

Réseaux Sans-Fil

Compte Rendu du TP2 - NS3

Nicolas MARRA - Benjamin METZGER

Introduction

Dans le cadre de ce TP, nous avons utilisé le simulateur de réseau ns-3 pour approfondir notre compréhension de la technologie LoRaWAN et Wi-Fi.

Dans le premier exercice, nous avons analysé l'impact de plusieurs paramètres de cette technologie, notamment le coding rate, le CRC et le mode de trafic, sur la consommation énergétique.

Dans le cadre de l'exercice 2, nous avons étudié la performance de la technologie LoRaWAN et Wi-Fi en fonction du nombre de nœuds. Notre objectif était d'évaluer l'impact du nombre de nœuds sur plusieurs métriques telles que le taux de succès des paquets (PDR), la latence moyenne et la consommation énergétique. Nous avons détaillé nos méthodologies de simulation (le script que l'on a développé ou amélioré) et analysé les résultats obtenus

Ce compte rendu expose en détail nos démarches pour faire les simulations et nos conclusions des résultats observés dans ces environnements de simulation.

Exercice 1 : LoRaWAN (suite TP1)

1. Expliquez l'architecture du réseau dans la simulation

L'architecture du réseau dans la simulation comprend deux composants principaux :

1. End-device (nœud terminal)

Dans cette simulation, un seul end-device est présent. Il utilise le protocole LoRaWAN pour transmettre des données vers la gateway.

2. Gateway (passerelle)

Dans cette simulation, une seule gateway est aussi installée, elle reçoit les données envoyées par l'end-device. Cette architecture est très simple, car aucun network server n'a été installé. Dans une application réelle et complexe, un network server aurait été installé et la gateway aurait alors pour rôle d'envoyer explicitement les données reçues par l'end-device au network server.

2. Utilisation de l'objet "PeriodicSenderHelper"

Pour mettre en place un trafic périodique en utilisant l'objet "PeriodicSenderHelper", on s'est inspirés du script dans `examples/complete-lorawan-network-example.cc` comme conseillé.

Pour ce faire, on a supprimé les lignes de code concernant le trafic en "OneShotSender".

Ensuite, on a ajouté les lignes de code suivantes :

```
#include "ns3/periodic-sender-helper.h"
```

```
int periodicSenderTime = 10;

LogComponentEnable("PeriodicSenderHelper", LOG_LEVEL_ALL);
LogComponentEnable("PeriodicSender", LOG_LEVEL_ALL);

PeriodicSenderHelper periodicSenderHelper;
periodicSenderHelper.SetPeriod(Seconds(periodicSenderTime));
periodicSenderHelper.SetPacketSize(20)
periodicSenderHelper.Install(endDevices);
```

Ces 6 dernières lignes ont été ajoutées dans la fonction `main()`.

On a activé les logs sur le 'PeriodicSender' pour vérifier le bon fonctionnement de notre implémentation.

3. Modification de la période d'envoi des paquets et ainsi que leur taille dynamiquement

Pour modifier dynamiquement la période d'envoi des paquets et leur taille, on a utilisé la ligne de commande de ns3.

Pour ce faire, on a ajoute les lignes de code suivantes:

```
int periodicSenderTime = 10;
int packet_size = 20;

CommandLine cmd;
cmd.AddValue("periodicSenderTime", "The period in seconds to be used by periodically transmitting by the end device", periodicSenderTime);
cmd.AddValue("packet_size", "Size of the packet sent by the end device", packet_size);
cmd.Parse(argc, argv);

PeriodicSenderHelper periodicSenderHelper;
periodicSenderHelper.SetPeriod(Seconds(periodicSenderTime));
periodicSenderHelper.SetPacketSize(packet_size);
periodicSenderHelper.Install(endDevices);
```

On a déclaré deux variables, `periodicSenderTime` et `packet_size`, pour stocker respectivement la période d'envoi des paquets et leur taille, avec des valeurs par défaut de 10 et 20. En utilisant la classe `CommandLine`, on a ajouté ces variables à la ligne de commande, alors l'utilisateur peut spécifier ces valeurs lors de l'exécution du programme.

Voici un exemple d'exécution du programme où l'utilisateur spécifie la taille des paquets et la période d'envoi dynamiquement : `./ns3 run simple-network-example.cc --periodicSenderTime=50 --packet_size=50`

Dans cet exemple, le programme est lancé avec une taille de paquet de 50 et une période d'envoi de 50 secondes. On peut vérifier le fonctionnement de notre implémentation à l'aide

des logs, le log indique que la taille réelle du paquet envoyé est de 59 octets (50 définis par l'utilisateur et 9 pour l'en-tête).

De plus, le log affiche aussi l'heure d'envoi des paquets. Dans cet exemple, un paquet a été envoyé à 7032.531182006 secondes, et le prochain envoi a eu lieu 50 secondes plus tard, soit à 7082.531182006 secondes.

```
+6982.531182006s 0 PeriodicSender:SendPacket(): Sent a packet of size 59
+7032.531182006s 0 PeriodicSender:SendPacket(0x5fead3330d10)
+7032.531182006s 0 PeriodicSender:SendPacket(): Sent a packet of size 59
+7082.531182006s 0 PeriodicSender:SendPacket(0x5fead3330d10)
+7082.531182006s 0 PeriodicSender:SendPacket(): Sent a packet of size 59
+7132.531182006s 0 PeriodicSender:SendPacket(0x5fead3330d10)
+7132.531182006s 0 PeriodicSender:SendPacket(): Sent a packet of size 59
+7182.531182006s 0 PeriodicSender:SendPacket(0x5fead3330d10)
+7182.531182006s 0 PeriodicSender:SendPacket(): Sent a packet of size 59
PeriodicSender::~PeriodicSender()
nicolasmarra@nicolasmarra-hp-15-da2xxx:~/Documents/School/2023_24/S2/Réseaux sans fil/TP/ns-allinone-3.41/ns-3.41$ ./ns3 run
simple-network-example.cc -- --periodicSenderTime=50 --packet_size=50
```

Fig.1 - Sortie du programme lancé avec une taille de paquet de 50 et une période d'envoi de 50.

4. Ajout un module de calcul de la consommation énergétique

Pour ajouter un module de calcul de la consommation énergétique au nœud (end-device), on s'est inspiré du script `examples/lorawan-energy-model-example.cc`, comme conseillé. On a ajouté les lignes de code suivantes dans la fonction `main()`:

```
LogComponentEnable("LoraRadioEnergyModel", LOG_LEVEL_ALL);

NetDeviceContainer endDevicesNetDevices = helper.Install(phyHelper, macHelper, endDevices);

BasicEnergySourceHelper basicSourceHelper;
LoraRadioEnergyModelHelper radioEnergyHelper;

basicSourceHelper.Set("BasicEnergySourceInitialEnergyJ", DoubleValue(10000)); // Energy in J
basicSourceHelper.Set("BasicEnergySupplyVoltageV", DoubleValue(3.3));

radioEnergyHelper.Set("StandbyCurrentA", DoubleValue(0.0014));
radioEnergyHelper.Set("TxCurrentA", DoubleValue(0.028));
radioEnergyHelper.Set("SleepCurrentA", DoubleValue(0.0000015));
radioEnergyHelper.Set("RxCurrentA", DoubleValue(0.0112));

radioEnergyHelper.SetTxCurrentModel("ns3::ConstantLoraTxCurrentModel",
                                     "TxCurrent", DoubleValue(0.028));

// install source on end devices' nodes
EnergySourceContainer sources = basicSourceHelper.Install(endDevices);
Names::Add("/Names/EnergySource", sources.Get(0));

// install device model
DeviceEnergyModelContainer deviceModels =
radioEnergyHelper.Install(endDevicesNetDevices, sources);

FileHelper fileHelper;
fileHelper.ConfigureFile("battery-level", FileAggregator::SPACE_SEPARATED);
fileHelper.WriteProbe("ns3::DoubleProbe", "/Names/EnergySource/RemainingEnergy", "Output");
```

On a activé le log sur `LoraEnergyModel` afin de vérifier le bon fonctionnement de notre implémentation, on pouvait aussi vérifier le fichier `battery-level`, car celui-ci a été configuré comme fichier de sortie et il nous permet donc de suivre l'évolution du niveau de batterie en fonction du temps.

```
+7184.582638016s 0 LoraRadioEnergyModel:SetLoraRadioState(): LoraRadioEnergyModel:Switching to state: STANDBY at time = 7184.58 s
+7184.582638016s 0 LoraRadioEnergyModel:ChangeState(): LoraRadioEnergyModel:Total energy consumption is 0.898627J
+7184.844782016s 0 LoraRadioEnergyModel:NotifySleep(0x55a8c81b96e0)
+7184.844782016s 0 LoraRadioEnergyModel:ChangeState(0x55a8c807bae0, 0)
+7184.844782016s 0 LoraRadioEnergyModel:DoGetCurrentTA(0x55a8c807bae0)
+7184.844782016s 0 LoraRadioEnergyModel:HandleEnergyChanged(0x55a8c807bae0)
+7184.844782016s 0 LoraRadioEnergyModel:HandleEnergyChanged(): LoraRadioEnergyModel:Energy changed!
+7184.844782016s 0 LoraRadioEnergyModel:SetLoraRadioState(0x55a8c807bae0, 0)
+7184.844782016s 0 LoraRadioEnergyModel:SetLoraRadioState(): LoraRadioEnergyModel:Switching to state: SLEEP at time = 7184.84 s
+7184.844782016s 0 LoraRadioEnergyModel:ChangeState(): LoraRadioEnergyModel:Total energy consumption is 0.899838J
```

Fig.2 - Fichier de log .

Sur cet extrait de fichier de log, on peut voir la consommation énergétique en J et ainsi que les changements d'état entre le mode SLEEP et STANDBY.

5. Étudier et expliquer l'impact de la période d'envoi et de la taille des paquets sur la consommation énergétique.

Pour analyser l'impact de la période d'envoi et de la taille des paquets sur la consommation énergétique, on a fait plusieurs tests en variant ces paramètres. Dans les choix de nos valeurs de test, on a pris en considération les limitations imposées par le duty-cycle ou rapport cyclique de LoRaWAN, qui impose une limitation sur le temps pendant lequel un appareil peut transmettre des données sur un canal de communication. On a ainsi évité les périodes d'envoi extrêmement courtes.

1. Impact de la période d'envoi sur la consommation énergétique

On a choisi une série de valeurs pour la période d'envoi, tout en commençant par des valeurs relativement courtes de 10 secondes et augmentant jusqu'à des valeurs plus longues, atteignant jusqu'à 5 minutes. On a utilisé une taille de paquets fixe de 20 octets.

Taille de paquet (octets)	Période d'envoi (s)	Consommation énergétique finale (J)
20	10	5.37881
20	30	1.81666
20	60	0.926078
20	180	0.332184
20	300	0.213239

Fig.3 - Impact de la période d'envoi sur la consommation énergétique (Tableau).

Impact de la période d'envoi sur la consommation énergétique - taille des paquets 20 octets

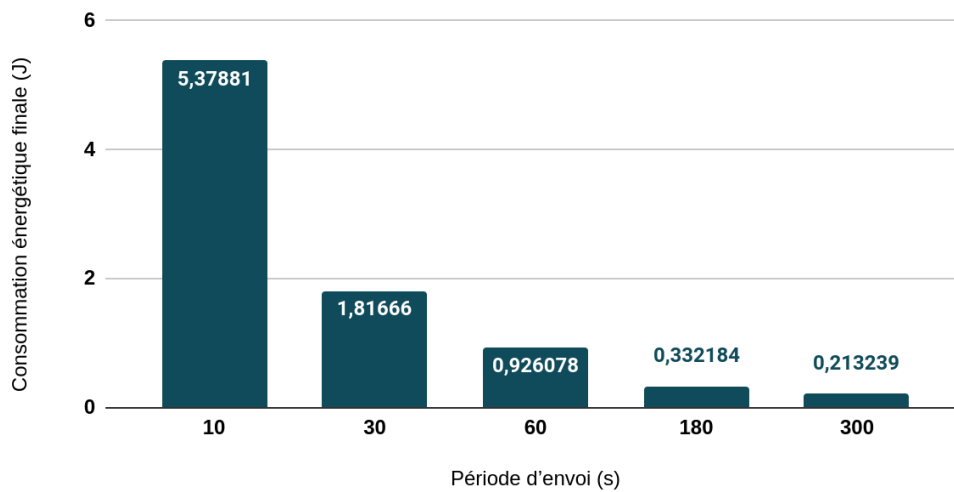


Fig.4 - Impact de la période d'envoi sur la consommation énergétique (Graphique) - Taille des paquets fixe de 20 octets

Le tableau et le graphique ci-dessus illustrent l'impact de la période d'envoi sur la consommation énergétique, on a mis sur le tableau et le graphique la valeur de la consommation énergétique finale en J au bout de 2h de simulation.

Nos résultats indiquent une tendance claire, on observe une diminution significative de la consommation d'énergie avec l'augmentation de la période d'envoi. Cette observation est cohérente, car une période d'envoi plus longue réduit la fréquence de transmissions de données et consomme donc moins d'énergie, alors qu'une période d'envoi plus courte augmente la fréquence des transmissions et entraîne une surutilisation de l'énergie.

Afin de mieux visualiser, nous avons aussi construit des graphiques en python de l'évolution de l'énergie en fonction du temps en fixant la taille des paquets à 20 octets ici et en faisant varier la période en secondes (10 - 60 - 300), voici le résultat graphique avec en bleu 10, orange 60 et vert 300:

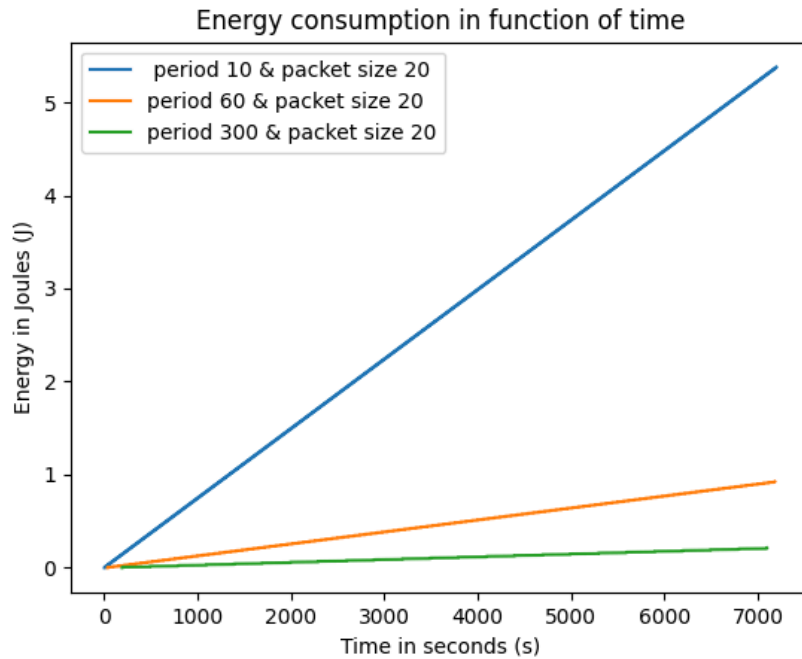


Fig.5 - Énergie consommée en fonction du temps (graphique).

Nous observons et confirmons avec ce graphique l'influence de la période sur l'énergie consommée. En effet, plus la période est basse plus le nombre de paquets envoyés sera important et l'énergie consommée aussi.

2. Impact de la taille des paquets sur la consommation énergétique

On a choisi une période d'envoi de 60 secondes et on a testé différentes tailles de paquets. D'ailleurs, dans le choix de nos valeurs de test pour la taille de paquet, on a respecté les limitations imposées par LoRaWAN, en ne dépassant pas 256 octets.

On a commencé avec des tailles de paquets de 10 octets et on a augmenté jusqu'à des valeurs plus élevées (120 octets).

Période d'envoi (s)	Taille des paquets (octets)	Consommation énergétique finale (J)
60	10	0.755775
60	20	0.926078
60	30	1.09638
60	60	1.60729
60	120	2.57234

Fig.6 - Impact de la taille des paquets sur la consommation énergétique (Tableau) - Période d'envoi fixe de 60 secondes.

Impact de la taille des paquets sur la consommation énergétique

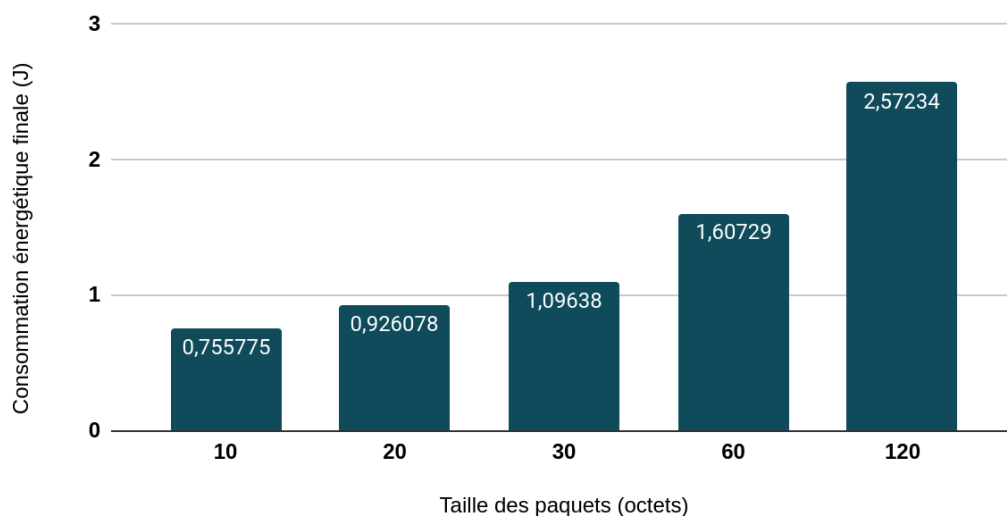


Fig.7 - Impact de la taille des paquets sur la consommation énergétique (Graphique) - Période d'envoi fixe de 60 secondes.

Ce tableau et ce graphique illustrent l'impact de la taille des paquets sur la consommation énergétique, on a inclus sur le tableau et ainsi que sur le graphique la valeur de la consommation énergétique finale en J au bout de 2h de simulation.

Ces résultats montrent une augmentation significative de la consommation énergétique avec l'augmentation de la taille des paquets. Nos résultats sont cohérents, car des paquets plus petits nécessitent moins d'énergie pour être envoyés et reçus sur LoRaWAN.

Afin de mieux visualiser, nous avons aussi construit des graphiques en python de l'évolution de l'énergie en fonction du temps en fixant la période à 60s ici et en faisant varier la taille des paquets (10 - 30 - 120), voici le résultat graphique avec en bleu 10, orange 30 et vert 120:

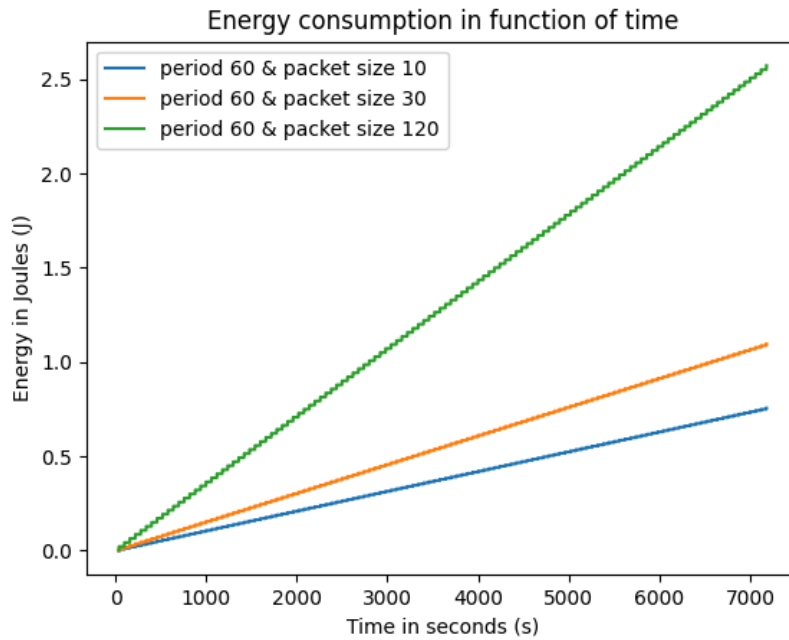


Fig.8 - Énergie consommée en fonction du temps (graphique).

Encore une fois, on peut observer que l'augmentation de la taille des paquets influe sur l'augmentation de la consommation énergétique.

6. Modification dynamique des valeurs attribuées aux paramètres de la technologie LoRa

Afin de modifier dynamique les valeurs attribuées aux paramètres de la technologie LoRa, on s'est intéressés à la méthode `SetSpreadingFactorUp(NodeContainer endDevices, NodeContainer gateways, Ptr<LoraChannel> channel)` de la classe `LoraWanMacHelper`.

```
sfQuantity = LorawanMacHelper::SetSpreadingFactorsUp( endDevices, gateways, channel);
```

En effet, à l'intérieur de cette méthode se trouve l'objet `mac` qui est une instance de la classe `ClassAEndDeviceLorawanMac`.

```
Ptr<ClassAEndDeviceLorawanMac> mac
=loranetDevice->GetMac()->GetObject<ClassAEndDeviceLorawanMac>();
```

On a remarqué que le coding rate et ainsi que le CRC peuvent être modifiés depuis la classe `ClassAEndDeviceLorawanMac`, et ils sont en fait modifiés depuis la méthode `SendToPhy(Ptr<Packet> packetToSend)` de cette classe.

```
void ClassAEndDeviceLorawanMac::SendToPhy(Ptr<Packet> packetToSend) {
    // ...
    LoraTxParameters params;
    params.codingRate = m_codingRate;
    params.crcEnabled = true;
    // ...
}
```

On a donc ajouté deux setters pour permettre la modification dynamiquement du coding rate et du CRC. Avant cela, on a mis la variable params comme un attribut public de la classe `ClassAEndDeviceLoraWanMac` afin que le params puisse être modifié depuis nos setters.

```
class ClassAEndDeviceLorawanMac : public EndDeviceLorawanMac
{
public:

    LoraTxParameters params;
```

- Dans le fichier .h de la classe en question :

```
void SetCodingRate(uint8_t codingRate);

void SetCRC(bool crcValue);
```

- Dans le fichier .cc de la classe en question :

```
void ClassAEndDeviceLorawanMac::SetCodingRate(uint8_t codingRate) {
    params.codingRate = codingRate;
}

void ClassAEndDeviceLorawanMac::SetCRC(bool crcValue) {
    params.crcEnabled = crcValue;
}
```

Étant donné que la méthode `SendToPhy(Ptr<Packet> packetToSend)` modifie aussi les paramètres coding rate et CRC, on a donc enlevé les lignes de code correspondant au changement de ces paramètres dans cette méthode, afin que ces paramètres ne soient modifiés que depuis les setters que l'on vient d'ajouter.

Une fois toutes les modifications faites dans la classe `ClassAEndDeviceLoraWanMac`, on a apporté des modifications à la méthode `SetSpreadingFactorUp(NodeContainer endDevices, NodeContainer gateways, Ptr<LoraChannel> channel)` de la classe `LoraWanMacHelper`, à l'intérieur de cette méthode, on fait appel aux setters que l'on vient d'ajouter à la classe `ClassAEndDeviceLoraWanMac`.

```
mac->SetCodingRate(1);
mac->SetCRC(true);
mac->SetMType(LoraMacHeader::UNCONFIRME_DATA_UP);
```

Pour ce qui concerne le mode de trafic, il existait déjà une méthode (setter) `SetMType(LorawanMacHeader::MType mType)` permettant de changer le mode de trafic.

Avec toutes ces modifications, on était capable de modifier ces trois paramètres de la technologie LoRa, cependant cette façon n'était pas très dynamique pour nous, on a donc changé la méthode `SetSpreadingFactorUp(NodeContainer endDevices,`

NodeContainer gateways, Ptr<LoraChannel> channel) afin que ces trois paramètres puissent être des paramètres de cette méthode.

Ces modifications ont été faites dans les fichiers suivants :

- Dans le fichier .h de la classe LoraWanMacHelper :

```
static std::vector<int>
SetSpreadingFactorsUp(NodeContainer endDevices, NodeContainer gateways,
    Ptr<LoraChannel> channel, uint8_t codingRate = 1,
    bool crc = true, uint8_t mtype = 0);
```

- Dans le fichier .cc de la classe LoraWanMacHelper :

```
static std::vector<int>
SetSpreadingFactorsUp(NodeContainer endDevices, NodeContainer gateways,
    Ptr<LoraChannel> channel, uint8_t codingRate = 1,
    bool crc = true, uint8_t mtype = 0)
{
    //...

    Ptr<ClassAEndDeviceLorawanMac> mac
    = loraNetDevice->GetMac()->GetObject<ClassAEndDeviceLorawanMac>();
    NS_ASSERT(mac);

    LorawanMacHeader::MType final_mtype =
        mtype == 0 ? LorawanMacHeader::UNCONFIRMED_DATA_UP
        : LorawanMacHeader::CONFIRMED_DATA_UP;

    mac->SetCodingRate(codingRate);
    mac->SetCRC(crc);
    mac->SetMType(final_mtype);

    //...
}
```

Avec ces modifications, on peut utiliser la ligne des commandes de ns3 pour passer en arguments ces 3 paramètres et ensuite les modifier dynamiquement depuis notre script principal. Pour le mode de trafic vu qu'il est représenté par un enum, on peut passer en arguments un entier soit 0 (UNCONFIRMED_DATA_UP) , soit 1 (CONFIRMED_DATA_UP).

Pour ce faire, on a ajouté les lignes de code suivantes :

D'abord, on a ajouté ces variables globales dans notre script principal :

```
uint8_t coding_rate = 1;
bool crc = true;
uint8_t mtype = 0;
```

Juste après, dans la fonction `main()` on a ajouté ces variables comme des valeurs qui peuvent être passé en argument de la ligne de commande :

```
cmd.AddValue("coding_rate", "Coding rate used by the end device",
coding_rate);
cmd.AddValue("crc", "CRC value used by the end device", crc);
cmd.AddValue("mtype",
"MType used by the end device : UNCONFIRMED_DATA_UP = 0 ou "
"CONFIRMED_DATA_UP = 1",
mtype);
```

Et pour finir, on a passé ces paramètres lors de l'appel de la méthode `SetSpreadingFactorsUp(NodeContainer endDevices, NodeContainer gateways, Ptr<LoraChannel> channel, uint8_t codingRate = 1, bool crc = true, uint8_t mtype = 0)` depuis la fonction `main()` de notre script principal:

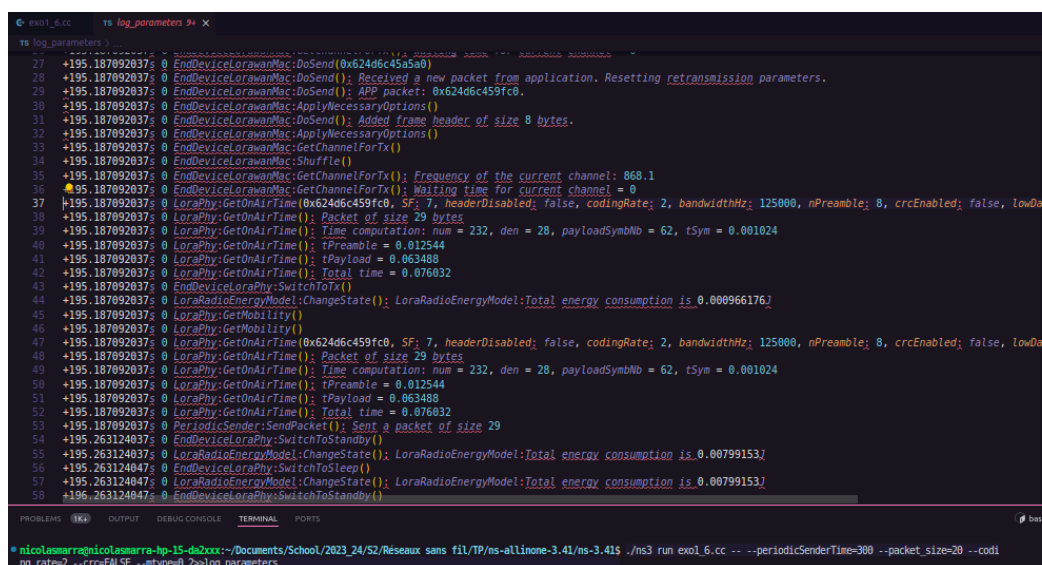
```
sfQuantity = LorawanMacHelper::SetSpreadingFactorsUp( endDevices, gateways, channel, coding_rate, crc,
mtype);
```

Pour vérifier le bon fonctionnement de notre implémentation, on a d'abord ajouté les logs suivants :

- `LogComponentEnable("LoraPhy", LOG_LEVEL_ALL);`
- `LogComponentEnable("EndDeviceLoraMac", LOG_LEVEL_ALL);`

Ensuite on a lancé le programme avec les arguments suivants :

```
./ns3 run exo1_6.cc -- --periodicSenderTime=300 --packet_size=20 --coding_rate=2
--crc=FALSE --mtype=0 2>>log_parameters
```



```
ts log_parameters
195.187892037s EndDeviceLoraMac::DoSend(0x624d6c45a5a0)
195.187892037s EndDeviceLoraMac::DoSend(): Received a new packet from application. Resetting retransmission parameters.
195.187892037s EndDeviceLoraMac::DoSend(): APP packet: 0x624d6c459fc0.
195.187892037s EndDeviceLoraMac::ApplyNecessaryOptions()
195.187892037s EndDeviceLoraMac::DoSend(): Added frame header of size 8 bytes.
195.187892037s EndDeviceLoraMac::ApplyNecessaryOptions()
195.187892037s EndDeviceLoraMac::GetChannelForTx()
195.187892037s EndDeviceLoraMac::Shuffle()
195.187892037s EndDeviceLoraMac::GetChannelForTx(): Frequency of the current channel: 868.1
195.187892037s EndDeviceLoraMac::GetChannelForTx(): Waiting time for current channel = 0
195.187892037s LoraPhy::GetOnAirTime(0x624d6c459fc0, SF: 7, headerDisabled: false, codingRate: 2, bandwidthHz: 125000, nPreamble: 8, crcEnabled: false, lowDataRate: false)
195.187892037s LoraPhy::GetOnAirTime(): Packet of size 29 bytes
195.187892037s LoraPhy::GetOnAirTime(): Time computation: num = 232, den = 28, payloadSymbNb = 62, tSym = 0.001024
195.187892037s LoraPhy::GetOnAirTime(): tPreamble = 0.012544
195.187892037s LoraPhy::GetOnAirTime(): tPayload = 0.063488
195.187892037s LoraPhy::GetOnAirTime(): Total time = 0.076032
195.187892037s EndDeviceLoraPhy::SwitchToTx()
195.187892037s LoraRadioEnergyModel::ChangeState(): LoraRadioEnergyModel: Total energy consumption is 0.000966176J
195.187892037s LoraPhy::GetMobility()
195.187892037s LoraPhy::GetMobility()
195.187892037s LoraPhy::GetOnAirTime(0x624d6c459fc0, SF: 7, headerDisabled: false, codingRate: 2, bandwidthHz: 125000, nPreamble: 8, crcEnabled: false, lowDataRate: false)
195.187892037s LoraPhy::GetOnAirTime(): Packet of size 29 bytes
195.187892037s LoraPhy::GetOnAirTime(): Time computation: num = 232, den = 28, payloadSymbNb = 62, tSym = 0.001024
195.187892037s LoraPhy::GetOnAirTime(): tPreamble = 0.012544
195.187892037s LoraPhy::GetOnAirTime(): tPayload = 0.063488
195.187892037s LoraPhy::GetOnAirTime(): Total time = 0.076032
195.187892037s PeriodicSender::SendPacket(): Sent a packet of size 29
195.263124037s EndDeviceLoraPhy::SwitchToStandby()
195.263124037s LoraRadioEnergyModel::ChangeState(): LoraRadioEnergyModel: Total energy consumption is 0.00799153J
195.263124047s EndDeviceLoraPhy::SwitchToSleep()
195.263124047s LoraRadioEnergyModel::ChangeState(): LoraRadioEnergyModel: Total energy consumption is 0.00799153J
195.263124047s EndDeviceLoraPhy::SwitchToStandby()
```

Fig. 8 - Fichier de logs contenant les paramètres de la technologie LoRa.

On peut voir sur la ligne 37 que le CRC est désactivé et que le coding rate vaut 2, ce sont les valeurs que l'on a passées en arguments de la ligne de commande.
Et pour vérifier le mode de trafic, on peut voir la ligne 12 de la figure ci-dessous.

```

1 +0.000000000s -1 LoraPhy::SetChannel(0x624d6c305c20, 0x624d6c43d0b0)
2 +0.000000000s -1 LoraPhy::SetDevice(0x624d6c305c20, 0x624d6c45a300)
3 +0.000000000s -1 EndDeviceLorawanMac::EndDeviceLorawanMac(0x624d6c45a300)
4 +0.000000000s -1 LoraPhy::SetChannel(0x624d6c340e40, 0x624d6c43d0b0)
5 +0.000000000s -1 LoraPhy::SetDevice(0x624d6c340e40, 0x624d6c45b380)
6 +0.000000000s -1 PeriodicSenderHelper::InstallPriv(0x7fff72040c00, 0x624d6c459370)
7 +0.000000000s -1 PeriodicSender::PeriodicSender()
8 +0.000000000s -1 PeriodicSender::SetInterval(0x624d6c308d10, +0ns)
9 +0.000000000s -1 PeriodicSender::SetInterval(0x624d6c308d10, +3e+11ns)
10 +0.000000000s -1 PeriodicSenderHelper::InstallPriv(): Created an application with interval = 0.0833333 hours
11 +0.000000000s -1 PeriodicSender::SetInitialDelay(0x624d6c308d10, +195187e+11ns)
12 +0.000000000s -1 EndDeviceLorawanMac::SetMyMac(); Message type is set to 2
13 +0.000000000s -1 EndDeviceLorawanMac::SetDataRate(0x624d6c45a300, 5)
14 +0.000000000s 0 PeriodicSender::StartApplication(0x624d6c308d10)
15 +0.000000000s 0 PeriodicSender::StartApplication(): Starting up application with a first event with a 195.187 seconds delay
16 +0.000000000s 0 PeriodicSender::StartApplication(): Event Id: 15
17 +195.187092037s 0 PeriodicSender::SendPacket(0x624d6c308d10)
18 +195.187092037s 0 EndDeviceLorawanMac::Send(0x624d6c45a300, 0x624d6c459fc0)
19 +195.187092037s 0 EndDeviceLorawanMac::GetNextTransmissionDelay()
20 +195.187092037s 0 EndDeviceLorawanMac::GetNextTransmissionDelay(): Waiting time before the next transmission in channel with frequency 868.1 is = 0.
21 +195.187092037s 0 EndDeviceLorawanMac::GetNextTransmissionDelay(): Waiting time before the next transmission in channel with frequency 868.3 is = 0.
22 +195.187092037s 0 EndDeviceLorawanMac::GetNextTransmissionDelay(): Waiting time before the next transmission in channel with frequency 868.5 is = 0.
23 +195.187092037s 0 EndDeviceLorawanMac::GetChannelForTx()
24 +195.187092037s 0 EndDeviceLorawanMac::Shuffle()
25 +195.187092037s 0 EndDeviceLorawanMac::GetChannelForTx(): Frequency of the current channel: 868.3
26 +195.187092037s 0 EndDeviceLorawanMac::GetChannelForTx(): Waiting time for current channel = 0
27 +195.187092037s 0 EndDeviceLorawanMac::DoSend(0x624d6c45a300)
28 +195.187092037s 0 EndDeviceLorawanMac::DoSend(): Received a new packet from application. Resetting retransmission parameters.
29 +195.187092037s 0 EndDeviceLorawanMac::DoSend(): APP packet: 0x624d6c459fc0.
30 +195.187092037s 0 EndDeviceLorawanMac::ApplyNecessaryOptions()
31 +195.187092037s 0 EndDeviceLorawanMac::DoSend(): Added frame header of size 8 bytes.

```

Fig.9 - Fichier de logs contenant les paramètres de la technologie LoRa

7. Attribuez différentes valeurs à chaque paramètre et étudiez l'impact sur la consommation énergétique.

Pour analyser l'impact de ces 3 paramètres de la technologie LoRa sur la consommation énergétique, on a conduit une série de tests, pour ce faire on a attribué différentes valeurs à chaque paramètre, ensuite on a analysé les résultats obtenus sur une période de simulation de 2h.

1. Impact du coding rate sur la consommation énergétique :

Afin d'étudier l'impact du coding rate sur la consommation énergétique, on a attribué 4 valeurs différentes au coding rate et on a maintenu le CRC et le mode de trafic constants. D'ailleurs, on a utilisé une période d'envoi de 60 secondes et une taille de paquet de 32 octets pour nos tests.

Résultats pour une période d'envoi de 600 secondes et une taille de paquet de 32 octets :

CRC	Mode de trafic	Coding rate	Consommation énergétique finale (J)
Activé	UNCONFIRMED	1 (4/5)	1.15315
Activé	UNCONFIRMED	2 (4/6)	1.30074
Activé	UNCONFIRMED	3 (4/7)	1.44834
Activé	UNCONFIRMED	4 (4/8)	1.59593

Fig.10 - Impact du coding rate sur la consommation énergétique (Tableau)

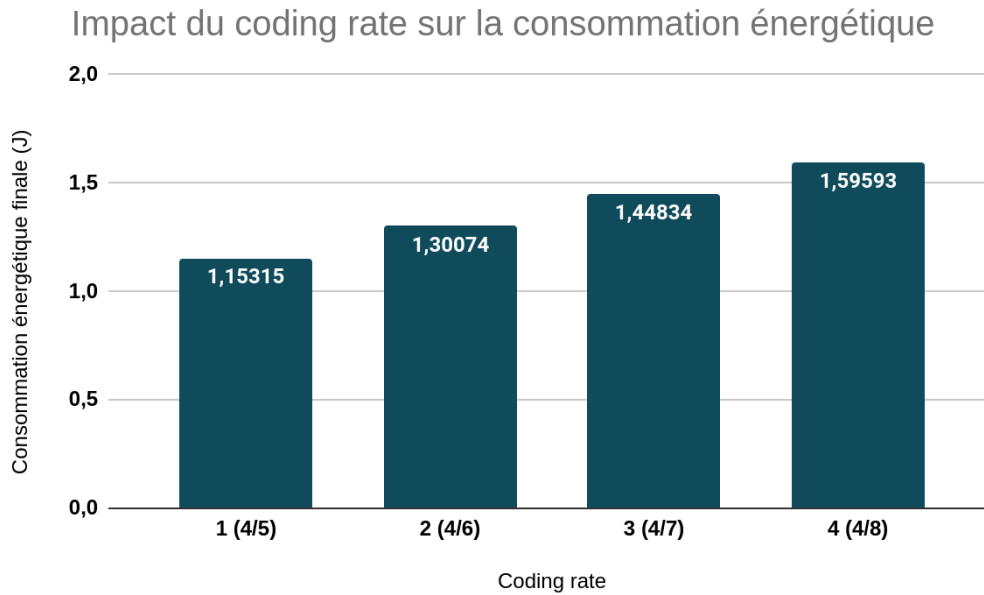


Fig.11 - Impact du coding rate sur la consommation énergétique (Graphique)

On observe une augmentation de la consommation énergétique lorsque le coding rate diminue ($1(4/5) > 2(4/6)$), cette observation est cohérente car un coding rate plus faible implique une redondance plus élevée de données et entraîne une transmission de données plus longue et par conséquent une consommation énergétique plus élevée.

Encore une fois, afin de vérifier cela, nous avons tracé l'évolution de la consommation d'énergie au cours du temps en fixant 2 paramètres, ici le CRC (à True) et le mode de trafic (à **UNCONFIRMED_DATA_UP**) afin de vérifier l'impact de l'augmentation du Coding rate :

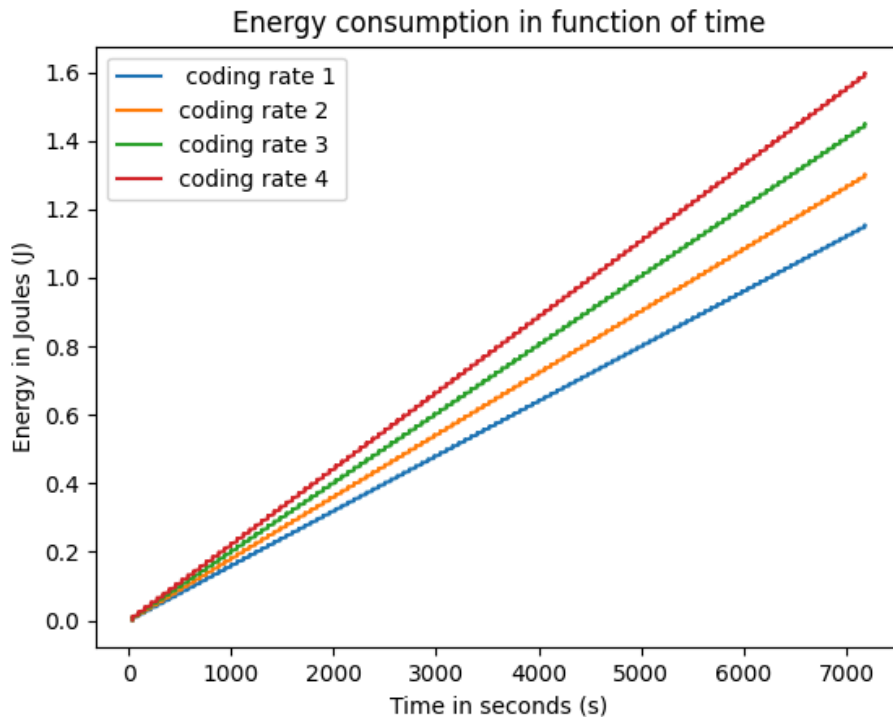


Fig.12 - Consommation énergétique en fonction du temps

Ce graphique permet de valider la conjecture précédente et de montrer que l'énergie consommée augmente parallèlement au Coding rate.

2. Impact du CRC sur la consommation énergétique :

Pour l'analyse de l'impact du CRC sur la consommation énergétique, on a utilisé les deux valeurs possibles pour le CRC (activé ou désactivé) et on a maintenu le coding rate constant et ainsi que le mode de trafic.

Résultats pour une période d'envoi de 60 secondes et une taille de paquet de 32 octets :

CRC	Mode de trafic	Coding rate	Consommation énergétique finale (J)
Activé	UNCONFIRMED	1 (4/5)	1.15315
Désactivé	UNCONFIRMED	1 (4/5)	1.09638

Fig.13 - Impact du CRC sur la consommation énergétique (Tableau)

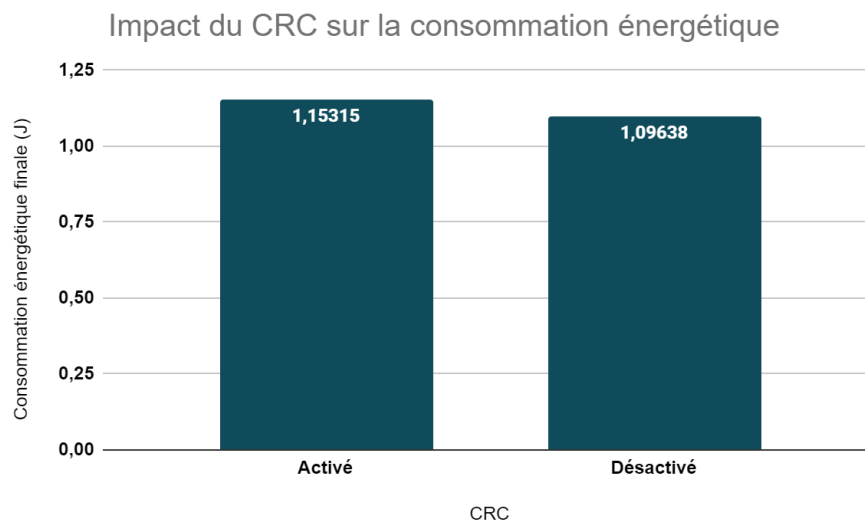


Fig.14 - Impact du CRC sur la consommation énergétique (Graphique)

On constate une légère augmentation de la consommation énergétique lorsque le CRC est activé, cette augmentation est logique car le mécanisme de détection d'erreur du CRC ajoute une surcharge de traitement afin de détecter l'erreur sur les données et entraîne une consommation énergétique supplémentaire.

Cette fois-ci, afin de vérifier cela, nous avons tracé l'évolution de la consommation d'énergie au cours du temps en fixant 2 paramètres, ici le Coding rate(à 1 puis 3) et le mode de trafic (à **UNCONFIRMED_DATA_UP**) afin de vérifier l'impact de la mise en place du CRC (Cyclic Redundancy Check) :

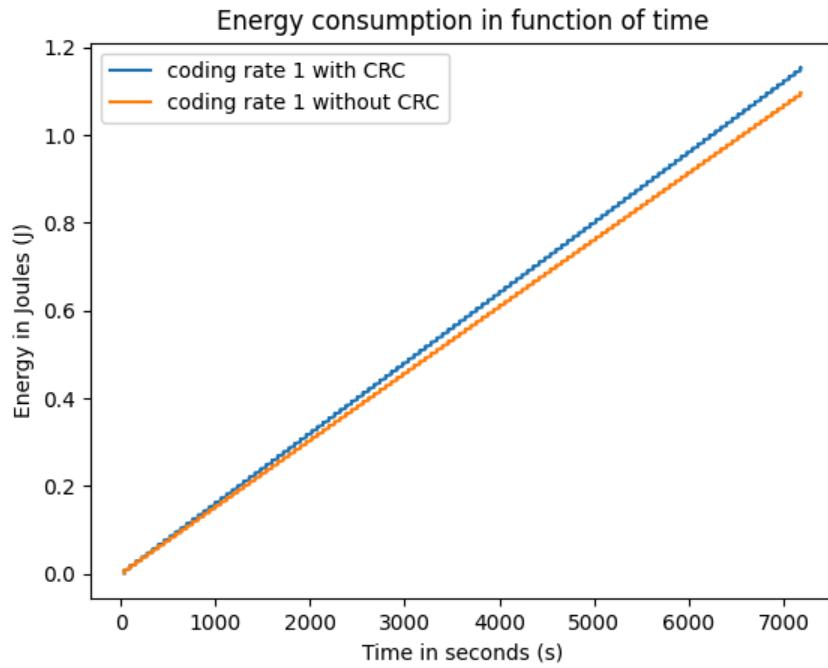


Fig.15 - Energie consommée en fonction du temps

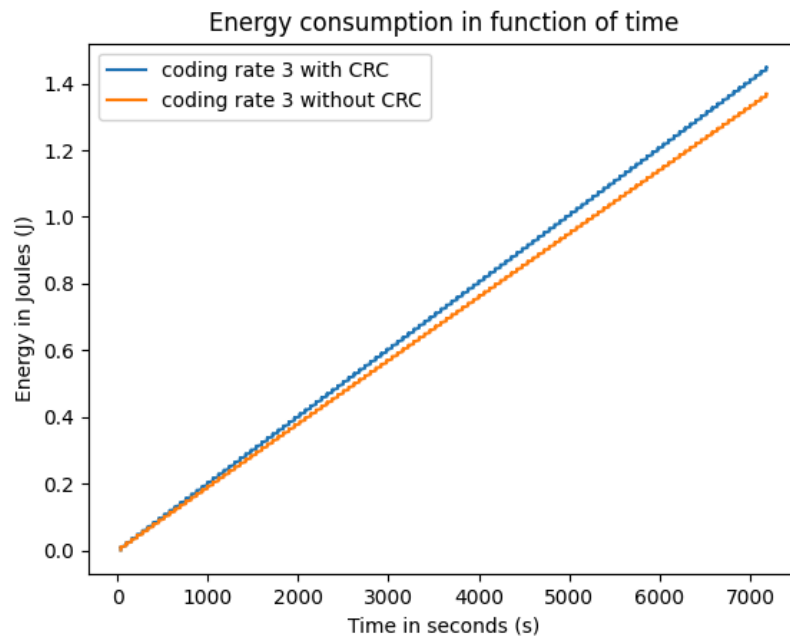


Fig.16 - Energie consommée en fonction du temps

On observe ici qu'en fixant ces deux paramètres et en comparant la consommation d'énergie, la consommation est plus importante quand le CRC est activé ce qui est explicable car le CRC est un mécanisme de détection d'erreurs donc son activation consomme plus d'énergie.

3. Impact du mode de trafic sur la consommation énergétique :

Pour analyser l'impact du mode de trafic sur la consommation énergétique, on a attribué deux valeurs "UNCONFIRMED" et "CONFIRMED" au mode de trafic, et on a maintenu le coding rate constant et ainsi que le CRC. On a dû modifier la valeur de période d'envoi car avec une valeur de période d'envoi petite et en utilisant le mode de trafic "CONFIRMED", cela entraîne facilement une erreur, en fait la tentative d'envoi du paquet est retardée en raison des contraintes du duty cycle et avec une période d'envoi de 60s et le mode de trafic CONFIRMED, on a eu ce problème avec une période d'envoi de 60s, c'est pour cette raison que l'on a augmenté la période d'envoi à 600s pour contourner les limitations du duty cycle.

Résultats pour une période d'envoi de 600 secondes et une taille de paquet de 32 octets :

CRC	Mode de trafic	Coding rate	Consommation énergétique finale (J)
Activé	UNCONFIRMED	1 (4/5)	0.191788
Activé	CONFIRMED	1 (4/5)	1.29245

Fig.17 - Impact du mode de trafic sur la consommation énergétique (Tableau)

Impact du mode de trafic sur la consommation énergétique

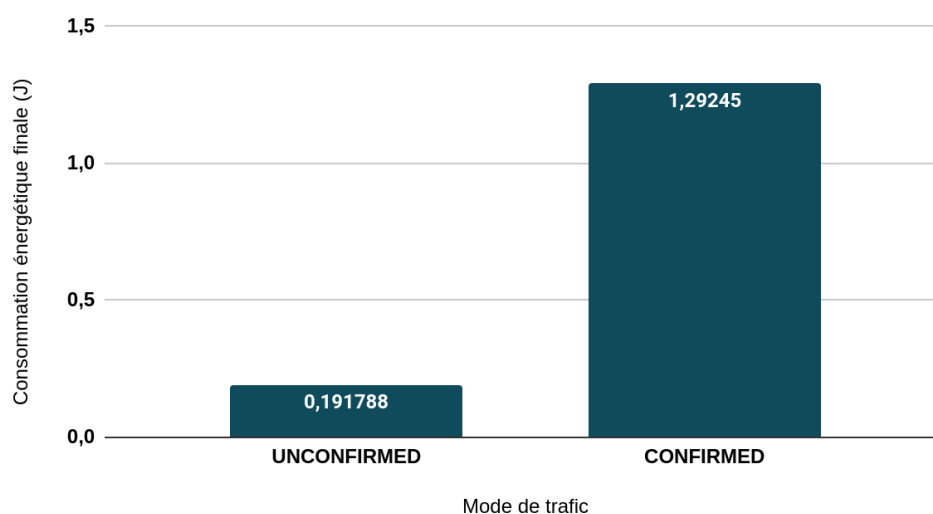


Fig.18 - Impact du mode de trafic sur la consommation énergétique (Tableau)

On observe une différence significative de consommation énergétique entre les modes de trafic "UNCONFIRMED" et "CONFIRMED", cette différence est tout à fait cohérente, car le mode de trafic "CONFIRMED" a besoin d'ajouter une transmission supplémentaire pour les accusés de réception et par conséquent ce mode de trafic consomme plus d'énergie. D'ailleurs, le mode de trafic "CONFIRMED" surcharge souvent le réseau et peut violer les réglementations du duty cycle, cette violation du duty qui implique une transmission retardée et une augmentation de la consommation d'énergie.

Exercice 2 : Évaluation de Performances - Wi-Fi + LoRaWAN

2.1 Wi-Fi

1. Déployez un réseau utilisant l'amendement 802.11ac avec les caractéristiques suivantes :

- Le nombre de noeuds doit varier entre 10 et 40,
- un seul point d'accès (AP),
- les noeuds doivent être positionnés aléatoirement dans un disque de rayon $R = 10m$,
- les noeuds envoient du trafic UDP à l'AP à un débit de 2 Mbps.

Pour faire cet exercice, on a utilisé le code `template.cc` du TP1.

On a utilisé la ligne des commandes ns3 afin de faire varier le nombre de noeuds dynamiquement, en ajoutant ces lignes de code :

```
uint32_t num_nodes = 10;
```

```
CommandLine cmd;  
cmd.AddValue("num_nodes", "Number of nodes", num_nodes);  
cmd.Parse(argc, argv);
```

Ensuite, on a créé n noeuds et un seul point d'accès, de la façon suivante :

```
NodeContainer ap;  
ap.Create(1);  
NodeContainer nodes;  
nodes.Create(num_nodes);
```

Pour positionner aléatoirement les noeuds dans un disque, on a utilisé l'objet `RandomDiscPositionAllocator`, de la façon suivante :

```
MobilityHelper mobility;  
mobility.SetPositionAllocator(  
    "ns3::RandomDiscPositionAllocator", "X", DoubleValue(0.0), "Y",  
    DoubleValue(0.0), "Rho",  
    StringValue("ns3::UniformRandomVariable[Min=0|Max=10]"));  
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");  
mobility.Install(nodes);  
mobility.Install(ap);
```

Pour que les noeuds envoient du trafic UDP à l'AP à un débit de 2Mbps, on a utilisé l'objet `packetSinkHelper` pour l'AP et l'objet `OnOffHelper` pour les noeuds, de la façon suivante :

```
uint16_t port = 50000;
```

```
Address sinkLocalAddress(  

```

```

InetSocketAddress(apInterfaces.GetAddress(0), port));

PacketSinkHelper sinkHelper("ns3::UdpSocketFactory", sinkLocalAddress);
ApplicationContainer sinkApp = sinkHelper.Install(ap.Get(0));

sinkApp.Start(Seconds(1.0));
sinkApp.Stop(Seconds(30.0));

OnOffHelper clientHelper("ns3::UdpSocketFactory", Address());
clientHelper.SetAttribute("Remote", AddressValue(InetSocketAddress(
    apInterfaces.GetAddress(0), port)));
clientHelper.SetConstantRate(DataRate("2Mb/s"), 1472);

ApplicationContainer clientApps = clientHelper.Install(nodes);
clientApps.Start(Seconds(2.0));
clientApps.Stop(Seconds(29.0));

```

2. Pour chaque nombre de nœuds, calculez :

Pour effectuer le calcul de ces métriques, on a utilisé des paquets de taille de 1472 octets. On a effectué les tests sur une simulation de 30.

- Le taux de succès des paquets (PDR) sur tout le réseau :

Pour calculer le PDR sur tout le réseau, on a utilisé l'objet `FlowMonitorHelper` qui permet de mesurer les performances du réseau, on a installé tous les nœuds (y compris l'AP) sur le flow monitor, de la façon suivante :

```

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

```

Ensuite, on a mis un compteur pour le nombre de paquets reçus et un autre pour le nombre de paquets envoyés et à l'aide du `FlowMonitor` on a pu parcourir chaque nœud (y compris l'AP) et avoir accès aux nombres de paquets reçus et envoyés.

```

Simulator::Stop(Seconds(30.0));
Simulator::Run();

monitor->CheckForLostPackets();
FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats();

uint32_t paquets_envoyes = 0;
uint32_t paquets_recus = 0;
double latence_moyenne = 0;
for (auto it = stats.begin(); it != stats.end(); ++it) {

    latence_moyenne +=
        it->second.delaySum.GetSeconds() / it->second.rxPackets;

    paquets_envoyes += it->second.txPackets;
}

```

```
    paquets_recus += it->second.rxPackets;
}
```

Après, on affiche le PDR en divisant le nombre de paquets reçus par le nombre de paquets envoyés :

```
std::cout << "Taux de succès des paquets (PDR): "
    << ((double)paquets_recus / (double)paquets_envoyes) * 100 << "%"
    << std::endl;
```

On a fait varier le nombre de nœuds pour calculer le PDR.

Nombre de nœuds	Paquets envoyés	Paquets reçus	PDR
10	49250	49246	99.9919%
15	73875	73849	99.9648%
20	98500	98286	99.7827%
25	123125	121359	98.5657%
30	147750	140154	94.8589%
35	172375	157484	91.3613%
40	197000	177247	89.9731%

Fig.19 - Le PDR par rapport au nombre de noeuds

- La latence moyenne des paquets reçus :

Pour calculer la latence moyenne des paquets reçus, on a aussi utilisé le **FlowMonitor**. Avec le **FlowMonitor**, on peut savoir le temps qu'un paquet a mis à être envoyé, après avec l'attribut **delaySum**, on peut avoir le délai total entre l'émission et la réception d'un paquet, on stocke cette valeur dans une variable et on divise aussi cette valeur par le nombre de paquets reçus. L'exemple de l'implémentation de cette approche est sur la section précédente.

On a fait varier le nombre de nœuds pour calculer la latence moyenne des paquets reçus.

Nombre de nœuds	Latence moyenne en s
10	0.012178
15	0.0265082
20	0.0475549

25	0.0930395
30	0.367377
35	1.0795
40	2.23915

Fig.20 - La latence moyenne en s des paquets reçus par rapport au nombre de noeuds

- La consommation énergétique moyenne des nœuds :

Pour calculer la consommation énergétique moyenne des nœuds, d'abord on a ajouté un modèle de calcul de consommation énergétique comme on l'a fait dans l'exo précédent.

Pour ce faire, on a utilisé l'objet `WifiRadioEnergyHelper` qui fonctionne comme l'objet `LoraRadioEnergyHelper`.

Ensuite, pour calculer l'énergie moyenne consommée, on a dû parcourir chaque device et pour obtenir sa consommation totale, et après cette consommation totale a été stockée dans une variable et après on l'a divisée par le nombre de nœuds.

```
double consommation_energetique = 0.0;
for (auto iter = deviceModels.Begin(); iter != deviceModels.End(); iter++) {
    consommation_energetique += (*iter)->GetTotalEnergyConsumption();
}

std::cout << "Consommation énergétique moyenne: "
    << consommation_energetique / num_nodes << std::endl;
```

On a fait varier le nombre de nœuds pour analyser son impact sur la consommation énergétique moyenne.

En fait, pour la consommation énergétique moyenne on a rencontré beaucoup de difficultés pour faire la simulation, par conséquent on a utilisé un modèle d'énergie basique et très faible pour tests concernant la consommation énergétique moyenne, avec une `BasicEnergySourceInitialEnergyJ` de 10000 J, notre programme n'arrive pas à finir la simulation pour un nombre de noeud supérieur à 15, alors on a réduit notre `BasicEnergySourceInitialEnergyJ` pour faire des tests sur la consommation énergétique, mais sur le script rendu la variable valeur sera celle que l'on mise auparavant. D'ailleurs, on a aussi changé notre modèle d'énergie afin de réaliser des tests sur la consommation énergétique moyenne, en conséquence nos valeurs pour la consommation énergétique moyenne seront très faibles, mais nous permettront de comprendre d'étudier l'impact du nombre de nœuds sur la consommation énergétique moyenne.

Par ailleurs, voici le modèle d'énergie que l'on a utilisé lors des simulations pour la consommation énergétique :

```
// configuration du model energy
```

```

/** Energy Model */
/*****
/* energy source */
BasicEnergySourceHelper basicSourceHelper;
// configure energy source
basicSourceHelper.Set("BasicEnergySourceInitialEnergyJ", DoubleValue(0.1));
// install source
EnergySourceContainer sources = basicSourceHelper.Install(nodes);
/* device energy model */
WifiRadioEnergyModelHelper radioEnergyHelper;
// configure radio energy model
radioEnergyHelper.Set("TxCurrentA", DoubleValue(0.0174));
// install device model
DeviceEnergyModelContainer deviceModels = radioEnergyHelper.Install(nodeDevices,
sources);
*****/

```

Voici donc nos valeurs pour la consommation énergétique moyenne pour une source d'énergie de 0.1J :

Nombre de nœuds	Consommation énergétique totale en J	Consommation énergétique moyenne en J
10	0.87698	0.087698
15	1.31433	0.0876217
20	1.75128	0.0875638
25	2.26909	0.0907636
30	2.72311	0.0907703
35	3.17554	0.0907296
40	3.63112	0.090778

Fig.21 - La consommation énergétique moyenne par rapport au nombre de noeuds

3. Générez une courbe d'évolution de chacun de ces paramètres, en commentant.

- Le taux de succès des paquets (PDR) sur tout le réseau:

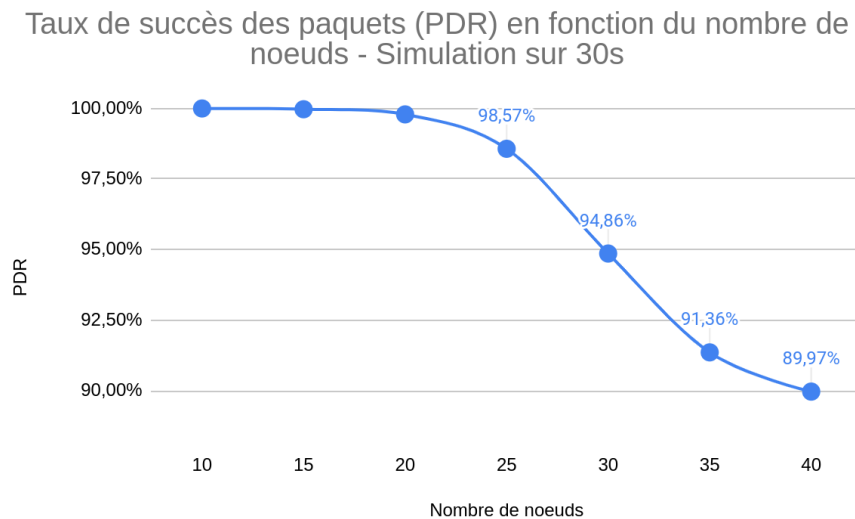


Fig.22 - La consommation énergétique moyenne des noeuds en J en fonction du nombre de noeuds

On constate que le taux de succès des paquets (PDR) diminue avec l'augmentation du nombre de nœuds, cette observation reste logique car une augmentation de nombre de nœuds entraîne une congestion du réseau et une augmentation des collisions.

- La latence moyenne des paquets reçus:

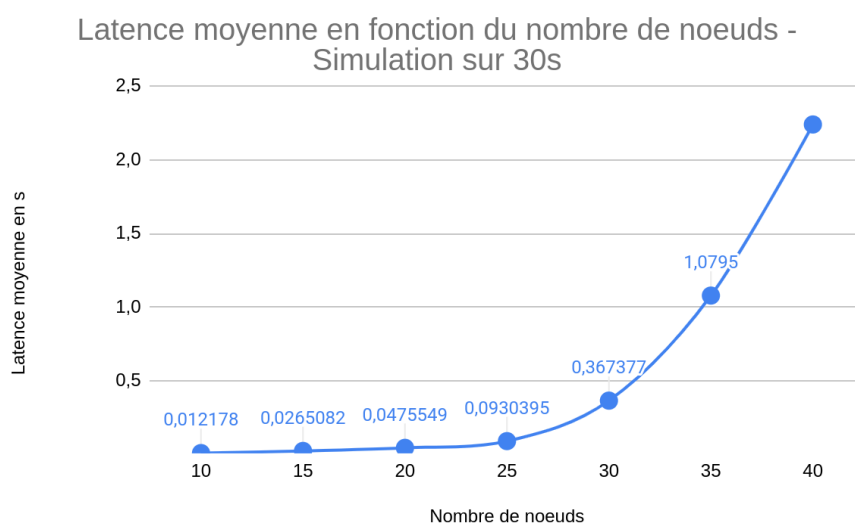


Fig.23 - La latence moyenne des paquets reçus en fonction du nombre de noeuds

Avec un nombre plus grand de nœuds, la latence moyenne tend à croître, cette observation est logique car l'augmentation du trafic sur le réseau peut avoir un impact sur les délais pour la transmission et la réception des paquets.

- La consommation énergétique moyenne des nœuds :

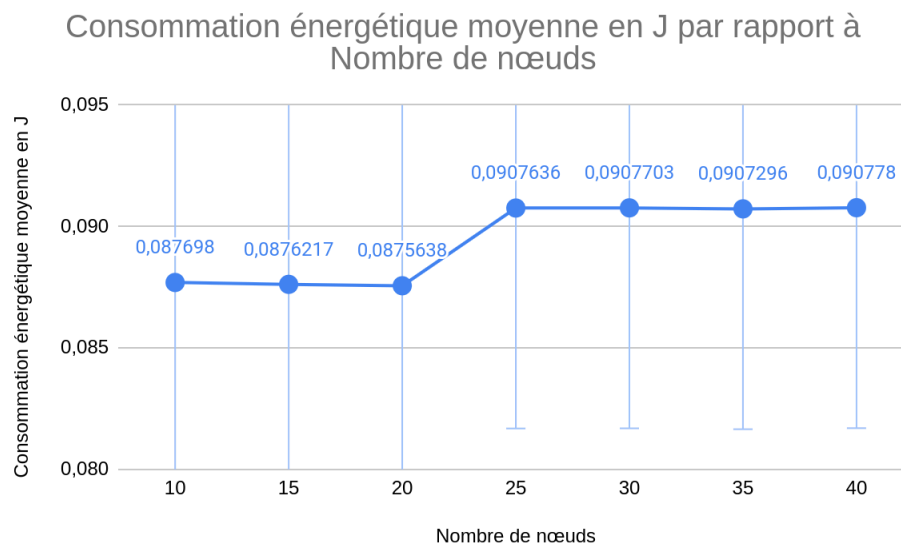


Fig.24 - La consommation énergétique moyenne des nœuds en J en fonction du nombre de nœuds

L'augmentation du nombre de nœuds entraîne aussi une augmentation énergétique totale du réseau, cependant la consommation énergétique moyenne par nœud reste un peu stable et avec un grand nombre de nœuds, elle ne varie presque pas. La topologie du réseau a aussi une petite influence sur la consommation énergétique moyenne des nœuds, car avec une répartition plus uniforme des nœuds peut permettre de réduire la consommation d'énergie par nœud, ainsi la variation non linéaire de la consommation énergétique peut être en raison de la variation de la densité de nœuds résultant de la topologie du réseau, en positionnant les nœuds aléatoirement.

2.2 LoRaWAN

1. Déployez un réseau utilisant LoRa avec les caractéristiques suivantes :

- Le nombre de nœuds doit varier entre 100 et 1000,
- une seule passerelle (gateway),
- les nœuds doivent être positionnés aléatoirement dans un disque de rayon $R = 1000\text{m}$,
- les nœuds envoient 1 paquet toutes les 10 minutes.

Pour faire cet exercice, on a utilisé le code de l'exercice précédent. Afin de faciliter le changement de nombre de nœuds, on a utilisé une variable `num_nodes` que l'on change dynamiquement à l'aide de la ligne de commande.

```
int num_nodes = 1000;
```



```
CommandLine cmd;
cmd.AddValue("num_nodes", "Number of nodes", num_nodes);
cmd.Parse(argc, argv);
```

Pour positionner aléatoirement les nœuds dans un disque de rayon R=1000 m, on a utilisé l'objet `RandomDiscPositionAllocator` de la façon suivante :

```
MobilityHelper mobility;

Ptr<RandomDiscPositionAllocator> allocator =
    CreateObject<RandomDiscPositionAllocator>();

Ptr<UniformRandomVariable> randomRadius =
    CreateObject<UniformRandomVariable>();

randomRadius->SetAttribute("Min", DoubleValue(0.0));
randomRadius->SetAttribute("Max", DoubleValue(1000.0));
allocator->SetX(0);
allocator->SetY(0);
allocator->SetRho(randomRadius);

mobility.SetPositionAllocator(allocator);
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
```

Pour que les nœuds envoient 1 paquet toutes les 10 minutes, on a utilisé une période d'envoi de 600s.

```
int periodicSenderTime = 600;
```

2. Pour chaque nombre de nœuds, calculez :

Pour effectuer le calcul de ces métriques, on a utilisé des paquets de taille de 20 octets. Initialement, on a effectué les tests sur une simulation de 2h, ensuite sur une simulation de 4h. (Les résultats présentés ici seront basés sur la simulation de 4h).

- Le taux de succès des paquets (PDR) sur tout le réseau :

Pour calculer le taux de succès des paquets (PDR) sur tout le réseau, on a activé les logs sur les end-device et sur la gateway :

```
LogComponentEnable("EndDeviceLorawanMac", LOG_LEVEL_ALL);
LogComponentEnable("GatewayLorawanMac", LOG_LEVEL_ALL);
```

Ces logs nous permettent de suivre à la fois les envois de paquets depuis les end-devices et les réceptions de paquets au niveau de la gateway. On a ensuite créé un script qui parse le fichier de log et compte le nombre de paquets envoyés en recherchant les occurrences de la méthode `DoSend()` dans les logs des end-devices, car c'est cette méthode qui est responsable de l'envoi effectif des paquets, si la vérification effectuée par la méthode `Send()` réussit, alors le paquet est envoyé avec la méthode `DoSend()`.

```
+77.016735454s 44 EndDeviceLorawanMac:DoSend(0x5981ddf65f10)
```

En ce qui concerne la réception des paquets, on s'intéresse aux logs de la gateway, comme on a activé les logs sur la gateway, alors on a enregistré aussi les logs de gateway dans un fichier de logs.

```
+77.083554277s 100 GatewayLorawanMac:Receive(): Received packet:
0x5981ddf6b1520
```

Notre script fait le parsing et utilise la formule du PDR pour son calcul.

Une fois que l'on a tout mis en place, on a fait varier le nombre de nœuds pour calculer le PDR.

Nombre de nœuds	Paquets envoyés	Paquets reçus	PDR
100	2400	2400	100.00%
200	4800	4792	99.83%
300	7200	7107	98.71%
400	9600	9437	98.30%
500	12000	11822	98.52%
600	14400	13983	97.10%
700	16800	16428	97.79%
800	19200	18440	96.04%
900	21600	20832	96.44%
1000	24000	23081	96.17%

Fig.24 - Le PDR par rapport au nombre de noeuds - Simulation sur 4h

- La latence moyenne des paquets reçus :

Pour calculer la latence moyenne des paquets reçus, on a aussi utilisé le script de parsing pour analyser les logs au niveau de la gateway et des end-devices. Cette fois-ci, on extrait le temps (horodatages) situés à gauche de chaque ligne de log à l'envoi et à la réception des paquets.

```
+77.016735454s 44 EndDeviceLorawanMac:DoSend(0x5981ddf65f10)
```

```
+77.083554277s 100 GatewayLorawanMac:Receive(): Received packet:
0x5981ddf6b1520
```

Par exemple, les valeurs à gauche représentent respectivement l'heure de l'envoi du paquet et de sa réception. En soustrayant l'heure de réception et l'heure de l'envoi des paquets, on

obtient le temps écoulé entre l'envoi et la réception de chaque paquet. Ensuite, on fait la somme de ces différences pour tous les paquets et on a divisé la somme totale par le nombre de paquets reçus pour obtenir la latence moyenne.

Nombre de nœuds	Latence moyenne en ms
100	70
200	70
300	70
400	70
500	70
600	60
700	60
800	60
900	60
1000	60

Fig.25 - La latence moyenne en s par rapport au nombre de noeuds - Simulation sur 4h

- La consommation énergétique moyenne des nœuds :

Pour calculer la consommation énergétique, on a essayé de suivre la même approche, celle d'analyser les logs, cette fois ci sur :

```
LogComponentEnable("LoraRadioEnergyModel", LOG_LEVEL_INFO);
```

Tout avait l'air de marcher, cependant une difficulté a été rencontrée, étant donné qu'il y a plusieurs nœuds et pas qu'un seul comme à l'exercice précédent, il y a donc la consommation énergétique de plusieurs nœuds, et ça a été un peu compliqué de trouver la consommation énergétique totale. On a donc décidé d'utiliser une autre méthode, on a utilisé la méthode `GetTotalEnergyConsumption()` de la classe `DeviceEnergyModel`

Ensuite, on a ajouté ces lignes à notre code pour calculer la consommation énergétique moyenne :

```
double consommation_energetique = 0.0;

for (auto iter = deviceModels.Begin(); iter != deviceModels.End(); iter++) {
    consommation_energetique += (*iter)->GetTotalEnergyConsumption();
}

std::cout << "Consommation énergétique totale: " << consommation_energetique
    << std::endl;
std::cout << "Consommation énergétique moyenne: "
```

```
<< consommation_energetique / num_nodes << std::endl;
```

Nombre de nœuds	Consommation énergétique totale en J	Consommation énergétique moyenne en J
100	24.7909	0.247909
200	49.5961	0.24798
300	74.4538	0.248179
400	99.1737	0.247934
500	123.958	0.247917
600	148.884	0.248139
700	173.701	0.248145
800	198.513	0.248141
900	223.24	0.248045
1000	248.278	0.248278

Fig.26 - La consommation énergétique moyenne des noeuds en J en fonction du nombre de noeuds - Simulation sur 4h

3. Générez une courbe d'évolution de chacun de ces paramètres, en commentant.

- Le taux de succès des paquets (PDR) sur tout le réseau:

Taux de succès des paquets (PDR) en fonction du nombre de nœuds - Simulation sur 4h

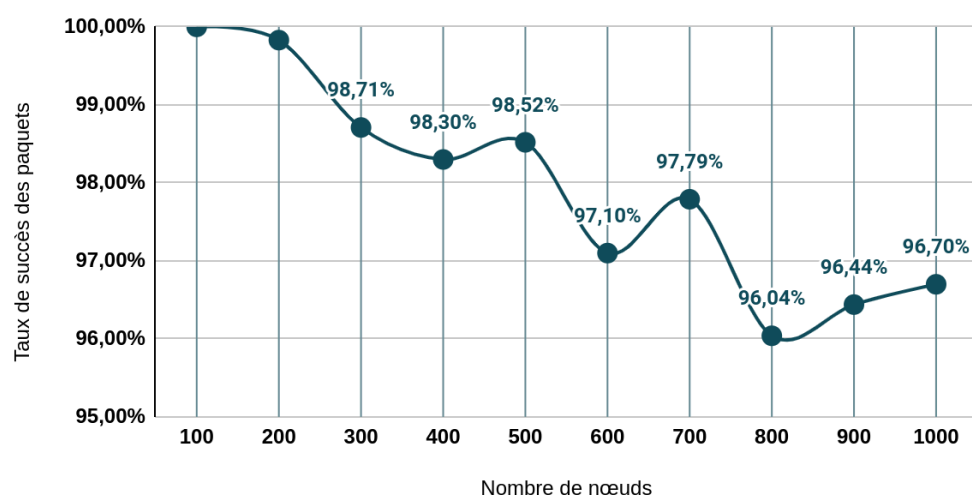


Fig.27 - Le PDR en fonction du nombre de noeuds - Simulation sur 4h

On observe que le taux de succès des paquets (PDR) diminue avec l'augmentation du nombre de nœuds. Ce comportement est cohérent car avec l'augmentation du nombre de nœuds, le nombre de paquets envoyés et reçus sur le réseau augmente aussi, par conséquent il peut y avoir une augmentation de la congestion du réseau, et aussi des collisions des paquets qui devient plus fréquentes à mesure que le nombre de nœuds augmente, tous ces facteurs contribuent à l'augmentation de la probabilité de défaillance lors de la transmission ou réception des paquets et en conséquence à la diminution du PDR.

Malgré la diminution du PDR, le réseau maintient une certaine stabilité dans ses performances, même avec un grand nombre de nœuds, le PDR reste élevé, au-dessus de 95%.

Par ailleurs, on constate que la diminution du PDR n'est pas toujours linéaire ou progressive. Par exemple, on peut observer une diminution du PDR pour un nombre de nœuds allant de 100 jusqu'à 600, puis il y a une légère augmentation pour 700 nœuds avant une nouvelle baisse pour 800, et ainsi de suite. Étant donné la topologie du réseau, où les nœuds sont positionnés aléatoirement dans un disque de 1000 m, on peut donc en déduire que cette topologie a une petite influence sur le PDR pour un nombre de nœuds plus grand.

- La latence moyenne des paquets reçus:

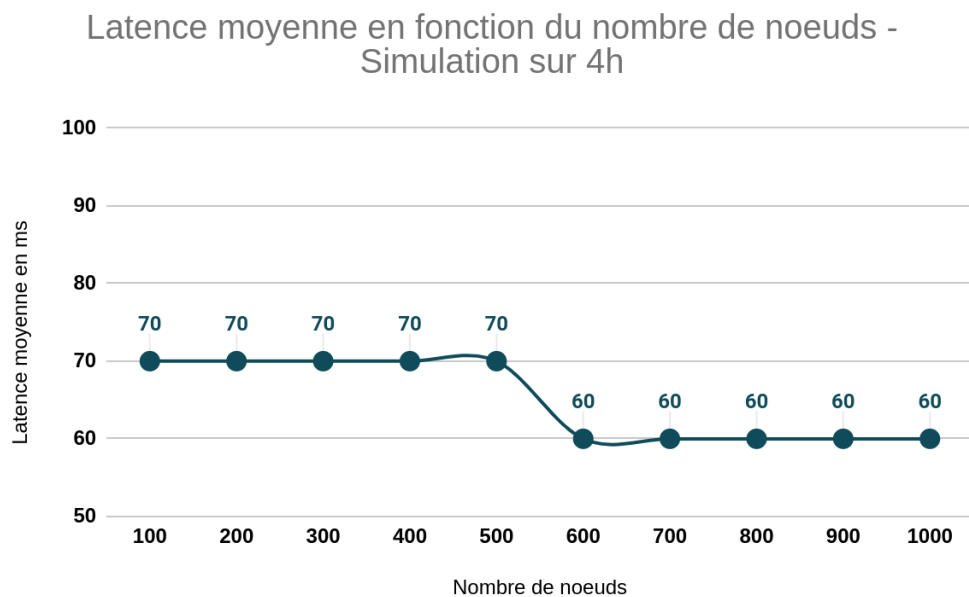


Fig.28 - La latence moyenne des paquets reçus en fonction du nombre de noeuds - Simulation sur 4h

La latence moyenne reste stable à environ 70 ms de 100 à 500 nœuds, puis diminue et reste stable à 60 ms 600 nœuds à 1000 nœuds. Cette stabilité montre que l'ajout de nœuds n'a pas d'impact significatif ou a un effet très limité sur la latence moyenne. D'ailleurs, en raison de cette stabilité, on peut donc en déduire que la distance aléatoire entre les nœuds dans un disque de 1000 m n'a pas d'impact sur la latence moyenne.

- **La consommation énergétique moyenne des noeuds :**

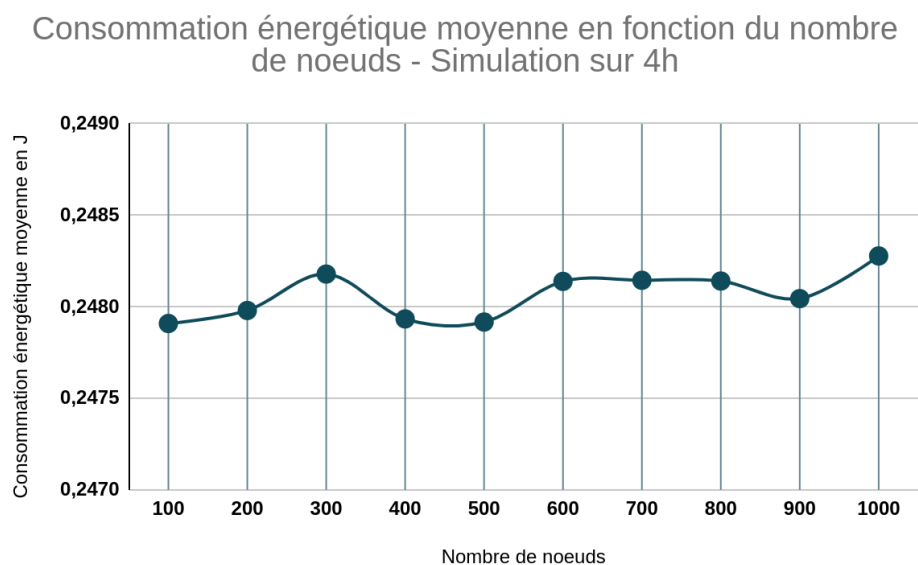


Fig.29 - La consommation énergétique moyenne des noeuds en fonction du nombre de noeuds - Simulation sur 4h

L'évolution de cette courbe montre une tendance relativement stable, cependant on observe une légère augmentation de la consommation moyenne à mesure que le nombre de noeuds augmente, passant de 0.2479J pour 100 noeuds à 0.2483J pour 1000 noeuds. Cette variation reste très faible et ne suit pas une progression linéaire stricte.