Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

RESEARCH ARTICLE

Simulating 6TiSCH Networks

Esteban Municio[1] | Glenn Daneels[1] | Mališa Vučinić[2] | Steven Latré[1] | Jeroen Famaey[1] | Yasuyuki Tanaka[5] | Keoma Brun[5] | Kazushi Muraoka[3] | Xavier Vilajosana[4] | Thomas Watteyne[5]

[1]IDLab, University of Antwerp - imec, Antwerp, Belgium
[2]Faculty of Electrical Engineering, University of Montenegro, Podgorica, Montenegro
[3]NEC Corporation, Kawasaki, Japan
[4]WiNe Lab, UOC, Barcelona, Spain
[5]EVA Team, Inria, Paris, France

Correspondence
Esteban Municio, Middelheimlaan 1, 2020 Antwerp, Belgium. Email: esteban.municio@uantwerpen.be

Abstract

6TiSCH is a working group at the IETF which is standardizing how to combine IEEE802.15.4 Time-Slotted Channel Hopping (TSCH) with IPv6. The result is a solution which offers both industrial performance and seamless integration into the Internet, and is therefore seen as a key technology for the Industrial Internet of Things. This article presents the 6TiSCH Simulator, created as part of the standardization activity, and which has been used extensively by the working group. The goal of the simulator is to benchmark 6TiSCH against realistic scenarios, something which is hard to do using formal models or real-world deployments. This article discusses the overall architecture of the simulator, details the different models it uses (i.e. energy, propagation), compares it to other simulation/emulation platforms, and presents 5 published examples of how the 6TiSCH Simulator has been used.

1 | INTRODUCTION

Reliable, deterministic and time-sensitive networking is critical for the Industrial Internet of Things (IIoT). The Time-Slotted Channel Hopping (TSCH) mode of IEEE802.15.4 is at the core of virtually all low-power wireless standards for the IIoT, including WirelessHART, ISA100.11a and IEEE802.15.4-2015 (1). Tens of thousands of TSCH-based low-power wireless networks are deployed today[1]. TSCH is a technology which has demonstrated to provide wire-like reliability and over a decade of battery lifetime (2). The IETF is standardizing a new TSCH-based technology, which combines the performance of TSCH with the ease-of-use of IPv6 (one of the main differentiators with previous standards). This work is being done at the IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) working group, which has just completed its base set of specifications (3, 4, 5, 6, 7).

Besides the purely standardization efforts, there are also many research and implementation works focusing on 6TiSCH. It is now supported by all the main open-source low-power wireless implementations: OpenWSN, Contiki-NG, RIOT and TinyOS (8). Improvements to 6TiSCH are being proposed by the academic community (9, 10, 11, 12). But while this activity is a clear indication of the activity of the 6TiSCH community, what is needed now to speed up market adoption is slightly different.

Before a company adopts (and invests in) 6TiSCH, it needs to know whether the technology works for its applications. This means answering very down-to-earth questions about the performance of the network: *What is the latency distribution 6TiSCH would offer in this particular industrial deployment? If I use 6TiSCH in my smart agriculture application, how often will I have to replace batteries? Would 6TiSCH offer over 99.999% end-to-end reliability in this particular smart parking deployment?* What is needed to answer these questions is a way of **benchmarking** the solution standardized today. That is, we need a tool which can turn a particular deployment and traffic pattern in key networking performance indicators such as per-node battery lifetime, end-to-end reliability, and latency distribution.

[1] One vendor alone – Emerson – reports over 35,000 networks deployed, which translates to over 10-Billion operating hours (https://www.emerson.com/en-us/expertise/automation/industrial-internet-things/pervasive-sensing-solutions/wireless-technology)

Of course, it would be possible to answer these questions through a comprehensive real-world experimentation campaign. And while extensive experimentation is happening (and will be happening more and more), we argue that it lacks the flexibility of doing quick "what-if" analysis. For that, simulation offers several advantages. First, it allows us to quickly estimate the performance for a given scenario and a set of parameters. Second, it allows us to tune the parameters in a flexible manner until the desired performance is reached and map these values to the real-world deployment. That being said, simulation is necessarily an imperfect model of physical reality, in particular the radio propagation/connectivity between nodes. And while simulation results must be taken with a grain of salt, they are a necessary first step before any deployment (discussed further in Section 3).

This article presents the 6TiSCH Simulator: a tool for simulating 6TiSCH networks and estimating their performance in a flexible manner. Other network simulators exist – and are surveyed in Section 3 – but they fail to capture the specifics of 6TiSCH networks in a manner that can be useful for a real-world deployment. The reason for creating a new simulator is that we wanted to give priority to simplicity over the extreme extensibility and generality existing in other platforms. We consider that the 6TiSCH simulator enables new users to deploy quick simulations and have direct control over the 6TiSCH parameters without the learning curve required for other platforms.

The simulator mimics exactly the behavior of the full 6TiSCH solution, including network formation, routing and scheduling. It analyzes different 6TiSCH sub-layers for a wide variety of conditions, while ensuring accuracy regarding the standard-compliant hardware implementations (13, 14). It allows end-users and potential adopters to assess whether 6TiSCH would "work" for their applications. It allows researchers to (easily) evaluate the performance of some custom addition to the otherwise fully standards-compliant implementation. The simulator allows for fully automated execution on multiple cores. A graphical interface allows the user to drive the simulation and visualize the behavior of 6TiSCH. In a number of published works on 6TiSCH, the simulator proves itself as a timely and usefully contribution (9, 10, 11, 12, 15, 16, 17, 18, 19, 20).

The main goal of this article is to provide an overview of the 6TiSCH Simulator, and discuss its design drivers and approach. The contribution is two-fold. First, we provide the 6TiSCH Simulator as open-source code[2]. Second, we validate the simulator through comparative tests and we provide use case examples that discuss relevant open research questions regarding 6TiSCH.

The remainder of this article is organized as follows. Section 2 introduces the 6TiSCH protocol stack. Section 3 discusses other network simulators/emulators, and why a new 6TiSCH-specific simulator was developed. Section 4 discusses the overall architecture of the simulator and details the different models it uses (including energy and propagation). Section 5 presents the results that validate the tool and Section 6 shows its performance. Section 7 presents 5 published examples of how the 6TiSCH Simulator has been used. Finally, Section 8 concludes this article.

2 | A 6TISCH PRIMER

6TiSCH targets building-sized networks and typical IIoT applications, such as monitoring and control. It builds upon the IEEE802.15.4 physical-layer standards, typically used in the license-free 2.4 GHz ISM band with a 250 kbps bit rate. IEEE802.15.4 splits that band into 16 orthogonal frequencies. The maximum PDU of IEEE802.15.4 is 127 bytes.

IEEE802.15.4 defines different modes on how the radio channel is accessed. The mode used by 6TiSCH is TSCH. The main principle of TSCH is to combine Time Division Multiple Access (TDMA) for collision-free access with channel hopping for robustness against multi-path fading and external interference. TDMA slots are wide enough (typically 10 ms) to accommodate the transmission of the largest frame and a short acknowledgement. A group of slots that repeat in time is called a *slotframe*.

Whether a node transmits, receives or sleeps during a slot is defined by the *schedule*. The schedule can be visualized as an $M \times N$ matrix, where M is the number of available physical channels, and N is the number of slots in the slotframe. Each element in the schedule matrix is called a *cell*. A cell is simply an abstraction that carries information needed for a node to handle the slot. This information is the type of cell (transmit, receive, sleep) and the address of the corresponding radio neighbor. A group of cells to a particular neighbor constitutes an (IP) link. A cell is uniquely identified by its *channel offset* and its *slot offset*. A cell's slot offset maps to the time the slot is executed and it is identified by a unique Absolute Slot Number (ASN). A cell's channel offset maps to a different frequency for each slotframe iteration, resulting in channel hopping.

How the schedule is built and communicated to the nodes is not defined by IEEE802.15.4 and is where 6TiSCH fits in. 6TiSCH builds the schedule in a completely distributed fashion. A cell in a 6TiSCH network can either be "hard" or "soft". Soft cells are installed in a dynamic fashion by the 6top sub-layer, hard cells are static.

[2] As an online addition to this article, the code of the simulator, published under an open-source BSD license, can be found at https://bitbucket.org/6tisch/simulator

RFC8180 (4) defines the usage of a single hard cell used for network bootstrap, referred to as the "minimal" cell. A node can use the minimal cell both for transmission and reception. Since all nodes in the network have this cell in their schedule, collisions are possible and handled by the exponential back-off algorithm defined in IEEE802.15.4. The nodes obtain the information about the minimal cell dynamically, when they receive a special frame called Enhanced Beacon (EB), used for initial synchronization with the network. The minimal cell carries all the broadcast traffic in the network as well as the unicast traffic required to build the network.

Soft cells are installed by the 6top Protocol (6P) (5). When a node A wants to add or delete N cells in its schedule with neighbor B, it triggers a 6P transaction. In a 2-step 6P transaction, A selects a candidate list of cells and sends it to B. B then selects N cells from the list and communicates it to A by sending a response. When B receives an acknowledgement of its transmission to A, it knows that A has received the response and adds the selected cells in its schedule. In a 3-step transaction, it is B which provides a list of candidate cells and A which selects the exact cells to be used. Apart from adding and deleting cells, 6P also allows the existing cells to be relocated to a different position in the schedule matrix.

6P provides a means for the cells to be installed in a dynamic fashion. The algorithm by which a node decides that it needs to add/delete cells is called a Scheduling Function (SF). The SF enables the deterministic nature of 6TiSCH: it decides when to add/delete cells and to which neighbor, how many cells to add/delete and which exact cells to use. This allows the optimization of different criteria, such as latency or energy consumption. There are many SF proposals in 6TiSCH. The Minimal Scheduling Function (MSF) (7), for example, monitors link utilization and adapts the number of cells to keep utilization within pre-defined bounds. Effectively, this allows MSF to dynamically adapt to traffic. The risk of distributed scheduling is that two disjoint pairs of nodes in the network decide to use the same cell. This results in a *schedule collision* and degrades performance as the transmissions during the corresponding slots may interfere at the receiver(s). To tackle this problem, MSF monitors the Packet Delivery Ratio (PDR) of each cell individually, and relocates the cell if it observes that the PDR is significantly lower than the average PDR of other cells with the same neighbor. Research around SFs has been very active (9, 11, 16, 12, 21) and we expect other proposals to be standardized in the future.

There is another research issue with regards to figuring out the policy that yields minimal cell utilization. The minimal cell carries routing information exchanged by the Routing Protocol for Low-Power and Lossy Networks (RPL), EBs transmitted by all the nodes in the network in order to extend the network range, and unicast traffic needed for the network to bootstrap. This unicast traffic includes the secure join protocol standardized in 6TiSCH (6), whose goal is to provide the new node with security keys once it has been authenticated. Moreover, it includes the first 6P transaction to a given neighbor. Because of this, the way EBs and RPL messages are sent significantly influences the time it takes for the network to form (17).

The remainder of the 6TiSCH stack (depicted in Figure 1) is standardized in other working groups of the Internet Engineering Task Force (IETF). 6TiSCH uses the RPL routing protocol. RPL optimizes multipoint-to-point routes, i.e. from the nodes towards the root of the network, and point-to-multipoint routes for the root to be able to reach each node. The multipoint-to-point routes are created by building a Destination Oriented Directed Acyclic Graph (DODAG) – each node selects a preferred parent based on the rank information received in the DODAG Information Object (DIO) messages. Once the DODAG is built, each node sends Destination Advertisement Object (DAO) messages to the root, which is then able to construct the source route and append it to each packet addressed to nodes in the network. A node can also switch parents. This is coordinated by RPL but has a direct impact on necessary TSCH link resources. The parent switch is handled by MSF in order to quickly allocate additional cells to the new parent and remove remnant cells with the old parent. At the transport and network layers, 6TiSCH uses User Datagram Protocol (UDP) and IPv6, which are compressed by 6LoWPAN compression mechanisms. 6TiSCH uses Constrained Application Protocol (CoAP) at the application layer, secured by an object security mechanism called Object Security for Constrained RESTful Environments (OSCORE).

3 | RELATED SIMULATION PLATFORMS

Network simulation is a common approach used in the design, implementation and performance evaluation of different algorithms and protocols when there is a need for assessing the behavior of a network given a set of models and constraints. In the low-power wireless case, discrete event-driven network simulators are widely used since: 1) they can be used not only as a flexible and inexpensive tool for testing the network without the need of having to deploy an actual physical network, and 2) they can be more accurate and realistic than mathematical models where usually simplifications and abstractions are assumed.

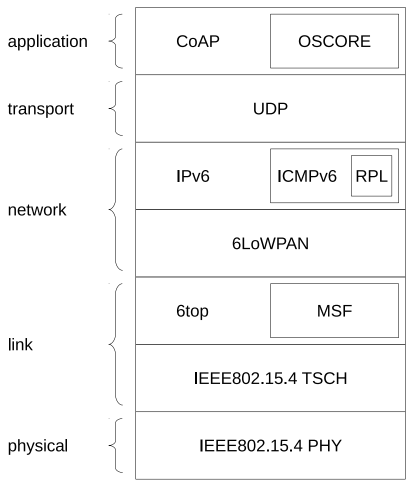


FIGURE 1 The IETF 6TiSCH Protocol Stack.

Although real-world deployments are the most reliable approach for evaluating network performance, it involves important practical difficulties (logistically and economically), not only for deployment and testing but also for modifying and debugging the network. This is especially relevant in large-scale deployments. Unlike real-world deployments, network simulators allows us to flexibly switch and exchange the different protocols and layers in the stack and quickly evaluate network performance for a number of configurations and scenarios.

Mathematical models and problem formulations, although very relevant for describing and predicting behaviors, sometimes lack the accuracy present not only in the standards and RFCs but also aspects inherent to real-world deployments. In this sense, network simulators can provide the required standard-compliant accuracy in a flexible manner. Additionally, complex models become impossible to solve in acceptable time when the network size increases. This makes network simulators a crucial research tool for bridging the mathematical models with real-world deployments.

The open-source network simulators NS-2 (22) and JSim (23) have been traditionally used for simulating low-power wireless networks. Currently, *NS-3* (24), *OMNet++* (25) and *TOSSIM* (26) are arguably the most widely used. However, none of these simulators fulfill the goals for which the 6TiSCH Simulator was designed: compliance with the standard, scalability and simplicity.

- *NS-3*, an advanced and more modular version of NS-2, is probably the most widely used network simulator nowadays. It has a number of modules for different technologies and protocols and the support of a big community of users. Although NS-3 is a very powerful simulator, its complexity and generic purpose involve a significant learning curve and non-significant programming time for a non-expert user. No 6TiSCH implementation currently exists.
- *OMNet++* is also a very extensively used simulator for simulating low-power wireless networks. Its INET framework implements most of the more common network protocols. A very realistic PHY layer is available in Castalia (27), an OMNet++-based simulator which implements accurate wireless channels and radio access models. As in NS-3, no 6TiSCH implementation is available.
- *TOSSIM* is a simulator for TinyOS (28). Although TinyOS has been one of the first low-power wireless implementations, support has officially ended in 2013.

Other emulation options are available, including *Cooja* (29) (the Contiki emulator)[3] and *OpenSim* (13) (the OpenWSN emulator). These platforms are very powerful on emulating the behavior of a 6TiSCH network without needing the actual hardware, since they run the same binary as goes on a low-power wireless device. The emulation approach has two main drawbacks. First, unlike discrete event-driven simulators, their scalability is rather poor, since the real-time requirements of the instances limit the size of the network up to few tens of nodes. Second, due to its bit-level accuracy, new implementations in routing, scheduling

[3]A less detailed, but fully-simulated version is also available, but is still significantly time consuming.

Simulator	Learning Curve	Scalability	6TiSCH implementation	Standard-Compliant
ns-3	High	Medium	None	N/A
OMNet++	High	Medium	None	N/A
TOSSIM	Medium	High	None	N/A
Cooja (emulator)	High	Low	Yes (Partial)	Partially
OpenSim (emulator)	High	Low	Yes	Yes (byte-accurate)
6TiSCH Simulator	Low	High	Yes	Yes (behavioral)

TABLE 1 Comparison between the 6TiSCH Simulator and the different network simulator alternatives.

functions or traffic management require significantly more effort than in discrete event-driven simulators where some functions can be abstracted.

A comparison between the different existing alternatives is shown in Table 1 . It evidences why the 6TiSCH Simulator has been developed. First, it has a 6TiSCH implementation that follows the requirements specified in the 6TiSCH RFCs and drafts (e.g. Cooja does not implement MSF). Second, it scales up easily up to several hundreds of nodes. Finally, it has low complexity, enabling to quickly implement and test different scheduling functions, RPL objective functions and traffic management strategies. These points are detailed in Section 4. In that sense, the 6TiSCH simulator is not really comparable to NS-3 or OMNet++, since they are generic purpose simulators which contain a high number of libraries supporting different protocols and models. The 6TiSCH simulator is a highly specialized protocol specific tool for fast prototyping.

4 | THE 6TISCH SIMULATOR

The 6TiSCH Simulator is a discrete-event simulator written in Python. Its design minimizes typical simulation drawbacks by careful abstractions specific to 6TiSCH. It makes no attempt at simulating physical behavior that can only be accurately studied with real hardware such as: synchronization issues due to imperfect crystals, bit-specific transmission errors, hardware-dependent processing delays. Instead, it focuses on simulating the behavior that is observed in the network *from* the MAC layer. We achieve this with two abstractions. First, we quantize time in TSCH slots: an event can only take place at the slot boundary. Second, we abstract the protocol messages to only carry semantically-relevant parameters: exchanged messages are not byte-accurate.

We build upon these two abstractions to provide a simulator that can accurately monitor:

- the behavior of the SF in response to generated traffic (in frames/second).
- the behavior of the routing protocol in response to topological changes.
- the behavior of 6P in response to MAC-layer drops.
- the behavior of the application in response to scheduling, routing, and network stack configuration.

Different simulator instances can be scheduled to execute in parallel on available processor cores (tested up to 56 cores). The simulator is implemented in approximately 8000 lines of Python code, spanning 6 core files and a graphical interface[4]. The internal architecture of the simulator is shown in Figure 2 . The main component is *Mote*, where the major part of the 6TiSCH stack is implemented. It is configured by *SimSettings*, which contains the input parameters introduced by the user. *Mote* also generates metrics for *SimStats* and *SimGUI* and programs events that are scheduled and processed by *SimEngine*. Some of these events are the TXs and RXs, which are evaluated by *Propagation* depending on the existing *Topology*.

The graphical user interface (Figure 3) shows the aggregated schedule of all nodes. The user can control the simulation run, and see real-time statistics on different cells, links and nodes. The simulator can also be run in a “headless” mode (i.e. no graphical interface), allowing for fully automated runs. Finally, several generic post-processing and plot scripts are also provided in the simulator.

[4]In contrast, other network simulators, such as NS-3 in the current 3.28 version, have up to 1483072 lines when considering ".*cc*", ".*h*" and ".*py*" files.

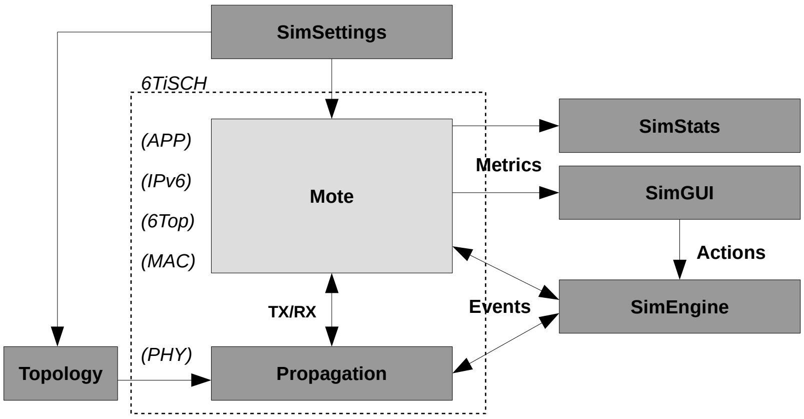
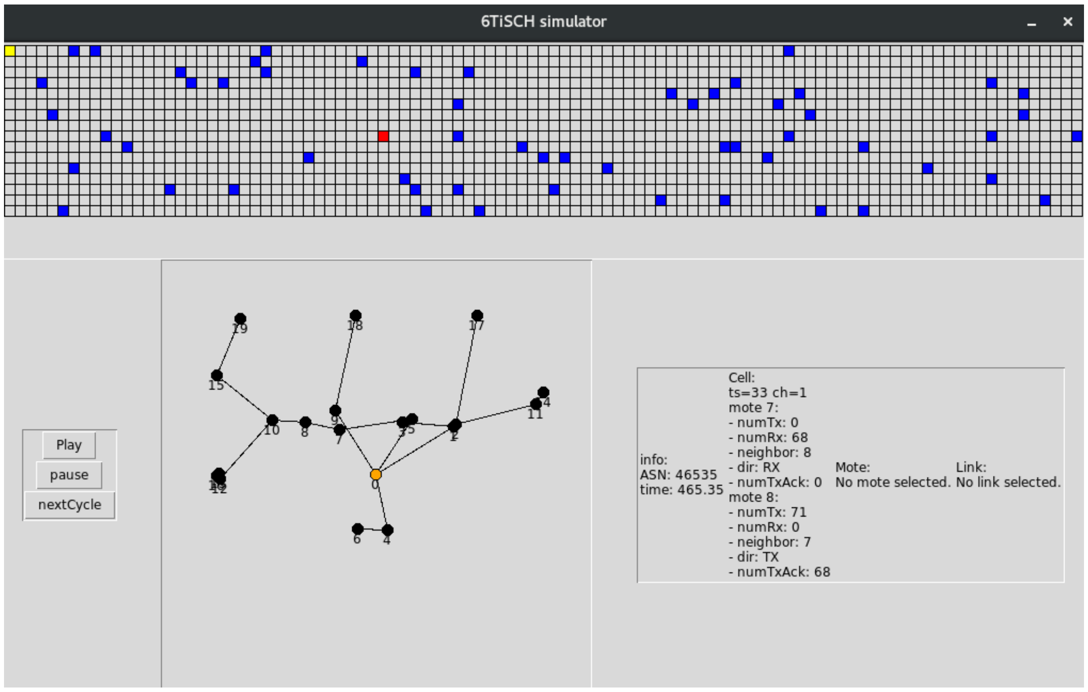


FIGURE 2 Internal architecture of the 6TiSCH Simulator.



```
info:
ASN: 46535
time: 465.35

Cell:
ts=33 ch=1
mote 7:
- numTx: 0
- numRx: 68
- neighbor: 8
- dir: RX
- numTxAck: 0
mote 8:
- numTx: 71
- numRx: 0
- neighbor: 7
- dir: TX
- numTxAck: 68

Mote:
No mote selected.

Link:
No link selected.
```

FIGURE 3 Screenshot of the 6TiSCH Simulator user interface.

4.1 | SimEngine

The event-driven core of the simulator is implemented in SimEngine. Events are generated by the *Mote* every time a new task needs to be scheduled in the future, such as increasing the ASN in the nodes, propagating a packet or firing a timeout. These events are uniquely identified by a tag formed by the node ID and a label, the event's UUID. Since more than one event can happen at the same time instant, events are registered with a specific priority and processed accordingly.

The set of events to be executed in the future, the Future Event Set (FES), is implemented using a Python *list*, in which events are added with the *insert* method and removed with the *pop intermediate* method. The time complexity for *insert* is $O(n)$ on average, where n is the number elements in the *list*. The time complexity for *pop intermediate* is $O(k)$ where k is the index of the event to be removed, which on average is also $O(n)$.

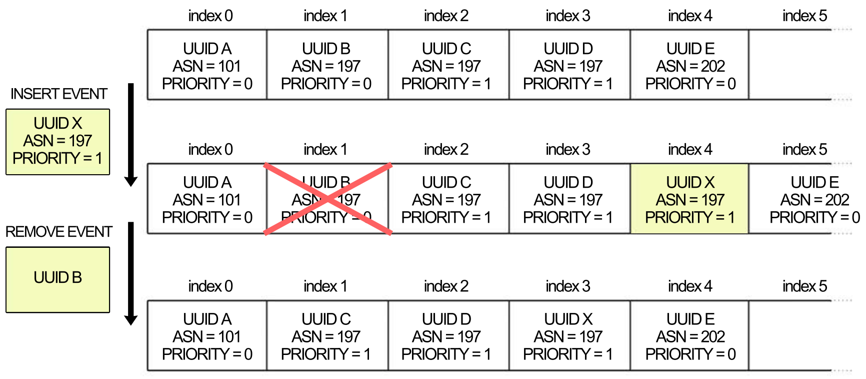


FIGURE 4 Example of how the FES is managed in the 6TiSCH Simulator.

Each instance of the 6TiSCH simulator is run sequentially in a single thread. This means that there are no concurrency risks when two events are scheduled at the same time. The scheduler always executes first the events with lower ASN regardless its priority. For the same ASN, events with the highest priority will be inserted before other events with lower priority. In case the ASN and the priority are the same, the new event will be inserted after the last event with the same ASN and priority (i.e. in order of arrival). The simulator engine also prevents events to be inserted in the current ASN in order to remove inconsistencies when popping the next event and avoid event loops. Figure 4 shows an example of how events are inserted and removed. For example, when adding a new event with UUID X at ASN 197 and priority 1, the scheduler will iterate through the *list* until it finds an event with a higher ASN and lower priority than the new event.

For canceling an event, e.g. UUID B at ASN 197 in Figure 4 , the list will be iterated until the event is found. Afterwards, the event is deleted and the remaining events in the list are shifted to the left. As with the insert method, the simulator engine does not allow to remove events in the current ASN. This avoids nondeterministic behavior when trying to execute an event that is about to be deleted or vice versa.

4.2 | Topology

In order to rule out the impact of a specific topology on network performance, the simulator generates a new random topology for each simulation run. The user specifies the size of the deployment area. Nodes are placed at random locations until each node has a stable physical link to a pre-configured number of neighbors. Ensuring a minimum number of neighbors per node is a common real-world deployment practice (30). Each link is assigned a PDR, as described in Section 4.3. Furthermore, additional topologies such as start topology and linear topology are easily configurable in the *Topology* file.

4.3 | Propagation Model

The default propagation model implemented in the 6TiSCH Simulator is based on the Pister-Hack model (31). This model is used to obtain the initial RSSI value of each pair of nodes in the network. This value is calculated by subtracting a uniform variance of 40 dB from the Friis model equation output. The model was independently verified in an experimental setting (32). RSSI values are then converted to Packet Delivery Ratio (PDR) values by a conversion table that is based on real-world deployments[5]. It accurately reflects the relationship between the RSSI and PDR in large indoors industrial scenarios at the 2.4 GHz band. The Pister-Hack model and the RSSI-PDR conversion table are depicted in Figure 5 and Table 2 , respectively.

The PDR values are used to calculate if a transmission is successful or not, by flipping a biased coin. In order to model the interference, the RSSI from the interfering neighbors at any given transmission is added to the ground noise in order to calculate the signal-to-interference-plus-noise ratio (SINR), which is converted eventually in the actual perceived PDR.

[5]The connectivity traces were obtained from the http://wsn.berkeley.edu/connectivity/ project at the University of California, Berkeley, lead by Thomas Watteyne and Prof. Kris Pister. The dataset used is "soda", created by Jorge Ortiz and Prof. David Culler.

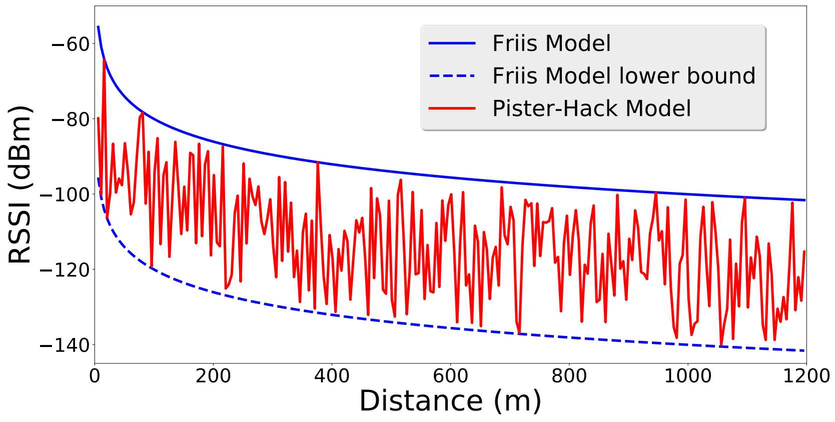


FIGURE 5 RSSI values generated using the Pister-Hack model.

RSSI	PDR
-97 dBm	0.0000
-96 dBm	0.1494
-95 dBm	0.2340
-94 dBm	0.4071
-93 dBm	0.6359
-92 dBm	0.6866
-91 dBm	0.7476
-90 dBm	0.8603
-89 dBm	0.8702
-88 dBm	0.9324
-87 dBm	0.9427
-86 dBm	0.9562
-85 dBm	0.9611
-84 dBm	0.9739
-83 dBm	0.9745
-82 dBm	0.9844
-81 dBm	0.9854
-80 dBm	0.9903
-79 dBm	1.0000

TABLE 2 RSSI-PDR translation.

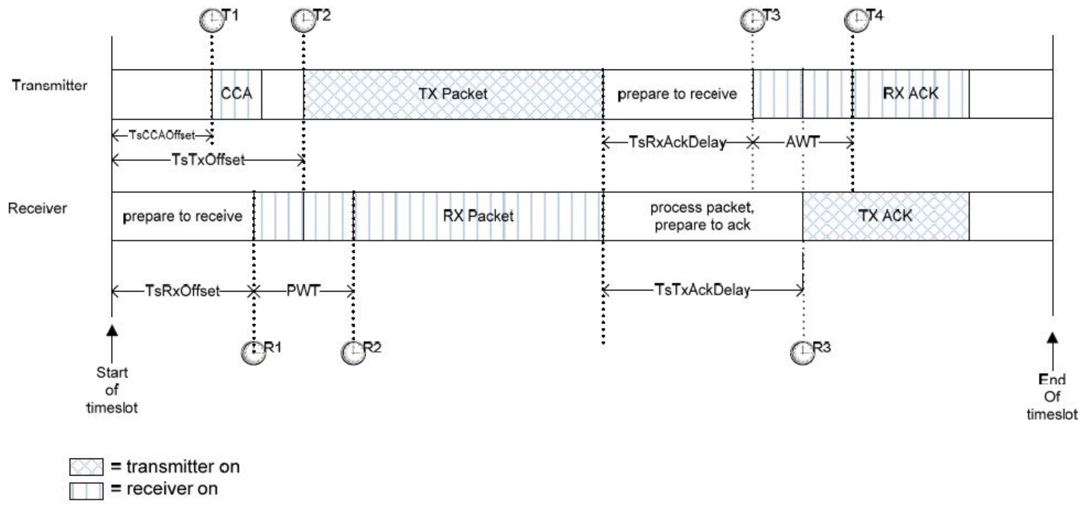


FIGURE 5 Timeslot timing and sequence of actions used to derive the energy consumption of a node.

4.4 | Energy Consumption Model

The simulator implements the energy consumption published by Vilajosana *et al.* (33). The model takes a component-based approach. It defines the energy consumption of the different types of slots and combines them according to the schedule configuration. Figure 5 presents the sequence of actions that occur during a TSCH timeslot. The energy model uses that sequence of actions to model the energy consumed by a node.

For example, an active slot transmitting a data packet and receiving an acknowledgement frame (referred to as *TxDataRxAck*) turns the radio on twice: once for sending the data frame (for a duration of *TX Packet*) and once for receiving the acknowledgement frame (for a duration of *RX ACK*). The node's CPU is turned on during these phases, while the node stays in a deep sleep mode in the others. The model characterizes the consumption of these sub-periods within the slot and takes into account the current draw of the radio when on, the number of bytes transmitted, the data rate, and the energy consumption of the CPU in its different modes and transitions.

The model implemented within the simulator considers different types of slots as per Table 3 . After the execution of the slot, the simulator aggregates the consumed energy. The default current draw for each operation matches that of the OpenMote platform (34) but those values can easily be adapted to other platforms.

TABLE 3 Possible slot types as per the energy consumption model implemented within the 6TiSCH Simulator.

State	Description
Idle	The node idle listens. This is a RX state where nothing is received. Hence it only listens for the duration of the guard time.
Sleep	The node Deep Sleeps. The slot is off so no CPU nor radio activity due to communication.
TxDataRxAck	The node transmits a packet and receives an ACK for it.
TxData	The node sends a broadcast packet not requiring ACK.
RxDataTxAck	The node receives a packet and responds with an ACK.
RxData	The node receives a frame that does not require to be acknowledged.

4.5 | Model of a 6TiSCH "Mote"

A "mote" is an abstraction of a 6TiSCH network node and implements its different sublayers in the *Mote* file. At boot time, the simulation engine instantiates the required number of *Mote* objects and assigns the role of root to one of them. The root triggers the network formation by adding periodically EBs and RPL DIOs to its transmission queue. Simultaneously, the remaining nodes start to listen in a randomly picked channel. At every slot, the *Propagation* model evaluates which nodes have scheduled transmissions at this ASN and which nodes are listening. For every packet, it will determine the outcome of the transmission regarding the signal strength and the interference level provoked by concurrent transmissions in the same radio vicinity. Nodes schedule TX/RX events according to their schedule. Initially the root only wakes up in the ASNs that correspond to the minimal cell, where it sends or receives a packet accordingly. The nodes that are not synchronized yet, schedule RX events for waking up at every ASN.

When a node eventually receives an EB, it synchronizes (enables its TSCH MAC layer (1)), and starts the joining process according to (7). After the node joins the network through its Join Proxy (6), is able to decipher DIOs messages that allows him to select a preferred parent and obtain a rank. Once a node has selected its preferred parent, it triggers its first dedicated cell allocation (4, 5). Nodes are only allowed to send data packets only when they have dedicated cells.

The booting sequence is triggered with the help of several housekeeping callback functions that are periodically scheduled to perform specific actions at every sublayer (TSCH, RPL, MSF, etc.). Also, different events are scheduled whenever a timeout is required (6P timeout, Join timeout, MSF timeout, etc.). These events are registered in the *SimEngine* and will trigger the callback functions when the timers expire.

Every sublayer in *Mote* is highly configurable through the *SimSettings* component. For example, it is possible to configure the TSCH layer frame size, timeslot duration, beacon period, etc. The 6top sublayer and the scheduling function MSF can be configured by changing the parameters originated from the draft (i.e. *MAX_NUMCELLS*, *LIM_NUMCELLSUSED_LOW*, *HOUSEKEEPINGCOLLISION_PERIOD*, etc.). The RPL layer is implemented in the non-storing mode and also supports different configurations (DIO period, DAO period, etc.). Finally, application traffic can be constant or variable, and can be injected at any time during the simulation. The variable traffic can be modeled according to different probability distributions and configurable traffic bursts can be scheduled.

4.6 | Metrics

During the execution of the simulation, event handlers trigger updates for the different metrics. Over 50 different metrics are currently implemented, however the addition of new metrics is supported and is easily implementable. Metrics can be defined as *per cycle* to monitor the evolution of a specific metric per TSCH slotframe cycle or as *absolute* in order to obtain the resulting total value after the simulation. We refer to slotframe cycle as the number of timeslots in a slotframe (i.e. 101 slots by default). Absolute metrics (such as charge consumed) can be obtained per node or aggregated over all the nodes in the network. RSSI values for every physical link are also logged. Table 4 details some of the most important metrics of the simulator.

The simulator by default logs the metrics of each simulation independently in a log file. However, simulation runs use different files and folders regarding the CPU ID and the network size. The logging directory structure can also be easily changed by the user in *SimSettings*. The simulator also provides with a set of helper scripts that allow the user to post-process and plot any desired metric in a fully automated manner.

TABLE 4 Some of the available metrics in the 6TiSCH Simulator

State	Type	**Description**
Average latency	Per cycle	Average latency of packets arriving at the root (in ASNs)
Charge consumed	Absolute	Charge consumed by all nodes during the simulation
Charge consumed at every node	Absolute	Total charge consumed by a node during the simulation
App packets generated/received	Per cycle	Number of data packets generated and received at every cycle
Number of TX/RX	Per cycle	Number of MAC frames sent and received at every cycle
Number of drops	Per cycle	Drops are classified by its cause: QueueFull, MaxRetries and NoRoute
Number of used cells	Per cycle	TX/RX/SHARED cells used by all the nodes
Colliding cells	Per cycle	Dedicated cells used in more than one link
Parent changes	Per cycle	Number of parent changes per cycle

5 | VALIDATION

In order to validate the 6TiSCH Simulator we have carried out several experiments in both the simulator and OpenWSN, which is the most up-to-date 6TiSCH stack implementation available.

For the first set of experiments we compare the booting procedure while measuring the network time (i.e. number of ASNs): the synchronization time (i.e. the time in which a node receives its first valid EB), the joining time (i.e. the time in which a node securely joins the network) and the *first cell* time (i.e. the time in which a node has its first dedicated cell with its preferred parent). We perform 2000 iterations for both platforms in a 2-node network (root node and leaf node), using the parameters defined in the current RFCs and drafts (4, 5, 6, 7). In this case, we have used OpenSim, which is the Python emulator platform of OpenWSN. OpenSim runs exactly the same 6TiSCH implementation that is used in the real hardware, but objectifies the files and runs them in an abstracted Python board that virtualizes the hardware functions.

Figure 6 shows that the CDF of the three different metrics are equivalent (synchronization time, joining time and first cell time). The values are obtained in ASN, but converted to seconds for a timeslot duration of 15ms. In the three cases, the maximum error between OpenWSN and the 6TiSCH Simulator is 3.89%, 1.73% and 1.35% for sync time, join time and first cell time respectively when averaging all the samples. This proves that the simulator, although an abstraction, behaves as an accurate tool when implementing the current 6TiSCH RFCs and drafts.

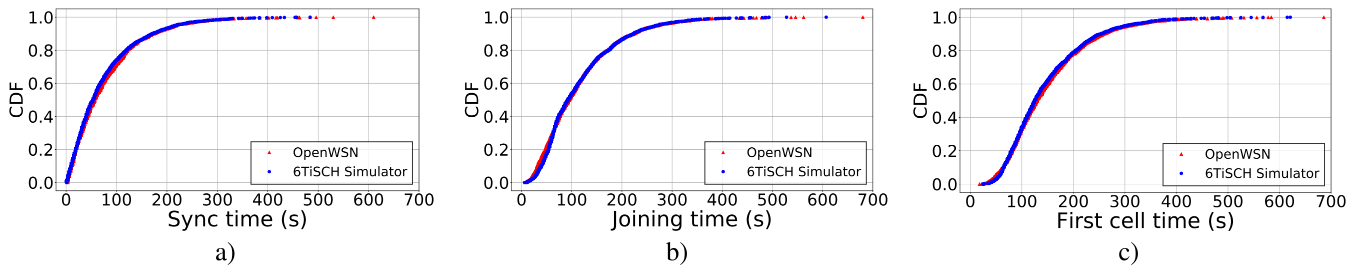
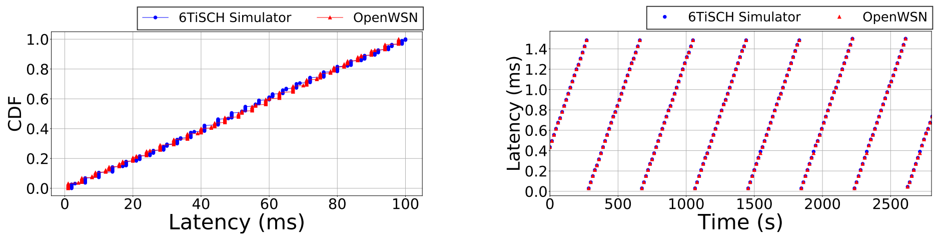


FIGURE 6 Comparison between the 6TiSCH Simulator and OpenSim in terms of synchronization time, joining time and the time it takes for a node to have the first dedicated cell since joining the network.

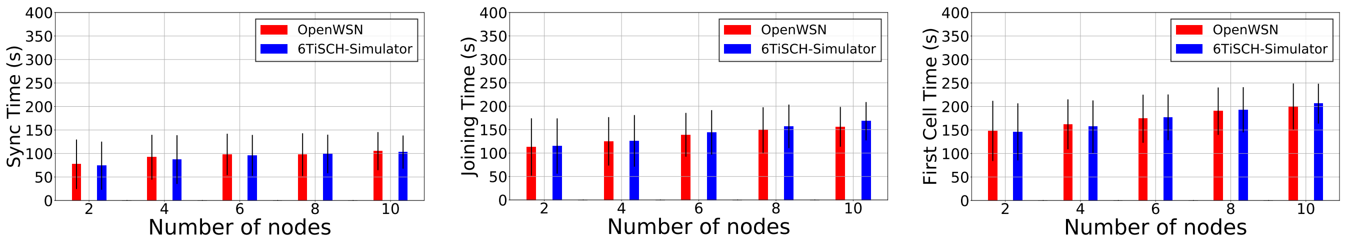
In the second set of experiments we asses how the performance in terms of latency is reproduced in the 6TiSCH Simulator in comparison to OpenWSN. In this case we use two real OpenMote devices that are running the OpenWSN firmware. The OpenMote devices consist of an OpenMote-CC2538 attached to an OpenUSB (34). Once the leaf node has synchronized, joined and scheduled its first dedicated cell, we send 15s periodic UDP data packets from the leaf node to the root. Figure 7 a) shows that the CDFs for both platforms match and that the average latencies are practically identical, with a difference of only 0.45%. Additionally, Figure 7 b) shows that the latency patterns due to uncoupled timing between the App timer and the TSCH timer are correctly produced. This experiment not only shows that the simulator is compliant with the OpenWSN 6TiSCH implementation but also produces equivalent results as the ones obtained with OpenWSN running on real hardware.



a) CDF for the latency b) Latency in function of time

FIGURE 7 Latency comparison between the 6TiSCH Simulator and OpenWSN.

The final set of experiments consists of simulating 6TiSCH mesh networks with different numbers of nodes. We have randomly generated up to 500 full-mesh networks and used them for both the 6TiSCH simulator and OpenWSN (OpenSim). As we did in the first set of experiments, we log the sync time, joining time and first cell time of each node and we plot them in Figure 8 as absolute values starting from the initialization of the network. The bars represent the average value of each run of all the nodes (except the root node). Figure 8 shows that times are similar for all networks. For example, in case of the first cell time, the differences are 1.35%, 3.08%, 1.69%, 1.55% and 2.91% for 2, 4, 6, 8 and 10 nodes respectively.



a) Sync Time b) Joining Time c) First Cell Time

FIGURE 8 Comparison between the 6TiSCH Simulator and OpenWSN (OpenSim) in terms of synchronization time, joining time and first cell time for different network sizes.

6 | PERFORMANCE EVALUATION

In order to give an overview of the computational performance of the 6TiSCH simulator and analyze its scalability, we have carried out simulations varying the number of nodes in the network. We have used two independent testbeds, Testbed 1 and Testbed 2, the first one being more powerful than the second one:

- **Testbed 1** consists of a server equipped with 56 Intel Xeon E5-2680 CPUs at 2.40GHz and 35840 KiB of CPU cache. Each CPU has 14 internal cores.
- **Testbed 2** consists of a desktop computer equipped with 4 Intel Core 2 Duo E4-500 CPUs at 2.20GHz and 2048 KiB of CPU cache. Each CPU has 2 internal cores.

The tests were performed using only one CPU per run without additional process load. Each run consisted of running a fully-meshed 6TiSCH network for 2000 cycles (3000 seconds when using a timeslot of 15 ms). Once all the nodes in the nodes in the network were synchronized, joined the network and allocated their first dedicated cell to their preferred parent, each node sent 1 packet every 5 seconds to the root. When considering DIOs, EBs, DAOs and DATA packets, each node sent on average of 1500 packets during the simulation.

Figure 9 (a) shows the CPU time (user time) used by the process running the 6TiSCH simulator. It shows that the time required for simulating a 6TiSCH network of 1000 nodes scales up to 31 one hours when using Testbed 1. When using Testbed 2, this time increases to 38 hours.

Regarding the memory usage, both testbeds show an approximately linear-like behavior, having a maximum of memory usage of less than 97 MiB for 1000 nodes.

These results are merely an indicator of the overall performance of the simulator since they are only valid for this specific scenario. CPU times and memory used for simulating different network sizes will vary depending on the number of cycles simulated and the traffic load. For example, 1000 runs of 1000 cycles each of a network with 50 nodes, can be simulated in less than 8 minutes.

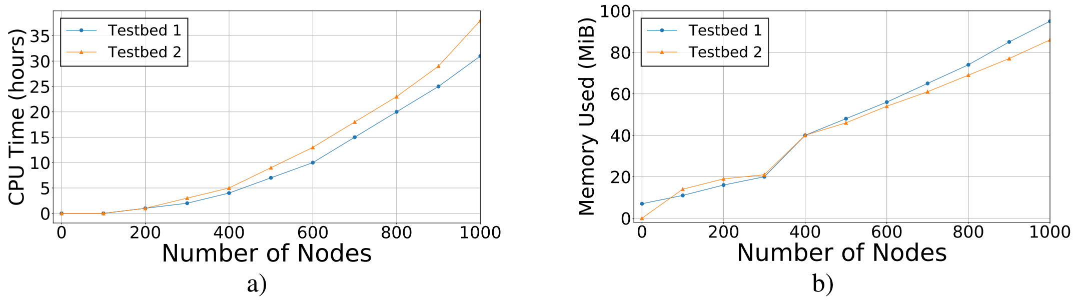


FIGURE 9 CPU Time and memory consumed for different network sizes in two different testbeds.

7 | EXAMPLE USE CASES

The 6TiSCH Simulator is an interesting tool to answer questions that commonly arise when running "what-if" scenarios with different topologies, traffic patterns and application-level networking requirements. This section gives 5 example use cases of the 6TiSCH simulation. The results of these examples have been published in previous works.

Network formation and joining process: *How does the number of nodes influences the network time formation?* Vučinić *et al.* conducted a study on the optimal broadcast transmission probability in 6TiSCH networks (17). One of the major research contributions of the paper is the given set of values for improving the network formation time. Figure 10 illustrates how the collision probability in a broadcast cell increases in terms of network density. For this work, the 6TiSCH Simulator easily allows to not only perform quick simulations with different broadcast parameters (i.e. different EB periods) but also implement new broadcast strategies (i.e. test the new Bayesian broadcast mode). One of the outcomes of this work is that by using probabilities of $P_{DIO} = 1/3$ and $P_{EB} = 0.1$ the network formation time is the lowest for networks of up to 45 nodes. Also, the Bayesian broadcast is introduced, which is implemented in both the 6TiSCH Simulator and OpenWSN.

Scheduling policies: *Which is the average latency of a packet for different scheduling alternatives?* Daneels *et al.* present a new scheduling function for recurrent traffic patterns, called ReSF (11). ReSF calculates minimal-latency paths from source to sink, only activating these paths when recurrent traffic is expected. They also compare ReSF with other state-of-the-art scheduling functions such as LLSF and SF0. Figure 11 shows how ReSF optimizes latency and outperforms the mentioned SFs, which can have delays of up to several seconds. This work is an example of how the 6TiSCH Simulator can be used for studying the performance of 6TiSCH networks (in this case, latency and energy consumption) and for proposing new methods to improve it. ReSF is implemented and tested for a number of configurations, while varying the traffic rate and its period change probability on networks up to 200 nodes. For this work, the 6TiSCH Simulator is an easier and more scalable solution than OpenWSN running in OpenSim or on real hardware.

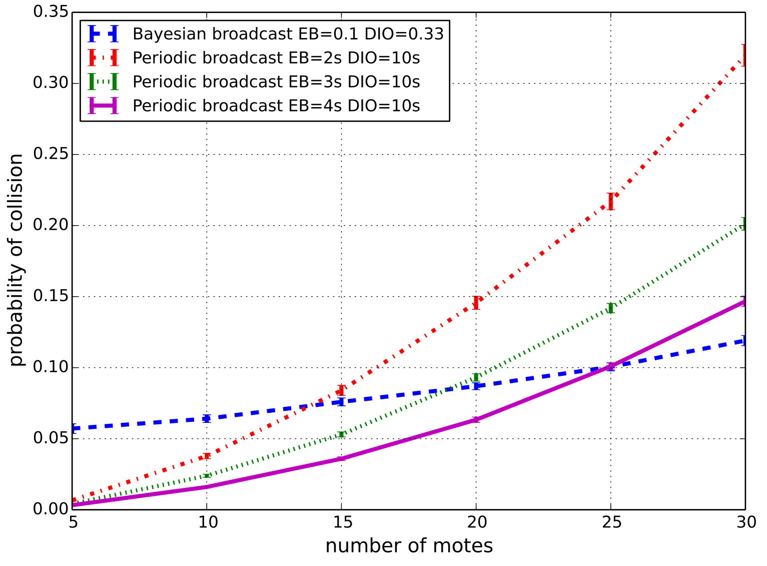


FIGURE 10 Probability of collision as a function of the number of nodes in the network. Reproduced from (17).

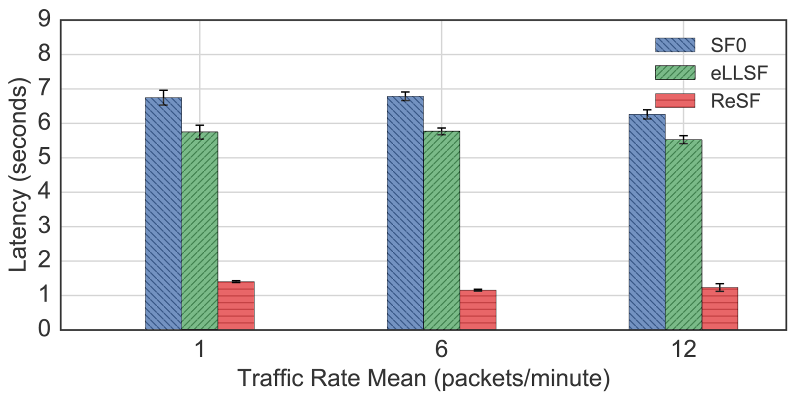
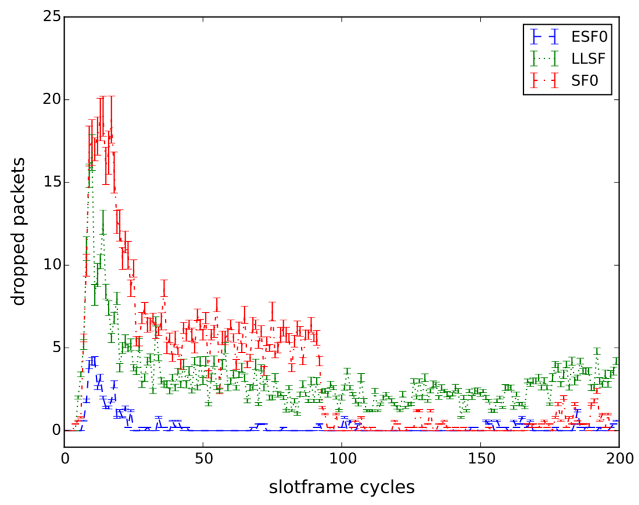


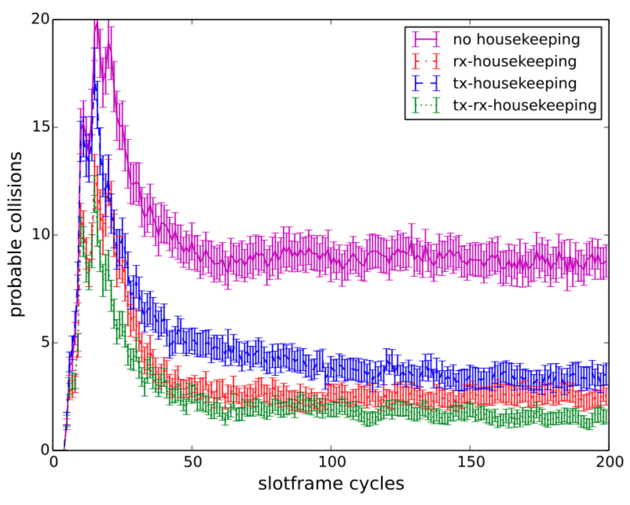
FIGURE 11 Latency obtained with 100 nodes for different scheduling functions (SF0, eLLSF and ReSF). Reproduced from (11).

Cell selection: *How can a SF avoid collisions when selecting cells?* Duy *et al.* propose the ESF0 scheduling function for selecting cells with low collision probability (9). By analyzing the cell density of a given portion in the schedule, nodes can select the portion which has the least number of used cells. As shown in Figure 12 a, collisions are notably reduced when using ESF0 due to portions of the schedule that are distributed more uniformly than in SF0 and LLSF so that the collision probability is lower. Another approach for avoiding scheduling collisions is presented by Muraoka *et al.* (15). In this work, a reactive policy of cell relocation is proposed to reduce packet collisions by measuring and tracking the PDR at every cell. A cell with a bad PDR is replaced if its PDR is unexpectedly lower than the average. Since these unexpected low PDRs are usually due to collisions, reactive cell allocation shows to be an effective approach for collision avoidance. Figure 12 b shows that, by running relocation algorithms at both transmitter and receiver, the number of collisions can be reduced significantly. These two works are also examples of using the 6TiSCH Simulator for fast and dynamic implementation of scheduling functions in order to reduce the 6TiSCH internal interference and increase network reliability.

Traffic performance: *How do queues behave when bursty traffic is present?* Kralevska *et al.* study the use of a newly proposed algorithm called Local Voting to optimize queue usage and perform load balancing (10). The Local Voting algorithm tries to minimize the maximum latency in the network by evenly distributing the traffic among the nodes in the network. This



a) Reproduced from (9)



b) Reproduced from (15)

FIGURE 12 How different scheduling functions tackle the packet collision problem in 6TiSCH.

traffic distribution is performed by monitoring the queue length of the neighbors of every node and adding or removing links accordingly to minimize latency and reduce power consumption.

Figure 13 illustrates, for bursty traffic of 5 packets per burst, how the queues are filled, showing the differences between the proposed algorithm Local Voting and SF0 for different cell thresholds. This last use case shows how the 6TiSCH Simulator metrics (in this case, queue length and queue drops) is a useful tool for implementing new scheduling functions to improve traffic management.

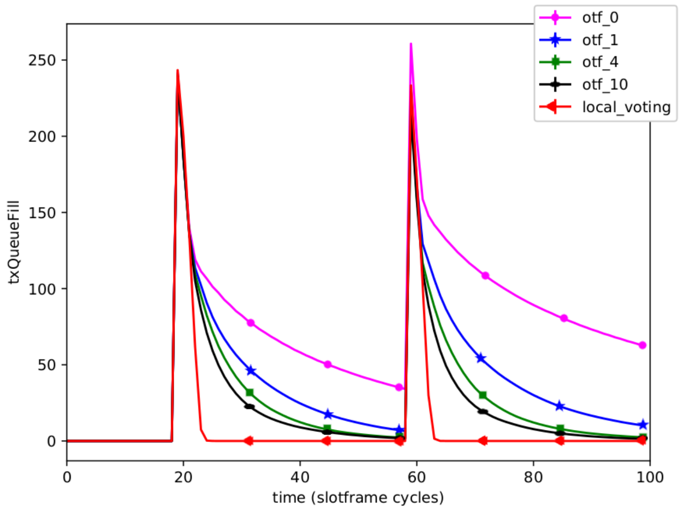


FIGURE 13 Queue usage during a simulation for different SF0 thresholds and when using a Local Voting scheme for load balancing. Reproduced from (10).

8 | CONCLUSIONS

This article introduces the 6TiSCH Simulator, a tool for easily evaluating the performance of a fully standards-based 6TiSCH network. It allows end-users and potential 6TiSCH adopters to conduct "what-if" scenarios and evaluate whether 6TiSCH is applicable to their application. It allows researchers to compare the performance of their optimization against the fully standards-complaint solution. It allows the 6TiSCH working group to verify the performance of what is being standardized. The simulator is published under an open-source license. This article validates the tool, does a computational performance analysis and shows 5 example use cases of previously-published works which use the 6TiSCH Simulator as the primary tool for performance evaluation.

ACKNOWLEDGMENTS

This work is partially supported by the Flemish FWO SBO S004017N IDEAL-IoT (Intelligent DEnse And Long range IoT networks) project, the ICON CONAMO project with support from VLAIO (Flanders Innovation and Entrepreneurship), the Spanish Ministry of Economy and the FEDER regional development fund through the SINERGIA project (TEC2015-71303-R), and the European Commission through the H2020 F-Interop and H2020 ARMOUR projects.

References

[1] IEEE . IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer. *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011).* 2012;:1–225.

[2] Watteyne Thomas, Weiss Joy, Doherty Lance, Simon Jonathan. Industrial IEEE802.15.4e Networks: Performance and Trade-offs. In: IEEE; 2015; London, UK.

[3] Watteyne Thomas, Palattella Maria Rita, Grieco Luigi Alfredo. *Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement.* RFC7554: IETF; 2015.

[4] Vilajosana Xavier, Pister Kris, Watteyne Thomas. *Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration.* RFC8180: IETF; 2017.

[5] Wang Qin, Vilajosana Xavier, Watteyne Thomas. *6top Protocol (6P).* draft-ietf-6tisch-6p-protocol-09 [work-in-progress]: IETF; 2017.

[6] Vučinić Mališa, Simon Jonathan, Pister Kris, Richardson Michael. *Minimal Security Framework for 6TiSCH.* draft-ietf-6tisch-minimal-security-03 [work-in-progress]: IETF; 2017.

[7] Chang Tengfei, Vučinić Mališa, Vilajosana Xavier. *6TiSCH Minimal Scheduling Function (MSF).* draft-chang-6tisch-msf-00 [work-in-progress]: IETF; 2017.

[8] Watteyne Thomas, Handziski Vlado, Vilajosana Xavier, et al. Industrial Wireless IP-based Cyber-physical Systems. *Proceedings of the IEEE.* 2016;104(5):1025–1038.

[9] Duy Thang Phan, Dinh Thanh, Kim Younghan. Distributed Cell Selection for Scheduling Function in 6TiSCH Networks. *Computer Standards & Interfaces.* 2017;53:80–88.

[10] Kralevska Katina, Vergados Dimitrios J, Jiang Yuming, Michalas Angelos. A Load Balancing Algorithm for Resource Allocation in IEEE 802.15.4e Networks. *arXiv preprint arXiv:1709.06835.* 2017;.

[11] Daneels Glenn, Spinnewyn Bart, Latré Steven, Famaey Jeroen. ReSF: Recurrent Low-Latency Scheduling in IEEE 802.15.4e TSCH networks. *Ad Hoc Networks.* 2018;69:100–114.

[12] Juc Iacob, Alphand Olivier, Guizzetti Roberto, Favre Michel, Duda Andrzej. *Stripe: a Distributed Scheduling Protocol for 802.15.4e TSCH Networks.* Research Report RR-LIG-54: Laboratoire d'Informatique de Grenoble; 2017. Les rapports de recherche du LIG - ISSN: 2105-0422.

[13] Watteyne Thomas, Vilajosana Xavier, Kerkez Branko, et al. OpenWSN: a Standards-based Low-power Wireless Development Environment. *Transactions on Emerging Telecommunications Technologies (ETT).* 2012;23(5):480–493.

[14] Duquennoy Simon, Elsts Atis, Nahas BA, Oikonomou George. TSCH and 6TiSCH for Contiki: Challenges, Design and Evaluation. In: IEEE; 2017; Ottawa, Canada.

[15] Muraoka Kazushi, Watteyne Thomas, Accettura Nicola, Vilajosana Xavier, Pister Kristofer S.J.. Simple Distributed Scheduling with Collision Detection in TSCH Networks. *IEEE Sensors.* 2016;16(15):5848–5849.

[16] Palattella Maria Rita, Watteyne Thomas, Wang Qin, et al. On-the-fly Bandwidth Reservation for 6TiSCH Wireless Industrial Networks. *IEEE Sensors Journal.* 2016;16(2):550–560.

[17] Vučinić Mališa, Watteyne Thomas, Vilajosana Xavier. Broadcasting Strategies in 6TiSCH Networks. *Internet Technology Letters.* 2017;.

[18] Municio Esteban, Latré Steven. Decentralized Broadcast-based Scheduling for Dense Multi-hop TSCH Networks. In: :19–24ACM; 2016.

[19] Gomes Pedro Henrique, Watteyne Thomas, Krishnamachari Bhaskar. MABO-TSCH: Multihop and Blacklist-based Optimized Time Synchronized Channel Hopping. *Transactions on Emerging Telecommunications Technologies (ETT).* 2017;:1–15.

[20] Municio Esteban, Spaey Kathleen, Latré Steven. A distributed density optimized scheduling function for IEEE 802.15. 4e TSCH networks. *Transactions on Emerging Telecommunications Technologies.* ;:e3420.

[21] Aijaz Adnan, Raza Usman. DeAMON: A Decentralized Adaptive Multi-Hop Scheduling Protocol for 6TiSCH Wireless Networks. *IEEE Sensors Journal.* 2017;17(20):6825–6836.

[22] NS-2 Simulator Homepage http://www.isi.edu/nsnam/ns/[Online; accessed 15-01-2018]; 1995.

[23] Tyan Hung-Ying. Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation. PhD thesisOhio State University2002.

[24] NS-3 Network Simulator Homepage https://www.nsnam.org[Online; accessed 15-01-2018]; 2011.

[25] OMNet++ Network Simulator Homepage http://www.omnetpp.org[Online; accessed 15-01-2018]; 2001.

[26] Levis Philip, Lee Nelson, Welsh Matt, Culler David. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: :126–137ACM; 2003.

[27] Castalia Network Simulator Homepage https://github.com/boulis/Castalia[Online; accessed 15-01-2018]; 2006.

[28] Levis Philip, Madden Sam, Polastre Joseph, et al. TinyOS: an Operating System for Sensor Networks. *Ambient Intelligence.* 2005;35:115–148.

[29] Dunkels Adam, Gronvall Bjorn, Voigt Thiemo. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: :455–462IEEE; 2004.

[30] Linear Technology, Application Note: Planning A Deployment, 2017. http://www.linear.com/docs/43189.

[31] Le Hanh-Phuc, John Mervin, Pister Kris. *Energy-Aware Routing in Wireless Sensor Networks with Adaptive Energy-Slope Control.* EE290Q-2 Spring: University of California, Berkeley; 2009.

[32] Brun-Laguna Keoma, Diedrichs Ana Laura, Dujovne Diego, Léone Rémy, Vilajosana Xavier, Watteyne Thomas. (Not So) Intuitive Results from a Smart Agriculture Low-power Wireless Mesh Deployment. In: :25–30ACM; 2016; New York, NY, USA.

[33] Vilajosana Xavier, Wang Qin, Chraim Fabien, Watteyne Thomas, Chang Tengfei, Pister Kris. A Realistic Energy Consumption Model for TSCH Networks. *IEEE Sensors Journal.* 2014;14(2):482–489.

[34] Vilajosana Xavier, Tuset Pere, Watteyne Thomas, Pister Kris. OpenMote: Open-Source Prototyping Platform for the Industrial IoT. In: :211–222EAI; 2015; San Remo, Italy.