Internet of Drones Simulator: Design, Implementation, and Performance Evaluation

Giovanni Grieco©, Graduate Student Member, IEEE, Giovanni Iacovelli©, Graduate Student Member, IEEE, Pietro Boccadoro©, Member, IEEE, and Luigi Alfredo Grieco©, Senior Member, IEEE

Abstract—The Internet of Drones (IoD) is a networking architecture that stems from the interplay between unmanned aerial vehicles (UAVs) and wireless communication technologies. Networked drones can unleash disruptive scenarios in many application domains. At the same time, to really capitalize on their potential, accurate modeling techniques are required to seize the fine details that characterize the features and limitations of UAVs, wireless communications, and networking protocols. To this end, the present contribution proposes the IoD simulator (IoD-Sim), a comprehensive and versatile open source tool that addresses the many facets of the IoD. IoD-Sim is a network simulator 3 (ns-3)-based simulator organized into a 3-layer stack, composed of: 1) the Underlying Platform, which provides the telecommunication primitives for different standardized protocol stacks; 2) the Core, that implements all the fundamental features of an IoD scenario; and 3) the Simulation Development Platform, mainly composed of a set of tools that speeds up the graphical design for every possible use case. In order to prove the huge potential of this proposal, three different scenarios are presented and analyzed from both a software perspective and a telecommunications standpoint. The peculiarities of this open-source tool are of interest for researchers in academia, as they will be able to extend it to model upcoming specifications, including, but not limited to, mobile and satellite communications. Still, it will certainly be of relevance in industry to accelerate the design phase, thus reducing the time to market of IoD-based services.

Index Terms—Internet of Drones (IoD), network, network simulator 3 (ns-3), simulator.

I. INTRODUCTION

THE Internet of Drones (IoD) [1] is one of the hottest research topics in telecommunications today [2]. At first, it might appear as an extension of the Internet of Things (IoT), with unmanned aerial vehicles (UAVs) playing the role of smart objects able to fly. Nevertheless, in the IoD, drones are tasked with completing mission plans with multiple objectives. Since they can also fly in organized groups, namely, swarms, it is worth remarking that they are made able to continuously optimize their trajectory, and coordinate among themselves. Drones are currently involved in the delivery of value-added services in many applications, especially in Smart Cities [3], [4], [5], [6], including goods delivery, environmental surveying, first-aid units in disruptive events [2], [6], and flying base station (BS) in fifth generation (5G) & Beyond scenarios, with multiple users requesting connectivity at the same time and in the same area [2], [6], [7], [8]. Smart cities are among the most challenging application scenarios, with everchanging players and behavioral patterns, which makes it hard to address public safety requirements, especially at scale [9]. All this turned the IoD from a niche subject to a mainstream research topic in networking. It must be noted that the adoption of drones in industry is also a huge commercial opportunity, as testified by the several billion forecasts already available for multiple business sectors [2].

Even though several applications are now including drones, and they may look like off-the-shelf utilities, the design of complex IoD systems still requires advanced methodologies to effectively unleash the potential of services based on networked drones. In 5G and beyond scenarios, ubiquitous connectivity and relaying capabilities are required to interact with both terrestrial entities, i.e., ground BSs and users, as well as aerospace ones, such as satellites [10]. In this regard, channel capacity, available/required data rates, dedicated bandwidth, and frequencies must be characterized, bearing in mind that every link may be realized with different telecommunication protocols. Moreover, given the variety of available drones on the market, an accurate suitability assessment based on their characteristics is required.

Current state-of-the-art IoD simulators [11], [12], [13], [14], [15], [16], [17] do not cover all the aforementioned aspects. In light of the foregoing, the key contributions of this work are hereby introduced.

1) A comprehensive open-source simulation platform, namely IoD Simulator (IoD-Sim), is proposed in this work. IoD-Sim is able to create realistic simulations by extending the available features of network simulator 3 (ns-3) to address the relevant aspects of the IoD. Since its first release [18], it has been carefully redesigned and thoroughly refactored. The overall architecture of the proposal is designed as a 3-layer stack:

https://github.com/telematics-lab/IoD_Sim

Manuscript received 16 August 2022; accepted 5 September 2022. Date of publication 16 September 2022; date of current version 6 January 2023. This work was supported in part by the PRIN Project titled “Realtime Control of 5G Wireless Networks: Taming the Complexity of Future Transmission and Computation Challenges” funded by the Italian MIUR under Grant 2017NS9F9Y; in part by the Project titled “The House of Emerging Technologies of Matera (CTEMT)” funded by the Italian MISE; in part by the Italian MIUR PON Projects AGREED under Grant ARS01_00254; and in part by the Warsaw University of Technology within IDUB Programme under Contract 1820/29/Z01/POB2/2021. (Corresponding author: Giovanni Grieco.) Giovanni Grieco, Giovanni Iacovelli, and Luigi Alfredo Grieco are with the Department of Electrical and Information Engineering, Politecnico di Bari, 70126 Bari, Italy, and also with the Consorzio Nazionale Interuniversitario per le Telecomunicazioni, 43124 Parma, Italy (e-mail: giovanni.grieco@poliba.it; giovanni.iacovelli@poliba.it; alfredo.grieco@poliba.it).

Pietro Boccadoro is with the Department of Electrical and Information Engineering, Politecnico di Bari, 70126 Bari, Italy, and also with the NXtome srl, 70014 Bari, Italy (e-mail: pietro.boccadoro@poliba.it).

Digital Object Identifier 10.1109/JIOT.2022.3207324
The present contribution is structured as follows. Section II summarizes the reference state-of-the-art of simulators in order to cover their peculiarities and motivate the proposal. Section III presents a general overview of the simulator architecture. Section IV describes the Underlying Platform and its design rationale. Section V discusses the Core of the simulator in detail, with dedicated sections focused on the main building blocks of the project. A thorough explanation of the involved mobility models is given together with all the supported communication technologies, and the included logical entities. Section VI focuses on the simulation design, discussing the details of helpers, thus, explaining the role and importance of scenario configurations. Section VII is dedicated to the simulation campaign; after an initial focus on scenarios description, the outcomes are discussed to highlight the main findings. Finally, Section VIII concludes the work and outlines future work possibilities.

II. RELATED WORKS

To improve and speed up both the design and the prototyping phases of IoD systems, simulations are widely conceived as a useful aid. Even though simulating drones is a challenging task, it has been dealt with by many contributions so far [11], [12], [13], [14], [15], [16], [17]. Overall, these works approach IoD simulations from two different points of view. The first focuses on the dynamics of the flight, thus including mechanical energy and kinetics; they employ Robot Operating System (ROS) [19] and Gazebo [20] as base platforms [14], [16]. The second, instead, focuses on accurate drone networking simulations [11], [12], [13], [15], [17], mainly based on ns-3 [21] and OMNeT++ [22], in which UAVs are envisioned as nodes exchanging data at certain frequencies using well-known protocols belonging to wireless networks, which can either be cellular or Wi-Fi.

The contribution presented in [11] models UAVs and discusses their functionalities and possible applications. In particular, the proposal introduces FlyNetSim, a software that aims at simulating not only flight operations but also networking communication primitives and principles. The simulator can work with a group of drones operating together in a reference ecosystem. The most interesting functionalities are: 1) UAV control over Wi-Fi; 2) multinetwork communications; 3) device-to-device communications for swarms; and 4) IoT scenarios and data streaming.

In [12], instead, it is proposed CUSCUS, a simulation architecture for control systems in the context of drones’ networks. The proposal is able to simulate the mechanisms for the control of drones operations and it is claimed to be highly flexible and scalable. The proposed simulator leverages the ns-3 capabilities to work with virtual interfaces simulating real-time systems, eventually composed by swarms.

The contribution presented in [13] describes AVENS, which is a hybrid network simulation framework specifically designed to evaluate the performance of intelligent aerial vehicles. Here, drones are communicating using some of the most well-known communication protocols for flying ad-hoc networks (FANETs). Different from other contributions, AVENS is focused on modeling realistic flight conditions. On top of that, it uses a layered architecture that acts as an interpreter and code generator, namely, LARISSA, thanks to specified simulation parameters and settings. All the results are obtained by the integration and interoperability with the OMNeT++ simulator.

The proposal in [14] is a simulation framework for unmanned aircraft systems traffic management. It leverages both ROS and Gazebo to implement high-level flight services. The simulator can be used for prototyping missions and controlling both rotary and fixed-wing drones flying in the same environment.

The work presented in [15] discusses a Java-based simulation framework for FANET networks and their applications. In particular, it models the coverage area of each device in the scenario. At the same time, it considers a mobility model for ground entities, i.e., humans in the operating area. Drones’ characterization is herein discussed in terms of limited autonomy and battery recharging needs. To achieve this aim, an energy consumption model has been included to evaluate the footprint associated with the flight of a drone. For the sake of completeness, it must be said that this work neglects the contributions due to collision issues and consequent behaviors.

CORNET 2.0 [16] is a middleware to simulate robots in general, both in physical and networking contexts. It reaches the aim of designing a path planning solution that is simulated by Gazebo and Mininet-WiFi.

The work presented in [17] proposes a discrete-time, event-based, co-simulation scheme for networks composed by
multiple drones, also configured in swarms. The simulator can carry out both flight and network simulations. This solution is of interest because there is an intrinsic codependency between the flight status and the networking operations carried out by each drone in the scenario. This contribution is of relevance because it is claimed to guarantee reliability and real-time availability thanks to the possible integration of existing simulators. This work also claims that other available simulators do not implement realistic and reliable mobility models for drones.

A comparison among the main characteristics and features of the aforementioned contributions is provided in Table I. Specifically, only the latter [17] shows some similarities with IoD-Sim. For example, both of them operate as discrete-time and event-driven simulators. Nevertheless, it is worth noting that the discrete-time operating mode of this work is motivated by the adoption of ns-3 [21] as a core network simulator. Another aspect is related to the synthetic trajectories that are implemented in IoD-Sim, that are described by closed-form mathematical expressions. Hence, in case the network simulator is substituted, the mobility models provided by IoD-Sim could be used even with continuous-time simulators.

III. ARCHITECTURAL OVERVIEW

The overall architecture of the proposed IoD-Sim simulator is depicted in Fig. 1. This diagram frames the complexity and clarifies the organization of the main building layers, each providing peculiar functionalities that are depicted as blocks. The joint adoption of these components enables different simulation scenarios, which are configurable by higher-level entities.

From the bottom-up, the first layer, i.e., Underlying Platform, has been created to group all the necessary components to carry out simulations. In particular, it includes: 1) the GNU Scientific Library for advanced mathematical operations; 2) the RapidJSON library which is specifically designed for high-performance parsing operations; and 3) the ns-3 simulator engine for a robust foundation of networking facilities. A detailed description of these components is given in Section IV.

The second layer, namely, Core, implements IoD-related simulation facilities and it is organized into three main subgroups. The World Definition is motivated by the fact that a realistic network simulation must be modeled, taking into account cyber–physical aspects. Hence, IoD-Sim allows to simulate the physical space in which the simulation takes place, such as Buildings and Regions of Interest (RoIs). Furthermore, Entities details all the aspects related to drones, zone service providers (ZSPs), and remote hosts, spanning from their peculiar characteristics, i.e., peripherals and mechanics for UA Vs, to general ones, such as applications. Finally, the Scenario Configuration Interface allows the entire layer to be highly configurable thanks to the definition of a high-level language. In light of the above, the entire layer can be fine-tuned to dig as deep as possible for the use case of interest, as explained in Section V.

The third and last layer, i.e., the Simulation Development Platform, provides high-level configuration and data analytics facilities to the end-user. It includes Airflow, a high-level visual configuration environment that drastically eases the interaction between the user and the simulator, i.e., scenario set-up and management. Moreover, a Report Module guarantees the readability of simulation results in a clear way.
extensible markup language (XML) schema. This module, together with the Resultis Aggregator, eases data processing with third-party tools. A detailed description of these components is given in Section VI.

It is of upmost importance to clarify that the above architecture is a multifaceted solution which applies the concept of modularity-by-design across the entire software implementation, thus proposing the peculiar functionalities described in what follows.

IV. UNDERLYING PLATFORM

The Underlying Platform is a foundation composed of the GNU scientific library (GSL), RapidJSON, and ns-3.

GSL is a numerical computing framework that implements numerous routines and low-level data structures, such as complex numbers, linear algebra, data analysis, and interpolation. Furthermore, it is offered in Linux-derived distributions with first-class support [23].

RapidJSON is a parser and generator of javascript object notation (JSON) code. It is one of the most adopted JSON libraries available for C++ projects. It eases the creation, traversal, validity check, and analysis of JSON codes [24]. RapidJSON has been chosen for its high performance and its extensive and flexible high-level application programming interfaces (APIs).

Finally, ns-3 emerges as the most relevant component: it is a solid and mature discrete-time event-based network simulator. ns-3 is an open-source project that provides a solid simulation engine and various models for network design and testing. Started in 2006, it is a collection of different C++ and Python objects that implements several aspects of networking elements. The fundamental building block of ns-3 is ns3::Node, an abstract object which represents a generic host in a network. It can be aggregated with other objects and models, e.g., the common transmission control protocol (TCP)/IP stack over Ethernet, to simulate networking behavior. Other interesting features in ns-3 are: 1) ns3::Channel, which simulates the communication channel between ns3::Node objects; 2) ns3::NetDevice, which represents the node networking interface; and 3) ns3::Application, which sits on top of the protocol stack to produce or consume high-level information.

Furthermore, an ns3::Node can be aggregated with Mobility Models and Energy Consumption Models. This possibility is not limited to those models, since the support can be extended to any other model that adds new features beyond basic networking. To this end, nodes have the potential to move in space and, hence, drain current from an ns3::EnergySource.

Besides, traces and probes allow to track and record simulation data in log files that are typically encoded in textual ASCII, or PCAP. In a nutshell, IoD-Sim treats ns-3 as a foundation, extending it with new features that are focused on accurate drone simulations, mobile wireless communications, energy consumption, and their integration with onboard peripherals and ground communication infrastructures.

V. CORE OF IOD-SIM

This section presents the building blocks of the IoD-Sim Core, which is the main part of the simulator.

A simulation scenario requires the definition of a simulated world, described by RoIs and buildings. In this world, entities, i.e., drones, ZSPs, and remotes, are simulated in a network topology defined by a set of communication models. Each drone is characterized by a mission plan defined by a set of points of interest (PpoI), which in turn describes a curvilinear trajectory. Furthermore, a drone can be equipped with an energy consumption model, which relies on a set of mechanical properties and a set of peripherals. Entities in general can host one or more communication stacks and applications. While drones and ZSPs are connected together according to the configuration of the IoD infrastructure, remotes are reachable through a backbone that simulates Internet behavior. All these blocks are configurable through an abstraction interface focused on interpreting a high-level description of the scenario encoded in JSON format.

A. World Definition

IoD-Sim offers the possibility to define parameters related to the simulated world, i.e., the environment in which the simulation takes place. The two main features are the buildings and the RoIs.

The virtual world in IoD-Sim is a theoretically infinite space. The space can be filled with entities, which could be Drones, ZSPs and Remotes, but also with RoIs and Buildings.

1) Buildings: The virtual world can be enriched with obstacles, i.e., Buildings. They are used to represent urban scenarios, thus, making simulations that are particularly suitable for research in Smart Cities. IoD-Sim provides an abstraction layer to configure and place buildings in the virtual world, relying on ns3::BuildingsHelper and ns3::Building objects. An ns3::Building is a collisionless 3-D object with the following properties.

1) boundaries, which defines the box dimension in the space. Boundaries can be defined by an array of two points organized as \( [P_x^{(1)}, P_x^{(2)}], [P_y^{(1)}, P_y^{(2)}], [P_z^{(1)}, P_z^{(2)}] \). A representation of these two points is given in Fig. 2.

2) type of building, which can be either commercial, residential, or office.

3) type of walls material, which can be wood, concreteWithWindows, concreteWithoutWindows, or stoneBlocks.

4) Number of floors.

5) Number of rooms along the x and y axis, per floor. The rooms are placed in a grid position. Such a feature is important for long-term evolution (LTE) communication fading, which varies according to the characteristics of each building.

2) Regions of Interest: An ns3::InterestRegion is a 3-D box placed on the simulated world defined, as for buildings, by a vector of two points. Throughout the simulation, it is possible to retrieve and to update the
Fig. 2. Example of box placement with two points, $P_1$ and $P_2$, in order to create a Building or a RoI in the simulated world.

TABLE II

| ns3::Drone Properties in IoD-Sim |
|----------------------------------|
| **Name**                         | **Unit of Measurement** |
| Mass                             | kg                      |
| Rotor Disk Area                  | $m^2$                   |
| Drag Coefficient                 | (dimensionless)         |
| Peripherals                      |                         |
| Weight Force                     | N                       |
| Air Density                      | $kg/m^3$                |

The current set of coordinates with `GetCoordinates()` and `SetCoordinates()` methods, respectively.

The whole set of these areas is managed by `ns3::InterestRegionContainer`, which helps to create Regions of Interest (RoIs) and group them. This utility object provides: 1) `Create()` method to generate and index RoIs; 2) `GetN()` to report the number of created regions; 3) `GetRoi()` to retrieve the $i^{th}$; and 4) `Begin()` and `End()` iterators to traverse the entire container. Moreover, the `InterestRegionContainer::IsInRegions()` method acknowledges the presence of a drone in multiple areas, thus granting the possibility to trigger specific events during the simulation. For instance, `Drone` operations can be restricted to a limited space, leading to an optimization of `Drone` power consumption.

**B. Drones**

IoD-Sim provides `ns3::Node` derivatives to consider the characteristics of key actors commonly found in an IoD simulation. The `ns3::Drone` class characterizes a rotary-wing UAV and it is registered as a new `TypeId` in ns-3, along with its mechanical properties, shown in Table II. While the first four properties can be defined by the user, the last two are a direct consequence of the given characterization. `ns3::Drone` properties can be set by means of ns-3 attributes or by its public object interface. Its mass can be updated at any time by means of `SetMass()`. Upon update, the drone weight force is also updated in cascade by multiplying the new mass with the constant gravity acceleration. The rotor disk area and its drag coefficient can be updated in the same manner by means of `SetArea()` and `SetDragCoefficient()` methods, respectively. Furthermore, `ns3::Drone` properties can always be read any time during the simulation through `ns3::attributes` and object getters, such as `GetMass()`, `GetWeight()`, `GetArea()`, and `GetDragCoefficient()`.

Drones can be grouped together in `ns3::DroneContainer` and can be statically referenced by their unique identifier in the simulation through `ns3::DroneList`.

According to the peculiar workflow of ns-3, to properly instantiate an `ns3::Drone` object, an `ns3::DroneContainer` is needed. The creation process consists of a call to the `ns3::Object::CreateObject` function, where T is replaced with `ns3::Drone`. In order to ensure full compatibility with all ns-3 methods involving `ns3::Node` or `ns3::NodeContainer` classes, a dedicated mechanism has been developed. Every `ns3::Drone` goes through a static cast procedure, i.e., `ns3::StaticCast`, which generates an `ns3::Node` object that is pushed into an `ns3::NodeContainer`. In this way, for each drone, two smart pointers refer to the same memory location but cast to the two required types. Besides, the `ns3::DroneContainer` class provides a specific iterator, together with two further methods which return the number of instantiated drones and a smart pointer to each. It is worth mentioning that only drones must use an `ns3::DroneContainer`, while ZSPs, together with other entities, must still be modeled as `ns3::Node` objects.

1) **Peripheral state**—which can either be set to ON, OFF, or IDLE. This simple finite state machine (FSM) allows the development of intelligent algorithms to find optimal energy management.
2) **Power consumption**—how much instantaneous power is required by the peripheral, expressed in Watts, for each state.
3) **Reference RoIs**—where the peripheral should be operating. This is extremely useful for modeling certain peripherals and missions that depend on particular regions in space. For instance, a photograph camera can be used and activated only when the drone is in the RoI, thus leading to an optimized use of power, storage, and data. If this parameter is not defined, the reference peripheral will be active over time.

`ns3::DronePeripheral` has been specialized in two subclasses.

`ns3::StoragePeripheral` represents a generic storage device characterized by an attribute describing the initial
amount of memory, which can be traced at runtime to record the empty space left. Device total capacity can be queried through GetCapacity() method. If a drone peripheral, e.g., a camera or any other sensor, wants to interact with the storage, it is possible to request space by specifying the amount of data through Alloc(). The inverse can be done with Free(). These operations can fail if there is no memory left or there are no data to be freed, respectively. For this reason, a boolean value is returned by these methods to indicate if the requested operation was successfully completed or not. In this work, it is assumed that at most one ns3::StoragePeripheral is installed on each drone.

ns3::InputPeripheral describes a generic input device, characterized by an acquisition DataRate, constant over a DataAcquisitionTimeInterval. Once it is created, installed on a drone, and attached to a particular storage peripheral with Install() method, the storage peripheral of reference can be changed with SetStorage(). If the peripheral is ON, AcquireData() simulates data acquisition at the given DataRate.

These two peripheral types are strongly connected since an ns3::InputPeripheral can offload acquired data to an ns3::StoragePeripheral through a boolean attribute. Nonetheless, the association between input and storage is not mandatory. In fact, in a real-world scenario, an ns3::InputPeripheral can deliver data directly to a processing unit or to a remote host, thus neglecting the need to permanently store the information.

A complete list of the attributes of these classes is given in Table III. It is worth specifying that all peripherals hold a reference to the drone they are equipped with.

Moreover, for each ns3::Drone, an ns3::DronePeripheralContainer object is created to manage all its peripherals. This container is responsible for the creation of peripherals and, through the ns3::DronePeripheralContainer::InstallAll() method, sets the correct references to the host drone, and, eventually, to the target ns3::StoragePeripheral.

2) Mechanics and Energy Consumption: ns-3 already models and manages all the energy-related aspects, such as consumption, harvesting, and monitoring, through the abstract class ns3::EnergySource. Although there is no specific energy source model available that is suitable for drones, the ns3::LiIonEnergySource is sufficiently general to be employed for simulation purposes [25], [26].

The ns3::DeviceEnergyModel class describes the ns3::NetDevice energy consumption by means of the drawn current. The installation procedure is eased by the helper class ns3::DeviceEnergyModelHelper, which employs the Install() method that links an ns3::EnergySource to an ns3::NetDevice.

When the battery object is initialized, it schedules an ns3::Event, which calls ns3::EnergySource::CalculateTotalCurrent(). This function retrieves the current drawn of every device associated with the ns3::EnergySource, by calling ns3::DeviceEnergyModel::GetCurrentA(). Subsequently, the energy consumption value is calculated and subtracted from the remaining one. Finally, the ns3::Event reschedules itself.

In this work a specialization of ns3::DeviceEnergyModel, i.e., ns3::DroneEnergyModel, is developed along with the helper class ns3::DroneEnergyModelHelper. Given a simulation duration T, the model splits it into n = 1, . . . , N equal discrete intervals. The power consumption model of the drone flying at speed v[n] = (v_x[n], v_y[n], v_z[n]), in the nth time slot, is the following [27]:

\[ P_{\text{UAV}}[n] = P_{\text{level}}[n] + P_{\text{vertical}}[n] + P_{\text{drag}}[n] \]  

where

\[ P_{\text{level}}[n] = \frac{W^2}{\sqrt{2} \rho A} \frac{1}{\sqrt{\Omega + \sqrt{\Omega^2 + 4V_h^4}}} \]  

being

\[ \Omega = \| (v_x[n], v_y[n]) \|^2 \]  

\[ P_{\text{vertical}}[n] = Wv_z[n] \]  

\[ P_{\text{drag}}[n] = \frac{1}{8} C_{DD} \rho A \| (v_x[n], v_y[n]) \|^3 \]  

\[ W = mg \] with m defining the mass of the drone and g as the gravitational acceleration. Moreover, \( \rho \) is the air density, A is the total rotor disk area, \( C_{DD} \) is the profile drag coefficient depending on the geometry of the rotor blades, and \( V_h = \sqrt{W/(2\rho A)} \) uses parameters to calculate the power required for hovering operations.

The energy model can be aggregated to a drone by means of the ns3::DroneEnergyModelHelper, which provides an Install() method that aggregates it to ns3::Drone. In this way, it is possible to simulate the energy characteristics of a drone, both for its mechanics and its peripherals, in addition to its networking operations.

Such mechanical power consumption model is implemented in the method ns3::DroneEnergyModel::GetPower(). Similarly, the method ns3::DroneEnergyModel::GetPeripheralsPowerConsumption()
returns the cumulative power consumption of all peripherals on board.

The ns3::DroneEnergyModel object, registered as a new ns3::TypeId with no attributes, implements ns3::DoGetCurrentA() inherited from ns3::Device EnergyModel. Such method returns the total drawn current related to both mechanics and peripherals, in addition to networking operations. The energy model can be aggregated to a drone by means of DroneEnergyModelHelper, which provides an Install() method that aggregates it to ns3::Drone.

It is worth specifying that ns3::DroneEnergyModel Helper implements the installation procedure in a different manner with respect to its parent, i.e., ns3::DeviceEnergyModelHelper. In fact, the ns3::Drone EnergyModelHelper::Install() method links an ns3::EnergySource to an ns3::Drone instead of an ns3::NetDevice. This aspect distinguishes the aim of IoD-Sim from the ns-3 one: to simulate all the relevant aspects of the drone, beyond the networking perspective. This justifies the implementation divergence from the ns-3 main goals.

During the simulation, it is possible that the drone runs out of energy. To this end, the event is propagated through the execution of HandleEnergyDepletion() of the energy model, for which the time of depletion is logged for successive data analysis.

C. Other Simulation Entities: ZSPs and Remotes

Entities beyond ns3::Drone are ZSPs and Remotes. ZSPs are smart entities, modeled as ns3::Node objects, equipped with multiple ns3::NetDevice which provide multiprotocol radio access, thus enabling communications between drones and the rest of the Internet. Typically, they are configured as ground entities that maintain a constant position in time [1], by means of ns3::ConstantPosition MobilityModel. Nonetheless, in IoD-Sim, their mobility model can be customized to fit simulation purposes, envisioning the adoption of dynamic wireless infrastructure proposed in 5G and Beyond architectures. Remotes, instead, are ns3::Node objects with no mobility model and only rely on installed applications which provide remote services to consumers. Remotes and ZSPs are interconnected through a backbone, simplified as a carrier sense multiple access (CSMA)-based bus network, that represents the Internet. This architecture allows service provisioning on different classes of nodes, employing Remotes in the case of applications with high computational costs, e.g., multimedia data processing and ZSPs in the case of low latency requirements, e.g., traffic management.

D. Mobility

ns-3 provides a basic foundation to represent the movement of drones (e.g., ns3::WaypointMobilityModel, ns3::ConstantAccelerationMobilityModel, and ns3::ConstantVelocityMobilityModel). However, an important gap arises when such models are analyzed in detail: none of the available ones is able to construct a curve trajectory that take into account how much a spot is relevant for the mission plan. Another aspect to consider is that models such as ns3::WaypointMobilityModel couples the position of the drone with a given time instant, without taking into account the limitations imposed by the maximum speed of the UAV. Therefore, if the user does not properly design the path, this could lead to a simulation which does not reflect the reality. Moreover, in the setup phase, it is necessary to specify all the points that create the trajectory.

To overcome these limitations, dedicated mobility models have been developed. In particular, the trajectory has been modeled using Bézier curves by specifying a set of PpoI. These are decoupled from the time of arrival, and the resulting trajectory is bounded to the mechanical characteristics of the drone. A specific structure implemented in IoD-Sim, namely, ns3::CurvePoint, describes the 3-D position vector of the Bézier curve together with the distances from the previous point and the starting one. Besides, a container object, i.e., ns3::Curve, is in charge of managing the points of the curve, i.e., ns3::CurvePoint, that are defined according to the interest points contained in a ns3::FlightPlan. When a ns3::Curve is instantiated, it populates the container according to the following.

Let $P = \{P_0, P_1, \ldots, P_{N-1}\}$ with $P_i \in \mathbb{R}^3 \quad \forall i = 0, \ldots, N - 1$ be an ordered sequence of $N$ interest points, $I = \{l_0, l_1, \ldots, l_{N-1}\}$, $l_i \in \mathbb{N}^+$, the interest level associated to each point, $\Lambda = (\sum_{i=0}^{N-1} l_i) - 1$ and $L_i = \sum_{h=0}^{i-1} l_h$. The Trajectory Generator can be expressed as

$$G(t) = \sum_{i=0}^{N-1} P_i \sum_{j=0}^{l_i-1} \left( \frac{\Lambda}{L_i + j} \right) (1 - t)^{L_i-j} t^{L_i+1}, \quad t \in [0, 1].$$

(6)

It is worth noting that (6) is a revised version of the original Bézier equation, which does not practically allow to reach the interest points, except for the first and last one. An increment in the interest level $l$ turns into a trajectory that passes closer to that point, as illustrated in Fig. 3. A special case takes place when $l = 0$. A specific mechanism is provided to split the trajectory into two contiguous curves so that the drone is forced.
to fly over them. In this case, a restTime can be defined to set the hovering duration in seconds.

Finally, the obtained trajectory is used by the new implemented models, i.e., ns3::ConstantAccelerationDroneMobilityModel and ns3::ParametricSpeedDroneMobilityModel.

1) Constant Acceleration Drone Mobility Model: This mobility model employs (6) and the uniform acceleration motion law to retrieve the points of the desired trajectory. Since the speed of the drone cannot increase indefinitely, after the maximum speed is reached, the uniform linear motion law is adopted. This object is implemented as an ns-3 model, and hence has its own TypeId attributes described in Table IV.

In each instant of the simulation, IoD-Sim calls two methods, DoGetPosition() and DoGetPositionVelocity(). They return both the position and the speed at current time of the drone, that is recomputed thanks to the Update() method.

2) Parametric Speed Drone Mobility Model: Similarly to Constant Acceleration Drone Mobility Model, this mobility model is implemented as a ns-3 model with its own TypeId. However, this takes a v(t) speed profile in a polynomial form and, thanks to the modified Bézier (6), it retrieves the discretized trajectory. To ease the implementation, a specific attribute, i.e., ns3::SpeedCoefficients, is introduced to serve as a container of the v(t) coefficients. These are elaborated (by employing the GSL) to constantly update the parameters by calling UpdateSpeed() and UpdatePosition(). A summary of the attributes of this mobility model is reported in Table V.

E. Applications

IoD-Sim offers simple applications that can be used to communicate telemetry from a drone to a ZSP or to a Remote by adopting client–server paradigm, via user datagram protocol (UDP). Moreover, relying on the same architecture, two TCP-based applications are available to enable reliable data transfer between hosts. Besides, a network address translation (NAT)-like application is provided to design relaying network architectures.

1) Telemetry Applications: These applications are modeled as classes named ns3::DroneClientApplication and ns3::DroneServerApplication. The model asks for the DestinationIpv4Address and a Port of the remote entity that hosts the server application. Data are sent every TransmissionInterval and, whereas the drone has a storage peripheral, it is possible to free an equivalent amount of memory space. The configuration parameters are summarized in Table VI.

When the application is started, through the ns3::Application::StartApplication() method, a UDP-based communication, employing application-level acknowledgments, takes place. It is worth specifying that the application is stateful in order to support the Rendezvous Process which discovers the application server in the network, if no address is given. This process starts with the client application in NEW state. Therefore, a HELLO packet is sent to the destination address (or in broadcast), thus implying a state transition in HELLO_SENT. If the application server receives such packet, it replies with an HELLO_ACK packet to confirm the reception. When the client receives the acknowledgment, its state changes again, into CONNECTED_ACK, which allows it to periodically send telemetry data. These packets are named UPDATE and UPDATE_ACK. The entire procedure is depicted in Fig. 4.

The JSON-encoded telemetry is periodically transmitted, through the SendPacket() method, and received by the application server, through the ReceivePacket() method. HELLO and UPDATE packets transport a payload which is formatted in JSON with ASCII encoding. Its content is a JSON object with the following properties.

1) The unique id of the drone in the simulation. This ensures that Drones communications can be tracked over complex scenarios.
2) An incremental sn that refers to the sequence number. It is used to easily check if a packet has been lost.
3) cmd that refers to the type of packet, whether if HELLO, UPDATE, or an acknowledgment.
4) gps coordinates with lat for latitude, lon for longitude, alt for altitude, and vel for the velocity vector.
For simulated drones, the GPS location refers to the virtual world coordinates. The UDP packet payload is summarized in Table VII. When the application is stopped, the StopApplication() method is called. Clearly, these applications are developed so that multiple instances can run concurrently on the same entity if different ports are specified. Moreover, they are independent of the particular communication technology adopted.

2) Generic Traffic Applications: These applications model a reliable data transfer between a client and a server, which are implemented as TcpPeriodicClientApplication and TcpEchoServerApplication objects, respectively. The aim is to transfer a certain amount of information between the two hosts according to the specified PayloadSize, expressed in bytes, and TransmissionFrequency, measured in Hz, set on the client. The server is characterized by a socket, composed by a listening Address and Port.

TABLE VII
UDP PAYLOAD

| Field Name | Data Type  | Description                           |
|------------|------------|---------------------------------------|
| id         | Unsigned Integer 32-bit | ns-3 Global Node Identifier          |
| sn         | Unsigned Integer 32-bit | Packet Sequence Number               |
| cmd        | String      | Packet Type                           |
| gps        | Object      | Drone location in space               |
| lat        | Double      | Drone latitude                        |
| lon        | Double      | Drone longitude                       |
| alt        | Double      | Drone altitude                        |
| vel        | Array of 32-bit Integers | Drone velocity in m/s               |

To facilitate traffic analysis, each packet has a Protocol Data Unit, formed by a 12 bytes header, and the payload. The former contains information-level sequence number and the timestamp of creation; the latter is characterized by a recurring sequence of 16 bits that is incremented over time. These applications provide dedicated TraceSource objects that notify communication-related events such as new/closed connections and sent/received packets.

An additional TCP-based client has been created to support drones that are typically equipped with a StoragePeripheral. To this end, TcpStorageClientApplication monitors the storage and, if memory is used, it transfers data to the remote server. If the transfer is acknowledged, memory is freed. This mechanism is relevant when drones are equipped with limited onboard memory. Indeed, the client can be used to transfer as much data as possible over the wireless medium to prevent out-of-memory events.

3) Relaying Application: The Relaying Application is implemented through the class ns3::NatApplication. It is a specialized networking application that, given an InternalNetDeviceId and an External NetDeviceId, provides an NAT-like mechanism to a set of drones placed in an internal network. The NetDeviceId is a numerical identifier that uniquely points to a network device mounted on the drone.

During initialization, i.e., DoInitialize() method, the application modifies the static routing table of the internal network device to redirect all traffic to the loopback device. A specific callback, namely, RecvPktFromNetDev(), notifies when a new frame arrives. It contains information, such as the Internet Assigned Numbers Authority standard L3 protocol identifier and the sender/receiver MAC addresses.

The NAT forwarding behavior leverages a hash map, i.e., NAT Table, where an external port number is coupled with the source IP address and port. Inbound frames are forwarded to the external network by replacing this information with one of the relaying drone. The same rationale is applied for frames received from the external network.

F. Scenario Configuration Interface

The Scenario Configuration Interface is an abstraction layer that allows the configuration of the entire simulation by means of JSON files. Indeed, they can be decoded and validated through RapidJSON in order to setup the simulation models. The output data classes are then used by the General Purpose Scenario to initialize objects that define the environment, the entities, and the simulator engine. To this end, the set of all objects that are used to characterize a scenario can be grouped into three categories.

1) Configuration Objects: Models that store parameters in a structured way, easily accessible in the C++ language.
2) **Configuration Helpers:** Checkers and decoders with the goal to produce a Configuration object or throw an error message.

3) **Simulation Helpers:** Objects that help organize pointers to structures commonly found in scenario development. They are used in the protocol stack matrix, shown in Table IX.

Additionally, **Factory Helpers** are defined as weakly-coupled extensions to ns-3 internal data structures to ease their initialization. They are made to minimize modifications made to the ns-3 core framework, which is used by IoD-Sim. The entire system has been made extensible by design so that it is possible to support further technologies and configurations with the addition of new configuration objects and helpers as needed. In this way, it is possible to further develop high-level configuration interfaces able to setup scenarios and hence to ease the design activity undertaken by the user.

1) **Scenario Configuration Objects and Helpers:** The core of the abstraction layer is the `ns3::ScenarioConfigurationHelper`, a low-level object that directly deals with the JSON configuration file. This helper returns a set of specific data classes that contain exclusively the parameters required to configure IoD-Sim models. Each of them is also loosely coupled with a JSON validator and parser, also known as configuration helpers. The information embedded in these classes is then deserialized and employed by higher-level objects.

1) `ns3::ModelConfiguration` describes `ns3::TypeId` objects through key-value pairs that reference the model attributes.

2) `ns3::EntityConfiguration` describes an entity, whether it is a `Drone`, a ZSP, or a `Remote`. The object retrieves and stores all parameters related to the `ns3::NetDevice` to be installed on the entity, the `Mobility Model` to be applied, and the `Applications`. Optionally, if the entity is a `Drone` there can be defined the mechanics, the battery, and the peripherals. Its parser is called `ns3::EntityConfigurationHelper`.

3) `ns3::RemoteConfiguration` denotes key characteristics of `Remotes`. Specifically, a remote needs to know the global network layer ID of reference and the configurations of applications to be installed. Its parser is `ns3::RemoteConfigurationHelper`.

4) `ns3::PhyLayerConfiguration` defines the required parameters needed to configure a PHY layer. It is the parent and interface of `ns3::LtePhyLayerConfiguration` and `ns3::WifiPhyLayerConfiguration` data classes. Its parser is `ns3::PhyLayerConfigurationHelper`.

5) The `ns3::LtePhyLayerConfiguration` gets all the information needed to setup a PHY layer for LTE, such as its propagation loss model and its spectrum model.

6) `ns3::WifiPhyLayerConfiguration` sets up the PHY layer of a Wi-Fi-based protocol stack. The PHY layer configuration requires the higher-level Wi-Fi standard to be used, the antenna Rx gain, the data rate, the propagation delay, and loss models.

7) `ns3::MacLayerConfiguration` collects the required parameters needed to configure a MAC Layer. It is the parent and interface of `ns3::WifiMacLayerConfiguration`. Its parser is `ns3::MacLayerConfigurationHelper`.

8) `ns3::WifiMacLayerConfiguration` configures a Wi-Fi basic service set (BSS). The service set identifier (SSID) and access point parameters are defined to create its basic infrastructure.

9) `ns3::NetworkLayerConfiguration` defines the required parameters needed to configure the appropriate network layer. It is parent to the `ns3::Ipv4NetworkLayerConfiguration`. Its parser is named `ns3::NetworkConfigurationHelper`.

10) `ns3::Ipv4NetworkLayerConfiguration` stores the network address and mask of the configured IPv4 Layer in the configuration file.

11) `ns3::LteBearerConfiguration` decodes all the relevant parameters for an LTE bearer, such as its type and the Quality of Service (QoS) defined as a tuple of Guaranteed Bit Rate and Maximum Bit Rate.
12) `ns3::LteNetDeviceConfiguration` collects the information needed by an LTE network device, such as its bearers. The role of the network device is then detected, whether it is a user equipment or an eNB.

13) `ns3::NetDeviceConfiguration` defines for a generic network device. The main parameter stored is the global network layer ID, which is used to detect the stack and network to be attached when the network device is created and installed on a `Node`. A specific configuration for Wi-Fi network devices is handled by `ns3::WifiNetDeviceConfiguration` with relevant MAC data to connect to the BSS.

2) Scenario Simulation Helpers: To enable complex scenarios that are related to the future IoT communication paradigms, IoD-Sim enables the simulation of IoD networks in which multiple telecommunication protocols are used at the same time, both for the drones and the ZSPs. Currently, IoD-Sim supports two communication technologies that can be used concurrently: LTE and the IEEE 802.11 family. Each protocol stack must be applied to a dedicated network device, i.e., `ns3::NetDevice`. The architecture of the simulator has been designed so that it eases the configuration phase.

In order to facilitate the implementation and the installation of protocol stacks on IoD entities, additional helpers named `Simulation Helpers` have been developed to arrange the necessary common infrastructure to simulate communications among nodes. Thus, the developed `Simulation Helpers` are as follows.

1) `ns3::WifiPhySimulationHelper`, that initializes the PHY layer of a Wi-Fi-based protocol stack.
2) `ns3::WifiMacSimulationHelper`, that creates the objects related to IEEE 802.11 MAC.
3) `ns3::LtePhySimulationHelper`, that allocates the necessary resources to enable LTE communications.
4) `ns3::Ipv4SimulationHelper`, that manages IPv4 networks for each protocol stack.

All the aforementioned can cooperate with the existing helpers in `ns-3`, such as `ns3::LteHelper`, `ns3::WifiHelper`, `ns3::YansWifiPhyHelper`, and `ns3::WifiMacHelper`.

3) General Purpose Scenario: A flexible and highly dynamic `General Purpose Scenario` has been developed in order to setup scenario’s entities and, at the same time, to provide abstractions which minimize the effort from a programming perspective. It is fully dependent on a semantic analyzer and allows the entire simulation platform to be compiled beforehand, providing ways to dynamically reconfigure the scenario at run-time. Its development started from the analysis and the detection of a common structure typically followed by the Open Systems Interconnection protocol stack. The entire workflow, depicted in Fig. 5, is described hereby.

`General Purpose Scenario` is composed of two main parts: 1) configuration and 2) run. `Scenario configuration`, executed through the constructor `Scenario()`, is interleaved with the `Scenario Configuration Interface`. The `run` part is identified by `operator()()` which is characterized by minimal C++ code that starts the ns-3 simulator engine. Moreover, it shows the progress status on the console and, optionally, it saves messages to a log file.

The `General Purpose Scenario` requires the initialization of the `Scenario Configuration Interface` through a JSON configuration file. Once the file is decoded, the number of entities are retrieved to create the initial structures, such as an `ns3::DroneContainer` and four `ns3::NodeContainer` objects. They keep track of ZSPs, Remotes, and nodes that participate in the Backbone Network.

Once the entities are created, they are registered to their respective global lists, such as `ns3::DroneList`, `ns3::ZspList`, and `ns3::RemoteList`.

After entity creation, the ns-3 static configuration parameters are applied to the simulation. The method is called `ApplyStaticConfig()`. These parameters are a set of key-value pairs that represent certain features of ns-3 models.

World definition is made through `ConfigureWorld()` method. It is related to the configuration of `Buildings` and `RoIs`. The virtual world setup is then followed by the configuration of PHY, MAC, and Network global layers.

As for the PHY layer part, if it is made for a Wi-Fi communication stack, the `ns3::WifiPhySimulationHelper` is employed with the specifications stored in `ns3::WifiPhyLayerConfiguration`. If the PHY layer is for LTE, instead, the `ns3::LtePhySimulationHelper` is setup with `ns3::LtePhyLayerConfiguration` parameters. The same procedure is applied for the global MAC layer configuration. The global `Network` layer is managed by `ns3::Ipv4SimulationHelper` for IPv4 networks with the specifications given by `ns3::NetworkLayerConfiguration`, i.e., network address, mask, and a default route.

Global stacks are then linked to the configured entities. Moreover, for LTE devices, the bearer is created to ensure that applications have a logical communication channel with desired properties. When the entity network configuration is done, the mobility model is configured and the applications
are installed. Furthermore, if the entity is a Drone, its peripherals are installed, together with the associated energy model.

Once all entities are ready, the virtual Internet backbone is configured. A CSMA bus is made for the backbone network, identified with address 200.0.0.0/8. Hosts that can be part of this backbone network are Remotes, Packet Gateways in the case of an LTE core network, or other routers in the case of the presence of a Wi-Fi BSS.

Finally, in the case of LTE networks, their Radio Environment Maps are setup to generate images that represent the radiation map of the radio access network (RAN).

4) JSON Configuration Schema: The entire scenario has been made parametric through the use of a JSON configuration file. Requested at startup, it is decoded and employed to configure and execute the simulation.

In this work, the following configuration schema has been chosen for the General Purpose Scenario.

1) name: A mandatory string representing the scenario name.

2) dryRun: An optional boolean to run only the semantic analyzer and check that the configuration file and model setup is valid. By default, it is set to false.

3) resultsPath: A mandatory string representing an existing path to store simulation output files.

4) logOnFile: A mandatory boolean to output scenario logging information on a file or on standard output.

5) duration: A mandatory integer that specifies the simulation duration in seconds.

6) staticNs3Config: A mandatory array of objects, each with name and value strings, to address ns-3 static configuration parameters. The array can be empty.

7) world: An optional object containing the description of the simulated space, in particular, whether to place buildings or RoIs.

8) phyLayer: A mandatory array of objects, each representing a PHY layer configuration to be used in the scenario. Each PHY object declares its type, which is a mandatory string. The chosen type must be supported by the semantic analyzer. Additional parameters are specific to the kind of PHY layer being configured, most notable are the chosen propagation delay model and the propagation loss model.

9) macLayer: Its description is similar to phyLayer.

10) networkLayer: Its description is similar to phyLayer.

11) drones: A mandatory array of objects, each representing a drone to be simulated. A drone requires the following properties to be configured: at least one netDevices in order to link it to a protocol stack and setup its network address assignment, a mobilityModel according to the ones available on IoD-Sim, at least one application that can be installed on a drone, a mechanics to define mechanical properties, and a battery. Optionally, a peripherals array can also be specified in order to equip I/O devices to the drone with a specific PowerConsumption indication. They may also be activated by specifying the RoI through RoITrigger parameter.

12) ZSPs: Its description is similar to drones.

13) remotes: A mandatory array of objects, each representing a remote that is described by its set of applications.

14) logComponents: A mandatory array of strings to enable log components available in IoD-Sim.

An example of JSON configuration file that realizes a simple scenario is shown in Fig. 6.

VI. SIMULATION DEVELOPMENT PLATFORM

The Scenario Configuration Interface, discussed in the previous section, eases the design and configuration of complex scenarios from a high-level perspective. Indeed, JSON greatly facilitates management and maintainability thanks to its dry and human-readable syntax. However, the user experience is still hindered by the following.

1) As IoD-Sim grows in size and introduces more complex and powerful models over time, the learning curve to effectively use this simulator steepens.

2) This project is continuously developed and upgraded with new features, technologies, and standards. A high-level abstraction helps reduce the barrier for scenario developers in approaching new features and the required effort to implement a scenario.

3) A general purpose configuration interface, provided in the form of JSON-encoded files, does not give any visual clue on scenario design. Indeed, plain text files alone require low-level knowledge of the simulator, thus, implying that the users have to rely on their experience and imagination to effectively know all the aspects related to a complex scenario configuration, such as the number of drones, their trajectories, their purpose, their equipment, the topology of the ground infrastructure, and the services exposed by remote nodes.

4) Error reporting messages cannot be easily understood by end-users, forcing the use of a debugger to isolate the problem. Therefore, a semantic analysis would be beneficial to detect problems at scenario configuration.

To address all the points above, the IoD-Sim Simulation Development Platform provides a set of extensions, i.e., the Report Module, output files for data analysis, and standalone applications for scenario design, such as Airflow. These tools ease scenario design and analysis, thus, ensuring that IoD-Sim can be easily introduced to newcomers, especially university students and researchers.

A. Report Module

The Report Module, illustrated in Fig. 7, is an extension of IoD-Sim which stores data at run-time and elaborates, at the end of simulation, a comprehensive summary. The aim of the extension is to introspect simulator’s data structures to gather relevant data to be reported (e.g., data traffic, trajectory, and telemetry). To provide a final report that is both
human and machine readable, the XML format has been chosen. Therefore, a schema is defined to describe the expected structure of the produced file.

```xml
{
  "name": "iod_sim_ftw",
  "resultsPath": "./results/",
  "logOnFile": true,
  "duration": 50,
  "staticNs3Config": [..],
  "world": {
    "buildings": [
      { "type": "residential",
        "walls": "concreteWithNoWindows",
        "boundaries": [0.0, 70.0, 0.0, 70.0, 0.0, 20.0],
        "floors": 0,
        "rooms": [5, 1]
      },
    ],
    "RegionsOfInterest": [ std::vector<DoubleVector>
      [170.0, 340.0, 180.0, 250.0, 0.0, 15.0]
    ],
    "PHY": [
      { "type": "lte",
        "channel": {
          "propagationLossModel": [..],
          "spectrumModel": [..]
        }
      },
    ],
    "MAC": [..],
    "networkLayer": {
      "type": "IP4",
      "IPv4NetworkLayerConfiguration"
    },
    "drones": [
      { "type": "lte",
        "networkLayer": 0,
        "role": "UE",
        "bearers": [..]
      },
      "MobilityModel": [..],
      "applications": [..],
      "Drones": [..],
      "peripherals": [..]
    ],
    "Zsps": [..],
    "remotes": [..]
  },
  "logComponents": [..]
}
```

Fig. 6. Excerpt of scenario configuration with an overlay of the models associated to the analyzed parts.

More insights about the structure of the proposed extension are provided hereby. The root XML element, i.e., Simulation, represents the summary of a scenario previously executed. The attributes that characterize the simulation are scenario, which is a string that carries the name of the scenario that was executed, and executedAt, which reports the date and time of execution of this simulation. Moreover, Simulation presents further information about simulation results, such as its duration, which is reported in real and virtual time, World, which contains the Buildings and InterestRegions, and entities containers.

The first of these containers is Zsps, which is a complex XML type that summarizes each ZSP through position described by the 3-D coordinates, and NetDevices, which is a list of configured network devices. Each of them is described by structures that represent the configuration of the PHY, MAC, and network layers, together with the data traffic. Each captured packet is expressed by direction, length in bytes, timestamp, and textual representation of the payload.

Similarly, Drones summarizes the state of each Drone. This structure maintains the NetDevices already discussed for Zsps. Additionally, particular characteristics of drones are reported, such as trajectory and the set of onboard Peripherals. The former is defined by a list of points, each of them with its own timestamp. The latter reports the characteristics of the used peripherals type.

Finally, Remotes are described only by their NetDevices.

This output XML file is put together with other files relevant to the simulation in the results directory.

B. Results Aggregator

Log Files gather all the relevant information and debug messages about the internal components of the simulation. Primarily, the General Purpose Scenario emits progress.log and Iod Sim.log files. The former is the output of the progress information messages that are also delivered on the standard output during scenario execution.
The latter contains all debug messages coming from the different internal components of IoD-Sim. The log components can be enabled by specifying them in the logComponents field of the scenario configuration JSON file.

`progress.log` file starts by determining the current date and time of the start of the simulation. For each second, it prints a status report on a single line. The status report presents the following fields.

1) The simulation time instant at which the report is referring to.
2) The speed up in simulating the scenario with respect to real time. This is dependent on the simulator performance and how many events are elaborated.
3) The number of events processed in the time interval relative to the previous status report.

The file then ends with the current date and time and the duration of the simulation as Elapsed wall clock.

Trace Files are ASCII-encoded text files that record all the activities regarding a specific Network Device. All the traces are bounded by what is sent or received at the MAC layer. A Trace File name is composed of three fields, separated by a hyphen: 1) the global layer name; 2) the unique identifier of the host in the network; and 3) the unique identifier of the host network device. For instance, Internet-2-1.tr indicates that the trace has been done on the first network device of the second host in the virtual Internet network.

LTE Log Files are ASCII-encoded text files that represent a series of statistics on relevant key performance indexes (KPIs). These log files are focused on specific low-level layers of the LTE stack, particularly PHY, MAC, Radio Link Control, and Packet Data Convergence Protocol. For each layer, there are two separate trace files: one for downlink and one for uplink communications. As part of the LTE Log Files, there are also PCAP traces of the SI-U interface that links the RAN with the Evolved Packet Core.

PCAP Files are well-known files that record network activity in the PCAP format and contain the traffic that occurred on a certain network device of a host. The filename format is similar to Trace Files. Due to the fact that these files are binary, a suitable decoder should be used to explore the data structure. A popular decoder is the libpcap open-source project, used by frameworks for PCAP data analysis, e.g., Scapy, and graphical user interface (GUI) programs such as Wireshark. As these PCAP Files are generated by a simulation, each captured frame is marked with the relative timestamp of the simulation. Therefore, each PCAP File starts with the transmission/reception of captured frames at 0 s.

C. Airflow

Airflow is a high-level abstraction tool that gives visual clues during simulation design, thus enriching the user experience, especially for newcomers. Airflow has been developed on top of Splash, a specialized transpiler for IoD-Sim. It scans the source code of the simulator and outputs visual blocks that can be referenced in the Core Editor to configure a scenario. Thanks to the GUI editor, a scenario can be exported into a JSON file that can be interpreted by IoD-Sim Scenario Configuration Interface. From a software design standpoint, as illustrated in Fig. 8, the Airflow project is entirely decoupled from IoD-Sim. Its integration with the simulator relies on interfaces that enable bidirectional communications.

1) Splash: Splash is a middleware that analyzes IoD-Sim source code and translates ns-3 models into visual block code used by Airflow. These blocks can be added to the editor as external packages. Splash enables the decoupling mechanism, able to ensure that Airflow and IoD-Sim can be developed asynchronously and updated when needed.

In particular, it accomplishes the following tasks.

1) Parses the source code of IoD-Sim by relying on Clang lexical and syntax analyzers, producing the abstract syntax tree (AST) that is stored into a binary file.
2) Scans the AST to find relevant simulation models, excluding internal structures and routines that are not relevant for the design of a scenario. This information is then encoded in an intermediate representation (IR).
3) Optimizes the IR by solving model hierarchies and removing redundancies.
4) Generates Python code that describes the models as Airflow visual blocks. This output can then be moved to the Airflow project folder for integration.

Concretely, this pipeline works as follows. The script splash.sh can be executed by passing the IoD-Sim project directory as an argument. The program then searches for any relevant C++ source code files in it. This process is eased by the ns-3 convention: models have the suffix -model.cc, -manager.cc, -mac.cc, and -application.cc in their filenames. To this end, other files are filtered out to optimize parsing operations and to prevent the exposure of the simulator’s internal structures.

For each file found, the clang command is used to analyze the source code and solve any include directives needed by the preprocessor. Finally, the output is an AST, which is encoded...
This program relies on an interactive command-line application, on boost-json to serialize C++ data structures in JSON, and on libclang to read the AST. The application requires the PCH file path as input with the output directory path in order to store the IRs. These IR files are encoded into JSON to ensure software interoperability and readability.

Once the command-line program is executed, the entire translation unit of the AST is scanned in order to look up for any model used in the simulator. A custom tree-traversal algorithm is used to optimize the parse time. It works as a hybrid implementation of the classical Breadth-first and Depth-first search algorithms. A high-level representation of the translation unit is given in Fig. 9. The key feature of this approach is the speed up introduced by the algorithm. In fact, it first traverses the tree using Depth-first to find the depth at which one or more ns3::TypeId can be found, and then, uses Breadth-first to analyze each model at the same depth. The same strategy is applied to extract all the attributes relevant to the simulator model. Each model is represented and exported into a JSON file having the following structure: the name of the parent model, the model name, and a list of attributes, each one described by a name, an optional description, and the ns-3 data type that characterizes it. Once the entire model hierarchy is solved and optimized, the attributes are copied from parent to children, if any. Then, a code generator is executed to create the visual blocks for the editor GUI. Each block name reflects the model’s one and the attributes are considered as block input parameters. The generated Python code is interpreted by the GUI to display a visual block with the model name as its title and model attributes as its inputs, as illustrated in Fig. 10.

2) Graphical User Interface: The Airflow GUI, shown in Fig. 10, is based on the open-source engine named Ryven, which is a dynamic runtime, flow-based visual programming environment for Python scripts. It offers: 1) a central rendering view to place blocks and link them together; 2) a settings area to customize options; 3) a variable management section to include and store data that can be integrated with the flow; and 4) a console to report errors. Ryven includes additional features to optionally debug internal routines with the help of console messages. Moreover, thanks to its modular design, it allows blocks generated by Splash to be aggregated into packages. Ryven has been deeply extended to interoperate with IoD-Sim, especially for its compatibility with the Scenario Configuration Interface.

The user interface is organized into the following components:

1) A menu bar at the top of the GUI window.
2) A Console on the left in order to monitor errors and messages coming from Airflow or IoD-Sim. Informative messages are reported in blue, while errors are displayed in red.
3) A central workspace to design the scenario by placing blocks and connecting them together.
4) A settings panel on the right.

The menu bar is divided into three categories: with File it is possible to import Airflow packages to extend the user experience with third-party visual blocks. Moreover, it provides features to save the project or export it as an IoD-Sim configuration file. View offers graphical options, such as changing the theme, making a screenshot of the project, and tuning performance parameters. Finally, Debugging enables technical features to ease troubleshooting of the program, such as increasing verbosity level on the Console.

The central workspace is the canvas where blocks and links are placed by the user to design a scenario. A block, as depicted in Fig. 10, consists of a set of inputs and outputs. Each input and output can be connected to other outputs and inputs of other blocks, in order to create a tree. The root block is named Scenario. Each block has a different meaning and function. As a general overview, blocks can be divided into the following categories: 1) operators; 2) helpers; and 3) IoD-Sim models. Operators are built-in blocks that can be used to work with values, constants, and data structures. Instead, helpers are special blocks that ease the configuration of a scenario, i.e., entities, Wi-Fi, and LTE configuration blocks. Usually, blocks provide a single output without a label. This output delivers the information of the block, along with all its inputs, to the next connected block. Blocks can be added to the workspace by a specific menu that is shown by clicking with the right mouse button. Moreover, each block can be right-clicked to show its contextual menu that can be used: 1) to remove it; 2) to refresh it (and hence to read all its inputs again); and 3) to use some particular features available in certain blocks. For instance, toList offers some additional controls to add or remove inputs.

In the settings panel, it is possible to set the IoD-Sim path in order to enable interoperability features, such as checking the scenario configuration for errors, or running the scenario and reporting the status on the Console. These features can be used by clicking on the Build and Run buttons, respectively.

Finally, a variable manager can be used to create, store, and
VII. SIMULATION CAMPAIGN

This section demonstrates the huge potential of IoD-Sim by means of an extensive simulation campaign which investigates the many facets of IoD scenarios. First, the discussion explains how the simulation can be designed. Second, three different scenarios with increasing complexity are presented.

In particular, the first scenario discusses the use-case of telemetry with a few drones flying in an RoI, which follow customized trajectories while gathering data. The purpose of this scenario is to demonstrate that it is possible to monitor one or more variables with onboard sensors while estimating the energy consumption associated with flight dynamics.

The second scenario has a wider perspective since it focuses on surveying and monitoring activities, further completed with the acquisition of multimedia signals by each drone. The possible applications include several real-world use cases in the fields of civil engineering, smart agriculture, or environmental monitoring, e.g., coastal erosion and other slow phenomena. In fact, in this scenario, drones are on a mission in neighboring areas since it is assumed that the information of interest needs to be contextualized, i.e., must be gathered at the same time. Furthermore, this case investigates the possibilities enabled by different data storage capabilities of drones. Also, the offloading functionality of the acquired data avoids the overload/saturation of onboard available resources. Once data are gathered, they can be involved in offline post-processing, evaluation, and analysis.

The third scenario has been specifically designed to be the reference benchmark for IoD applications. It is set in the context of smart cities and involves clusters of low-power IoT sensors. This scenario models real-world applications and, hence, shadowing and pathloss phenomena are included, thanks to the adoption of propagation models that are influenced by the presence of buildings. In order to guarantee a reliable communication, drones are in charge of relaying traffic to ensure coverage to all sensors in the city.

A. Scenario Design

Airflow represents the foremost application for visual scenario development. To better understand how to design simulations, a simple configuration setup is provided hereby. The envisioned scenario considers a drone that follows an arc-like trajectory and communicates telemetry to a ZSP by means of Wi-Fi. Specifically, the drone acts as a station and the ZSP as an access point. The entire configuration is depicted in Fig. 11, where all the visual components, encompassed in the Airflow workspace, are properly setup and linked together. Starting from the right, the block Scenario glues some configuration input values, e.g., Name and Duration, with more complex components, such as: 1) PHY/MAC/NET Layers; 2) Drone List; and 3) ZSP List.

In particular, the communication layers are configured to implement the Wi-Fi stack. The WiFi PHY Layer object defines the PHY layer to be used with particular propagation and loss models. The WiFi MAC Layer, instead, specifies the SSID of the network and the Wi-Fi Manager object that handles the MAC control plane. Further, the IPv4 Network Layer determines the address and mask of the overlying network.

Both Drone List and ZSP List properties are connected to the simulated entities, namely, Drone and ZSP. These two components share different properties, such as Applications, Mobility Model, and Network Devices. However, the Drone block is also characterized by its unique features, i.e., Peripherals, Mechanics, and Battery. In this configuration, the ConstantPosition MobilityModel allows placing the ZSP at a fixed location, while the ParametricSpeedMobilityModel is employed to define the drone trajectory. In this regard, the Trajectory component, linked to the FlightPlan property of the mobility model, facilitates the design of the desired path.

The Network Devices property of both drone and ZSP is linked to a WiFi Net Device block. While
StaWifiMac characterizes the device of the former, ApWifiMac is associated with the latter. Finally, a LiIonEnergySource defines the power supply of the drone.

The development strategy discussed above represents the common ground for the design of the following three scenarios.

### B. Scenario #1—Telemetry

The first scenario, as depicted in Fig. 12, envisions three drones with the same mechanical characteristics, all equipped with an inertial measurement unit (IMU). In this scenario, drones are flying in the same RoI at a constant speed, following different trajectories. Moreover, a ZSP is deployed on the ground. The latter is released in $[60 \ 45]^T$, which continuously monitors drones’ operations by acquiring telemetry through Wi-Fi.

UAVs’ trajectories are based on the ParametricSpeedDroneMobilityModel, which is configured to guarantee a constant speed of 5, 3, and 4 m/s, respectively. They are also equipped with IMUs, which are generic drone peripherals that provide basic telemetry data to the ZSP thanks to a dedicated application, as mentioned in Section V-E1. It is worth specifying that drones’ IMUs have different power consumption, i.e., 12, 5, and 6 W.

The outcome of the simulation is hereby discussed. Figs. 13 and 14 depict the power consumption trend with respect to time and trajectories. In the former, the three curves share an initial peak which corresponds to the energy required to take off. Indeed, acquiring altitude requires more power than flying along the $xy$ plane, as highlighted. This phenomenon is further remarked in Drone #2 landing maneuver. It includes a little parabola that yields a peak in the last part of the associated curve of Fig. 13, which is also present in Fig. 14(b). After $\sim 10$ s, the drones reach and almost maintain a target altitude.
The corresponding power consumption, for Drones #1 and #3, is characterized by peaks due to hovering over the interest points for 1 and 3 s, respectively. These points are identified by the vertices of the snake-like and octagon-shaped trajectories. Instead, this phenomenon is not present on Drone #2, since its trajectory describes a continuous curve. When the drones enter the RoI, the peripherals become active, and hence, the IMUs power contribution is nonzero. Spikes can be noticed in the curves of Fig. 13, especially in Drones #1 and #2, since they are equipped with two more energy-demanding peripherals. As soon as drones exit such a region, the peripherals go into standby mode, which preserves energy.

Fig. 15 illustrates the measured received signal strength indicator (RSSI) of each drone during the mission. Measurements are carried out by the ZSP. In general, such values can be conceived as an assessment of ranging operations carried out by a single node when its position is fixed. From this Figure, it clearly emerges that, on average, Drones #1 and #2 maintain a better signal quality with respect to the UAV #3. Obviously, the higher altitude, and hence, the greater distance from the ZSP, worsens the communication quality due to the Friis propagation loss employed to model the fading effects in this scenario.

C. Scenario #2—Multimedia Signals Acquisition

The second scenario is depicted in Fig. 16. A swarm of four drones is in charge of acquiring multimedia signals in an operating area that is $10^6$ m$^2$ wide. Acquired data are stored onboard and off-loaded to a remote server as soon as the drone is able to communicate with a ground infrastructure. The latter, which allows data upload, is composed of three ZSPs, also referred to as eNBs, that are deployed on the ground in three different locations: [50 800]$^T$, [900 200]$^T$, and [700 900]$^T$, respectively. All the entities involved in the mission, which lasts 250 s, are equipped with LTE interfaces, where the Okumura-Hata propagation loss model has been employed. Drones follow snake-like trajectories, each different from the other in terms of amplitude and frequency. Nevertheless, they adopt the same mobility model with a constant acceleration of 4 m/s$^2$ and a maximum velocity between 15 and 20 m/s. Each drone is equipped with cameras that operate at different data rates, 2, 1.6, 1.3, and 1 Mbps, respectively. The communication between each UAV and the remote server is handled by Generic Traffic Applications (see Section V-E2), with a payload size of 1024 bytes and a TCP Max Segment Size of 1380 bytes.

In the same figure, it can be further observed the attachment of the drones to the ZSPs. Throughout the mission, Drones #2 and #3 remain linked to the same eNB, i.e., ZSP #2 and #1. On the other hand, UAV #1 and #4 perform a handover procedure which changes the reference ZSP from #1 to #2 and from #2 to #3, respectively. It is worth noting that despite Drone #1 takes off in the same area where Drone #2 lands, they are not attached to the same ZSP. Indeed, even if the two trajectories share the same direction, they have opposite verse: while one approaches an eNB, as the mission goes by, the other flies away from the ZSP without really getting closer to another one. Fig. 17 shows the throughput for each drone on the associated ZSP over time. It is shown that UAV #1 experiences an average data rate of $\sim 1$ Mbps until the handover procedure takes place, which increases this value by $\sim 50\%$. Similarly, the average throughput of Drone #4 is also ameliorated since it increases from $\sim 800$ kbps to $\sim 1.1$ Mbps. It is worth noting that there exists a pattern correspondence between the throughput and the occupied storage curves (see Fig. 18). This is particularly evident for Drones #3 and #4: when the occupied memory lowers and goes to zero, the data rate decreases as well, and tends to zero. Indeed, for the information causality principle, it is not possible that a larger amount of information is transmitted with respect to the stored one. Notice that this happens as long as the acquisition rate remains lower or equal to the channel capacity which, for instance, is not the case of Drone #1.

D. Scenario #3—Smart Cities

The third scenario reproduces a smart city context, in which drones are in charge of relaying traffic coming from clusters of ground users (GUs), using Wi-Fi technology, to a remote server over the Internet, through LTE. In this regard, the presence of buildings plays an important role both in trajectory design and in fading phenomena. The envisioned scenario is designed starting from the map of an urban area in the neighborhood of the Central Station of Bari, Puglia, Italy. The $xy$ coordinates: 1) are extracted from OpenStreetMap with the aid of OpenCV [28]; 2) rescaled according to their real profile; and 3) transposed into the spatial reference system of the simulator. Finally, the buildings’ heights are generated using a random variable uniformly distributed in [24, 30], which corresponds to the characteristic height (in meters) of the buildings in that area. As shown in Fig. 19, four GUs clusters of different size are present on the ground. Each of them is served by a drone, which relays the traffic by means of the NAT application discussed in Section V-E3. The entire simulation lasts 180 s and employs the ns3::HybridBuildingsPropagationLossModel to take into account the fading caused by the presence of buildings. It includes a combination of the Okumura-Hata model and COST231 for long-range communications, ITU-R P.1411 for short-range communications, and ITU-R P.1238 for indoor ones. This allows to support a wide range of frequencies spanning from 200 MHz to 2600 MHz. Moreover, each building is characterized by
a window per room and is assumed to be built with concrete walls. The Wi-Fi stack has been configured based on the 802.11ax standard operating at 2.4 GHz and is controlled by the `ns3::IdealWifiManager`, which allows to keep track of the SINR. Thanks to this mechanism, it is possible to always choose the best transmission mode to be used, i.e., a combination of modulation, coding scheme, and data rate.

As for the network level, each cluster is connected to its relay according to the 10.0.0.0/24 network address range, while LTE uses 7.0.0.0/8. Drones’ trajectories are designed to the layout of the streets in order to minimize the shadowing effects and maximize the Line of Sight with the GUs. Furthermore, the path also maximizes energy efficiency as the translation in the $xy$ plane is less costly when compared to changes of altitude. At each angle of the trajectory,
the drones pause for 1 s in order to simulate an accurate 90° yaw.

Accordingly, each relay drone flies at a constant altitude of 50 m at 5 m/s. Drones are equipped with the \texttt{ns3::NatApplication}, which implements a simple Port-based NAT strategy for UDP communications. Each GUs has a constant position and is equipped with a simple \texttt{ns3::UdpEchoClientApplication}, which periodically sends a packet of 1024 bytes to the remote address 200.0.0.1:1337 with a frequency of 10 Hz. Each packet is equipped with an application header that reports an incremental sequence number and the time of creation. Finally, the remote has a \texttt{ns3::DroneServerApplication}, which records via log messages the received packets.

The only ZSP, located at \([60, 120, 40]\)T, provides LTE access to the drones, thus allowing the communication with the remote host. Figs. 20 and 21 clearly show the advantage brought by the relay activity by the drones. In the relay case (Fig. 20), all the GUs experience an average latency of \(\sim 25 \text{ ms}\), a result that is achieved also thanks to the proposed trajectory design.

On the contrary, in absence of relay drones (see Fig. 21), while the GUs that are closer to the ZSP are affected by a latency similar to the previous case, the farther ones register a significant delay, which inevitably compromises the reliability of the link and, hence, the QoS. Nevertheless, this comes with a tradeoff as highlighted in Fig. 22, which shows the PLR in both cases. In the former, all nodes are able to transmit data to the remote, but with a loss ratio of \(\sim 10\%\) for the cluster #2 and #3. It is worth noting that this result can be further improved by properly optimizing the trajectory design to target the desired tradeoff. In the latter, instead, six nodes have 100\% PLR, which means that there is no exchange of data.

### E. Performance Evaluations

To evaluate the performance of the simulator, and hence its scalability, the performance metrics of the simulated scenarios are analyzed, and compared hereby. The runtime environment is characterized by the following hardware and software specifications: 1) Intel Xeon Bronze 3106 at 1.70 GHz with 16 cores and no hyper-threading; 2) RAM 92-GB DDR4 at 2666 MHz; 3) 7200-RPM hard drives; and 4) OS Fedora 35 on LXD container [29]. It is worth specifying that the present
TABLE X
COMPARISON OF THE TOTAL NUMBER OF EVENTS, THE REAL TIME TAKEN TO EXECUTE, AND THE SIMULATED TIME OF EACH SCENARIO

| Scenario # | Events [#] | real time [s] | Sim. Time [s] |
|------------|------------|---------------|---------------|
| 1          | 57,437     | 5             | 10            |
| 2          | 18,226,323 | 761           | 250           |
| 3 LTE      | 37,178,812 | 4,620         | 180           |
| 3 Wi-Fi & LTE | 28,903,306 | 2,858         | 180           |

Fig. 23. Performance evaluation of the different simulated scenarios. (a) Number of events. (b) Simulation speedup time.

assessment is made leveraging a single-core configuration, although multiprocessing support is available. To fairly compare the simulations, two metrics are selected. The former takes into account the number of events processed per second for each simulation, thus providing an insight related to the scenario complexity. The latter considers the ratio between the simulated time and the real time, thus, further addressing the complexity of the designed missions. Moreover, Table X summarizes the total number of events, the time taken to simulate (real time), and the simulated time of each scenario. It is worth noting that all scenarios are constructed differently and hence are difficult to compare. However, some clear indications can be derived from the following analysis. Indeed, Fig. 23 shows that in Scenario #1 the employment of Wi-Fi technology slows the number of events processed per second, which means that the complexity is higher. On the contrary, the adoption of LTE (either mixed with Wi-Fi) reduces the overall computational complexity. However, in the first case (Scenario #1), the speedup is greater with respect to the second case (remaining scenarios): this is due to the fact that the number of generated events is way lower. This is particularly evident in Scenario #3, where the simulation time and the number of GUs are the same, as shown in Table X. Overall, even if the number of actors increases when drone relays are employed (LTE and Wi-Fi), the lower number of events generated guarantees better performance.

Moreover, in order to further investigate the simulator performance and derive more insights regarding the required resources to run a computationally complex scenario, the following final evaluation is provided. A square area is partitioned into four quadrants, each one with a central drone relay. According to a uniform random distribution, a set of GUs is generated and symmetrically placed into the four regions with respect to the BS, which is placed in the center of the area. The number of considered GUs is then increased in accordance with the power of 2. Given this scenario, execution time and the maximum reserved memory are considered reference metrics and are reported in Fig. 24. As it can be deduced, both exponentially grow with an increasing number of GUs, as confirmed by the regression performed on the obtained data. Indeed, it is possible to predict the time and memory required for a specific simulation as

\[ ae^{b x} + ce^{dx} \]  

where \( x \) is the number of GUs, and \( a, b, c, d \) are the fitted coefficients provided in Table XI.

VIII. CONCLUSION

The IoD paradigm enables trailblazing applications, as the flexibility proposed by drones may significantly boost the effectiveness of existing activities, e.g., first response, monitoring, delivery, and surveillance. Moreover, drones are already becoming pervasive in several industrial sectors, such as smart agriculture, proactive maintenance, civil engineering, and many more.

As a matter of fact, the large-scale adoption should be evaluated after a prototyping phase that can be time-consuming and may require unfeasible costs. To tackle this problem, simulators are an essential tool to facilitate the testing phase and state the readiness for real-world exploitation. At the same time, simulators can be a learning tool for young professionals, engineering students, and researchers to improve their knowledge and explore scenarios never considered before.

In these regards, IoD-Sim is a thorough and user-welcoming tool that can be used to evaluate the many facets of IoD scenarios, including trajectory design, networking functionalities, mechanical characteristics, and data analytics. Nevertheless, IoD-Sim has been created as a modular tool that can be updated and upgraded as needed. A Visual Programming Editor for IoD-Sim has also been developed, relying on

![Fig. 23. Performance evaluation of the different simulated scenarios. (a) Number of events. (b) Simulation speedup time.](image)

![Fig. 24. Performance evaluation of the simulator with respect to the number of GUs. (a) Execution time. (b) Maximum reserved memory.](image)

TABLE XI
COEFFICIENTS OF EQUATION (7)

| Metric   | \( a \) | \( b \) | \( c \) | \( d \) |
|----------|--------|--------|--------|--------|
| Time     | 117.2  | 1.197  | 87.45  | 3.961  |
| Memory   | 64.35  | -0.05182 | 85.62  | 1.717  |
compilers’ theory and tools to dynamically update its contents based on the main simulator platform, ensuring that such project can be maintained with ease in the long term. Moreover, a predictable build environment is used to ease the installation due to its dependencies that require careful setup and knowledge about the underlying simulator, libraries, and compilers.

Even though IoD-Sim is a reliable solution, in the future more efforts will be focused on the improvement of the entire project, especially along the following research and development lines.

1) Extend the support to design scenarios using technologies such as MAVLink, satellite communications, and 5G-New Radio.
2) Speedup in Splash compilation with the use of parallel multiprocessing and optimized algorithms.
3) Develop interactive visual blocks to preview or design more accurate simulations in less time.
4) Improve the overall User Experience of the visual editor.
5) Allow the employment of multiprocessing systems and clusters.
6) Directly compares the performance and the features with other IoT simulation platforms.

Finally, the birth of a thriving and empowering community on open-source collaboration platforms will be crucial in assessing the future development efforts of this work.

REFERENCES

[1] M. Gharibi, R. Bounta, and S. L. Walander, “Internet of Drones,” IEEE Access, vol. 4, pp. 1148–1162, 2016.
[2] P. Boccadoro, D. Striccoli, and L. A. Grieco, “An extensive survey on the Internet of Drones,” Ad Hoc Netw., vol. 122, Nov. 2021, Art. no. 102600. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870521001335
[3] M. A. Hoque, M. Hossain, S. Noor, S. M. R. Islam, and R. Hasan, “IoTaaS: Drone-based Internet of Things as a service framework for smart cities,” IEEE Internet Things J., vol. 9, no. 14, pp. 12425–12439, Jul. 2022.
[4] S. Horsmanheimo et al., “5G communication QoS measurements for smart city UAV services,” in Proc. 16th Eur. Conf. Antennas Propagat. (EuCAP), 2022, pp. 1–5.
[5] S. H. Alsamhi, O. Ma, M. S. Ansari, and S. K. Gupta, “Collaboration of drone and Internet of public safety things in smart cities: An overview of qos and network performance optimization,” Drones, vol. 3, no. 1, p. 13, 2019.
[6] N. S. Labib, M. R. Brust, G. Danoy, and P. Bouvry, “The rise of Drones in Internet of Things: A survey on the evolution, prospects and challenges of unmanned aerial vehicles,” IEEE Access, vol. 9, pp. 115466–115487, 2021.
[7] S. H. Alsamhi, O. Ma, M. S. Ansari, and F. A. Almalki, “Survey on collaborative smart drones and Internet of Things for improving smartness of smart cities,” IEEE Access, vol. 7, pp. 128125–128152, 2019.
[8] S. H. Alsamhi et al., “Machine learning for smart environments in BSG networks: Connectivity and QoS,” Comput. Intell. Neurosci., vol. 2021, 2021, Art. no. 6805151.
[9] S. Liao, J. Wu, J. Li, A. K. Bashir, and W. Yang, “Securing collaborative environment monitoring in smart cities using blockchain enabled software-defined Internet of Drones,” IEEE Internet Things Mag., vol. 4, no. 1, pp. 12–18, Mar. 2021.
[10] Y. Zeng, I. Guvenc, R. Zhang, G. Geraci, and D. W. Matolak, UAV Communications for 5G and Beyond. Hoboken, NJ, USA: Wiley, 2020.
[11] S. Baidya, Z. Shaikh, and M. Levorato, “FlyNetSim: An open source Synchronized UAV network simulator based on Ns-3 and Ardupilot,” in Proc. 21st ACM Int. Conf. Model. Anal. Simul. Wireless Mobile Syst., New York, NY, USA, 2018, pp. 37–45. [Online]. Available: https://doi.org/10.1145/3242102.3242118
[12] N. R. Zerna, A. Trotta, G. Sanahuja, E. Natalizio, M. Di Felice, and L. Bononi, “CUSCUS: An integrated simulation architecture for distributed networked control systems,” in Proc. 14th IEEE Annu. Consum. Commun. Netw. Conf. (ICCCN), 2017, pp. 287–292.
[13] E. A. Marconato et al., “Avenus–A novel flying ad hoc network simulator with automatic code generation for unmanned aircraft system,” in Proc. 50th Hawaii Int. Conf. Syst. Sci., 2017, pp. 1–10.
[14] J. A. Millan-Romera, J. J. Acevedo, A. R. Castaño, H. Perez-Leon, C. Capitán, and A. Ollero, “A UTM simulator based on ROS and gazebo,” in Proc. Workshop Res. Educ. Develop. Unmanned Aerial Syst. (RED UAS), 2019, pp. 132–141.
[15] M. Tropea, P. Fazio, F. De Rango, and N. Cordeschi, “A new SANET simulator for managing drone networks and providing dynamic connectivity,” Electronics, vol. 9, no. 4, p. 543, 2020. [Online]. Available: https://www.mdpi.com/2079–9292/9/4/543
[16] S. Achariya, B. Amrutur, M. Bharatheesha, and Y. Simmhan, “CORNET 2.0: A co-simulation middleware for robot networks,” in Proc. COMSNETS, 2022, pp. 684–690.
[17] S. Park, W. G. La, W. Lee, and H. Kim, “Devising a distributed co-simulator for a multi-UAV network,” Sensors, vol. 20, no. 21, p. 6196, 2020.
[18] G. Grieco, R. Artuso, P. Boccadoro, G. Piro, and L. Grieco, “An open source and system-level simulator for the Internet of Drones,” in Proc. IEEE Int. Workshop Internet Mobile Things (IoMT) (IPIMRC), Istanbul, Turkey, Sep. 2019, pp. 1–6.
[19] M. Quigley et al., “ROS: An open-source robot operating system,” in ICRA Workshop Open Source Softw., vol. 3, Kobe, Japan, 2009, p. 5.
[20] N. P. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), vol. 3, 2004, pp. 2149–2154.
[21] G. F. Riley and T. R. Henderson, The NS-3 Network Simulator. Berlin, Germany: Springer, 2016, pp. 15–34. [Online]. Available: http://doi.org/10.1007/978–3-642–12331-3_2
[22] A. Varga, OMNeT++ Berlin, Germany: Springer, 2010, pp. 35–59. [Online]. Available: https://doi.org/10.1007/978–3-642–12331-3_3
[23] M. Galassi et al., GNU Scientific Library. Bristol, U.K.: Netw. Theory Ltd., 2002. [Online]. Available: https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf
[24] M. Yip, “RapidJSON—A fast JSON parser/generator for C++ with both SAX/DOM style API,” THL A29, 2015. [Online]. Available: https://github.com/miloyip/rapidjson
[25] C. Shepherd, “Theoretical design of primary and secondary cells. part 3. battery discharge equation,” Naval Res. Lab Washington DC, Washington, DC, USA, Rep. 5908, 1963. [Online]. Available: https://apps.dtic.mil/sti/pdfs/AD0405904.pdf
[26] O. Tremblay, L.-A. Dessaint, and A.-I. Dekkiche, “A generic battery model for the dynamic simulation of hybrid electric vehicles,” in Proc. IEEE Veh. Power Propul. Conf., 2007, pp. 284–289.
[27] Y. Sun, D. Xu, D. W. Ng, L. Dai, and R. Schober, “Optimal 3D-trajectory design and resource allocation for solar-powered UAV communication systems,” IEEE Trans. Commun., vol. 67, no. 6, pp. 4281–4298, Jun. 2019.
[28] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, “A brief introduction to OpenCV,” in Proc. 35th Int. Conf. Med. MIPRO, 2012, pp. 1725–1730.
[29] S. S. Kumanar, Practical LXC and LXD: Linux Containers for Virtualization and Orchestration. New York, NY, USA: Springer, 2017.

Giovanni Grieco (Graduate Student Member, IEEE) received the Dr.Eng. degree (with Hons.) in telecommunications engineering from the Politecnico di Bari, Bari, Italy, in October 2021. He is currently pursuing the Ph.D. degree with the Department of Electrical and Information Engineering, Politecnico di Bari, and the Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Parma, Italy.

His research interests include Internet of Drones (IoD), cybersecurity, and future networking architectures. He is the Principal Maintainer of IoD_Sim.
Giovanni Iacovelli (Graduate Student Member, IEEE) received the Dr.Eng. degree (with Hons.) in information engineering from the Politecnico di Bari, Bari, Italy, in July 2019. He is currently pursuing the Ph.D. degree with the Department of Electrical and Information Engineering, Politecnico di Bari, and the Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Parma, Italy. His research interests include Internet of Drones, machine learning, optimization, and telecommunications.

Pietro Boccadoro (Member, IEEE) received the Dr.Eng. degree (with Hons.) in electronic engineering and the Ph.D. degree from the Politecnico di Bari, Bari, Italy, in July 2015 and 2021, respectively. From November 2015 to October 2020, he collaborated as a Researcher with Politecnico di Bari. He is currently with the Department of Electrical and Information Engineering, Politecnico di Bari, Bari, Italy, and Nextome srl, Conversano, Italy. His research interests include Internet of Drones, robotic-aided IoT, and future Internet architectures.

Luigi Alfredo Grieco (Senior Member, IEEE) received the Dr. Eng. degree (with Hons.) in electronic engineering from the “Politecnico di Bari,” Bari, Italy, in October 1999, and the Ph.D. degree in information engineering from the “Università di Lecce,” Lecce, Italy, in December 2003. From January 2005 to October 2014, he was an Assistant Professor with the “DEI—Politecnico di Bari.” He was a Visiting Researcher with INRIA, Sophia Antipolis, France, working on the topic of Internet Measurements from March 2009 to June 2009, and LAAS-CNRS, Toulouse, France, working on Information Centric Networking Design of M2M Systems from October 2013 to November 2013, and an Associate Professor of Telecommunications with the Politecnico di Bari (DEI) from November 2014 to December 2018, where he has been a Full Professor since December 2018. He is also with the Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Parma, Italy. He authored around 200 scientific papers published in venues of great renown that gained more than 8000 citations. His current research interests include Industrial Internet of Things, information centric networking, and nano-communications.

He is the Founding Editor-in-Chief (EiC) of the Internet Technology Letters Journal (Wiley) and served as the EiC of the Transactions on Emerging Telecommunications Technologies (Wiley) from mid-2016 to 2019. He also serves as an Associate Editor for the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY (for which he has been awarded as top associate editor in 2012 and 2017). He has been constantly involved as a technical program committee member of many prestigious conferences. Within the Internet Engineering Task Force (Internet Research Task Force), he contributed (as author of RFC 7554) new standard protocols for industrial IoT applications (new standard architectures for tomorrow ICN-IoT systems). Since January 2019, he has been a Founding Member and the Subarea-Chair of the IEEE SIG on Intelligent Internet Edge. In 2020, he has been nominated as a Scientific Coordinator of the IoT4.0 Lab.