SIFRAN: Evaluating IoT Networks with a no-code Framework based on NS-3

Samir Si-Mohammed
Univ Lyon, ENS de Lyon, Université Claude Bernard Lyon 1, Inria, CNRS
Stackeo
Lyon, Rhône-Alpes, France
samir.si-mohammed@ens-lyon.fr

Malasri Janumporn
Univ Lyon, ENS de Lyon, Université Claude Bernard Lyon 1, Inria, CNRS
Lyon, Rhône-Alpes, France
malasri.janumporn@etu.univ-lyon1.fr

Thomas Begin
Univ Lyon, ENS de Lyon, Université Claude Bernard Lyon 1, Inria, CNRS
Lyon, Rhône-Alpes, France
thomas.begin@ens-lyon.fr

Isabelle Guérin Lassous
Univ Lyon, ENS de Lyon, Université Claude Bernard Lyon 1, Inria, CNRS
Lyon, Rhône-Alpes, France
isabelle.guerin-lassous@ens-lyon.fr

Pascale Vicat-Blanc
Stackeo
Lyon, Rhône-Alpes, France
pascale@stackeo.io

ABSTRACT

With the tremendous ascension of the Internet of Things over the recent years, NS-3 has consolidated its position in terms of popularity among the research community. Indeed, it has become one of the most used open-source network simulators, with an important community of users and contributors. However, the growth of this community is somehow constrained by the networking and programming skills required to use NS-3. This strongly reduces the NS-3 traction within the industrial community, since many IoT specialists lack these skills. In this paper, we present SIFRAN, a no-code framework for IoT networks simulation using NS-3. The main objective of SIFRAN is to extend the use of NS-3 to a community of non-programmers by making them able to benefit from its features without writing a single line of code. We show how the framework can be used via a simple web interface for simulating Wi-Fi- and LoRaWAN-based IoT setups, and how the programmers' community of NS-3 can contribute to the framework by adding more IoT network technologies.

CCS CONCEPTS

• **Networks** → **Network simulations**.

KEYWORDS

Internet of Things, NS-3, No-code, Framework, Simulation.

1 INTRODUCTION

The Internet of Things, or IoT, defined as the convergence of the digital and physical worlds, has become a fundamental trend underlying the digital transformation of enterprises and is becoming the beating heart of their operations. A wide range of connectivity options are now offered to IoT users. New low-power communication technologies like LoRaWAN or Sigfox have emerged. Remarkable advances have been made in network technology and protocols to serve an increasing number of IoT use cases. These technologies differ from each other, whether in terms of inherent parameters or in terms of their targeted applications. The variety of options can be seen as an opportunity to widen the range of possible IoT use cases. However, they often make it hard for researchers and industrial companies to make the right technology choice and configuration setting, yet these are crucial decisions. Indeed, under- or over-sizing a network has to be avoided to ensure profitability. A good trade-off between cost and QoS has to be found. To address this problem, simulation appears as a key enabler for the IoT network technology selection. Indeed, it can provide good insights about the performance of a technology at low cost since no real IoT material is needed.

The network simulator 3, or commonly called NS-3, is a discrete-event simulator that has been developed to provide an open and extensible network simulation platform for networking research and education. Due to its highly available documentation and the important set of network technologies it supports, it has become one of the most used simulators in the network community. However, NS-3 is targeting programmers rather than IoT architects and solution vendors. This is due to the fact that it requires network expertise and C++ programming skills, while industrial teams generally lack these capabilities. Therefore, having a no-code approach for using NS-3 would be an efficient way of reaching an important community of IoT professionals and make them able to benefit from NS-3 features. No-code [4] is becoming very popular in IoT, as it empowers manufacturers and operation managers to program their IoT applications while reducing the time and expertise needed.

For that reason, and in order to extend the use of NS-3 to a community of non-programmers, we propose, in this work, a no-code framework for users to set up and run NS-3 IoT networks simulations without writing a single script. We believe it can, on the one hand, expand the community of users and accelerate their IoT journey, and, encourage contributions to NS-3 towards further inclusion of more IoT technologies for industrial purposes on the other. The contributions of this work are the following:

- An intuitive web application to setup and run simulations by selecting and tuning scenario parameters.
- A set of relevant KPIs (Key Performance Indicators) for IoT simulations and their automatic calculation.
- A set of generic templates of NS-3 script for IoT use cases, and their implementation for Wi-Fi and LoRaWAN technologies, and guidelines on how they can be modified for other IoT networks.

The remainder of this paper is organized as follows: The problem formulation is established in Section 2, and an overview of our framework is given in Section 3. Section 4 first describes the developed templates, and then provides some integration guidelines for further contributions. Conclusion and future works are given in Section 5.

2 PROBLEM FORMULATION

The objective of this section is to propose a comprehensible way of defining an IoT scenario and the targeted output metrics that a user wants to gather using simulation. To do so, we need to clearly state what must be taken into consideration when running an IoT network simulation: the input parameters that define a scenario, and the output metrics that need to be gathered for evaluating the performance. To illustrate this, we consider the case wherein an IoT solution provider, offering a smart water management service based on LoRaWAN, needs to deploy a private network for a customer. One of the main questions that could be asked in this case is how robust will the network be, considering the radio parameters and the topology (number of sensors, their location, etc.). In other words, the IoT company has to know how much percentage of packets will successfully be transmitted, without omitting the fact that the minimum required packet delivery rate for such application is typically around 90% [2]. A way of answering the question would be to deploy the network and evaluate its performance. However, knowing that one LoRaWAN gateway can handle at least dozens of sensors, deploying them to answer the question can turn out to be very costly. Thus, using simulation instead would make it possible to answer that question while lowering costs (such an application has been studied in [5] using NS-3). As we can see, two aspects need to be defined for running such IoT scenario simulation: The scenario description, in terms of traffic and topology (e.g., the number, location and data rate of smart water sensors in the previous example), and the KPIs that need to be analyzed and will give insights to answer the question (e.g., the packet delivery ratio). We describe these two aspects in the following.

2.1 Scenario description

A scenario is defined by a list of parameters representing the network topology, the considered IoT network technology and the traffic specifications. They can be divided as follows: (1) the number of end-devices and their location, (2) the number of gateways and their location, (3) the IoT network technology defined by its physical and mac layers, (4) the low-level parameters related to the radio channel and the propagation model, the frequency and bandwidth of the radio channel and (5) the traffic type and workload (defined by the packet size and the inter-packets period).

The IoT traffic types can be classified according to (i) their direction: upstream (from end-devices to gateways or the cloud) or downstream (from the cloud or gateways to end-devices) and (ii) their profile: periodic or stochastic. We call periodic the traffic with a fixed data rate, while the traffic with a variable rate is referred to as stochastic. Although some applications have bidirectional traffic, the majority of IoT applications have upstream traffic. Table 1 lists the different IoT traffic types. We illustrate each traffic type with a possible application. Figure 1 depicts a classical IoT system architecture where the end-devices can either be sensors or actuators, depending on the traffic direction, upstream or downstream respectively.

Traffic type	Traffic profile	Traffic direction	Example
1	Periodic	Upstream	Smart metering
2	Periodic	Downstream	Webcast
3	Stochastic	Upstream	Video Surveillance
4	Stochastic	Downstream	Notifications or remote commands

Table 1: Traffic types characteristics.

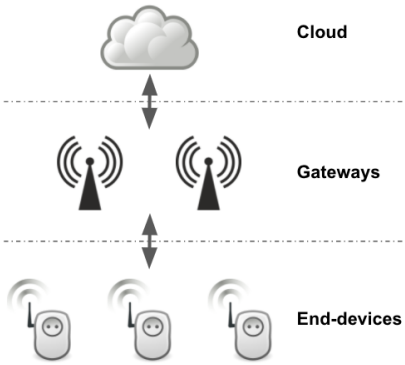

Figure 1: Classical high-level overview of an IoT architecture.

2.2 KPIs

We propose to gather five metrics, which together provide a fair representation of the performance of an IoT network technology for a given scenario. These parameters are: (i) packet throughput, (ii) packet latency, (iii) packet delivery, (iv) energy consumption and (v) battery lifetime.

Packet throughput, packet latency and packet delivery are common performance parameters in network performance evaluation. Packet throughput represents the data rate delivered to each IoT device or gateway. Packet latency is the time a packet takes to transit from its source to its destination. Packet delivery is the ratio (percentage) of successfully received out of all the packets sent.

Energy is extremely important in the IoT industry where end-devices are often equipped with a battery, and thus have a limited power supply. Energy consumption represents the amount of energy consumed during a period of time. It can be measured for the overall network or on each IoT end-device, in joules. The battery lifetime gives an indication on the IoT device's autonomy without recharging its battery.

3 FRAMEWORK OVERVIEW

In this section, we present our no-code simulation framework. We begin by describing its architecture, then we show how to use it through a web platform by providing an example of application.

3.1 Architecture

The architecture of our framework (Fig. 2) consists on an NS-3 environment where the simulations are executed, a web platform which serves as a user interface for entering scenario parameters and displaying KPIs, and a database to store both scenarios and KPIs. We describe each component in what follows:

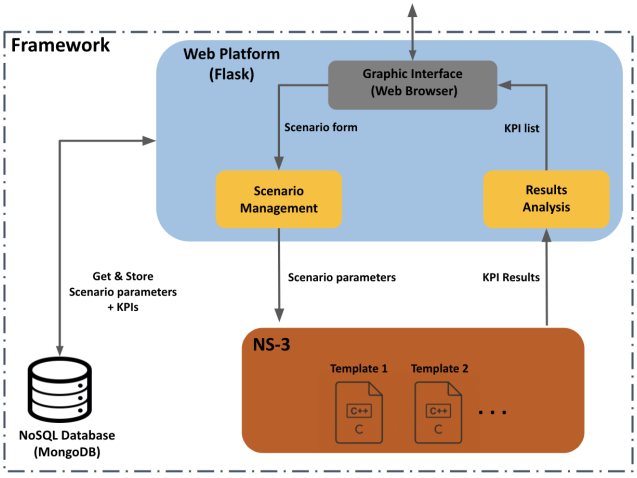

Figure 2: Framework architecture.

- **Web platform**: It is used to enter the scenario parameters via a form, specific to each scenario traffic type and IoT technology. The form contains a complete list of the traffic related parameters such as the packet size, the distance between gateways and end-devices, the data rate, etc. and low-parameters such as channel bandwidth, transmitting power, etc. A process of validation of the entered values, in terms of data ranges and types, is done before moving to the simulation step. The web platform is also used to display the list of KPIs returned from NS-3 after the end of the simulation. The web application has been developed using Flask [3], which is a Python-based web development framework.
- **NS-3**: Once the parameters have been entered by the user and validated by the engine, they are passed from the webapp on to NS-3 (which can be hosted in a virtual machine) through a command line. Depending on the chosen IoT technology, an NS-3 script, which we call simulation template, will be executed with the passed input parameters. Once the simulation is over, the KPIs returned by the template are passed back to the web platform, to finally be displayed through the user interface to the user. The version of NS-3 which is used in the current SIFRAN software is ns-3.33.
- **Database**: As users may need to get access to their previous simulations, we store both scenario parameters and the resulting KPIs in a database. Note that users have to create an account on the web platform beforehand if they want to store their simulations and KPIs and have access to them. We have opted to a NoSQL database using MongoDB [1], which is a document-oriented database program.

3.2 Usage

We illustrate in the following section the usage of the platform. From the homepage, users have the possibility to create an account through the "Register" button, which will give them access to their previous simulations. After that, they can directly fill a new IoT scenario form, either by assigning values to each parameter, or by selecting a preset which holds a set of predefined ones. As said before, all the parameter values are validated in terms of data type and range before the form is submitted. Once the form is correctly filled, the input parameters are sent to the NS-3 environment to be executed. The results of the simulation (KPIs) are calculated at that level before being sent back to the platform, which finally displays them. An example of a scenario form and simulation results is shown in Figure 3.

4 TEMPLATE DESCRIPTION AND INTEGRATION GUIDELINES

In this section, we show how to implement a template for simulating an IoT network scenario, then we give some integration guidelines on how to contribute to this framework for other IoT network technologies.

4.1 Template Description

We call a template the translation of an IoT scenario in the NS-3 environment language. It consists of C++ code globally working as follows: i) take input parameters which define the scenario, ii) create the corresponding nodes and traffic, iii) calculate the KPIs obtained from the simulation.

We considered two IoT technologies in our templates: Wi-Fi and LoRaWAN. The Wi-Fi stack is completely implemented in the official release of NS-3. Even though different Wi-Fi amendments are available, we focused on the 802.11ac and 802.11ax amendments, as they are the most recent ones.

Regarding LoRaWAN, its stack is not implemented in the official release of NS-3. However, a link to a public LoRaWAN module [6] is provided in the official website of NSNAM. The steps for installing this module are provided in the link.

Even if we implemented one template per IoT technology, the structure of both is almost identical. We describe bellow the template implemented for Wi-Fi technology, by giving screenshots of code:

(1) **Input parameters definition**: All the scenario parameters mentioned in Section 2 are set here. They take as values the parameters filled by users through the scenario forms shown in the previous section. They include both traffic and low-level parameters. Clearly, the considered parameters will most likely differ depending on the implemented IoT technology.

```
/* Input parameters definition */
// Simulation time in seconds
double simulationTime = 10;
// Number of end-devices
```

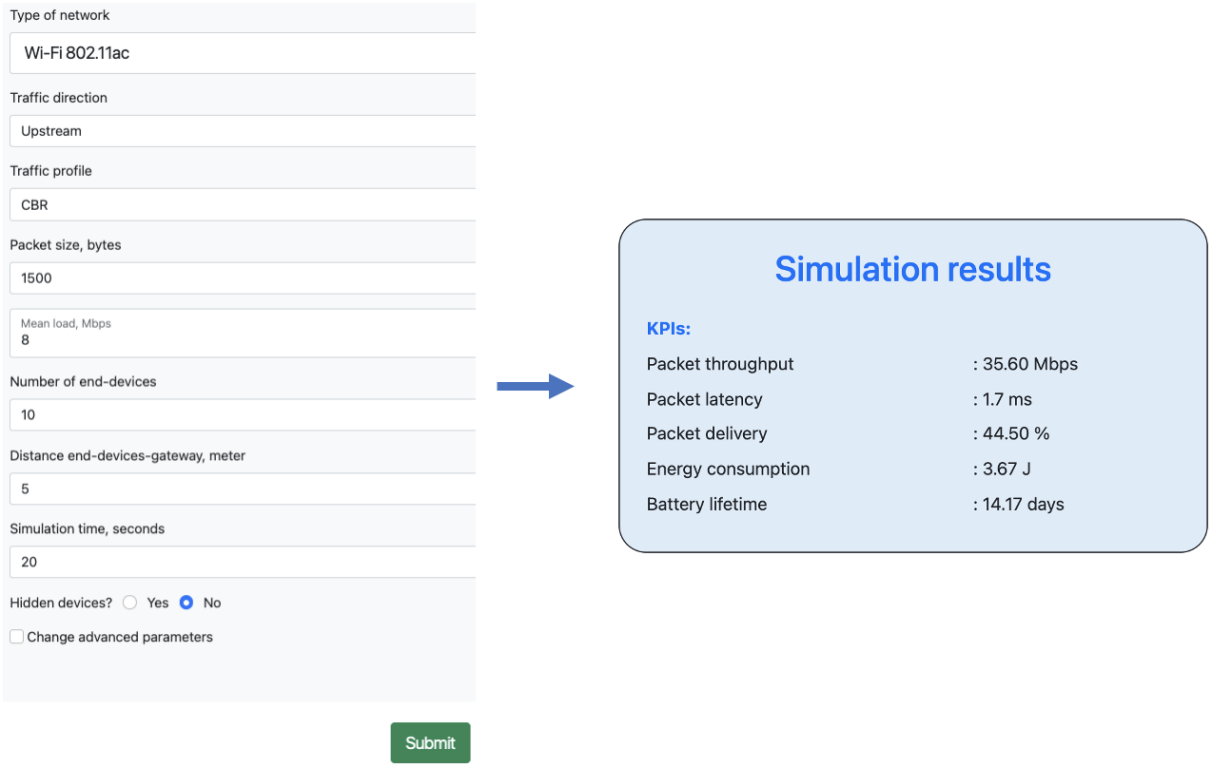

Figure 3: Application homepage.

```
uint32_t nWifi = 10;
std::string trafficDirection = "upstream";
// Payload size in bytes
uint32_t payloadSize = 1024;
// Packet period in seconds
std::string period = "1";
// Distance between AP and end-devices in meters
double distance = 1.0;
// Delay propagation model
std::string propDelay =
    "ConstantSpeedPropagationDelayModel";
// Loss propagation model
std::string propLoss = "LogDistancePropagationLossModel";
// Channel bandwidth in MHz
int channelWidth = 80;
// Indicates whether Short Guard Interval is enabled or
    not
int sgi = 0;
// Allow or not the packet agregation
bool agregation = false;
// Modulation and Coding Scheme
uint32_t MCS = 0;
// Transmitting power in dBm
uint32_t txPower = 9;
// Number of spatial streams
int spatialStreams = 1;
// Tx current draw in mA
double txCurrent = 107;
// Rx current draw in mA
double rxCurrent = 40;
// CCA_Busy current draw in mA
double ccaBusyCurrent = 1;
// Idle current draw in mA
double idleCurrent = 1;
```

Listing 1: Input parameters definition.

(2) **Nodes placement**: This part of code creates all the nodes (end-devices and gateways) using the `NodeContainer` object, and places them in three dimensional space, using the `ConstantPositionMobilityModel` object.

```
/* Positioning Nodes */
for (uint32_t i = 0; i < nWifi; i++) {
    positionDevices->Add (Vector (distance, 0.0, 0.0));
}

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

Ptr<ListPositionAllocator> positionAp =
    CreateObject<ListPositionAllocator> ();
positionAp->Add (Vector (0.0, 0.0, 0.0));
mobility.SetPositionAllocator (positionAp);
mobility.SetMobilityModel
    ("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNode);
```

Listing 2: Nodes creation & placement.

(3) **Layers configuration**: The technology is defined here by setting the `YansWifiPhyHelper`, `WifiMacHelper` and `WifiHelper` objects as the physical, mac and network layers for Wi-Fi nodes. For LoRaWAN, the `LoraPhyHelper`, `LorawanMacHelper` and `LoraHelper` objects are used for the physical, mac and network layers. The Wi-Fi amendment is also specified here with the `SetStandard ()` method.

```
/* Layers installation */
YansWifiPhyHelper phy;
phy.SetChannel (channel.Create ());
```

```
WifiMacHelper mac;
WifiHelper wifi;
wifi.SetStandard (WIFI_STANDARD_80211ac);

std::ostringstream oss;
oss << "VhtMcs" << MCS;
wifi.SetRemoteStationManager
      ("ns3::ConstantRateWifiManager", "DataMode",
      StringValue (oss.str ()), "ControlMode",
      StringValue (oss.str ()));

Ssid ssid = Ssid ("ns3-80211ac");

// Installing phy & mac layers on the end-devices
mac.SetType ("ns3::StaWifiMac", "Ssid", SsidValue (ssid));
NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac,  wifiStaNodes);

// Installing phy & mac layers on the AP
mac.SetType ("ns3::ApWifiMac", "EnableBeaconJitter",
      BooleanValue (false), "Ssid", SsidValue (ssid));
NetDeviceContainer apDevice;
apDevice = wifi.Install (phy, mac, wifiApNode);
```

Listing 3: Layers configuration.

(4) **Low-level parameters configuration**: The low-level parameters which have been declared on Listing 1 such as the short guard interval, the bandwidth, the spreading factor for LoRaWAN, etc. are instantiated and set at the nodes level here.

```
/* Low-level parameters configuration */
// Set channel width
Config::Set
      ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/
       ChannelWidth", UintegerValue (channelWidth));

// Set guard interval
Config::Set
      ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/
        HtConfiguration/ShortGuardIntervalSupported",
            BooleanValue (sgi));

// Set txPower in the end-devices
for (uint32_t index = 0; index < nWifi; ++index) {
  Ptr<WifiPhy> phy_tx = dynamic_cast<WifiNetDevice*>
       (GetPointer ( (staDevices.Get(index))))->GetPhy();
  phy_tx->SetTxPowerEnd(txPower);
  phy_tx->SetTxPowerStart(txPower);
}

// Set txPower in the AP
Ptr<WifiPhy> phy_tx = dynamic_cast<WifiNetDevice*>(
  GetPointer((apDevice.Get(0))))->GetPhy();
phy_tx->SetTxPowerEnd(txPower);
phy_tx->SetTxPowerStart(txPower);
```

Listing 4: Low-level parameters configuration.

(5) **IP address configuration**: In case the IP addresses are supported in the nodes (not supported in LoRaWAN), we configure them in this part using the `Ipv4AddressHelper`, in order to make the nodes accessible to each other.

```
/* IP addresses configuration */
Ipv4AddressHelper address;
address.SetBase ("10.0.0.0", "255.255.0.0");
Ipv4InterfaceContainer apInterface;
apInterface = address.Assign (apDevice);
Ipv4InterfaceContainer staInterface;
staInterface = address.Assign (staDevices);
```

Listing 5: IPV4 addresses configuration.

(6) **Application traffic specification**: This part is where the traffic definition is made. Depending on the traffic type, applications are defined and installed in the nodes (as stated in Table 2), with fixing the destination address. We detail this process in what follows:

- **Periodic**: For simulating a periodic traffic, we install the `UdpClient` and `UdpSocket` objects in the sender and the receiver nodes respectively. The needed parameters for the `UdpClient` object are the packet period and the packet size, which are specified by the user. The implementation of such a traffic is given below. For LoRaWAN, the `PeriodicSenderHelper` object is used with setting the period and the packet size with the `SetPeriod ()` and `SetPacketSize ()` methods respectively. Since LoRaWAN does not allow communications with big data rates, we can only simulate periodic traffics with relatively low data rates.
- **Constant Bit Rate**: The difference between this traffic and the previous one is that the parameter which is specified is the data rate (in Mbps or bps) instead of the packet period. In some cases, it may be simpler for the end user to express the application needs in terms of data rate than the packet period. The objects used in this case are the `OnOff` and `UdpSocket`. We need to specify in this case the application data rate (in Megabits per second) which is a parameter of the `OnOff` object.
- **Variable Bit Rate**: For this kind of traffic, since packets can have different sizes and periods, we generate them using random variables (X for the packet size and Y for the packet period) following Normal laws with mean and variance defined by user. Thus, we use in this case a function which takes as a parameter a `Socket` object, and which schedules for every realization of X a sending of a packet which size is a realization Y.

Traffic type	Client application	Server application	Fixed parameter
Periodic	UdpClient	UdpSocket	Period
CBR	OnOff	UdpSocket	Data rate
VBR	Socket	UdpSocket	Trace file

Table 2: Wi-Fi traffic types implementation.

```
/* Setting traffic applications */
ApplicationContainer sourceApplications, sinkApplications;
uint32_t portNumber = 11;
double min = 0.0;
double max = 0.5;
double periodSeconds = std::stof(period)
if (trafficDirection == "upstream") {
  auto ipv4 = wifiApNode.Get(0)->GetObject<Ipv4> ();
  const auto address = ipv4->GetAddress (10).GetLocal
       ();
  InetSocketAddress sinkSocket (addressportNumber);
  PacketSinkHelper
       packetSinkHelpe("ns3::UdpSocketFactory",
       sinkSocket);
  sinkApplications.Ad(packetSinkHelper.Install
       (wifiApNode.Get(0)));

  for (uint32_t index = 0; index < nWifi; ++index) {
    if (agregation == false) {
    // Disable A-MPDU & A-MSDU in ea  station
    Ptr<NetDevice> dev = wifiStaNodes.G
         (index)->GetDevice (0);
    Ptr<WifiNetDevice> wifi_dev
         DynamicCast<WifiNetDevice> (dev);
    wifi_dev->GetMac ()->SetAttribu
         ("BE_MaxAmpduSize", UintegerValue (0));
    wifi_dev->GetMac ()->SetAttribute
         ("BE_MaxAmsduSize", UintegerValue (0));
```

```
  }

    // UDP Client application to be install  in the
         stations
    UdpClientHelper echoClient(addres  portNumber);

    echoClient.SetAttribute("MaxPackets",
         UintegerValue(100000));
    echoClient.SetAttribute("Interval",
         TimeValue(Seconds(periodSeconds)));
    echoClient.SetAttribute("PacketSize",
         UintegerValue(payloadSize));

    // Desynchronize the sending applications
    Ptr<UniformRandomVariable> x =
         CreateObject<UniformRandomVariable> ();
    x->SetAttribute ("Min", DoubleValue (min));
    x->SetAttribute ("Max", DoubleValue (max));

    double value = 1 + x->GetValue ();

    ApplicationContainer sourceApplications =
         echoClient.Install (wifiStaNodes.Get(index));
    sourceApplications.Start(Seconds(value));
    sourceApplications.Stop(Seconds(simulationTime+2));
  }
}
```

Listing 6: Wi-Fi periodic traffic specification.

(7) **Energy configuration**: To keep trace of the consumed energy during the simulation, an energy source and a draining model have be configured on nodes. The energy source can be seen as a battery from where the energy is drained from. We use for that the `BasicEnergySourceHelper` object that drains energy in a linear way. A non-linear draining can be also be considered and implemented using the `RVBatteryModelHelper` object (more details about this can be found in [7]). `WifiRadioEnergyModelHelper` and `LoRaRadioEnergyModelHelper` objects are used for the draining models of the two technologies. They are two models both based on state-machines which assign to each physical state a current draw consumption in milliamperes for Wi-Fi and LoRaWAN. An example of the Wi-Fi model is given in Listing 7.

```
/* Installing energy models */
DeviceEnergyModelContainer deviceModels;

double capacityJoules = (batteryCap / 1000.0) * voltage *
     3600;

WifiRadioEnergyModelHelper radioEnergyHelper;

radioEnergyHelper.Set ("IdleCurrentA", DoubleValue
     (idleCurrent));
radioEnergyHelper.Set ("TxCurrentA", DoubleValue
     (txCurrent));
radioEnergyHelper.Set ("CcaBusyCurrentA", DoubleValue
     (ccaBusyCurrent));
radioEnergyHelper.Set ("RxCurrentA", DoubleValue
     (rxCurrent));


BasicEnergySourceHelper basicSourceHelper;
basicSourceHelper.Set  ("BasicEnergySupplyVoltageV",
     DoubleValue  (voltage));
basicSourceHelper.Set
     ("BasicEnergySourceInitialEnergyJ", DoubleValue
     (capacityJoules));
EnergySourceContainer sources =
     basicSourceHelper.Install(wifiStaNodes);

deviceModels = radioEnergyHelper.Install  (staDevices,
     sources);
```

Listing 7: Energy configuration.

(8) **Trace files generation**: There is the possibility in NS-3 of generating pcap (Packet Capture) and trace files which contain all the packets that have flowed through the network. It can be done using the `AsciiTraceHelper` object for some IoT technologies. To the best of our knowledge, there is no tracing system (neither pcap nor trace files) proposed using the LoRaWAN module. It is worth noting that pcap files can be opened by software like Wireshark, while the trace files can be read using any text editor.

```
/* Traces files generation */
AsciiTraceHelper ascii;
phy.SetPcapDataLinkType
     (WifiPhyHelper::DLT_IEEE802_11_RADIO);
std::string s = "trace";
phy.EnableAsciiAll (ascii.CreateFileStream(s+".tr"));
phy.EnablePcap (s+".pcap", apDevice.Get(0), false, true);
```

Listing 8: Traces files generation.

(9) **KPIs calculation**: At the end of the template, we gather all the wanted KPIs from our simulation, as the following:

- **Packet Throughput**: For this KPI, the `GetTotalRx ()` and `CountMacPacketsReceived ()` methods are used for Wi-Fi and LoRaWAN respectively. Both methods return the amount of bytes received by a node. This value is converted and divided by the simulation time to get the throughput, in Mbps.
- **Packet Delivery**: The way to get the ratio of successfully received of data over the total amount sent differs according to the traffic type. In case it is periodic, we can simply get it by dividing the simulation time by the packet period. This will give us the number of received bytes, which we divide by the number of sent ones to get the packet delivery. For the CBR case, in order to have a precise metric, we added an attribute in the `OnOff` application that contains the exact number of sent bytes. Then, we just divide it by the same `GetTotalRx ()` method used for the packet throughput. Finally, if the traffic is VBR, we keep trace of the number of sent bytes in a variable incremented with each sending corresponding to a realization of the X random variable during the simulation. Then it is just divided by the results returned by the same `GetTotalRx ()` method.

```
/* Calculating packet throughput and packet delivery */
totalPacketsThrough = DynamicCast<PacketSink>
     (sinkApplications.Get (0))->GetTotalRx ();
throughput += ((totalPacketsThrough * 8) /
     ((simulationTime) * 1024 * 1024)); // Mbps
std::cout << "Packet Throughput: " << throughput <<
     std::endl;
double successRate =  (totalPacketsThrough /
     totalpacketsSent / nWifi) * 100;
std::cout << "Packet Delivery: " << successRate <<
     std::endl; // %
```

Listing 9: Packet throughput and delivery calculation.

- **Packet Latency**: If the traffic is relatively low, we can get it directly using the logging system of the simulator, for each sent packet. In case the traffic is important, there may be overloaded buffers in the sending nodes, which will increase latency. It would be of benefit to get rid of this problem since the latency in this case would more depend on the buffer sizes than on the network state. A

way of doing so and getting a representative value of the latency is by adding a probing node in the network, which only sends data periodically in the same direction as the other nodes in the network, and get the latency only from the packets sent by this node. This allows us to avoid the queue time in the nodes buffers. The objects we install at the probing end-device and the gateway respectively are the `UdpEchoClient ()` `UdpEchoServer ()` which print the times of sending and arrival of packets.

- **Energy consumption**: The energy consumption is obtained using the `GetTotalEnergyConsumption ()` method of the energy model which returns the total amount of consumed energy, in joules. This method is called at the end of the simulation, for one end-device, since we consider that all of them have the same behaviour.
- **Battery Lifetime**: The battery lifetime is directly derived from the energy consumption, by dividing the capacity of the battery (in Joules) by the energy consumed, which gives us the number of simulations of same length that can be supported by the battery. We then multiply it by the simulation time to get how much will the battery last in seconds.

```
/* Calculating Energy KPIs */
double energy = 0, battery_lifetime = 0;
DeviceEnergyModelContainer::Iterator iter ;
for (iter = deviceModels.Begin (); iter !=
     deviceModels.End (); iter ++) {
  double energyConsumed =
       (*iter)->GetTotalEnergyConsumption ();
  NS_LOG_UNCOND ("End of simulation (" <<
       Simulator::Now ().GetSeconds () << "s) Total
       energy consumed by radio (End-device) = " <<
       energyConsumed << "J");
  std::cout << "Total energy consumed by radio
       (End-device): " << energyConsumed << std::endl;
  battery_lifetime = ((capacityJoules /
       energyConsumed) * simulationTime);
  battery_lifetime = battery_lifetime / 86400; // Days
  std::cout << "Battery lifetime: " <<
       battery_lifetime << std::endl;
  energy = energyConsumed;
  break; // Energy in only one station
}
```

Listing 10: Energy KPIs calculation.

4.2 Integration guidelines: Example with 6LoWPAN

We now present guidelines to the community for contributing to SIFRAN by writing new templates in order to enhance it with more IoT network technologies. Overall, the structure of the templates remains the same but obviously some parts need to be updated due to the peculiarities of the newly considered technology. Table 3 summarizes the guidelines for each defined and labelled portion of code. Table 3 also shows how to implement templates for the short range IoT technology 6LoWPAN (based on 802.15.4 standard).

5 CONCLUSION AND FUTURE WORKS

In this work, we have presented SIFRAN, a no-code framework with the objective of enabling IoT simulation through NS-3 without coding. We began by clearly identifying the most salient aspects that need to be taken into consideration for simulating an IoT scenario, and the required KPIs for the network performance evaluation. Then, we detailed the architecture of SIFRAN which consists of a web application, a database and NS-3 templates. The latter were illustrated with the example of a Wi-Fi template and a LoRaWAN one. We then provided guidelines to the community in the hope that new IoT network modules will be developed in NS-3 and then incorporated in SIFRAN. An example of how to proceed with the example of 6LoWPAN has also been given.

The next step is to share the SIFRAN framework with IoT user communities such as the NS-3[1] Group and the FIT IoT-Lab[2] in order to gather feedback from them.

In terms of future works, we plan to provide the following enhancements:

- Refine the web application to make it more user friendly, taking into account feedback from the user community.
- Extend the list of supported technologies with additional NS-3 templates.
- Extend SIFRAN to let it handle scenarios with multiple gateways.
- Explore range of values for a given parameter to appraise its influence over a KPI. One could for instance see the influence of the number of end-devices on the battery lifetime.

A first version SIFRAN has just been made publicly available at https://sifran.labs.stackeo.io/ and we hope that it will attract contributions from other developers.

ACKNOWLEDGMENT

This work was performed within the framework of the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR) and the technical support of Stackeo (https://stackeo.io).

REFERENCES

[1] Chodorow, K. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[2] Feltrin, L., Buratti, C., Vinciarelli, E., De Bonis, R., and Verdone, R. Lorawan: Evaluation of link- and system-level performance. *IEEE Internet of Things Journal 5*, 3 (2018), 2249–2258.

[3] Grinberg, M. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.", 2018.

[4] Korkan, E., Regnath, E., Kaebisch, S., and Steinhorst, S. No-code shadow things deployment for the iot. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)* (2020), pp. 1–6.

[5] Lalle, Y., Fourati, L. C., Fourati, M., and Barraca, J. P. A Comparative Study of LoRaWAN, SigFox, and NB-IoT for Smart Water Grid. *2019 Global Information Infrastructure and Networking Symposium, GIIS 2019* (2019).

[6] Magrin, D., Centenaro, M., and Vangelista, L. Performance evaluation of LoRa networks in a smart city scenario. *IEEE International Conference on Communications* (2017).

[7] Rakhmatov, D. N., and Vrudhula, S. B. An analytical high-level battery model for use in energy management of portable electronic systems. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers* (2001), 488–493.

[1] https://groups.google.com/g/ns-3-users
[2] https://www.iot-lab.info/community/publications/

Code label	Required changes	Example with 6LoWPAN
1	Traffic parameters remain the same, while low-level parameters change depending on the chosen IoT technology.	Declare here parameters such as the Perimeter Area Network (PAN) id.
2	No changes.	/
3	The layers must change since it is precisely here that the IoT technology layers are specified.	Use the `LrWpanHelper` for the phy and mac layers (802.15.4 norm) and the `SixLowPanHelper` for the network layer.
4	The low-level parameters which are specific to the IoT technology are set here.	The ID of the PAN can be set here using the `AssociateToPan ()` method.
5	This part may remain the same in case the target IoT technology supports IPV4 addresses for the nodes. It can also change if the IP layer is not supported, or in IPV6 must be used instead.	Use the `Ipv6AddressHelper` object for the addressing process.
6	Depending on the traffic type, the same applications as for Wi-Fi can be used if they are supported.	The `OnOffHelper` and `UdpClientHelper` objects can also be used for 6LoWPAN.
7	The energy source in this part remains the same, but the draining model should correspond to the target IoT technology since each has its own PHY states and corresponding current draw consumption.	No energy model is implemented for 6LoWPAN in the official release of NS-3.
8	This optional part here may be unavailable for some IoT technologies due to the lack of implementation. We advice the community to refer to the Tracing system in the NS-3 documentation to get more details about this.	The `AsciiTraceHelper` also provides pcap and trace files. for 6LoWPAN
9	• **Packet throughput & Packet delivery**: Since the same application as for Wi-Fi can be used, the way of gathering packet throughput and packet delivery is identical. • **Packet latency**: As stated before, the probing mechanism can be used to get the packet latency in the case the traffic is not periodic. Otherwise, using the tracing or the logging system is sufficient. • **Energy consumption**: The energy source remains the same, while the model should correspond to the IoT technology which has its own PHY states and their current draw consumption. • **Battery lifetime**: Just as for the energy source, the way of calculating how long the battery will last in the scenario conditions is identical whatever the IoT technology is.	The same way of calculating KPIs as for Wi-Fi can be used for 6LoWPAN since the same traffic applications can be used.

Table 3: Integration guidelines.