Web of Things System Continuous Integration

Muthuraman Chidambaram
Technical University of Munich (TUM)
Munich, Germany
muthuraman.chidambaram@tum.de

Ege Korkan
Technical University of Munich (TUM)
Munich, Germany
ege.korkan@tum.de

Sebastian Steinhorst
Technical University of Munich (TUM)
Munich, Germany
sebastian.steinhorst@tum.de

***Abstract*—Number of Internet of Things (IoT) devices are growing exponentially with the development of the smart city, connected car and industry 4.0. As of August 2019, there are 26 billion (and growing active) IoT devices in business and consumer space. But 42 billion connected IoT devices are required by the year 2025. We are supposed to produce 16 billion flawless connected devices in the next 5 years. On top of it, the major drawback of the IoT ecosystem is its highly fragmented nature. Even if infinite solutions are arising, in both the consumer and business market, most of them are incompatible and reflect the needs of particular use-cases and technologies. The World Wide Web Consortium Web of Things (W3C WoT) is intended to enable interoperability across IoT platforms and application domains. Though the deployment time is reduced by WoT, it is a problem of the tester to perform black-box testing on each Thing in the WoT ecosystem during the testing phase. This paper is applying the concept of Continuous Integration (CI) into WoT, by considering the WoT devices as contributors and WoT system as a single software project to speed up the production of IoT devices in WoT ecosystem.**

***Index Terms*—Internet of Things, Thing Description, Web of Things, System Testing**

I. Introduction

Internet of Things (IoT) is a system of interrelated computing devices like smart home appliances, smart medical and health-care appliances, smart wearables, and computer devices. Each device that is attached with a sensor and an actuator and can communicate via the Internet is an IoT device. The number of IoT devices are growing exponentially with the development of the smart city, connected car and industry 4.0. As of August 2019, there are 26 billion (and growing active) IoT devices in business and consumer space. But, the major drawback of the IoT ecosystem is its highly fragmented nature. Even if infinite solutions are arising, in both the consumer and business market, most of them are incompatible and reflect the needs of particular use-cases and technologies. As a consequence, IoT system architects are forced to either use a single system (thus limiting the number of available devices) or to face several compatibility issues. Moreover, different standards cannot leverage the device to device concepts, as they will always require an adaptation element to ensure interoperability. [1]

The World Wide Web Consortium (W3C) Web of Things (WoT) is intended to enable interoperability across IoT platforms and application domains. Overall, the goal of the WoT is to preserve and complement existing IoT standards and solutions. The WoT achieves interoperability through wider adoption of the Web principles, standardized Thing Description, metadata and semantic web technologies.

A Thing Description describes the metadata and interfaces of Things, where a Thing is an abstraction of a physical or virtual entity that provides interactions to and participates in the WoT. Thing Descriptions provide a set of interactions based on a small vocabulary that makes it possible both to integrate diverse devices and to allow diverse applications to interoperate. Thing Descriptions, by default, are encoded in a JSON format that also allows JSON-LD processing.

WoT can be easily integrated into the IoT platforms to enable horizontal business cases and applications. Apart from that, utilization of the Web Technology reduces testing time and deployment of RESTful web services [2]. Though the deployment time is reduced by WoT, t is a problem of the tester to perform black-box testing on each Thing in the WoT ecosystem during testing phase. As the testers are supposed to deploy it on each and every device and test it manually during the testing phase.

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.[3]

In this paper we are applying the concept of Continuous Integration (CI) into WoT by considering the WoT devices as contributors and WoT system as a Single software project to speed up the production of IoT devices in WoT ecosystem. First, we describe the State of the art of Continuous Integration in WoT. Second, we state the methods and materials used in solving the problem. Third, we explain our approach used in our system. Fourth, we share the results of our work. Finally, we state the future work in the field and give our conclusion to the solution.

II. State of the Art

Testing an IoT device is significantly different from testing a typical hosted software as the IoT devices are resource constrained devices [4]. Since end-to-end testing in IoT system involves physical devices, a complete automation testing suite usually cannot be deployed on the IoT device. IoT software components are strongly coupled with the hardware [5].

As stated earlier in this paper, each Thing in the WoT ecosystem has a Thing Description (TD) which describes the metadata and interfaces of Things, wherein a Thing is a set of smart embedded device which can sense and communicate with other devices via Internet. Thereby, this Thing Description a JSON object is further used for testing the Things in WoT ecosystem. We have a testbench tool, which can test a WoT

Thing by executing interactions automatically, based on its Thing Description. Each interaction is executed based on the Representational State Transfer (REST), is a software architectural style that defines a set of constraints to be used for creating Web services. The Application Programming Interfaces (APIs) are created on the server to allow the client to talk to Server. APIs that adhere to the REST architectural constraints are called RESTful APIs. But testbench is a tool which can test Thing Description of a Thing, when the Thing Description is provided to the tool.

Most commonly, in today's world IoT device software is tested with the help of simulators and emulators. Karlesky and Williams presented an approach [6] to test the software developed for embedded devices that are resource constrained. There are multiple benefits while using an emulator or simulator, Hardware and software can be developed parallelly as there is no dependency hardware for testing the software. An end to end test setup can be developed with the help of Continuous Integration and Continuous Development (CI/CD) tools like Jenkins, CircleCI and more. Therefore, this setup will be much faster than the normal testing setup. But the problem with the emulator is that, it is impossible with the current technology to emulate all possible sensors and actuators in the IoT device market. This problem is an extensive constraint for the developers. Therefore, this approach can't be used for our problem.

Adam Dunkels developed Cooja Simulator, A cross-layer java-based wireless sensor network simulator distributed with Contiki. It allows the simulation of different levels from physical to application layer, and also allows the emulation of the hardware of a set of sensor nodes. This system can simulate wireless communication medium and perform a complete white-box testing. This approach also shares the benefits and drawbacks of the approach of Karlesky and Williams.

Unit Testing for Wireless Sensor Networks is the paper written by Okola and Whitehouse. This paper presents a new unit testing framework that addresses this concern by defining tests as a single Python script that executes on the PC. This script automatically generates test programs, coordinates the inputs, and collects outputs from all nodes in the network. This approach allows validation of distributed state, while simultaneously reducing the amount of code needed to define a unit test. But this tool requires to know each variable and function and 1 line of code is required for each function or variable. Therefore, this tool addons the manual work to the developer. Thus, it is still not solving our problem [7].

Increasing the Reliability of Wireless Sensor Networks with a Distributed Testing Framework is the paper written by Woehrle and team. The paper presents essential features of a framework for testing a broad range of WSN applications. Here the Self-testing is achieved by explicitly specifying the inputs and the expected outputs of the software under test in an executable format. This allows for full automation of the test procedure. Automated testing promotes frequent execution of the tests and enables the designer to continuously compare the implementation's behavior with the specification in multiple

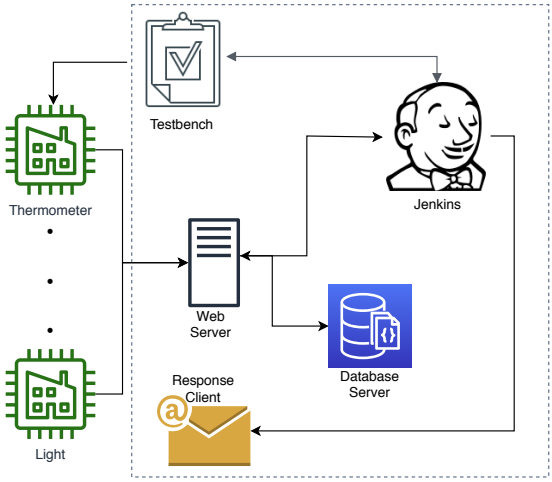

Figure 1: Architectural Overview

platforms. But still the developer has to explicitly specify the inputs and outputs for Software Under Test (SUT).

Though the approaches above are satisfying most of our problem case, we don't have a complete solution yet to our problem. The methods which rely mostly on either emulators or simulators are elegant in terms of the offered test workflow. However, they are also difficult to implement for projects which support many different device types [5].

III. A Comprehensive WOT-CI Test System Architecture

We propose a test system architecture for WoT software development that is not only scalable in the number of supported device types, but also in the number of test scenarios. It allows any device that is built with the support of WoT Thing Description. Our Architecture offers to store the Thing Description in a database. Our system is designed to perform a complete black box testing on all WoT devices in the WoT ecosystem In this section, we initially give an overview of the architecture, then justify the design decision and finally discuss their results.

A. Architectural Overview:

Our architecture consists of WoT Things, Web API server, database server, automation server, testing framework and a Response Client. A test platform can be one or more WoT devices connected to internet and the devices can talk in multiple protocols as the test framework.

The WoT Thing is any device that has a set of sensors and actuators and can connect to internet via its interface. Most importantly the Thing must be designed under WoT Thing Description. The task of the WoT Thing is to push their Thing Description with WoT TD standards to the Web Server as http post request. Things are expected to send their Thing Description, whenever there is software or hardware update on the Thing or on a failover scenario or during network change. A Thing can also communicate with other devices during testing to perform integration testing.

The Web Server is a programmatic interface consisting of one or more publicly exposed endpoints to a defined request–response message system. The Web API Server is additionally assigned to trigger the automation server whenever there is a new Thing Description added to the database and store the result from automation server to database. In our system we have exposed end points to perform all database operations like read, write, update or delete. The task of the Web Server is to process the request and perform the corresponding action in the database and send response to the Things or to a API Client.

The Database server, adopt any database as a backend to store and retrieve. In our system, the database server has two models or tables, first to store the Thing Description and second to store the credentials of automation server. The task of the database server is to perform the actions sent by requests from Web API Server and return a response with success or failure. Additionally, the database server is supposed to maintain the last five versions of Thing Description and consecutive two versions should not have the same Thing Description.

The Automation server is a tool which enables developers around the world to reliably build, test, and deploy their software. In our architecture we need an automation server which doesn't require a codebase to perform automation. As we are using the concept of continuous integration to trigger a build for an update on database. Our Automation server must be capable of sending the test results to the users. The task of the automation server is to clone or download the Web server and test framework repositories and build both the repositories and to start the build to fetch all the Thing Descriptions from the database with the help of Web server and run all the Thing Descriptions against the test framework and get the results and logs from the test framework and compress and send it to users.

The Test Framework is a set of guidelines or rules used for creating and designing test cases. In our architecture the test framework must be based on Thing Description from W3C WoT. The task of the test framework is to test the Thing Descriptions in a sequential manner. A Thing Description should represent capabilities of a device. This implies that if a device supports the interactions that a client can execute based on the device's Thing Description, it doesn't comply to its own Thing Description. The test framework must generate fake data for each variable in the Thing Description and send, read request and write request to the Thing and validate the user access rights and additionally send command to all actions in the Thing Description via IoT protocols and validate the response.

The Response Client is a desktop application that enables configuring one or more contact identities to receive, read, compose and share information with interested groups. In our architecture, we require an response client which can collaborate with our automation server. The task of the response client is to fetch the result and log from test framework and compress it as an attachment and send it to the corresponding recipients and return the status of the response client send function to the automation server. A response is supposed to be send for every build irrespective of build success or failure.

B. Architectural Elements

1) Things: The WoT Things used in this architecture is developed with code guidelines of WoT-SYS tum-esi in github. Wot-SYS is a project that contains the source code, description, guides etc. of the WoT System that will be used by students of Embedded Systems and Internet of Things (ESI) in Technical University of Munich.

2) Web Server: The Web Server in this architecture is developed with NodeJS programming language.

- The Web Server holds function to get and post Thing Description from/to the database.
- The Web server is additionally configured to trigger automation server, whenever a Thing Description is added to the database.
- The Web server is designed to store five versions of Thing Description from the same ID tag with a feature of not holding consecutive Thing Description with same value.

NodeJS is used for the asynchronous call feature. Therefore, with the asynchronous feature NodeJS is lighter and faster than other scripting languages. In programming languages like e.g Java or C hash the "main program flow" happens on the main thread or process and "the occurrence of events independently of the main program flow" is the spawning of new threads or processes that runs code in parallel to the "main program flow". This is not the case with JavaScript. That is because a JavaScript program is single threaded and all code is executed in a sequence, not in parallel. In JavaScript this is handled by using what is called an "asynchronous non-blocking I/O model".

3) Automation Server: Jenkins is used as an automation server. The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project. Major reason for using Jenkins is to use the concept of freestyle project in Jenkins, wherein we need not link the automation server to repository. Since Jenkins is a highly customizable opensource tool, we are using Jenkins to perform the Continuous integration part. Jenkins, in addition to the continuous integration part, it can collaborate with an response client and send out responses. Jenkins runs the build on any machine. In our architecture, Jenkins is configured to run on the same machine and to delete the workspace before every built.

4) MongoDB: MongoDB is widely used across various web applications as the primary data store. One of the most popular web development stacks, the MEAN stack employs MongoDB as the data store (MEAN stands for MongoDB, ExpressJS, AngularJS, and NodeJS). MongoDB is a document database, which means it stores data in JSON-like documents. In our architecture, MongoDB is configured with a database with two collections,

- First collection is for storing the Jenkins credentials
- Second collection is for storing the Thing Description

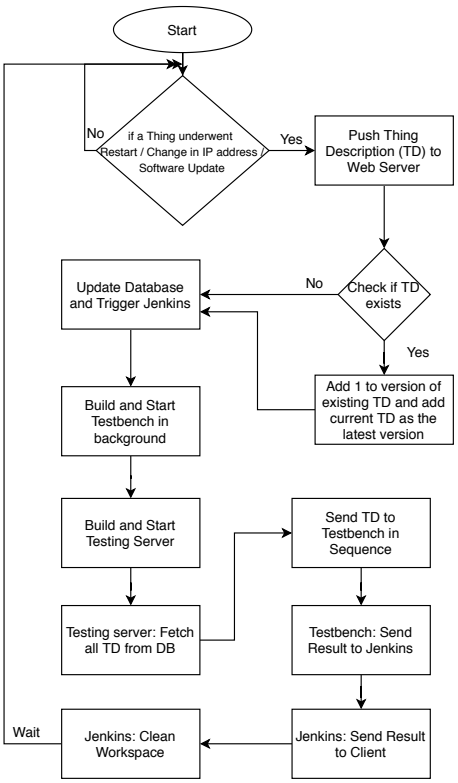

Figure 2: System Design

5) Test Framework: The test framework used in this architecture is the testbench tool from tum-esi in github. Tests a WoT Thing by executing interactions automatically, based on its Thing Description. A Thing Description should represent capabilities of a device. This implies that if a device supports the interactions that a client can execute based on the device's TD, it doesn't comply to its own TD. Test bench tests whether: (1) Every interaction written in the TD can be executed, (2) Writable properties are indeed writable, (3) Each interaction returns the described data type (DataSchema of TD Spec). The testbench generate fake data for each variable in the Thing Description and send read request and write request to the Thing and validate the user access rights and additionally send command to all actions in the Thing Description via REST calls and validate the response.

6) Response Client: In this architecture google mail is used as an response client as it enables us to generate app password. An App Password is a 16-digit passcode that gives a non-Google app or device permission to access your Google Account. App Passwords can only be used with accounts that have 2-Step Verification turned on.

IV. Approach

A. Generation of Thing Description:

A WoT Thing is supposed to push their own Thing Description to Web API Server on three different occasions:

- When the Thing is hard/soft restarted.
- When the Thing is assigned with a new Internet Protocol Address.
- When there is software or hardware update on the Thing.

B. Update Thing Description to Database and Trigger Jenkins Build:

Once the Web API Server receives the Thing Description. The Thing Description "ID" received is compared against the existing Thing Description in the database. If there are no similar existing Thing Description, then add a version tag is added in addition to Thing Description and the Thing Description saved to the database. Response from the database is updated to the user. If there are few existing similar Thing Descriptions available in the database, then compare the received Thing Description and existing latest version of Thing Description for exact match. If there is an exact match, then respond the user stating that the Thing Description sent is similar to the previous one and don't update it to the database. If there is no exact match, then update the version number of the existing Thing Descriptions and add the received Thing Description as the latest Thing Description to the database. Since we maintain only last five versions of Thing Description, the oldest version is deleted from database if we have five versions for a particular Thing Description ID. As soon as the database is updated with a new Thing Description, Jenkins build is called to perform the Continuous Integration concept.

C. Jenkins Build:

On receiving the build trigger for an item, Jenkins is configured to clean up the previous build log and perform the following task:

- Clone the testbench repository from tum-esi in github, build the repository and start the testbench tool.
- Clone the wot-ci repository from tum-esi in github, build the repository and start the testing server.
- Send the result of the test to the users via email.

The task of testing server is to fetch all the latest Thing Description from the database and test it by running it against the testbench. The results and logs and accumulated from the Jenkins workspace and sent to the interested user or user group.

D. Testing Framework:

Testbench started by Jenkins build, will be waiting for a Thing Description. Once it receives a Thing Description, Fake data and calls are generated by testbench and waits for an API call from the testing server, once the testing server send an API call, the call is executed and the Thing Description is tested based on the user requirement. Testbench can be used to perform unit test as well as integration test. The "fasttest" action in the testbench tool is used for the integration test. As a result of this function a test report is generated and returned to the testing server.

V. Conclusion

The combination of resource constrained platform and the highly fragmented nature has made IoT system testing a challenge. Unfortunately, we are unable to achieve the expectations with the emulators. In this paper, we analyzed

the challenges of system testing in IoT devices and presented a test system architecture to perform a automated system test for IoT devices. The presented architecture is supposed to be used for blackbox system testing for W3C WoT ecosystem. Additionally, it takes only one minute and thirty seconds for fifteen device in a isolated ecosystem under a Local Area Network (LAN). Furthermore, we have overcome the interoperability problem with the support of WoT TD. Though we have achieved continuous integration in the WoT ecosystem, it is still not equivalent to end to end automation. A manual work is required to deploy the software into microcontrollers. Since every microcontroller cannot be accessed through telnet or secure shell, it is still a research topic to build an application to deploy the microcontroller software into microcontroller without physical contact. On the other side we have many simulators and emulators on the market to simulate such microcontrollers. But the problem is, it is impossible with the current technology to emulate all possible sensors and actuators in the IoT device market. This problem is an extensive constraint for the developers. One approach we are considering is to use terraform to do deploy the software into microcontrollers to complete the full automation, but many IoT devices are not accessible via secure shell or telnet connections.

REFERENCES

[1] R. Fantacci, T. Pecorella, R. Viti, and C. Carlini, "Short paper: Overcoming iot fragmentation through standard gateway architecture", in *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, 2014, pp. 181–182.

[2] S. K. Datta and C. Bonnet, "Advances in web of things for iot interoperability", in *2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*. IEEE, 2018, pp. 1–2.

[3] M. Fowler and M. Foemmel, "Continuous integration", *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, vol. 122, p. 14, 2006.

[4] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks", *Internet Engineering Task Force (IETF): Fremont, CA, USA*, pp. 2070–1721, 2014.

[5] P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, "A distributed test system architecture for open-source iot software", in *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems*. ACM, 2015, pp. 43–48.

[6] M. Karlesky, G. Williams, W. Bereza, and M. Fletcher, "Mocking the embedded world: Test-driven development, continuous integration, and design patterns", in *Proc. Emb. Systems Conf, CA, USA*, 2007, pp. 1518–1532.

[7] M. Okola and K. Whitehouse, "Unit testing for wireless sensor networks", in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. ACM, 2010, pp. 38–43.