A Distributed Test System Architecture for Open-source IoT Software

Philipp Rosenkranz, Matthias Wählisch, Emmanuel Baccelli, Ludwig Ortmann

A Distributed Test System Architecture for Open-source IoT Software

Philipp Rosenkranz
Freie Universität Berlin
philipp.rosenkranz@fu-berlin.de

Matthias Wählisch
Freie Universität Berlin
m.waehlisch@fu-berlin.de

Emmanuel Baccelli
INRIA, France
emmanuel.baccelli@inria.fr

Ludwig Ortmann
Freie Universität Berlin
ludwig.ortmann@fu-berlin.de

ABSTRACT

In this paper, we discuss challenges that are specific to testing of open IoT software systems. The analysis reveals gaps compared to wireless sensor networks as well as embedded software. We propose a testing framework which (a) supports continuous integration techniques, (b) allows for the integration of project contributors to volunteer hardware and software resources to the test system, and (c) can function as a permanent distributed plugtest for network interoperability testing. The focus of this paper lies in open-source IoT development but many aspects are also applicable to closed-source projects.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools, Diagnostics, Distributed debugging; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; H.3.4 [**Systems and Software**]: Distributed systems

Keywords

open-source IoT, interoperability, test system architecture

1. INTRODUCTION

An ever growing number of heterogeneous smart objects interconnect and communicate over the Internet using a wide variety of different protocols. Billions of such smart objects are expected to be "Things", i.e., cheap devices that are extremely constrained in memory, CPU, and energy resources [4]. These devices typically reside at the Internet edge, and extend the latter in what is known as Internet of Things (IoT).

However, today the IoT is still just a vision. A big obstacle on the way to make this vision a reality are interoperability issues. Significant challenges in this area are located at the system and network level [7]. In the worst case, the lack of system level interoperability can lead to code being rewritten for every type of "Thing" an IoT application is meant to support. However, what is even worse for IoT systems is the lack of network level interoperability. This can lead to smart objects being unable to communicate which each other, thus completely contradicting the idea of the IoT.

Another major issue in IoT is that programming smart objects is non-trivial due to their constrained resources, compared to programming typical Internet hosts (e.g., desktop and laptop computers, smartphones and tablets). For instance, while programming a standard network client is a straightforward task on a typical Internet host, it is a challenge on IoT devices. The main reason for this is that functionalities on IoT devices are expected to be the same as on typical Internet hosts (to guarantee network interoperability), while the available resources (e.g., memory) on IoT devices can be less than a millionth of the resources available on a typical Internet host.

In order to alleviate this problem, smaller operating systems have recently been developed and used as software platforms upon which to build network stacks and applications running on IoT devices. These operating systems aim at providing an API comparable to what is available on typical Internet hosts, while remaining extremely frugal in terms of required resources, to match IoT devices constraints [4] (e.g., fitting the whole OS, the network stack and applications in a few kBytes or RAM and ROM). Typically, such operating systems come with their own network protocol stack which may or may not be compatible with other implementations. Examples of such operating systems include RIOT [1], Contiki [5], FreeRTOS [15] or TinyOS [10]. Note that in this paper, we will focus on open-source community-driven solutions because, we argue it is the most trustworthy approach to develop and maintain a piece of software as complex as a modern network stack—let alone a full operating system.

However, in this context, an important question arises: What tools and frameworks can be used to efficiently test IoT applications, IoT operating systems, and network stacks? This question is relevant for two reasons. First of all, to the best of our knowledge, the topic of IoT software testing has been mostly overlooked so far, both by research and industry. Second, as we will see, testing IoT software in an open-source, community-driven context is more challenging than it may look at first glance. In particular, a holistic method is needed to efficiently test and debug network-related features. The design of such a method, especially one which also incorporates interoperability testing, is a difficult task.

In this paper, we (i) identify the unique challenges one faces with open-source IoT software and IoT interoperability testing (see § 2), (ii) review available building blocks, testing tools and frameworks that are most relevant to open-source, community-driven IoT software development (see § 3), and (iii) present a holistic test system architecture which builds upon the concept of hardware and software volunteering in order to address the issues we identified (see § 4). Our approach also leverages interoperability testing (see § 5).

2. CHALLENGES

Testing software components in the IoT differs significantly from testing software running on typical Internet hosts, because IoT devices are resource-constrained [4]. This has two implications. First, a full testing suite usually cannot be deployed on the IoT device. Second, IoT software components are strongly coupled with the hardware. In this section, we discuss these implications in more detail and relate them to two basic approaches for testing IoT systems at a functional level and debugging: virtualization techniques and hardware based tests.

Challenge 1—IoT virtualization: Virtualization typically includes the use of emulators or simulators. In the IoT, the *system under test (SUT)* is inherently networked. Therefore, both emulators and simulators must provide network capabilities. For testing purposes we only look at *simulators* which precisely model hardware, and *emulators* which only implement a specific operating system's APIs, such as the *native* platforms of Contiki and RIOT. Several network [11] and hardware simulators for different architectures, applicable for IoT, have been developed [6]. However, a simulator needs to model not only each supported architecture, but also each hardware component—network controllers, sensors, and actuators—to be complete. A simulation-based approach to testing IoT systems is therefore often not doable, either because an appropriate simulator does not exist, or because it is not open-source, or not publicly available. Emulators, on the other hand, only model the APIs of a particular operating system. While their extension for new hardware components is relatively easy, they are not available for every operating system, and their accuracy is very limited. Due to the emergent properties of network algorithms, hardware platforms and operating systems, the use of network simulators inherits all the limitations that apply to the virtualization techniques for IoT hardware. Thus, while emulators and simulators are useful for basic testing, they are often insufficient, or not even available, and must be complemented by tests on target platforms and in testbeds.

Challenge 2—IoT testbeds: Hardware-based testing usually involves a testbed. Recent surveys (e.g., the survey conducted by Musznicki and Zwierzykowski [16]) show that several large-scale testbeds have been deployed over the last few years. System tests and functional network tests can thus be performed in a network consisting of real IoT devices available on open testbeds. However, this approach also has drawbacks. Testbeds are usually very expensive to setup and maintain, and often difficult to use, even though the latter two issues have been considerably eased by the emergence of testbed management and control frameworks like OMF [14]. Furthermore, testbed hardware is outdated very soon and typically homogeneous (e.g., tens or hundreds of identical nodes). This contrasts with the IoT in practice, since new hardware is available on a monthly basis and heterogeneity is the norm. With respect to this volatility, testbeds—which represent a static deployment—are not sufficient.

Challenge 3—Heterogeneity of IoT hardware: Some tests can be conducted in software using device emulators or simulators. However, in order to ensure that the combination of software and hardware works as specified, system tests must be performed on real devices. Hence, every device supported by the SUT must be available for these tests. This conflicts with community-driven projects, in particular in the context of IoT, and the concept of distributed maintenance. The plethora of IoT hardware makes it unlikely that core maintainers centralize or own all the necessary hardware in the long run. This platform availability problem leads to situations in which the overall quality of the SUT can only be determined by some developers who own the hardware. Testing, though, should be conducted by third parties.

Challenge 4—Heterogeneity of IoT toolchains: Usually, each constrained device comes with its own dedicated combination of compiler and debugger software. This software is either proprietary and costly, or free and open-source, and depends on device manufacturer and the microcontroller in use. Software tools are commonly supplemented by hardware debugging interfaces, such as JTAG to allow for access to debugging output. These interfaces can be used to connect the SUT running on an IoT device with a software debugger running on a developer workstation. Additionally, other devices (e.g., logic analyzers) can be employed to check for the correct behavior of low-level software components such as drivers.

The huge number of different configurations of compilers, devices and debugging utilities can be difficult to handle in a centralized fashion. The problem becomes even more complex in the context of open-source development, when hardware-specific tools are not publicly available. A developer, thus, might not be able to compile or test the SUT for every supported platform, as the developer does not have access to the necessary tools.

Challenge 5—Interoperability testing: In order to ensure network level interoperability, it is not enough to simply adhere to protocol specifications as those often contain subtle ambiguities which can result in partly or completely incompatible implementations. In addition to this, developers might choose to optimize a protocol implementation which could have a negative impact on a communication partner using a different implementation. Therefore, it is extremely important for developers to be able to test their code changes against other implementations. However, testing for network level interoperability usually requires all possible software and hardware combinations to be present at one place, which is difficult to manage for companies and, even more so, for open-source projects, especially since the number of possible combinations is typically very high.

Intermediate conclusion: Due to the above issues, an open-source community in the IoT domain will typically have a development processes based on GitHub (or equivalent) and mailing list discussions etc., according to which different people test code when asked per email, on the subset of hardware they have "at home", and report back. This highlights the need for a distributed architecture, which automates this process in a scalable, future-proof manner. A useful test system for the IoT must (i) handle the platform availability problem, (ii) allow for test runs, even in conjunc-

tion with proprietary hardware interfaces and tools, and (iii) offer a clear and easy way to conduct network interoperability and functional tests.

3. CURRENT STATE OF THE ART

We are not aware of publications that address the issue of (open-source) IoT software testing for constrained devices. Therefore, we discuss existing research in related fields with respect to their applicability in open-source IoT software testing.

Testing of embedded software: Karlesky and Williams [8] presented a test-based approach for embedded software development. Even though this paper does not explicitly mention constrained devices, it deals with testing software being developed for a very similar class of devices. The approach clearly shows that principles of test-driven development (TDD) can also be applied to software development for constrained devices. The concept relies heavily on clean software modularization, unit-testing and continuous integration techniques. In order to simplify the test process, Karlesky and Williams propose the use of a hardware emulator for most tests. Deploying an emulator offers multiple benefits: (i) it decreases the dependency on the hardware for the development, (ii) tests can run directly on an external developer host or a test server in conjunction with readily available test software, and (iii) tests often execute much faster on an emulator than on the target hardware, thus providing quicker feedback to the developer. However, using an emulator is only practical if the SUT supports a small set of different device types, since an emulator is needed for each type. For open-source IoT projects like RIOT, which exhibit a growing number of different device types, this approach might therefore not be feasible.

Testing of wireless sensor networks: Similar to the IoT, wireless sensor networks (WSN) need to deal with the testing of network functionality on constrained devices. A WSN consist of sensor nodes, which are essential constrained devices that collect and report physical data to a data sink. They often use networking technologies which are similar to those used in IoT systems. Most of the following publications focus on the simplification of the test process for WSN. They all do so by providing facilities which allow a tester to manipulate the state of nodes in the network.

Okala and Whitehouse [12] propose a test system in which a central test server can run a script that communicates via remote procedure calls (RPC) with sensor nodes. Based on this interface, the test server can manipulate and read the memory from each node, thus allowing full access to the internal state of the SUT running on the node. However, a drawback of this system is that the RPC system needs to map state variables reliably to memory regions and vice versa. This might be difficult to implement.

Woehrle *et al.* [18] generalize the client-server concept, such that a single test server can interact with nodes either in a WSN or in a simulated testbed but do not require direct access to memory of the nodes. An important feature of this architecture is that the same test cases can be used across two different test platforms. In addition to this, it also automates the most tedious aspects of testing on testbeds, such as log trace collection and transferring the SUT to multiple nodes.

Eriksson *et al.* [6] present a different approach to testing WSN by focusing on interoperability testing, using the wireless sensor network simulator COOJA. This system can execute compiled machine code for a range of 8 and 16-Bit devices and, at the same time, simulate the wireless communication medium. The use of COOJA for functional network tests allows for true white-box testing, since the exact state of each node in the simulated network is known over the complete testing time. However, this approach has the same drawbacks as the one proposed by Karlesky and Williams, in that a hardware simulator needs to be written for each supported device type, which may be infeasible for projects supporting a large number of device types.

Interoperability: Standardization bodies for communication technologies like the European Telecommunications Standards Institute (ETSI) and Bluetooth SIG have well defined guidelines and processes [17, 3] to ensure network interoperability. The process usually involves both conformance testing against a specification and plugtest-events. Plugtest-events are physical meetings of technology implementers, in which each party brings their systems with the intent to test those against systems from other parties. Since such meetings come with a very high organizational overhead, they happen only sporadically. Even though plugtests are mainly frequented by industry entities, they are also visited by open-source IoT communities and research institutes. A good example for this are the plugtests organized by ETSI for the Constrained Application Protocol (CoAP) [9]. However, the sporadic occurrence of these events makes it possible for an interoperability issue to remain potentially undiscovered for a long time. In addition to this, such events require the presence of at least one test engineer or developer in order to be effective. Sending people to such events might be financially difficult for open-source communities without sponsors.

Open problems: Even though all of the presented approaches solve a few of the issues discussed in the previous section, it is worth noting that neither solves all of those challenges completely. 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. The WSN approaches allow for automation of network tests but suffer from similar problems, since they rely on the availability of testbeds for specific target devices. Therefore, we argue that these solutions are insufficient, on their own, for functional IoT testing. The current approach to interoperability testing can also only be described as lacking, due to their reliance on physical plugtest-events. However, if we carefully combine all mentioned techniques together with an alternative to plugtests and a solution for the test platform availability problem, they can form the foundation of a holistic test system for IoT software development.

4. A COMPREHENSIVE IOT TEST SYSTEM ARCHITECTURE

We propose a test system architecture for open-source IoT software development that (i) is not only scalable in the number of supported device types, but also in the number of test methods and build environments, (ii) allows for the volunteering of hardware and software resources to the test process, and (iii) offers a easy to use workflow for functional network related tests. In order to achieve this, our architecture is based on the concept of crowd computing. We extend

methods known from the testing of software for wireless sensor networks as well as embedded systems (see § 3).

We first give an overview of the architecture, then justify the design decisions for each group of components, and finally briefly discuss their implications.

4.1 Overview about the general architecture

Our architecture consists of a central continuous integration (CI) broker and a test cluster. The test cluster consists of clients running on computers connected to the Internet. The clients either provide access to one or more attached test platforms, provide specific build environments, or both. A test platform can be, for example, one or more IoT devices attached to one computer, whole IoT testbeds, or specialized device simulators.

The task of the *CI broker* is, first of all, to function as a CI system. As such, it should be able to trigger test runs on request (e.g., in the event of detected code changes) and report test results to the developers. Additionally, the CI broker works as a coordinator for a test cluster. It allows (i) test cluster clients to register or unregister test platforms and build environments, (ii) keeps track of capabilities and configuration details of those and, most importantly, (iii) acts as a broker between test cases and test platforms or build environments, based on the required and offered capabilities.

The *cluster clients* build sources for tests, execute test cases on connected test platforms, or do both, instructed by the CI broker. As soon as they start, they register with and send their capabilities and configuration details to the CI broker. A configuration detail could be the number of IoT devices in an attached test platform, their device type, or simply the available build environments on the client computer. An important feature of our architecture is that these clients can be distributed and be operated by external parties which support the test effort. An example is illustrated in Figure 1. Note that there are two different build environments connected to the clients for platform A, whereby the second build environment is not connected to the same client as test platform A. This is possible because the test cluster provides transparent access to each resource provided by a client.

A *test case* consists of three components: the SUT, a test configuration, and multiple scripts. The test configuration is described by information about the build environment and the test platform requirements of a test case. These requirements could be a minimal number of devices in a test platform, the presence of at least one device with a specific peripheral attached to it, or a minimum version number for a specific compiler.

The execution and monitoring of the test case is based on multiple scripts. Each script implements a dedicated task: providing test stimuli to the SUT, collecting output traces, extracting information relevant for the test case from collected traces, and checking if a test case passed or failed based on the extracted information. Instead of interacting directly with a platform, scripts interface with the API of a test platform abstraction layer.

The *test platform abstraction layer* allows for test cases to interact with test platforms using one unified API. Thence, it supports multiple different types of test platforms such as IoT testbeds or device simulators, and exposes their functions through a single API. Typical examples for exposed functions are sending commands over serial line to one or more devices, flashing devices, collecting output traces from one or more device, or getting a list of devices in the test platform.

4.2 CI broker and test cluster

The common approach to continuous integration (CI) is the deployment of a central server, which controls all build or test configurations of interest for a project. The server is usually supported by client programs, so called slaves, which execute tests and builds accordingly to the central configuration. Many open-source CI systems, such as Jenkins or Buildbot, support this concept out of the box.

However, centralized CI systems increases the complexity in open-source IoT projects as development is distributed and no single authority exists. When a new test platform or build environment is added, the server configuration has to be updated by an administrator. This usually involves the following steps: (a) an external developer sends configuration details about the test platform to the CI server administrator, (b) the administrator changes the build or test configuration accordingly, and (c) finally the external developer installs and configures a slave. If the developer then decides to add a different test platform to the test system, the same steps will be repeated.

We decided to use a different approach, which simplifies the process of adding new test platforms and build environments to the system. The proposed CI broker and test cluster concept allows an external developer to support the testing effort by only configuring a test cluster client. Hereby, the configuration effort of the client is limited to adjusting details about the hardware or software resource an external developer contributes to the test system (e.g., the compiler version of a build environment, or the number or types of IoT devices attached). As soon as a client is configured, it can be started and will connect to the test cluster by registering with the CI broker. It is worth noting that this approach differs from common CI systems, and is necessary as open-source IoT projects require many different test platforms and build environments.

The proposed test cluster approach can either be implemented on top of the well-known CI framework Buildbot or as a plugin for Jenkins.

4.3 Test cases and platform abstraction layer

One might argue that splitting up test cases into SUT, scripts, and a configuration might complicate deployment. However, this decision, in conjunction with the test platform abstraction layer, offers multiple benefits. First, it separates the test case from the test platform, thus allowing test cases to be used across different platforms. Second, the use of a platform abstraction layer simplifies network related tests since it eliminates most of the challenges when dealing with many IoT devices at once, generally, and IoT testbeds, in particular. Finally, it allows the CI server to determine suitable test platforms for a test case dynamically.

The test cases themselves can be implemented in any programming language. However, as a foundation for the platform abstraction layer, we argue for a comprehensive description language such as the Network Experiment Programming Interface (NEPI) [13]. NEPI already offers the required functionality, is under active development, and support for IoT testbeds is planned.

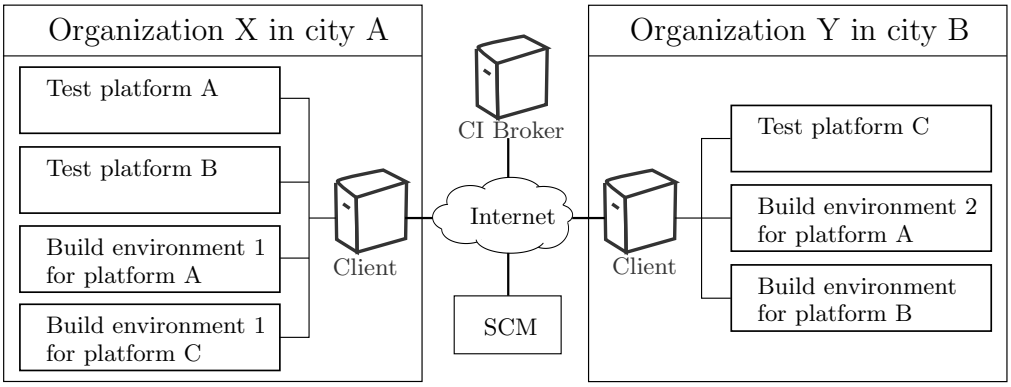

Figure 1: General architecture exemplified for two contributing organizations.

5. INTEROPERABILITY TESTING

Our approach on interoperability testing builds upon the flexibility and the distributed nature of our test system architecture, in order to allow for both automated standard conformance and interoperability tests. Since the former can be easily implemented as a suite of test cases on top of our architecture, we will concentrate, in this section, on network interoperability testing. Hereby, we extend the previously described architecture in order to create a permanent distributed plugtest which provides quick interoperability test feedback to developers and involved third parties.

The basic idea of our approach to network interoperability testing is simple. A third party can integrate their IoT system similarly to regular IoT devices, as described previously. For such systems, which run potentially proprietary software, this means that a minimal amount of code must be written so that it can be controlled by the platform abstraction layer. At the very minimum, the code must implement functions to set the system into a state suitable for interoperability tests (e.g., resetting the device and/or configuring network interfaces etc.). Test cases which instrument both the integrated third party system and an already integrated communication partner, can then be written on top of the platform abstraction layer, in order to test for interoperability issues. Note that the term IoT system encompasses, in this context, both software and hardware third party solutions.

A major drawback of this approach is that it requires both communication partners to be present in the same location. This problem can be alleviated by introducing transparent network bridges to the distributed test system architecture. These bridges forward communication traffic to and from test platforms or third party systems located in entirely different sites. While there are different options to implement the bridge between sites (e.g., distributed messaging services or VPNs), the end-points of the bridge need to be adjusted to fit the communication systems used in the involved SUTs. This means that, in order to create a bridge between systems using IEEE 802.15.4 wireless transceivers, the end-points of this bridge must also use IEEE 802.15.4 transceivers. This is not a disadvantage since, for most wireless communication technologies employed in IoT systems (e.g., IEEE 802.15.4, IEEE 802.11b/g/n and Bluetooth), cheap off-the-shelf solutions are available. For added flexibility, some wireless end-points could consist in software-defined radio (SDR) devices. Currently, such devices cost more than off-the-shelf hardware we mentioned above, but (i) they can be reconfigured to function like (nearly) any needed transceiver type, and (ii) their price can be expected to drop. While relevant open-source SDR implementations (e.g., GNU Radio-based IEEE802.15.4 implementation [2] by Bloessl et.al) so far lack crucial features, it can be expected that they will be enriched in the near-future.

Interoperability tests can be either conducted with a transparent bridge or without. While the bridge-less method allows for a variety of different types of interoperability tests to be conducted, the method using a bridge comes with some limitations. The main limitation is that the bridge introduces an artificial delay between two communicating SUTs. Additionally, the bridge also has the potential to mask issues which could be observable in a bridge-less scenario. This is especially true in wireless networks in which some parameters of the communication medium (e.g., noise, interference, multipath loss, etc.) substantially differ from a bridge-less test scenario.

6. CONCLUSIONS

The combination of constrained computing platforms and the requirement for complex network functionality makes IoT software development challenging. Unfortunately, the topic of testing such software has been mostly ignored. In this paper, we analyzed the challenges in testing IoT software and presented a comprehensive test system architecture designed to tackle these issues. The proposed architecture introduces test clusters, allowing virtually anyone to contribute test platforms to this system. These distributed test platforms allow shared access to single IoT devices or fully fledged IoT testbeds. Additionally, the system enables test cases across different test platforms, due to the use of a platform abstraction layer. The same abstraction layer also simplifies network-based functional and interoperability tests. Furthermore, we showed that our architecture can be used for network interoperability testing by forwarding communication traffic from one system under test to another, regardless of their physical location, thus creating a permanent distributed plugtest.

In future work, we will extend our current implementation of the described architecture and evaluate this approach in the wild, based on our open-source IoT operating system RIOT. Furthermore, we will concentrate on the issue of trust for the test cluster and the detection of faulty test

platforms. The first problem could be solved with processes already in place in many open-source projects. Faulty test platforms could be detected by developing special test platform test cases, which could be run periodically on the platforms. Based on our experience with RIOT, we will refine these strategies. Additionally, we will examine techniques that would allow our implementation to be easily integrated into existing testbed federations. One approach we consider is combining our implementation with the testbed management and control framework OMF.

Acknowledgements

This work was partially supported by ANR and BMBF within the projects SAFEST and Peeroskop.

7. REFERENCES

[1] Baccelli, E., Hahm, O., Günes, M., Wählisch, M., and Schmidt, T. C. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the 32nd IEEE INFOCOM. Poster* (Piscataway, NJ, USA, 2013), IEEE Press.

[2] Bloessl, B., Leitner, C., Dressler, F., and Sommer, C. A GNU Radio-based IEEE 802.15.4 Testbed. In *12. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN 2013)* (Cottbus, Germany, September 2013), pp. 37–40.

[3] Bluetooth Special Interest Group. The Bluetooth SIG Interoperability Program White Paper, June 2007.

[4] Bormann, C., Ersue, M., and Keranen, A. Terminology for Constrained-Node Networks. RFC 7228, May 2014.

[5] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of IEEE LCN* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 455–462.

[6] Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., and Marrón, P. J. COOJA/MSPSim: Interoperability testing for wireless sensor networks. In *Proc. Simutools* (2009), ICST, pp. 27:1–27:7.

[7] IEEE Standards Association. IEEE-SA Internet of Things (IoT) Ecosystem Study, Jan. 2015.

[8] Karlesky, M., Williams, G., Bereza, W., and Fletcher, M. Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *Proc. Emb. Systems Conf, CA, USA* (2007).

[9] Lerche, C., Hartke, K., and Kovatsch, M. Industry adoption of the internet of things: A constrained application protocol survey. In *Proceedings of the 7th International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2012)* (Kraków, Poland, Sept. 2012).

[10] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence* (2004), Springer Verlag.

[11] Musznicki, B., and Zwierzykowski, P. Survey of simulators for wireless sensor networks. *International Journal of Grid and Distributed Computing 5*, 3 (2012), 23–50.

[12] Okola, M., and Whitehouse, K. Unit testing for wireless sensor networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications* (2010), SESENA '10, ACM, pp. 38–43.

[13] Quereilhac, A., Lacage, M., Freire, C., Turletti, T., and Dabbous, W. NEPI: An integration framework for network experimentation. In *Proc. of SoftCOM* (2011), pp. 1–5.

[14] Rakotoarivelo, T., Ott, M., Jourjon, G., and Seskar, I. Omf: A Control and Management Framework for Networking Testbeds. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 54–59.

[15] Real Time Engineers ltd. FreeRTOS web-site. http://freertos.org/.

[16] Tonneau, A.-S., Mitton, N., and Vandaele, J. A survey on (mobile) wireless sensor network experimentation testbeds. *Proc. of IEEE DCOSS* (2014), 263–268.

[17] van der Veer, H., and Wiles, A. Achieving Technical Interoperability - the ETSI Approach. ETSI White Paper No. 3, Apr. 2008.

[18] Woehrle, M., Plessl, C., Beutel, J., and Thiele, L. Increasing the reliability of wireless sensor networks with a distributed testing framework. In *Proceedings of the 4th Workshop on Embedded Networked Sensors* (New York, NY, USA, 2007), ACM, pp. 93–97.