In Situ Mutation for Active Things in the IoT Context

Noura Faci[1], Zakaria Maamar[2], Thar Baker[3], Emir Ugljanin[4] and Mohamed Sellami[5]
[1]*Université Lyon 1, Lyon, France*
[2]*Zayed University, Dubai, U.A.E.*
[3]*Liverpool John Moores University, Liverpool, U.K.*
[4]*State University of Novi Pazar, Novi Pazar, Serbia*
[5]*ISEP Paris, Paris, France*

Keywords: IoT, Lua, Mutation, NodeMCU, and Policy.

Abstract: This paper discusses mutation as a new way for making things, in the context of Internet-of-Things (IoT), active instead of being passive as reported in the ICT literature. IoT is gaining momentum among ICT practitioners who see a lot of benefits in using things to support users have access to and control over their surroundings. However, things are still confined into the limited role of data suppliers. The approach proposed in this paper advocates for 2 types of mutation, active and passive, along with a set of policies that either back or deny mutation based on specific "stopovers" referred to as permission, prohibition, dispensation, and obligation. A testbed and a set of experiments demonstrating the technical feasibility of the mutation approach, are also presented in the paper. The testbed uses NodeMCU firmware and Lua script interpreter.

1 INTRODUCTION

Internet of Things (IoT) is gaining momentum among ICT practitioners who see a lot of benefits in the role that things could play in allowing users to have access to and control over their surroundings. Different figures and reports back this momentum. For instance, a Gartner report states that 6.4 billion connected things were in use in 2016, up 3% from 2015, and will reach 20.8 billion by 2020[1]. Moreover, McKinsey mentions that "*The market for Internet of Things devices, products, and services appears to be accelerating in view of four critical indicators: supplier attention, technological advances, increasing demand, and emerging standards*" (Bauer et al., 2017). Despite the bright side of IoT (sometimes mixed with a lot of hype), IoT raises many concerns that could refrain its expansion and adoption in the future. A concern, that we deem worth addressing, is that things are still passive being restricted to sensing the surroundings and sharing the outcomes of this sensing (sometimes after processing/actuating) with third parties. A DZone group's 2017 report (DZone, 2017) along with Mzahm et al. (Mzahm et al., 2013) highlight the passive nature of things, which does not help develop a dynamic ecosystem of active things.

In this paper, we propose ways of making things active. The first way is about agentifying things whose details are given in (Maamar et al., 2017). The second way, which is this work's aim, is about thing mutation in the sense that things will bind and/or unbind capabilities on the fly (and as they see fit). To ensure a successful mutation, we consider first, the context (i.e., surrounding) in which things operate and second, the policies that impact the decisions of things to bind/unbind capabilities. For the sake of setting-up a dynamic ecosystem of active things, we motivate mutation decisions with 3 reasons: *performance* so, that, a thing remains competitive/attractive, *adaptation* so, that, a thing remains responsive, and *survivability* so, that, a thing remains in business.

In support of the aforementioned reasons, we develop policies that will "steer" the mutation through specific "stopovers": *permission* for a thing to mutate when all necessary and sufficient contextual conditions are satisfied, *prohibition* for a thing to mutate when all necessary and sufficient contextual conditions are unsatisfied, *dispensation* for a thing to mutate/not to mutate (despite the permission/prohibition) due to changes that made certain necessary and sufficient contextual conditions unsatisfied/satisfied, and *obligation* for a thing to mutate (despite either the no-permission or the prohibition) due to changes that made certain necessary and sufficient contextual conditions satisfied. Contextual conditions, that reflect changes in a thing's surrounding, result from (*i*) acti-

[1] www.gartner.com/newsroom/id/3165317.

Faci, N., Maamar, Z., Baker, T., Ugljanin, E. and Sellami, M.
In Situ Mutation for Active Things in the IoT Context.
DOI: 10.5220/0006840607250732
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 725-732
ISBN: 978-989-758-320-9

ons that a thing (itself) takes, (*ii*) actions that an owner makes her thing take, (*iii*) actions that other things take, and (*iv*) interactions that a thing has with users. The first 2 points fall into a thing's inner-control and the last 2 fall into a thing's outer-control.

Our contributions include (*i*) definition of mutation in an IoT context, (*ii*) identification of reasons that support thing mutation, (*iii*) specification of policies for approving/denying thing mutation, (*iv*) tracking of the mutation process's approval/denial through "stopovers", and (*v*) a testbed for thing mutation. The rest of this paper is organized as follows. Section 2 is an overview of thing mutation in the literature. Section 3 presents our thing mutation approach in terms of thing's lifecycle and policies that either approve or deny thing mutation. Section 4 presents the mutation testbed along with some experiments. Concluding remarks and future work are presented in Section 5.

2 RELATED WORK

Despite the growing interest in IoT, our literature review revealed, to the best of our knowledge, the limited number of references that tackle the challenge of thing mutation. Prior to proceeding with the literature review, we begin with some definitions from the field of genetics. In (NLM,), a gene mutation is a permanent alteration in the DeoxyriboNucleic Acid (DNA) sequence that makes up a gene. Moreover, mutation can affect anything from a single DNA building block to a large segment of a chromosome that includes multiple genes.

Back to ICT field, Bölöni and Marinescu propose a formal description of mutability in multiagent systems (Bölöni and Marinescu, 2005). This description is about a strategy that consists of planes (each referring to as a set of intended actions) that deal with different parts of the world. The planes are scheduled in a way that only one would be active at once. To model the agent's behavior, the authors use finite state machines where a state corresponds to a multi-plane strategy and a transition refers to some multi-plane strategy change. The authors formally define a set of mutation operators (e.g., add a state to the agent behavior and add a transition between 2 states) on the multi-plane state machines.

Raner (Raner, 2006) discusses the mutator pattern as a simple way of applying a series of successive changes to a mutable object instead of successively creating new object instances that would cater to these changes. Though Raner does not explicitly define what a mutable object is, he recommends a set of cases where the mutator pattern would be appropriate such as applying an algorithm on a sequence of complex objects whose individual creation is rather expensive and creating objects, who do not exist yet, on-the-fly. Benefits of the mutator pattern include saving time by eliminating the repetitive creation of objects and saving memory by using a single mutable object. Contrarily, drawbacks of the pattern include the necessity of having mutable objects that could be complex to handle compared to immutable objects and the necessity of satisfying a good number of prerequisites that could limit its applicability.

Yun et al. (Yun et al., 2017) analyze mutation in the context of testing policies in a system of systems. This latter is a set of constituent systems that are forced, thanks to policies (predefined rules), to collaborate when goals cannot be achieved individually. Obstacles called faults by Yun et al. could arise at the system of systems level but not at the system constituent level calling for a mutation analysis that would tackle these obstacles. This analysis is a systematic way of evaluating test cases using artificial faults called mutants and is demonstrated with a traffic management case-study. According to Yun et al., *"mutation testing is a fault-based testing technique proposed in 1970s by Lipton (Lipton, 1971) and developed by DeMillo (Lipton et al., 1978). It originated from the idea that if a test case can detect an artificially seeded fault, the test case also can detect a real fault"*. The program that receives a seeded fault is called mutant and the rules for injecting this fault into the program are called mutation operators. Finally, if the outcome of executing a mutant is different from that of the original program for a test case, it is said that the mutant is killed by the test case.

In line with Yun et al. (Yun et al., 2017), Polo Usaola et al. (Polo Usaola et al., 2017) analyze software testing using mutation operators. This software is about context-aware, mobile applications that feature errors/faults. Mutation operators insert faults into a system like those that programmers would intentionally introduce in their system.

Similar to mutation, Terdjimi et al. use adaptation to discuss the changes that affect behaviors of avatars in the Web of things (Terdjimi et al., 2017). They consider avatar as a virtual extension of a thing that relies on a semantic architecture so, that, it processes and reasons about semantically-annotated information. Triggers of changes are due to non-functional concerns like quality of service, energy efficiency, and security related to natural conditions, computing resources, and user preferences. To ensure a successful adaptation, Terdjimi et al. raise a couple of questions that they address in their work, for instance, "*which* *protocols should the application use to communicate with things, which thing capability should be involved in a given terminal functionality*", and "*which functionality should be exposed to clients and other avatars*?" The adaptation is exemplified with watering a vineyard in which drones acting as avatars take photos of the field to identify the parts that are dry, for example, and hence, need to be watered. Weather forecast details are, also, taken into account during the watering decision.

As stated in the first paragraph, thing mutation in the context of IoT remains "undiscovered" and hence, many questions are unaddressed from different perspectives such as technical, legal, and "ethical".

3 OUR MUTATION APPROACH

Some argue that things are not prepared, yet, to take the mutation leap due to multiple technical constraints. Contrarily, Taivalsaari and Mikkonen mention that "*hardware advances and the availability of powerful but inexpensive integrated chips will make it possible to embed connectivity and fully edged virtual machines and dynamic language run-times everywhere*" (Taivalsaari and Mikkonen, 2017).

3.1 Mutation Process as a Lifecycle

Prior to defining the lifecycle of a mutable thing, we deem necessary discussing mutation in terms of *type* (weak *versus* strong), *mode* (active *versus* passive), *impact* (on thing itself *versus* on capability), and *initiator* (thing itself *versus* thing's owner *versus* thing's peers). Because of the simplicity of last 2 points, we only explain the first 2.

1. Weak mutation means that the thing still complies with the owner's original specification after mutation. Contrarily, strong mutation means that the thing's specification radically changes. Simply put, weak mutation leads to a similar thing while strong mutation leads to a new thing.
2. Active mutation means that the thing/capabilities continue to operate/be used while mutation is taking place. Contrarily, passive mutation requires putting on standby/suspending the thing/ongoing capabilities and then activating/resuming it/them after mutation.

To concretize mutation, many actions could be taken reflecting the impact of mutation on a thing's capability and/or thing itself. These actions are, but not limited to, as follows:

- Unbind/bind a capability means unloading/loading the capability. An example is to upload an existing capability following the disposal of a peer from the ecosystem that used to offer this capability.
- Split a thing means decomposing the thing into different things. An example is to create more things that will be assigned (some) separate capabilities initially linked to an existing thing (will retain some capabilities). The creation could be due to the arrival of extra requests.
- Merge things means composing things along with their respective capabilities into a single thing. An example is to group things into one due to scarcity of resources.

Fig. 1 represents the lifecycle of a mutable thing represented as a statechart. On the one hand, states include not-activated (i.e., the mutant waits for certain conditions to be satisfied), activated (i.e., the mutant enables necessary capabilities), done (i.e., the mutant successfully completes the enabled capabilities[2]), and mutated passively (i.e., the mutant performs some mutation action). On the other hand, transitions between states include *initial operation* (i.e., handling requests), *suspension* (i.e., suspending ongoing capabilities in preparation of mutation), *resumption* (i.e., resuming ongoing capabilities after mutation), *active mutation* (i.e., performing some mutation action), *completion* (i.e., finalizing the enabled capabilities), *extra-operation* (i.e., performing some additional mutation action), and *final completion* (i.e., confirming the release of capabilities). Note that *mutated passively* along with *suspension* and *resumption* correspond to the passive mutation and that *activated* along with *active mutation* correspond to the active mutation.

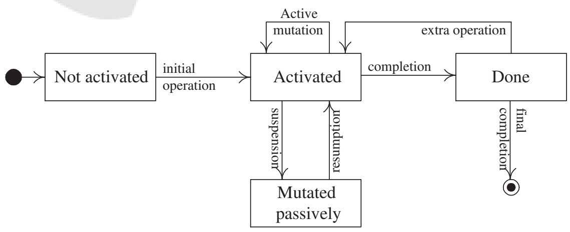

Figure 1: Lifecycle of a mutable thing as a statechart.

After an initial operation (not activated $\xrightarrow{initial\ operation}$ activated) and a regular completion of capabilities (activated $\xrightarrow{completion}$ done $\xrightarrow{final\ completion}$ end), 3 cases could arise illustrating mutation:

[2]For the sake of simplicity, capability failure is not handled.

- The mutant puts on hold the enabled capabilities in preparation of mutation: activated $\xrightarrow{suspension}$ mutated passively $\xrightarrow{resumption}$ activated.
- The mutant proceeds with the enabled capabilities during mutation: activated $\xrightarrow{active\ mutation}$ activated
- The mutant successfully completes the enabled capabilities and decides on new an extra mutation: done $\xrightarrow{extra\ operation}$ activated.

The 3 cases could be connected together leading to a chain of mutation actions in response to certain detected events and/or received requests. We back this chain of mutation with policies that oversee the mutation progress from one state to another in the mutation's lifecycle.

3.2 Mutation Decisions as Policies

We rely on policies to "steer" the decision making process that would lead to either approve or deny thing mutation. For a proper "steering", we associate the progress of this process with 5 stopovers (Fig. 2): permission (*pe*) to mutate, prohibition (*pr*) to mutate, dispensation (*d*) (specialized into dispensation to-not-mutate despite permission (d_{pe}; e.g., too risky and too costly) and dispensation to-mutate despite prohibition (d_{pr}, e.g., too rewarding)), and obligation (*ob*) to mutate. Moving from one stopover to another depends on assessing the sufficient and/or necessary contextual conditions that could change due to things' actions, owners' decisions, peers' actions, and things' interactions with users.

Below is the connection between the stopovers that would lead to mutation approval (where y/n stands for yes/no; not all connections are shown due to lack of space):

1. $pe(\text{y}) \rightarrow d_{pe}(\text{n}) \rightarrow pr(\text{n})$. The sufficient and necessary contextual conditions that led to approving the mutation did not change over time so there was neither a dispensation from mutating nor a prohibition to mutate.

Below is the connection between the stopovers that would lead to mutation denial (not all connections are shown due to lack of space):

1. $pe(\text{n}) \rightarrow ob(\text{n})$. The sufficient and necessary contextual conditions that led to denying the mutation did not change over time so there was no obligation to mutate.
2. $pe(\text{y}) \rightarrow d_{pe}(\text{y}) \rightarrow ob(\text{n})$. Some sufficient and necessary contextual conditions that led to approving the mutation have become unsatisfied leading to dispensing the mutation. In addition, this dispensation was supported by an obligation of to-not-mutate due to changes in these and may be other sufficient and necessary conditions.

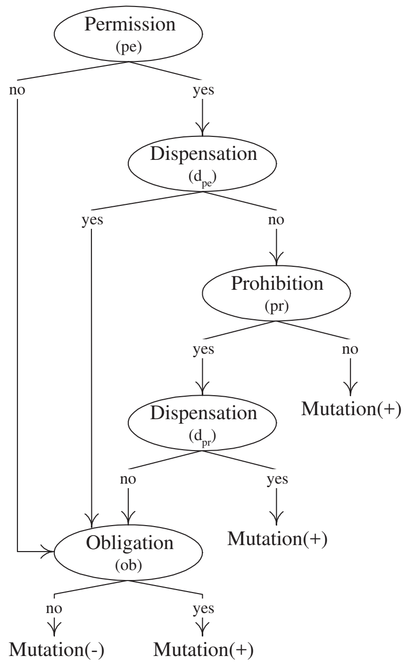

Figure 2: Approval(+) *versus* Denial(-) of thing mutation.

4 MUTATION TESTBED

This section presents the testbed demonstrating the technical feasibility of thing mutation and discusses, afterwards, some experiments in support of this feasibility.

4.1 Testbed Architecture

Building upon an open-source project[3] for *OTA Web management & esp8266 Lua client for Over-the-Air* (OTA)[4] script update, our testbed corresponds to a mutation control application for managing things that could mutate according to the different actions listed in Table 1. For the time being, only "reconfigure thing" and "reconfigure capability" actions are implemented and tested. This testbed's architecture is represented in Fig. 3 where the numbers correspond to the chronology of operations.

The control application consists of the following in-house developed components:

[3]github.com/kovi44/NODEMCU-LUA-OTA-ESP8266.
[4]en.wikipedia.org/wiki/Over-the-air-programming.

Table 1: Examples of trigger-action per mutation pattern.

		Actions to take (✓ for applicable)				
		On thing			On thing's capability	
	Triggers	split	merge	reconfigure	bind/unbind	reconfigure
Group_1	Handling of "unseen" demands (e.g., request to sense body temperature on top of ambient temperature)			✓		✓
	Increase in workload (e.g., reception of extra requests)		✓	✓		
	Adjusting quality of service (e.g., changes in ecosystem conditions)	✓	✓		✓	✓
Group_2	Unexpected arrival of new things (e.g., forming ad-hoc partnerships)		✓	✓	✓	✓
	Disposing existing things (e.g., contacting partners of disposed things)	✓		✓	✓	✓
	Securing more marketshare (e.g., changes in ecosystem conditions)			✓	✓	

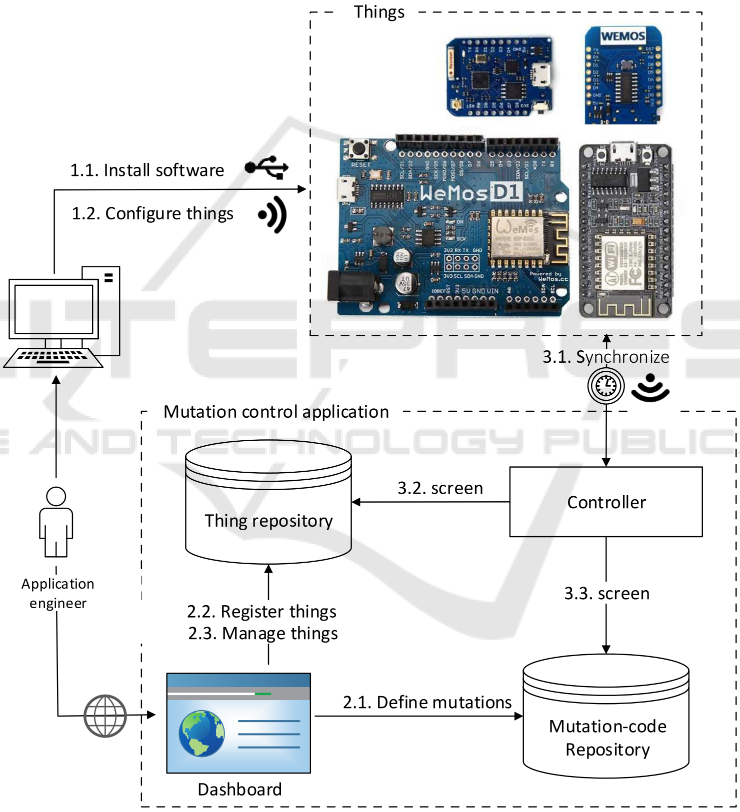

Figure 3: Architecture of thing-mutation testbed.

1. *Dashboard* that allows the engineer to register things (referred to as devices in the below) in the testbed so, that, she can access and configure them. The dashboard also enables the engineer to develop mutation actions (referred to as scripts in the below) such as reconfigure, split, and merge (Table 1).
2. *Mutation-code repository* that stores the developed scripts along with their identifiers.
3. *Thing repository* that stores details on devices such as manufacturer unique-chipID, active mutationID that refers to the current mutation action's identifier in the mutation-code repository, and update flag that lets a device know if it has been approved for mutation by the engineer (in compliance with the outer-control mutation decision, Section 3.2).
4. *Controller* that supports the interactions between devices and the mutation application. These in-

teractions take place wirelessly, i.e., OTA using REST.

In preparation for thing mutation, some work needs to be completed as per the following 2 steps:

1. First, the engineer installs from scratch (and sometimes customizes[5]), using certain tools such as ESPlorer[6] for uploading scripts and esptool.py[7] for flashing firmware, some required software on devices (1.1). This software includes a firmware (NodeMCU[8] in our testbed) and a standalone script interpreter (Lua[9] in our testbed). On the one hand, NodeMCU firmware supports communication protocols (e.g., MQTT (Message Queuing Telemetry Transport), HTTP (Hyper Text Transfer Protocol), and COAP (Constrained Application Protocol)) with third parties and includes some built-in functions (e.g., file management, GPIO (General Purpose Input/Output) usage, and SJSON (Simplified JSON parser)). The engineer selects the appropriate modules (e.g., MQTT) for implementing the mutation scripts when building the firmware. On the other hand, Lua script interpreter is used for synchronizing devices with the controller prior to hot-plugging[10] (Baker et al., 2013) scripts, and interprets the new scripts after being fully downloaded to the things. More details on NodeMCU firmware's modules are available at nodemcu-build.com..
2. Second, the engineer configures each device (1.2) separately so, that, it communicates with the control application. After uploading the necessary software onto the device as per the previous step, the device is rebooted in the HTTP server mode and proceeds with broadcasting its WiFi access point. When the engineer connects to the same access point, she accesses the device's configuration panel so, that, necessary parameters are setup such as wireless network name/password, panel access details (e.g., server IP, domain name, and script path), and synchronization time. Upon completing the configuration, the device restarts and synchronizes with the controller checking if it is subject to any mutation specified by the engineer (using a flag).

[5]In the case of customization, the engineer must flash the device with fresh firmware containing the desired modules that are downloaded from NodeMCU cloud build tool.

[6]esp8266.ru/esplorer.

[7]nodemcu.readthedocs.io/en/master/en/flash.

[8]github.com/nodemcu/nodemcu-firmware.

[9]NodeMCU firmware is based on Lua. But, other options, such as PJON (github.com/gioblu/PJON.) and ModuleInterface (github.com/fredilarsen/ModuleInterface.), are available subject to the used firmware.

[10]Hot-plugging means download, interpret, and reboot (MicroTCA and Specification,).

4.2 Testbed Operation

The mutation control application is a Web application, hosted on a Linux Apache server, developed in PHP, JavaScript, HTML, and CSS, and uses MySQL database. The application allows the engineer to add/drop devices to/from the testbed whenever necessary using add/delete buttons, describe existing and/or new devices (i.e., adding a new name, narration/commentary, and chipID, which acts as an identifier) using the edit button, and to develop scripts (2.1) that will be linked to devices.

To run the testbed, the engineer registers the devices (2.2) in the thing repository using the dashboard and proceeds with developing the necessary scripts in Lua. The devices that exemplify things in our testbed are equipped with an ESP8266 chip and have at least 4MB flash. This minimum flash requirement guarantees 1MB space for the NodeMCU firmware. The remaining space permits to store the interpreted mutation scripts that are available for execution.

The tested devices include WemosD1, WemosD1 mini, and NodeMCU. These are microcontrollers equipped with wireless modules for communicating with third parties like sensors and computers utilizing protocols included in their firmwares. The engineer also manages (2.2) the devices that will be subject to mutation in compliance with the outer-control mutation decision. In term of managing devices (2.2), the engineer could consider different devices for mutation and different mutation scripts, as she sees fit. Afterwards, the devices periodically send requests to the controller to check whether there is some update. As a result of these periodic requests, the controller screens the thing repository to verify whether the device is listed/known and its corresponding mutation flag (true/-false) is raised. If this is the case, the controller looks for the corresponding script in the mutation-code repository so the appropriate script is sent to the device. This one hot-plugs the script after uploading 4 files: *init.lua* (a file loaded every time the device boots itself and makes a decision should it boot in HTTP server mode (via server.lua) or with mutation script (via *client.lua*)), *server.lua* (for starting up the HTTP server when the device is booted for first time), *client.lua* (for synchronization with the control application and interpreting new scripts), and *config.htm* (html configuration form for storing parameters, where these parameters are stored in a separate file hosted by the device).

To further elaborate the mutation procedure, Algorithm 1 is the pseudocode for *init.lua* file, at the thing/device end, as/when the device reboots. It starts by creating an object *s* and loading configuration parameters' values (e.g., *id*, *pwd*, and *boot*) from the device configuration file (lines 1 & 2, respectively). It should be noted that if the device was booted for the first time, the configuration file would have not been created yet; consequently the value of *s.host* parameter would have been empty as in (line 2). In this case, the device loads *server.lua* file (line 14), which is in charge for booting the device in a HTTP server mode, where it acts as an access point and allows the engineer to configure it. Otherwise, if the *s.host* parameter holds a value, it implies that the device has already been configured by the engineer and it is ready to get connected to WiFi (line 4). The device, then, checks if there is an update waiting in a defined time interval, as in (line 5 and 6) via calling *checkForUpdate* function. The later triggers the server to check the update flag, for that particular device, at the server side (Listing 1). The server identifies the requesting device along with its corresponding flag and associated mutation code (if exist) via using the device *id* (i.e., *id=chipid*, Listing 1). If *update = true* for that device, it implies new mutation code exist, hence the server will release the update, and hand the control back to the device. Back to the thing side, if a new mutation script is downloaded and compiled, *s.boot* parameter will not be empty (line 8) and the device will boot the compiled script, as per (line 9). Contrarily, if *s.boot* parameter is empty, it will load *client.lua* file, which will download a new mutation code, compile it, alter *s.boot* parameter and reboot device.

Algorithm 1: Runtime thing mutation via *init.lua*.

1: s = {ssid=" ", pwd=" ", host=" ", path=" ", boot=" ", update = 0};
2: s = ReadConfiguration();
3: **if** ($s.host \neq$ " ") **then**
4:     connectToWiFi();
5:     **if** ($s.update$) ≥ 1 **then**
6:         timer (s.update, function() checkForUpdate() end);
7:     **end if**
8:     **if** ($s.boot \neq$ " ") **then**
9:         dofile(s.boot);
10:     **else**
11:         dofile("client.lua");
12:     **end if**
13: **else**
14:     dofile("server.lua");
15: **end if**

Listing 1: Checking update at server side.

```
<?php
$result = mysqli_query($conn, "SELECT*
FROM esp WHERE id='$_GET['chipid']'");
    $fetch = mysqli_fetch_assoc( $result );
    if ($fetch [update]== true){
      makeUpdateAvailable ();
      $toupdate = mysql_query ("UPDATE
      'esp' SET 'update'=false , timestamp=
      now() WHERE id='$_GET['chipid']'");}?>
```

5 CONCLUSION

This paper presents a novel way (backed by a test-bed along with all its associated technologies such as NodeMCU and Lua, and components such as a Web-based mutation control application) to mutate things, in the context of Internet-of-Things (IoT). Mutation types, capabilities, and policies for different mutation actions are also discussed in this paper. However, for the time being, only 2 mutation actions, namely "reconfigure thing" and "reconfigure capability", are implemented and tested. For runtime mutation (i.e., new code injection), Lua script interpreter has been used for synchronizing devices with the mutation control application to hot-plugging new code. The proposed way proves that things can be active rather than passive, compared to what has been previously stated in the literature, by mutating things according to different actions/triggers (Table 1). In addition, things can provide various behaviors based on their technical capabilities in terms of hardware and software.

As future work, we seek to implement additional actions listed in Table 1 such as splitting and merging things. We also seek to define patterns that would offer better understanding of when mutate (e.g., secure more marketshare and reduce resource cost) and ensure mutation consistency across available thing platforms. Finally, we seek to investigate the benefits of mutation in developing a safer IoT. Mutation could be the way for protecting things from threats and attacks.

REFERENCES

Baker, T., Mackay, M., Randles, M., and Taleb-Bendiab, A. (2013). Intention-oriented programming support for runtime adaptive autonomic cloud-based applications. *Computers and Electrical Engineering*, 39:2400–2412.

Bauer, H., Patel, M., and Veira, J. (http://www.mckinsey.com/industries/semiconductors/our-insights/the-internet-of-things-sizing-up-the-opportunity, 2014 (visited in August 2017)). The internet of things: Sizing up the opportunity. Technical report.

Bölöni, L. and Marinescu, D. (2005). *Adaptation and Mutation in Multi-Agent Systems and Beyond.* Springer Berlin Heidelberg.

DZone (https://dzone.com/guides/iot-applications-protocols-and-best-practices, 2017 (visited in May 2017)). The Internet of Things, Application, Protocls, and Best Practices. Technical report.

Lipton, R. (Carnegie Mellon University, 1971). Fault Diagnosis of Computer Programs. Technical report.

Lipton, R., DeMillo, R., and Sayward, F. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4).

Maamar, Z., Faci, N., Kallel, S., Sellami, M., and Ugljanin, E. (2017). Software Agents Meet Internet of Things. *Internet Technology Letters, Wiley*.

MicroTCA and Specification, O. M. C. Picmg mtca.4 pci express hot plug design guide.

Mzahm, A. M., Ahmad, M. S., and Tang, A. Y. C. (2013). Agents of Things (AoT): An intelligent operational concept of the Internet of Things (IoT). In *Proceedings of the 13th International Conference on Intellient Systems Design and Applications (ISDA'2013)*, Bangi, Malaysia.

NLM. NLM, US National Library of Medicine. https://www.nlm.nih.gov.

Polo Usaola, M., Rojas, G., Rodriguez, I., and Hernandez, S. (March 2017). An Architecture for the Development of Mutation Operators. In *Proceedings of the 12th International Workshop on Mutation Analysis (Mutation'2017) held in conjunction with the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'2017)*, Tokyo, Japan.

Raner, M. (October 2006). The Mutator Pattern. In *Proceedings of the 2006 Conference on Pattern Languages of Programs (PLoP'2006)*, Portland, OR, USA.

Taivalsaari, A. and Mikkonen, T. (2017). A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software*, 34(1).

Terdjimi, M., Médini, L., Mrissa, M., and Maleshkova, M. (2017). Multi-purpose Adaptation in the Web of Things. In *Proceedings of the 10th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT'2017)*, Paris, France.

Yun, W., Shin, D., and Bae, D. (May 2017). Mutation Analysis for System of Systems Policy Testing. In *Proceedings of the 2017 IEEE/ACM Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (JSOS@ICSE'2017)*, Buenos Aires, Argentina.