



EECE5155: Wireless Sensor Networks and the Internet of Things Computer Laboratory Assignment 2

Author (s):

- Ziyu Bian
- Liangshe Li
- Group number 21 on Canvas:

Date: 2020.11.13

Task 1

Define a Wireless Local Area Network (WLAN) operating in Ad-hoc Mode with 5 nodes. Nodes move by following a 2D random walk in a rectangular area defined by the lower-left corner ($x=-90$ m, $y=-90$ m) and the upper-right corner ($x=90$ m, $y=90$ m). Consider the following specifications:

- Channel: Default wireless channel in ns-3
- Physical Layer:
 - o Default parameters in IEEE 802.11G standard
 - o Adaptive rate control given by the AARF algorithm (default)
- Link Layer:
 - o Standard MAC without quality of service control
 - o Remember: the network should operate in ad-hoc mode
- Network Layer:
 - o Standard IPv4
 - o Address range: 192.168.1.0/24
 - o Assume that all the nodes behave as ideal routers and can exchange their routing tables in the background
- Transport Layer:
 - o UDP
- Application Layer:
 - o UDP Echo Server at Node 0: Listening on port 20
 - o UDP Echo Client at Node 4: Sends 2 UDP Echo packets to the server at times 1s and 2s
 - o UDP Echo Client at Node 3: Sends 2 UDP Echo packets to the server at times 2s and 4s
 - o Packet size: 512 bytes
- Additional parameters:
 - o Set up a packet tracer ONLY on node 2

1. Experimental setup:

Task1 requires us to establish the Wireless Local Area Network in Ad-hoc mode. The first thing we need to task consideration is the header files. Clearly, there is no need to build the CSMA network and point-to-point network which means these network's head files are not needed in the code. Owing to the fact, the head files we used in the part is just shown as the following:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/mobility-module.h"
#include "ns3/internet-module.h"
```

```
#include "ns3/yans-wifi-helper.h"
#include "ns3/ssid.h"
#include "ns3/config.h"
```

For this part, we need to make specific illustration to the head file of ssid. Actually, its full name is “Service Set Identifier”. Such a technology can divide one wireless local area network into several subnets which requires different identification. That is to say, only the people who pass the specific identification can get access to the corresponding subnet.

Next step we just gift the namespace with the name we wanted. The code is just shown as the following:

```
using namespace ns3;
```

In the next part, we are going to claim the log component whose name is “ThirdScriptExample”, we can turn on or turn off the output of the log component with the method of quoting the name, the code is just shown as the following:

```
NS_LOG_COMPONENT_DEFINE ("ThirdScriptExample");
```

After the above preparatory work done, we will begin to deal with the main function.

Just like previous files, we need to add some parameters of command lines to turn on or turn off some log components or just change the numbers of device we need to use.

```
bool verbose = true;
uint32_t nWifi = 5;
bool useRts = false;
```

Just like the code shown above, you can see one of the variables is called “useRts”. That means we want to use RTS/CTS method to deal with the issue of hidden stations. While it is true, then the real exchange of data only occurs after the signal of RTS/CTS protocol exchanged successfully. Then for the rest two variables, the variable whose name is verbose is just used to determine whether to open the two UdpApplications’ logging components, if it is true, then the components are opened, while for the other variable nWifi, we just use it to define how many station nodes we need in the wifi network. Since it is 5, then there are 5 station nodes in the wifi network we built.

For the next part, what we are going to do is just to create the object of CommandLine, we just name it as cmd. Then we will add corresponding parameters to it.

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

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

And for the next part, we are going to make some restriction to wifi network’s parameters. In the following code, you can clearly see that we set the parameter as 18. That’s because of the configuration of the grid position allocator. If it is larger than 18, then the grid will exceed boundary.

Afterwards, what we are going to do is just set the verbose parameter and useRts as True, so that we can track the file and work out the hidden station issue. The corresponding code is shown as the following:

```
if (nWifi > 18)
{
    std::cout << "nWifi should be 18 or less; otherwise grid layout exceeds the bounding box" << std::endl;
    return 1;
}

if (verbose)
```

```

{
    LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
}

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

```

This part we can get involved the work of network topology construction. First of all, we can construct the wifi network. What we need to do at the first step is just that we can create the wifi network nodes. The code used for creating these nodes are just shown as the following:

```

NodeContainer wifiAdhocNodes;
wifiAdhocNodes.Create (nWifi);

```

After nodes established, we can get down to channel setting as well as Wifiphy setting. Here we will use two helper classes: YansWifiChannelHelper and YansWifiPhyHelper. We need to make extra instruction to the definition of “phy”. Actually, that means Port Physical Layer, and phy is just the simplified name like OSI.

```

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

```

With these codes above, we can set three models, fist code can make us establish two models, they are constant speed propagation delay model and logdistance propagation loss model. With the code of second line, we can set NistErrorRate model. Finally, we just use the default configuration for PHY as well as default channel model.

After that, we can set WifiMac and install Netdevice in the nodes. There are two helper classes we will use here, they are WifiHelper and WifiMacHelper. Here is the code we used for the part:

```

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

WifiMacHelper mac;
Ssid ssid = Ssid ("ns-3-ssid");
mac.SetType ("ns3::AdhocWifiMac",
            "Ssid", SsidValue (ssid));

```

In this part, the method of SetRemoteStationManager can tell the helper to use which kind of algorithm. In the part1, we will use AARF algorithm. Then for the WifiMacHelper part, we are going to create the object of the class Ssid so that we can use it to set the value of SSID in the MAC layer. Finally, we will install the ad-hoc nodes in the Wifi network.

And this part we are going to build mobile model. The program will use Cartesian coordinate system to label the location of each node. The helper class we will use for here is MobilityHelper. We will make illustration to the following code:

```

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 (-90, 90, -90, 90)));
mobility.Install (wifiAdhocNodes);

```

While it comes to the model setting for moving nodes, we can divide the model into two parts. The first part is the original distribution of nodes while the second one is the moving trace model for the nodes. Clearly in the previous parts we have already determined to use grid pattern. That is the reason why we set the GridPositionAllocator as the allocator we used. After that, we can put our nodes into the 2-dimensional Cartesian coordinate system. DeltaX and DeltaY is the distance between each node in x-axis and y-axis. Then the "GridWidth" means the maximum number of nodes in each row. After the grid position allocator set, we can construct movement trajectory model in the part. As you see, the model we use is "RandomWalk2dMobilityModel". The default range of the speed for the nodes is between 2 m/s and 4m/s. And you can see that the parameter set in the instruction of SetMobilityModel is just to set the boundary for the region of nodes moving. And according to the issue, that should be -90, 90, -90 and 90. For the last step, we are going to install nodes in the model with the instruction "mobility.Install(wifiAdhocNodes)". Just like shown above.

```

InternetStackHelper stack;
stack.Install (wifiAdhocNodes);

Ipv4AddressHelper ipv4;

ipv4.SetBase ("192.168.1.0", "255.255.255.0");
Ipv4InterfaceContainer i = ipv4.Assign (adhocDevices);

```

Here we will install the TCP/IP protocol stack, the helper class we will use here is InternetStackHelper. First of all, we will create one object for the class. Then we will use the 'Install' instruction to install the nodes we created into the stack. Then we create one object for the class Ipv4AddressHelper with the name ipv4. What we will do here is to use the object to allocate IP address for the network devices.

- Network Layer:
 - o Standard IPv4 o Address range: 192.168.1.0/24
 - o Assume that all the nodes behave as ideal routers and can exchange their routing tables in the background

And according to the requirement of the issue, the address we will allocate to the wifi network is 192.168.1.0 and 255.255.255.0. And we will complete the part with the instruction "Assign".

- Transport Layer:
 - o UDP
- Application Layer:
 - o UDP Echo Server at Node 0: Listening on port 20
 - o UDP Echo Client at Node 4: Sends 2 UDP Echo packets to the server at times 1s and 2s
 - o UDP Echo Client at Node 3: Sends 2 UDP Echo packets to the server at times 2s and 4s
 - o Packet size: 512 bytes

After that we will build the application layer to meet the requirement of the issue. And we know there are two clients and one server in the whole network. The code we used to deal with this part is shown as following:

```

UdpEchoServerHelper echoServer (20);

```

```

ApplicationContainer serverApps = echoServer.Install (wifiAdhocNodes.Get (nWifi-5));
serverApps.Start (Seconds (0.5));
serverApps.Stop (Seconds (10.0));

```

```

UdpEchoClientHelper echoClient1 (i.GetAddress(nWifi-5), 20);
echoClient1.SetAttribute ("MaxPackets", UIntegerValue (2));
echoClient1.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient1.SetAttribute ("PacketSize", UIntegerValue (512));

```

```

ApplicationContainer clientApps1 =
    echoClient1.Install (wifiAdhocNodes.Get (nWifi - 1));
clientApps1.Start (Seconds (1.0));
clientApps1.Stop (Seconds (10.0));

```

```

UdpEchoClientHelper echoClient2 (i.GetAddress(nWifi-5), 20);
echoClient2.SetAttribute ("MaxPackets", UIntegerValue (2));
echoClient2.SetAttribute ("Interval", TimeValue (Seconds (2.0)));
echoClient2.SetAttribute ("PacketSize", UIntegerValue (512));

```

```

ApplicationContainer clientApps2 =
    echoClient2.Install (wifiAdhocNodes.Get (nWifi - 2));
clientApps2.Start (Seconds (2.0));
clientApps2.Stop (Seconds (10.0));

```

```

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

```

First of all, we create the echoServer who is the object of UdpEchoServer. The parameter is the index of port where it listens to the response. According to the issue, it is set as 20.

After the server program set, we need to install it into one specific node. Since the issue asks us to install it at Node 0. We have already known the total number of nodes nWifi is 5. Then we can write the index as 0 or nWifi-5 just like it shown above.

The part we need to be cautious is that they have two clients, so we need to use UdpEchoClientHelper for twice.

- Additional parameters:
 - o Set up a packet tracer ONLY on node 2

```

Simulator::Stop (Seconds (10.0));

```

```

phy.EnablePcap ("mythird1", adhocDevices.Get (2));

```

```

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

```

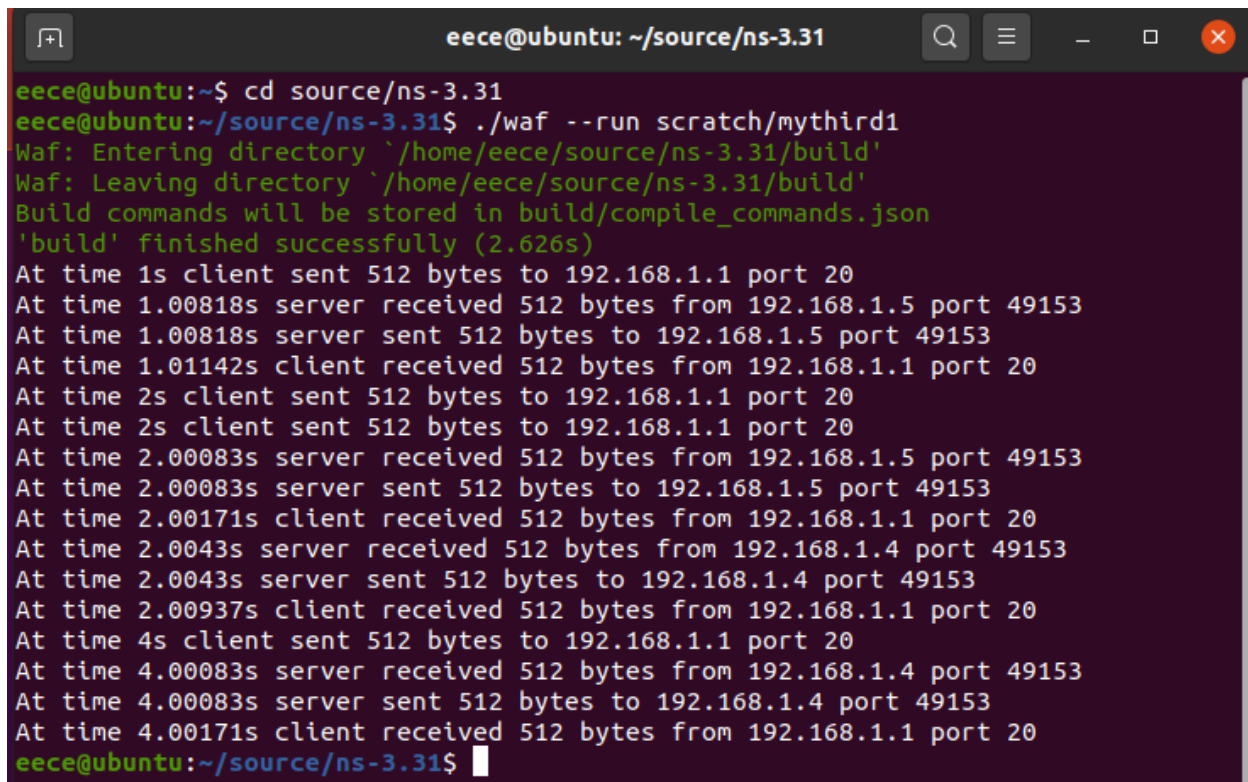
First of all, we can see the class we used for the first line is Ipv4GlobalRoutingHelper. We create one object of that class to help us to create routing table for each node based on the link-state advertisement of each node.

One thing that we need to notice is that the simulation we created will never naturally stop. That is because we asked the wireless access points to generate beacons. It will generate beacons forever and this will result in simulator events being scheduled into the future indefinitely. What does that mean? We have to tell the simulator to stop even though it might have beacon generation events scheduled. That second line of code shown above will help us to tell the simulator to stop so that we don't simulate beacons forever and enter what is essentially an endless loop.

Then as you can see above, we need to only trace the package on node 2. And the index of node begins with 0. That means the parameter for EnablePcap is just 1 like we did above. Finally, the last three lines of code can make us run the simulation, clean up and then exit the program.

2. Results:

As Fig.1-1 shows, we first see a packet going from the client (IP address: 192.168.1.5 port 49153) to the server (IP address:192.168.1.1 port 20) at time 1s. The propagation delay time is 8.18ms. The server then sends back the same packet. The propagation delay time is 3.24ms. Then we see a packet going from the client (IP address: 192.168.1.5 port 49153) to the server (IP address:192.168.1.1 port 20) at time 2s. The propagation delay time is 0.83ms. The server then sends back the same packet. The propagation delay time is 0.88ms. Then we see that the server (IP address:192.168.1.1 port 20) receives a packet going from the client (IP address: 192.168.1.4 port 49153) at time 2.0043s. The server then sends back the same packet. The propagation delay time is 5.937ms. Then we see a packet going from the client (IP address: 192.168.1.5 port 49153) to the server (IP address:192.168.1.1 port 20) at time 4s. The propagation delay time is 0.83ms. The server then sends back the same packet at time 4.00083s. The propagation delay time is 0.88ms.



```
eece@ubuntu:~/source/ns-3.31$ cd source/ns-3.31
eece@ubuntu:~/source/ns-3.31$ ./waf --run scratch/mythird1
Waf: Entering directory `/home/eece/source/ns-3.31/build'
Waf: Leaving directory `/home/eece/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (2.626s)
At time 1s client sent 512 bytes to 192.168.1.1 port 20
At time 1.00818s server received 512 bytes from 192.168.1.5 port 49153
At time 1.00818s server sent 512 bytes to 192.168.1.5 port 49153
At time 1.01142s client received 512 bytes from 192.168.1.1 port 20
At time 2s client sent 512 bytes to 192.168.1.1 port 20
At time 2s client sent 512 bytes to 192.168.1.1 port 20
At time 2.00083s server received 512 bytes from 192.168.1.5 port 49153
At time 2.00083s server sent 512 bytes to 192.168.1.5 port 49153
At time 2.00171s client received 512 bytes from 192.168.1.1 port 20
At time 2.0043s server received 512 bytes from 192.168.1.4 port 49153
At time 2.0043s server sent 512 bytes to 192.168.1.4 port 49153
At time 2.00937s client received 512 bytes from 192.168.1.1 port 20
At time 4s client sent 512 bytes to 192.168.1.1 port 20
At time 4.00083s server received 512 bytes from 192.168.1.4 port 49153
At time 4.00083s server sent 512 bytes to 192.168.1.4 port 49153
At time 4.00171s client received 512 bytes from 192.168.1.1 port 20
eece@ubuntu:~/source/ns-3.31$
```

Fig.1-1 Operation results of mythird1.cc

All packets are 512 bytes. In a word, we see that the operation results fit the requests of Task1. At the same time, mythird1.cc produces mythird1-2-0.pcap which is a tracing packet of Node2. We can now get the promiscuous trace of the Wi-Fi network like Fig.1-2:


```

9.995467 Beacon (ECCESISS) [0.0* 9.0 12.0* 18.0 24.0* 30.0 48.0 54.0 MBL] ESS
eece@ubuntu:~/source/ns-3.31$ tcpdump -nn -tt -r mythird1-2-0.pcap
reading from file mythird1-2-0.pcap, link-type IEEE802.11 (802.11)
1.007146 ARP, Request who-has 192.168.1.1 (ff:ff:ff:ff:ff:ff) tell 192.168.1.5, length 32
1.007292 ARP, Reply 192.168.1.1 is-at 00:00:00:00:00:01, length 32
1.007352 Acknowledgment RA:00:00:00:00:00:01
1.008178 IP 192.168.1.5.49153 > 192.168.1.1.20: UDP, length 512
1.008238 Acknowledgment RA:00:00:00:00:00:05
1.010324 ARP, Request who-has 192.168.1.5 (ff:ff:ff:ff:ff:ff) tell 192.168.1.1, length 32
1.010470 ARP, Reply 192.168.1.5 is-at 00:00:00:00:00:05, length 32
1.010530 Acknowledgment RA:00:00:00:00:00:05
1.011419 IP 192.168.1.1.20 > 192.168.1.5.49153: UDP, length 512
1.011479 Acknowledgment RA:00:00:00:00:00:01
2.000826 IP 192.168.1.5.49153 > 192.168.1.1.20: UDP, length 512
2.000886 Acknowledgment RA:00:00:00:00:00:05
2.001712 IP 192.168.1.1.20 > 192.168.1.5.49153: UDP, length 512
2.001772 Acknowledgment RA:00:00:00:00:00:01
2.003146 ARP, Request who-has 192.168.1.1 (ff:ff:ff:ff:ff:ff) tell 192.168.1.4, length 32
2.003292 ARP, Reply 192.168.1.1 is-at 00:00:00:00:00:01, length 32
2.003352 Acknowledgment RA:00:00:00:00:00:01
2.004304 IP 192.168.1.4.49153 > 192.168.1.1.20: UDP, length 512
2.004364 Acknowledgment RA:00:00:00:00:00:04
2.008324 ARP, Request who-has 192.168.1.4 (ff:ff:ff:ff:ff:ff) tell 192.168.1.1, length 32
2.008470 ARP, Reply 192.168.1.4 is-at 00:00:00:00:00:04, length 32
2.008530 Acknowledgment RA:00:00:00:00:00:04
2.009365 IP 192.168.1.1.20 > 192.168.1.4.49153: UDP, length 512
2.009425 Acknowledgment RA:00:00:00:00:00:01
2.011450 ARP, Request who-has 192.168.1.4 (ff:ff:ff:ff:ff:ff) tell 192.168.1.1, length 32
2.011596 ARP, Reply 192.168.1.4 is-at 00:00:00:00:00:04, length 32
2.011656 Acknowledgment RA:00:00:00:00:00:04
4.000826 IP 192.168.1.4.49153 > 192.168.1.1.20: UDP, length 512
4.000886 Acknowledgment RA:00:00:00:00:00:04
4.001712 IP 192.168.1.1.20 > 192.168.1.4.49153: UDP, length 512
4.001772 Acknowledgment RA:00:00:00:00:00:01
eece@ubuntu:~/source/ns-3.31$

```

Fig.1-2 Monitor mode of mythird1-2-0.pcap

When we open mythird1-1-0.pcap, we can get Fig.1-3:

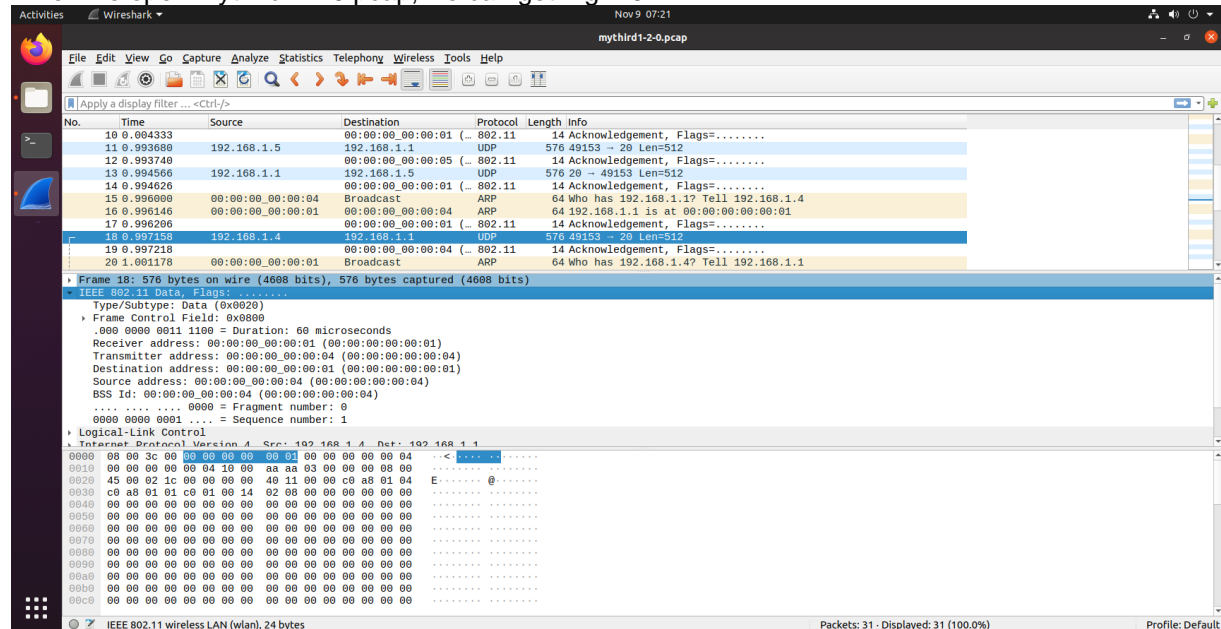


Fig.1-3 Mythird1-2-0.pcap

We can see that the protocol of Wi-Fi is 802.11, and the datagram protocol is UDP. When one client sends the message to the server, 64 bytes will be added into the frame to compose the header of the frame.

3. Answering Questions:

Q1: Are all the frames acknowledged? Explain why.

A: From Fig.1-2 we can find that every time one node sends a message to another one, there is an Acknowledgement Replying Address (Acknowledgement RA), and all frames have such addresses so we can say all the frames are acknowledged.

Q2: Are there any collisions in the network? Explain why. How have you reached this conclusion?

A: (1) There is no collision in the network. Although Node4(Client1) and Node3(Client2) send the message to Node0(Server) at the same time (time 2s), the message from Node4 will reach Node0 first for that the Node4 has already set the channel with Node0, while Node3 needs to send ARP to ask for the address of Node0. So these two packets will not reach the server at the same time.

(2) From Fig.1-2, we can see first (time 2.000826s) the message from Node4 will reach Node0 and then (time 2.003146s) Node3 will send ARP broadcast to ask the address of Node0.

Q3: How can you force the nodes to utilize the RTS/CTS handshake procedure seen in class? What is the reasoning behind this procedure?

A: (1) In the codes we add a Bool value whose name is "useRts" and a head file whose name is "ns3/config.h". We use "useRts" to control the opening of RtsCtsThreshold. When the value of "useRts" is true, the nodes will utilize the RTS/CTS handshake procedure. If the value is false, the nodes will not use RTS/CTS procedure.

(2) When a node needs to transmit a frame, it will listen to the channel to see if anyone else is transmitting: If the channel is free, it waits for a short period of time and transmits a Request-To-Send (RTS) frame to the receiver; If the receiver properly receives the RTS and no other nodes are active in its neighborhood, it replies with a Clear-To-Send (CTS) frame after waiting for a very short time; If the transmitter properly receives the CTS and no other nodes are active in its neighborhood, it transmits the data frame after waiting for a very short time; If the receiver properly receives the data frame and no other nodes are active in its neighborhood, it transmits an ACK frame after waiting for a very short time.

Q4: Force the utilization of RTS/CTS in the network:

o Are there any collisions now?

o Which is the benefit or RTS/CTS?

o Where can you find the Network Allocation Vector information?

A: (1) There are no collisions now.

(2) The benefit of RTS/CTS is that we can avoid data frames collisions. The RTS frame specifies how much data the node needs to transmit (so the receiver can compute for how long will the channel be occupied). The CTS frame specifies for how long the channel will be occupied (so its neighbors do not attempt to transmit during that time). The length of the RTS and CTS frames is much shorter than the length of the data frame.

(3) The Network Allocation Vector (NAV) is the CTS frame in IEEE 802.11 Wi-Fi Standard. So when we make the value of "useRts" become true and get a new mythird1-1-0.pcap (open it), we can find the Network Allocation Vector information like Fig.1-4. The CTS frame is 14 bytes.

The image shows a Wireshark packet capture window titled "mythird1-2-0.pcap". The packet list pane shows several packets, with packet 7 selected. The packet details pane shows the structure of the IEEE 802.11 Clear-to-send (CTS) frame. The packet bytes pane shows the raw data of the frame.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	00:00:00_00:00:05	Broadcast	ARP	64	Who has 192.168.1.1? Tell 192.168.1.5
2	0.000086	00:00:00_00:00:01 (...)	00:00:00_00:00:05 (...)	802.11	20	Request-to-send, Flags=.....
3	0.000146	00:00:00_00:00:01 (...)	00:00:00_00:00:01 (...)	802.11	14	Clear-to-send, Flags=.....
4	0.000274	00:00:00_00:00:01 (...)	00:00:00_00:00:05	ARP	64	192.168.1.1 is at 00:00:00:00:00:01
5	0.000334	00:00:00_00:00:01 (...)	00:00:00_00:00:01 (...)	802.11	14	Acknowledgement, Flags=.....
6	0.000420	00:00:00_00:00:05 (...)	00:00:00_00:00:01 (...)	802.11	20	Request-to-send, Flags=.....
7	0.000480	00:00:00_00:00:05 (...)	00:00:00_00:00:05 (...)	802.11	14	Clear-to-send, Flags=.....
8	0.001288	192.168.1.5	192.168.1.1	UDP	576	49153 → 20 Len=512
9	0.001348	00:00:00_00:00:05 (...)	00:00:00_00:00:05 (...)	802.11	14	Acknowledgement, Flags=.....
10	0.003434	00:00:00_00:00:01 (...)	Broadcast	ARP	64	Who has 192.168.1.5? Tell 192.168.1.1
11	0.003520	00:00:00_00:00:05 (...)	00:00:00_00:00:01 (...)	802.11	20	Request-to-send, Flags=.....

Frame 7: 14 bytes on wire (112 bits), 14 bytes captured (112 bits)
IEEE 802.11 Clear-to-send, Flags:

0000 c4 00 64 03 00 00 00 00 05 00 00 00 ...d.....

Fig.1-4 the CTS frames

4. Learnt lessons:

In this first task, we have simulated a WLAN operating in Ad-hoc Mode and we use the Wireshark to show the condition of network by tracing Node2. After that we utilize the RTS/CTS procedure in the network and observe the difference.

Task 2

Define a Wireless Local Area Network (LAN) operating in Infrastructure Mode with 5 nodes and access point. Nodes move by following a 2D random walk in a rectangular area defined by the lower-left corner (x=-90 m, y=-90 m) and the upper-right corner (x=90 m, y=90 m). The network name (SSID) should be EECE5155. To start, do not force the handshaking process. Consider the following specifications:

- Channel: Default wireless channel in ns-3
- Physical Layer:
 - o Default parameters in IEEE 802.11G standard
 - o Adaptive rate control given by the AARF (default)
- Link Layer:
 - o Standard MAC without quality of service control
 - o Remember: the network should operate in infrastructure mode
- Network Layer:
 - o Standard IPv4
 - o Address range: 192.168.2.0/24
 - o Assume that all the nodes behave as ideal routers and can exchange their routing tables in the background
- Transport Layer: o UDP
- Application Layer:
 - o UDP Echo Server at Node 0: Listening on port 21
 - o UDP Echo Client at Node 3: Sends 2 UDP Echo packets to the server at times 2s and 4s
 - o UDP Echo Client at Node 4: Sends 2 UDP Echo packets to the server at times 1s and 4s
 - o Packet size: 512 bytes
- Additional parameters:
 - o Set up a packet tracer ONLY on node 4(one of the clients) and on the AP.

1. Experimental setup:

First task requires us to establish the Wireless Local Area Network in Ad-hoc mode. The first thing we need to task consideration is the header files. Clearly, there is no need to build the CSMA network and point-to-point network which means these network's head files are not needed in the code. Owing to the fact, the head files we used in the part is just shown as the following:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/mobility-module.h"
#include "ns3/internet-module.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/ssid.h"
```

For this part, we need to make specific illustration to the head file of ssid. Actually, its full name is "Service Set Identifier". Such a technology can divide one wireless local area network into several subnets which requires different identification. That is to say, only the people who pass the specific identification can get access to the corresponding subnet.

Next step we just gift the namespace with the name we wanted. The code is just shown as the following:

```
using namespace ns3;
```

In the next part, we are going to claim the log component whose name is "ThirdScriptExample", we can turn on or turn off the output of the log component with the method of quoting the name, the code is just shown as the following:

```
NS_LOG_COMPONENT_DEFINE ("ThirdScriptExample");
```

After the above preparatory work done, we will begin to deal with the main function.

Just like previous files, we need to add some parameters of command lines to turn on or turn off some log components or just change the numbers of device we need to use.

```
bool verbose = true;
uint32_t nWifi = 5;
bool useRts = true;
```

Just like the code shown above, you can see one of the variables is called "useRts". That means we want to use RTS/CTS method to deal with the issue of hidden stations. While it is true, then the real exchange of data only occurs after the signal of RTS/CTS protocol exchanged successfully. Then for the rest two variables, the variable whose name is verbose is just used to determine whether to open the two UdpApplications' logging components, if it is true, then the components are opened, while for the other variable nWifi, we just use it to define how many station nodes we need in the wifi network. Since it is 5, then there are 5 station nodes in the wifi network we built.

For the next part, what we are going to do is just to create the object of CommandLine, we just name it as cmd. Then we will add corresponding parameters to it.

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

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

And for the next part, we are going to make some restriction to wifi network's parameters. In the following code, you can clearly see that we set the parameter as 18. That's because of the configuration of the grid position allocator. If it is larger than 18, then the grid will exceed boundary.

Afterwards, what we are going to do is just set the verbose parameter and useRts as True, so that we can track the file and work out the hidden station issue. The corresponding code is shown as the following:

```
if (nWifi > 18)
{
    std::cout << "nWifi should be 18 or less; otherwise grid layout exceeds the bounding box" << std::endl;
    return 1;
}

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

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

This part we can get involved the work of network topology construction. First of all, we can construct the wifi network. What we need to do at the first step is just that we can create the wifi network nodes. We need to create nodes of two different types. First type of node is STA nodes while the other is Ap node. The code used for creating these nodes are just shown as the following:

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

After nodes established, we can get down to channel setting as well as Wifiphy setting. Here we will use two helper classes: YansWifiChannelHelper and YansWifiPhyHelper. We need to make extra instruction to the definition of “phy”. Actually, that means Port Physical Layer, and phy is just the simplified name like OSI.

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

With these codes above, we can set three models, fist code can make us establish two models, they are constant speed propagation delay model and logdistance propagation loss model. With the code of second line, we can set NistErrorRate model. Finally, we just use the default configuration for PHY as well as default channel model.

After that, we can set WifiMac and install Netdevice in the nodes. There are two helper classes we will use here, they are WifiHelper and WifiMacHelper. Here is the code we used for the part:

```
WifiHelper wifi;
wifi.SetRemoteStationManager ("ns3::AarfWifiManager");
```

```
WifiMacHelper mac;
Ssid ssid = Ssid ("EECE5155");
mac.SetType ("ns3::StaWifiMac",
            "Ssid", SsidValue (ssid));
```

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

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

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

In this part, the method of SetRemoteStationManager can tell the helper to use which kind of algorithm. In the part1, we will use AARF algorithm. Then for the WifiMacHelper part, we are going to create the object of the class Ssid so that we can use it to set the value of SSID in the MAC layer. From codes above, we can see the value of Ssid has become “EECE5155”. Finally, we will install the wifiStaNodes in the Wi-Fi network. The wifiApNode is set as the same.

And this part we are going to build mobile model. The program will use Cartesian coordinate system to label the location of each node. The helper class we will use for here is MobilityHelper. We will make illustration to the following code:

```
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 (-90, 90, -90, 90)));
mobility.Install (wifiStaNodes);

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

```

While it comes to the model setting for moving nodes, we can divide the model into two parts. The first part is the original distribution of nodes while the second one is the moving trace model for the nodes. Clearly in the previous parts we have already determined to use grid pattern. That is the reason why we set the GridPositionAllocator as the allocator we used. After that, we can put our nodes into the 2-dimensional Cartesian coordinate system. DeltaX and DeltaY is the distance between each node in x-axis and y-axis. Then the "GridWidth" means the maximum number of nodes in each row. After the grid position allocator set, we can construct movement trajectory model in the part. As you see, the model we use is "RandomWalk2dMobilityModel". The default range of the speed for the nodes is between 2 m/s and 4m/s. And you can see that the parameter set in the instruction of SetMobilityModel is just to set the boundary for the region of nodes moving. And according to the issue, that should be -90, 90, -90 and 90. For the last step, we are going to install nodes in the model with the instruction "mobility.Install (wifiStahocNodes)". Just like shown above. Next the AP node just needs "ConstantPositionMobilityModel".

```

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

Ipv4AddressHelper ipv4;

ipv4.SetBase ("192.168.2.0", "255.255.255.0");
Ipv4InterfaceContainer i = ipv4.Assign (staDevices);
ipv4.Assign (apDevices);

```

Here we will install the TCP/IP protocol stack, the helper class we will use here is InternetStackHelper. First of all, we will create one object for the class. Then we will use the 'Install' instruction to install the nodes we created into the stack. Then we create one object for the class Ipv4AddressHelper with the name ipv4. What we will do here is to use the object to allocate IP address for the network devices. Since there are two types of nodes, we need to repeat the step of address assigning twice.

- Network Layer:
 - o Standard IPv4 o Address range: 192.168.2.0/24
 - o Assume that all the nodes behave as ideal routers and can exchange their routing tables in the background

And according to the requirement of the issue, the address we will allocate to the Wi-Fi network is 192.168.2.0 and 255.255.255.0. And we will complete the part with the instruction "Assign".

- Transport Layer: o UDP
- Application Layer:
 - o UDP Echo Server at Node 0: Listening on port 21

- o UDP Echo Client at Node 3: Sends 2 UDP Echo packets to the server at times 2s and 4s
- o UDP Echo Client at Node 4: Sends 2 UDP Echo packets to the server at times 1s and 4s
- o Packet size: 512 bytes

After that we will build the application layer to meet the requirement of the issue. And we know there are two clients and one server in the whole network. The code we used to deal with this part is shown as following:

```
UdpEchoServerHelper echoServer (21);
```

```
ApplicationContainer serverApps = echoServer.Install (wifiStaNodes.Get (nWifi-5));
serverApps.Start (Seconds (0.5));
serverApps.Stop (Seconds (10.0));
```

```
UdpEchoClientHelper echoClient1 (i.GetAddress(nWifi-5), 21);
echoClient1.SetAttribute ("MaxPackets", UIntegerValue (2));
echoClient1.SetAttribute ("Interval", TimeValue (Seconds (2.0)));
echoClient1.SetAttribute ("PacketSize", UIntegerValue (512));
```

```
ApplicationContainer clientApps1 =
    echoClient1.Install (wifiStaNodes.Get (nWifi - 2));
clientApps1.Start (Seconds (2.0));
clientApps1.Stop (Seconds (10.0));
```

```
UdpEchoClientHelper echoClient2 (i.GetAddress(nWifi-5), 21);
echoClient2.SetAttribute ("MaxPackets", UIntegerValue (2));
echoClient2.SetAttribute ("Interval", TimeValue (Seconds (3.0)));
echoClient2.SetAttribute ("PacketSize", UIntegerValue (512));
```

```
ApplicationContainer clientApps2 =
    echoClient2.Install (wifiStaNodes.Get (nWifi - 1));
clientApps2.Start (Seconds (1.0));
clientApps2.Stop (Seconds (10.0));
```

First of all, we create the echoServer who is the object of UdpEchoServer. The parameter is the index of port where it listens to the response. According to the issue, it is set as 21.

After the server program set, we need to install it into one specific node. Since the issue asks us to install it at Node 0. We have already known the total number of nodes nWifi is 5. Then we can write the index as 0 or nWifi-5 just like it shown above.

The part we need to be cautious is that they have two clients, so we need to use UdpEchoClientHelper for twice. According to the issue, the clients are at node 4 and node 3. To make the input more convenient, we just set index of 4 and 3 as nWifi - 1 and nWifi - 2.

Furthermore, you can see that two clients will both send two packages. Client 1 will send its package at 1s and 4s, while the client 2 will send its packages at 2s and 4s. So our server will begin to listen at 1s to avoid package loss.

- Additional parameters:
 - o Set up a packet tracer ONLY on node 4(one of the clients) and on the AP.

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

```
Simulator::Stop (Seconds (10.0));
```

```
phy.EnablePcap ("mythird2", staDevices.Get (4));
phy.EnablePcap ("mythird2", apDevices.Get (0));
```

```

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

```

First of all, we can see the class we used for the first line is `Ipv4GlobalRoutingHelper`. We create one object of that class to help us to create routing table for each node based on the link-state advertisement of each node.

One thing that we need to notice is that the simulation we created will never naturally stop. That is because we asked the wireless access points to generate beacons. It will generate beacons forever and this will result in simulator events being scheduled into the future indefinitely. What does that mean? We have to tell the simulator to stop even though it might have beacon generation events scheduled. That second line of code shown above will help us to tell the simulator to stop so that we don't simulate beacons forever and enter what is essentially an endless loop.

Then as you can see above, we need to only trace the package on node 4. And the index of node begins with 0. That means the parameter for `EnablePcap` is just 1 like we did above.

Finally, the last three lines of code can make us run the simulation, clean up and then exit the program.

2. Results:

As Fig.2-1 shows, we first see a packet going from the client (IP address: 192.168.2.5 port 49153) to the server (IP address:192.168.2.1 port 21) at time 1s. The propagation delay time is 7.75ms. The server then sends back the same packet. The propagation delay time is 7.58ms. Then we see a packet going from the client (IP address: 192.168.2.4 port 49153) to the server (IP address:192.168.2.1 port 21) at time 2s. The propagation delay time is 4.58ms. The server then sends back the same packet. The propagation delay time is 6.47ms. Then we see the server (IP address:192.168.2.1 port 21) receives a packet from the client ((IP address:192.168.2.5 port 49153)) at time 4.00171s. The server then sends back the same packet. The propagation delay time is 1.77ms. Then we see the server (IP address:192.168.2.1 port 21) receives a packet from the client ((IP address:192.168.2.4 port 49153)) at time 4.00534s. The server then sends back the same packet. The propagation delay time is 1.74ms.

```

eece@ubuntu: ~/source/ns-3.31
eece@ubuntu:~/source/ns-3.31$ ./waf --run scratch/mythird2
Waf: Entering directory '/home/eece/source/ns-3.31/build'
[2814/2887] Compiling scratch/mythird2.cc
[2847/2887] Linking build/scratch/mythird2
Waf: Leaving directory '/home/eece/source/ns-3.31/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (5.381s)
At time 1s client sent 512 bytes to 192.168.2.1 port 21
At time 1.00775s server received 512 bytes from 192.168.2.5 port 49153
At time 1.01533s client received 512 bytes from 192.168.2.1 port 21
At time 2s client sent 512 bytes to 192.168.2.1 port 21
At time 2.00458s server received 512 bytes from 192.168.2.4 port 49153
At time 2.01105s client received 512 bytes from 192.168.2.1 port 21
At time 4s client sent 512 bytes to 192.168.2.1 port 21
At time 4s client sent 512 bytes to 192.168.2.1 port 21
At time 4.00171s server received 512 bytes from 192.168.2.5 port 49153
At time 4.00171s server sent 512 bytes to 192.168.2.5 port 49153
At time 4.00348s client received 512 bytes from 192.168.2.1 port 21
At time 4.00534s server received 512 bytes from 192.168.2.4 port 49153
At time 4.00534s server sent 512 bytes to 192.168.2.4 port 49153
At time 4.00718s client received 512 bytes from 192.168.2.1 port 21
eece@ubuntu:~/source/ns-3.31$

```

Fig.2-1 operation of mythird2.cc

All packets are 512 bytes. In a word, we can see that the operation results fit the requests of Task2.

At the same time, mythird2.cc produces mythird2-4-0.pcap which is a tracing packet of Node4 and mythird2-5-0.pcap which is a tracing packet of the access point (AP). We can see the monitor modes of mythird2-4-0.pcap and mythird2-5-0.pcap in Fig.2-2 and Fig.2-3:

```
eece@ubuntu:~/source/ns-3.31$ tcpdump -nn -tt -r mythird2-4-0.pcap
reading from file mythird2-4-0.pcap, link-type IEEE802_11 (802.11)
0.062667 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.120034 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.120422 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.120482 Acknowledgment RA:00:00:00:00:00:03
0.120600 Assoc Response AID(1) :: Successful
0.120660 Acknowledgment RA:00:00:00:00:00:06
0.120757 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.120913 Acknowledgment RA:00:00:00:00:00:05
0.121061 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.121121 Acknowledgment RA:00:00:00:00:00:02
0.121287 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.121347 Acknowledgment RA:00:00:00:00:00:01
0.121474 Assoc Response AID(2) :: Successful
0.121490 Acknowledgment RA:00:00:00:00:00:06
0.121914 Assoc Response AID(3) :: Successful
0.121974 Acknowledgment RA:00:00:00:00:00:06
0.122218 Assoc Response AID(4) :: Successful
0.122278 Acknowledgment RA:00:00:00:00:00:06
0.122714 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.122774 Acknowledgment RA:00:00:00:00:00:04
0.122892 Assoc Response AID(5) :: Successful
0.122952 Acknowledgment RA:00:00:00:00:00:06
0.165067 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.267467 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.369867 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.472267 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.574667 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.677067 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.779467 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.881867 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.984267 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
1.005034 ARP, Request who-has 192.168.2.1 (ff:ff:ff:ff:ff:ff) tell 192.168.2.5, length 32
1.005206 Acknowledgment RA:00:00:00:00:00:05
1.005352 ARP, Request who-has 192.168.2.1 (ff:ff:ff:ff:ff:ff) tell 192.168.2.5, length 32
1.005498 ARP, Reply 192.168.2.1 is-at 00:00:00:00:00:01, length 32
1.005558 Acknowledgment RA:00:00:00:00:00:01
1.005839 ARP, Reply 192.168.2.1 is-at 00:00:00:00:00:01, length 32
1.005855 Acknowledgment RA:00:00:00:00:00:06
1.005933 IP 192.168.2.5.49153 > 192.168.2.1.21: UDP, length 512
1.006785 Acknowledgment RA:00:00:00:00:00:05
1.007746 IP 192.168.2.5.49153 > 192.168.2.1.21: UDP, length 512
1.007806 Acknowledgment RA:00:00:00:00:00:06
```

Fig.2-2 Monitor mode of mythird2-4-0.pcap

```
eece@ubuntu:~/source/ns-3.31$ tcpdump -nn -tt -r mythird2-5-0.pcap
reading from file mythird2-5-0.pcap, link-type IEEE802_11 (802.11)
0.062563 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.120422 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.120438 Acknowledgment RA:00:00:00:00:00:03
0.120516 Assoc Response AID(1) :: Successful
0.120660 Acknowledgment RA:00:00:00:00:00:06
0.120853 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.120869 Acknowledgment RA:00:00:00:00:00:05
0.121061 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.121077 Acknowledgment RA:00:00:00:00:00:02
0.121287 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.121303 Acknowledgment RA:00:00:00:00:00:01
0.121390 Assoc Response AID(2) :: Successful
0.121534 Acknowledgment RA:00:00:00:00:00:06
0.121577 Assoc Response AID(3) :: Successful
0.121830 Assoc Response AID(3) :: Successful
0.121974 Acknowledgment RA:00:00:00:00:00:06
0.122134 Assoc Response AID(4) :: Successful
0.122278 Acknowledgment RA:00:00:00:00:00:06
0.122714 Assoc Request (EECES155) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.122730 Acknowledgment RA:00:00:00:00:00:04
0.122808 Assoc Response AID(5) :: Successful
0.122952 Acknowledgment RA:00:00:00:00:00:06
0.164963 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.267363 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.369763 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.472163 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.574563 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.676963 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.779363 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.881763 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
0.984163 Beacon (EECES155) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS
1.005146 ARP, Request who-has 192.168.2.1 (ff:ff:ff:ff:ff:ff) tell 192.168.2.5, length 32
1.005162 Acknowledgment RA:00:00:00:00:00:05
1.005240 ARP, Request who-has 192.168.2.1 (ff:ff:ff:ff:ff:ff) tell 192.168.2.5, length 32
1.005498 ARP, Reply 192.168.2.1 is-at 00:00:00:00:00:01, length 32
1.005514 Acknowledgment RA:00:00:00:00:00:01
1.005727 ARP, Reply 192.168.2.1 is-at 00:00:00:00:00:01, length 32
1.005899 Acknowledgment RA:00:00:00:00:00:06
1.006725 IP 192.168.2.5.49153 > 192.168.2.1.21: UDP, length 512
1.006741 Acknowledgment RA:00:00:00:00:00:05
1.006954 IP 192.168.2.5.49153 > 192.168.2.1.21: UDP, length 512
1.007806 Acknowledgment RA:00:00:00:00:00:06
```

Fig.2-3 Monitor mode of mythird2-5-0.pcap

From Fig.2-2 and Fig.2-3 we can see the network name is EECES155. When opening mythird2-4-0.pcap and mythird2-5-0.pcap using Wireshark, we can get Fig.2-4 and Fig.2-5:

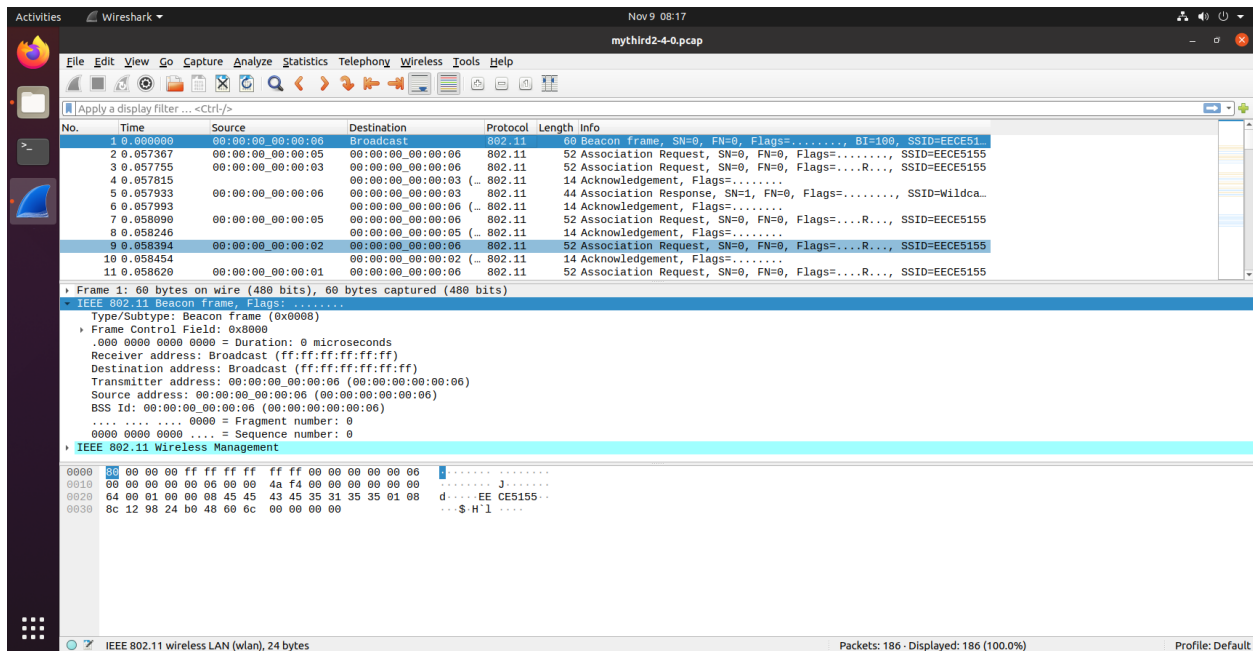


Fig.2-4 mythird2-4-0.pcap

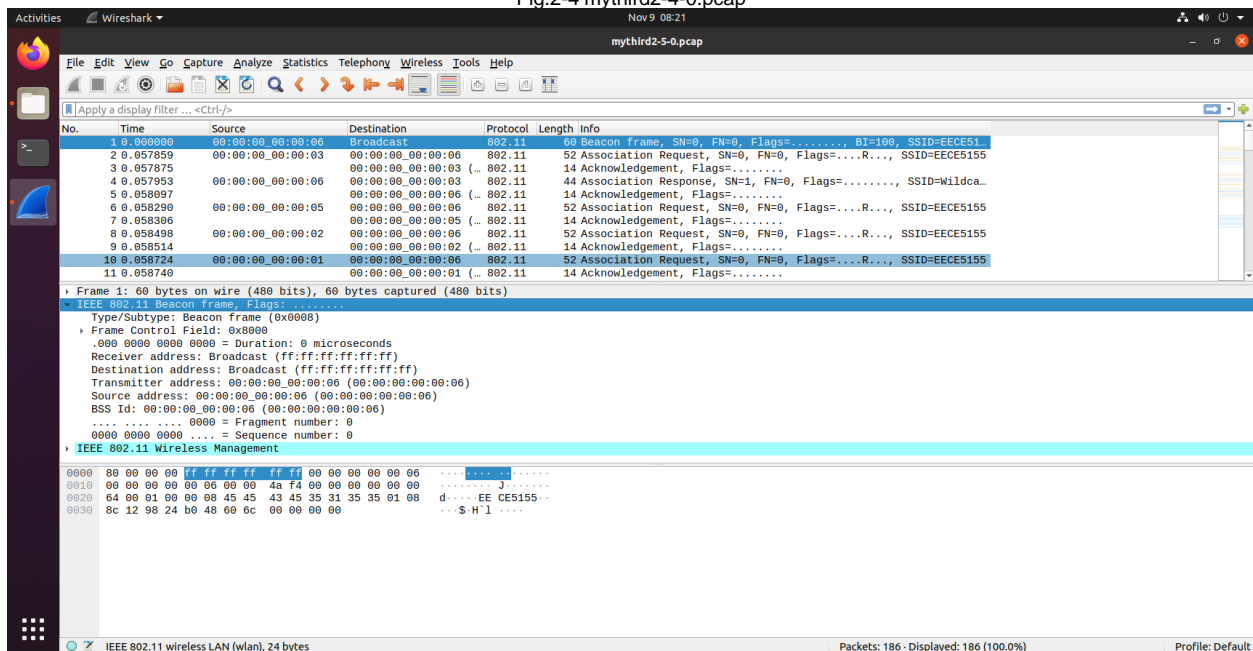


Fig.2-5 mythird2-5-0.pcap

3. Answering Questions:

Q1: Explain the behavior of the APs. What is happening since the very first moment the network starts operating?

A: From Fig.2-4 and Fig.2-5, we can see the AP (00:00:00:00:00:06) sends 60 bytes Beacon frame broadcast. The behavior is to find which nodes are connected to this AP. When these nodes receive the Beacon frames, they will send Association Request to the AP.

Q2: Take a look to a beacon frame. Which are the most relevant parameters defined in it?

A: It mainly contains fixed parameters and tagged parameters. In fixed parameters there is timestamp, beacon interval and capabilities information. In tagged information there is SSID parameter set and supported rates.

Q3: Are there any collisions in the network? When are these collisions happening?

A: There is one collision in the network. From Fig.2-1 we can see client1 (Node3) and client2 (Node4) will send a message to the server (Node0) at the same time (time 4s). Because the two clients have already set the channel with the server, the propagation time of these two messages is almost the same. When all two messages reach the server, the collision will happen. It is called Hidden Terminal Problem.

Q4: As in Task 1, force the utilization of the handshaking process and repeat the simulation. Are there any collisions now? Explain why.

A: There is no collision. Because when one node gets the CTS frame and prepare to send a message, another node will get Network Allocation Vector information which tells this node how long the channel will be occupied. So another node will not attempt to transmit during that time.

4. Learnt lessons:

In this task, we first simulate a WLAN with Infrastructure Mode. We use the Wireshark to monitor the network by tracing Node4 (one of clients) and the access point. And then we utilize the RTS/CTS procedure like Task1.

Task 3

In this last task, we are going to simulate a LoRaWAN network with 1 gateway and 6 end devices. Proceed as follows:

- Open the terminal and navigate to your ns-3.31 folder (e.g., cd source/ns-3.31 if you are using the class VM).
- Copy one of the LoRaWAN extension examples: cp src/lorawan/examples/parallel-reception-example.cc scratch/mylora.cc
- Run the mylora.cc without changing anything: ./waf --run=mylora

1. Experimental setup:

The network structure we are going to achieve is shown as the following:

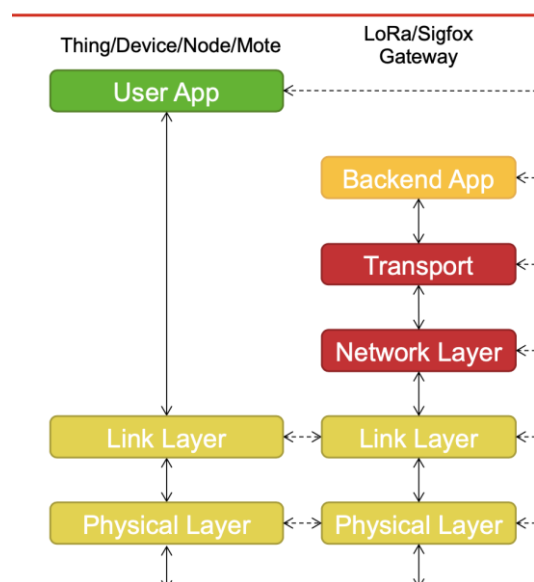


Fig. 3-1 structure of LoRaWAN

This part we just explain the generate structure of the LoraWan network.

```
#include "ns3/end-device-lora-phy.h"
#include "ns3/gateway-lora-phy.h"
#include "ns3/end-device-lorawan-mac.h"
#include "ns3/gateway-lorawan-mac.h"
#include "ns3/lorawan-mac-helper.h"
#include "ns3/simulator.h"
#include "ns3/log.h"
#include "ns3/constant-position-mobility-model.h"
#include "ns3/lora-helper.h"
#include "ns3/mobility-helper.h"
#include "ns3/node-container.h"
#include "ns3/position-allocator.h"
#include "ns3/one-shot-sender-helper.h"
#include "ns3/command-line.h"
#include <algorithm>
#include <ctime>

using namespace ns3;
using namespace lorawan;

NS_LOG_COMPONENT_DEFINE ("ParallelReceptionExample");

int
main (int argc, char *argv[])
{
    LogComponentEnable ("ParallelReceptionExample", LOG_LEVEL_ALL);
    LogComponentEnable ("GatewayLoraPhy", LOG_LEVEL_ALL);
    LogComponentEnable ("SimpleGatewayLoraPhy", LOG_LEVEL_ALL);
    LogComponentEnable ("GatewayLorawanMac", LOG_LEVEL_ALL);
    LogComponentEnableAll (LOG_PREFIX_FUNC);
    LogComponentEnableAll (LOG_PREFIX_NODE);
    LogComponentEnableAll (LOG_PREFIX_TIME);
```

First of all, there is one extra namespace called lorawan we will use for the part. That means we will use multiple namespace to avoid collision. Then we will build the logfile we will used here. Afterwards we will use "LogComponentEnable" method to execute the logfile or we can say use the instructions to make log-file in valid.

We are going to construct the channels with the following codes:

```
NS_LOG_INFO ("Creating the channel...");

Ptr<LogDistancePropagationLossModel> loss = CreateObject<LogDistancePropagationLossModel> ();
loss->SetPathLossExponent (3.76);
loss->SetReference (1, 7.7);

Ptr<PropagationDelayModel> delay = CreateObject<ConstantSpeedPropagationDelayModel> ();

Ptr<LoraChannel> channel = CreateObject<LoraChannel> (loss, delay);
```

Then we are going to create various of helpers. As you can see, there are LoraPhyHelper and LoraMAcHelper used, from here you can learnt that our next step is to use the object to build physical layer and link layer.

```

NS_LOG_INFO ("Setting up helpers...");

MobilityHelper mobility;
Ptr<ListPositionAllocator> allocator = CreateObject<ListPositionAllocator> ();
allocator->Add (Vector (0, 0, 0));
mobility.SetPositionAllocator (allocator);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");

// Create the LoraPhyHelper
LoraPhyHelper phyHelper = LoraPhyHelper ();
phyHelper.SetChannel (channel);

// Create the LorawanMacHelper
LorawanMacHelper macHelper = LorawanMacHelper ();

// Create the LoraHelper
LoraHelper helper = LoraHelper ();

```

Afterwards we are going to build network layer, which is known as end devices and gateway in the lo-raWAN network.

```

NS_LOG_INFO ("Creating the end device...");

NodeContainer endDevices;
endDevices.Create (6);

mobility.Install (endDevices);

phyHelper.SetDeviceType (LoraPhyHelper::ED);
macHelper.SetDeviceType (LorawanMacHelper::ED_A);
macHelper.SetRegion (LorawanMacHelper::SingleChannel);
helper.Install (phyHelper, macHelper, endDevices);

NS_LOG_INFO ("Creating the gateway...");
NodeContainer gateways;
gateways.Create (1);

mobility.Install (gateways);

phyHelper.SetDeviceType (LoraPhyHelper::GW);
macHelper.SetDeviceType (LorawanMacHelper::GW);
helper.Install (phyHelper, macHelper, gateways);

```

Then this part we are going to construct the application on the end devices:

```

OneShotSenderHelper oneShotSenderHelper;

oneShotSenderHelper.SetSendTime (Seconds (1));
oneShotSenderHelper.Install (endDevices);

```

Finally, we just set the transmission rate and simulate it, destroy it and clean it:

```

for (uint32_t i = 0; i < endDevices.GetN (); i++)
{
    endDevices.Get (i)

```

```

->GetDevice (0)
->GetObject<LoraNetDevice> ()
->GetMac ()
->GetObject<EndDeviceLorawanMac> ()
->SetDataRate (5-i);
}

Simulator::Stop (Hours (2));

Simulator::Run ();

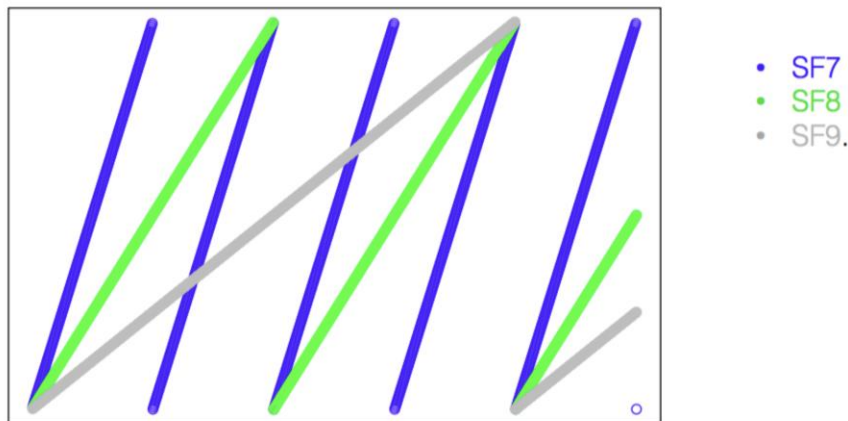
Simulator::Destroy ();

return 0;
}

```

And you can see that for each node, we get the physical layer and link layer it transmits package. And we set the datarate with the formula 5-1, just because of the following image:

Spreading Factor (SF)



The spreading determines how quickly the frequency is changed and, thus, the symbol time

Fig.3-2 Spreading Factor

Since different spreading factor has different speed of frequency changing, and like the slope of line shown in the image above. That means if we set the same spreading factor, then the system can't distinguish which package is sent by which node, that will cause collision.

2. Answering Questions

Q1: How many packets are sent in this network in total?

A: Six. Each one end device sends one packet to the gateway and there are six end devices, so six packets are sent in total.

Q2: How many simultaneous packets are sent in this network?

A: Six simultaneous packets. From Fig.3-3, we can see that all six packets begin to be sent at time 1s,


```

es  Terminal Nov 10 01:03
eece@ubuntu: ~/source/ns-3.31

+0.000000000s -1 GatewayLoraPhy:AddReceptionPath()
+0.000000000s -1 GatewayLoraPhy:ReceptionPath()
+0.000000000s -1 GatewayLoraPhy:AddReceptionPath()
+0.000000000s -1 GatewayLoraPhy:ReceptionPath()
+0.000000000s -1 GatewayLoraPhy:AddReceptionPath()
+0.000000000s -1 GatewayLoraPhy:ReceptionPath()
+0.000000000s -1 GatewayLoraPhy:AddReceptionPath()
+0.000000000s -1 GatewayLoraPhy:ReceptionPath()
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(0x55562f421960, 0x55562f38a380, 6.3, +51455999.0ns, 868.1)
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(): Scheduling reception of a packet, occupying one demodulator
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(0x55562f421960, 0x55562f470630, 6.3, +102911999.0ns, 868.1)
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(): Scheduling reception of a packet, occupying one demodulator
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(0x55562f421960, 0x55562f3b4ce0, 6.3, +185343999.0ns, 868.1)
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(): Scheduling reception of a packet, occupying one demodulator
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(0x55562f421960, 0x55562f46f830, 6.3, +329728000.0ns, 868.1)
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(): Scheduling reception of a packet, occupying one demodulator
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(0x55562f421960, 0x55562f4794c0, 6.3, +659456000.0ns, 868.1)
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(): Scheduling reception of a packet, occupying one demodulator
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(0x55562f421960, 0x55562f479990, 6.3, +1318912000.0ns, 868.1)
+1.000000000s 6 SimpleGatewayLoraPhy:StartReceive(): Scheduling reception of a packet, occupying one demodulator
+1.051455999s 6 SimpleGatewayLoraPhy:EndReceive(0x55562f421960, 0x55562f38a380, (1 s - 1.05146 s), SF7, 6.3 dBm, 868.1 MHz)
+1.051455999s 6 SimpleGatewayLoraPhy:EndReceive(): Packet with SF 7 received correctly
+1.051455999s 6 GatewayLorawanMac:Receive(0x55562f4141b0, 0x55562f38a380)
+1.051455999s 6 GatewayLorawanMac:Receive(): Received packet: 0x55562f38a380
+1.102911999s 6 SimpleGatewayLoraPhy:EndReceive(0x55562f421960, 0x55562f470630, (1 s - 1.10291 s), SF8, 6.3 dBm, 868.1 MHz)
+1.102911999s 6 SimpleGatewayLoraPhy:EndReceive(): Packet with SF 8 received correctly
+1.102911999s 6 GatewayLorawanMac:Receive(0x55562f4141b0, 0x55562f470630)
+1.102911999s 6 GatewayLorawanMac:Receive(): Received packet: 0x55562f470630
+1.185343999s 6 SimpleGatewayLoraPhy:EndReceive(0x55562f421960, 0x55562f3b4ce0, (1 s - 1.18534 s), SF9, 6.3 dBm, 868.1 MHz)
+1.185343999s 6 SimpleGatewayLoraPhy:EndReceive(): Packet with SF 9 received correctly
+1.185343999s 6 GatewayLorawanMac:Receive(0x55562f4141b0, 0x55562f3b4ce0)
+1.185343999s 6 GatewayLorawanMac:Receive(): Received packet: 0x55562f3b4ce0
+1.329728000s 6 SimpleGatewayLoraPhy:EndReceive(0x55562f421960, 0x55562f46f830, (1 s - 1.32973 s), SF10, 6.3 dBm, 868.1 MHz)
+1.329728000s 6 SimpleGatewayLoraPhy:EndReceive(): Packet with SF 10 received correctly
+1.329728000s 6 GatewayLorawanMac:Receive(0x55562f4141b0, 0x55562f46f830)
+1.329728000s 6 GatewayLorawanMac:Receive(): Received packet: 0x55562f46f830
+1.659456000s 6 SimpleGatewayLoraPhy:EndReceive(0x55562f421960, 0x55562f4794c0, (1 s - 1.65946 s), SF11, 6.3 dBm, 868.1 MHz)
+1.659456000s 6 SimpleGatewayLoraPhy:EndReceive(): Packet with SF 11 received correctly
+1.659456000s 6 GatewayLorawanMac:Receive(0x55562f4141b0, 0x55562f4794c0)
+1.659456000s 6 GatewayLorawanMac:Receive(): Received packet: 0x55562f4794c0
+2.318912000s 6 SimpleGatewayLoraPhy:EndReceive(0x55562f421960, 0x55562f479990, (1 s - 2.31891 s), SF12, 6.3 dBm, 868.1 MHz)
+2.318912000s 6 SimpleGatewayLoraPhy:EndReceive(): Packet with SF 12 received correctly
+2.318912000s 6 GatewayLorawanMac:Receive(0x55562f4141b0, 0x55562f479990)
+2.318912000s 6 GatewayLorawanMac:Receive(): Received packet: 0x55562f479990
eece@ubuntu:~/source/ns-3.31$

```

Fig.3-3 operation of mylora.cc

Q3: If more than one, are there any collisions?

If yes, explain how you have reached this conclusion.

If no, explain how this is possible.

A: From Fig.3-3, we can see every packet reach the destination correctly, so there is no collision. The reason is that different spreading factors can lead to different data rates, so each packet will not reach the destination at the same time for the different data rates. So there will be no collision.

Q4: What is the duration of each packet?

A: From Fig.3-3, we can see that the duration of the first packet is 0.05146s, the second is 0.10291s, the third is 0.18534s, the fourth is 0.32973s, the fifth is 0.65946s, the sixth is 1.31891s.

Q5: Edit your mylora.cc to force all the users to use the same spreading factor (e.g., SF7).

o What is now the result?

A: From Fig.3-4, we can see all the users use the same spreading factor (SF7). Now there are several collisions in the network, because all packets are destroyed. The duration of all packets is 0.05146s.

