Simulation of Distributed Systems in Constrained Environments Using ESDS: the Arctic Tundra Case

Loic Guegan*, Issam Raïs*
Department of Computer Science, UiT The Arctic University of Norway, Tromso, Norway*
Corresponding authors: {name.surname}@uit.no

***Abstract*—Studying Distributed Systems (DS) is important. They are used in many computer systems such as Internet of Things (IoT), Wireless Sensor Networks (WSN), Cyber-Physical Systems (CPS). Conducting research studies on DS is possible with simulation. However, depending on the research context, having access to the proper simulator is challenging. Existing simulation frameworks can be complex and not suitable to study DS outside of commonly seen contexts, such as systems deployed in constrained environments. Often, authors from state of the art rely on building their own simulator. This is a time consuming, delicate and dangerous approach, especially when such simulators are not validated.**

This paper presents the Extensible Simulator for Distributed Systems (ESDS) in the context of systems deployed in constrained environments. In our case, the Arctic Tundra. This simulation framework is simple to use and suitable for the study of DS. The architecture of the simulation framework is detailed at a fine grain level. Results show that, ESDS can be used to conduct energy consumption and network performance studies of distributed systems such as CPS.

***Index Terms*—Simulation, distributed systems, networking, energy consumption, Cyber-Physical System, arctic tundra, Internet Of Things, Wireless Sensor Networks**

I. Introduction

Distributed System (DS) is a widely used paradigm [1]. A DS comprises networks of communicating computers which can be wired, wireless or both, like in Fog infrastructures. The Internet of Things (IoT) [2], Wireless Sensors Networks (WSN) [3] and Cyber-Physical Systems (CPS) [4] are timely examples of DS. But these systems are complex, with numerous factors affecting computers (also called nodes) such as the network infrastructure quality, failures, battery depletion.

Being able to study DS to assess behaviors of the nodes, and the system as a whole is important. To conduct these studies, different approaches are used. One of them consists in performing test-bed experiments. This approach requires to have access to a test-bed and can be time consuming as the experiments need to be deployed on real nodes. Another approach is to build node prototypes. This is useful if no test-bed is available, but it is time consuming and potentially costly. Another approach to study DS is through simulation. It allows to save time and potentially money. Compared to test-bed and prototyping, simulations are not limited by any physical platform. It also permits to study systems in various conditions such as varying the network performance, the energy characteristics, the number of nodes and several use-case related parameters. However, having access to a simulation framework that fits with the intricacies of the studied use case can be challenging.

The Distributed Arctic Observatory (DAO) project comprises a group of researchers working on the use of DS to create a monitoring infrastructure for the Arctic Tundra (AT). To build this infrastructure, the DAO uses nodes that are interconnected by wireless communications. Because of the harsh weather conditions (temperatures, rain and snow), nodes are most of the time not able to communicate and be humanly accessible. Thus, nodes are expected to operate for a long time period, while being isolated and constrained in energy budget. Being able to perform in-depth studies of DS characteristics in the DAO context is crucial to ensure their compatibility.

In the literature, hundreds of simulators are available [5]. But having access to a simulation framework that is suitable for the DAO context is difficult. Existing frameworks are complicated, and most works related to the DAO fallback to prototyping [6], [7]. Other authors build a simulator specific to a given study [8]. Nevertheless, creating a simulator is time consuming and error prone. The implemented models must be carefully validated before being used in a research context.

The Extensible Simulator for Distributed Systems (ESDS) is an open source simulator written in Python, available online [9], with its wireless and power consumption model validated [10]. In this paper, we present the usage of ESDS to simulate distributed systems deployed in scarce resource environments. More specifically, we apply ESDS to the context of systems deployed in the Arctic Tundra, within the DAO.

This paper is organized as follow. Section II presents the arctic tundra use-case. Section III presents the approaches used in the literature to study DS in the DAO context. Section IV presents the architecture of ESDS, its models and API. Section V details the simulated scenarios used to evaluate the capabilities of ESDS in the DAO context. Section VI presents the results and Section VIII concludes the work.

II. Use-case

A. The Arctic tundra

The Arctic Tundra (AT) is one of the largest terrestrial biomes located at the Northern part of Earth [11]. It forms a circumpolar area on the north pole. The AT is characterized by its extremely cold temperatures with an annual mean around -15°C to 1.5°C. Several regions of the AT have a ground temperature that remains below 0°C, also known as permafrost. Despite these extremes conditions, the AT has a diverse ecosystem. It is divided into three regions (high Arctic, low Arctic and sub-Arctic). Each one has their own environmental characteristics and ecosystem [12].

The AT biome is sensible to climate changes. By the end of the century, its average temperature is expected to raise up to 10°C [13]. Significant changes on the vegetation are documented with an increase of the plants biomass, a phenomenon known as *Arctic greening*. Due to warmer weather conditions, the number of invasive species is expected to increase leading to a significant impact on the ecosystem. Other key indicators of the AT climate change are presented in [14]. It includes changes in the permafrost, the carbon cycling, precipitations, humidity etc. Observing these key indicators is crucial to measure, and forecast the impact of climate change.

B. The COAT initiative

The Climate-Ecological Observatory for Arctic Tundra (COAT) [13], is in charge of providing an observation system for the AT. COAT is an initiative for long-term and adaptive monitoring of the AT based on a food web approach. It focuses on the study of two Norwegian regions namely, the Low Arctic Varanger and the high Arctic Svalbard. To conduct their monitoring campaigns, scientists from COAT rely on several techniques [15]. They use technologies such as ground-based nodes, drones and satellites. These technologies allow for higher resolution compared to traditional on-site, human measurements. In particular, the use of ground-based nodes provides in-situ observations. Coupled to the hard weather conditions (low temperatures, snow, ice etc.), the design of such monitoring is challenging. Having access to ground-based nodes is difficult in such an environment. Nodes are expected to operate for a long time period and be energy efficient. Moreover, the AT provides low to no network coverage, making wireless communications challenging.

C. The Distributed Arctic Observatory

The Distributed Arctic Observatory (DAO) project tackles these issues. It comprises several interdisciplinary research groups that aims at providing the next generation of monitoring system for the AT. The DAO focuses on using Cyber-Physical Systems (CPS) as an in-situ monitoring infrastructure. Such an infrastructure aims at being energy efficient and offering a level of resiliency in communications. An overview of this monitoring infrastructure is presented in [16].

This architecture comprises Observation Nodes (ON) deployed in the Arctic Tundra. ONs are energy efficient monitoring nodes. They are reachable remotely using wireless technologies. They are accessible by DAO and COAT scientists, using a dedicated API. Scientists are able to send Unmanned Aerial Vehicles (UAV) on the deployment site to perform various operations on ON, such as energy replenishment.

A detail picture of an ON prototype is shown on Figure 1. An ON is based on Single Board Computers (SBC) that

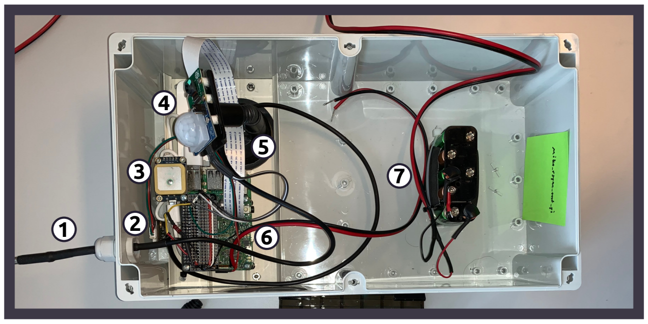


Fig. 1. Observation Nodes (ON) from the DAO project.

allow the use of a complete software development stack and more resources compared to micro-controller-based (MCU) ONs. The prototype on Figure 1 comprises a Raspberry-Pi SBC combined to a Sleepy-Pi MCU ⑥. Regarding sensors, this prototype provides optical and proximity cameras ⑤, temperature and humidity (inside ② and outside ①) and a GPS ③. From the Raspberry-Pi, several wireless technologies are available such as 4G ④, Wi-Fi, Bluetooth, 4G LTE and LTE Cat M1. ONs are battery-powered ⑦. Nodes software are implemented in Go and Python.

Using well-known hardware and software technologies allows to build a monitoring system for the AT relatively easily with commonly available Information Technology (IT) resources. From the deployment of such ONs, practical experiences can be derived [6]. However, prototyping and deployments of ONs are time consuming. Having a simulation environment for the DAO can greatly improve the research.

III. State of the Art

This section presents existing simulation works in the DAO context and in general, alongside the motivation of this work.

A. Simulation for scarce resource environments

For scarce resource environments, it is crucial to evaluate contributions by performing experiments. Two approaches are possible: building prototypes [6], [7], [16], [17] or performing simulations [8], [18]–[20]. Creating a prototype allows for accurate studies. But it is time consuming and costly. For these reasons, simulation is often used as an alternative (or combined [21]) to prototyping.

In [20], the authors propose to study the use of WSN for animal tracking in a scarcely resourced environment. This works includes a study the ON/OFF strategy to increase the nodes lifespan. To perform an energy consumption study of this strategy, an ad-hoc simulator is built using C++. In [19], a reduction of image dimension, taken from battery-powered devices, is proposed. An ad-hoc simulator is used to estimate the energy consumption of the devices and model the network performance. In [8], an alternative to LoRaWaN called LoRaLitE is proposed. Similarly to previous works, an ad-hoc simulator is used to perform DS simulations. In [22], a study of the impact of images compression on the CNN classification performance in the DAO context is proposed. The impacts of image compression on a monitoring system deployed in the AT are not covered. Performing this study using an adapted simulator can greatly improve the contribution with a small implementation time overhead. Finally, in [18], a study of the impact of loosely coupled data dissemination on nodes energy consumption and dissemination efficiency is proposed. The experiments are conducted with a simulator for distributed systems called SimGrid. The simulator implemented by authors provided in[1] is complex as SimGrid is not primarily design for studying wireless DS. This suggests that, a significant amount of time must be invested into understanding the simulation framework, to implement scenarios from the DAO context.

Discussed papers show that the current state of the art related to simulating scarcely resourced environment can be greatly improved by using the proper simulation tool.

B. Simulation models and frameworks

Most state of the art simulators fall into the following categories [23]: 1) Packet-level 2) Flow-level, which significantly impacts the simulator performance and accuracy. It also constrains its domain of application.

A packet-level simulator (e.g: ns-3 [24], OMNET++/INET [2] etc.) is able to simulate networks at a fine grain. It reflects the impact of network protocols and the physical layer of wireless communications. Since they provide fine-grained models, they are slower compared to flow-level simulators. They are not suitable for large-scale network studies. Configuring and calibrating a packet-level simulator requires advanced network knowledge. It can be time consuming and error prone due to the granularity of the models and the complex simulators API.

Flow-level simulators (e.g: SimGrid, FLEO [25] etc..) were introduced to mitigate these issues. They aim at simulating large-scale networks with coarse-grained models. Despite using these types of models, their accuracy is sufficient for numerous domain of applications such as Cloud [26], [27], HPC [28], [29], P2P [30], FoG. Flow-level simulators tend to extend their models granularity towards the one previously exclusive to packet-level simulators [31]. Simplicity is another key aspect of flow-level simulators, as they are using coarse grained models, which are simpler to instantiate and less error prone compare to packet-level simulators.

Despite the large amount of existing simulators [5], none of them is particularly used and adapted to model scenarios from the DAO-CPS.

C. Motivations

Packet-level simulators provide programming frameworks that are heavy and difficult to control when modeling complex scenarios. To simplify the programming interface, simulators such as CupCarbon [32], use DSLs as a programming language for nodes implementation. In the DAO context, this is limiting as nodes are implemented with programming languages such as Python or Go. Thus, relying on existing packet-level simulators to study the DAO is fairly complicated.

Flow-level simulators offer simpler programming interfaces, allowing control over the entire simulation environment and parameters. However, to the best of our knowledge, no flow-level simulators from the literature is designed for the simulation of wireless distributed systems while supporting simulation of extremely constrained scenarios. Implementing wireless DS with existing simulators is thus challenging.

This work proposes a simulation study of a ONs deployment in the DAO-CPS context. This study uses ESDS, proposed and validated in [10]. This simulator aims to be simple, modular and intuitive. It provides flow-level models for the simulation of DS. The remaining of the work presents its intricacies and its applicability to constrained environments, such as the DAO.

IV. Architecture of ESDS

This section presents the architecture of ESDS. Its components, models, API and plugins mechanism are detailed.

A. Components

The ESDS simulator comprises two major components: the simulated nodes (SN) and the Simulation Orchestrator (SO). The SNs are agents that runs concurrently. Their implementations are provided by the user and reflect the simulated scenario. The SO is in charge of coordinating the SNs execution.

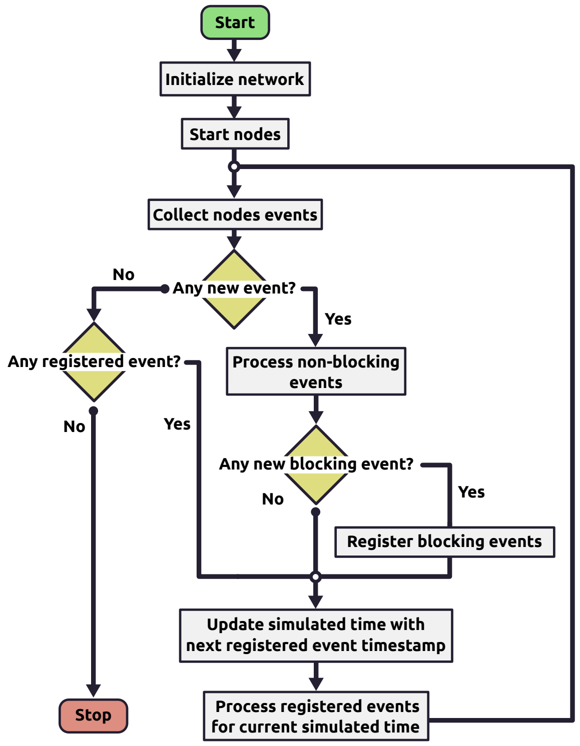


Fig. 2. Flow diagram of the ESDS Simulation Orchestrator (SO).

Figure 2 depicts the SO flow diagram. At the beginning of the simulation, the SO initializes the network settings provided by the user. These settings contain networks parameters used

[1]https://gitlab.com/manzerbredes/loosely-coupled-dss
[2]https://inet.omnetpp.org/

TABLE I
ESDS API CALLS AVAILABLE IN THE SIMULATED NODES

Call	Blocking	Description
`send()`	yes	Send data
`sendt()`	yes	Timed out send data
`receive()`	yes	Wait for and fetch incoming data
`receivet()`	yes	Timed out wait for and fetch incoming data
`wait()`	yes	Wait for a specific amount of simulated time
`wait_end()`	yes	Wait until the end of the simulation
`log()`	no	Print a message in the SO standard output
`read()`	no	Read in the SO current state
`turn_on()`	no	Turn the node on
`turn_off()`	no	Turn the node off

by SNs during the simulations (e.g, bandwidth, latency, nodes reachability). Next, the SO starts SNs execution by running the implementations written by the user. Then, the SO enters the simulation main loop and performs the following actions: 1) Collects the events generated by the SNs 2) Processes the non-blocking events (the ones that can be processed immediately such as log events) 3) Registers the blocking events (the ones processed at later simulated time such as communications events) 4) Processes the events for the current simulated time 5) Updates the simulated time accordingly. This main loop is repeated until no more events are generated by SNs.

B. Wireless network model

As in real computer networks, SNs can be part of several networks through different interfaces. Each of them with their own characteristics. In ESDS, a network is characterized by two matrix: a latency matrix $\boldsymbol{L}$ and a bandwidth matrix $\boldsymbol{B}$. Similarly to [33], the duration T_c of a communication c from SN i to SN j, transmitting n bytes of data is defined as:

$$T_c = \frac{n}{\boldsymbol{B}_{(i,j)}} + \boldsymbol{L}_{(i,j)} \tag{1}$$

In the case where $\boldsymbol{B}_{(i,j)} = 0$, the SNs i and j are considered unreachable. A set of validation experiments for this wireless communication model implemented in ESDS is available in [10]. By default, ESDS does not provide models for network protocols such as TCP/UDP. The aims is to give the user control over the data that are transmitted during communications. But, these protocols can be implemented on the SNs using SN plugins, as detailed later. The model defined by Equation 1, allows to simulate the majority of scenarios from the DAO, and more generally, from CPS networks.

C. Simulated Nodes: API

SNs implementations provided by the user follow a common API. This API offers entry points to interact with ESDS. Table I reports all the available blocking and non-blocking API calls. The following gives details about their intricacies:

a) Communications: send() and *receive*() are used by the SNs to communicate with each other. When a call to *send*() occurs, a communication from the calling SN to the receiver(s) is created by the SO by registering a new communication event. When the communication ends, the data transmitted by the sender are added to the receiver(s) queue. Data can then be accessed by the receiver(s) via the *receive*() call. Calls *sendt*() and *receivet*() are equivalent, but bounded with a timeout.

b) Wait calls: wait() call allows SNs to wait for a specific amount of simulated time. This call is crucial to implement scenarios with actions occurring at specific times. *wait_end*() allows SNs to wait for the end of the simulation.

c) Logging: log() call prints a message in the SO standard output. It allows the user to report various information during the simulation.

d) Access to the SO internal state: read() call provides access from the SN, to the SO internal variables. It can be used to read the current simulated time (e.g: `read("clock")`).

e) Node operating state: Nodes can switch between the on and off state using the *turn_on*() and *turn_off*() calls. This affects the reachability of SNs and its energy consumption.

D. Simulated Nodes: plugins

SN plugins allow to implement reusable features, usable by all SNs during the simulation. Network communication protocols are examples of features that can be implemented as SN plugins. Each SN can have its own set of active plugins.

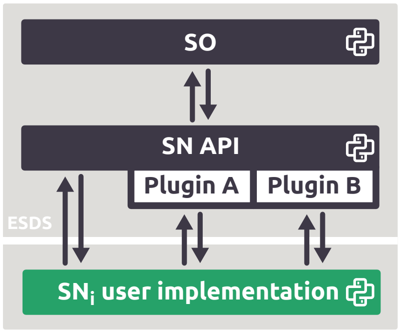


Fig. 3. Interactions of SN_i implementation supplied by the user with the ESDS API and plugins system. Two plugins, noted A and B are used by a given SN_i implementation.

The plugins system is presented Figure 3. This figure shows the interactions between a given user supplied SN implementation (written in Python) with the ESDS API and plugin system. This SN labeled i, has two active plugins noted A and B. The SN implementation interacts with them using their respective plugin API. The SN implementation can interact directly with ESDS using its SN API.

Currently, ESDS provides plugins that measure the energy consumed by the node. They are based on the power state model [34]. This model approximates the SN energy consumption by assuming that, physical nodes go through several discrete power consumption states during their operation. By keeping track of each power state, and the duration spent by the nodes in each states, the energy consumption can be estimated. Hence, for a set of power states $S = \{P_1, .., P_n\}$, the energy consumed by a SN is expressed as:

$$E_{\text{SN}} = \int_0^t P(t)dt \approx \sum_{i=1}^{n} P_i \times \Delta_i \tag{2}$$

With Δ_i corresponding to the duration spent by the SN on the power state i with a power consumption of P_i. Validation of this model implemented in ESDS is available in [10]. Using this model, the overall energy consumption of a given communicating system can be studied. More specifically, energy constrained scenarios can be studied.

V. Simulate the DAO-CPS

This section presents the use of ESDS for studying DS in scarcely resourced environments, like the DAO-CPS.

A. Simulation scenario

To document the use of ESDS in the DAO context, this work proposes to reproduce various DS experiments from [18]. This work contributes to the DAO by providing loosely coupled data dissemination policies. Based on this contribution, this work implements the *Baseline* policy using ESDS. Other policies are extensions and can be derived from *Baseline*. For space reasons, only *Baseline* related experiments are covered.

The scenario presented in [18] consists in having 13 nodes comprising one sender and 12 receivers. Nodes are off most of the time. With the *Baseline* policy, nodes wake up every hours at a random time for a short up-time duration of either $60s$ or $180s$. During its up-time, the sender strives to transmit $1MB$ of data to the receivers that are available (overlapping up-time period between sender and receivers). This work focuses on the $180s$ case for space reasons.

Regarding wireless communications, technologies from LPWAN are used in the DAO-CPS context. Either LoRa with a bandwidth of $50kbps$ or NbIoT with $200kbps$. During the experiment, the nodes energy consumption is estimated. In [18], the hardware used by the nodes is based on a Raspberry Pi Zero that consumes $0.4W$ on idle time. The energy consumed during communications is also accounted. For LoRa, $0.13W$ is consumed during transmission and reception. For NbIoT, $0.64W$ is consumed during transmission and reception. The values of each simulation parameters are extracted from [18].

B. ESDS implementation of Baseline

To perform this implementation with ESDS, two files are required. The first one, called *platform file* uses the YAML syntax. It is used to configure ESDS. It sets up the simulation parameters, the simulated nodes and their communication interfaces. The second file, contains the nodes implementation written in Python where the ESDS API (presented in Section IV-C) is used to implement node behaviors. It is where the user is able to implement the nodes logic.

The platform file used for the implementation of the *Baseline* policy is presented in Listing 1. This listing contains two main sections: A *nodes* and an *interfaces* section.

```
 1 ##### Nodes Setup #####
 2 nodes:
 3     count: 13
 4     implementations:
 5         - all node.py
 6     arguments:
 7         0: { "type": "sender", "uptime": 180, "
        datasize": 1000000, "wireless": "lora"}
 8         1-@: {"type": "receiver", "uptime": 180, "
        wireless": "lora"}
 9
10 ##### Nodes Interfaces #####
11 interfaces:
12     lora:
13         type: "wireless"
14         nodes: all
15         links:
16             - all 50kbps 0s all
17         txperfs:
18             - all 50kbps 0s
19     nbiot:
20         type: "wireless"
21         nodes: all
22         links:
23             - all 200kbps 0s all
24         txperfs:
25             - all 200kbps 0s
```

Listing 1: Platform file configured to use LoRa

The *nodes* section does defines the number of SNs (line 3). In our case, 13 nodes are simulated. It binds the implementation files to each SN (line 5). Here, `node.py` is used by all the SNs. It defines the arguments given to each SN (line 7-8). These arguments are accessible in the SNs implementation file. The arguments set node 0 as the sender, and the remaining nodes as receivers. It sets the up-time duration of the SN to be $180s$ and the amount of data transmitted by the sender for each communication to 1MB. Finally, the interface used by the SNs to communicate is set to be ”*lora*”.

The *interfaces* section defines two communication interfaces usable by the SNs: LoRa (line 12) and NbIoT (line 18). Each of these interfaces is wireless (line 13 and 20). All nodes that belong to the network, accessible using the interface, are referenced on line 14 and 21. For each interface, the communications performance are provided. As an example, line 15 set the latency and bandwidth matrices presented in Section IV-B. In this case, all communications, from all nodes, to all nodes, have a bandwidth of 50kbps and a latency of 0s. For wireless interfaces, an additional performance parameter called *txperfs* is required. It defines the transmission performance of each SN on the given interface. It is used to estimate the duration of the wireless transmissions on the interface for each SN. In this implementation, all nodes have the same transmission performance (bandwidth of 50kbps and a latency of 0s).

The Listing 2 provides the implementation of the SNs. Each implementation must define an *execute()* function with an *api* argument. This argument, provides access to the ESDS API detailed in Section IV-C. In this listing, the power state plugin is initialized on line 9, to set up the node power consumption. At each node state change (line 19 and 29), the node power consumption is updated accordingly. On line 11 and 12, the power consumption of network interfaces is defined for LoRa and NbIoT. Each of the three power states (idle, transmission and reception) is given to the plugin that will automatically switch to the correct power state during communications.

```python
1  #!/usr/bin/env python
2
3  import random
4  from esds import RCode
5  from esds.plugins.power_states import *
6
7  def execute(api):
8      ##### Setup node power consumption ####
9      node_cons=PowerStates(api, 0)
10     comms_cons=PowerStatesComms(api)
11     comms_cons.set_power("lora", 0, 0.16, 0.16)
12     comms_cons.set_power("nbiot", 0, 0.65, 0.65)
13     ##### Start node implementation #####
14     rand=random.Random(api.node_id) # Reproducible
15     api.turn_off() # Node off on start
16     for hour in range(0,24): # 24 hours
17         api.wait(rand.randint(0,3600-api.args["
       uptime"]))
18         api.turn_on()
19         node_cons.set_power(0.4)
20         wakeat=api.read("clock")
21         wakeuntil=wakeat+api.args["uptime"]
22         while api.read("clock") < wakeuntil:
23             if api.args["type"] == "sender":
24                 api.sendt(api.args["wireless"],"my
       data",api.args["datasize"],None, wakeuntil-api.
       read("clock"))
25             else:
26                 code, data=api.receivet(api.args["
       wireless"],wakeuntil-api.read("clock"))
27                 if code == RCode.SUCCESS:
28                     api.log("Receive "+data)
29         node_cons.set_power(0)
30         api.turn_off()
31         api.wait(3600*(hour+1)-api.read("clock"))
32     node_cons.report_energy()
33     comms_cons.report_energy()
```

Listing 2: Node implementation for the *Baseline* communication policy working with LoRa or NbIoT

The implementation of the *Baseline* scenario starts on line 15 where all the nodes are turned off. Then, for each hour of the day (line 16), the nodes stay off for a random duration (line 17). They then turn on (line 18). When being up (up-time), nodes idle power consumption is set to 0.4W (line 19). When being the sender (line 23), the node sends the data (line 24). When being a receiver, the node strives to collect the data that it potentially received for the duration of its up-time (line 26). Finally, nodes are turned off at the end of their up-times (line 30), and their idle power consumption is set to 0W (line 29). At the end of the simulation, the nodes energy consumption are reported by ESDS (line 32 and 33).

C. *Running simulations*

To execute a simulation, the command *"esds run [platform_file]"* is executed. 1000 runs are performed for each wireless interface (LoRa and NbIoT). For each run, the seed parameter (line 14 in Listing 2) is changed. Hence, a different node schedule is used in each run, and the impact of the node schedule on the results is quantified.

VI. Simulation Results

This section presents results obtained from ESDS simulations. These results document the possibilities of ESDS to study DS in terms of energy consumption, network performance and nodes behaviors. All results presented in this section are derived from the ESDS standard output.

A. *Energy consumption*

The Figure 4(a), shows the nodes energy consumption results using LoRa, extracted from the ESDS standard output. For each node specified in Listing 1, its average energy consumption over 1000 runs is shown. This energy consumption comprises the idle (in yellow) and communications part (in dark blue). The standard deviations for the communications part, obtained over the 1000 runs, are in red.

These results show that, with LoRa, the major part of the energy consumed by the nodes comes from the idle. The power state model estimates that, each node consumes 1728J per day for the idle part. This value is constant across runs. On the receiver nodes, the estimated energy consumed by communications is relatively small (less than 100J with a small standard deviation) compared to the idle. However, on the sender (node id 0), the energy consumed during communications is higher. It represents around 30% of the energy consumed by the sender. As the sender is continuously transmitting during its up-time, the energy consumed by the sender during communications is constant across runs, thus it has standard deviation of 0J.

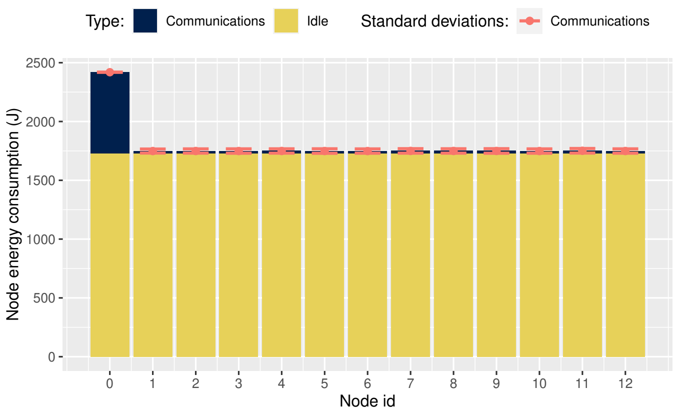
Similarly, the Figure 4(b) depicts the nodes energy consumption results using the NbIoT wireless technology. The estimated energy consumed during idle periods is similar to the results using LoRa. However, NbIoT offers better communication performance but consumes more energy, compared to LoRa. It translates into a significant increase on the energy consumed during communications. This has a major impact on the sender, as it is continuously sending data during its up-time. In addition, an increase in its standard deviation of energy consumed during communication is visible.

B. *Network performance and nodes behavior*

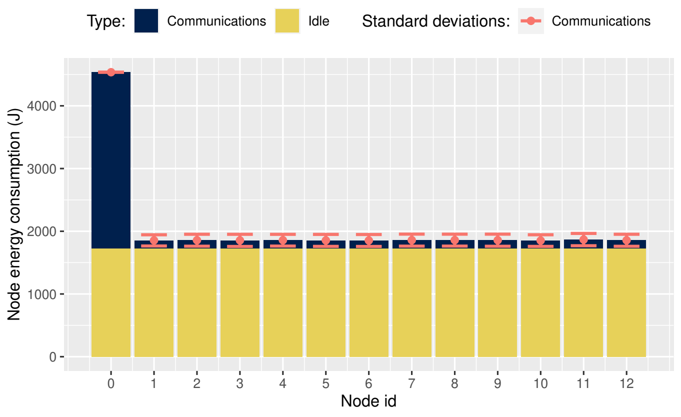
Table II provides results extracted from ESDS. It shows the estimated value of various metrics, using LoRa and NbIoT. Standard deviations are in parenthesis.

In a day, for an up-time of $180s$, if a node turns on every hour, it is on 5% (1h12m) off 95% (22h48m) of the time. Using LoRa, the sender sends 48MB of data each day with a standard deviation of 0MB, since the nodes up-time ratio per day is constant and data are sent continuously. This confirms the results of Figure 4, where the sender have a constant energy consumption for the communications over the 1000 runs (standard deviation of 0J).

These results reveal that, in average, with LoRa, over the 48MB of data transmitted, only 2MB are received by the receiver nodes (id 1 to 12, with ±1.1MB). Only 4% (with ±2%) of the data transmitted by the sender are received. The remaining transmitted data (≈ 46MB) are lost and lead to an increase of the energy consumed by the sender. These results show that in average, for each run, 1.9 nodes receive the



(a) Nodes energy consumption using LoRa



(b) Nodes energy consumption using NbIoT

Fig. 4. Simulated nodes energy consumption using LoRa and NbIoT. Results are an average over 1000 runs for each wireless technology. The standard deviation of the energy consumed during communications is given in red.

TABLE II
SIMULATIONS RESULTS USING LORA AND NBIOT. THE STANDARD DEVIATIONS OF THE GIVEN METRICS ARE IN PARENTHESIS.

	Wireless Technology		
Description	*LoRa*	*NbIoT*	Difference
Number of runs	1000	1000	-
Simulated duration for each run	86400s *or 24h00m0s*	86400s *or 24h00m0s*	-
Cumulative up-time per node	4320s *or 01h12m0s*	4320s *or 01h12m0s*	-
Cumulative off-time per node	82080s *or 22h48m0s*	82080s *or 22h48m0s*	-
Up-time ratio per day	5.0%	5.0%	-
Off-time ratio per day	95.0%	95.0%	-
Average amount of data sent per day	48.0MB(0.0)	120.0MB(0.0)	+72.0MB
Average amount of data received per day	2.1MB(1.1)	46.2MB(10.4)	+44.1MB
Average percentage of data received per day	4.0%(2.0)	38%(9.0)	+34.0%
Average number of node that receive the data per day	1.9(1.0)	10.1(1.3)	+8.2

data. This quantifies the (in)efficiency of the *Baseline* policy at propagating data in a network using LoRa.

Using the NbIoT wireless technology, these results show a significant increase in the data transmitted by the sender (+72MB compared to LoRa). Since NbIoT offers better network performance, more data are received on the receiver nodes. In average, 38% of the transmitted data are received. In addition, ESDS estimates that an average of 10 nodes per run received the data transmitted by the sender. Consequently, this scenario offers a better dissemination of the data.

VII. Discussions

A. Simulations results

The results provided by the ESDS allow for the study of DS over several aspects: node behaviors, network performance and energy consumption. The implementation of the *Baseline* policy from [18] permits to compare LoRa or NbIoT as a wireless technology. It documents the nodes behavior (e.g up-time and off-times), the network performance (e.g amount of data transmitted per day) and the nodes energy consumption (e.g idle and network communications).

From these observations, research directions can be derived. The impact of the *Baseline* policy on the sender energy consumption is not negligible compared to its impact on the receiver nodes. Increasing the number of nodes that receive the data per day is a possible optimization (e.g: fine tuning up-time duration, node schedule). A trade-off between the node idle energy consumption, and the average percentage of data received per day can be derived. Conducting these studies allow to characterize the system on various performance metrics (e.g: network performance, energy efficiency). Thus, in the DAO-CPS context, ESDS allows to anticipate and optimize energy budget, prior node deployments.

B. ESDS architecture and API

The experiments conducted show that the architecture of ESDS is adapted to the study of DS in the context of the DAO-CPS. Experiments show that the SN API is sufficiently versatile to implement various distributed systems scenarios that involve: network communications, change of nodes operating states (turning on and off) and energy consumption estimations, even in resourced constrained environments.

C. Current limitations

ESDS can simulate various scenarios that involve wireless distributed systems. However, it has limitations. ESDS does not provide a validated model for wired networks. Studying wired distributed systems is not currently possible.

As ESDS is a relatively new, the amount of available plugins is not high. Several other use-case specific features need to be implemented. As an example, the design of a battery plugins can improve the study of scenarios with nodes that have a limited budget. Studying energy variability is also critical [35] and requires an additional plugin. Introducing communications error models, based on physical parameters such as temperature and humidity, is an interesting research axis for simulating DAO-CPS-like contexts.

VIII. Conclusion

In systems deployed in constrained environments, like the DAO-CPS, existing state of the art packet-level and flow-level simulators are not adapted. Prior works use ad-hoc simulators. This approach is time consuming and models used by these simulators are most of the time not validated.

This work uses the Extensible Simulator for Distributed Systems (ESDS). This simulator framework aims at, being simple to use and versatile, to perform studies on distributed systems. This paper details the architecture of ESDS along with its network models, energy models and nodes API. A scenario that is specific to the DAO-CPS, a CPS deployed in constrained environments [18], is implemented with ESDS.

Details of the proposed implementation shows that ESDS can be used to model systems in the DAO-CPS context with ease. The results document that ESDS is suitable to perform research studies on energy consumption and network performance of distributed systems. An open source implementation of ESDS, in Python, is available online [9]. We are panning to tackle the limitations presented in Section VII-C.

Acknowledgment

The DAO project is supported by the Research Council of Norway (RCN) IKTPluss program, project number 270672.

References

[1] Cisco, "Cisco Annual Internet Report," 2020.

[2] R. Hassan and Qamar et al., "Internet of Things and Its Applications: A Comprehensive Survey," *Symmetry*, vol. 12, no. 10, Oct. 2020.

[3] Kandris et al., "Applications of Wireless Sensor Networks: An Up-to-Date Survey," *Applied System Innovation*, vol. 3, no. 1, Feb. 2020.

[4] R. S. Nandhini and R. Lakshmanan, "A Review of the Integration of Cyber-Physical System and Internet of Things," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 4, 2022.

[5] S. Idris et al., "Survey and Comparative Study of LoRa-Enabled Simulators for Internet of Things and Wireless Sensor Networks," *Sensors*, Jul. 2022.

[6] M. J. Murphy and Tveito et al., "Experiences Building and Deploying Wireless Sensor Nodes for the Arctic Tundra," in *IEEE/ACM 21st International Symposium CCGrid*, Melbourne, Australia, May 2021.

[7] L. Sergiusz, O. J. Anshus, and J. Markus, "Flexible Devices for Arctic Ecosystems Observations."

[8] L. S. Michalik, L. Guegan, I. Rais, O. Anshus, and J. M. Bjørndalen, "LoRaLitE: LoRa protocol for Energy-Limited environments," 2022.

[9] L. Guegan, "ESDS - extensible simulator for distributed systems," https://gitlab.com/manzerbredes/esds.

[10] L. Guegan, I. Raïs, and O. Anshus, "Validation of ESDS Using Epidemic-Based Data Dissemination Algorithms." The 19th Annual International Conference DCOSS-IoT, 2023.

[11] Pedersen *et al.*, "Climate-Ecological Observatory for Arctic Tundra (COAT)."

[12] L. Hislop, "The View from the Top: Searching for responses to a rapidly changing Arctic," 2013.

[13] R. A. Ims, J. U. Jepsen, S. Audun, and S. G. Yoccoz, "Science Plan for COAT: Climate-Ecological Observatory for Arctic Tundra," 2013.

[14] J. E. Box and Colgan et al., "Key indicators of Arctic climate change: 1971–2017," *Environmental Research Letters*, vol. 14, no. 4, Apr. 2019.

[15] J. Feldner, C. Hübner, H. Lihavainen, R. Neuber, and A. Zaborska, "The State of Environmental Science in Svalbard," 2021.

[16] Rais *et al.*, "UAVs as a Leverage to Provide Energy and Network for Cyber-Physical Observation Units on the Arctic Tundra," in *15th International Conference DCOSS*. IEEE, May 2019.

[17] Soininen et al., "Under the snow: A new camera trap opens the white box of subnivean ecology," *Remote Sensing in Ecology and Conservation*, 2015.

[18] I. Rais, L. Guegan, and O. Anshus, "Impact of loosely coupled data dissemination policies for resource challenged environments," 2022.

[19] Rais *et al.*, "Trading Data Size and CNN Confidence Score for Energy Efficient CPS Node Communications," in *20th IEEE/ACM International Symposium CCGRID*. IEEE, May 2020.

[20] Vera-Amaro et al., "Design and Analysis of Wireless Sensor Networks for Animal Tracking in Large Monitoring Polar Regions Using Phase-Type Distributions and Single Sensor Model," *IEEE Access*, vol. 7, 2019.

[21] Younus et al., "Proposition and Real-Time Implementation of an Energy-Aware Routing Protocol for a Software Defined Wireless Sensor Network," *Sensors*, vol. 19, no. 12, Jun. 2019.

[22] Randrup and others, "Impact of image compression on CNN performance metrics for CPS nodes at the Arctic Tundra," in *2021 IEEE International Conferences iThings-GreenCom-CPSCom-SmartData-Cybermatics*. Melbourne, Australia: IEEE, Dec. 2021.

[23] B. Liu and Figueiredo et al., "A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation," in *Proceedings IEEE INFOCOM*, 2001.

[24] G. F. Riley and T. R. Henderson, "The ns-3 Network Simulator," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[25] Anggono et al., "A Flow-Level Extension to OMNeT++ for Long Simulations of Large Networks," *IEEE Communications Letters*, 2017.

[26] M. Zakarya, "PerficientCloudSim: A tool to simulate large-scale computation in heterogeneous clouds," *The Journal of Supercomputing*, 2021.

[27] Courageux-Sudan et al., "Automated performance prediction of microservice applications using simulation," in *29th International Symposium MASCOTS*. Houston, TX, USA: IEEE, 2021.

[28] A. B. M. Fanfakh, "Predicting the Performance of MPI Applications over Different Grid Architectures," *Journal of University of Babylone for Pure and Applied Sciences*, Apr. 2019.

[29] F. C. Heinrich and Cornebize et al., "Predicting the Energy-Consumption of MPI Applications at Scale Using Only a Single Node." IEEE, 2017.

[30] M. Quinson, C. Rosa, and C. Thiéry, "Parallel Simulation of Peer-to-Peer Systems." IEEE, May 2012.

[31] L. Guegan, B. L. Amersho, A.-C. Orgerie, and M. Quinson, "A Large-Scale Wired Network Energy Model for Flow-Level Simulations," in *Conference AINA*, 2019.

[32] R. Johari and S. Adhikari, "Routing in IoT Network using CupCarbon Simulator," in *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*. Noida, India: IEEE, Feb. 2020.

[33] P. Velho and A. Legrand, "Accuracy study and improvement of network simulation in the SimGrid framework," in *Proceedings of the Second International ICST Conference on Simulation Tools and Techniques*. Rome, Italy: ICST, 2009.

[34] Wu et al., "An Energy Framework for the Network Simulator 3 (ns-3)," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. Barcelona, Spain: ACM, 2011.

[35] S. Tofaily, I. Raïs, and O. Anshus, "Quantifying the variability of power and energy consumption for iot edge nodes," IEEE. The 19th Annual International Conference DCOSS-IoT, 2023.