WoT-CI: Continuous Integration in Web of Things Systems

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 2019, there are 26.6 billion IoT devices. But, the major drawback of the IoT ecosystem is its highly fragmented nature. Even if infinite solutions are arising, most of them are incompatible and reflect the needs of particular use-cases and technologies. The World Wide Web Consortium (W3C) Web of Things (WoT) is intended to enable interoperability across IoT platforms and application domains. In the act of using WoT for deploying IoT platform, WoT has reduced a considerable amount of time for the IoT platform developers. Though the platform deployment time is reduced by WoT, it is still a problem of the platform developers to make sure the IoT platform is working as per the requirements after individual device upgrades. In this paper we introduce WoT-CI, a method that uses the principles of Continuous Integration (CI) in the context of WoT, by considering the WoT devices as contributors and WoT system as a single software project. We show that this system can reduce system integration effort while requiring no manual input from the developer, thereby, assuring that the WoT devices are functioning as per the requirement after any change to the hardware, software or network.**

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

I. Introduction

Internet of Things (IoT) is a system of interrelated computing devices like smart home appliances, smart medical appliances 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 is 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. All these devices are built under different architecture and principles. IoT platform developers are supposed to interact with these devices in completely different methods due to its highly fragmented nature. 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 W3C WoT achieves interoperability through wider adoption of the Web principles, standardized device

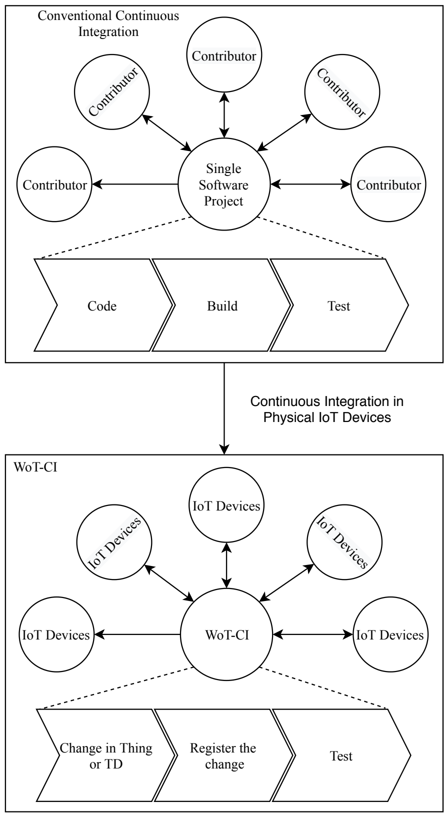

Figure 1: Describes the use of Continuous Integration in WoT-CI: The principles of Continuous Integration (CI) in the context of WoT, by considering the WoT devices as contributors and WoT system as a single software project.

description via Thing Description (TD) standards, metadata and semantic web technologies.

A TD 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.

Problem. 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]. Even though the platform deployment time is reduced by WoT, it is still a problem of the platform developers to make sure the IoT platform is working as per the requirement after any change to the hardware, software or network.

In the conventional software development, Continuous Integration (CI) 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]

Contribution. In this paper we introduce WoT-CI, a method that uses the principles of Continuous Integration (CI) in the context of WoT, by considering the WoT devices as contributors and WoT system as a single software project. Our contribution in this paper shows that this system can reduce system integration effort while requiring no manual input from the developer, thereby, assuring that the WoT devices are functioning as per the requirement after any change to the hardware, software or network. Refer figure: 1

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 TDwhich describes the metadata and interfaces of Things. Thereby, this TD a JSON object is further used for testing the Things in WoT ecosystem. The testbench tool can test a WoT Thing by executing interactions automatically, based on its TD. Each interaction is executed based on the Representational State Transfer (REST) and Publisher Subscriber Model. But testbench is a tool which can test TD of a Thing, when the TD is provided to the tool.

Most commonly, IoT device software is tested with the help of simulators and emulators. The approach in [6] is 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.

In [7] Cooja Simulator, a cross-layer Java-based Wireless Sensor Network (WSN) 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 [6].

[8] 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, while requiring one line of code for each function or variable. Therefore, this tool adds manual work to the developer.

[9] 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 partial 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 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 for getting rid human intervention to assure that the IoT devices are functioning as per the requirement after any change to the hardware, software or network. 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 TD. 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, backend server, database server, automation server, testing framework and a notification service. A test platform can be one or more WoT devices connected to the Internet and the devices can talk in multiple protocols as the test framework.

The WoT Thing Figure 2(a) is any device that has a set of sensors and actuators and can connect to internet via its interface. Most importantly the Thing must have WoT TD. The task of the WoT Thing is to register their TD with WoT TD standards to the Web Server. Things are expected to send their TD, whenever there is software or hardware update on

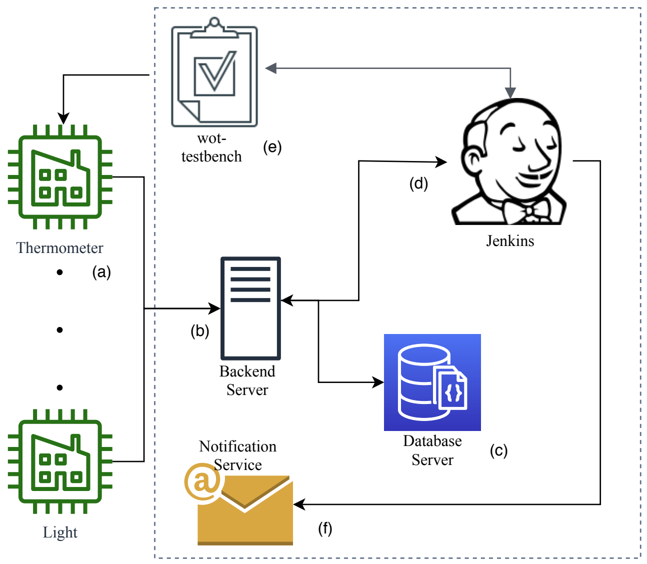

Figure 2: Depiction of the architectural overview of WoT-CI, where Things, Backend server, Jenkins, WoT-Testbench, and Database can be seen

the Thing or on a failover scenario or during network change. Manual registration of TD is also possible. A Thing can also communicate with other devices during testing to perform integration testing.

The backend server Figure 2(b) is a programmatic interface consisting of one or more publicly exposed endpoints to a defined request–response message system. The backend server is additionally assigned to trigger the automation server whenever there is a new TD 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 backend 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 Figure 2(c), 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 TD and second to store the credentials of automation server. The task of the database server is to perform the actions sent by requests from backend server and return a response with success or failure. Additionally, the database server is supposed to maintain the last five versions of TD and consecutive two versions should not have the same TD.

The automation server Figure 2(d) 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 does not 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.
- Build both repositories.
- Start the build to fetch all the Thing Descriptions from the database with the help of the Web server.
- Run all the Thing Descriptions against the test framework.
- Send the test results to the developer.

The test framework Figure 2(e) is a set of guidelines or rules used for creating and designing test cases. In our architecture the test framework must be based on TD from W3C WoT. The task of the test framework is to test the Thing Descriptions in a sequential manner. A TD 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. The test framework must generate fake data for each variable in the TD 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 TD via IoT protocols and validate the response.

The notification service Figure 2(f) is a any application that enables configuring one or more contact identities to receive, read, compose and share information with interested groups. In our architecture, we require a client which can collaborate with our automation server. The task of the notification service is to fetch the result and log from test framework and send it to developers 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

The Architectural overview presented earlier is implemented with the following elements.

1) Things: The WoT Things considered are developed using different protocols such as HTTP, MQTT and CoAP and are programmed differently[1].

2) Backend Server: The backend express server in this architecture is developed with NodeJS programming language.

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

3) Automation Server: Jenkins is used as an automation server. The 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

[1]Source code for these devices are available here: https://github.com/tum-esi/wot-sys.git

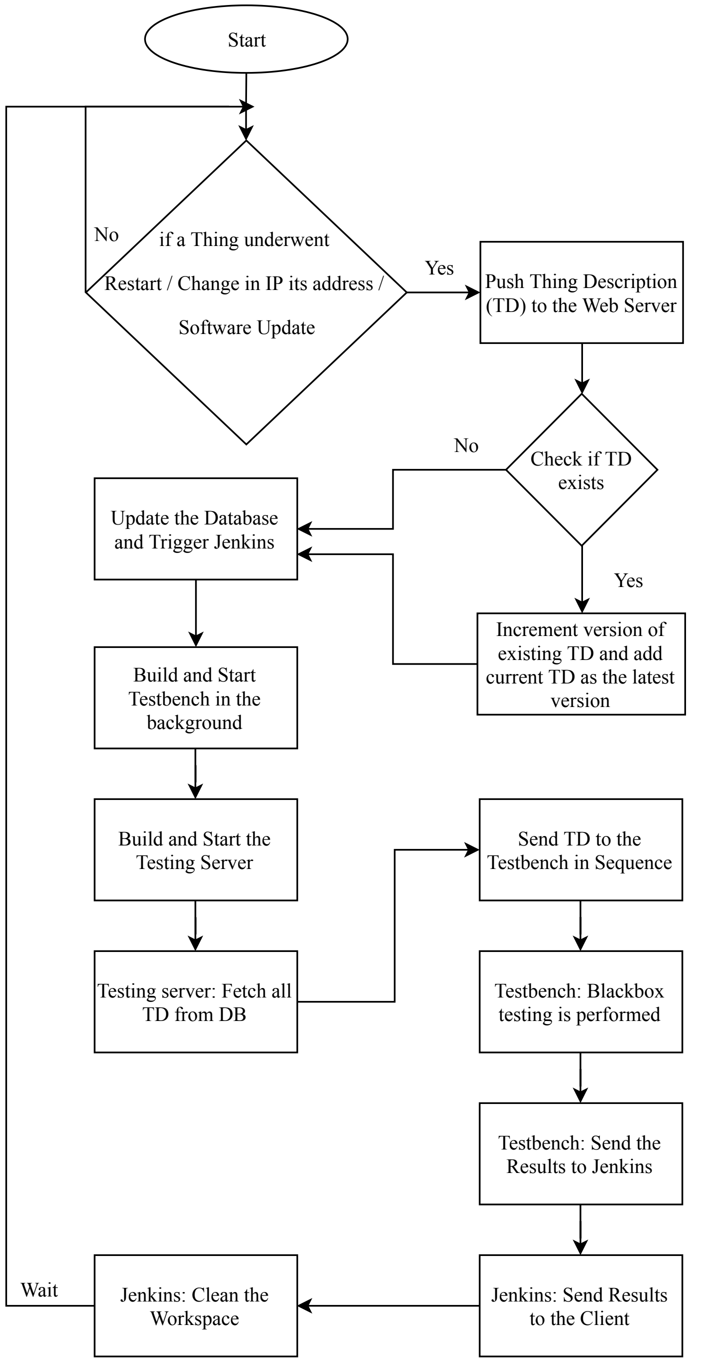
Start
No
if a Thing underwent
Restart / Change in IP its address /
Software Update
Yes
Push Thing Description (TD) to the Web Server
No
Check if TD exists
Update the Database and Trigger Jenkins
Yes
Increment version of existing TD and add current TD as the latest version
Build and Start Testbench in the background
Build and Start the Testing Server
Send TD to the Testbench in Sequence
Testing server: Fetch all TD from DB
Testbench: Blackbox testing is performed
Testbench: Send the Results to Jenkins
Wait
Jenkins: Clean the Workspace
Jenkins: Send Results to the Client

Figure 3: The flowchart above illustrates the flow of the WoT-CI, starting from the system inspecting for a change on Thing to testing new or updated Thing and notifying the users with the results of the Things

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 TD

5) Test Framework: In this architecture, we are using a publicly available wot-testbench[2] tool from tum-esi in github. Test bench tests whether:

- Every interaction written in the TD can be executed
- Writable properties are indeed writable
- Each interaction returns the described data type[3].

The testbench generates fake data for each variable in the TD 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 TD 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

The approach illustrated in the Figure 3 is explained below.

A. Generation of TD:

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

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

B. Update TD to Database and Trigger Jenkins Build:

Once the Web API Server receives the TD.

- The TD “ID” received is compared against the existing TD in the database.
- If there are no similar existing TD,
  then a version tag is added in addition to TD and the TD saved to the database.
- Response from the database is forwarded to the user.
- If there are few existing similar Thing Descriptions available in the database,
  then the received TD is compared with the existing latest version of TD for exact match.
- If there is an exact match,
  then the user is responded, stating that the TD sent is similar to the previous one and the TD is not updated in the database.
- If there is no exact match,

[2]wot-testbench tool is available in https://github.com/tum-esi/testbench.git
[3]DataSchema of TD Specification

then the version number is updated for the existing Thing Descriptions and the received TD is saved as the latest TD to the database[4].

- When the database is updated with a new TD, 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, build the repository and start the testbench tool.
- Clone the wot-ci repository, 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 TD 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 TD. Once it receives a TD, 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 TD 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. Future work

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. 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.

VI. 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. In this paper we introduce WoT-CI, a method that uses the principles of Continuous Integration (CI) in the context of WoT, by considering the WoT devices as contributors and WoT system as a single software project. We show that this system can reduce system integration effort while requiring no manual input from the developer, thereby, assuring that the WoT devices are functioning as per the requirement after any change to the hardware, software or network.

[4]Since we maintain only the last five versions of TD, the oldest version is deleted from database if we have five versions for a particular TD ID.

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”, vol. 122, p. 14, 2006. [Online]. Available: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf

[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] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, “Cross-level Sensor Network Simulation with COOJA”, in *First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, 2006.

[8] 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.

[9] M. Woehrle, C. Plessl, J. Beutel, and L. Thiele, “Increasing the reliability of Wireless Sensor Networks with a distributed testing framework”, in *Proceedings of the 4th workshop on Embedded networked sensors*. ACM, 2007, pp. 93–97.