# Scalable and Interactive Simulation for IoT Applications With TinySim

Gonglong Chen, Wei Dong, *Member, IEEE*, Fujian Qiu, Gaoyang Guan, Yi Gao, *Member, IEEE*, and Siyu Zeng

***Abstract*—Recent years, the rapid development of Internet of Things (IoT) technologies and applications have been witnessed. Three important features are characterized in modern IoT applications: 1) device heterogeneity; 2) long-range communication; and 3) cloud/edge-device integration. Difficulties are raised by the above features toward IoT application developers, e.g., predicting and evaluating the performance of the entire IoT application system. To deal with the above difficulties, we design and implement an IoT simulator, TinySim, which satisfies the requirements of high fidelity, high scalability, and seamless transplantation. TinySim takes advantage of the hardware-independent features of TinyLink programming language. Hence, a similar code can be used for both simulation and execution on real hardware platforms. Many virtual IoT devices can be simulated by TinySim at the PC end. These IoT devices can send or receive messages from the cloud or smartphones, making it possible for the developers to evaluate the entire system without the actual IoT hardware. We connect TinySim with Unity 3-D to provide high interactivity. To reduce the event synchronization overhead between TinySim and Unity 3-D, a dependence graph-based approach is proposed. We design an approximation-based approach to reduce the number of simulation events, greatly speeding up the simulation process. We carefully evaluate TinySim using benchmarks and two concrete case studies. TinySim can simulate representative IoT applications, such as smart flowerspot and shared bikes. We conduct extensive experiments to evaluate the performance of TinySim. Results show that TinySim can achieve high accuracy with an error ratio lower than 9.52% in terms of energy and latency. Further, TinySim can simulate 4000 devices within 11.2 physical-minutes for ten simulation-minutes, which is about 3× faster than the state-of-art approach.**

***Index Terms*—Interactive development, Internet of Things (IoT), scalability, simulation, speculative execution.**

## I. Introduction

THE RECENT years have witnessed the rapid development of Internet of Things (IoT) technologies and applications [1]. Compared with early IoT systems like sensor networks [2], [3], modern IoT systems are characterized by the following important features.

1) *Device Heterogeneity:* IoT devices become increasingly heterogeneous, including both resource-constrained devices like MSP430 and much more capable devices like Raspberry PI.[1]
2) *Long Range Communication:* Low power wide area networks (LPWAN), e.g., NB-IoT [4] and LoRaWAN [5] are widely considered promising technologies to interconnect a vast number of IoT devices into the Internet.
3) *Cloud/Edge-Device Integrated:* For example, the Mobike bicycle sharing system, consisting of more than five million orange-colored bicycles with smart locks,[2] is a representative IoT application. A Mobike smart lock is an IoT device that interacts with users via smartphones and tracks users' trajectory by continuously transmitting location information to the cloud. As such, the entire Mobike application uses an architecture consisting of the IoT device (smart lock), the cloud (for management), and the smartphone (for interaction with users). Note that the cloud servers can be also replaced with other similar services, e.g., the edge services. Especially under the delay-sensitive scenario, the edge server can provide shorter response latency.

Developers are often confronted with difficulties in implementing a real-world IoT application. For example, developers may have difficulties in evaluating the entire application before the IoT devices are designed and deployed. Unlike PCs and smartphones, IoT devices are special ones which vary with different applications. Hence, the design takes time.

The above difficulties would be tackled by using an accurate and scalable IoT simulator. With its availability, IoT developers can quickly simulate the entire application and evaluate the feasibility of their innovative ideas.

Unfortunately, existing simulation tools are insufficient and cannot help answer the above questions [6], [7], [8]. Existing IoT simulators, such as SmartThings [9], Seebo [10], and IOTIFY [11], only provide simulation at the functionality level for the IoT device, and cannot capture the detailed timing and energy performance. Embedded simulators, such as Avrora [6], PoLite [8] albeit accurate in simulating the device level behaviors, suffer from low simulation speed and thus cannot scale

Manuscript received 8 March 2023; revised 12 April 2023 and 25 May 2023; accepted 4 June 2023. Date of publication 16 June 2023; date of current version 21 November 2023. This work was supported in part by the National Key R&D Program of China under Grant 2019YFB1600700; in part by the National Natural Science Foundation of China under Grant 62072396 and Grant 62272407; in part by the "Pioneer" and "Leading Goose" R&D Program of Zhejiang Province under Grant 2023C01033; and in part by the National & Zhejiang Provincial Youth Talent Support Program. *(Corresponding author: Wei Dong.)*

The authors are with the College of Computer Science and the Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University, Hangzhou 310027, China (e-mail: desword@zju.edu.cn; dongw@zju.edu.cn; qiufj@zju.edu.cn; ggy@zju.edu.cn; gaoyi@zju.edu.cn; zengsy@zju.edu.cn).

Digital Object Identifier 10.1109/JIOT.2023.3285244

[1] https://www.raspberrypi.org

[2] China's Mobike raises U.S. $600M to expand its bikes on-demand service worldwide, https://techcrunch.com/2017/06/15/mobike-raises-600-million/.

to a large number of IoT devices in a typical system. Existing sensor network simulators, e.g., EmStar [7], usually simulate a multihop wireless network and thus lack support for LPWAN protocols. Existing wireless or network simulators (e.g., ns-2 [12], ns-3 [13], NetSim [14], and OMNeT++ [15]) albeit accurate in simulating the networking behaviors, lose accuracy in simulating device-level behaviors.

In this article, we aim to design and implement a simulator for modern IoT applications satisfying the following requirements.

1) *High Fidelity:* The simulator should capture the device-level behaviors in a fine-grained manner. Otherwise, it is impossible to evaluate the timing and energy performance of the system.
2) *High Scalability:* Future IoT systems would consist of a vast number of IoT devices. The simulator should execute at a high speed and can scale to many IoT devices.
3) *High Interactivity:* The simulator should provide a user-friendly debug and program UI. Developers can conveniently create scenarios or change environments to trigger events to verify the functionality of IoT applications.

We have designed and implemented TinySim, to meet the above requirements. To use TinySim, the developer writes a device code in a hardware-independent language, TinyLink language, which can be directly used for our simulation. The use of TinyLink code also allows us to capture fine-grained device-level behaviors. We map TinyLink code into hardware-dependent instructions without actually running them. This approach allows us to obtain high fidelity without overhead to execute the instructions.

To increase scalability, we propose an approximation-based approach to reduce the time-consuming events, e.g., the transmission events and the collision events. The behaviors of the time-consuming events are trained using a machine algorithm, e.g., long short-term memory (LSTM) [16]. The results (e.g., the transmission delay) after executing events are predicted by the machine learning algorithm. The simulation speed is further sped up by distributing events to many machines. TinySim supports common LPWAN protocols, including NB-IoT [4] and LoRaWAN [5]. We have also provided a development framework based on a state machine language state map compiler (SMC) [17] to facilitate incorporating other IoT protocols.

To provide the high interactivity, we connect TinySim with a powerful gaming engine Unity 3-D [18]. The different simulation speeds of TinySim and Unity 3-D lead to the large overhead of the event synchronization. It also makes existing simulators incapable of being integrated directly. To deal with this problem, we propose a dependence graph-based approach to reduce the synchronization overhead while maintaining a good programming experience.

The contributions can be summarized as follows.

1) We design and implement TinySim, a simulator for modern IoT applications, satisfying three requirements of high fidelity, high scalability, and high interactivity. We have open sourced our simulation code at [19].
2) We propose an approximation-based approach that can reduce the number of simulation events to accelerate the simulation speed. With the improved simulation speed, TinySim achieves larger scalability. The simulation speed is further improved by distributing simulation tasks to multiple machines. With our technique, TinySim can support 4000 simulated devices using eight machines.
3) We have connected TinySim with a powerful virtual scenario engine, Unity 3-D. The developers can create IoT application easily. Through carefully designing an event synchronization approach for TinySim and Unity 3-D that accepts actions from the real world, the developers can also interact with the simulated devices using the real-world smartphone. The above features enable more interactive IoT application development.
4) We extensively evaluate TinySim using benchmarks and two concrete case studies. Results show that: a) TinySim can simulate representative IoT applications and b) TinySim can achieve high fidelity at the device end. TinySim can now simulate three mainboards (Arduino UNO, Raspberry Pi 2, and BeagleBone Black), six frequently used peripherals and two communication protocols (NB-IoT and LoRaWAN) with simulation error at most 9.52%.

## II. Related Work

This section reviews existing simulation tools in the areas of IoT, embedded sensor networks, and wireless networks.

*Simulation Tools for IoT:* SmartThings [9] and Seebo [10] are IoT development platforms based on Web-IDE. To validate the functionality of IoT applications, they provide a simulator that can present sensed information by generating virtual data. However, the functions provided by the above simulator are rather limited in developing real IoT applications considering device constraints and user requirements. For example, minimizing the power consumption for a smart spot while meeting monitoring requirements.

IOTIFY [11] and BevyWise [20] are IoT cloud platform simulation systems. By setting the network conditions (e.g., jitter time), developers can evaluate the device accessing delay when using different IoT message exchange protocols (e.g., MQTT [21]). IoTSim [22] simulates the performance of MapReduce-based cloud computing network when connecting a large number of IoT devices. EdgeMiningSim [23], [24] is a simulation-driven methodology for enabling IoT Data Mining. Such a methodology drives the domain experts in disclosing actionable knowledge, namely, descriptive or predictive models for taking effective actions in the constrained and dynamic IoT scenario. TinySim simulates an IoT system not only at the network level but also at the device level. The device level simulation consists of fine-grained activities, such as timing behaviors and power profiles.

*Simulation Tools for Embedded Sensor Networks:* TOSSIM [25] is a discrete event simulator using a probabilistic bit error model for network simulations. PowerTOSSIM [26] and TimeTOSSIM [27] extend TOSSIM to provide detailed energy consumption models and timing behavior profile. Simwet [28] is a TOSSIM-based simulator

provides a power consumption model for energy harvesting and transfer scenarios. EmStar [7] offers a range of runtime environments from simulations to actual deployments. TinySim also integrates the simulated (e.g., the IoT devices and the base station) and the real devices/servers (e.g., the smartphone and the cloud) in the IoT application system. In fact, TinySim and EmStar are orthogonal, and the differences are that the two simulators place the real part at different positions. For example, TinySim can also replace one of the simulated devices with the real devices by carefully designing the incorporating interfaces. By providing the replaceable components, it is possible to improve the fidelity of the specific IoT devices. The above simulators only target at multihop wireless protocols, which is rather not enough for simulating heterogeneous IoT applications. To tackle this problem, we propose a development framework for simulating various wireless protocols.

Avrora [6] is a cycle-accurate instruction-level sensor network simulator. PoLite [8] improves the simulation efficiency by reducing the synchronization interval at radio-level and MAC-level. D-SnapSim [29] skips all thread synchronization and just executes threads. When erroneous simulation results are found, they will rollback and re-execute. Different from the above simulators, TinySim utilizes a machine learning-based approach to reduce the number of simulation events. Moreover, the simulation is further sped up by distributing simulation tasks to multiple machines.

Cooja [30] is a simulator for Contiki OS [3]. It enables cross-level simulation, including the network level, the operating system level, and the instruction level. However, the individual node is always simulated at one of these levels. It means that Cooja still holds the drawback of each level simulation, e.g., lack of high fidelity at the network level and losing efficiency at the instruction level. Different from Cooja, TinySim not only achieves high fidelity by capturing fine-grained device behaviors but also maintains high efficiency with a distributed simulation approach.

*Simulation Tools for Wireless Networks:* LoRaSim [31] is a simulator for LoRaWAN [5] and it captures specific LoRa link behaviors such as collision. TinySim can simulate both NB-IoT and LoRaWAN. Different from LoRaSim, TinySim simulates an entire IoT system and LoRaSim can be incorporated into TinySim as a component. NetSim [14] and OMNeT++ [15] are all network-level simulators. They provide a wide range of network protocol simulation. However, it is not fine-grained enough to profile detailed IoT device behaviors such as the MCU timing behaviors. ns-2 [12] and ns-3 [13] are also network-level simulators and provide power profiling and time profiling with plugins, however, they can not support heterogeneous hardware.

Table I compares various approaches concerning five key desired features.

1) *Power Profile:* Power is an important factor, especially for resource-constrained IoT devices.
2) *Time Profile:* Accurate timing behavior profiling is of importance for analyzing IoT application performance. The simulator should support the above two profiles to provide high fidelity.

TABLE I
RELATED WORK COMPARISON

| Approach | Fidelity | | Interactivity | Scalability | Heterogeneity |
|---|---|---|---|---|---|
| | Power Profile | Time Profile | Virtual Scenario | High Speed | Hetero. H/W |
| SmartThings[9] | × | × | × | × | ✓ |
| Seebo [10] | × | × | × | × | ✓ |
| IOTIFY [11] | × | × | × | × | ✓ |
| BevyWise [20] | × | × | × | × | ✓ |
| IoTSim [22] | × | × | × | × | × |
| TOSSIM [24] | × | × | partial | × | × |
| PowerTOSSIM [25] | ✓ | ✓ | partial | × | × |
| TimeTOSSIM [26] | × | ✓ | partial | × | × |
| Simwet [27] | ✓ | ✓ | × | × | × |
| EmStar [7] | × | × | × | × | × |
| Avrora [6] | × | ✓ | × | × | × |
| PoLite [8] | × | ✓ | × | ✓ | × |
| D-SnapSim [27] | × | ✓ | × | ✓ | × |
| ns-2 [12], ns-3[13] | ✓ | ✓ | × | × | × |
| NetSim [14] | × | × | × | × | × |
| OMNeT++ [15] | ✓ | ✓ | partial | × | × |
| Cooja [29] | ✓ | ✓ | partial | × | × |
| LoRaSim [30] | × | × | × | × | × |
| **TinySim** | ✓ | ✓ | ✓ | ✓ | ✓ |

3) *Virtual Scenario:* Providing an interactive programming interface is also important for creating IoT applications associated with scenarios. Note that several simulators can only partially support scenarios because they can only change the position of wireless devices, but not people. The ability to create virtual scenarios can provide higher interactivity for the developers.
4) *High Speed:* It is important to scale to a vast number of IoT devices. With higher simulation speed, the simulator can support more devices, thus achieving larger scalability.
5) *Heterogeneous H/W Support:* IoT applications consist of various hardware components and it is thus of importance to support heterogeneous hardware.

## III. TINY LANGUAGE

*TinySim Is a New Programming Style:* Our design can be used in many discrete event-based simulator. We can use Arduino-like programming language to program IoT applications. We utilize Tinylink language as an example.

Tiny language is a hardware-independent programming language that enables the rapid development of IoT applications. We have designed our Tiny language compiler to produce efficient code for several IoT platforms, e.g., Arduino, Raspberry Pi, and BeagleBone. In addition, Tiny language provides APIs to easily connect to mainstream IoT cloud platforms, including Ali IoT Cloud [32] and IBM Watson [33].

Fig. 1 shows an example of Tiny language code for an IoT application that monitors the environment of a houseplant. The application periodically samples and updates soil data into the cloud via NB-IoT [4]. Function `setup()` is used to initialize the parameters of establishing the connection of NB-IoT module (lines 3–6). Function `loop()` is used

```
1  void upload();
2  void setup() {
3    TL_NB.init();
4    TL_DBG("Node", "Setup.");
5    MQTT mqtt = TL_NB.fetchMQTT();
6    mqtt.connect(servName, port);
7  }
8  void loop() {
9    TL_DBG("Node", "Node wake up.");
10   TL_Soil_Moisture.read();
11   upload();
12 }
13 void upload() {
14   TL_DBG("Node", "Setup.");
15   String light = String(TL_Light.data());
16   String outMsg;
17   outMsg = String("http://hostname/cl.php?");
18   outMsg += String("light=") + light;
19   mqtt.publish(outMsg);
20 }
```

Fig. 1. Application code using Tiny language.

```
1 "nodenum": 10,        // the number of node.
2 "base_station": 2,    // the number of BS.
3 "area": {             // area constraints.
4   "area_x": 1000,     // x is within [0,1000m].
5   "area_y": 1000},
6 "channel":["Node"], // log output channel.
7 "hardware":[          // tracked hardware.
8   "NB", "Light", "Soil_Moisture"],
```

Fig. 2. Example of simulation configuration (env.json).

to periodically sample soil data, and upload the data into the cloud via NB-IoT (lines 9–11). By setting the necessary configurations (e.g., device ID, product ID, and token) on the cloud, the IoT application can connect and upload data to the cloud. The soil data is uploaded into the cloud via proper setting of `hostname` and data structure (lines 17–19). Note that `TL_DBG()` is used for printing debug information and will be ignored when compiling the code into a real device.

## IV. TINYSIM USAGE

In this example, we present how to simulate a smart flower spot application monitoring the moisture of a houseplant. This application collects sensing data to the cloud and displays the data on smartphones.

*Step 1 (IoT Application Development):* The developer can use and configure existing IoT cloud, e.g., IBM Watson. The IoT cloud serves as a bridge between the IoT device and the smartphone. The developer can also develop mobile apps for displaying the sensing data. Most importantly, the developer can use Tiny language to develop IoT applications (App.cpp) at the device end in a hardware-independent manner (e.g., as shown in Fig. 1). The developer can upload the application code to the TinySim cloud via "TinySim -a App.cpp."

*Step 2 (Simulation Configuration):* The developer can set the simulation configuration as shown in Fig. 2. TinySim provides a series of useful settings for developers. For example, the developer can set the attribute "area" to limit the range of areas that IoT devices and gateways will be randomly placed (e.g., 1000 m × 1000 m area). To print the log information, the developer can add values to the attribute

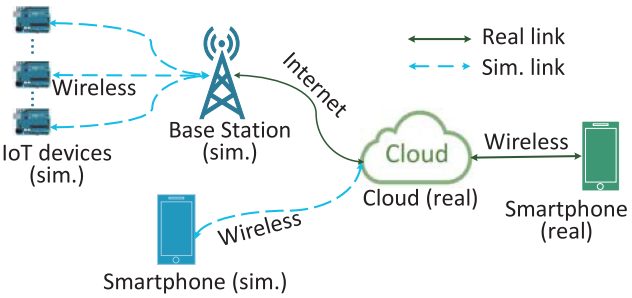


Fig. 3. Network architecture of simulated entire IoT applications in TinySim.

"channel." The attribute "hardware" is used for enabling the tracking of timing behavior and power consumption of the corresponding hardware.

*Step 3 (Execute the Simulation):* The simplest way for the developer to execute the simulation is "TinySim -r." Then, TinySim will not stop running the application at the cloud until the developer sends "TinySim -s." The developer can also define the running time on the simulation by "TinySim -r -t 3600" (it means executing one simulation-hour). After finishing the simulation, TinySim would output the debug information and the profile of the hardware enabled in the configuration file (env.json).

TinySim also supports the interactions between real mobile devices and the simulated IoT devices. The developer needs to write a proper event handler for dealing with the arrival messages from mobile devices (a concrete example will be shown in Section IX). To enable this execution mode, the developer can send the command "TinySim -r -m REAL." By default, the simulation is executed with simulated mobile devices (VIRTUAL mode).

*Step 4 (Interact With the Simulation):* The developer can enter the interactive mode by sending "TinySim -r -debug" With the parameter "-debug," the developer can debug the application step by step. The interactive commands are similar with gdb. The developer can also inspect devices' profiles by sending "TinySim -r -i." For example, when the developer sends "ShowHWState -t 2," in the example shown in Fig. 1, states of NB-IoT are periodically changed between SEND and SLEEP.

## V. OVERVIEW

### A. TinySim Network Overview

Fig. 3 depicts the network architecture of TinySim, which covers most of a typical IoT application, i.e., the devices, the base station, the cloud, and the smartphone.

*The Communication Links Between Devices and the Base Station:* TinySim provides the simulated communication links between the devices and the base station. Currently, TinySim supports the two most promising LPWAN technologies: 1) NB-IoT and 2) LoRaWAN. Because the implementation of NB-IoT is closed source, we implement most of the key procedures in NB-IoT according to the 3GPP definition [4]. LoRaWAN is open source [34] and we reuse most of the source code. The hardware component is replaced with the simulated version (e.g., the radio and ADC).

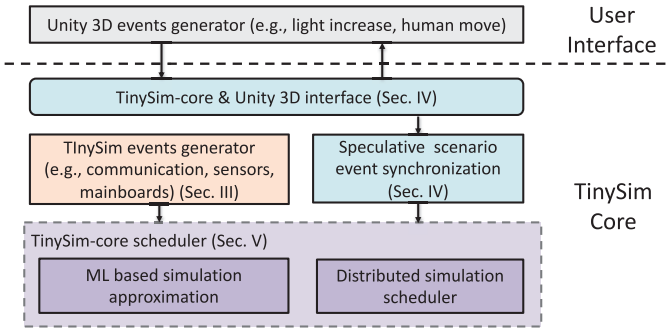


Fig. 4. Overview of TinySim architecture.

*The Communication Links Between the Base Station and the Cloud:* This is simulated using the real Internet via wired connections between the PC and the cloud. It is a simple but effective approach especially under certain scenarios [35], e.g., the predictable delay at midnight.

*The Communication Links Between the Smartphone and the Cloud:* For this link, TinySim supports both the real mode and the simulated mode. For the reality mode, by installing the customized Android application, the developer can send messages to the cloud via WiFi using an application layer protocol (e.g., CoAP [36]). With an appropriate message handler implemented in the IoT devices, the developer can interact with the devices via the cloud. For the simulated mode, TinySim simulates the behavior of sending messages to the cloud from the smartphone. In this article, we focus on modeling the important behaviors of the IoT devices and assume the link quality between the smartphone and the cloud is favorable in most cases.

In TinySim, the Cloud part is connected with real services since the services are now mature and easy to configure and deploy through buying the services from the provider, e.g., the Ali Cloud, Tencent Cloud, etc. With real services, the simulation can be more accurate. On the other hand, the deployment of the base stations may be difficult for developers especially those provided by the operators. For the smartphone, TinySim provides two options for developers to use in the simulation, i.e., the interactive experiments between the simulated devices and the real smartphone APP, or the scalable experiments using simulated smartphone. The design goal of the integrated TinySim is to improve simulation accuracy, while making the IoT application simulation easy.

### B. TinySim Architecture Overview

Fig. 4 presents the overview of the TinySim architecture. TinySim is based on the discrete event execution model. All of the behaviors are abstracted as events (e.g., communication between IoT devices, capturing sensor data, etc.) and fed into the TinySim-core to schedule the execution of the events.

*1) Event Engine: Event Generation:* TinySim is a discrete event simulator. The events are generated by the hardware-independent function or the hardware-dependent components. The hardware-independent functions are directly separated by the event handler and inserted into the event queue. For the hardware-dependent components, TinySim generates random events following the data generation model (e.g., the illumination data for light sensors). The data generation model is obtained from real-world traces. The hardware-dependent functions (e.g., function read() of the light sensor) are replaced with the version that can be executed at the PC.

*Interruption Simulation:* Interrupts can be triggered by the hardware components (e.g., the mainboard) or other communication parts (e.g., the smartphone). When the interrupts are from the hardware components (e.g., the clock), it is inserted into the discrete event queue directly by the event scheduler. To capture the clock drift due to the interrupt events, we estimate the elapsed time of the interrupt execution at the instruction level. The drifted clock is then compensated by the estimated execution time. For the interrupts from the other communication part, TinySim simulates the interrupts by generating the random events following the data generation model (e.g., the CoAP messages from the smartphone). The data generation model is obtained from real-world traces.

*2) Expressive User Interface:* We connect TinySim-core with an expressive UI, Unity 3-D [18], to provide a better development experience for IoT application developers. Developers can simply add new modules and debug the IoT application interactively. TinySim-core captures key events (e.g., the changes on the light) generated from the Unity 3-D engine through the interface (as described in Section VI). There are inconsistences between Unity 3-D and TinySim-core due to different execution speed. Therefore, we provide a speculative event synchronization approach to eliminate the inconsistences while minimizing the synchronization overhead (as detailed in Section VI).

*3) TinySim-Core:* The TinySim-core plays an important role on scheduling the discrete events. It handles the events not only generated from the application codes written by developers but also from the Unity 3-D. All the above generated events are fed into the TinySim-core scheduler. TinySim-core provide three types of event scheduler: 1) the ML-based simulation approximation; 2) the distributed simulation; and 3) the FIFO scheduler (as detailed in Section VII). The ML-based simulation approximation approach can improve the simulation speed of the communication-related events. We train an offline model to directly predict the performance results of the communication behaviors. The distributed simulation scheduler utilizes multiple machines to execute the events concurrently.

The design goal of TinySim is to provide developers the estimated evaluation results of the delay, the packet reception ratio (PRR), the energy consumption, and the detailed timing of hardware states (e.g., the transmission, the reception, the sleep, etc.). To this end, TinySim translated the execution results of the above events into the final performance metrics.

In the next, we will detail the important modules of TinySim, e.g., the interactive development with the virtual scenario creation and the scalable simulation.

## VI. Interactive Virtual Scenario

One of the most attractive features of TinySim is the virtual scenario creation. With this feature, developers can easily create scenarios and rapidly verify the ideas of IoT applications. The efficiency of the simulation development is important for pre-examing the IoT applications before large scale

deployment. The ability to create IoT scenarios is difficult in the current IoT development. With a representative development platform, the developers can create IoT scenarios easily, and produce better IoT applications targeting at the specific scenarios. To this end, we connect TinySim with Unity 3-D, a powerful cross-platform game engine [18]. Now, two typical scenarios are supported in TinySim: 1) the room scenario with furniture (indoor) and 2) the city scenario with buildings and trees (outdoor). More scenarios will be extended in the future.

The unique features supported by TinySim are that the developers can not only run simulated IoT applications in the virtual scenarios but also interact with the simulated IoT devices using real-world devices. With the hybrid interaction scheme, the developers should examine the functionalities of IoT applications before large scale deployments, especially for the IoT applications involving the smart phones (e.g., the smart home applications that enable the interactions through users). Therefore, it is important to achieve real time and high fidelity virtual scenario functionalities, which are very different from existing simulators that only run simulated applications in simulated environments but not a hybrid simulation involving interactions between the simulation world and the real world. Providing such a powerful programming interface, however, faces several challenges. First, to ensure the correctness of the simulation, the sensor data simulated by TinySim should be reasonable when presented in Unity 3-D. For example, the illumination sensed by TinySim should be a small value when the light sensor is placed under the shadow area of Unity 3-D. Second, due to the different execution speeds between Unity 3-D and TinySim, we shall carefully design an event synchronization approach to ensure the correctness and reduce the simulation overhead.

### A. Environment Emulation

To ensure that the sensor data is reasonable at both TinySim and Unity 3-D, We divide the sources of sensing data into three categories.

*Sensing Data From Unity 3-D:* The sensor data is actually maintained in Unity 3-D. TinySim directly requests the sensor data from Unity 3-D engine.

Before the data is returned from Unity 3-D, the execution of TinySim is blocked.

*Sensing Data From the Existing Model:* TinySim takes the data distribution model from existing works. And, TinySim will periodically inform Unity 3-D to update the data presentation when the sensor data is changed. For example, there is an existing model for PM 2.5 [37], and it takes parameters related to the environments, e.g., the wind speed, the wind direction, and the temperature.

*Sensing Data From the Trained Model:* It is similar to the second source, the difference is that the sensor data distribution model is trained from trace using a fitting algorithm, e.g., closest-fit pattern matching [38].

### B. Interactivity Problems Between Unity 3-D and TinySim

The execution speed of Unity 3-D engine and TinySim are different. The event inconsistency may occur when developers change the environment of Unity 3-D, e.g., TinySim turned on the lamp at the simulated time $T_0$ but Unity 3-D may reach at $T_0$ latter and allow the developer to turn it off manually. To ensure the simulation correctness, an efficient event synchronization approach is important.

*Drawbacks of Existing Approaches:* In existing parallel simulators [8], [29], threads stop at known functions (e.g., send, receive, and channel sampling) to synchronize events. For example, in CSMA mechanism, the thread simulating the transmitter stops at the channel sampling function at the time $T_0$ and waits for other threads simulating neighbors to arrive at $T_0$ to safely judge whether the channel is free. However, in the scenario of synchronizing Unity 3-D and TinySim, there are no known functions to be waited for. Because the unsynchronized events are mostly caused by developers' interactions in Unity 3-D, which are unpredictable. There are also approaches [6], [7] performing synchronization per given interval. However, the proper interval is relatively hard to be determined. For example, a longer synchronization period will cause a larger rollback overhead if the event inconsistencies exist. On the other hand, a shorter synchronization period leads to significant overhead of message exchanging frequently.

To ensure the simulation's correctness while maintaining efficiency, we propose a speculative approach to synchronize the events. TinySim is enabled to run always faster than Unity 3-D following the default scenario settings and interaction logics. When interactions from Unity 3-D violates the execution logic speculatively processed by TinySim, we let TinySim selectively roll back the events based on a dependency graph, i.e., the events that violate the data value or the control flow caused by the Unity 3-D interactions. The speed and the simulation accuracy are traded at a good balance.

Two natural questions arrive: 1) How to build the dependency graph? and 2) How to reduce the rollback overhead?

### C. Speculative Event Synchronization: Dependency Graph Construction

TinySim adopts the following approach to construct the dependency graph (See Fig. 5).

1) In a dependency graph, the vertex denotes the events, e.g., `TL_Fan.open()`.
2) The edge denotes that there are relationships between the vertices in terms of the data flow or the control flow. Given the direction from the event $A$ to the event $B$, for the data flow, it means that after reading/writing the memory $M$ by the event $A$, the event $B$ will write/read the same memory $M$ later. It is the common resource race definition in parallel execution, i.e., read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) [39]. As for the control flow, it indicates that there is a conditional jump or direct jump from the event $A$ to the event $B$. We utilize the control flow analysis tool to construct the edges between the vertices, e.g., taint analysis [40].
3) The dependency graph is constructed at the compile time, therefore the construction procedure will not incur overhead when TinySim is running.

Fig. 5(b) shows an example dependency graph extracted from the application code shown in Fig. 5(a). It is a smart

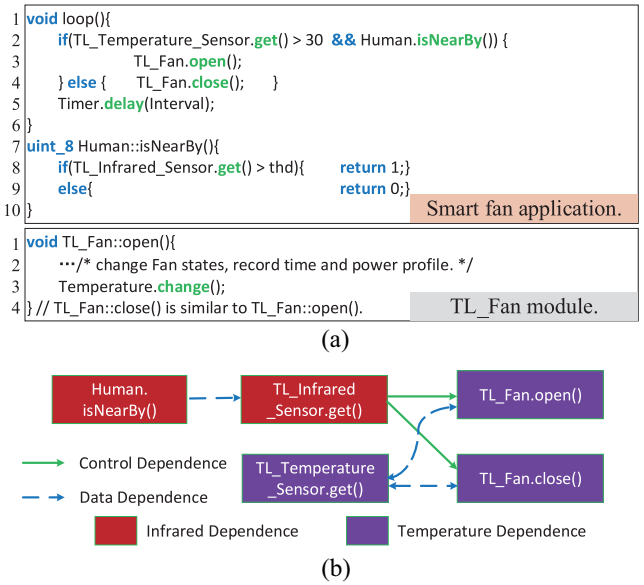


Fig. 5. Illustrative example of constructing a dependency graph from the application code. Vertex: The event. Edge: Data or control dependence. (a) Code snippets of the smart fan application and $TL_{Fan}$ module. (b) Dependency graph analyzed from the code (a).

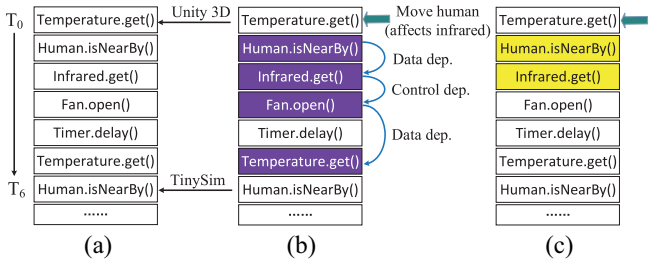


Fig. 6. Roll back execution example. (a) TinySim runs faster than Unity 3-D. (b) Changing the human's position, resulting in an infrared change. (c) Only keeps the events that will change the original flow.

fan application that lets the fan equipped with a temperature sensor and an infrared sensor. The smart fan is only turned on when people are detected nearby and the temperatures are greater than 30 [shown in line 2 of Fig. 5(a)]; otherwise, the smart fan is shutdown. The people is detected when the reading of the infrared sensor is greater than the threshold. The detection function performs periodically with the `Interval`.

As shown in Fig. 5(b), the vertex labeled with the same color means they access the same data object. For example, `TL_Temperature_Sensor.get()` reads the global data structure `temperature` to return the sensing data, while `TL_Fan.open()` modifies the value in `temperature` since it will cool down the temperature. The solid line denotes the control flow dependency, while the dotted line indicates the data flow dependency. For example, the line from `TL_Temperature_Sensor.get()` to `TL_Fan.open()` is generated since there is a conditional jump between them [shown in lines 2 and 3 of Fig. 5(a)].

### *D. Speculative Event Synchronization: Rollback Overhead Reduction*

Based on the dependency graph, TinySim reduces the rollback overhead following the approach below (see Fig. 6).

1) When there is an interaction from Unity 3-D, TinySim analyzes which data objects are affected by the interaction. For example, when developers turn on the fan in Unity 3-D, it leads to the changes in the `temperature` objects. Note that the relationship between the interaction and the affected data objects can be setup off-line.
2) TinySim looks in the event queue for the event that is the closest to the current trigger point in the simulation time and operates on the same data object. The event is set as the starting event in the dependency graph.
3) TinySim begins from the starting event and labels the affected events based on the dependency graph. The above procedures are performed iteratively until the simulated time in TinySim.
4) To further reduce the rollback overhead, TinySim only keeps the events that will cause a flow change following the directions in the dependency graph. The changes can be the data change or the control flow considering the edge type. For example, the readings of the temperature sensor will do change because the fan is turned on.

For the example shown in Fig. 6(a), the unity 3-D executes at the simulated time $T_0$ and TinySim is at $T_6$. At this point, suppose that the people is already near the fan and the temperature is still larger than 30, therefore the fan has been turned on.

Assume that the developer changes the human's position in the Unity 3-D but still near the fan [see Fig. 6(b)]. Therefore, the interaction arrives at the simulated time $T_0$ and it will affect the value of `infrared` object. The first event of `Temperature.get()` is thus skipped since it is only related to the `temperature` object. And, the next event in the queue `Human.isNearBy()` is labeled and set as the starting event in the dependency graph. Then, following the direction of the dependency graph, three events are labeled: 1) `Infrared.get()`; 2) `Fan.open()`; and 3) `Temperature.get()`. To further reduce the rollback overhead, only two events are kept: 1) `Human.isNearBy()` and 2) `Infrared.get()` [see Fig. 6(c)]. Because the results of the conditional jump does not change from the `Infrared.get()`. The analysis is done and finally, TinySim only needs to roll back and re-execute the events of `Human.isNearBy()` and `Infrared.get()` to update the data object `infrared`.

The above approach has the following features. First, compared with existing parallel simulators which roll back all events to maintain the simulation correctness, TinySim can fall back only a small number of affected events based on the dependency graph, which greatly improves the efficiency of virtual scenario development. Second, the proposed approach provides an on-demand synchronization scheme. The synchronization is triggered only when the violated execution results caused by Unity 3-D interactions are detected. Compared with the existing periodic event synchronization scheme [6], [7], the efficiency is greatly improved. Third, the approach deliberately makes Unity 3-D run slower than TinySim, which reduces the frame drop rate of Unity 3-D, and thus provides a better development experience.

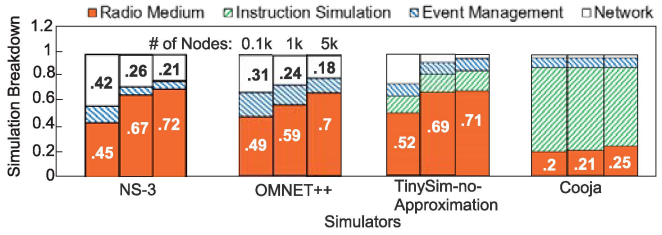


Fig. 7. Simulation breakdown in terms of components. The radio medium module occupies the most time portion (e.g., over 70%).

## VII. Scalability

High scalability is important for simulators to handle large networks with thousands of IoT devices. One of the main factors that affect the simulation scalability is the number of time-consuming simulation events, which is increased with the number of simulated devices. TinySim utilizes a machine learning-based approach to reduce the number of simulation events. Moreover, TinySim can further speed up the simulation by distributing simulation tasks to multiple hosts.

### A. Simulation Speed Bottleneck Analysis

*The Bottleneck of the Existing Simulators:* To understand the bottlenecks that affect the large-scale simulation, we conduct experiments to obtain a time breakdown of three typical existing simulators (i.e., Cooja, OMNET++, and NS-3) and TinySim in terms of different simulation components. Three existing simulators simulate an 802.15.4 multihop network to collect sensor data. As the number of simulated devices rises, the number of devices per hop gradually increases. TinySim simulates an NB-IoT network. As the number of nodes devices rises, the number of devices managed by the base station increases.

Fig. 7 presents the top four modules that spend the simulation time: 1) radio medium; 2) instruction simulation; 3) discrete event management; and 4) network simulation.

1) *Radio Medium:* It simulates how devices communicate over the channel via different radio technologies. Typical functions like channel contention, RSSI update, etc. This module lies in most simulators.
2) *Network Simulation Module:* It simulates the protocol behaviors, e.g., packet construction, transmission state machine, etc. It lies in many network simulators, e.g., Cooja with the network level mode, NS-3, and OMNET++.
3) *Instruction Simulation Module:* It captures device-dependent behaviors in the machine code level. It can capture not only the protocols behaviors but also the platform characteristics. It achieves high fidelity (e.g., Cooja [30]).
4) *Discrete Event Management:* It is responsible for managing the event queue, e.g., inserting and popping the events.

Fig. 7 shows that with the increase in the number of simulation nodes, the proportion of simulating the radio medium increases in the four simulators. Note that the time proportion of the radio medium is higher than 45% for TinySim, NS-3, and OMNET++ even when simulating 100 devices, indicating that this is a common module resulting in large execution overhead. For Cooja with the instruction level simulation, although the main overhead falls into the instruction simulation module, the overhead of the radio medium also increases to 25% when simulating 5000 devices.

To further understand why the radio medium module consumes so much time, we carefully analyze the source codes of existing three simulators, and find that there are many time consuming operations on the radio-related data structure, e.g., channel condition detection in CSMA mechanism, and collision behavior simulation. To simulate the collision behavior or the channel condition detection, the simulator needs to wait for all devices that may interfere with each other to execute to a certain time before they can determine whether the transmission is conflicted or not. Before to do this, the transmission behaviors were suspended and stored in a queue (e.g., collisionQ in TinySim) as shown in Fig. 8. As the scale of the simulation increases, the overhead of enumeration and selection from the collisionQ becomes larger. Therefore, the simulation of the radio medium takes more and more time.

Similarly, without the simulation approximation, TinySim also suffers from high overhead in the radio medium module. Fig. 8 presents typical functionalities to simulate the radio medium, and existing simulators, such as Cooja, OMNET++, and NS-3 have similar functions. To increase the scalability, we propose a machine learning-based approach to reduce the above time-consuming operations. Specifically, by learning the transmission behavior of every transmission link (i.e., the pair of the transmitter and the receiver), the number of transmissions (including retransmissions) and the related statistics (e.g., the delay and the power consumption) for one transmission can be directly obtained. As shown in Fig. 8, when a transmission arrives, the approximation module can directly provide the detailed statistics. Note that the approximation module can also be utilized to estimate whether the channel is busy for the CSMA mechanism.

### B. Scalable Simulation

*Machine Learning-Based Simulation Approximation:* However, to put this idea into a functional system faces several challenges: 1) Which features should we extract? 2) Which algorithm should be chosen? and 3) How to tradeoff the speed and the accuracy?

*Feature Selection:* The model is trained based on the transmission link; therefore, the features should be related to the link. For example, the PRR of this link, the number of nodes that do not synchronize with the receiver (e.g., causing potential collisions), the number of nodes that synchronize with the receiver, etc. The above features are selected as the most important ones using the correlation-based feature selection approach [41], which has been widely used in machine learning. It can evaluate the worth of a subset of features by considering the individual predictive ability of each feature along with the degree of redundancy between them.

*The Combination of the Black Box and the White-Box Estimation:* One of the important features of TinySim is

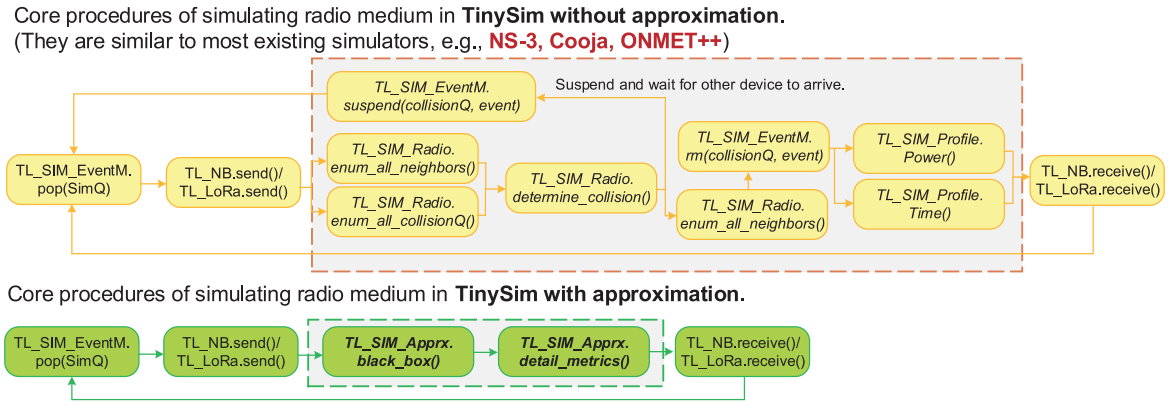


Fig. 8. Core procedures of simulating radio medium with and without the approximation approach.

that developers can obtain the application's statistics (e.g., the delay and the power consumption) to further optimize their algorithms. For example, the energy consumption of the NB-IoT module in terms of different states, including the transmission, the reception, the listen, and the sleep. Therefore, the output of the machine learning algorithm should be the statistics.

The existing work [42] also utilizes a machine learning-based approach to speed up the simulation. However, they can only produce two outputs that are very insufficient for TinySim, e.g., at least eight outputs for the energy consumption and the delay of the NB-IoT module. To deal with this challenge, a straightforward approach is to train dedicated models for different statistics separately. However, it will incur large computation overhead and slow down the simulation speed.

On the contrary, we propose an approach by combining a black-box method and a white-box method. With the black-box method, we obtained high-level statistics, e.g., the number of retransmissions $K$ and the delay $D$ (successfully receiving packets or dropping packets). Then, based on the high-level statistics, we derive a concrete model to estimate the detailed statistics, e.g., the delay and the power consumption of different states of the NB-IoT module.

*For the black-box method*, we choose long LSTMs [16] since it can produce two outputs: 1) the cell state $C$ and 2) the memory $h$. The two outputs are mapped to the number of retransmissions $K$ and the delay $D$ (successfully receiving packets or dropping packets), respectively. However, the values of these two outputs range from $-1$ to 1. We need to normalize the number of retransmissions and the delay to the range of $-1$ to 1 when training the model. Specifically, given the maximum number of retransmissions $K_{\max}$, and the maximum delay $D_{\max}$ for waiting replies from the sender, the normalized retransmissions $K_{\text{nor}}$ and delays $D_{\text{nor}}$ are derived as

$$K_{\text{nor}} = -1 + \frac{2 * (K+1)}{K_{\max}}$$
$$D_{\text{nor}} = -1 + \frac{2 * (D+1)}{D_{\max}}. \quad (1)$$

LSTMs is a special kind of recurrent neural network (RNN), which is composed of a series of neural nodes that take input features, and perform a series of matrix multiplications on them based on each node's activation function.

**Algorithm 1:** Derive Detailed Timing Statistics of Hardware States at the Sender Side (White-Box Approach)

**Input** : Whole delay D, transmission number K, data packet length L, ack packet length A, nack packet legnth N, ack time out AT, nack time out NT

**Output**: Time for transmission $t_{tx}$, time for reception $t_{rx}$, time for listening $t_{listen}$, time for idle $t_{idle}$

1 $t_{tx}$ = packet_on_air(L)*K;
2 **if** *Retransmission mechanism is ACK-based* **then**
3 $t_{rx}$ = packet_on_air(A);
4 $t_{listen}$ = (K-1)*AT;
5 **else if** *Retransmission mechanism is NACK-based* **then**
6 $t_{rx}$ = packet_on_air(N)*(K-1);
7 $t_{listen}$ = NT;
8 **else if** *No retransmission mechanism* **then**
9 /*$t_{rx}$ and $t_{listen}$ are zero since there are no reliability guarantee.*/
10 $t_{idle}$ = D - $t_{rx}$ - $t_{tx}$ - $t_{listen}$;

*For the white-box method*, we split the existing transmission mechanism as three types: 1) the ACK-based (NB-IoT HARQ); 2) the NACK-based (reliable multicast transmission); and 3) no retransmissions (LoRa unconfirmed). Algorithm 1 shows how to estimate the four important timing statistics at the sender side. The first two inputs (Whole delay D, transmission number K) are estimated by the black box, while other inputs can be obtained from the settings in the communication module. For the ACK-based retransmission mechanism, we assume that the sender performs retransmissions when the ACK waiting time (AT) is passed. Therefore, the time for the listening state is $(K-1)$*AT. The only time for the receiving state is receiving the ACK packet, i.e., the packet on air time of the ACK packet (as shown in lines 2–4 in Algorithm 1). As for the NACK-based retransmission mechanism, the sender only performs retransmissions when receives a NACK packet. The time for receiving state, including receiving all NACK packets (as shown in lines 5–7 in Algorithm 1). For the no retransmission scenario, the sender only transmits the data packet and then switch other states.

For the timing statistics of the receiver, the $t^r_{rx}$ and $t^r_{tx}$ are the corresponding timing of $t_{tx}$ and $t_{rx}$ of the sender. The rest of the time are regarded as the listening state $t^r_{\text{listen}} = D - t^r_{rx} - t^r_{tx}$.

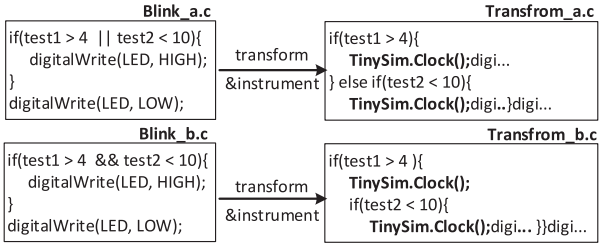


Fig. 9. Example of performing beyond source-line level instrumentation on a blink application of Arduino UNO.

To get the energy consumption, we only need to multiply the estimated elapsed time of each state with the current accordingly.

*Tradeoff Between the Speed and the Accuracy:* Reducing transmission and collision-determination events of all nodes using machine learning can significantly speed up the simulation, however, resulting in the degradation of the simulation accuracy. Instead of fixing the fraction of nodes optimized with machine learning, TinySim gives developers the ability to adjust the simulation speed on demand. Developers can determine the fraction of nodes using the approximation with the provided parameter. TinySim will finally output the estimated accuracy based on the given parameter.

*Discussion of the Approximation-Based Approach:* The above approach has the following benefits. First, it can not only greatly reduce the simulation overhead but also provide detailed statistics for developers to optimize their IoT applications. Second, the approximation fraction can be adjusted such that developers can control the simulation speed and the fidelity on demand. It is flexible for the developers. Third, through the detailed analysis of the common time-consuming modules of the existing simulators, the approximation approach may be used as a stand-alone module to improve the scalability of the existing discrete event-based simulators.

*Distributed Simulation:* The speed of TinySim can be further improved through the distributed simulation. As stated in many previous works [8], [29], [43], when executing discrete events in distributed hosts, the major overhead affecting scalability comes from the interactions among hosts, e.g., packet transmissions between nodes running on different hosts. To minimize synchronization overhead among hosts while balancing the load (e.g., the number of discrete events) over all hosts, we utilize a parallel graph partition tool Zoltan [44] to solve the task distribution problem. It uses multilevel heuristic approaches to achieve effective graph repartition [44].

*Discussion of the Distributed Simulation:* With the distributed simulation, TinySim can further improve the simulation speed, resulting in a higher scalability. We leave the distributed simulation as an optional choice for the developers that have more computation resources. In this way, the architecture of TinySim is more flexible.

## VIII. Implementation

TinySim utilizes TinyLink as the IoT application implementation language. TinyLink language is a hardware-independent programming language that enables the rapid development of IoT applications. By adopting a top-down developing process, TinyLink abstracts the IoT hardware in a way such that developers can focus on the application logic without experiences in embedded systems. By analyzing the application logic and requirements from developers' code, TinyLink language compiler can automatically select the most appropriate hardware components as well as their connections. In TinyLink language, hardware components can be divided into mainboards and peripherals (e.g., sensors and communication modules).

For the IoT devices, we carefully split into the multiple modules, e.g., the mainboard, sensor and display, and communication module. These modules play import role in simulating the fine-grained timing behaviors of the IoT devices. In the following, we provide details about how these modules are implemented.

### A. Implementation of Simulation Modules

*1) Mainboard Simulation:*

1) *Capturing the Timing Behaviors:* Mainboard consists of key electronic components of a system, e.g., MCU and memory. The main behavior of the mainboard is executing the IoT application by the MCU. Therefore, TinySim simulates the mainboard by capturing the MCU behaviors. To this end, we decompile the application using the corresponding cross compiler to obtain a clear mapping from the source codes to the MCU instructions at the source-line level. The elapsed time of applications is calculated based on the mapping between MCU instructions and execution time documented in datasheet. To capture the timing behavior beyond the source line level (e.g., the conditional operations), we propose to separate the conditional operations before performing instrumentations. For example, as shown in Fig. 9, the single line OR (or AND) operation is separated into two conditional operations. Then, by performing instrumentation on the transformed code (e.g., Transform_a.c), the MCU behaviors of both "test2 < 10" and "test1 > 4" are captured.
2) *Capturing the Power Consumption:* To capture the power consumption of mainboard, it is essential to track the amount of time the MCU spends in each power state (e.g., active and sleep). The sleep state of MCU can be inferred from certain functions which are usually a mixed-code of assembly and source code. For example, __asm("WFI") denotes that Cortex-M3 enters into the sleep mode and waits for interruptions [45]. Any interruptions can wake up the MCU and change the state of MCU back to the active mode, e.g., the clock interruption. Then, the MCU state can be tracked by instrumenting the power monitor code at the power state transition functions. When the MCU power states are tracked, the power consumption can be calculated by multiplying the previously captured timing behaviors of MCU instructions and the current of each state referred from the datasheet.

*2) Sensor and Display Module Simulation:* TinySim simulates the sensors by measuring the elapsed time and the

consumed power of the provided APIs (e.g., read the temperature of soil by the Soil Moisture sensor). Most peripherals cannot be measured directly due to its limited interface, we measure the elapsed time and the power consumption by connecting them into the mainboard and the results are obtained by subtracting the mainboard measurements.

*3) Communication Module Simulation:* Among the various communication protocols, we select to model one of the LPWAN protocols, NB-IoT, as an example because of its wide application scenarios and large-scale deployments. The timing behaviors of NB-IoT falls into two protocol layers: 1) PHY-layer and 2) MAC layer.

1) *Capturing the PHY-Layer Timing Behaviors:* TinySim simulates the communication behaviors at the bit level. After channel encoding (e.g., Turbo coding for uplink transmission and Tail Bitting Convolutional Coding for downlink transmission [4]) and rate matching, the NB-IoT frame is repeated to enhance the transmission range of NB-IoT. Then, the bits of the repeated frames are transmitted to the receiver with the error probability $p$. $p$ is related to the channel quality and the distance which can be characterized by a path loss model [46]. To simulate the collision scenario, TinySim invokes a collision event to determine whether these transmitted bits will be collided with other transmissions.
2) *Capturing the MAC-Layer Timing Behaviors:* The MAC layer behaviors include the back off timing and the packet on-air time. The backoff timing strictly follows 3GPP definition, e.g., when there is no response to the preamble, then devices will backoff a random time within [0, 960] ms [47]. The packet on-air time is related to the payload size, the allocated subcarriers and MCS. Then, based on the above three parameters, devices or the base station will pick a proper transport block size (TBS) from the TBS Table [48]. For example, given the TBS is 2536 bits, and the required resource unit (RU) is seven according to the TBS table. Suppose most subcarriers are allocated (e.g., 1 ms/RU), therefore, the packet on-air time is 7 ms.
3) *Capturing the Power Consumption:* To capture the power consumption of NB-IoT, we instrument at the key state transition functions, such as the send() and sleep().

For the communication modules that the source codes are not public yet (e.g., NB-IoT [4]), we propose a generic development framework that can help developers easily extend the new communication module into TinySim. The framework contains a series of basic radio states libraries, e.g., send(), recv(), and sleep(). Developers who are familiar with the protocols only need construct the state transition graph by writing simple SMC [17] codes. SMC is a state machine compiler for automatically generating state patterns based on a state machine description into a target object-oriented language.

### B. Programming Style

Currently, TinySim intends to provide an easy-to-use `setup()` and `loop` programming style for a wide variety of IoT platforms. It is easy to transplant the code to Arduino platforms since their programming styles are quite similar. We implement all the APIs of the Tiny language for the TinySim simulator and the Arduino IDE by using their own native APIs, respectively. Similarly, we do the same things for Raspberry Pi and BeagleBone platforms that run the Linux system. The difference is that we build a preprocessor for transforming the TinySim code into C/C++ code by injecting proper header files and announcements, rearranging `setup` and `loop` into the `main` function, etc. In the future, we will support event-driven programming like TinyOS [2] and Contiki [3] by utilizing the process-based execution methods like TOSThread [49] where we implement the `setup` and `loop`.

### C. Hardware Heterogeneity

Supporting various IoT platforms is quite challenging due to the great heterogeneity in processor, memory, and flash.

Since the IoT platforms possess various CPU/MCU/SoC, TinySim's compilers maintain a capability table that records the CPUs' capabilities like hardware timer, interrupt, power save mode, etc. TinySim will give developers a warning if the target IoT platform does not support capabilities written in the code during compilation. Then, TinySim tries to facilitate compilation by using additional libraries, e.g., using TinySim's software timer library for platforms like Raspberry Pi and BeagleBone that do not contain hardware timers. Otherwise, TinySim will throw an exception and terminate the compilation process. Moreover, TinySim maintains a mapping table that eliminates type differences, e.g., transforming a 4-bytes `int` to `long` on Arduino and `int` on Raspberry Pi. It also warns developers when detecting direct bitwise operations since some processors (e.g., Arduino UNO and Intel x86) use little-endian, and others (e.g., Raspberry Pi, AVR32) use big-endian, which may cause different results.

Besides CPU/MCU, memory and flash also raise several issues for TinySim. The size of memory space varies drastically on different IoT platforms, e.g., 2-KB RAM on Arduino UNO and 1-GB RAM on Raspberry Pi 2. Besides warning and avoiding uploading programs that have oversized memory space, TinySim adopts several techniques to optimize the memory usage for memory-constrained platforms, e.g., using built-in function `F()` on Arduino platforms that can avoid loading strings to memory space for the presence of lots of debugging logs. Similarly, TinySim also adopts techniques to save flash space for flash-constrained platforms, e.g., storing strings and block data to EEPROM on Arduino platforms, removing unused libraries and functions, etc.

To let TinySim run on x86 platforms, we have reimplemented hardware related APIs of Tiny language with simulated hardware modules. For example, the API `TL_Soil_Moisture.read()` is reimplemented with a simulated sensor module that can periodically generate moisture data from a real-world trace. We classify the hardware modules into three categories (i.e., mainboard, sensor and display module, and communication module) and provide details of how these hardware can be simulated to be executed at x86 platforms.

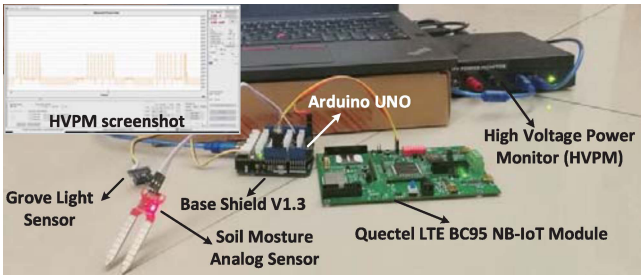


Fig. 10. Monitoring power consumption using HVPM.

## IX. Evaluation

### A. Experiment Setup

*Hardware:* We use two real IoT applications to evaluate the fidelity of TinySim. One is the smart flowerspot application. To evaluate the downlink of NB-IoT module [50], we add a MQTT message handler to respond to the sensed data. The assembled smart flowerspot device is shown in Fig. 10. We replace the ESP8266 ESP01 WiFi module [51] in [52] with the BC95 NB-IoT module [50]. Another application is the voice controlled LED lamp which is the same with [52]. To accurately measure the power consumption, we use high-voltage power monitor (HVPM) of Monsoon[3]).

HVPM supports the main channel output voltage range of 0.8–13.5 V and up to 6A continuous current. Fig. 10 presents how HVPM can measure the power consumption of the smart flowerspot device. The voice control application and other single hardware components can be measured similarly. To run TinySim, the master machine is a PC with 3.2-GHz CPU, eight processors, 12G RAM. There are at most eight slave machines with 2.5 GHz, four processors, 8G RAM each.

*Macro Benchmark:* We give a set of actions for the two IoT applications (i.e., the smart flowerspot and the voice controlled LED). For example, uploading and requiring soil data for the smart flowerspot using NB-IoT; performing voice control to the LED lamp. The macro benchmarks are: overall power consumption and delay in given actions.

*Micro Benchmark:* The micro benchmarks are different in terms of the hardware component (e.g., mainboard, radio hardware, sensor, and controller). 1) *For the mainboards*, we validate the MCU idle and MCU active in terms of the power consumption and the elapsed time. We run the delay function and power save function for 10 s to present the active and idle states of MCU. Three popular mainboards are evaluated, i.e., Arduino UNO (UNO for short), Raspberry Pi 2 (RP2 for short), and BeagleBone Black (BBB for short). 2) *For the sensors and displayers* we validate all of the provided API from them with regard to the power consumption and the elapsed time. Due to the space limit we only present the six frequently used micro benchmarks according to [52]. Because most sensors and controllers do not provide enough interfaces for power consumption measurement, we thus plug them with the mainboard and the final measurement results are subtracted with the overhead from the mainboard. 3) *For the communication modules*, we validate three important states, (e.g., send, receive, and sleep) for BC95 NB-IoT module [50] and Dragino LoRa Shield [53].

[3]"HVPM," https://www.msoon.com.

TABLE II
Macro Benchmark Performance

| Benchmark | Power (mJ) | | | Delay (s) | | |
|---|---|---|---|---|---|---|
| | Sim. | Meas. | Err | Sim. | Meas. | Err |
| Upload | 902.95 | 862.75 | 4.66% | 0.72 | 0.691 | 4.53% |
| Require (SNR=5dB) | 1195.80 | 1239.65 | -3.54% | 1.07 | 1.1 | -2.51% |
| Require (SNR=0dB) | 1377.29 | 1357.68 | 1.44% | 1.51 | 1.45 | 4.14% |
| Require (SNR=-2dB) | 1399.13 | 1475.67 | -5.19% | 1.69 | 1.75 | -3.43% |
| Voice control | 19000.00 | 19983.23 | -4.92% | 2.00 | 2.1 | -4.76% |

TABLE III
Micro Benchmark Performance of Mainboard

| Benchmark | Power (mJ) | | | Time (s) | | |
|---|---|---|---|---|---|---|
| | Sim. | Meas. | Err | Sim. | Meas. | Err |
| MCU active (UNO) | 30882.24 | 30893.96 | -0.04% | 12 | 12 | 0% |
| MCU idle (UNO) | 560 | 558 | 0.36% | 12 | 12 | 0% |
| MCU active (RP2) | 167311.4 | 167333.2 | -0.01% | 12 | 12 | 0% |
| MCU idle (RP2) | 14232.1 | 13200 | 7.82% | 12 | 12 | 0% |
| MCU active (BBB) | 27995.01 | 27601.23 | 1.43% | 12 | 12 | 0% |
| MCU idle (BBB) | 12612.5 | 12613.123 | 0.00% | 12 | 12 | 0% |

*Case Study:* We use two case studies to illustrate the attractive features of TinySim, e.g., interactive programming and expanding to the online learning platform. The details are shown in Section IX-G.

In the remainder of the section, we will evaluate the performance of TinySim in terms of the three key requirements stated in the beginning of this article, that is, high fidelity, (Section IX-B) high scalability (Sections IX-C and IX-D ) and high interactivity (Sections IX-E and IX-F).

### B. Fidelity: Simulated Hardware

Table II shows the overall accuracy of profiling power consumption and the action delay with regard to the specific actions in two real IoT applications. The experiment of each action is repeated ten times and the results are averaged. Results show that TinySim can accurately simulate the power consumption and the action delay with low average error rate (e.g., 3.8% for the power consumption and 3.95% for the action delay). The delay simulation error mainly stems from the path loss simulation. In reality, the communication conditions are complicated especially in long-range transmissions. For example, the unpredictable obstacles and the people in the movement. However, TinySim can still achieve relatively high accuracy in a certain scenario, e.g., at the mid-night, therefore, this error is acceptable with lower than 4.8%.

In addition to the overall accuracy, we also evaluate the accuracy of simulating three mainboards, (Arduino UNO, Raspberry Pi 2, and BeagleBone Black), six frequently used peripherals (Grove light, humidity sensor, temperature sensor, microphone, LED Lamp, and display screen) and two communication protocols (NB-IoT and LoRaWAN). The simulation errors are at most 9.52% and the detailed results are eliminated due to space limits.

Tables III and IV show the micro-benchmark results of evaluating hardware components. For all of the three mainboards, TinySim achieves 100% time estimation accuracy, because the

TABLE IV
MICRO BENCHMARK PERFORMANCE OF PERIPHERAL

| Benchmark | Power (mJ) | | | Time (s) | | |
|---|---|---|---|---|---|---|
| | Sim. | Meas. | Err | Sim. | Meas. | Err |
| Send (NB-IoT) | 1304.20 | 1267.50 | 2.90% | 1.71 | 1.69 | 1.18% |
| Receive (NB-IoT) | 303.05 | 307.80 | -1.54% | 1.7 | 1.71 | -0.58% |
| Sleep (NB-IoT) | 0.0176 | 0.0189 | -6.64% | 1.9 | 2.1 | -9.52% |
| Send(LoRa) | 562.93 | 608.40 | -7.47% | 1.7 | 1.69 | -0.58% |
| Receive(LoRa) | 63.34 | 61.56 | 2.90% | 1.6 | 1.71 | -3.64% |
| Sleep(LoRa) | 0.0120 | 0.0126 | -4.77% | 2.1 | 2.1 | 1.21% |
| Light On (Grove) | 75.20 | 76.18 | -1.28% | 5 | 5 | 0.00% |
| Hum. Read (Soil) | 79.10 | 77.57 | 1.98% | 5 | 5 | 0.00% |
| Temp. Read (Soil) | 27.00 | 27.56 | -2.04% | 5 | 5 | 0.00% |
| Microphone Read | 2050.62 | 2250.62 | -8.89% | 5 | 5 | 0.00% |
| LED Lamp On | 45510.10 | 45010.10 | 1.11% | 5 | 5 | 0.00% |
| Display On (LCD) | 1590.30 | 1510.30 | 5.30% | 5 | 5 | 0.00% |

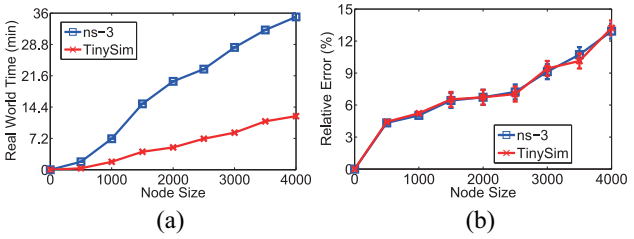


Fig. 11. Simulation speed and accuracy comparisons with existing simulator (Ten simulation minutes). (a) Simulation speed. (b) Simulation accuracy.

programs used for micro-benchmarks are relatively simple and almost all of the instructions can be mapped accurately. The power consumption simulation error is as lower as 1% or less. The subtle errors may due to the current jitters of mainboards.

For all of the evaluated peripherals, TinySim achieves the simulation error as small as 1.655% and 2.78% on average in terms of profiling power and time, respectively. For the five states of the BC95 NB-IoT module, TinySim achieves low simulation error at 2.6% and 2.54% on average for profiling power and time. The power and time simulation error steams from the MCU behaviors, which TinySim cannot directly capture, on BC95 board. The error is relatively low especially for the states of listen, sleep, and idle (e.g., lower than 1.08%). For the state send and receive, the simulation error raises a little (e.g., about 6.09% on average) due to the unstable wireless transmission.

### C. Scalability: Comparison With Existing Simulator

We compare TinySim with the most related simulator ns-3 that is with the NB-IoT module [54]. We disable the interactions from Unity 3-D. We simulate the smart flowerpot application that samples the humidity periodically to trigger the water pouring operation. The sensor data is sent to the cloud via the NB-IoT radio.

We evaluate the impact of the number of simulated nodes on the speed and the accuracy. Fig. 11 show that TinySim can not only achieve slightly better simulation accuracy than ns-3 (e.g., 2.3% smaller relative error on average) because of the detailed simulation of the hardware and the communication behaviors, but also a faster simulation speed (about 3× speed improvements when simulating 4000 nodes) with the simulation approximation and enabled distributed simulation.

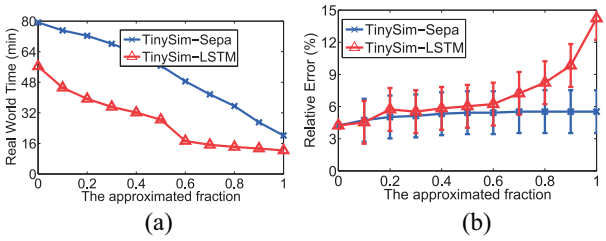


Fig. 12. Impact of the fraction of the approximation simulation (Ten simulation minutes, 4000 nodes). (a) Simulation speed. (b) Simulation accuracy.

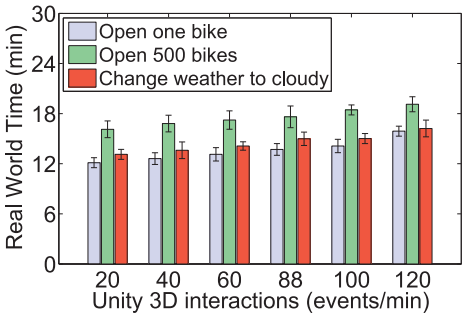


Fig. 13. Impact of the interaction frequency of Unity 3-D on the simulation speed (Ten simulation minutes, 4000 nodes).

### D. Scalability: The Approximation-Based Simulation

TinySim provides a parameter to adjust the fraction of using approximation-based approach to speed up the simulation. We let TinySim simulates a shared bike application for 10 min and repeat the experiments 30 times. We simulate 4000 nodes and utilize eight machines to perform distributed simulations. We replace different machine learning algorithms in TinySim to compare the approximation performance, e.g., LSTM [16] (TinySim-LSTM), train separate models for different metrics (TinySim-Sepa).

Fig. 12 presents the impact of changing the approximated fraction from 0 to 1 on the simulation speed and the simulation accuracy. Results show that TinySim-Sepa achieves the highest simulation accuracy since it trains dedicated models separately for different metrics (the relative error is smaller than 5%), however, its simulation speed is far too slow than TinySim-LSTM, specifically 47.6% slower. Note that the relative error of TinySim-LSTM is only 1.8% lower than TinySim-Sepa, indicating the effectiveness of combining the black-box (LSTM) and the white-box approach.

### E. Interactivity: Impact of Interactions From Unity 3-D

Unity 3-D can provide user-friendly development experience, however, it may result in the degradation of simulation speed due to the roll back of TinySim. We simulate a shared bike application to evaluate the impact of developer interaction frequency (the number of interactions per minute) and event types on the simulation speed. In this application, each shared bike is equipped with a smart lock, a GPS, an NB-IoT module for receiving unlocking events and a solar cell. The smart

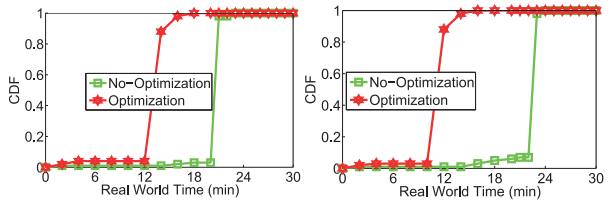


Fig. 14. Benefits of using the dependency graph to optimize the roll back overhead (Ten simulation minutes, 4000 nodes), two types of events are both 60 events/min. (a) Open one bike. (b) Open 500 bikes.

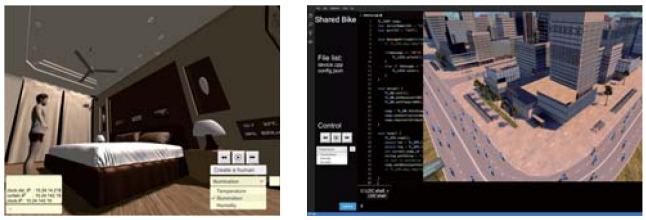


Fig. 15. Case studies. (a) Case 1: A smart home application on local TinySim. developers can interactively debug IoT applications, (b) Case 2: A shared bike application on online TinySim. TinySim is integrated with a online IDE. (a) Case 1: Local simulator. (b) Case 2: Online simulator.

lock will be unlocked only when receiving a message through the NB-IoT module. The GPS module will periodically upload location information to the cloud, and its data only changes when the smart lock is opened for users to ride. The solar cell only performs charging when the weather is sunny. We simulate three interaction events supported by Unity3-D and TinySim: 1) open the smart lock of one bike, and it will change the data of the GPS module; 2) generate 500 smart lock opening messages, and it will trigger the NB-IoT module to receive messages from the cloud, and the GPS data will be changed too; and 3) change the weather to the cloudy, and it only impacts the behaviors of the solar cell of all shared bikes.

Fig. 13 shows the impact of developer interaction frequency (the number of interactions per minute) and event types on the simulation speed. Results show that for the interactions of opening one lock and changing the weather, the simulation speed is slightly decreased (i.e., 17.2% on average) when the interaction frequency changes within 20 and 60 msg/min. Since both two interactions only effect few events to be rolled back. Note that this range is the common case when developers debug. It reveals that the partial roll back strategy based on the dependency graph is effective. On the other hand, when opening more shared bikes, i.e., 500 smart locks, the simulation speed will be lowered down more since the amount of events that can be optimized is reduced.

### F. Interactivity: Roll Back Overhead Optimization

Fig. 14 presents the effectiveness of using dependence graph-based roll back optimization. Results show that for both interaction types, the roll back optimization can improve the simulation speed significantly by 38.4% on average, comparing with the approach of rolling back all events. Specifically, for the interaction of opening one bike, the optimization approach has the best improvements (45.2%), since with the dependency graph only a few events should be re-executed. The roll back is therefore reduced significantly.

### G. Case Study

*Case 1 (Interactive Programming and Debugging):* Fig. 15(a) presents how to interactively program and debug a typical smart home application with TinySim. Developers can run TinySim locally and upload the application code to the corresponding IoT devices, e.g., curtain and fan. There are several attractive features help developers to pre-examine the behaviors of IoT applications before real deployments.

*Actively Change the Simulation Environment:* Developers can actively inject events to change the environment to verify the functionality of IoT applications. This is achieved by accepting interactions from the Unity 3-D modules, and then, the events generated from Unity 3-D are delivered to the TinySim. Fig. 15(a) shows the example that developers can lower down the illumination intensity to verify whether the smart lamp can detect the change of illumination and turn on the light.

*Inject a Randomly Walking Human:* Developers can also create a human in the scenario to randomly trigger events. The trajectory of the human can be generated from TinySim, and is delivered to the Unity 3-D engine. The triggered events can then be predicted and the simulation efficiency is improved. Fig. 15(a) presents that a human is put in the indoor room. Developers can examine that whether the smart door will detect the human's approaching and open the door.

*Change the Simulation Speed to Examine the Complete Cycle:* Another attractive feature of TinySim is that developers do not need to run the simulator in a fixed speed. On the contrary, they can speed up TinySim when current events are not interested, and slow down TinySim when they need to focus on the behaviors of specific devices at the certain time. The speed is controlled by changing the number of threads in the thread-pool. Fig. 15(a) shows that TinySim supports multiple speed options.

*Watch Variable Values:* When developers find abnormal behaviors of IoT application, stopping the simulator and watching variable values will be a very useful function to help developers quickly lock the bugs. TinySim runs on x86 machine and variable values are captured through the gdb tool. Fig. 15(a) presents that the smart curtain behaves abnormally since it is not opened when the simulation time is at 10:00 A.M. TinySim is stopped and outputs the IP address of the smart curtain and destination IP address of the smart clock. Developers can find that the smart clock did not transmit the time value successfully to the smart curtain. The bug is that the destination IP address (should be the smart curtain) is set wrongly in the smart clock.

*Case 2 (Expand TinySim to the Online Programming System):* TinySim can also be connected to a remote programming system. With this property, developers can easily create IoT applications and and verify them without setting up a local development environment. Fig. 15(b) shows that TinySim is integrated into a remote programming system,

e.g., LinkLab [55]. Developers are programming and verifying the shared bike application. Application codes are directly written and uploaded into the simulated bike in the online IDE. Developers can speed up the simulation time to quickly examine how different hours of a day change the bike usage pattern. Developers can also verify that whether the shared bike will send alert signals when it goes to a wrong area, e.g., dropped into a lake. Changing the communication modules in the shared bike to compare which is with the smaller reaction delay, i.e., the delay of unlocking the shared bike, or which is with the longer lifetime.

## X. Conclusion and Future Work

In this article, we presented TinySim, an IoT simulator for providing entire support to IoT applications. TinySim satisfies the requirements of completeness, high fidelity, high scalability, and high interactivity. TinySim can simulate representative IoT applications, such as smart flowerpot and shared bicycles. We conducted extensive experiments to evaluate the performance of TinySim. Results show that TinySim achieves high simulation accuracy with lower than 9.52% error. TinySim improves the simulation speed around 3× comparing to the state-of-art approach. Nevertheless,further improvement is possible in two areas.

*Gaps in Simulating the Delay Between the Base Station and the Cloud:* Currently, to simulate the communication links between the base station and the cloud, TinySim utilizes the Internet connection between the PC and the cloud through the wired connection. Without the knowledge of the routing path and the topology of the core network, it is relatively complicated to accurately model the delay between the base station and the cloud [35]. It is nontrivial to accurately simulate the delay, especially when there are a large number of requests for accessing the devices (e.g., opening the shared bike at peak time). In fact, TinySim can still achieve relatively high simulation accuracy under specific scenarios (e.g., the traffic delay maybe predictable when transmission happens at mid-night). To provide accurate simulation in more scenarios, a promising approach is to open more parameters to the developer to determine the routing metrics and the network topology. In this way, most routing behaviors in the core networks can be simulated and the accuracy is thus improved.

*Enabling Direct Control From Users to the Simulated Device:* TinySim now supports the indirect control to the simulated devices, i.e., through the cloud. We choose this indirect design because of two reasons. First, it is an easy installation that connecting the IoT devices to the cloud. Second, TinySim provides the simulation of LPWAN which typically includes a cloud to forward the message from the users. And, most smartphones currently do not support direct access to LPWAN. However, it may incur large communication overhead when the network condition is bad. Therefore, as one of the possible future works, we would like to incorporate in more short-range communication technologies. With most short-range communication technologies are already available in smartphones, it is possible to control devices by accessing AP (e.g., WiFi) or directly accessing the device (e.g., BLE).

In summary, the future work includes two directions. First, extending TinySim with more communication protocols, e.g., WiFi. Providing simulations of the short range communication protocols can better express the IoT application behavior at the smart home. e.g., how are WiFi and Bluetooth interacted over the unlicensed band. Second, extending TinySim with core network simulation. In the current version, TinySim only provides the behavior simulation at the radio access network (RAN) of NB-IoT. The behaviors of the core part of NB-IoT are more complex and may also have impacts on the performance of NB-IoT client, e.g., how the NB-IoT clients adapt the modulation parameters and how the base station allocates resources for the clients. Providing more details about the above behaviors can improve the simulation fidelity.

## References


[1] "Newsroom: Gartner says 8.4 billion connected things will be used." Gartner. 2017. [Online]. Available: https://www.gartner.com/newsroom/id/3598917

[2] P. Levis et al., "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*. Heidelberg, Germany: Springer, 2005, pp. 115–148.

[3] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—A lightweight and flexible operating system for tiny networked sensors," in *Proc. IEEE Local Comput. Netw.*, 2004, pp. 455–462.

[4] "Narrowband IoT (NB-IoT)," 3GPP, Sophia Antipolis, France, document TSG69 RP151621, 2015. [Online]. Available: http://www.3gpp.org/ftp/tsg_ran/TSG_RAN/TSGR_69/Docs/RP-151621.zip

[5] "A low power, wide area (LPWA) networking protocol LoRaWAN," LoRa Alliance, San Ramon, CA, USA, White Paper, 2017. [Online]. Available: https://www.lora-alliance.org/

[6] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proc. IEEE/ACM IPSN*, 2005, pp. 477–482.

[7] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "EmStar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Trans. Sens. Netw.*, vol. 3, no. 3, p. 35, 2007.

[8] Z.-Y. Jin and R. Gupta, "Improving the speed and scalability of distributed simulations of sensor networks," in *Proc. IEEE/ACM IPSN*, 2009, pp. 169–180.

[9] N. Bak, B.-M. Chang, and K. Choi, "Smart block: A visual programming environment for SmartThings," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, 2018, pp. 32–37.

[10] V. Damjanovic-Behrendt and W. Behrendt, "An open source approach to the design and implementation of digital twins for smart manufacturing," *Int. J. Comput. Integr. Manuf.*, vol. 32, nos. 4–5, pp. 366–384, 2019.

[11] J. P. Dias, F. Couto, A. C. R. Paiva, and H. S. Ferreira, "A brief overview of existing tools for testing the Internet-of-Things," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, 2018, pp. 104–109.

[12] Q. Chen, F. Schmidt-Eisenlohr, D. Jiang, M. Torrent-Moreno, L. Delgrossi, and H. Hartenstein, "Overhaul of IEEE 802.11 modeling and simulation in ns-2," in *Proc. 10th ACM Symp. Model., Anal., Simulat. Wireless Mobile Syst.*, 2007, pp. 159–168.

[13] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM Demonstration*, vol. 14, no. 14, p. 527, 2008.

[14] A. K. Rathi and A. J. Santiago, "The new NETSIM simulation program," *Traffic Eng. Control*, vol. 31, no. 5, pp. 317–320, 1990.

[15] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*. Heidelberg, Germany: Springer, 2010, pp. 35–59.

[16] H. Sepp and S. Jürgen, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] D. Buehler, S. Whitaker, and J. Canaris, "Sequence invariant state machine compiler," in *Proc. IEEE 1st Great Lakes Symp. VLSI*, 1991, pp. 318–323.
[18] S. L. Kim, H. J. Suk, J. H. Kang, J. M. Jung, T. H. Laine, and J. Westlin, "Using unity 3D to facilitate mobile augmented reality game development," in *Proc. IEEE World Forum Internet Things (WF-IoT)*, 2014, pp. 21–26.
[19] "TinySim source code." 2019. [Online]. Available: https://github.com/TinySim/TinySim
[20] N. D. Patel, B. M. Mehtre, and R. Wankar, "Simulators, emulators, and test-beds for Internet of Things: A comparison," in *Proc. 3rd Int. Conf. I-SMAC (IoT Social, Mobile, Anal. Cloud) (I-SMAC)*, 2019, pp. 139–145.
[21] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.
[22] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *J. Syst. Archit.*, vol. 72, pp. 93–107, Jan. 2017.
[23] S. Claudio and F. Giancarlo, "A simulation-driven methodology for IoT data mining based on edge computing," *ACM Trans. Internet Technol.*, vol. 21, no. 2, pp. 1–22, 2021.
[24] V. Barbuto, C. Savaglio, M. Chen, and G. Fortino, "Disclosing edge intelligence: A systematic meta-survey," *Big Data Cogn. Comput.*, vol. 7, no. 1, p. 44, 2023.
[25] P. Levis, N. Lee, M. Welsh, and D. Culler, "ToSSIM: Accurate and scaleable simulation of entire TinyOS applications," in *Proc. ACM Sensys*, 2003, pp. 126–137.
[26] V. Shnayder, M. Hempstead, B. R. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proc. ACM Sensys*, 2004, pp. 188–200.
[27] O. Landsiedel, H. Alizai, and K. Wehrle, "When timing matters: Enabling time accurate and scalable simulation of sensor network applications," in *Proc. IEEE/ACM IPSN*, 2008, pp. 344–355.
[28] M. H. Alizai, Q. Raza, Y. Chandio, A. A. Syed, and T. M. Jadoon, "Simulating intermittently powered embedded networks," in *Proc. ACM EWSN*, 2016, pp. 35–40.
[29] H. Jiang, J. Zhai, S. K. Wahba, B. Mazumder, and J. Hallstrom, "Fast distributed simulation of sensor networks using optimistic synchronization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 2888–2898, Nov. 2014.
[30] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with COOJA," in *Proc. IEEE Local Comput. Netw.*, 2006, pp. 641–648.
[31] M. C. Bor, U. Roedig, T. Voigt, and J. M. Alonso, "Do LoRa low-power wide-area networks scale?" in *Proc. ACM Int. Conf. Model., Anal. Simulat. Wireless Mobile Syst.*, 2016, pp. 59–67.
[32] "Ali Yun cloud." 2023. [Online]. Available: http://www.aliyun.com
[33] "IBM Watson." 2023. [Online]. Available: https://www.ibm.com/watson/
[34] (LoRa Alliance, San Ramon, CA, USA). "Open source of a low power, wide area (LPWA) networking protocol LoRaWAN." 2017. [Online]. Available: https://github.com/Lora-net
[35] B. Nguyen et al., "Towards understanding TCP performance on LTE/EPC mobile networks," in *Proc. 4th Workshop Things Cellular Oper., Appl. Challenges*, 2014, pp. 41–46.
[36] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny Internet nodes," *IEEE Internet Comput.*, vol. 16, no. 2, pp. 62–67, Mar./Apr. 2012.
[37] I. Hinojosa-Baliño, O. Infante-Vázquez, and M. Vallejo, "Distribution of PM2.5 air pollution in Mexico city: Spatial analysis with land-use regression model," *Appl. Sci.*, vol. 9, no. 14, p. 2936, 2019.
[38] L. HyungJune, C. Alberto, and P. Levis, "Improving wireless simulation through noise modeling," in *Proc. IPSN*, 2007, pp. 1–10.
[39] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS Oper. Syst.*, vol. 37, no. 5, pp. 237–252, 2003.
[40] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 1–15.
[41] M. A. Hall, "Correlation-based feature subset selection for machine learning," Ph.D. dissertation, Dept. Comput. Sci., Univ. Waikato, Hamilton, New Zealand, 1998.
[42] W. K. Charles, S. João, K. N. Kelvin, L. Vincent, and H. U. Lyle, "Fast network simulation through approximation or: How blind men can describe elephants," in *Proc. ACM HotNets*, 2018, pp. 141–147.
[43] Y. Wen, R. Wolski, and G. Moore, "DiSenS: Scalable distributed sensor network simulation," in *Proc. 12th ACM SIGPLAN Symp. Princ. Practice Parallel Program.*, 2007, pp. 24–34.
[44] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine, "The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring," *Sci. Program.*, vol. 20, no. 2, pp. 129–150, 2012.
[45] M. Samek, *Use an MCU's Low-Power Modes in Foreground/Background*, Quantum Leaps, Pittsboro, NC, USA, 2007.
[46] Y. Mo, C. Goursaud, and J.-M. Gorce, "Theoretical analysis of UNB-based IoT networks with path loss and random spectrum access," in *Proc. 27th IEEE Int. Symp. Pers., Indoor Mobile Radio Commun. (PIMRC)*, 2016, pp. 1–6.
[47] *Medium Access Control (MAC) Protocol Specification (Release 14), V14.3.0*, 3GPP Standard TS 36.321, Jun. 2017.
[48] *Multiplexing and Channel Coding (Release 14), V14.3.0*, 3GPP Standard TS 36.212, Jun. 2017.
[49] K. Klues et al., "TOSThreads: Thread-safe and non-invasive preemption in TinyOS," in *Proc. SenSys*, 2009, pp. 127–140.
[50] C. Y. Yeoh, A. Bin Man, Q. M. Ashraf, and A. K. Samingan, "Experimental assessment of battery lifetime for commercial off-the-shelf NB-IoT module," in *Proc. IEEE 20th Int. Conf. Adv. Commun. Technol. (ICACT)*, 2018, pp. 223–228.
[51] F. Wibowo, "Wireless communication design of Internet of Things based on FPGA and WiFi module," in *Proc. J. Phys. Conf. Series*, 2020, Art. no. 12035.
[52] G. Guan, W. Dong, Y. Gao, K. Fu, and Z. Cheng, "TinyLink: A holistic system for rapid development of IoT applications," in *Proc. ACM MobiCom*, 2017, pp. 1–29.
[53] A. Mackey and P. Spachos, "LoRa-based localization system for emergency services in GPS-less environments," in *Proc. Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, 2019, pp. 939–944.
[54] A. K. Sultania, C. Delgado, and J. Famaey, "Implementation of NB-IoT power saving schemes in ns-3," in *Proc. Workshop Next Gener. Wireless NS-3*, 2019, pp. 1–4.
[55] Y. Gao, J. Zhang, G. Guan, and W. Dong, "LinkLab: A scalable and heterogeneous testbed for remotely developing and experimenting IoT applications," in *Proc. IEEE/ACM 5th Int. Conf. Internet-Things Design Implement. (IoTDI)*, 2020, pp. 176–188.