2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovations

T-Suite: A RESTful Distributed Development and Runtime Environment on Things Devices

Xiaohui Peng*, Lu Chao*†, Yifan Wang*†, Zhiwei Xu*
*Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190
†University of Chinese Academy of Sciences, Beijing, 100190
Email:{pengxiaohui,chaolu,wangyifan2014,zxu}@ict.ac.cn

***Abstract*—Tens of billions of IoT devices are connecting to the Internet, entailing several essential issues such as high response latency of IoT applications, network congestion, and privacy. New computing paradigms are rising and trying to offload the computation from could to edge and things devices. Due to the extremely resource-constrained environment and distributed architecture, that developing and deploying applications on heterogeneous low-end IoT devices to afford the computing load is a significant challenge for developers. In this paper, we propose T-Suite, a lightweight script application development and runtime framework based on REST style for IoT devices. T-Suite includes a RESTful development and deployment environment and a mixed runtime environment which can improve the performance of script applications. A series of micro-benchmarks are conducted, and the results show that T-Suite enables scripting on MCU based IoT devices with a little increment of computing resource consumption.**

***Index Terms*—Internet of Things, Development and Runtime Environment, Script Language, Resource-Constrained, REST**

1. Introduction

Recently, new paradigms such as edge computing [1], fog computing [2] are proposed with the vision of moving some response sensitive computations from the cloud to the devices at the network edge. However, many devices in the edge computing system are resource-constrained and the system is highly heterogeneous. Therefore, developing and deploying applications on these devices (i.e., Low-end IoT device, hereafter referred as **"things device"**) are difficult. To realize these computing models and move some computation tasks from cloud to devices, the following concerns should be addressed: (1) How to dynamically deploy the programs on-the-fly and support the massive heterogeneous things devices with the least efforts? (2) How to debug the programs in a fully-dynamic and distributed environment without attaching and suspending the devices? The developers can't reproduce and locate the bugs which happened in the physical world;

Applications on the devices are modified frequently by on-the-fly firmware upgrades, which replaces the entire firmware image including the operating system. In new computing paradigms, this issue becomes more critical because of the service interruptions and the complex compiling and deploying procedures. Most of the computational tasks on things device are quite simple. These tasks include data filtering, coarse feature extraction, automatic controls(e.g., IF-THEN rules) and etc. They can be divided into a **native part** and a **script part**. The native part usually interacts with low-level system resources such as reading the temperature value from a register and turning on a machine through system APIs. The script part usually describes business logic such as calculating the average temperature using values from thermometers deployed in different sections of a room. Such a design principle had already been applied in the mini-program of WeChat [3], and it has shown a good performance of quickly developing and deploying mini programs. We also adopt such a layered application design principle in this paper.

Z. Xu et al. proposed the Φ-Stack [4] which tries to extend the RESTful [5] Web system architecture to things devices. This paper is the preliminary effort of the ΦDK which tires to enable uniform interface and code-on-demand principles on things devices [5]. According to the observations and challenges, we design and implement a RESTful distributed development and runtime environment on these devices, called T-Suite, which contains the following two parts:

- **RESTful Distributed Development Environment(RDDE)**, which is a RESTful distributed Integrated Development Environment(IDE), transforms the traditional tight-coupled compiler, debugger and other toolchains into loose-coupled RESTful modules and objects. RDDE enables the abilities to compile, debug and deploy the things application through a remote, batched, and unified manner.
- **Mixed Runtime Environment(MRE)**, which is a runtime environment that accelerates the execution of the code with a native execution engine, and improves the portability of the scripting code with a script engine.

Evaluations on the prototype system of T-Suite show that RDDE enables a distributed manner of debugging and deploying applications on things devices. MRE significantly improves the flexibility with the script code and improves the performance by introducing a native code engine.

978-1-5386-9380-3/18/$31.00 ©2018 IEEE
DOI 10.1109/SmartWorld.2018.00184

2. RESTful Distributed Development Environment

2.1. Architecture

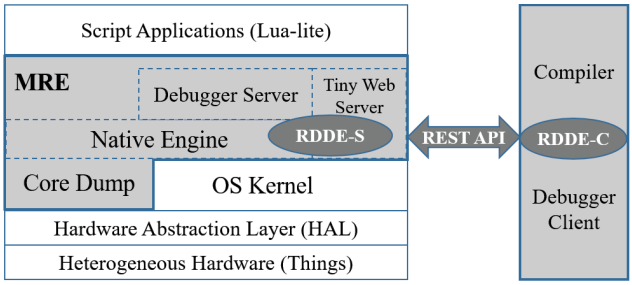

Figure 1. T-Suite components and architecture.

As shown in Figure 1, the RDDE includes a server side component (RDDE-S) and a client-side component (RDDE-C). They run on things devices and desktop computers respectively and communicate with RESTful APIs. RDDE-S consists of a tiny web server, a core dump component, a debugger server, and a mixed runtime engine which supports script and native code execution. RDDE-C includes a compiler and a debugger client. Developers can create script applications in RDDE-C and upload them to RDDE-S for execution and debug on things devices using RESTful APIs.

2.2. Compiler and Debugger

The compiler compiles script code to bytecode on the client-side computer. The debugger consists of a server and a client. Debugger client is responsible for transmitting the debugging instructions issued by users. The debugger server on things executes the script programs interactively by receiving the commands issued by the client when debugging the application. It collects all the runtime errors from the mixed runtime environment and the core dump module. The main debugging functions include getting the current runtime context, single-step debugging, breakpoint debugging, and backtracking. Our investigation shows that Lua [8] is faster than representative script languages such as Perl, Ruby, and Python. Thus, we implement the prototype compiler and debugger based on Lua 2.5.1. The debugger is implemented using two functions provided by Lua 2.5.1:

- *extern lua_LHFunction lua_linehook*
- *extern lua_CHFunction lua_callhook*

Lua_linehook adds a hook to target lines of the script source code. Before executing the target line, we can firstly execute the developer-defined functions. *Lua_callhook* is the hook function that is executed every time you enter and leave the function. In Lua-lite, we implement five different hook functions based on *lua_linehook* function provided by Lua.

- NONE_HOOK: initialize the debugger, which sets the initial value of the function stack depth as -1;
- CONTINUE_HOOK: move to the next executable line of the code;
- STEP_HOOK: single-step debugging. When the sub-function is encountered, enter into the sub-function and continue to perform single-step debugging, otherwise, move to the next executable code line;
- NEXT_HOOK: execute a single step. Different from STEP_HOOK, if the sub-function is encountered, it will stop after finishing the execution of sub-function instead of entering the sub-function;
- RUN_HOOK: continue to execute the program from the current line of code until the program ends or encounters a breakpoint.

2.3. Core dump

Core dump refers to the phenomenon that when the system gets interrupted due to the crash of the program or any other reasons, the records of system states, such as register value, stack pointer and memory management information, will be stored in the file systems for future accessing. The core dump component is located in the lower layer of the OS kernel, and can communicate with the hardware of things devices. The results of the core dump procedure can be sent to the developers directly through the RESTful APIs, or they can be stored in the ROM of the devices. When the crash happens, only a few extra storage will be needed to save the exception information and execution states.

We implement the core dump mechanism based on the STM32L152 development board dealing with 4 types of critical runtime faults: HardFault, BusFault, UsageFault, and MemoryManagementFault. When the fault occurs, core dump module records function stack pointers of each layer at the specified address of the ROM with the help of SP and LR register. The complete function callback stack can be recovered by reading the records in ROM with the help of tools such as addr2line.

2.4. Tiny Web Server and RESTful Protocols

The tiny web server runs in the operating system, as a common interfaces for interacting with external. It provides uniform access interfaces for things devices to expose their data and functions. In the T-Suite, Online debugging and scripts loading are performed via the RESTful services provided by the tiny web server. It greatly simplifies the application development and deployment compared to the traditional embedded systems in which developers have to build complex and fragile cross compiling environment to edit, compile, load and debug applications.

3. Mixed Runtime Environment

3.1. Layered Application Architecture

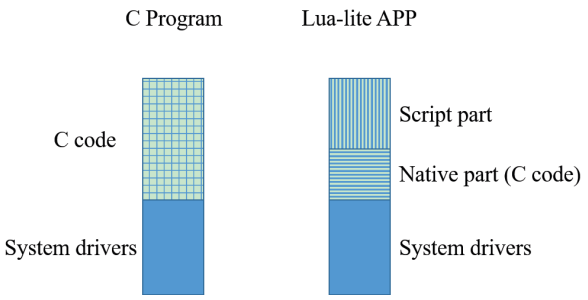

Figure 2. The demonstration of layered application architecture.

As shown in 2, we propose a layered application architecture in which applications are divided into two parts:

- Script part: script part contains business logic that decouples from lower drivers. It invokes standard APIs provided by native part to deal with system and hardware specific operations.
- Native part: native part handles all the system and hardware related operations and provides standard interfaces for script part.

This design will potentially improve the efficiency of application development and execution on things devices. WebChat's mini programs, which also adopts this design principle, has been proved to be efficient for developing micro programs on smart phones. To support this application architecture, we design a mixed runtime environment in T-Suite and introduce it in following sections.

3.2. Lua-lite Script Engine

Lua-lite script engine is built as a script virtual machine. It provides the runtime environment and manages the life-cycle of script applications. The virtual machine running on the things devices only accepts the compiled bytecode file, which will reduces the footprint significantly. Lua [8] is a lightweight script language that can run by embedding into applications written by other languages. Its footprint is smaller than most of the scripting languages. Moreover, it follows the ANSI C standard strictly which means good portability.

The Lua-lite and its runtime environment is implemented based on Lua kernel 2.5.1. In T-Suite, the Lua-lite script is first compiled into bytecode, and uploaded to the device and interpreted as the program instructions in Lua. We split the Lua runtime into two parts and distribute them on desktop PC and things devices respectively to reduce the memory occupation of resource-constrained devices. We also add the native engine which executes the native part of an application, and discard the instruction generation component.

3.3. Native Engine

Native engine is located above the operating system. It provides the script code with unified access and control interface of peripherals that connected to the MCU. Sensors (e.g., temperature, humidity, acceleration) and other peripherals (e.g., GPIO, UART, SPI, FLASH) can be registered as REST resources in the native engine. Due to the diversity of peripherals of the MCU-based device, it is very hard to program with the peripheral resources using script languages. Developers have to write application driver for each peripheral in the traditional embedded system. Other programmers may need to read the SDK documents or source code to create applications based on these drivers. We abstract resource accessing interface to limited meta-methods (e.g., Create, Read, Write and Delete) following REST uniform interface principle. It provides loosely-coupled RESTful services based on the CoAP [9] or HTTP protocol. The main functions of the native engine are listed as follows:

- Call the CPU, RAM memory resources of IoT devices to realize the basic programming language function, such as selection, branch, and loop;
- Provide unified access to resources of IoT devices such as GPIO, FLASH, UART, etc;
- Provide the dynamic registration and cancellation functions of the RESTful resources.

4. Evaluation

4.1. Experiment Setup

The main objective of the experiments is to evaluate the performance and efficiency of developing script applications with T-Suite on things devices. We try to examine the increment of computational resource consumption for enabling scripting function and adopting REST architecture style. As shown in Figure 3, we use a STM32 Nucleo-64 development board which is equipped with NUCLEO-L152RE MCU to implement a prototype system for the evaluation. The communication and sensing capacity are

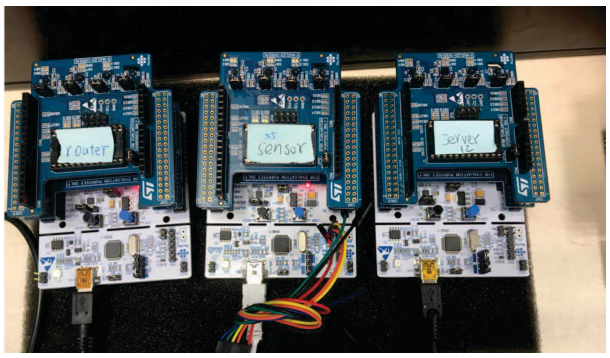

Figure 3. Experimental devices.

TABLE 1. HARDWARE AND SOFTWARE PARAMETERS

STM32 Nucleo-64 Board	MCU	NUCLEO-L152RE	Processor	ARM Cortex-M3 32 MHZ
			RAM	80 KB SRAM
			ROM	512 KB
			OS	Contiki 3.1
	Extension board	X-NUCLEO-IDS01A5	Frequency	ISM 915 MHz
		X-NUCLEO-IKS01A1	Sensors	LSM6DS0 (Acceleration and gyro-scope) HTS221 (Temperature and humidity) LIS3MDL (Magnetometer) LPS25HB (Pressure)
DELL OPTIPLEX 9020 (PC)			CPU	3.59 GHz Intel Core i7
			Memory	8 GB DDR3
			Disk	500 GB SATA HDD
			Network	1 Gbps
			OS	Ubuntu 16.04

enabled by adding X-NUCLEO-IDS01A5 RFID module and a MEMS environment sensor module. The detailed hardware parameters of the experimental environment are shown in Table 1.

We write applications with same functions using both C language and Lua-lite script language respectively and compare their performance in terms of memory usage, MCU load and response delay and the easy-to-use feature. These two programs are deployed on device to provide service of getting temperature value.

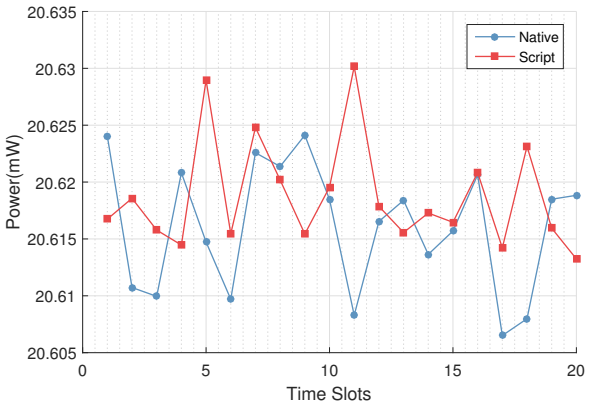

Figure 4. Comparison of average (20 times, 100 HTTP GET request per time) MCU power load between C program and the Lua-lite script for fetching temperature from a device.

4.2. Memory Usage

The memory usage of each module of the T-Suite on the experimental platform is shown in Table 2. T-Suite requires 33 KB of ROM and 3 KB of RAM in total. Its core components, including a tiny web server, native engine and Lua-lite script engine, only occupy 21 KB ROM and 1.5 KB of RAM. The results indicate that it has a very small footprint and can be run on various resource constrained devices.

TABLE 2. MEMORY USAGE OF EACH MODULE OF THE T-SUITE RUNTIME ENVIRONMENT

	ROM (Byte)	RAM (Byte)
Tiny Web Serve	11954	1534
Native Engine	968	33
Script Engine	20250	1464
Total	33172	3031

4.3. MCU Power Load

Since it is difficult to measure the CPU cycles directly, we examine the increment of MCU power load of applications developed with T-Suite instead. We launch the 'energest' module in the Contiki operating system to print the MCU power load per second using a serial port during the clinet on a PC continuously visiting a device for getting the temperature. The experiment is conducted for 20 times, and send 100 HTTP GET requests each time to the device for fetching the raw temperature value. Finally, the average MCU power load is calculated for each test and is shown in Figure 4. The average MCU power load for the C program about 20.616 mW for 100 HTTP requests, and is 20.619 mW for the Lua-lite script program. In other words, the script application only brings 0.014% increment of MCU power load in this test.

4.4. Response Latency of HTTP Request

The response latencies of C program and Lua-lite script are compared in this section. Both of these two programs

provide temperature reading service on a device and a client requests the temperature value for 100 times from a PC and the response delay of the requests is recorded. The latency is obviously increased and various due to the limited computational resources of things devices and the distributed architecture of edge computing systems. The latency lowers the quality and stability of services and increases the uncertainty of applications. Assume the maximum tolerance latency time of a single request is T. We randomly choose N samples from the requests of the client and denote their response time as T1, T2, ..., Tn. There are k times of requests that can return within the maximum tolerance latency time (Tk < T), then we define the average latency time of a single request as:

$$Lat_{Avg} = \frac{\sum_{i=1}^{i=k} T_i}{N}$$

And we adopt Standard Deviation (SD) to evaluate the variation of the latency:

$$SD_{Var} = \sqrt{\frac{1}{N}\sum_{i=0}^{i=N}(T_i - Lat_{Avg})^2}$$

Figure 5 shows the response latencies of 100 requests in these two applications. We can conclude that the latency of the C program is lower than Lua script program a little. The Lat_{Avg} of the service provided by the C program is 101.90 ms with a SD_{Var} of 2.75 ms; while for Lua script program, they are 105.06 ms and 3.81 ms. This experiment shows that T-Suite only increases the latency of service by only 3.10% and latency variation by 38.55% by introducing script application environment with the layered architecture.

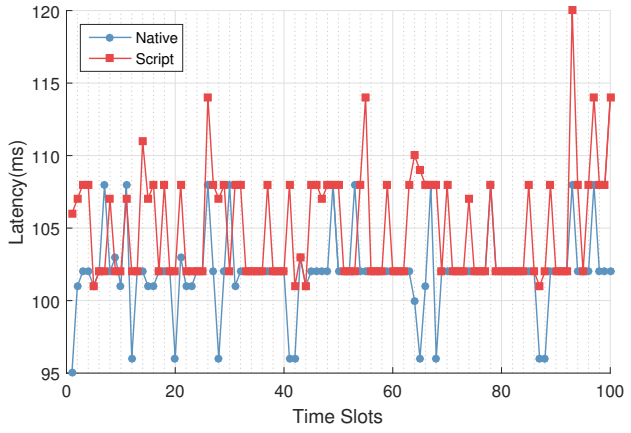

Figure 5. Comparison of response latency between services provided C program and the Lua-lite script for fetching temperature value from a device.

4.5. Programming Friendliness

Lines of code (LOC) is a key index for evaluating user-friendliness of a programming language. We still use the two programs that written in C and Lua-lite respectively for fetching the temperature from a device to compare their LOCs. Both of them contain the drivers written in C, but the difference is the upper layer. The C program writes the all the business logic in C code which is more difficult for developers. Lua-lite program divides the upper layer into the native part and the script part. The native part is written in C by invoking standard interfaces provided by the native engine, while the script part is written in Lua-lite and interpreted in Lua-lite script engine. As shown in Table 3, the number of lines of code for the upper layer decreases from 21 in C to 10 in Lua-lite. It indicates that Lua-lite is more easy-to-use than native C when programming on a device. In addition, it brings a very little increment of system footprint as shown in Table 2.

TABLE 3. LINES OF CODE FOR C AND LUA-LITE

	Component	Programming Language	Lines of Code
C Program	Drivers	C	363
	C code	C	21
Lua-lite APP	Drivers	C	363
	Native Part	C	8
	Script Part	Lua-lite	2

In this section, we evaluated the performance of the T-Suite prototype system by comparing memory usage, CPU load, response latency and easy-to-use feature with native C programs. The developers need to be equipped with professional skills and experience in developing traditional embedding applications. While T-Suite extends the Web architecture to things devices and enables scripting function for things devices which greatly eases the application development and deployment in edge computing systems.

5. Related Work

A key factor that expediting proliferation of IoT is the ability of quick creating, testing and deploying applications on the heterogeneous connected devices [10], [11]. IoT.js [7] is a state-of-the-art application framework for IoT devices. It adopts JerryScript [12] to interpret and execute JavaScript applications. The compiled package size of IoT.js with JerryScript engine is about 500 KB [7]. Thus, IoT.js cannot run on most of the things devices whose RAM and ROM is extremely small. Arduino [13] is a popular platform for developing innovative applications on IoT devices. However, it is still luxurious for most of the resource constrained devices. Mbed [14] is another IoT platform designed by ARM. It includes the operating system, cloud services, tools to make the development of large-scale IoT solutions easy. A key feature is that Mbed provides an online IDE that is accessible for the boards through a web browser without installing any additional software.

Steve Hodges et al. designed an IoT device prototyping tool, named Microsoft .NET Gadgeteer which is a modular platform consisting of a CPU and several sockets [10]. They argued that the restructural ability of their platform

allows quick constructing, re-configuring, and extending the prototype of new devices. Han-Chuan Hsieh et al. proposed an agent-based script framework, named ScriptIoT [15], for the Internet of Things applications. Their framework enables ordinary users who have little knowledge of programming to develop IoT applications easily. Matthias Kovatsch et al. proposed a RESTful architecture which provides "Web-like scripting for low-end devices" [16]. They designed a runtime container, named actinium, to provide configuration management functions for scripts via RESTful APIs. Some researchers discussed the challenges of debugging Wireless Sensor Network applications [17] [6]. Takuro Yamamoto et al. extended mRuby [18] for their component-based development framework to improve the efficiency of Cyber-Physical-Systems. They claimed that the performance of scripts executed in their framework was same as those of C language.

However, the dynamic resource-constrained environment poses a great challenge for developing and debugging programs on things devices and existing tool chains for powerful traditional systems are not suitable. The current protocol stack is too heavy for these devices. In addition, the distributed environment causes many uncertainties [19] when developing and debugging applications which usually involves multiple distributed devices. In this paper, we aim at extending the web to the resource-constrained environment in which devices are usually designed using a micro controller.

6. Conclusion and Future Work

In this paper, we design and implement T-Suite which includes a RESTful distributed Development and Deploy Environment(RDDE) and the Mixed Runtime Environment(MRE) for things devices. It enables programmers to develop and debug scripts on MCU-based things devices efficiently. A mixed runtime environment and a RESTful debugger are designed to ease the application development procedure and improve the execution efficiency on things devices. Developers do not have to establish a complex and fragile cross-compile environment for creating, debugging and deploying their applications as traditional embedded systems did. Experimental results indicate that we enabled script programming on things devices with very little increment of computing resource consumption and only a few side effects (memory occupy: 36 KB, latency increment: 3.10%, MCU power load increment: 0.014%). In addition, the code written in Lua-lite is far less than that written in C language for the same function.

The initial practice of layered application architecture in T-Suite shows that it can improve the efficiency of applications on things devices with little overhead but ease the programming procedure greatly by introducing a mixed runtime environment. However, there are still a lot of open issues should be addressed in the future. First and most important is a unified lightweight script language which supports Web and artificial intelligence for things devices. The applications developed by the script language should be decomposable and schedulable on heterogeneous devices to further improve the system efficiency. In addition, a more efficient native engine is needed to improve the portability of the script programs on things devices in advance.

References

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu, Edge computing: Vision and challenges, IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637646, 2016

[2] Chiang Mung, and Tao Zhang, Fog and IoT: An Overview of Research Opportunities, IEEE Internet of Things Journal, vol. 3, no.6, pp. 854-864, 2017

[3] Tencent Inc. (2018, April 7) WeChat. http://weixin.qq.com/

[4] Zhiwei Xu, Xiaohui Peng, Lei Zhang, et al., The Φ-stack for smart web of things, in Proc of the Workshop on Smart Internet of Things (Smart IoT), in conjunction with SEC 2017, San jose CA, Oct. 2017, Article No. 10

[5] R. T. Fielding and R. N. Taylor, Principled Design of the Modern Web Architecture, In Proc of the 22nd International Conference on on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000, pp. 407-416.

[6] A. Shahshahani, A. Tolstikhin, and Z. Zilic, Enabling Debug in IoT Wireless Development and Deployment with Security Considerations, In 2016 IEEE 25th North Atlantic Test Workshop (NATW), 2016, pp. 5358.

[7] SAMSUNG Inc. (2018, April 7) IoT.js. http://iotjs.net

[8] Lua programming language. (2018, April 7). Networks [Online]. Available: http://www.lua.org/

[9] B. Carsten, A. P. Castellani, and Z. Shelby, CoAP: An Application Protocol for Billions of Tiny Internet Nodes, IEEE Internet Computing, vol. 16, no. 2, pp. 62-67, 2012.

[10] S. Hodges, et al., Prototyping Connected Devices for the Internet of Things, Computer, vol. 46, no. 2, pp. 26-34, 2013.

[11] S. Mora et al, RapIoT toolkit: Rapid prototyping of collaborative Internet of Things applications, Future Generation Computer Systems, 2018. Available: https://doi.org/10.1016/j.future.2018.02.030.

[12] E Gavrin et al., Ultra lightweight JavaScript engine for internet of things, In Proc of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity ACM, 2015, pp. 19-20.

[13] M. Banzi, and M. Shiloh, Getting started with Arduino: the open source electronics prototyping platform, San Francisco, CA: Maker Media Inc., 2015.

[14] ARM Inc. (2018, April 7). Mbed OS 5 [Online]. Available: https://os.mbed.com/

[15] H.C. Hsieh et al. ScriptIoT: A Script Framework for and Internet-of-Things Applications, IEEE Internet of Things Journal, vol. 3, no. 4, pp. 628-636, 2016.

[16] M. Kovatsch, M. Lanter, and S. Duquennoy, Actinium: A RESTful runtime container for scriptable Internet of Things applications, In Proc of 2012 3rd International Conference on the Internet of Things, Wuxi China, 2012, pp. 135-142.

[17] P. Eugster, V. Sundaram, and X. Zhang, Debugging the Internet of Things: The Case of Wireless Sensor Networks, IEEE Software, vol. 32, no. 1, pp. 38-49, 2015.

[18] T. Yamamoto et al, Lightweight ruby framework for improving embedded software efficiency, In Proc of IEEE International Conference on Cyber-Physical Systems, Networks, and Applications, Nagoya, 2016, pp.71-76.

[19] F. Ramparany et al, A Semantic Approach for Managing Trust and Uncertainty in Distributed Systems Environments, In Proc of 2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS), Dubai, Nov. 2016, pp.63-70.