Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 1068261, 11 pages
https://doi.org/10.1155/2018/1068261

Research Article

Software-Defined Virtual Testbed for IoT Systems

Joanna Sendorek, Tomasz Szydlo, and Robert Brzoza-Woch

AGH University of Science and Technology, Department of Computer Science, Krakow, Poland

Correspondence should be addressed to Tomasz Szydlo; tszydlo@agh.edu.pl

Received 31 August 2018; Accepted 29 October 2018; Published 8 November 2018

Academic Editor: Manuel Fernandez-Veiga

The Internet of Things concept has found applications in a wide range of solutions, including smart homes, smart cities, enterprise systems, and others. Due to the limited resources available on physical devices and the intermittent availability of wireless networks, IoT may hugely benefit from Mobile Cloud Computing as it may provide the necessary processing power and the storage. This, however, increases the complexity of IoT systems and calls for a flexible testing environment to simplify the development process. In this paper we propose the concept of a software-defined IoT testbed which enables an IoT system to be immersed and tested in a virtual environment in order to evaluate its behavior under controllable conditions. Additionally, the features of the implemented testbed prototype are discussed on the basis of a specific use case.

1. Introduction

Internet of Things is among the most quickly progressing domains of technology. This concept has found applications in a wide range of solutions, including smart homes, smart cities, enterprise systems, and sensor networks. One of the most notable consequences of the growing importance of IoT is the increase in the amount of data that can be gathered from networks of connected devices and processed in order to provide business and scientific value. This possibility and ease of data collection permits easier development of software in terms of data management and data-driven applications. At the same time, increasing physical capabilities of hardware have resulted in the proliferation of embedded devices and sensors, but also of network equipment and media. In effect, the growing importance of IoT is affecting multiple layers of the standard IT architecture.

IoT provided a way to address the scarcity of resources available on hardware devices, augmenting them with the capabilities of cloud environments. In particular, the Mobile Cloud Computing concept can impart several benefits to IoT [1] such as unlimited data storage, ability to harness the processing power of the cloud environment, security, and easy access to application data [2]. Integration of IoT with cloud environments inevitably increases the systems' complexity. Tight coupling of software, transmission media, and hardware in IoT solutions creates challenges in many aspects of design and testing. Based on [3], the following characteristics of IoT system can be identified:

(i) *distributivity*: imposed by multiple data sources;

(ii) *interoperability*: required both among devices and between devices and sensors;

(iii) *scalability*: implied by the ability to manage increasing amounts of data;

(iv) *resources scarcity*: caused by constraints of low-level devices;

(v) *security*: demanded by acquired standards.

In effect, a typical IoT system may be regarded as highly complex, and many architectural aspects need to be covered in the design of such systems, from low-level sensors to interdevice communication and data processing applications. From the design perspective, IoT systems require compatibility between all of their components. Challenges stemming from the aforementioned characteristics can be addressed in a piecemeal fashion [4], but any solution targeted at a single subsystem may prove inadequate for other subsystems. The problem of design and implementation of IoT systems is therefore challenging, but may be ameliorated by suitable development tools, such as sensor network simulators [5, 6]. Nevertheless, such systems do not permit testing of high-level applications and their interaction with cloud services developed for smart devices.

In this paper we propose a testbed aimed at end-to-end testing of IoT systems from the application point of view. The proposed testbed is meant for testing high-level application protocols and interaction with cloud services rather than analyzing low-level nuances of, e.g., medium access control in wireless communications. The main idea behind the proposed testbed is to immerse the components under test in a virtual environment imitating real-world conditions. We therefore ground our approach in simulating real-world environments, devices, and sensors [7] which, on the one hand, does not require custom hardware for application development, while on the other hand permits integration with existing IoT hardware devices depending on the maturity of the tested IoT system. In our concept, the following elements can be substituted by their virtual counterparts in the testbed:

(i) *environment*: where real-world environments are replaced by simulated ones;

(ii) *sensors*: where sensors are virtual and measurements simulated;

(iii) *devices*: where devices are also virtual.

In order to ensure flexibility of virtual testbed components creation, we introduce `software-defined` management of the testbed itself. The process of connecting virtual sensors to virtual devices, i.e., *virtual wiring*, and the creation of a virtual environment, along with a scenario which expresses its behavior, is defined by the software. Further below we will define immersion levels that can be achieved during the testing process. By applying the software-defined concept together with a simulation-based testbed we may test all the aforementioned characteristics of IoT systems, with the following added benefits:

(i) decrease in the cost of testing by reduced demand for hardware;

(ii) full control over the course of testing;

(iii) ability to test applications under conditions which mimic their respective production environments;

(iv) ability to design and evaluate system architectures;

(v) ability to imitate sensor measurements, especially those related to anomalous conditions.

The scientific contribution of the paper is as follows: (i) the concept of a software-defined IoT testbed, (ii) the proposed architecture of a software-defined IoT testbed, and (iii) verification of the environment on the basis of a use case.

The paper is organized as follows. Section 2 describes the related work. Section 3 presents the concept of a software-defined testbed. Section 4 covers the environment emulator component while Section 5 describes the virtual sensors. In Section 6 the details of virtual devices are presented. In Section 7 the evaluation of the proposed solution is discussed and in Section 8 we conclude the paper.

2. Related Work and Existing Concepts

The main domain of our work presented in this paper is IoT testing, a topic already well researched and described in multiple publications, several of which we discuss in Section 2.3. However, our proposed methodology involves adoption of the *software-defined* concept in this domain in order to provide a novel solution answering common IoT testing issues which stem from integration with cloud services. To the best of our knowledge, a similar concept has not previously been described; therefore, we divide related work into three parts corresponding to the areas mentioned. In Section 2.1 we describe the role of Mobile Cloud Computing in the context of IoT, in Section 2.2 we try to place our work in the context of other *software-defined IoT* solutions, and in Section 2.3 we describe other IoT testing approaches.

2.1. Mobile Cloud Computing and IoT. The role of cloud computing in the context of IoT in large-scale data processing systems has been broadly described in several publications. The authors of [8] introduce a new term, `CloudIot paradigm`, which represents this specific coupling of both concepts. In the paper, the complementary nature of cloud computing and IoT is highlighted, where cloud-based environments provide computing and storage resources while IoT devices remain responsible for data gathering and user interaction. The need to integrate cloud architectures with IoT solutions is described in [9]. In this work, the authors describe not only cloud computing, but also a related concept—Fog Computing—which focuses on moving data processing nearer to devices in order to maximize the overall efficiency of the entire IoT solution. The paper describes Mobile Cloud Computing as one of the prospective solutions for enhancing the mobility of such systems. In [10] the Mobile Cloud Computing idea is described as a means to integrate cloud computing with mobile environments. Similarly to the IoT domain, the cloud serves as both the computational and storage resource provider. According to the authors of [11] the combination of Internet of Things and Mobile Cloud Computing can be very beneficial to the development of big data applications. Thus, the development of such solutions requires new testbed environments where end-to-end solutions and applications can be verified and validated.

2.2. Software-Defined Concept in IoT. There have already been several attempts to adopt the software-defined approach in the IoT domain. For instance, in [12] the authors describe the *Software Defined IoT* concept which involves controllers mediating communication between devices and the network infrastructure. In the work mentioned, the *software-definition* term relates to the use of software to ensure optimal network parameters, similarly to the original *Software Defined Networks* [13] idea. An approach which addresses the transmission layer of the IoT architecture is described in detail in [14, 15].

A concept somewhat similar to the one presented in this paper is described in [16]. The authors present a software-defined architecture which assists in on-demand

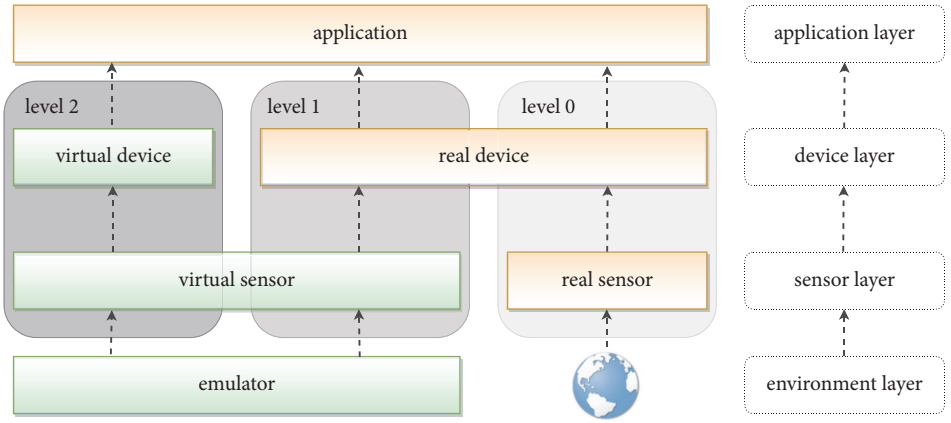

Figure 1: Conceptual diagram of SD-IoT test system.

provisioning of IoT applications with the required sensor-gathered data. The solution is based on *SD-VirtualSensors* and *SD-IoT controllers* treated as additional management layers interposed between the application and hardware. Although a similar nomenclature has been adopted in this paper, the role and mechanism of virtual sensor are quite different. We propose virtual sensors as the means of alternative instances of devices, equivalent in terms of behavior and capabilities and, crucially, able to replace physical hardware in the IoT testing scenario. In the described paper, *SD-VirtualSensors* are software modules which act as a proxy to the real sensors.

2.3. IoT Testing Methods. As previously described, testing an IoT systems introduces numerous challenges related to their complex nature. Therefore, there is a lot of ongoing research and solutions which address selected issues. In [17] the authors focus on conformance and interoperability testing of IoT systems. These aspects are not the main focus of our paper; however the solution applied in the cited publication is interesting since it also strongly relies on software test management. The authors define the concept of IoT Testing as a Service, which resolves three main issues identified in the paper: testing coordination, testing costs, and testing scalability. Our testing environment also addresses those exact subjects, but we do not limit ourselves to conformance and interoperability testing.

In [6] the authors propose the Internet of Things-Test Event Generator, able to simulate specific events and incoming data for the purposes of testing IoT data-driven application. While this approach provides the possibility of simulating multiple data sources and, in consequence, testing the scalability of an IoT application, it leaves out the testing of lower-level layers of the IoT system. For instance, testing system scalability is not limited to processing data on the application level, but also takes into account device capabilities and communication channels.

In terms of testing sensor networks, simulation and emulation are common approaches, implemented by multiple frameworks and libraries [18]. Our testbed resembles those solutions in that it also imitates the operation of a system built from low-level devices. The aforementioned solutions are often quite user-friendly; they provide robust graphical user interfaces, statistical modules, and visualization features; however, they focus only on networking and communication among sensors. In our testbed focus is on the big picture of an IoT system; for instance, we provide virtual devices suitable for developing applications in system-specific conditions.

3. Software Defined IoT Testbed Concept

In the paper, we propose a novel methodology and architecture for an IoT testbed which is based on the immersion of system components under test in a virtual environment. In order to enable testing of IoT applications, we have developed a software-defined IoT testbed. The overall schema of an IoT system relying on sensor data can be described in terms of a three-layer model [3], reflecting the dataflow between the source of data, i.e., (i) sensors, through (ii) devices to the (iii) user application. Figure 1 presents a conceptual view of the proposed testbed environment and its decomposition into the previously mentioned layers of the IoT architecture. The layer closest to the data source will be called the `sensor layer`. For all components (e.g., electronic sensors) used in the system this layer will comprise selected measurements of the environment. Next, we define the `device layer` containing all the IoT devices to which sensors from the `sensor layer` are directly connected. Finally, the `application layer` contains the applications which are being developed and tested. These applications can obtain access to cloud services and other components of the IoT system.

The general idea behind the testbed described in this paper is to provide virtual counterparts of components in the `device layer` and `sensor layer`, all interchangeable with real components for the purposes of testing. In subsequent sections we will refer to them as `VDevices` and `VSensors`, respectively. Setting up the environment—creating `VDevices`, wiring to `Vsensors`, and managing virtual environments—is performed in a programmable way.

From the perspective of functionality, the main difference between emulated sensors and real ones is the

Table 1: Supported configurations of the testing environment and typical use cases.

Immersion level	Device	Sensor	Example use cases
0	real	real	(i) typical usage of the implemented IoT system (ii) final product testing
1	real	virtual	(i) simulating edge conditions for the IoT system (e.g. extreme temperatures measured by sensors) (ii) testing device capabilities (iii) choosing appropriate sensors in the design phase
2	virtual	virtual	(i) low-cost simulation of multiple data sources (ii) testing horizontal scalability of the system (iii) testing applications working on the edge

source of measurements available to the system. For real sensors, this is obviously the surrounding world, while for emulated sensors a scenario-based `environment emulator` component is responsible for providing `VSensors` with simulated measurements according to the user-defined testing scenario. Since the specifics of these measurements should be fully configurable by the user and independent of the implementation of `VSensors`, we introduce additional layer in the architecture schema, the `environment layer` which expresses the virtual environment instantiated by the `environment emulator` component. In the following sections, details of the system layers will be discussed.

In our concept, the components of the IoT system can be replaced by their virtual counterparts. The process of gradual replacement of real components with virtual ones will be referred to as `immersion` of the system under testing, in accordance with the layered architecture described above. We can identify three levels as presented in Table 1.

The zero-immersion level corresponds to a real-life situation when no part of the system is virtual. Thus, it may be regarded as the referential level. In this case we are dealing with clearly separated components on each layer of the architecture: real sensors are connected to real hardware devices. All sensor measurements come from the real surrounding world, which means that they undergo constant changes and are usually not repeatable across system tests. This may add complexity to the development process.

The purpose of the first immersion level is to enable simulation of certain environment conditions which, although possible in the real world, may be hard to recreate under testing. To achieve this, we introduce `VSensors`, software which imitates the behavior of real sensors according to the given model of measurement values. Since VSensors need to be connected to real devices, they have to be implemented on an external electronic board controlled by the environment emulation module. Additionally, they have to implement exactly the same sensor-device communication protocols as real sensors in order to provide interchangeability. Only under these conditions we may claim equivalence of `Vsensors` and real sensors from the perspective of devices they are connected to. The ability to recreate specific environment conditions simplifies testing because conditions may be replicated, and it is also possible to simulate various anomalous scenarios.

On the second immersion level, the aforementioned concept is taken further and thus not only sensor logic is replaced by software, but also the whole device environment. This enables replacement of all hardware platforms used in the system. We introduce `VDevices`, fully virtual software which is able to emulate peripheral access for the corresponding platform and communication protocols used both for sensor-device communications and interdevice communications. Again, we aim at equivalence between virtual and real devices from the perspective of the application layer. To ensure portability between real and virtual devices, the following platform properties have to be emulated: libraries and modules required by applications, network interfaces, and access to peripherals. This is further discussed in Section 6.

The proposed solution should lower the costs of system design and testing by eliminating hardware modules while retaining the following properties of the system:

(1) type and format of data gathered by sensors;

(2) application-level protocols used for communication between IoT system components, such as MQTT, HTTP, etc.;

(3) low-level protocols used by the sensors to communicate with devices, such as I^2C, GPIO, SPI, and so on;

(4) network-layer protocols for interdevice communication, such as IP.

4. Environment Layer

The `environment layer` is the one closest to the data source in the described IoT system architecture. The main responsibility of this layer is to provide sensors with measurement values. In the case of real sensors, data is gathered directly from the surrounding world and for `VSensors` it is provided by a component developed for this purpose, the `environment emulator` [7]. Its architecture is depicted in Figure 2 and consists of the following subcomponents:

(1) `test scenario`: script describing the execution of the simulation;

(2) `virtual world`: component which defines the virtual space and changes depending on the test scenario;

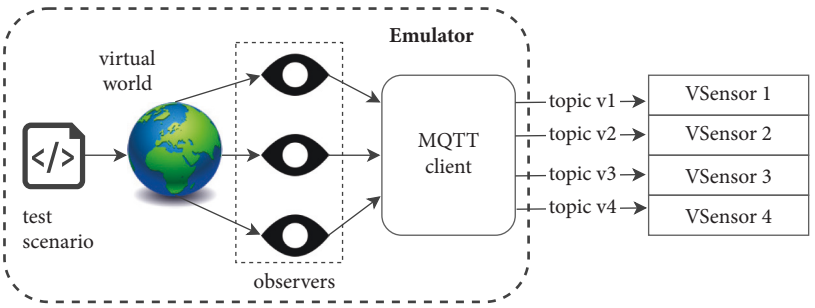

Figure 2: Overview of the emulator component architecture.

(3) `observers`: units assigned to particular places in the virtual space, able to notice ongoing changes in the virtual world;

(4) `communication module`: communication between observers and `VSensors` necessary for delivery of simulated measurements.

The idea behind observers is that each of them corresponds to one of the `VSensors` and constitutes its representation in the virtual world. In other words, placing the observer in the virtual space is equivalent to placing a real sensor at particular coordinates of real space, for example, placing a thermometer near an entrance, a smoke detector on a wall, or a humidity sensor in the soil. Since data is generated in the virtual world and then separately registered by observers, it becomes possible to change the placement or type of a sensor during test scenario execution without restarting that scenario. The computational complexity of the virtual world simulation should not burden the time accuracy of the simulation. It should be possible to simulate the virtual world in real-time.

Regarding integration of the `environment emulator` with other components of the testbed, delivery of measurements data to `VSensors` is a crucial aspect. In the proposed solution we use `Message Queuing Telemetry Transport (MQTT)` which is a lightweight protocol popular in the realm of IoT. On the side of `emulator`, the MQTT client receives data from observers and dispatches it to a topic bound to the `VSensor` instantiated by each observer.

5. Sensor Layer

As previously described, the main goal of the sensor layer is to provide measurement values for device and application layers. Two main aspects can be identified with regard to communication between sensors and the devices they are connected to: the low-level protocol used by real sensors (e.g., I2C, ADC/DAC, GPIO, etc.) and the internal logic of the sensor. In order to replace real sensors with their virtual counterparts, the `VSensors` have to imitate the exact behavior of both elements.

The main research goal was to design `VSensors` in such a way that they can remain independent of the immersion level of the tested IoT system. This is enabled by the *virtual breadboard* module which implements the internal logic of sensors while low-level communication (e.g., GPIO, I2C) is abstracted by operations of the corresponding low-level communication protocols (e.g., `vSensorManager_setPinState()`). Usage of the module depends on the immersion level of the system being tested:

(i) On the first immersion level, the IoT device undergoing tests is connected to the virtual sensor; thus the *virtual breadboard* module imitating real hardware sensors is executed on an external microcontroller board, and communication protocol operations are mapped to real hardware components,

(ii) on the second level the IoT device is also virtual; therefore, low-level protocol operations are mapped directly to libraries used by the tested IoT system for sensor communication.

Further below we describe how this issue has been addressed and present details of `VSensor` implementation.

The architecture of the software-defined `VSensor` management environment is depicted in Figure 3. As presented, this module consists of the following three components:

(i) *toolbox*: set of implemented `VSensor` classes emulating the behavior of corresponding sensor;

(ii) *virtual breadboard*: a virtual base used to wire virtual sensors to devices;

(iii) *sensor manager*: the main component responsible for controlling the lifecycle of the `Vsensors`, wiring, and handling messages from the environment emulator to the sensors following initialization.

The process of `VSensor` creation is as follows:

(1) The `Sensor manager` receives a creation message from the *environment emulator* module;

(2) Based on the event data, the manager creates an object of the appropriate virtual sensor type. The object is assigned the following properties based on event data:

(i) unique `id` corresponding to the acquired `manager table slot`; the maximum number `k` of sensors depends on resources available to the device which implements the module;

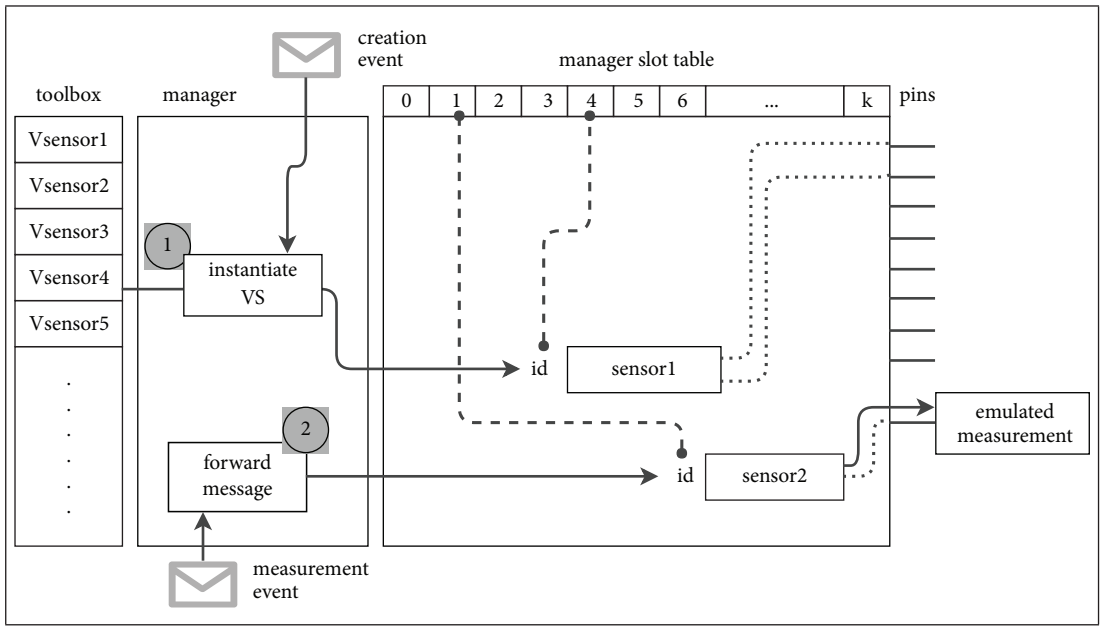

Figure 3: Management of virtual sensors.

Table 2: Virtual sensors.

sensor-device communication	sensor name	measurements
I2C	BME280	temperature, pressure, humidity
GPIO	switch, LED	two states (1/0)
ADC	LM35	temperature

(ii) pins which the sensor will be attached to;

(iii) sensor name, used as a reference in future messaging.

Following successful creation of the `VSensor`, the `sensor manager` is able to communicate with that sensor and pass specific messages, for example, measurements to be emulated. This process (2) consists of the following steps:

(1) the manager receives a message event from the `environment emulation` module with a forward flag and passes it to the `VSensor` with a given name;

(2) the `VSensor` converts the event into an adequate measurement emulated on the pins it is attached to.

The entire process, starting with setup, through communication and emulation of measurements, is dynamically defined by software and event-driven. It is worth noting that the described flow of actions enables `VSensors` to behave exactly like real sensors with regard to the low-level communication protocol. Table 2 lists the supported low-level communication protocols and sensor types. For each protocol, we have chosen one representative sensor implemented in the testbed prototype.

6. Device Layer

As described in Section 2.2, introducing `VDevices` enables the application to be executed in a virtual environment in the same way as with real devices. In the preceding sections, we mentioned three aspects which need to be addressed: communication with other devices and cloud services, provisioning libraries, modules, etc., and access to device peripherals. The third aspect is arguably the most complex, since it involves communication between the sensor layer and device layer within the aforementioned architecture schema, which is very hardware-specific. Depending on the embedded platform used, this may accomplished in various ways; however, we may distinguish two broad categories of embedded devices: Linux-based and non-Linux-based. This distinction underpins our characterization of `VDevice` implementation which is presented in the following sections.

6.1. Linux-Based Devices. The Linux operating system provides a hardware abstraction layer to user applications. In most cases, from the application point of view the specific hardware architecture of the processor used in the embedded device is unimportant; it could be, for example, ARM, x86, or MIPS. Applications typically rely on system libraries that expose well-known APIs and are compiled for specific hardware architectures. This property has been used to enable virtualization of Linux-based devices using lightweight containers.

In this class of devices, peripherals are usually exposed via files in the */dev* directory operated by the kernel. Since different boards use different file structures and different pin numbering, the code operating on those files is highly platform-dependent and requires modifications to match the specific device it is running on. Additionally, each I/O

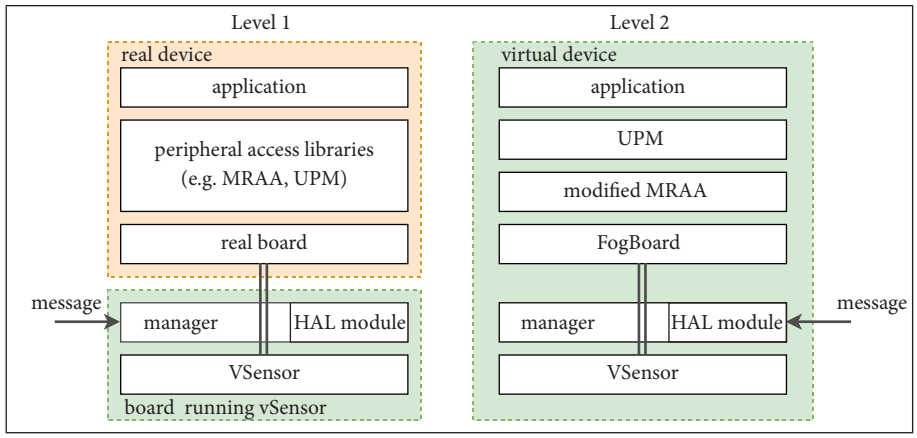

Figure 4: Connecting VSensors to real and virtual devices on immersion levels 1 and 2.

protocol implements its own file usage schema, all of which hampers development of IoT applications. There are several software libraries for accessing hardware peripherals developed for particular types of devices, but in order to ease the development process (given the multitude of board types), the `MRAA` (https://github.com/intel-iot-devkit/mraa) library has been developed by Intel. It provides a unified API for many types of boards (a full list is included in its documentation (https://iotdk.intel.com/docs/master/mraa/index.html) and communication protocols, such as I^2C, GPIO, LED, SPI, and more. On the one hand, it provides a convenient API to access peripherals and on the other hand, it provides applications with portability via runtime board detection. Additionally, it enables developers to write code in several programming languages: Python, C++, Java, and Node.js. All of these interfaces are based on a core C implementation which is the one we use in the presented testbed. In addition to the MRAA library, the Intel IoT Developer Kit contains the UPM (https://github.com/intel-iot-devkit/upm) tool which is a repository of drivers for many sensors. UPM adds convenience to software sensor management by covering the hardware protocol calls from MRAA with a higher-level API. In the proposed testbed we use both described projects to emulate peripheral access in the `VDevice`.

Figure 4 depicts the architecture of the `VDevice`, its interaction with the `VSensor` (immersion level 2) and a comparison with a real device interacting with the `VSensor` on an external electronic module (immersion level 1). As described in Section 5, `VSensors` controlled by the manager expose their measurements to the device using a specific low-level hardware communication protocol. Platform-dependent aspects of this communication are provided by the `HAL module` which takes care of the actual implementation of the protocol used by the board the `VSensor` is running on.

We may compare the architecture of the `VDevice` with the real architecture by discussing interactions between the sensor and device layers in the aforementioned architecture model depending on the immersion level:

(1) level 1: real device reads measurements from `VSensors` operating on an external hardware board; IoT application may use any software library to access hardware peripherals of the IoT device,

(2) level 2: both the device and `VSensor` run in a fully virtual environment; IoT application has to use a modified version of the MRAA library.

In the case of level 1, an instance of `VSensor` runs on the external board and connects to the IoT device via appropriate pins. Since `VSensor` uses the exact implementation of selected hardware protocol via the `HAL module`, the device is not aware of differences between a real sensor and a `VSensor`. In this case, any library for accessing device peripherals can be used. For example, we may use the original version of MRAA and UPM libraries, or platform-specific solutions such as Raspberry Pi or Intel Galileo.

For the second immersion level we have extended the MRAA library to handle additional type of boards, the virtual `FogBoard`. This enables MRAA to be run not on hardware, but under any Linux operating system, and provides access to virtual peripherals. Automatic board detection and the rest of MRAA logic, including the exposed API, are preserved so that the original UPM can be used to manipulate sensors. The difference between using `FogBoard` and other types of boards concerns access to peripherals which, in the case of real boards, is realized by kernel-managed files, while in the case of a virtual board is performed by the virtual breadboard.

6.2. Other Operating Systems and Hardware Architectures. In a situation where the operating system for embedded devices is tightly coupled with the hardware platform, e.g., as in ContikiOS, TinyOS, FreeRTOS, or the application is deployed directly on bare-metal resources, such a versatile virtualization method as the one presented in the previous section may not be feasible. In such circumstances it is necessary to emulate the hardware platform on low-level electronics. A commonly used emulation solution for these types of deployments is QEMU (https://www.qemu.org/). It enables virtualization of a full hardware architecture on a platform with another hardware architecture. Integration of *VSensors* with QEMU is not as versatile as with Linux-based devices and has to be performed for each hardware platform

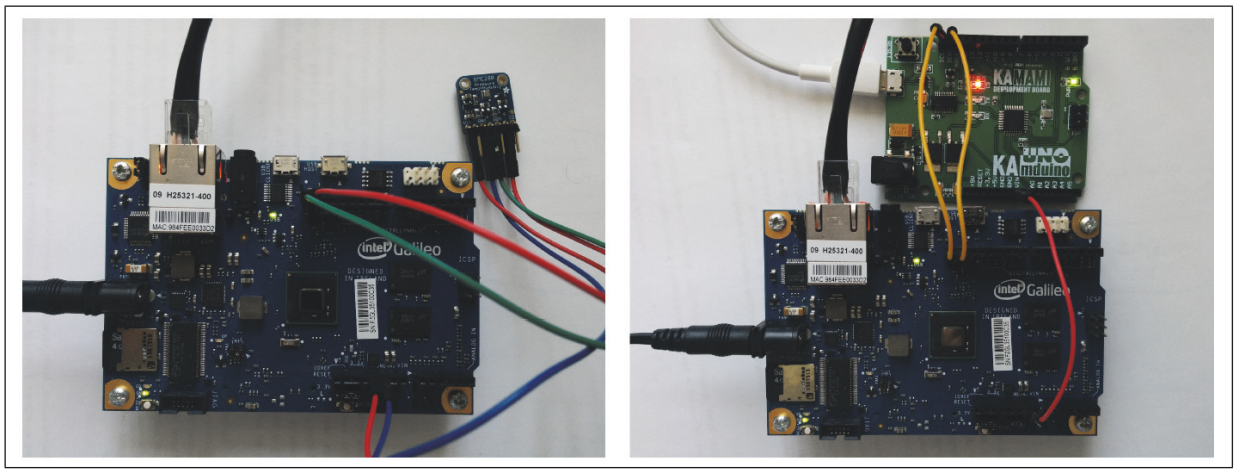

Figure 5: Device with level 0 and level 1.

separately. Nevertheless, QEMU can also be deployed in Linux-based virtual containers.

6.3. Automation of Virtual Testbed Creation. All of the described components of `VDevice` are compiled and configured under a Linux-based operating system. In order to isolate `VDevices` from the platforms they are running on and enable fully automatic assembly of many `VDevices`, we have decided to enclose each of them in a Docker container. They provide independence across devices, which enables development of applications in an environment similar to the one provided by the Linux board. Since limiting the amount of resources required to run each `VDevice` was our priority when selecting a specific distribution of Linux, all our containers were prepared using an Alpine image (https://hub.docker.com/_/alpine/). This image is the basic version of a Linux system, stripped of unnecessary dependencies. However, any other Linux image can be used in the testbed without altering its logic and architecture. In order to automate the process of creating many `VDevices`, we use *Ansible* (https://www.ansible.com/), which defines configurations in setup files for direct execution. With *Ansible*, the user can define all virtual devices in a single *yaml* configuration file and ensure all of Docker containers are properly created.

An important aspect of emulating the environment, which has not been addressed, is the networking between `VDevices`. To fully cover all of the aspects of an IoT system in the testbed, it is required to simulate the network parameters of real devices, such as bandwidth, transmission delay, network privacy, addressing, and many more. Docker itself enables configuration of addresses and domains on the level of the OSI/ISO network layer, enabling creation of complex network topologies and interaction with real devices or cloud services. Since this mechanism relies on the underlying network available to the Docker host, tools such as `pumba` (https://github.com/alexei-led/pumba) can be used for chaos testing. To simulate all network parameters and topologies, the GNS3 (https://gns3.com/) framework may be applied.

7. Use Case and Verification

In this section we present the usage of the software-defined testbed for IoT. In our use case, the goal of the IoT application is to measure the environment conditions in a greenhouse. The prototype is based on an Intel Galileo board and the BME280 sensor, as presented in Figure 5 (left side). The application, presented in Listing 1, reads the environment temperature and sends it to the IoT platform deployed on the local server. The use case presented in the paper is intentionally simplified in order to emphasize the benefits of the presented software-defined testbed. In accordance with the previously discussed specification of immersion levels, level zero represents a real device connected to real sensors. Figure 6(left side) depicts the observed temperature values.

In order to determine how the IoT application behaves under certain conditions, a device with a BME280 sensor has been immersed on level one in the testbed and thus the real BME280 has been replaced by its virtual counterpart. In this case, the virtual manager is deployed on the Arduino UNO board (the source code of the virtual manager is available on https://github.com/tszydlo/emulator/). The virtual vBME280 sensor is created and virtually wired to the A4 and A5 pins of the Arduino board, i.e., SDA and SCL signals of the I2C protocol. As shown in Figure 5 (right side), Intel Galileo is wired to the Arduino board on which the virtual sensor is instantiated. The measured values are provided by the emulator following a simple scenario, as presented in Listing 2 where temperature oscillates according to a sinusoidal pattern. The collected temperature values are then sent to the IoT platform as presented in Figure 6 (right side).

Instead of running the IoT application on real hardware, the device can be virtual (immersion level 2); i.e., the application can be executed by a virtual device instantiated as a Docker container. Listing 3 presents the code for Docker container setup used in the use case for virtual devices. In this case, instead of an external electronic module with virtual sensors, these sensors can be created and virtually wired directly in the modified MRAA library as discussed in the previous section. The main benefit of this immersion level

```
import mosquitto
from upm import pyupm_bmp280 as sensorObj
Import ...
id = 7504343
def main() :
    mqttc = mosquitto.Mosquitto()
    mqttc.connect("172.17.88.230", 1883, 60)
    # Instantiate a BME280 instance using default i2c bus and address
    sensor = sensorObj.BME280()
    while (1):
        sensor.update()
        json_template = { 'time' : datetime.datetime.now().strftime("%Y-%m-%d
            %H:%M:%SZ" ),
                          'id': str(id),
                          'temp': sensor.getTemperature()}
        mqttc.publish("/bme",
            json.dumps(json_template), 0, True)
    time.sleep(1)
if __name__ == '__main__':
    main()
```

Listing 1: IoT application used in the use case.

```
Import ...
#we are using virtual sensors on Arduino connected via USB
s_client = SerialClient("COM4")
#virtual wiring – virtual BME280 sensor exposed on I2C bus by default (pin PD4 and PD5)
env_sensor = vBME280(1,s_client)
@every(start=2, seconds=3)
def generator_step():
    tt = executor.get_time()
    env_sensor.set_temp_v(math.sin((tt % 3600) / 3600.0 * 2 * 3.14)*10 + 20 )
    env_sensor.set_pressure_v(math.cos((tt % 3600) / 3600.0 * 2 * 3.14) * 10 + 950)
    env_sensor.set_humidity_v(math.sin((tt % 3600) / 3600.0 * 2 * 3.14) * 40 + 50)
if __name__ == '__main__':
    start_executing()
```

Listing 2: Environment emulator code used in the use case.

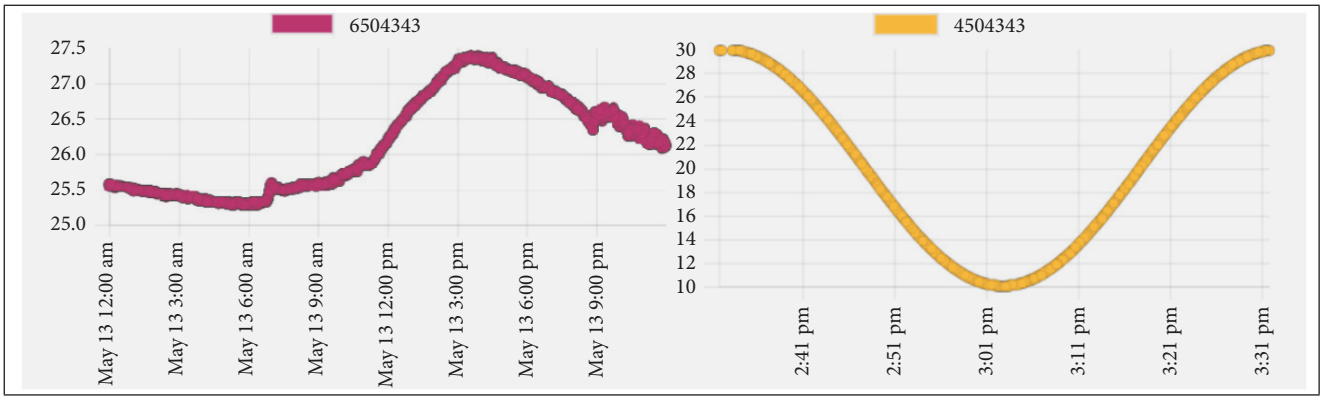

Figure 6: Temperature obtained by the application deployed on the Intel Galileo board from BME280 and vBME280 sensors, respectively.

```
FROM tszydlo/docker−fogdevices
RUN apk add py-setuptools
RUN easy_install-2.7 requests
RUN apk update && apk add ca-certificates wget && update-ca-certificates
RUN export MRAA_FOGDEVICES_PLATFORM_ID="$DEVICE_ID"
RUN export MRAA_FOGDEVICES_PLATFORM_BROKER="xxx.xxx"
RUN cd /root
RUN wget https://foo.net/application.py
CMD python application.py $SERIAL_NUMBER
```

Listing 3: Virtual device container configuration.

```
– name: Start many device containers
        docker_container:
          name: "{{ item.name_of_container }}"
          image: device_image
          state: started
          env:
            DEVICE_ID: "{{ item.device_id }}"
        with_items:
          – {name_of_container: 'fog_device1' , device_id: 'fog_device_1' , serial_number: 'xxx'}
          – {name_of_container: 'fog_device2' , device_id: 'fog_device_2' , serial_number: 'xxx '}
```

Listing 4: Fragment of Ansible script responsible for virtual devices creation.

is that there is no need for any external IoT device to test the IoT application. It is also worth noting that the same application can be used on different immersion levels offered by the testbed platform. The main difference between a real and virtual device lies in hardware access, which, in the case of the presented Docker container, is realized using modified MRAA. Listing 4 presents the fragment of automated setup of environment consisting of multiple devices as described in Section 6.3. Details of the approach and its possible applications to simulation environments are presented in [7].

8. Summary and Future Work

In the paper we have presented the concept of a software-defined environment for IoT testing. Such an environment provides a flexible way to test and validate IoT applications under development. A prototype implementation of the testbed has been verified with a use case which shows how the IoT system may be immersed in a virtual environment. The proposed solution can be weaved in the continuous development process of the IoT systems and thus the functionality of the application will be verified in the virtual environment that reflects real-world conditions.

As future work, the presented concept of a software-defined testbed will be further extended in two directions. First, we will undertake the development of new sensor types such as pressure sensors, structural monitoring sensors, and sensors related to human-computer interactions. The second direction of development involves provisioning data to virtual sensors in the form of a time series. This would be especially useful, for example, in applications which aim at detecting engine failures by analyzing vibration using accelerometers.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The research presented in this paper was partially supported by the National Centre for Research and Development (NCBiR) under Grant no. LIDER/15/0144/L-7/15/NCBR/2016.

References

[1] K. E. Psannis, S. Xinogalos, and A. Sifaleras, "Convergence of internet of things and mobile cloud computing," *Systems Science & Control Engineering*, vol. 2, no. 1, pp. 476–483, 2014.

[2] C. Stergiou and K. E. Psannis, "Efficient and secure BIG data delivery in Cloud Computing," *Multimedia Tools and Applications*, vol. 76, no. 21, pp. 22803–22822, 2017.

[3] M. R. Abdmeziem, D. Tandjaoui, and I. Romdhani, "Architecting the internet of things: state of the art," in *Robots and Sensor Clouds*, vol. 36 of *Studies in Systems, Decision and Control*, pp. 55–75, Springer, Cham, Switzerland, 2016.

[4] E. J. Marinissen, Y. Zorian, M. Konijnenburg et al., "IoT: Source of test challenges," in *Proceedings of the 2016 IEEE European Test Symposium (ETS)*, pp. 1–10, Amsterdam, Netherlands, May 2016.

[5] J. Eriksson, F. Österlind, N. Finne et al., "COOJA/MSPSim: interoperability testing for wireless sensor networks," in *Proceedings of the 2nd International ICST Conference on Simulation Tools and Techniques (SIMUTools '09)*, Brussels, Belgium, March 2009.

[6] L. Gutiérrez-Madroñal, I. Medina-Bulo, and J. Domínguez-Jiménez, "IoT–TEG: Test event generator system," *The Journal of Systems and Software*, vol. 137, pp. 784–803, 2018.

[7] T. Szydlo and J. Sendorek, "Leveraging virtualization for scenario based IoT application testing," in *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems*, vol. 13, pp. 229–235, September 2017.

[8] A. Botta, W. De Donato, V. Persico, and A. Pescape, "On the integration of cloud computing and internet of things," in *Proceedings of the 2nd International Conference on Future Internet of Things and Cloud (FiCloud '14)*, pp. 23–30, Barcelona, Spain, August 2014.

[9] M. Yannuzzi, R. Milito, R. Serral-Gracia, D. Montero, and M. Nemirovsky, "Key ingredients in an IoT recipe: fog computing, cloud computing, and more fog computing," in *Proceedings of the IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD '14)*, pp. 325–329, Athens, Greece, December 2014.

[10] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[11] C. Stergiou and K. E. Psannis, "Recent advances delivered by Mobile Cloud Computing and Internet of Things for Big Data applications: a survey," *International Journal of Network Management*, vol. 27, no. 3, Article ID e1930, 2017.

[12] T. Ninikrishna, S. Sarkar, R. Tengshe et al., "Software defined IoT: Issues and challenges," in *Proceedings of the 2017 International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 723–726, Erode, India, July 2017.

[13] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and OpenFlow: from concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[14] S. Bera, S. Misra, and A. V. Vasilakos, "Software-Defined Networking for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1994–2008, 2017.

[15] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif, "Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications," *IEEE Communications Magazine*, vol. 53, no. 9, pp. 48–54, 2016.

[16] T. M. Nguyen, D. B. Hoang, and T. Dat Dang, "Toward a programmable software-defined IoT architecture for sensor service provision on demand," in *Proceedings of the 2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–6, Melbourne, Australia, November 2017.

[17] H. Kim, A. Ahmad, J. Hwang et al., "IoT-TaaS: Towards a Prospective IoT Testing Framework," *IEEE Access*, vol. 6, pp. 15480–15493, 2018.

[18] M. Imran, A. M. Said, and H. Hasbullah, "A survey of simulators, emulators and testbeds for wireless sensor networks," in *Proceedings of the International Symposium on Information Technology (ITSim '10)*, vol. 2, pp. 897–902, Kuala Lumpur, Malaysia, June 2010.