Integrantes:

- Andres Felipe Sanchez Sanchez
- Sergio Alejandro Pineda Vejar
- Cristian David Gonzalez Carrillo
- Cristian Adolfo Baquero Pico

Diseño de la aplicación

Para el diseño de la aplicación, se plantearon tres simuladores distintos, que brindan un desarrollo de los numerales propuestos. El primer simulador simula una red wifi adhoc con 20 nodos dispuestos en una topología en forma de anillo inicialmente y con un modelo de movilidad que simula un movimiento aleatorio de velocidad constante en donde se envían diversos tipos de tráfico con distintas distribuciones a través de 3 experimentos. El segundo simulador, simula una red de tecnología LoRa/LoRaWAN de 20 nodos (18 end devices, 1 gateway y 1 network server), con una topología inicial en anillo y un modelo de movilidad que simula un movimiento aleatorio de velocidad constante, que al igual que en la simulación anterior cuenta con 3 experimentos que varían el tipo de tráfico y su distribución, además cabe resaltar que esta simulación también cuenta con un modelo de demoras y pérdidas en la propagación. Finalmente, el simulador final cuenta con las mismas características de la segunda simulación, pero cuenta con llamados callback que permiten que la simulación de ns3 sea el ambiente junto con el sistema, que es modificado por un agente externo que desempeña el papel de observatorio para redefinir parámetros de la red y por medio del aprendizaje por refuerzo en base a ciertas observaciones del estado del sistema tomar acciones que permitan mejorar las métricas de la red.

Implementación de los Simuladores
Aspectos Transversales:

La Clase Experiment:
Se define la clase Experiment que recibe como parámetros en su método Run los string onTime y offTime que controlan la distribución del tráfico de red, el entero nodes que define la cantidad de nodos en la red, el entero stopTime que

determina el número de segundos que durará la simulación, el entero packetSizeOnOff que define el tamaño de los paquetes que se envían por la red y finalmente el entero radius, que permite controlar el radio del anillo que define inicialmente la topología.

```
class Experiment {
  public:
    Experiment();
       void Run(StringValue onTime, StringValue offTime,
uint32_t     nodes,     uint32_t     stopTime,     uint32_t
packetSizeOnOff, uint32_t radius);
};
```

Los modelos de movilidad desarrollados denotan una topologías de anillo inicialmente y permiten que los nodos se muevan de forma aleatoria con una velocidad constante

```
//
  // The ad-hoc network nodes need a mobility model so we
aggregate one to
  // each of the nodes we just finished building.
  //
  MobilityHelper mobility;

mobility.SetPositionAllocator("ns3::UniformDiscPositionAllo
cator", "rho", DoubleValue(radius),
    "X", DoubleValue(0.0), "Y", DoubleValue(0.0));

mobility.SetMobilityModel("ns3::RandomDirection2dMobilityMo
del",
    "Bounds", RectangleValue(Rectangle(-500, 500, -500,
500)),
    "Speed",
StringValue("ns3::ConstantRandomVariable[Constant=1]"),
    "Pause",
StringValue("ns3::ConstantRandomVariable[Constant=0.2]"));
  mobility.Install(backbone);
```

appHelper

Se define el objeto appHelper cuyo tipo es RandomPeriodicSenderHelper, con el fin de utilizar tres métodos en especial:

Método para establecer el periodo de la variable aleatoria:

```
appHelper.SetPeriodRandomVariable(trafficDistribution);
```

En este método se dará un valor a la variable m_pktPeriodRv por el valor del parámetro brindado a la función, para este caso trafficDistribution

```
void
RandomPeriodicSenderHelper::SetPeriodRandomVariable(Ptr<Ran
domVariableStream> periodRand)
{
     m_pktPeriodRV = periodRand;
}
```

Al igualar esta variable al valor de trafficDistribution permitirá enviar los paquetes dada una distribución probabilística que es dada como parámetro.

```
appHelper.SetPacketSize(packetSize);
```

En este método se dará un valor a la variable m_pktSize por el valor del parámetro brindado a la función, para este caso packetSize

```
void
RandomPeriodicSenderHelper::SetPacketSize(uint8_t size)
{
     m_pktSize = size;
}
```

De tal manera, al igualar esta variable a packetSizer se definirá la cantidad de paquetes que enviará la aplicación.

```
ApplicationContainer appContainer =
appHelper.Install(endDevices);
```

En este caso se enviará como parámetro al método Install los dispositivos finales endDevices, con el fin de llevarlos a la clase ApplicationContainer que hará del arreglo de aplicaciones de la simulación. El método InstallPriv tiene como objetivo retornar un puntero hacia la aplicación.

```
RandomPeriodicSenderHelper::Install(Ptr<Node> node) const
{
   return ApplicationContainer(InstallPriv(node));
}
```

Configuración de Nodos en LoRaWAN

La función configureNode recibe como parámetros un puntero al nodo que se desea configurar (node), un entero que denota el nuevo tamaño la tasa de datos de la segunda ventana de recepción (newWindowValue) del nodo (lo cual es una característica de los end devices en una red LoRaWAN), un entero que denota la nueva tasa de datos de la capa mac del end device (newDataRate) y un entero que denota el spreading factor de la capa física de la interfaz del end device.

Esta función realiza un set del spreading factor, la tasa de datos y la tasa de datos de la segunda ventana de recepción del end device al que apunta node. Esto permitirá cambiar el alcance de la red (un spreading factor mayor proporciona un mayor alcance de la red pero disminuye la velocidad de transmisión y aumenta también el time on air de los paquetes), la tasa de transferencia de datos que permite controlar la velocidad de modulación y transmisión de las señales que representan los paquetes (y que se debe configurar dependiendo del spreading factor)

```
void configureNode(Ptr<Node> node, u_int8_t newWindowValue,
u_int8_t newDataRate, uint8_t newSpreadingFactor)
{
  Ptr<NetDevice> dev = node->GetDevice(0);
  Ptr<LoraNetDevice> lora_dev =
DynamicCast<LoraNetDevice>(dev);
```

```
  Ptr<LorawanMac> lora_mac = lora_dev->GetMac();
  Ptr<ClassAEndDeviceLorawanMac> endDeviceMac =
DynamicCast<ClassAEndDeviceLorawanMac>(lora_mac);
  Ptr<LoraPhy> phy = lora_dev->GetPhy();
  Ptr<EndDeviceLoraPhy> end_device_phy =
DynamicCast<EndDeviceLoraPhy>(phy);
  end_device_phy->SetSpreadingFactor(newSpreadingFactor);

endDeviceMac->SetSecondReceiveWindowDataRate(newWindowValue
);
  endDeviceMac->SetDataRate(newDataRate);

}
```

Declaración de Estructuras y Enums

Dentro del nombre de espacio estándar, declaramos un enum llamado PacketOutcome que puede tomar como valores RECEIVED (si el paquete es recibido), INTERFERED (si el paquete no fue recibido por interferencia), NO_MORE_RECEIVERS (si el paquete no pudo ser recibido por ningún gateway), UNDER_SENSITIVITY (si el paquete no pudo ser recibido por baja sensibilidad en el receptor) y UNSET (si el paquete aun no se ha clasificado en uno de los anteriores). Por otro lado, también se define una estructura de PacketStatus que contiene el paquete, el id del nodo que lo envió (senderId), el número de resultado (outcomeNumber) y un vector de resultados (outcomes) que almacena los PacketOutcome con que se categoriza al paquete.

```
namespace std
{
  enum PacketOutcome
  {
    RECEIVED,
    INTERFERED,
    NO_MORE_RECEIVERS,
    UNDER_SENSITIVITY,
    UNSET
  };
```

```
  struct PacketStatus
  {
    Ptr<Packet const> packet;
    uint32_t senderId;
    int outcomeNumber;
    std::vector<enum PacketOutcome> outcomes;
  };

};
```

Gestión de Recepción de Paquetes en las Gateway de una red LoRaWAN

En este método, se recibe un iterador de un mapa que asocia un paquete con su PacketStatus. Esta función en principio verifica si el paquete ha sido recibido por todas las gateways y dependiendo de sus valores en el arreglo outcomes del packetStatus, actualiza las estadísticas.

```
void CheckReceptionByAllGWsComplete(std::map<Ptr<Packet
const>, std::PacketStatus>::iterator it)
{
  // Check whether this packet is received by all gateways
  if ((*it).second.outcomeNumber == nGateways)
  {
    // Update the statistics
    std::PacketStatus status = (*it).second;
    for (int j = 0; j < nGateways; j++)
    {
      switch ((int)status.outcomes.at(j)) //por si acaso
castear a entero lo del switch
      {
      case std::RECEIVED:
      {
        received += 1;
        break;
      }
      case std::INTERFERED:
      {
```

```
        interfered += 1;
        break;
      }
      case std::NO_MORE_RECEIVERS:
      {
        noMoreReceivers += 1;
        break;
      }
      case std::UNDER_SENSITIVITY:
      {
        underSensitivity += 1;
        break;
      }

      case std::UNSET:
      {
        break;
      }
      default:
      {
        break;
      }
      }
    }
    // Remove the packet from the tracker
    packetTracker.erase(it);
  }
}
```

Callbacks para trazabilidad de paquetes:

El mapa realiza un mapeo de un paquete a su estado, lo cual ha sido definido anteriormente en la sección de declaración de estructuras y enums.

```
std::map<Ptr<Packet const>, std::PacketStatus>
packetTracker;
```

El llamado de vuelta de la transmisión consiste en tener los paquetes y la identificación en la simulación, en este método se cambian diferentes estados por lo reportados en los parametros del metodo, para este caso son el paquete, el id de quien envia, el numero de salida, los resultados "outcomes" a ser UNSET y se realiza el mapeo.

```
void TransmissionCallback(Ptr<Packet const> packet,
uint32_t systemId)
{
 //NS_LOG_DEBUG("Transmitted a packet from device " <<
systemId);
 // Create a packetStatus
     std::PacketStatus status;
     status.packet = packet;
     status.senderId = systemId;
     status.outcomeNumber = 0;
     status.outcomes =
std::vector<std::PacketOutcome>(UNSET);
     packetTracker.insert(std::pair<Ptr<Packet const>,
     std::PacketStatus>(packet, status));
     count = count + 1;
}
```

El llamado de vuelta de la recepción del paquete consiste en realizar el mapeo del iterador y aumentar el número del resultado y el resultado "outcome". Además del cambio de estado a RECEIVED.

```
void PacketReceptionCallback(Ptr<Packet const> packet,
uint32_t systemId)
{
 //NS_LOG_INFO("A packet was successfully received at
gateway " << systemId);
 std::map<Ptr<Packet const>, std::PacketStatus>::iterator
it = packetTracker.find(packet);
```

```
 (*it).second.outcomes.at(systemId - nDevices) =
std::RECEIVED;
 (*it).second.outcomeNumber += 1;
 CheckReceptionByAllGWsComplete(it);
}
```

El llamado de vuelta de interferencia está encargado de realizar el mapeo del iterador it y aumenta el valor del resultado y el número del valor de resultado. Además del cambio de estado a INTERFERED.

```
void InterferenceCallback(Ptr<Packet const> packet,
uint32_t systemId)
{
 //NS_LOG_INFO("A packet was interferenced at gateway " <<
systemId);

 std::map<Ptr<Packet const>, std::PacketStatus>::iterator
it = packetTracker.find(packet);
 it->second.outcomes.at(systemId - nDevices) =
std::INTERFERED;
 it->second.outcomeNumber += 1;
}
```

El llamado de vuelta de no más recepciones está encargado de realizar el mapeo del iterador it y aumenta el valor del resultado y el número del valor de resultado. Además del cambio de estado a NO_MORE_RECEIVERS.

```
void NoMoreReceiversCallback(Ptr<Packet const> packet,
uint32_t systemId)
{
 // NS_LOG_INFO ("A packet was lost because there were no
more receivers at gateway " << systemId);

 std::map<Ptr<Packet const>, std::PacketStatus>::iterator
it = packetTracker.find(packet);
```

```
 (*it).second.outcomes.at(systemId - nDevices) =
std::NO_MORE_RECEIVERS;
 (*it).second.outcomeNumber += 1;

 CheckReceptionByAllGWsComplete(it);
}
```

El llamado de vuelta de bajo sensibilidad está encargado de realizar el mapeo del iterador it y aumenta el valor del resultado y el número del valor de resultado. Además del cambio de estado a UNDER_SENSITIVITY.

```
void UnderSensitivityCallback(Ptr<Packet const> packet,
uint32_t systemId)
{
 // NS_LOG_INFO ("A packet arrived at the gateway under
sensitivity at gateway " << systemId);

 std::map<Ptr<Packet const>, std::PacketStatus>::iterator
it = packetTracker.find(packet);
 (*it).second.outcomes.at(systemId - nDevices) =
std::UNDER_SENSITIVITY;
 (*it).second.outcomeNumber += 1;

 CheckReceptionByAllGWsComplete(it);
}
```

Simulador 1 :

Configuración de la capa física y mac de la red wifi adhoc

Inicialmente configuraremos una variable tipos string llamada phyMode que define una tasa de modulación DSSS (Direct-sequence spread spectrum) de 5.5 Mbps:

```
 //
  // First, we declare and initialize a few local variables
that control some
  // simulation parameters.
  //
```

```
  std::string phyMode("DsssRate5_5Mbps");
```

Posteriormente creamos un contenedor de nodos con el número de nodos recibido como parámetro

```
  //
  // Create a container to manage the nodes of the adhoc
(backbone) network.
  // Later we'll create the rest of the nodes we'll need.
  //
  NodeContainer backbone;
  backbone.Create(nodes);
```

Por medio del wifi helper configuramos el estándar de la capa física de wifi que trabajara que será en este caso 802.11b , configuramos la mac como tipo AdhocWifiMac. Finalmente con ayuda del helper YansWifiPhyHelper configuramos el canal con la constante DLT_IEEE802_11_RADIO que denota el formato de header de radiotap que provee de información adicional de cada frame al realizar su transmisión. Por último, se instala en los dispositivos la capa física y mac configurada previamente con la sentencia wifi.Install, lo que nos permite inicializar el contenedor de dispositivos de red backboneDevices

```
 WifiHelper wifi;
  wifi.SetStandard(WIFI_PHY_STANDARD_80211b);

wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager"
,
    "DataMode", StringValue(phyMode),
    "ControlMode", StringValue(phyMode));
  WifiMacHelper mac;
  mac.SetType("ns3::AdhocWifiMac");

  YansWifiPhyHelper wifiPhy;

wifiPhy.SetPcapDataLinkType(WifiPhyHelper::DLT_IEEE802_11_R
ADIO);
```

```
  YansWifiChannelHelper
wifiChannel=YansWifiChannelHelper::Default();


  //Wifi channel creation
  wifiPhy.SetChannel(wifiChannel.Create());

  NetDeviceContainer backboneDevices =
wifi.Install(wifiPhy, mac, backbone);
```

Instalación del Stack Internet

Se procede con la determinación de la lista de protocolos de enrutamiento con ayuda del helper Ipv4StaticRoutingHelper , en la que se incluye OLSR con mayor prioridad y enrutamiento estático con menor prioridad, así mismo se configura con esta lista de protocolos el stack de internet por medio del InternetStackHelper y se instala en los dispositivos. Finalmente se asigna una dirección ip a cada uno de los dispositivos en la red 198.168.0.0 con mascara de subred 255.255.255.0 por medio de la sentencia Assign del helper Ipv4AddressHelper

```
  // We enable OLSR (which will be consulted at a higher
priority than
  // the global routing) on the backbone ad hoc nodes
  NS_LOG_INFO("Enabling OLSR routing on all backbone
nodes");
  OlsrHelper olsr;
  Ipv4StaticRoutingHelper staticRouting;

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

  //
```

```
  // Add the IPv4 protocol stack to the nodes in our
container
  //
  InternetStackHelper internet;
  internet.SetRoutingHelper(list);  // has effect on the
next Install ()
  internet.Install(backbone);

  //
  // Assign IPv4 addresses to the device drivers (actually
to the associated
  // IPv4 interfaces) we just created.
  //
  Ipv4AddressHelper ipAddrs;
  ipAddrs.SetBase("192.168.0.0", "255.255.255.0");
  Ipv4InterfaceContainer i =
ipAddrs.Assign(backboneDevices);
```

Configuración de la Aplicación

Se configura la aplicación onOff que simula el envío de tráfico con un patrón on-off que tiene un tiempo de envío denotado por el onTime y un tiempo de "reposo" en el que no se envían paquetes denotado por OffTime que se van alternando. En este caso denotamos el primer nodo como el nodo fuente y el último nodo como el nodo sink (sumidero). Con ayuda del helper OnOffHelper, hacemos un set de los atributos offTime y onTime que vienen como parámetros (del método run), el packetSize que también viene como parámetro y la tasa de datos que fijamos en 5Mbpsy el número máximo de bytes de transmisión que fijamos en 10989173 . Asimismo iniciamos esta aplicación en el segundo 3 y finalizamos un segundo antes del tiempo de simulación.

```
  //
  // OnOff Application
  //

  uint32_t sourceOnOffNode = 0;
  uint32_t sinkOnOffNode = nodes - 1;
```

```
  uint16_t appport = 5000;

  Ptr<Node> appSourceOnOff =
NodeList::GetNode(sourceOnOffNode);
  // We want the sink to be the last node created in the
topology.
  //
  Ptr<Node> appSinkOnOff =
NodeList::GetNode(sinkOnOffNode);
  // Let's fetch the IP address of the last node, which is
on Ipv4Interface 1
  Ipv4Address remoteAddr = appSinkOnOff->GetObject<Ipv4>
()->GetAddress(1, 0).GetLocal();

  OnOffHelper onoff("ns3::UdpSocketFactory",
Address(InetSocketAddress(remoteAddr, appport)));
  onoff.SetAttribute("OffTime", offTime);
  onoff.SetAttribute("OnTime", onTime);
  onoff.SetAttribute("MaxBytes", UintegerValue(10989173));
  onoff.SetAttribute("DataRate", StringValue("5Mbps"));
  onoff.SetAttribute("PacketSize",
UintegerValue(packetSizeOnOff));
  ApplicationContainer appsOnOff =
onoff.Install(appSourceOnOff);
  appsOnOff.Start(Seconds(3));
  appsOnOff.Stop(Seconds(stopTime - 1));

  // Create a packet sink to receive these packets
  PacketSinkHelper sinkOnOff("ns3::UdpSocketFactory",
InetSocketAddress(Ipv4Address::GetAny(), appport));
  appsOnOff = sinkOnOff.Install(appSinkOnOff);
  appsOnOff.Start(Seconds(2));
  Ptr<PacketSink> packetSinkServer =
DynamicCast<PacketSink>(appsOnOff.Get(0));
```

Monitoreo de las métricas de red

El monitoreo de las métricas de red lo hacemos por medio del módulo FlowMonitor, que instanciamos antes de iniciar la simulación, y del cual podemos extraer puntualmente las siguientes métricas: paquetes enviados (Tx), bytes enviados (Tx Bytes), la carga ofrecida, los paquetes recibidos (Rx packets), los bytes recibidos (Rx bytes), la tasa de transferencia efectiva (Throughput), la demora promedio (Mean delay) , la media del Jitter y el porcentaje de paquetes recibidos.

```
 // Flow monitor
  Ptr<FlowMonitor> flowMonitor;
  FlowMonitorHelper flowHelper;
  flowMonitor = flowHelper.InstallAll();

  Simulator::Stop(Seconds(stopTime));
  Simulator::Run();
  Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>
(flowHelper.GetClassifier());
  std::map<FlowId, FlowMonitor::FlowStats > stats =
flowMonitor->GetFlowStats();
  std::cout << std::endl << "***Flow monitor statistics
***" << std::endl;
  std::cout << "  Tx Packets:   " << stats[1].txPackets <<
std::endl;
  std::cout << "  Tx Bytes:   " << stats[1].txBytes <<
std::endl;
  std::cout << "  Offered Load: " << stats[1].txBytes *8.0
/ (stats[1].timeLastTxPacket.GetSeconds() -
stats[1].timeFirstTxPacket.GetSeconds()) / 1000000 << "
Mbps" << std::endl;
  std::cout << "  Rx Packets:   " << stats[1].rxPackets <<
std::endl;
  std::cout << "  Rx Bytes:   " << stats[1].rxBytes <<
std::endl;
  std::cout << "  Throughput: " << stats[1].rxBytes *8.0 /
(stats[1].timeLastRxPacket.GetSeconds() -
```

```
stats[1].timeFirstRxPacket.GetSeconds()) / 1000000 << "
Mbps" << std::endl;
  std::cout << "  Mean delay:   " <<
stats[1].delaySum.GetSeconds() / stats[1].rxPackets <<
std::endl;
  std::cout << "  Mean jitter:   " <<
stats[1].jitterSum.GetSeconds() / (stats[1].rxPackets - 1)
<< std::endl;
  flowMonitor->SerializeToXmlFile("data.flowmon", true,
true);
  std::cout << "Number of OnOffPackets received: " <<
packetSinkServer->GetTotalRx() / packetSizeOnOff <<
std::endl;
  std::cout << "% of OnOffPackets received: " << (100 *
(packetSinkServer->GetTotalRx() / packetSizeOnOff) /
stats[1].txPackets) << std::endl;

  Simulator::Destroy();
```

Simulación de Distintos Escenarios

Finalmente, ajustamos como parámetros por defecto el número de nodos con un valor de 20, el tiempo de simulación con un valor de 30 segundos, el tamaño de los paquetes con un tamaño de 1000 bytes y un radio para la topología de anillo de 10. Posteriormente se procede a simular 3 escenarios distintos: el primero simula tráfico de video bajo demanda con un offTime que obedece a una estructura probabilística dada por la primitiva LogNormalRandomVariable, el segundo simula tráfico de llamadas que obedece a una estructura probabilística dada por la primitiva ExponentialRandomVariable, el tercero simula tráfico uniforme que obedece a una estructura probabilística dada por la primitiva UniformRandomVariable.

```
int main(int argc, char *argv[]) {

  uint32_t nodes = 20;
  uint32_t stopTime = 30;
  uint32_t packetSize = 1000;
```

```
  uint32_t radius = 10;

  //
  // For convenience, we add the local variables to the
command line argument
  // system so that they can be overridden with flags such
as
  // "--nodes=20"
  //
  CommandLine cmd;
  cmd.AddValue("nodes", "Number of nodes", nodes);
  cmd.AddValue("stopTime", "Simulation stop time
(seconds)", stopTime);
  cmd.AddValue("packetSize", "Packet size (bytes)",
packetSize);
  cmd.AddValue("radius", "The radius of the area to
simulate", radius);

  //
  // The system global variables and the local values added
to the argument
  // system can be overridden by command line arguments by
using this call.
  //
  cmd.Parse(argc, argv);

  if (stopTime < 10) {
    std::cout << "Use a simulation stop time >= 10 seconds"
<< std::endl;
    exit(1);
  }

  Experiment experiment;
```

```
  // Based on this paper:
http://www.scielo.org.co/pdf/dyna/v84n202/0012-7353-dyna-84
-202-00055.pdf
  NS_LOG_UNCOND("Traffic video on demand");
  StringValue
offTime("ns3::LogNormalRandomVariable[Mu=0.4026|Sigma=0.035
2]");
  StringValue
onTime("ns3::WeibullRandomVariable[Shape=2|Scale=10]");
  experiment = Experiment();
  experiment.Run(onTime, offTime, nodes, stopTime,
packetSize, radius);

  NS_LOG_UNCOND("Traffic calls");
  offTime =
StringValue("ns3::ExponentialRandomVariable[Mean=2.0|Bound=
10]");
  experiment = Experiment();
  experiment.Run(onTime, offTime, nodes, stopTime,
packetSize, radius);

  NS_LOG_UNCOND("Traffic Uniform");
  offTime =
StringValue("ns3::UniformRandomVariable[Max=30|Min=0.1]");
  experiment = Experiment();
  experiment.Run(onTime, offTime, nodes, stopTime,
packetSize, radius);
}
```

Simulador 2

Inicialización de variables

Se inicializan las variables:

- nDevices: denota el número de end devices de la red de lorawan.
- nGateways: denota el número de gateways de la red de lorawan.

- radius: denota el radio de la topología de anillo inicial
- simulationTime: denota el número de segundos que dura la simulación
- packetSize: denota el tamaño de los paquetes en bytes.
- realisticChannelModel : denota una variable booleana, si es verdadero el simulador hace el escenario más realista.
- count: esta variable contará el número de paquetes enviados
- received: esta variable cuenta el número de paquetes recibidos.
- noMoreReceivers: esta variable cuenta el número de paquetes que no se reciben por ningún gateway.
- interfered: esta variable cuenta el número de paquetes que no se reciben por interferencia.
- underSensitivity: esta variable cuenta el número de paquetes que no se reciben por baja sensibilidad del receptor.

```
// Network settings
int nDevices = 18;
int nGateways = 1;
double radius = 6400;
double simulationTime = 600;
int packetSize = 20;

// Channel model
bool realisticChannelModel = true;

/************************/
/* Lorawan Tracker */
/************************/

int count = 0;
int received = 0;
int noMoreReceivers = 0;
int interfered = 0;
int underSensitivity = 0;
```

Creación de los modelos de demora y de pérdida

Se proponen los modelos de demora y pérdida, dado que en este caso el modelo es realista, se ajusta la simulación para representar interferencia dada por objetos u obstáculos en la red.

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

  if (realisticChannelModel)
  {
    // Create the correlated shadowing component
    Ptr<CorrelatedShadowingPropagationLossModel> shadowing
=
      CreateObject<CorrelatedShadowingPropagationLossModel>
();

    // Aggregate shadowing to the logdistance loss
    loss->SetNext(shadowing);

    // Add the effect to the channel propagation loss
    Ptr<BuildingPenetrationLoss> buildingLoss =
CreateObject<BuildingPenetrationLoss> ();

    shadowing->SetNext(buildingLoss);
  }

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

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

Creación de los helpers

Se crean los helpers para la capa física, la capa mac, la creación del network server y el forward helper para la redirección de paquetes en las gateways.

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

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

  // Create the LoraHelper
  LoraHelper helper = LoraHelper();
  helper.EnablePacketTracking();  // Output filename
  // helper.EnableSimulationTimePrinting ();

  //Create the NetworkServerHelper
  NetworkServerHelper nsHelper = NetworkServerHelper();

  //Create the ForwarderHelper
  ForwarderHelper forHelper = ForwarderHelper();
```

Creacion e instalacion de los dispositivos de red end devices:
Se generan las direcciones y se instalan las capas mac y física en los end devices. Luego se configura a cada capa física de cada end device el callback para la transmisión de paquetes y un datarate con valor de 0, un spreading factor de 12 y datarate para la segunda ventana de recepción de 1.

```
// Create the LoraNetDevices of the end devices
  uint8_t nwkId = 54;
  uint32_t nwkAddr = 1864;
  Ptr<LoraDeviceAddressGenerator> addrGen =
    CreateObject<LoraDeviceAddressGenerator> (nwkId,
nwkAddr);

  // Create the LoraNetDevices of the end devices
  macHelper.SetAddressGenerator(addrGen);
  phyHelper.SetDeviceType(LoraPhyHelper::ED);
  macHelper.SetDeviceType(LorawanMacHelper::ED_A);
```

```
  helper.Install(phyHelper, macHelper, endDevices);
 // Connect trace sources
  for (NodeContainer::Iterator j = endDevices.Begin(); j !=
endDevices.End(); ++j)
  {
    Ptr<Node> node = *j;
    Ptr<LoraNetDevice> loraNetDevice =
node->GetDevice(0)->GetObject<LoraNetDevice> ();
    Ptr<LoraPhy> phy = loraNetDevice->GetPhy();
    phy->TraceConnectWithoutContext("StartSending",
MakeCallback(&TransmissionCallback));
    configureNode(node, 1, 0, 12);
  }
```

Instalación de los dispositivos de red en los dispositivos gateways
Se instala la capa mac y la capa física en los gateways y se configuran los callbacks de interferencia, recepción baja sensibilidad y no recepción por parte de ninguna gateway, los cuales se explicaron anteriormente.

```
 // Create the gateway nodes (allocate them uniformely on
the disc)
  NodeContainer gateways;
  gateways.Create(nGateways);

  Ptr<ListPositionAllocator> allocator =
CreateObject<ListPositionAllocator> ();
  // Make it so that nodes are at a certain height > 0
  allocator->Add(Vector(0.0, 0.0, 15.0));
  mobility.SetPositionAllocator(allocator);
  mobility.Install(gateways);

  // Create a netdevice for each gateway
  phyHelper.SetDeviceType(LoraPhyHelper::GW);
  macHelper.SetDeviceType(LorawanMacHelper::GW);
  helper.Install(phyHelper, macHelper, gateways);
```

```
  for (NodeContainer::Iterator j = gateways.Begin(); j !=
gateways.End(); j++)
  {
    Ptr<Node> gNode = *j;
    Ptr<NetDevice> gNetDevice = gNode->GetDevice(0);
    Ptr<LoraNetDevice> gLoraNetDevice =
gNetDevice->GetObject<LoraNetDevice>();
    NS_ASSERT(gLoraNetDevice != 0);
    Ptr<GatewayLoraPhy> gwPhy =
gLoraNetDevice->GetPhy()->GetObject<GatewayLoraPhy>();
    gwPhy->TraceConnectWithoutContext("ReceivedPacket",

MakeCallback(&PacketReceptionCallback));

gwPhy->TraceConnectWithoutContext("LostPacketBecauseInterfe
rence",

MakeCallback(&InterferenceCallback));

gwPhy->TraceConnectWithoutContext("LostPacketBecauseNoMoreR
eceivers",

MakeCallback(&NoMoreReceiversCallback));

gwPhy->TraceConnectWithoutContext("LostPacketBecauseUnderSe
nsitivity",

MakeCallback(&UnderSensitivityCallback));
  }
```

Definición de la topología simulando la disposición de los dispositivos en un edificio

Se simula la disposición de los dispositivos en un edificio, como el canal es realista la grilla tendrá ancho y largo distinto a cero.

```
 double xLength = 130;
  double deltaX = 32;
  double yLength = 64;
```

```
  double deltaY = 17;
  int gridWidth = 2 *radius / (xLength + deltaX);
  int gridHeight = 2 *radius / (yLength + deltaY);
  if (realisticChannelModel == false)
  {
    gridWidth = 0;
    gridHeight = 0;
  }
  Ptr<GridBuildingAllocator> gridBuildingAllocator;
  gridBuildingAllocator =
CreateObject<GridBuildingAllocator> ();
  gridBuildingAllocator->SetAttribute("GridWidth",
UintegerValue(gridWidth));
  gridBuildingAllocator->SetAttribute("LengthX",
DoubleValue(xLength));
  gridBuildingAllocator->SetAttribute("LengthY",
DoubleValue(yLength));
  gridBuildingAllocator->SetAttribute("DeltaX",
DoubleValue(deltaX));
  gridBuildingAllocator->SetAttribute("DeltaY",
DoubleValue(deltaY));
  gridBuildingAllocator->SetAttribute("Height",
DoubleValue(6));
  gridBuildingAllocator->SetBuildingAttribute("NRoomsX",
UintegerValue(2));
  gridBuildingAllocator->SetBuildingAttribute("NRoomsY",
UintegerValue(4));
  gridBuildingAllocator->SetBuildingAttribute("NFloors",
UintegerValue(2));
  gridBuildingAllocator->SetAttribute(    "MinX",
DoubleValue(-gridWidth *(xLength + deltaX) / 2 + deltaX /
2));
  gridBuildingAllocator->SetAttribute(    "MinY",
DoubleValue(-gridHeight *(yLength + deltaY) / 2 + deltaY /
2));
```

```
  BuildingContainer bContainer =
gridBuildingAllocator->Create(gridWidth *gridHeight);

  BuildingsHelper::Install(endDevices);
  BuildingsHelper::Install(gateways);
```

Instalación de las aplicaciones en los end devices

```
 /*********************************************
   *Install applications on the end devices  *
   *********************************************/

  Time appStopTime = Seconds(simulationTime);
  RandomPeriodicSenderHelper appHelper =
RandomPeriodicSenderHelper();
  appHelper.SetPeriodRandomVariable(trafficDistribution);
  appHelper.SetPacketSize(packetSize);
  ApplicationContainer appContainer =
appHelper.Install(endDevices);

  appContainer.Start(Seconds(0));
  appContainer.Stop(appStopTime);
```

Creación del servidor de red y redirección de las gateways a este

```
  /**************************
   *Create Network Server  *
   **************************/

  // Create the NS node
  NodeContainer networkServer;
  networkServer.Create(1);

  // Create a NS for the network
  nsHelper.SetEndDevices(endDevices);
  nsHelper.SetGateways(gateways);
```

```
  nsHelper.Install(networkServer);

  //Create a forwarder for each gateway
  forHelper.Install(gateways);
```

Simulación de Distintos Escenarios

Posteriormente se procede a simular 3 escenarios distintos: el primero simula tráfico de video bajo demanda con un offTime que obedece a una estructura probabilística dada por la primitiva LogNormalRandomVariable, el segundo simula tráfico de llamadas que obedece a una estructura probabilística dada por la primitiva ExponentialRandomVariable, el tercero simula tráfico uniforme que obedece a una estructura probabilística dada por la primitiva UniformRandomVariable.

Simulador 3:

El tercer simulador integra el agente de OpenAI, para lo cual se definen las siguientes funciones (cabe resaltar que solo se explicaran a profundidad las funciones relacionadas con la integración del agente dado que el diseño de este simulador es muy similar al diseño del simulador 2 explicado previamente):

Función getPosition

Dado un nodo, esta función retorna el vector que denota la posición del mismo en la red.

```
Vector getPosition(Ptr<Node> node)
{
                Ptr<MobilityModel>        mobility        =
node->GetObject<MobilityModel> ();
  Vector position = mobility->GetPosition();
  return position;
}
```

Función distanceNode

Esta función recibe como parámetro los vectores de posición de dos nodos y retorna la distancia entre ambos.

```
double distanceNode(Vector pos1, Vector pos2)
{
```

```
    return sqrt(pow(pos1.x - pos2.x, 2) + pow(pos1.y -
pos2.y, 2) + pow(pos1.z - pos2.z, 2));
}
```

Función MyGetObservationSpace

Esta función define el espacio de observación, en este caso, se define con un valor mínimo de 0 y un valor máximo de 1000, con una dimensión de el número de nodos x 1 dado que nuestro espacio de observación almacenará la distancia de cada nodo a la gateway con el fin de poder tomar una decisión en base a esta en el agente.

```
/*
Define observation space
*/
Ptr<OpenGymSpace> MyGetObservationSpace(void)
{
  uint32_t nodeNum = NodeList::GetNNodes ();
  float low = 0.0;
  float high = 1000.0;
  std::vector<uint32_t> shape = { nodeNum,
  };
  std::string dtype = TypeNameGet<double> ();
                Ptr<OpenGymBoxSpace>        space        =
CreateObject<OpenGymBoxSpace> (low, high, shape, dtype);
  NS_LOG_UNCOND("MyGetObservationSpace: " << space);
  return space;
}
```

Función MyGetActionSpace

Esta función define el espacio de acciones, que en este caso será de tipo entero y deberá estar entre 7 y 12 dado que estos son los valores posibles del spreading factor

```
/*
Define action space
*/
```

```
Ptr<OpenGymSpace> MyGetActionSpace(void)
{
  uint32_t nodeNum = 1;
  float low = 7.0;
  float high = 12.0;
  std::vector<uint32_t> shape = { nodeNum,
  };
  std::string dtype = TypeNameGet<uint32_t> ();
                Ptr<OpenGymBoxSpace>        space        =
CreateObject<OpenGymBoxSpace> (low, high, shape, dtype);
  NS_LOG_UNCOND("MyGetActionSpace: " << space);
  return space;
}
```

Función MyGetGameOver

Esta función podría ser quien levanta la bandera para finalizar la interacción con el agente, pero en este caso retorna siempre falso.

```
/*
Define game over condition
*/
bool MyGetGameOver(void)
{
  bool isGameOver = false;
  NS_LOG_UNCOND("MyGetGameOver: " << isGameOver);
  return isGameOver;
}
```

Función MyGetObservation

Esta función retorna la observación en un momento dado en la simulación, como puede apreciarse calcula la distancia de cada nodo a la gateway y la almacena en el contenedor de opengym box

```
Ptr<OpenGymDataContainer> MyGetObservation(void)
{
  uint32_t nodeNum =NodeList::GetNNodes (); ;
  std::vector<uint32_t> shape = { nodeNum,
```

```
  };
        Ptr<OpenGymBoxContainer    <    double>>    box    =
CreateObject<OpenGymBoxContainer < double>> (shape);
  NodeList::Iterator gat_i = gateways.Begin();
  Ptr<Node> gatewayNode = *gat_i;
  Vector gatewayPosition = getPosition(gatewayNode);
   for  (NodeList::Iterator  i  =  endDevices.Begin();  i  !=
endDevices.End(); ++i)
  {
    Ptr<Node> node = *i;
    Vector nodePosition = getPosition(node);
          double   distance   =   distanceNode(nodePosition,
gatewayPosition);

    box->AddValue(distance);
  }

  NS_LOG_UNCOND("MyGetObservation: " << box);
  return box;
}
```

Función MyGetExtraInfo

Esta función retorna un string que en este caso no contiene información relevante.

```
std::string MyGetExtraInfo(void)
{
  std::string myInfo = "linear-wireless-mesh";
  myInfo += "|123";
  NS_LOG_UNCOND("MyGetExtraInfo: " << myInfo);
  return myInfo;
}
```

Función MyExecuteActions

Esta función toma el spreading factor que determina el agente y lo configura en todos los end devices. Cabe aclarar que el mapa dataRateCorrespondece contiene la

correspondencia entre cada spreading factor y su respectiva tasa de datos que se acogen al estándar de LoRaWAN.

```
bool MyExecuteActions(Ptr<OpenGymDataContainer> action)
{
  NS_LOG_UNCOND("MyExecuteActions: " << action);

       Ptr<OpenGymBoxContainer    <    uint32_t>>    box    =
DynamicCast<OpenGymBoxContainer < uint32_t>> (action);
  std::vector<uint32_t> actionVector = box->GetData();
  int i = 0;
  u_int32_t newSpreadingFactor;
  for (NodeContainer::Iterator j = endDevices.Begin(); j !=
endDevices.End(); ++j)
  {
    Ptr<Node> node = *j;
    newSpreadingFactor = actionVector.at(0);
                                configureNode(node,        1,
dataRateCorrespondence[newSpreadingFactor],
newSpreadingFactor);
    i++;
  }

  return true;
}
```

Función MyGetReward

Esta función retorna la recompensa al agente por haber ejecutado cierta acción previamente, en este caso la recompensa es el número de paquetes recibidos a partir de dicha acción.

```
float MyGetReward(void)
{
  static float lastValue = 0.0;
  float reward = (double) received - lastValue;
  lastValue = (float) received;
  return reward;
}
```

Funcion ScheduleNextStateRead

Esta función programa las lecturas de las observaciones por parte del agente.

```
void ScheduleNextStateRead(double envStepTime,
Ptr<OpenGymInterface> openGymInterface)
{
  Simulator::Schedule(Seconds(envStepTime),
&ScheduleNextStateRead, envStepTime, openGymInterface);
  openGymInterface->NotifyCurrentState();
}
```

Api del Agente

Con la interfaz de Open Gym, se configuran los callbacks para que el agente pueda observar y ejecutar acciones sobre el sistema y su entorno

```
// OpenGym Env
  uint16_t port = 5555;
  double envStepTime = 10;
          Ptr<OpenGymInterface>     openGymInterface     =
CreateObject<OpenGymInterface> (port);

openGymInterface->SetGetActionSpaceCb(MakeCallback(&MyGetAc
tionSpace));

openGymInterface->SetGetObservationSpaceCb(MakeCallback(&My
GetObservationSpace));

openGymInterface->SetGetGameOverCb(MakeCallback(&MyGetGameO
ver));

openGymInterface->SetGetObservationCb(MakeCallback(&MyGetOb
servation));

openGymInterface->SetGetRewardCb(MakeCallback(&MyGetReward)
);
```

```
openGymInterface->SetGetExtraInfoCb(MakeCallback(&MyGetExtraInfo));

openGymInterface->SetExecuteActionsCb(MakeCallback(&MyExecuteActions));

  Simulator::Schedule(Seconds(0.0), &ScheduleNextStateRead, envStepTime, openGymInterface);

  ////////////////
  // Simulation //
  ////////////////

  // Flow monitor
  Simulator::Stop(appStopTime + Seconds(180));

  NS_LOG_INFO("Running simulation...");
  Simulator::Run();

  openGymInterface->NotifySimulationEnd();

  Simulator::Destroy();
```

Agente de OpenAI Gym

Este agente realiza acciones aleatorias con un 80% de probabilidad, mientras que en el restante lleva a cabo un razonamiento en base a los rewards que obtiene de las acciones llevadas a cabo anteriormente, y selecciona según el nivel de la observación (que es proporcional a la distancia promedio de los nodos al gateway) aquella acción que le haya generado un mayor reward

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import argparse
import time
import numpy as np
```

```
from ns3gym import ns3env
import random

__author__ = "Piotr Gawlowicz"
__copyright__ = "Copyright (c) 2018, Technische Universität
Berlin"
__version__ = "0.1.0"
__email__ = "gawlowicz@tkn.tu-berlin.de"


parser      =      argparse.ArgumentParser(description='Start
simulation script on/off')
parser.add_argument('--start',
                    type=int,
                    default=0,
                        help='Start simulation script 0/1,
Default: 0')
parser.add_argument('--iterations',
                    type=int,
                    default=1,
                      help='Number of iterations, Default:
1')
args = parser.parse_args()
startSim = bool(args.start)
iterationNum = int(args.iterations)

port = 5555
simTime = 600 # seconds
stepTime = 10 # seconds
seed = 0
simArgs = {"--simTime": simTime,
           "--testArg": 123,
           "--distance": 500}
debug = False
```

```
env     =     ns3env.Ns3Env(port=port,     stepTime=stepTime,
startSim=startSim,      simSeed=seed,       simArgs=simArgs,
debug=debug)
env.reset()


ob_space = env.observation_space
ac_space = env.action_space
print("Observation space: ", ob_space,  ob_space.dtype)
print("Action space: ", ac_space, ac_space.dtype)
q_table = np.zeros([10,13])*12

stepIdx = 0
currIt = 0
allRxPkts = 0

alpha = 0.1
gamma = 0.6
epsilon=0.8
def calculate_cw_window(num):



    k=np.random.randint(low=10,high=90, size=1)
       action  =  np.ones(shape=len(state),  dtype=np.uint8)
*10#* k[0]


    return action

try:
    while True:
        state = env.reset()
        reward = 0
        mean=np.mean(np.array(state))
        level=int(np.floor(mean/1000))
```

```
        print("Start iteration: ", currIt)
        print("Step: ", stepIdx)
        print("---state: ", state)
        print("---level: ",level)

        while True:
            stepIdx += 1

            allRxPkts += reward
            if random.uniform(0,1)<epsilon:
                action=env.action_space.sample()
            else:

                action = [np.argmax(q_table[level])]


            print("---action: ", action)

                        next_state,  reward,  done,  info  =
env.step(action)
                                            print    ("distancia
promedio:",np.mean(np.array(next_state)))
            mean=np.mean(np.array(next_state))
            level=int(np.floor(mean/1000))
             q_table[level,action]=q_table[level,action] if
q_table[level,action]>reward else reward
            print("Step: ", stepIdx)
                  print("---state,  reward,  done,  info:  ",
next_state, reward, done, info)









            if done:
```

```
                stepIdx = 0
                print("All rx pkts num: ", allRxPkts)
                allRxPkts = 0

                if currIt + 1 < iterationNum:
                    env.reset()
                break

        currIt += 1
        if currIt == iterationNum:
            break

except KeyboardInterrupt:
    print("Ctrl-C -> Exit")
finally:
    env.close()
    print("Done")
```