Cross-Platform Scenario Module for Internet of Things Testing Architecture

Osama Abu Oun(✉), Christelle Bloch, and François Spies

FEMTO-ST Lab (CNRS) - University of Franche-Comte,
1 Cours Leprince-Ringuet, 25200 Montbéliard, France
{oabuoun,chirstelle.bloch,francois.spies}@femto-st.fr

Abstract. The Internet of Things (IoT) represents a vision in which the Internet extends into the real world embracing everyday objects [9]. Billions of objects are already connected to the Internet. These objects would intercommunicate without any human intervention and they would have different operating systems. Enterprises and developers should produce different versions of each application (same functionality) for each operating systems. For testing and evaluating all these versions, developers and testers should develop/redevelop the same scenario for all versions of this application. Cross-Platform Scenario Module is designed to solve this problem by separating the testing scenario (events and actions) from the application that will execute these events and actions on the object. Same scenario might be written using data serialization formats (such as, Extensible Markup Language (XML), JavaScript Object Notation (JSON) or Concise Binary Object Representation (CBOR)) for all versions of the application. As an example, the format used in this research is the XML.

Keywords: Internet of things · Testing IoT · Evaluating · Scenario module · IoTaaS

1 Introduction

According to Cisco Internet Business Solutions Group (IBSG), IoT is simply the point in time when more "things or objects" were connected to the Internet than people. Cisco projects that by 2020 there will be nearly 50 billion devices on the IoT [4]. Physical items are no longer disconnected from the virtual world. They can be controlled remotely and can act as physical access points to Internet services [6]. Within the IoT literally anything can be connected to a computer network, via an IP address like the one in your computer, and allowed to transfer data without the need for human-to-human or human-to-computer interaction. A "Thing" could be a car, an animal with a bio-chip transponder, a fitness band on your wrist, a refrigerator, the jet engine of an airplane, or your cat's collar. These objects, in addition to billions of others, could become connected to the Internet with the help of sensors and actuators [7].

B. Mandler et al. (Eds.): IoT 360° 2015, Part II, LNICST 170, pp. 385–395, 2016.
DOI: 10.1007/978-3-319-47075-7_43

Designing an architecture for testing and evaluating IoT systems requires more than providing a server and several devices. It requires the ability to build/rebuild all the details of a real environment where the system will be deployed/run. Such environment could be built using a mix of the following items: real things (devices), simulators and emulators. One of the most important aspects in any testing environment is the ability to generate and regenerate several scenarios in order to test an application. Nowadays, each application has different versions, each of them works on different operating system or on different releases of the same operating system. In this paper, we will present a Cross-platform Scenario Module. This module is one of the components of our proposed architecture for building a cloud environment for offering IoT testing as a service. The remainder of this paper is organized as follows: Sect. 2 presents the state of the art of this work and we survey related work. Section 3 presents (briefly) the main architecture used to build an IoTaaS. Sections 4 elaborates designing our cross-platform scenario module for IoT testing architecture. The implementation and the results are discussed in Sect. 5. And finally, some concluding remarks and future work are mentioned in Sect. 6.

2 State of the Art

Many researches have been conducted in order to simplify application testing and evaluation. Most of these researches were targeting the desktop, web and mobile applications. One of the main differences between these applications and IoT applications is the human intervention. In IoT, a lot of objects and devices will be able to communicate directly without any human intervention. So any testing tools for the IoT should be able to automate testing scenarios in order to be executed without human intervention. To the best of our knowledge, cross-platform scenarios for testing IoT applications and devices yet to be introduced in the literature. Mobile application testing is the closest domain to the IoT testing. In mobile application testing, many companies have developed frameworks in order to help developers and testers to automate application testing. One of these projects is Appium [2]. Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS and Android platforms. It is based on Client/Server architecture. It allows tester to write tests against multiple platforms (iOS, Android), using the same API. This enables code reuse between iOS and Android test suites. There are client libraries (in Java, Ruby, Python, PHP, JavaScript, and C#) which support Appium's extensions to the WebDriver protocol. Robotium [8] is an open-source test framework for writing automatic gray box testing cases for Android applications. Each scenario needs new app to be generated. The new app can only send events to the application that has the same signature.

Our Cross-platform Scenario module depends on XML files, so testers don't have to write a new application for each scenario. It can work in both modes (client-server and standalone). It can be used in any operating system if it allows injecting event. The module is designed to permit the mix between static and dynamic scenarios.

3 IoTaaS Architecture

An IoT environment would consist of any device might be able to exchange data directly/indirectly with Internet/Intranet. In general, a simple IoT environment is formed of:

- **Sensor:** It is a device which can detect a physical state and convert it into data (readable by computer).
- **Actuator:** It is a device which can change a physical state.
- **Gateway:** It is a dedicated device or an application to read data from sensors, send data to actuators.
- **Network:** Many types of networks would exist in an IoT environment. Some networks would be traditional ones, such as Wi-Fi, while others would use technologies which have been developed for IoT.
- **Application:** It is the software which processes data. It could be hosted on any type of devices, such as: server, personal computer (PC), mobile, tablet, etc.

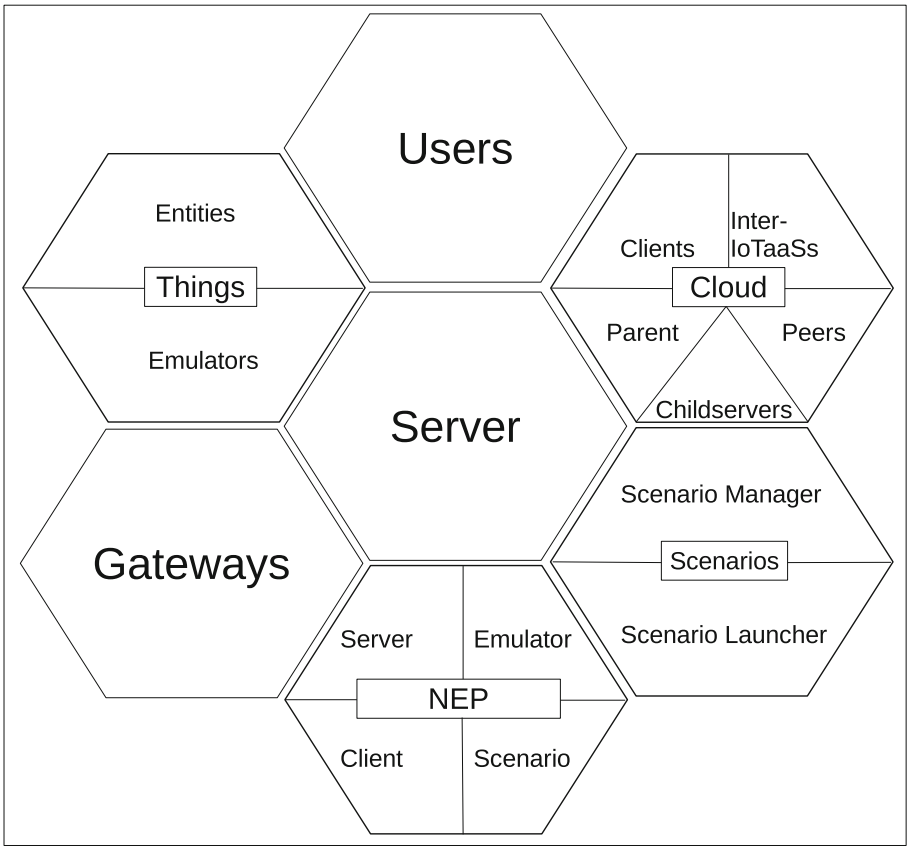

Fig. 1. IoTaaS architecture

The existence of these components could be realized by providing the devices (ex. sensors), or by using emulators. An IoTaaS might be used to test and evaluate any component of an IoT. For example, new sensors should be tested with certain IoT frameworks or with certain gateways, an updated IoT frameworks should be tested in order to test their compatibility with the old actuators, etc. The proposed IoTaaS architecture consists of 7 modules (components) (Fig. 1). These components are the following: Things, Gateways, Emulators, Network Emulation Protocol (NEP), Scenarios, Cloud and the Server.

4 Scenarios

A scenario is an outline or model of an expected or supposed sequence of events. For any application, there is a certain number of sequences of actions and events. This number varies depending on the application itself. For some applications, it could be infinite number of scenarios.

For standalone applications, it is possible to prepare a testing scenario in order to test some (or all) sequences of events. The same application could have different versions for same operating systems, or different versions for different operating systems. In this case, each version should have its own scenario.

It is getting much more complicated when talking about testing network-based applications, where events depend on results or actions from other applications on same device or different device. In nowadays applications, most of testing scenarios depend on user input. In IoT applications, the majority of events and actions take place between devices and applications directly (user input isn't needed). A standalone-based testing wouldn't be sufficient to cover all possible scenarios for a system, nor for an application.

The proposed solution is client-server-based scenario module (Fig. 2). The module consists of three main components: Scenario Files, Scenario Manager (Server) and Scenario Launcher (Client). The main idea of this module is to separate the actions on a given framework from the sequence of events (scenario) for an application. In other words, the same scenario file (XML file in this design) could be sent to different systems in order to run the same sequence of events. In the following subsections, we discuss in details these three components.

4.1 Scenario Files

Scenarios should be written and described using a cross-platform machine-readable language, such as XML. The following attributes define a general ***Trigger*** used in this file (repeatedly):

- **Type:** In general, there are 5 different actions which could be used as a trigger:
  1. **counter:** a counter is set to 0. The countdown will be initialized starting from the integer value in the ***Value*** attribute.
  2. **time:** reaching the time specified in the ***Value*** attribute. Time format is YYYY-MM-DDThh:mm:ss.sTZD (ISO 8601). Time attributes will be ignored on devices which have no internal clocks.

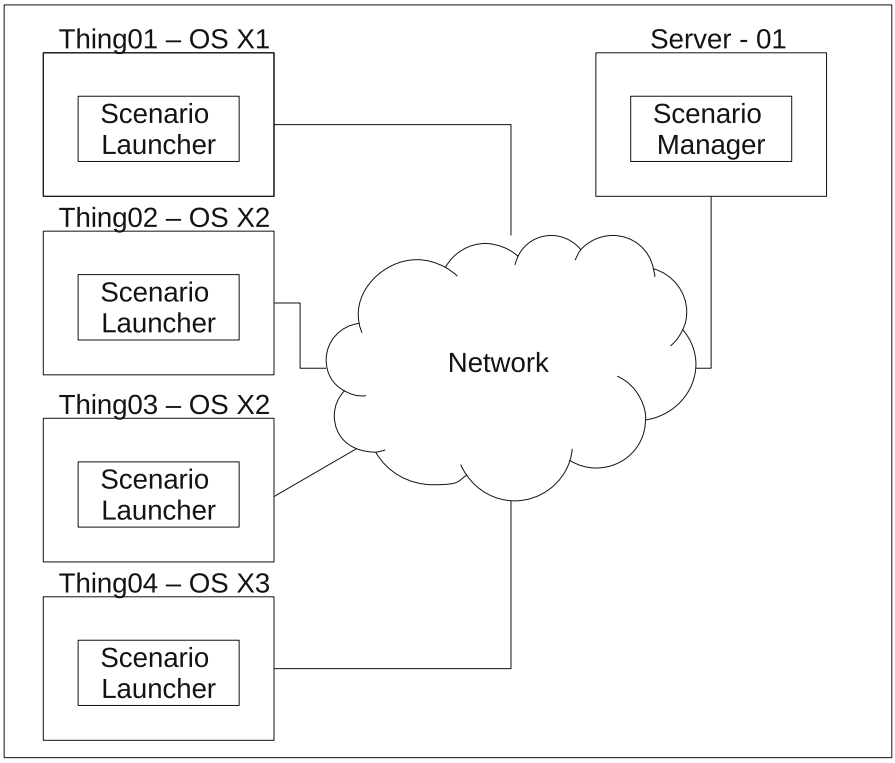

Fig. 2. IoTaaS - Scenario's general architecture

3. **action:** an action is triggered. The ***Value*** attribute should contain this action name or ID.
4. **delay:** a countdown timer has reached zero. The ***Value*** would have this time in the format PnYnMnDTnHnMnS (ISO 8601).
5. **never:** infinity.

– **Value:** This attribute could have different type of values depending on ***Type***'s attribute.

In the following, we list the main tags and attributes in this file (Code 5.8.1):

– **Scenario:** It consists of the attributes which define the scenario in general. This is a mandatory tag. It has 5 main attributes:
  – **ID:** It is the scenario identifier. Each scenario should have a unique identifier. This identifier is used to collect the results and to prepare the statistics.
  – **Name:** It is human-readable name, it could be used to distinguish different scenarios.
  – **Type:** In case of having different categories of scenarios, this attribute could be used to indicate the type.

- **Start**: This is an optional tag. By default, the scenario will be executed directly unless of using this tag in order to fix a condition. It has a trigger **StartTrigger (Type/Value)** which is used to define when the scenario should be started.
- **Loop**: A scenario could be executed more than once. This is an optional tag (in case of its absence, scenario will be executed only once). The following attributes define loop settings:
  - **Enabled:** This is a boolean attribute determining whether the scenario will be applied once (=0), or it would be repeated (=1).
  - **StopLoopTrigger (Type/Value):** It is a trigger used to define when the scenario should be stopped.
- **End**: This is an optional tag. By default, the scenario will be stopped directly after the last action is applied (loop isn't defined). It has a trigger **StopTrigger (Type/Value)** which is used to define when the scenario should be stopped. In case of using this tag, the scenario will be stopped as soon as the trigger is hit, even in the middle of execution of the scenario. This tag should have a priority over all other events.
- **action**: Each scenario consists of a sequence of actions. Each action has 4 main attributes
  - **ID:** A unique identifier for each action is required. This ID is usually given by the framework.
  - **Name:** It could be the same as the ID. It could be a human-readable name.
  - **Command:** In case that the action isn't predefined, a command could be provided in order to be executed.

  An action could have parameter(s). Each parameter has the following attributes:
  - **ID:** This is a local identifier. Each parameter of the same action should have its own ID.
  - **Name:** It is a human-readable name. It can have the same value of ID.
  - **Type:** It defines the type of the parameter. The type could be any simple type, such as: integer, short, float, string, etc. The complex types could be: file or server, which means that it should read the parameter from a file, or it should contact an external server.
  - **Value**: In case of simple types, this attribute contains the value of this parameter. Otherwise, this attribute will have file path or server URL. This attribute could be omitted in case of having an array of values.

  A parameter could have an array of values. Each value is defined as follows:
  - **ID:** A unique identifier for each value is required.
  - **Name:** It is a human-readable name. It can have the same value of ID.
  - **Value:** In case of simple types, this attribute contains the value of this parameter. Otherwise, this attribute will have file path or server URL.

Other tags and attributes could be defined. It is important to mention that the same file should be used for all platforms.

Code 4.1 - Scenario File Architecture (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<scenario id="" name="" type="">
  <start startTriggerType="" startTriggerType=""/>
  <loop enabled="0/1" stopLoopTriggerType=""
                  stopLoopTriggerValue=""/>
  <end type="" value=""/>
  <actions>
     <action id="" name="" command="">
           <parameters>
                 <parameter id="" name="" type="" value=""/>
                 <parameter id="" name="" type="" >
                    <values>
                         <value id="" name="" value=""/>
                         <value id="" name="" value=""/>
                    </values>
                 </parameter>
           </parameters>
     </action>
  </actions>
</scenario>
```

4.2 Scenario Manager

This component preforms all functions on server side. Scenario Manager could be written in any programing language and it can run on any operating system. There are four main subcomponents which are responsible of performing server's functions (Fig. 3):

- **Dispatcher:** The principal function of this subcomponent is to maintain connections with scenario launchers (clients) in order to exchange data. The data exchanged is divided into four types:
  - **Scenario Files:** Dispatcher sends to launcher a scenario file (if exists). Scenario files could be stored directly on the server, separated database or on external storage server. Each scenario file should be categorized according to: the application, the scenario and the client.
  - **Parameters:** Scenario Launchers would need certain parameters from the scenario manager. Scenario Launchers send a request, the dispatcher send back the requested parameter.
  - **Dynamic Actions:** Dispatcher can send certain actions to certain clients. These actions would be obtained from the application analyzer.
  - **Results:** A Scenario Launcher sends the results of executing certain scenario (or certain dynamic actions) as soon as it has a connection with its own scenario manager. The results should be categorized according to: the application, the scenario and the client.

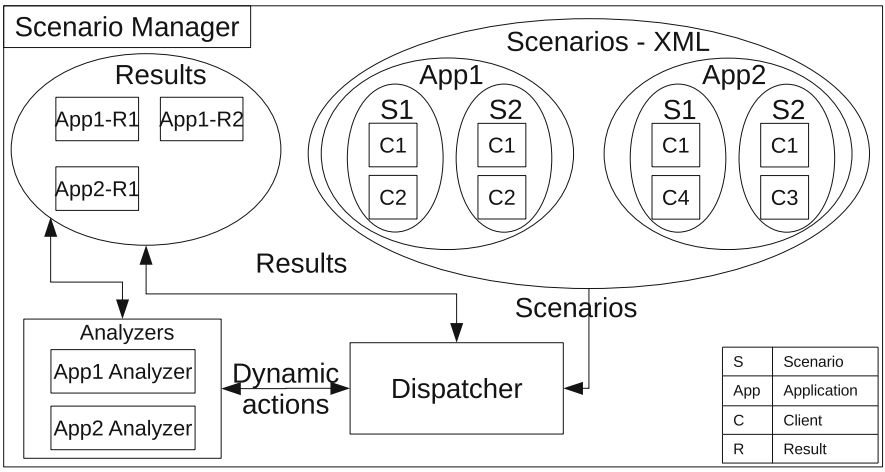

Fig. 3. IoTaaS - Scenario manager

Dispatcher and Scenario Launchers can communicate using a cross-platform distributed application protocol, such as: Web Services.

- **Results:** Results are all data received from Scenario Launchers as a result of executing a scenario or a dynamic action. Dispatcher receives these results and store them directly on the server, dedicated database or on an external server.
- **Analyzers:** Scenario Manager could have analyzers for certain application. An analyzer would analyze the results received from the Scenario Launchers. Depending on this analyze, it could send direct actions to certain clients through the Dispatcher. Analyzers can be a good tool in order to give dynamicity to a testing scenario. Analyzer output could be added as results.

Several Scenario Managers could be used in a given environment. Scenario Launchers could be configured to have a backup Scenario Manager. Other solutions could be provided using load-balancers techniques.

4.3 Scenario Launcher

Scenario Launcher is the component on the client device that perform testing end evaluating functions. This component varies depending on the device's framework. Each framework (or each version of framework) should have its own version of Scenario Launcher, it should be developed using a programing language supported by the framework, such as Java for Android and Objective C for OSX and iOS, etc.

There are four main subcomponents in any Scenario Launcher. In the following, we present these subcomponents (Fig. 4):

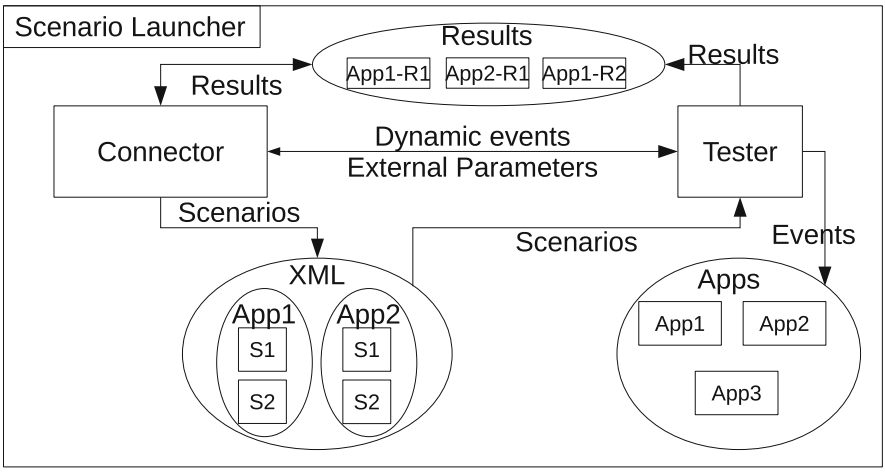

Fig. 4. IoTaaS - Scenario launcher

- **Connector:** This subcomponent is responsible of maintaining the connection with a Scenario Manager. This connection is used in order to exchange following types of data:
  - **Scenario Files:** Scenario Launcher receives the scenario file(s) from a Scenario Dispatcher. It stores Scenario File(s) locally. Each scenario file should be categorized according to: the application and the scenario.
  - **Parameters:** When Scenario Launcher needs certain parameters from the scenario manager. Scenario Launcher sends a request, the dispatcher sends back the requested parameter.
  - **Dynamic Actions:** When the Scenario Launcher receives a dynamic action from the Scenario Manager, this action will have a priority over all other actions.
  - **Results:** The results of scenario executed (or dynamic actions), they will be sent directly or from the local storage.

  Connector could be considered as the bridge that connects between the Scenario Manager and the other subcomponents in the Scenario Launcher.
- **Tester:** It is responsible of executing the scenarios which have been received from the Scenario Manager. The tester should have the permissions to:
  - inject events into other applications on the same device.
  - listen to events dispatched by other applications or by the operating system itself.
  - read/write files from/to the local storage.
  - set/get operating system parameters.

  Tester should be multi-threaded application in order to execute multiple scenarios at the same time.
- **Results:** In case of having a connection with the Scenario Manager, the Scenario Launcher sends the results of executing certain scenario (or certain

dynamic actions) directly, otherwise the results would be stored locally awaiting for a connection with a Scenario Manager. The results should be categorized according to the application and the scenario.
– **Applications (Apps):** On the same device, one application at least should be the target of testing and evaluating. The same application would have different versions depending on the device's framework. The testing application(s) should permit other applications to inject events.

For security reasons, and since the Scenario Launcher would have access to all the functions on the device, Scenario Launcher applications should run only in special mode (ex. debugging mode). This mode should be turned on/off manually by the user or the tester. This could protect normal devices from being hacked by using these privileges an permissions.

5 Implementation

Our experiments and scenarios aim at demonstrating the possibility to implement the cross-platform scenario module. We selected the following parameters for our testing environment:

– **Client:** The client used is a mobile phone working prepared with Android operating system.
– **Server:** The server's operating system is linux Ubuntu 14.04 server 64 bits.
– **application:** Notepad application.

This version consists of Java application (server) and an Android service (Scenario Launcher) on the client. The Android service has been developed in order to keep working in the background. As soon as it receives a scenario file from the server, it launches the target application on the mobile and start injecting the events into this application. The target application should have *inject_events* permission. Inject_events allows an application to inject user events (keys, touch, trackball) into the event stream and deliver them to ANY window [3]. The scenario launcher has to be signed with the system signature in order to be able to use this privilege. The Android service could read any scenario and inject its events into any application on the mobile phone. The performed experiment campaigns demonstrated the feasibility of this new testing and evaluation concept, and they showed, besides, that it is possible to change the scenario file in order to run the new scenario directly without changing the Android service.

6 Conclusion

The result of the implementation proves the possibility of separating the scenario files and the scenario application. Instead of wasting the time of testers and developers in developing different testing application for each scenario. The scenario launcher would be a general application capable of executing any scenario file by injecting the events into any application on the mobile phone. Our future work

consists of developing the other functions, porting the application to the other operating systems used in IoT frameworks and building the client-server module using RESTful API [5] and representing scenarios using different languages (XML, JSON, CBOR) in order to handle implementation on constrained devices. Future work will focus on studying the impact of our design on performance of constrained devices. We are planning to exchange messages between the server and the clients using our protocol Connectionless Data Exchange (COLDE). COLDE utilizes Wi-Fi Management Frames to exchange data between devices without being connected to any network [1]. This will permit the clients to exchange data with the server as part of the management frames which keeps the primary network of the device available for the testing tasks.

References

1. Abu Oun, O., Abdou, W., Bloch, C., Spies, F.: Broadcasting information in variably dense environment using connectionless data exchange (CoLDE). In: Mellouk, A., Fowler, S., Hoceini, S., Daachi, B. (eds.) WWIC 2014. LNCS, vol. 8458, pp. 283–296. Springer, Heidelberg (2014). http://dx.doi.org/10.1007/978-3-319-13174-0_22
2. Appium: Automation for the apps. Technical report, Appium. http://appium.io. Accessed 2015
3. Developers, A.: Manifest-permission. Technical report, Android. http://developer.android.com/reference/android/Manifest.permission.html. Accessed 2015
4. Evans, D.: How the next evolution of the internet is changing everything. Technical report, Cisco (2011). https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf
5. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, aAI9980887 (2000)
6. Mattern, F., Floerkemeier, C.: From the internet of computers to the internet of things. In: Petrov, I., Guerrero, P., Sachs, K. (eds.) Buchmann Festschrift. LNCS, vol. 6462, pp. 242–259. Springer, Heidelberg (2010)
7. Morien, C.: Connectivity 101: the internet of things. Technical report, The University of Texas at Austin. https://identity.utexas.edu/id-perspectives/connectivity-101-the-internet-of-things. Accessed 2015
8. Robotium: robotium recorder. Technical report, Robotium. http://robotium.com/. Accessed 2015
9. Weiser, M.: The computer for the 21st century. Sci. Am. **265**(3), 94–104 (1991)