == 3)
{  MobilityHelper mobility;
    mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
  "MinX", DoubleValue (0.0),
  "MinY", DoubleValue (0.0),
  "DeltaX", DoubleValue (250),
  "DeltaY", DoubleValue (250),
  "GridWidth", UintegerValue (5),
  "LayoutType", StringValue ("ColumnFirst"));
  mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel", "Bounds", RectangleValue (Rectangle
(0, 1050, 0, 1050)));
  mobility.Install (wifiadhocnodes);
}

  /** Energy Model **/
  /***************************************************************************/
  /* energy source */
  BasicEnergySourceHelper basicSourceHelper;
  // configure energy source
  basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue (1000.0));
  // install source
  EnergySourceContainer sources = basicSourceHelper.Install (wifiadhocnodes);
  /* device energy model */
  WifiRadioEnergyModelHelper radioEnergyHelper;
  // configure radio energy model
  radioEnergyHelper.Set ("IdleCurrentA", DoubleValue (0));
  radioEnergyHelper.Set ("CcaBusyCurrentA", DoubleValue (0));
  radioEnergyHelper.Set ("TxCurrentA", DoubleValue (1.0));
  radioEnergyHelper.Set ("RxCurrentA", DoubleValue (0.0));
  radioEnergyHelper.Set ("SwitchingCurrentA", DoubleValue (0));
  radioEnergyHelper.Set ("SleepCurrentA", DoubleValue (0));
  DeviceEnergyModelContainer deviceModels = radioEnergyHelper.Install (devices, sources);
  /***************************************************************************/

  // Enable OLSR
  OlsrHelper olsr;
  Ipv4StaticRoutingHelper staticRouting;

  Ipv4ListRoutingHelper list;
  list.Add (staticRouting, 0);
  list.Add (olsr, 10);

  InternetStackHelper internet;
```

```
  internet.SetRoutingHelper (list); // has effect on the next Install ()
  internet.Install (wifiadhocnodes);

  Ipv4AddressHelper ipv4;
  NS_LOG_INFO ("Assign IP Addresses.");
  ipv4.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer i = ipv4.Assign (devices);

  double stoptime = 500; // seconds

  UdpServerHelper server (9);
  ApplicationContainer serverApps = server.Install (wifiadhocnodes.Get (nAdhoc-1));
  serverApps.Start (Seconds (30.0));
  serverApps.Stop (Seconds (stoptime));



  UdpClientHelper client (i.GetAddress (nAdhoc-1), 9);
  client.SetAttribute ("MaxPackets", UintegerValue (numPackets));
  client.SetAttribute ("Interval", TimeValue (Seconds (interval)));
  client.SetAttribute ("PacketSize", UintegerValue (packetSize));

  for (uint32_t i = 0; i<nSrc; i++)
  { ApplicationContainer clientApps = client.Install (wifiadhocnodes.Get (i));
    clientApps.Start (Seconds (30.0));
    clientApps.Stop (Seconds (stoptime));
  }


if(enable_pcap)
{      AsciiTraceHelper ascii;
       wifiPhy.EnableAsciiAll (ascii.CreateFileStream ("wifi-simple-adhoc.tr"));
       wifiPhy.EnablePcap ("wifi-simple-adhoc", devices);
       // Trace routing tables
       Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> ("wifi-adhoc.routes",
std::ios::out);
       olsr.PrintRoutingTableAllEvery (Seconds (2), routingStream);
       Ptr<OutputStreamWrapper>     neighborStream     =     Create<OutputStreamWrapper>     ("wifi-
adhoc.neighbors", std::ios::out);
       olsr.PrintNeighborCacheAllEvery (Seconds (2), neighborStream);
}

 Simulator::Stop (Seconds (stoptime));

if(node_energy)
{  Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource> (sources.Get (0));
   basicSourcePtr->TraceConnectWithoutContext ("RemainingEnergy", MakeCallback (&RemainingEnergy));
}

if(static_position || random_mobility)
{ for(uint32_t i=0; i<nAdhoc; i++)
  {
      Ptr<Node> node = wifiadhocnodes.Get(i);
      Ptr<MobilityModel> mob = node->GetObject<MobilityModel> ();
      Vector pos = mob->GetPosition ();
      std::cout << "Node " << i << " is at (" << pos.x << ", " << pos.y << ")\n";
  }
}

if(total_energy)
{  uint32_t j;
   for( j=0 ; j<nAdhoc ; j++)
   {
      Ptr<BasicEnergySource> basicSourcePtr = DynamicCast<BasicEnergySource> (sources.Get (j));
      basicSourcePtr->TraceConnectWithoutContext ("RemainingEnergy", MakeCallback(&TotalEnergy));
   }
```

```
}


Config::Connect("/NodeList/0/ApplicationList/0/$ns3::UdpClient/Generate",MakeCallback(&PacketGenera
ted));

Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxBegin",MakeCallback(&PacketT
ransmitted));
  std::ostringstream oss;
  oss << "/NodeList/" << nAdhoc-1 << "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd";
  Config::Connect(oss.str(),MakeCallback(&PacketReceived));

Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/MacTx",MakeCallback(&PacketMacTx)
);

Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxDrop",MakeCallback(&PacketTx
Drop));
  //if(static_position == false)
  //{
  //Config::Connect ("/NodeList/0/$ns3::MobilityModel/CourseChange", MakeCallback(&CourseChange));
  //}


  Simulator::Run ();
  std::cout << "Packets generated: " << packets_generated << std::endl;
  std::cout << "Packets Mac tx: " << packets_mac_tx << std::endl;
  std::cout << "Packets transmitted: " << packets_transmitted << std::endl;
  std::cout << "Packets received: " << packets_received << std::endl;
if(random_mobility)
{       for(uint32_t i=0; i<nAdhoc; i++)
        {
                Ptr<Node> node = wifiadhocnodes.Get(i);
                Ptr<MobilityModel> mob = node->GetObject<MobilityModel> ();
                Vector pos = mob->GetPosition ();
                std::cout << "Node " << i << " is at (" << pos.x << ", " << pos.y << ")\n";
        }
}
  Simulator::Destroy ();
if(total_energy)
{
    std::cout << "TOTAL ENERGY CONSUMED BY NETWORK " << total_network_energy << "J" << std::endl;
}
  return 0;
}
```

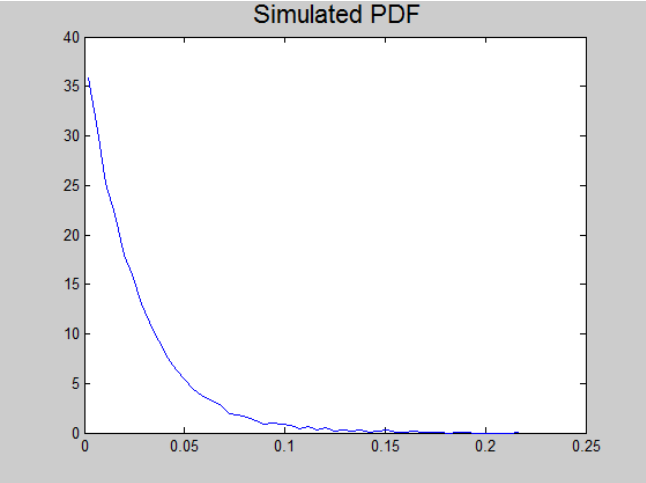
Annex 3: Output parsing python script

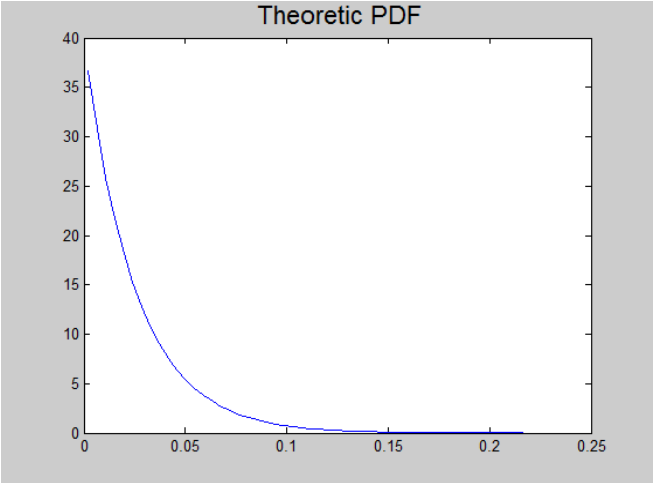
```
import re
import sys

#print 'Number of arguments:', len(sys.argv), 'arguments.'
#print 'Argument List:', str(sys.argv)
total_packets_transmitted = [0 for i in range (25)]
total_packets_received = [0 for i in range (25)]
total_olsr_packets_transmitted = [0 for i in range (25)]
total_olsr_packets_received = [0 for i in range (25)]
total_udp_packets_transmitted = [0 for i in range (25)]
total_udp_packets_received = [0 for i in range (25)]
total_other_packets_transmitted = [0 for i in range (25)]
total_other_packets_received = [0 for i in range (25)]
total_retries = [0 for i in range (25)]

in_f='wifi-simple-adhoc.tr'
ou_f='wifi-simple-adhoc-results.txt'

input_file=open(in_f,'r')
print('Input file: '+in_f)

while True:
        line=input_file.readline()
        if not line: break

        start_search = line.find ('/NodeList/') + 10
        end_search = line.find('/', start_search)
        node = line[start_search:end_search]
        #print(node)
        start_search = line.find ('Retry=') + 6
        end_search = line.find(',', start_search)
        retry = line[start_search:end_search]

        if retry == '1':
                total_retries[int(node)] += 1
        if (line.startswith('t')):
                total_packets_transmitted[int(node)] += 1
                if 'ns3::UdpHeader' in line:
                        if 'ns3::olsr::' in line:
                                total_olsr_packets_transmitted[int(node)] += 1
                        else:
                                total_udp_packets_transmitted[int(node)] += 1
                else:
                        total_other_packets_transmitted[int(node)] += 1

        else:
                total_packets_received[int(node)] += 1
                if 'ns3::UdpHeader' in line:
                        if 'ns3::olsr::' in line:
                                total_olsr_packets_received[int(node)] += 1
                        else:
                                total_udp_packets_received[int(node)] += 1
                else:
                        total_other_packets_received[int(node)] += 1


print('\n'+'--------------------------TRANSMITTED---------------------------------'+'\n')
print(total_packets_transmitted)
print(total_udp_packets_transmitted)
print(total_olsr_packets_transmitted)
print(total_other_packets_transmitted)
print('\n'+'--------------------------RECEIVED---------------------------------'+'\n')
print(total_packets_received)
print(total_udp_packets_received)
print(total_olsr_packets_received)
print(total_other_packets_received)
print('\n'+'--------------------------RETRIES---------------------------------'+'\n')
print(total_retries)
print('\n'+'------------------------------------------------------------------------'+'\n')
```

Annex 4: Exponential PDF of Interval Time and Length of simulated packets

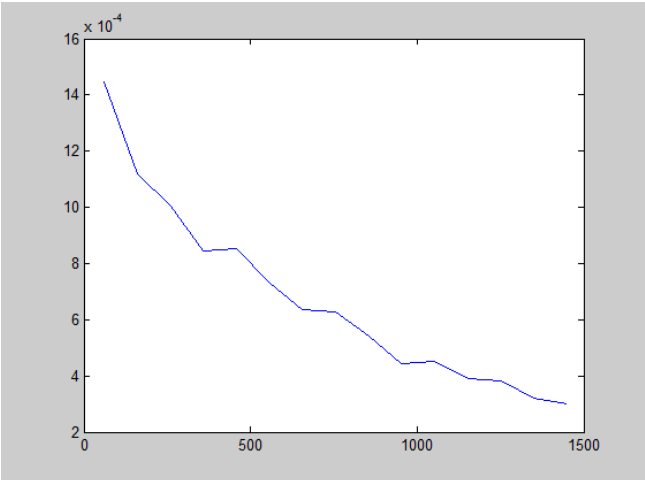
To demonstrate the exponential behaviour on the modifications we made to the NS3 source code, we have printed a list of 10000 packets and its inter-arrival generation times, after configuring a mean time of 25 ms and 1000 bytes. Computing with this list on Matlab, after a few lines provided below, the result is as expected:

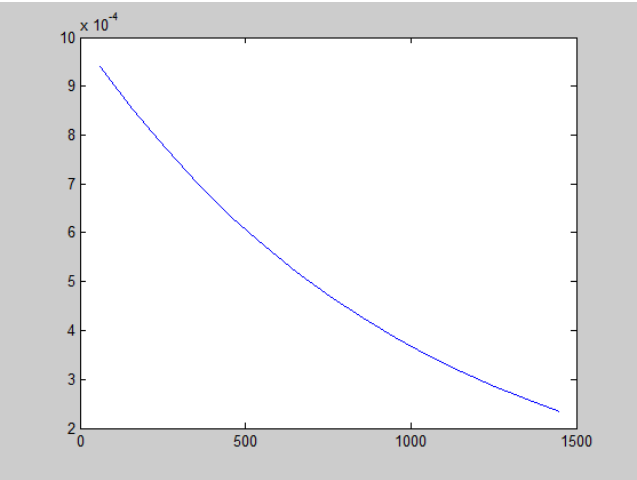
```
packets = importdata('packets.txt');
time = packets(:,1);
[ht,x] = hist(time,50);
area = (x(2)-x(1))*sum(ht);
pdf = ht./area;
plot(x,pdf)
plot(x,40*exp(-40*x))
```





But in the case of the packet length, the deviation from the theoretic exponential is much far than the interval simulations:





The reason why this happens is because we are cutting the packet length simulated to a minimum of 10 bytes, and a maximum of 1500. So the result is not a pure exponential random variable but a similar one (truncated exponential).

Glossary

WLAN: Wireless Local Area Network.

STA: Station.

AP: Access Point.

ST: Basic Station.

BSS: Basic Service Set.

ESS: Extended Service Set.

DS: Distribution System.

OLSR: Optimized Link State Protocol.

WMN: Wireless Mesh Network.