Smartphone-based Real-time Sensing and Actuation with the Cumulocity Internet of Things Platform

Satish Narayana Srirama*†, Jakob Mass*, Madis-Karli Koppel*, Andreas Sepp* and Shalva Avanashvili*
*Mobile & Cloud Lab,
Institute of Computer Science, University of Tartu
Ülikooli 17, r324, Tartu, Estonia
† Email: srirama@ut.ee

***Abstract*—The successful adoption of IoT and scenarios in different domains have led to the development of several software platforms for managing IoT devices and data. These platforms are based either on cloud-centric or decentralized models. Cumulocity is a proprietary cloud-centric platform which is being used in several industrial IoT scenarios. While Cumulocity supports different types of sensors and devices, integrating mobiles is not yet so straight-forward. To support mobile developers in quickly and utilizing the Cumulocity platform, a generic API is developed. The API supports streaming mobile sensor data to the cloud platform in real time and in return sending commands from the cloud to the mobile applications to trigger actions. The API is demonstrated with a reference Android application implementation where several internal mobile sensors such as accelerometer, gyroscope, and magnetic sensor are integrated. Further, a web application was implemented to visualize the results in real-time.**

***Index Terms*—Internet of Things, Cumulocity, Android, Mobile Sensing**

1. Introduction

Internet of Things (IoT) represents a comprehensive network where physical objects, also known as *things*, with sensing and actuating capabilities are connected and communicate over the Internet, along with the participating individuals. There are billions of such devices which are already connected and are being used in scenarios from different domains such as logistics, ambient assisted living (AAL), healthcare etc. In the traditional model of IoT, i.e. cloud-centric IoT (CIoT) [1], [2], the sensor data from the things is extracted and collected over the gateways and Internet, then stored and processed at the centralized cloud instances. Later, control signals are sent back to the devices to achieve the desired functionality. For example, in an IoT case study where buildings are managed to be energy-efficient, sensor data such as temperature and CO_2 levels of different rooms can be extracted and processed over the cloud, so that the heating and ventilation can be controlled, thus saving the energy usage of the buildings.

While there are several sensors which are being used in IoT scenarios, the mobile phone which is a ubiquitous companion for people and has access to wide range of built-in sensors such as accelerometer, magnetic sensor etc., can also be used in different IoT scenarios. However, current IoT software solutions mainly provide mobile phone support in terms of just using the mobile as a user interface (UI) client to the IoT cloud-hosted application, overlooking the mobile's potential as a sensing-actuating device. Recognizing this potential of using the mobile as an IoT device broadens the scope of possible mobile and IoT applications. For example, one opportunity is to employ CIoT and the mobile for use in mobile crowdsensing [3] or sensing as a service [4], alternatively the mobile could be involved as part of intelligent decision-making to determine user context or interact with the user.

There are several platforms which support the CIoT model, one such is, Cumulocity [5] which is a popular proprietary platform that is being used mainly in industrial IoT scenarios[1]. Cumulocity is a software framework for managing IoT devices, data and integration, including real-time processing, while exposing its functions through Application Programming Interfaces (APIs). Cumulocity also provides means to package user-oriented web applications so that they can be hosted on the cloud.

While Cumulocity supports different types of sensors and devices, integrating mobiles is not yet so straight-forward. Although demo applications of such solutions exist, an open reference implementation of using smartphones as sensor-actuator devices is missing. Moreover, the Cumulocity API is not trivial and involves a learning curve for the developers. To deal with these issues and to support mobile application developers in the task of integrating mobiles with Cumulocity based CIoT, we developed an API. The main goal of the study is to provide a developer tool for using the Cumulocity platform to stream mobile sensor data to a web application in real time and in return send commands from the cloud to the mobile applications to trigger actions.

The main contributions of the paper are:

- Support for integration of different Android based sensors and actuators through a generic API, thereby simplifying streaming the sensor data to the Cumulocity platform

[1] http://cumulocity.com/reference-cases/

- Web interfaces to display the live data stream from different mobiles

The rest of the paper is organized a follows: Section 2 discusses the related background regarding cloud IoT platforms and smartphone sensing therein. Section 3 provides the proposed system architecture and discusses how the API fits into it. Finally, section 4 concludes the work and provides pointers for future work.

2. Background and Related Work

Thanks to the current hype around the IoT phenomena, a large number of software platforms for managing IoT devices and their data exist, with varying characteristics: free-to-use, commercial [6], open-source and closed-source [7].

The core features for an IoT platform can be summarized as:

- **Device Integration**, i.e. providing the means to integrate devices. There must exist some standardized interface, using which, devices can communicate with the platform so that the device functionalities such as sensor reading or actuating can be invoked. Device integration may additionally involve device discovery, device management (e.g. software updates), fault-tolerance.
- **Data Management**. As most IoT-solutions are heavily data-driven, the platform should take care of data storage, structuring and organization. The data management may involve historic data, real-time values or both.
- **Automation, "smart rules".** Integrating devices, their capabilities and data into one platform creates the opportunity of setting up automated decisions (or sequences of decisions, workflows). The platform should provide means of setting up these rules. This may be through some programming or scripting interface, through a user interface, or both.
- **Integration with external (web) services** Oftentimes, the automation rules involve invoking external web services. For example, sending a tweet on Twitter or creating an event in some external system. To this end, many platforms have built-in support for certain web services such as IFTTT.com [8].

In the following, we provide an quick overview of existing Cloud-based IoT platforms. While there is also a significant amount of research and existing solutions in terms of non-cloud-based (decentralized) IoT frameworks, such as OpenHAB [9], due to the scope of this paper we are focusing only at the CIoT solutions.

2.1. Cloud-based (centralized) IoT platforms

2.1.1. Thingspeak. Thingspeak[2] is a open-source, free-to-use IoT platform with MATLAB analytics support. Thingspeak organizes device data into *channels*, and limits data fields to 8 per channel. Devices can be integrated using HTTP or MQTT [10]. It provides a web interface for assembling data visualizations. Thingspeak allows users to create analysis and decision-making based on MATLAB scripts. Combining MATLAB analytics with other webservice integration supported by the platform, such as Twitter, users can create mashup applications.

2.1.2. Cumulocity. Cumulocity[3], established by Cumulocity GmbH is a commercial, free-to-try IoT platform. It provides API-s for integrating devices and web services via technologies such as RESTful HTTP, MQTT, ZigBee, LoRa and others. Cumulocity takes a device-oriented approach in device integration, meaning that all connections between the device and cloud platform are initiated and handled by the device. As a result, there is no need for the device to expose any ports in order to listen for connections, for instance.

Cumulocity also features an ecosystem for building end-to-end UI applications, a high-level real-time processing language for data transformation and decision-making. The real-time processing can be used via a custom language (Cumulocity Event Language). The platform does not provide a visual analytics tool, but individual data streams can still be visualized and custom visualizations of the data can be assembled using web widgets.

Cumulocity also supports connecting to other web services through extension plugins.

2.1.3. Thingworx. Thingworx [11] by PTC is a commercial IoT platform oriented at the industrial domain. Thingworx employs a model-based application development approach.

Like Cumulocity, Thingworx devices initiate the connection to the Thingworx cloud. To this end, Thingworx provides tools such as the AlwaysOn^{TM} protocol, which is based on the WebSocket, HTTP and TLS technologies [12].

Thingworx additionally provides a visual mashup tool for combining the data services available within ThingWorx to create applications. Further, it features an analytics toolset which allows creating machine-learning based analytics applications with a UI based tool for describing data and creating prediction models, anomaly detection and so forth. Most of the analytics features can be used via the visual interface or via the REST API.

2.2. Mobile Phones as Sensors

Cumulocity provides a demo application called the *Android Cloud Sensor app* [13]. However, the solution is not open-source and is meant as a demonstration of the platforms capabilities. The application supports the following sensor data: acceleration, (GPS) location, luxometer, gyroscope, RSSI, magnetic field, rotation and the following actuator functions: display message, vibrate.

If a mobile developer wishes to create their own Cumulocity-based mobile sensing application, the Cloud Sensor App however does not provide anything in terms

[2]https://thingspeak.com/

[3]https://www.cumulocity.com

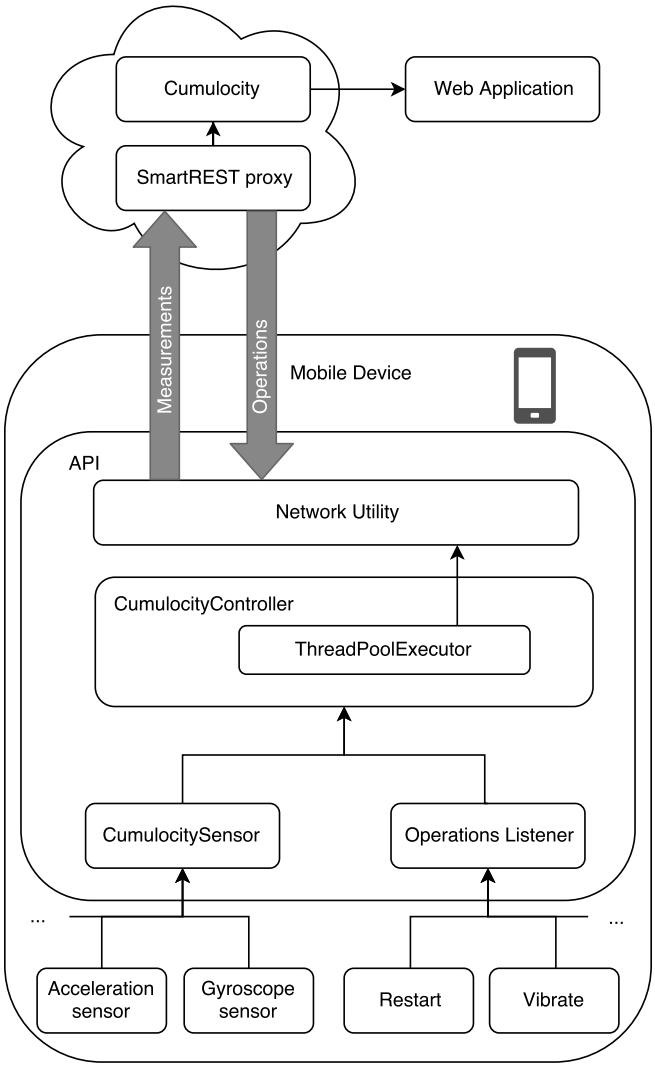

Figure 1. System Overview

of extensible developer tools or APIs, as the developer has to implement their own sensing and actuating integration using the Cumulocity API.

3. System Design & Implementation

Our envisioned mobile sensing system (see Fig. 1) consists of 3 main components: the Cumulocity cloud platform; a mobile application for sending data and receiving commands; and a web application for data visualization (analysis).

The mobile application has been implemented with extensibility in mind, to this end it includes an API for defining new sensors and actuators. The API consists of generic interfaces which enable developers to focus on implementing the mobile-specific aspects of creating sensor readings and actuator behaviour, while minimizing the effort of integrating with the cloud platform.

In the following subsections, we describe each of the 3 components in depth. Note, that while we present an Android implementation of the mobile API in this work, we have also implemented a corresponding iOS version, available on Bitbucket[4].

3.1. Android Application

The Android application is native, all action takes place inside a single Activity. It is meant as a guideline or reference implementation on how to use the APIs *CumulocitySensor* and *CumulocityControllerListener* classes. For that, an example *CumulocityController* and *CumulocitySensorImpl* have been created. Two helper classes, that are platform specific, are also included: *RegistrationHelper*, that helps with registering and requesting credentials, and *NetworkUtil*, that deals with requests for sending and receiving data from the cloud platform.

3.1.1. Modular Design, API. As shown in Fig. 1, ”Mobile Device” , Java interfaces are provided to the developer to simplify communication with the Cumulocity API. It reduces the learning curve by generalizing and reducing the number of steps required for basic flows.

Adding new sensors is as generic and easy as possible. In this case, a sensor can be anything the user wants, as long as it is implements the CumulocitySensor interface. The interface is very simple - all it requires from the user is to first register it with a sensor name, names of values as an array (e.g. ”x”, ”y”, ”z”) and the minimal update frequency. After doing this, whenever the user has new data available for their abstract sensor, they can call a function on the interface which will send the data to Cumulocity if at least the minimal update time period has passed.

Implementations of interfaces are also provided. CumulocitySensorImpl provides an example of the interface implementation, specifically targeting easy to add internal targets. If the user wants to create their own custom CumulocitySensor which does not use the internal sensors, they’ll have to create their own class which implements the interface.

The CumulocityController is a utility class that handles connecting to the cloud, device registration and respective callbacks. For initiating a connection to the cloud platform, the user merely needs to provide the URL of Cumulocity instance, everything else is automated.

Once the device has been successfully registered, the user can call a method in the controller to register a new CumulocitySensor. Once all sensors are added, another method of the controller starts updating the sensors if they have any new data available. The controller is implemented as a generic class so that it can be integrated into common Android components such as Activity or a Service without having to change anything internal to it.

[4]https://bitbucket.org/mobilecloudlab/madp_fall2017_cumulocitystreaming

3.1.2. Device bootstrapping. Life cycle of the Android application is divided into two main parts. First, credentials from Cumulocity are fetched to register the device. This is only required once, as the device stores the credentials in its preferences and they are later used for additional requests.

This step is integrated inside *CumulocityController*, *RegistrationHelper* and *NetworkUtil* classes. The developer must only create an instance of *CumulocityController* and add a listener to it that implements *CumulocityControllerListener* and listens to *onDeviceRegistered()* event. All intermediate steps - credential fetching, device creation, registration - are performed in the background and are hidden from the developer.

3.1.3. Sending and receiving data. The second part of the life cycle is sending sensor data (measurements) and receiving operations from Cumulocity. After the device is registered, the application will try to register a SmartREST template for each sensor.

SmartREST is a Cumulocity tool for compressing payload sizes, thus making the application more lightweight, further details on SmartREST are described in the next section.

After template registration, the application starts sending sensor data using the templates.

In addition to sending sensor data, the application also queries for operations from Cumulocity. To do this, the application subscribes to operations with the given device ID and starts waiting by long polling. Demo application is waiting for *c8y_Restart* and *c8y_Vibrate* operations. Once again, this is coupled away into internal classes. The developer only has to decide which operations to listen for using *CumulocityController*'s *initOperationListener()*. Once an operation is received, the application Activity is notified via *onOperation()*.

3.2. Cumulocity API

In the previous section we demonstrated how our API can be used. In this subsection we describe the standard Cumulocity components which our API internally uses, highlighting the work involved in implementing a similar solution without our API.

3.2.1. SmartREST. SmartREST is an extension of the standard HTTP REST protocol developed by Cumulocity. It is designed to reduce used bandwidth by introducing a template system for sending only minimal requests.

Before being able to use SmartREST, one should create a template for the request with a specific identifier (X-Id). The template can be thought of as a shortened format of the REST request body which will be sent to Cumulocity. It includes the number of parameters, their types, the name of the template and so forth. An example template registration request is shown in Fig. 2. The *json* body of the request describes the template itself. Based on this template, SmartREST requests will be translated (uncompressed) at the cloud to the corresponding original JSON format which the normal REST API expects.

```
POST /s

{
  "c8y_Acceleration":{
        "x":{"value": "&&" },
        "y":{"value": "&&" },
        "z":{"value": "&&" }
  },
  "time": "&&",
  "source":{"id":"&&"},
  "type":"c8y_Acceleration"
}
```

Figure 2. Sample SmartREST template registration request

The SmartREST proxy endpoint URL on Cumulocity is also shortened to "/s" instead of for example "/measurement/measurements" when using the standard REST API, again decreasing request sizes. After registering the templates, the provided X-Id from the template registration response is stored.

Measurements request headers consists of the same X-Id and Authorization header. A request example is shown in Fig. 3, it corresponds to the template presented in Fig. 2.

```
POST /s
200,1,2,3,433765
```

Figure 3. Sample SmartREST sensor measurement request body

The first number is the template id, next three numbers are the sensor values and the last one is the device id.

SmartREST is used for sending Sensor data in Android and iOS demo applications. A network bandwidth Comparison of sending four sensors with and without SmartREST is shown on Fig. 4. It is visible that the difference is about six times when four sensors are used. Approximate data sent per hour is 28 MB, compared to 120 MB without using SmartREST.

3.2.2. Operations. Operations allow sending actions to connected devices, such as restarting or taking a picture. Our demo application implements the restart and vibrate

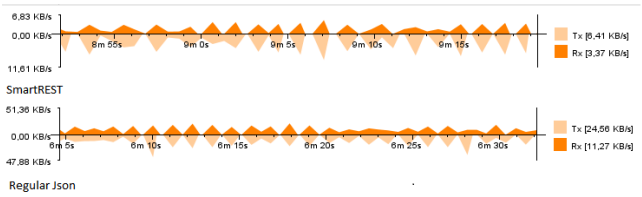

Figure 4. Comparison of bandwidth usage

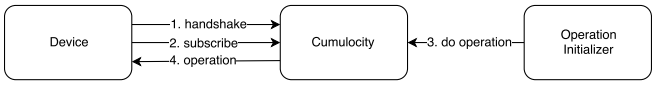

Figure 5. Simplified operations structure

operations. More detailed information about operations can be found from official documentation[5,6].

The implementation of operations involves three distinct steps, also shown in Fig. 5:

1) handshake
2) subscribe
3) poll

First step in operations procedure is the handshake. It is a POST request ”80” (SmartREST shortcut) to */devicecontrol/notifications* and the response is $< clientID >$.

Next, this $clientID$ is then used to subscribe for operations for the device. The subscription request format is shown in Fig. 6. One $clientID$ can be used to subscribe to multiple channels.

```
POST /cep/realtime

[{
 "channel": "/meta/subscribe",
 "clientId": "<clientID>",
 "subscription": "operations/<deviceID>"
}]
```

Figure 6. Operations subscribe request format

The third step is polling for new operations, as shown in Fig. 7.

```
POST /devicecontrol/notifications
[{
  "channel":"/meta/connect",
  "connectionType":"long-polling",
  "clientId":"< clientId>"
}]
```

Figure 7. Operations polling request format

The device should not close the connection and start waiting for response, it only receives a response once an operation is sent. When the device receives a response it would be good to also push to */devicecontrol/operations/<operationId>* value $\{$”$status$” : ”$SUCCESSFUL$”$\}$ to let Cumulocity know that operation was completed. After the operation is received the device should start long-polling again to wait for new operations.

Operations are invoked either by using the Cumulocity web UI or by POST-ing to /devicecontrol/operations/. The body should be a json, that contains the operation and deviceID, for example:

```
{"deviceId": "433765", "c8y_Vibrate":{} }
```

At the moment only receiving keyword-based operations is supported. However, adding support for parameters is easy as it only requires changing the way json is parsed.

3.3. Web Application

The third component of the presented system is a web application that fetches data from Cumulocity instance and visualizes it to the end user.

The web application has two main views. The first one is used for entering Cumulocity instance URL and user credentials, which are used to fetch the list of available devices. The compatible devices are denoted by the Cumulocity ”type” field being set to ”c8y_SensorPhone”. Once a compatible device is selected, the user is forwarded to the visualization page.

The second view is the visualization page, shown in Fig. 9. The user is presented with a *three.js* [14] 3D canvas, containing a phone model which visualizes the rotation of the phone in physical space.

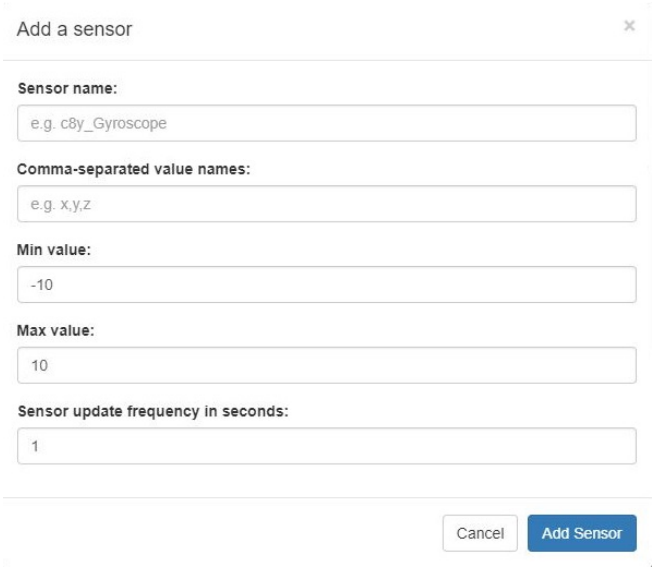

Figure 8. Adding new sensor in web application

Additionally, there are three default graphs available which show real-time data fetched from Cumulocity. The default graphs are for accelerometer, gyroscope and rotation sensors. However, more sensors of any type can be easily added either by code or by using the user interface, shown in Fig. 8. To add a new sensor, the user has to know the Cumulocity sensor name along with the names of its values, e.g. ”c8y_Gyroscope” and [”x”, ”y”, ”z”] respectively. Additionally, the user can input the minimum and maximum value for the axes so the graph only shows a specific region of values on the y-axis. Lastly, the user can input the update

[5]https://www.cumulocity.com/guides/reference/device-control/
[6]https://www.cumulocity.com/guides/reference/real-time-notifications/

interval time, which controls how frequently Cumulocity is queried for new data. The same data can also be embedded in the source code with a single line, with several examples available in the code repository.

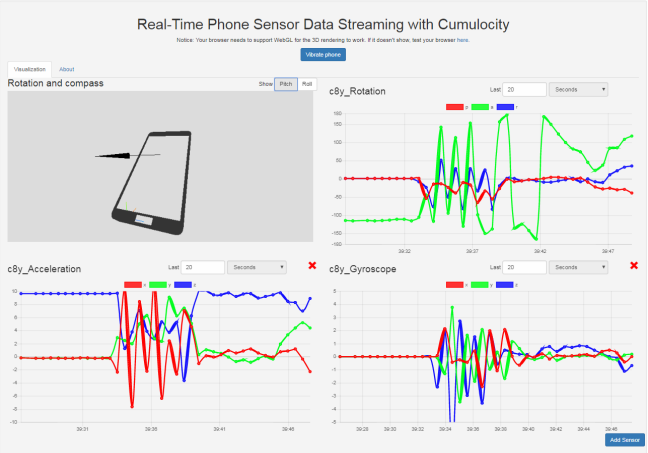

Figure 9. Visualization page in web application

By default, all the charts fetch last 20 seconds of data from Cumulocity, but the interval can be changed individually for each chart above it. In addition, the graphs can be removed to reduce bandwidth usage and to focus on specific measurements.

4. Conclusion & Future Work

Cloud-based platforms provide a solid foundation towards creating innovative IoT solutions. However, the ubiquitous smartphone sensors lack direct platform support in the case of platforms such as Cumulocity. In this paper, we presented a mobile API to enable rapid mobile sensor and actuator integration with the Cumulocity cloud platform for developers. The API minimizes the learning curve of the Cumulocity API, allowing the developers to focus at the mobile aspects.

We also provided a reference Android implementation of the API, where it is part of a real-time streaming system with a web application component for data visualization, including 3D visualization of the phone's orientation. The implementation involves sensors such as gyroscope, acceleration and actuator capabilities such as vibrate.

While the current communication is based on the light HTTP SmartREST protocol, the communication bandwidth could further be reduced by using MQTT instead of HTTP for sending and receiving data from the cloud, which we have left as part of future work.

Acknowledgments

The authors would like to thank Telia Eesti AS for helping us in establishing the Internet of Things and Smart Solutions Laboratory (IoTSS Lab), where the research has been conducted and for providing a Cumulocity license. This work is also supported by IT Academy/ StudyITin.ee.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] S. N. Srirama, "Mobile web and cloud services enabling internet of things," *CSI transactions on ICT*, vol. 5, no. 1, pp. 109–117, 2017.

[3] M. Talasila, R. Curtmola, and C. Borcea, "Mobile crowd sensing," *Handbook of Sensor Networking: Advanced Technologies and Applications*, 2014.

[4] X. Sheng, X. Xiao, J. Tang, and G. Xue, "Sensing as a service: A cloud computing system for mobile phone sensing," in *Sensors, 2012 IEEE*, pp. 1–4, IEEE, 2012.

[5] Cumulocity GmbH, "Cumulocity. about." [Online] https://www.cumulocity.com/about/.

[6] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the internet of things," in *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pp. 1–8, IEEE, 2015.

[7] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of internet-of-things platforms," *Computer Communications*, vol. 89-90, no. Supplement C, pp. 5 – 16, 2016. Internet of Things: Research challenges and Solutions.

[8] S. Ovadia, "Automate the internet with if this then that (ifttt)," *Behavioral & Social Sciences Librarian*, vol. 33, no. 4, pp. 208–211, 2014.

[9] K. Kreuzer, "openhab 2.0 and eclipse smarthome," 2015.

[10] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-sa publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*, pp. 791–798, IEEE, 2008.

[11] P. E. I. Solutions, "Platform technology: Thingworx. 2016," *URL: https://www. thingworx. com/*.

[12] PTC, "Thingworx edge sdks and websocket-based edge microserver (ws ems) help center," 11 2017. [Online] http://support.ptc.com/help/thingworx_hc/thingworx_edge/#page/thingworx_edge_sdks_ems%2Fems_wsems_topics%2Fv5.4.0_c_ems_wsems_features.html%23.

[13] Cumulocity GmbH, "Android Cloud Sensor App." [Online] http://cumulocity.com/guides/users-guide/android-cloud-sensor-app/.

[14] R. Cabello *et al.*, "Three. js," *URL: https://github. com/mrdoob/three. js*, 2010.