Industry Adoption of the Internet of Things: A Constrained Application Protocol Survey

Christian Lerche
Universität Rostock, Germany
Email: christian.lerche@uni-rostock.de

Klaus Hartke
Universität Bremen TZI, Germany
Email: hartke@tzi.de

Matthias Kovatsch
Institute for Pervasive Computing
ETH Zürich, Switzerland
Email: kovatsch@inf.ethz.ch

***Abstract*—The Constrained Application Protocol (CoAP) has been designed for RESTful machine-to-machine communication, thereby enabling an Internet of Things. CoAP is based on the principles of the Web, but takes the limited resources of tiny embedded devices such as wireless sensor nodes into account. Despite being relatively new and only about to become an IETF Internet Standard, several implementations of the protocol already exist—each with its own background and supported set of features. In this paper, we give an overview of current CoAP implementations and discuss the results of the first formal interoperability meeting, organized by the European Telecommunications Standards Institute (ETSI) in March 2012. We note that, despite the young age of the protocol, interoperability between the participating implementations is very high, although the non-essential parts of the protocol currently receive significantly less coverage and exhibit slightly more interoperability problems.**

I. Introduction

The vision of an Internet of Things is turning into a reality with tiny embedded devices that are directly connected to the Internet over IP. The Internet Engineering Task Force (IETF) has been standardizing several protocols to incorporate resource-constrained devices that are very limited in energy, memory, computational power, and bandwidth: 6LoWPAN [11] to adapt IPv6 to low-power, lossy networks (LLNs), RPL [24] to route over the fluctuating links, and compression to optimize higher protocol layers [1]. The emerging Constrained Application Protocol (CoAP) [20] is completing this stack to have a fully standardized protocol suite. This makes the IP-based IoT interesting for industrial application: industry standards bodies such as ETSI M2M[1] and the IPSO Alliance[2] adopted the light-weight RESTful protocol to achieve global interoperability for networked embedded systems.

When the idea of an Internet of Things came up, everyday objects were at first interconnected through their virtual representations using barcodes and later RFID. Later, application-level gateways where used to bridge between different communication solutions for embedded systems. Then came compact implementations of IP [4], [10] for resource-constrained devices and things could be accessed directly over IP. The standardization work of the IETF only reached up to the transport layer, though. So projects continued to implement custom application protocols over UDP, with the result that 6LoWPAN devices were not interoperable.

[1]http://www.etsi.org/website/technologies/m2m.aspx
[2]http://www.ipso-alliance.org

A prominent candidate to change the situation at the application layer is HTTP. The Web protocol is very scalable, robust, and virtually ubiquitous, being the de facto application layer of the Internet. However, the performance of TCP in LLNs is up to five times worse than UDP [10]. TCP also requires a comparatively large amount of memory per connection. General-purpose 6LoWPAN implementations are thus not very efficient in combination with HTTP. There were HTTP implementations over UDP, such as HTTPU known from UPnP [22] or the EBHTTP draft [21], but none were adopted in a larger scale. Once the embedded TCP/IP suites mature, cross-layer optimization might become an option to countervail HTTP performance for low-power networks. Smews [5], for instance, achieves reasonable performance with up to 256 parallel HTTP connections.

Another candidate is CoAP, an application-layer protocol similar to HTTP, but specifically designed for constrained nodes and networks. Unlike previous attempts, CoAP does not try to compress HTTP, but is a new protocol based on the same architectural principles of REST [7]. This means that it has a few limitations compared to HTTP, but also that it does not inherit all design decisions and thus can be designed to better address the requirements of embedded devices. In particular, CoAP also exceeds the capabilities of HTTP in regard to machine-to-machine communication (M2M), for instance, through its support for push notifications and IP multicast.

Having two RESTful protocols to cover the full spectrum of device types, the IPSO Alliance has started the next step for device interoperability in commercial products. The so-called "IPSO Profile" defines a standard resource structure for embedded Web servers. It uses Web Linking [16] and the CoRE Link Format [19] to define resource types with well-known functionality and content types. At the time of writing, it covers device and location information (e.g., manufacturer, battery level, and coordinates), messaging for status updates and alarms, general purpose I/O, power and light control, and generic sensors. For the latter, the profile adopts the Unified Code for Units of Measure (UCUM) specification[3], which provides unique string identifiers for physical units (e.g., `W` for Watt or `Cel` for degree Celsius) including prefixes (e.g., `M` for mega).

[3]http://unitsofmeasure.org

In this paper, we present a survey on the current state of the art of lightweight REST implementations with insights on the first formal interoperability event for CoAP. First, we give a brief introduction to CoAP in Section II. The survey in Section III gives an overview of CoAP's early industry adoption as well as available open-source implementations. We also analyze the outcome of the ETSI Plugtest held in Paris, France in March 2012. Our results in Section IV go beyond the information in the official ETSI report [6], as they distinguish between different implementations and not just companies, and are discussed from a developer's point of view. Finally, we summarize our experience and give take-away points in our conclusion in Section V.

II. Constrained Application Protocol

The Constrained Application Protocol (CoAP) is the result of the work of the "Constrained RESTful Environments" (CoRE) working group at the IETF. It is an application-layer protocol that aims to enable RESTful interactions like HTTP, while being more suitable for the low bandwidth and implementation limitations of highly resource-constrained devices and networks.

CoAP consists of two sub-layers: a messaging layer and a request/response layer. The messaging layer adds a thin control layer on top of UDP that provides duplicate detection and, if desired, reliable delivery of messages based on a simple stop-and-wait retransmission with exponential back-off. The request/response layer enables RESTful interaction through uniform interfaces addressed by URIs, well-known methods such as `GET`, `PUT`, `POST` and `DELETE`, and the transfer of self-describing representations of the addressed information (resources). To be more light-weight than HTTP, CoAP supports only a constrained subset of HTTP's features and uses a compact, binary encoding that was designed with serialization and parsing on small devices in mind.

In addition to this core functionality, CoAP provides the following features:

1) Resource observation: CoAP enables clients to "observe" resources for state changes through a simple publish/subscribe mechanism [9]. The server keeps track of interested clients and pushes a resource representation to each client whenever the observed resource changes. The mechanism follows a best-effort approach and aims to guarantee eventual consistency of the state observed by the client and the actual resource state.

2) Block-wise transfers: When resource representations become larger than can be comfortably transported in one datagram, CoAP provides a mechanism to send them in a block-wise fashion [3]. This enables, e.g., firmware updates without resorting to an alternate protocol to transfer large data to or from devices. Block-wise transfers also enable the incremental generation of large representations on the fly, which is beneficial for devices which can only process a limited amount of data at a time.

3) Group communication: CoAP is intended to provide group communication based on IP multicast. However, only some aspects of this feature have been specified so far. [18]

4) Resource discovery: For the discovery of resources that a server provides, CoAP makes use of a well-known URI path `/.well-known/core` following RFC 5785 [17]. The resources are described in the CoRE Link Format [19] which is based on Web Linking [16]. It defines additional link attributes for a semantic resource type ("rt"), interface usage ("if"), content format ("ct"), and the maximum expected size ("sz") of a resource. The list of resources can be filtered.

III. Survey of CoAP Implementations

We conducted a survey among participants of the first formal CoAP interoperability event organized by the European Telecommunications Standards Institute (ETSI) [6] and the IPSO Interop Event, both held in Paris, France between March and April 2012. The events were attended by several international companies and research institutions of which 15 participated in our survey. Table I summarizes the questionnaires in which we asked about basic data such as programming language and supported platforms, but also the targeted application domain.

A. Covered Device Classes and Domains

The goal of CoAP is to provide RESTful interaction for "resource-constrained devices." To better describe this rather fuzzy term, the IETF "Light-Weight Implementation Guidance" (LWIG) working group defined two classes of low-end devices [2]:

- **Class 1**: $\sim$10 kB of data and $\sim$100 kB of code
- **Class 2**: $\sim$50 kB of data and $\sim$250 kB of code

(The computational power is unspecified, as it is secondary to protocol implementation considerations with current chips.) Depending on the running application, the processors usually vary between 8-bit microcontrollers and low-power ARM cores such as the Cortex-M0. However, the full ecosystem of CoAP nodes typically encompasses not only constrained devices, but also higher-end nodes in various roles.

In the survey, we identified the following roles (from large to small):

- Often, CoAP-based applications are built around *back-end systems* that are mostly implemented in Java or C++ and run on a resource-rich machines.
- Mobile devices such as smartphones and tablets are mostly used as *user agents* to commission and manage the nodes in the network.
- Some vendors specialize in the implementation of *proxies* that bridge the gap between the CoAP and the HTTP world. These proxies run on embedded (but still powerful) systems such as routers and access points.
- Actual "things" start with *non-constrained embedded devices*, which are part of industrial machines, for instance, for factory automation.
- Class 1 and 2 devices are used for *sensors and simple actuators*. These are mainly programmed in C.

TABLE I
CoAP Implementations

Company / Implementation	*License*	*Language*	*Platform*	
Consorzio Ferrara Ricerche		NesC / C	TinyOS	Own “SiGLoWPAN” IPv6/6LoWPAN stack for Class 1 devices
ETH Zürich “Californium”	3-clause BSD	Java	JVM	Framework for unconstrained devices; provides client, server, and proxy stubs
ETH Zürich “Copper”	3-clause BSD	JavaScript	Firefox	Management and testing tool as a browser extension; focus on user interaction
ETH Zürich “Erbium”	3-clause BSD	C	Contiki	Class 1 devices such as sensor nodes
Hitachi		C	Embedded Linux	Chipset vendor platform for Class 2 and larger; server-only implementation for embedded sensor devices
IBBT		C++	Click Modular Router	Framework for unconstrained devices; can be configured as client, server, or proxy and can also take the role of a border router
Intecs	Commercial	C++	POSIX	Back-end systems; embedded proxies
KoanLogic “evcoap”	2-clause BSD	C	Linux	General purpose protocol implementation
NXP		Visual C	Windows XP	Application-layer gateway to JenNet-IP devices
Patavina Technologies	Commercial	C++	proprietary OS	Wired and wireless embedded devices and sensor nodes; working on a port to uC/OS by Micrium
Sensinode “NanoService Device Library”	Commercial	C		OS-independent library for Class 1 and 2 devices
Sensinode “NanoService Device Library”	Commercial	Java	JVM	Protocol implementation for unconstrained devices; for embedded PCs, smartphones/tablets, and back-end systems
Universität Bremen TZI “libcoap”	GPLv2 and 2-clause BSD	C	POSIX and Contiki	General purpose library for Class 1 and 2 devices and up
Universität Bremen TZI “CoapBlip”	BSD-style	C	TinyOS	TinyOS-port of “libcoap”; runs on Class 1 devices
Universität Bremen TZI “Bonsai”		C#	.NET	Protocol implementation for unconstrained devices; mainly for the verification of specifications
Universität Bremen TZI “coap.me”		Ruby		Back-end systems; a testing tool at http://coap.me provides an HTTP front-end to crawl CoAP servers, and a CoAP server for interoperability testing
Universität Rostock “jCoAP”	Apache 2.0	Java	JVM	Protocol implementation for unconstrained devices; also targets mobile and embedded platforms
Watteco		C	Contiki	Class 1 devices; based on “Erbium” to provide a CoAP interface for different sensors and actuator products
(Anonymized)		C	UNIX	Client-only implementation for embedded devices
(Anonymized)		Java	JVM	Protocol implementation for mobile and embedded devices

Most of the participants in the CoAP interoperability test provide application-independent stack implementations and services. Sensinode, for instance, offers everything from device libraries to Cloud services. Besides that, there are hardware vendors like Watteco who are evaluating CoAP for their sensing and actuation modules (e.g., CO_2, illuminance, humidity, electical power consumption, etc.) which come with different physical layers (e.g., RF 2.4 GHz, RF 868 MHz, or Powerline). Other vendors are extending their solutions, such as home and building automation systems, with CoAP. For example, NXP is building an application-level gateway that connects their SNAP devices based on JenNet-IP and SNMP (e.g., light bulbs) to the Internet using CoAP. The academic participants were mainly from the Wireless Sensor Networks and Web of Things research communities.

B. Open-Source Implementations

Just as vendors found different niches for which they provide their solutions, the available open-source implementations also follow different goals. This paper gives guidance on which implementations appear to be useful for what kind of projects. We focus on the implementations that were present at the ETSI event and are regularly updated to the latest draft versions. Additional projects can be found in the survey by Villaverde et al. [23]. We note, however, that their list of available CoAP implementations includes information that is partly inaccurate.

1) Californium: Californium[4] [14] is a CoAP framework for Java developers. It supports the observation of resources and block-wise transfer of data. The framework automatically generates the CoRE Link Format for its resources, allows filtering for `/.well-known/core`, and can parse the link-format into stub objects for resource discovery.

The goal of Californium is to provide an API that allows the creation of clients and servers with minimal effort and knowledge about CoAP by developers. Being feature-rich, the framework has comparatively high memory requirements and is used best for back-end systems. The main deficiency is that there is no multi-threading support by the framework and

[4]https://github.com/mkovatsc/Californium

concurrency has to be managed by the resource handler implementations. At the time of writing there was an experimental branch, though, which adds a multi-threaded design together with cross-protocol proxy functionality.

2) jCoAP: jCoAP[5] is a Java implementation for non-constrained devices and embedded systems such as Java-based smartphones and mobile devices (e.g., Android). It also includes a CoAP-to-HTTP and HTTP-to-CoAP proxy implementation which can perform protocol translations between the two protocols. This allows HTTP user agents (such as Web browsers) to access resources on CoAP servers, and conversely CoAP clients to access HTTP resources. As a result, clients can access resources over both protocols without the need to implement them at the same time.

3) Erbium: Contiki's Erbium REST Engine[6] [13] mostly avoids CoAP-specific calls in the application code. It provides a REST-centric API to define resource handlers, access the header options, and process the payload. Erbium is set up to enable replacing CoAP with HTTP by simply linking a different module. At the time of writing, there is no HTTP engine available, though.

Contiki[7] is a light-weight operating system from the wireless sensor network community, so Erbium is well-suited for Class 1 devices with a variety of supported platforms. All of CoAP's features are implemented, although resource observation only is provided on the server side; an easy-to-use client API for it is still missing.

4) libcoap: libcoap[8] [15] is a library for CoAP message parsing, serialization, and transmission. It is very flexible and portable and has been ported to different embedded system architectures, in particular the operating systems Contiki and TinyOS. However, libcoap requires more boilerplate code than Erbium to implement clients and servers. Support for the TinyOS blip-rpl stack is provided by CoapBlip[9].

TinyOS[10] is an operating system for Class 1 devices using a specialized C dialect called NesC. The language uses a components and wiring concept that is reminescent of VHDL. Thus, CoapBlip is recommended for developers that already have experience with TinyOS or want to reuse other projects from the wireless sensor network community around TinyOS.

5) evcoap: evcoap[11] is another general-purpose library supporting the full feature set. Its event handling machinery is based on libevent[12] version 2.0.

Unlike the previous implementations, evcoap's background lies in embedded Web servers rather than wireless sensor networks. It is mainly used to explore experimental mechanisms for the CoAP protocol suite, for example, the Subscribe and Monitor options [8] that target sleepy nodes with a different radio duty cycling model: In wireless sensor networks, duty-cycled nodes are virtually always on, with extremely short channel checks at a few Hertz to allow for idle duty cycles way below one percent. Sleepy nodes turn off their radios entirely for longer periods in the order of hours and hence have to be treated differently.

6) Copper: Copper[13] [12] is a Firefox add-on written in JavaScript. It only implements the client side, but provides a graphical user interface (GUI) for all CoAP features including resource observation and the block-wise transfer of data. Copper also provides renderers for a number of content types such as JSON or the CoRE Link Format. This makes it a useful testing tool for application as well as protocol development.

IV. Results of the ETSI Plugtest

A. ETSI Plugtests

The European Telecommunications Standards Institute (ETSI) is a non-profit standards organisation. It organizes a series of events called ETSI Plugtests to enable interoperability of telecommunication technologies in a multi-vendor, multi-network or multi-service environment. The first IoT CoAP Plugtest meeting was held in conjunction with the IETF #83 Meeting in Paris, 24-25 March 2012. Most companies and universities listed in Table I participated in this event.

B. Setup of the CoAP Plugtest

The Plugtest was driven by a test specification [6] which defined the test cases to be performed as well as the test environment.

The test specification defined 16 mandatory test cases and a further 11 optional test cases. Each test case was to be performed by pairs of one CoAP client and one CoAP server of different vendors. The mandatory tests covered the CoAP core specification [20] including basic operations on resources, separate responses, header option processing and retransmissions in lossy networks. The optional tests covered resource observation [9], block-wise transfers [3], and the CoRE Link Format [19]. Table II provides an overview of all tests defined by the test specification.

The result of each test performed was recorded after each session in a database. The possible results were as follows:

- the test passed,
- the test failed,
- the test was not applicable (for example, because the feature was not implemented), or
- the time for testing ran out.

Participants brought their own devices of various sizes. They were connected to a single LAN, which provided the environment for most test cases. For test cases concerning operations in a lossy context, a *lossy gateway* was provided to emulate a lossy link by randomly dropping packets according to a configurable rate (around 80% during the tests).

[5]http://code.google.com/p/jcoap/
[6]http://contiki.git.sourceforge.net/
[7]http://www.contiki-os.org/
[8]http://libcoap.sourceforge.net/
[9]http://www.comnets.uni-bremen.de/~mab/git/tinyos-main.git
[10]http://www.tinyos.net/
[11]https://github.com/koanlogic/webthings/tree/master/bridge/sw/lib/evcoap
[12]http://libevent.org/
[13]https://github.com/mkovatsc/Copper

TABLE II
TEST CASES

TestID	Description	passed/ failed
Mandatory		2586/163
CORE_01	Perform GET transaction (CON mode)	176/8
CORE_02	Perform POST transaction (CON mode)	176/7
CORE_03	Perform PUT transaction (CON mode)	175/7
CORE_04	Perform DELETE transaction (CON mode)	180/3
CORE_05	Perform GET transaction (NON mode)	174/9
CORE_06	Perform POST transaction (NON mode)	167/16
CORE_07	Perform PUT transaction (NON mode)	169/14
CORE_08	Perform DELETE transaction (NON mode)	172/10
CORE_09	Perform GET transaction with delayed response (CON mode, no piggyback)	151/27
CORE_10	Handle request containing Token option	167/5
CORE_11	Handle request not containing Token option	181/0
CORE_12	Handle request containing several Uri-Path options	170/5
CORE_13	Handle request containing several Uri-Query options	152/11
CORE_14	Interoperate in lossy context (CON mode, piggybacked response)	119/3
CORE_15	Interoperate in lossy context (CON mode, delayed response)	98/19
CORE_16	Perform GET transaction with delayed response (NON mode)	159/19
Optional		266/26
LINK_01	Access to well-known interface for resource discovery	39/2
LINK_02	Use filtered requests for limiting discovery results	32/4
BLOCK_01	Handle GET blockwise transfer for a large resource (early negotiation)	34/2
BLOCK_02	Handle GET blockwise transfer for a large resource (late negotiation)	33/2
BLOCK_03	Handle PUT blockwise transfer for a large resource	15/6
BLOCK_04	Handle POST blockwise transfer for a large resource	15/6
OBS_01	Handle resource observation	25/0
OBS_02	Stop resource observation	23/2
OBS_03	Client detection of deregistration (Max-Age)	13/0
OBS_04	Server detection of deregistration (client OFF)	19/2
OBS_05	Server detection of deregistration (explicit RST)	18/0

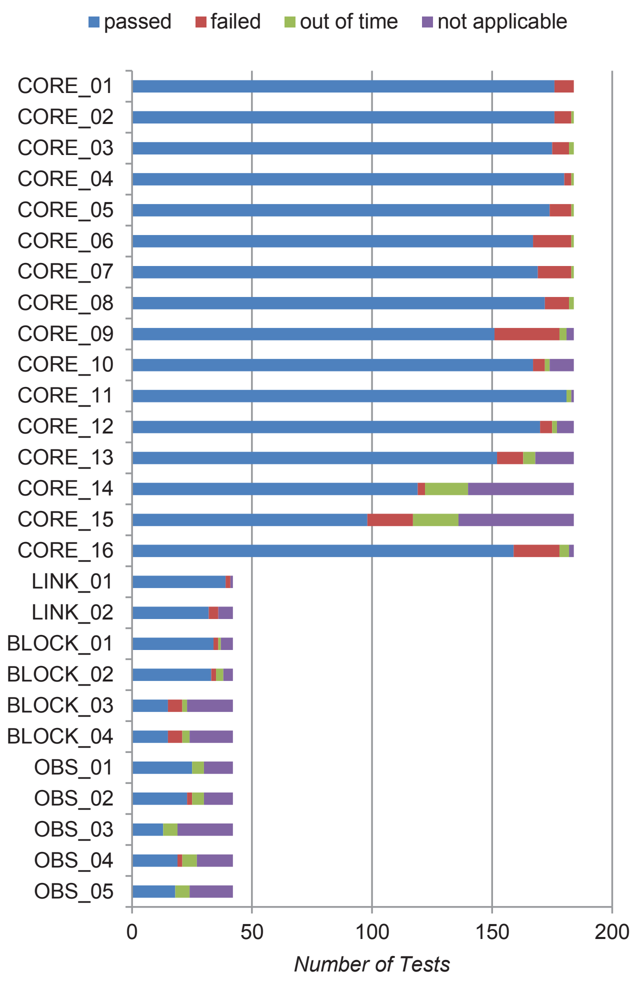

Fig. 1. CoAP Plugtest Results

C. Plugtest Results

From the Plugtest result database, we identified 18 CoAP server implementations and 16 client implementations. This means there were 288 possible client-server combinations. As many libraries support both client and server implementations, this includes self-tests which were not included in the testing schedule of the Plugtest meeting. Some participants provided their self-test results, though. Beyond this, also tests between different implementations from the same company were not considered by the schedule. Furthermore, some tests were performed several times. We assume that some issues were found and fixed immediately during the Plugtest; for our evaluation, we ignore all results except for the most recent ones.

In total, 3406 Tests were made during the Plugtest. 365 of them had the result “not applicable” or “out of time”. These are ignored in the following evaluation, resulting in a number of considered tests of 3041.

Figure 1 shows all test results in detail; Figure 2 provides a summary of the results. As can be seen, in most cases more than 90% of the tests performed did pass. This means that a high interoperability is given. But it can also be seen that the mandatory test cases were performed much more often than the optional ones. On average, mandatory test cases were performed 171 times, while optional test cases were performed only 27 times. Assuming that the optional tests were not performed because the respective feature were not implemented, it can be concluded that fewer CoAP implementations support optional features. However, if they are implemented, a high interoperability is given here as well.

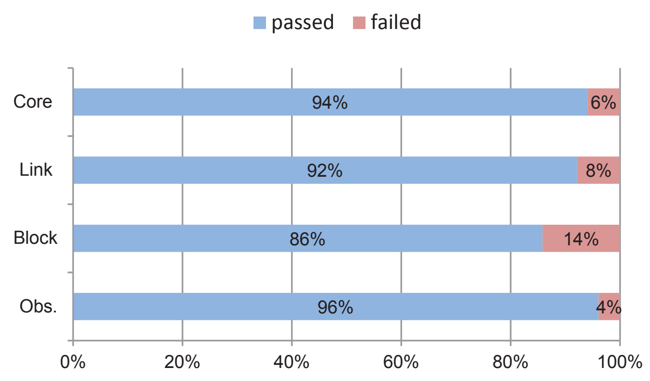

Fig. 2. CoAP Plugtest Results Summary

D. *Lessons learned*

As might be expected, the test case with the highest success rate is a simple GET request without a Token option (CORE_11). This test never failed. The mandatory test that performed worst is a separate response in a lossy context (CORE_15). However, this test case still yielded a high interoperability with 84% of the tests performed being successful.

With only 71% of the tests performed being successful, the test cases BLOCK_03 and BLOCK_04 are those with the overall worst outcome compared to the other test cases. Both of these test cases involving the block-wise transfer of data from a client to a server.

Summing up, it can be stated that a very high interoperability between current CoAP implementations is given. Except for the two test cases involving the block-wise transfer of data from clients to servers, all test cases passed with more than 80% of the tests performed being successful. This implies that the current draft specification is quite clear and without ambiguity.

V. Conclusion

In this paper we gave an overview of current CoAP implementations. Although CoAP is a very new protocol, over 20 different implementations in various programming languages are known for devices ranging from resource-constrained devices such as wireless sensor nodes up to smartphones and servers. This implies that CoAP is going to be used in many different applications and domains.

Nearly all of the implementations were present at the first Plugtest meeting organised by ETSI in March 2012. At the Plugtest 18 server and 16 client implementations were tested against each other in 27 different test cases. As a result of the Plugtest evaluation, it can be stated that a high interoperability between current implementations is given. Most tests passed with more than 90%. This also implies a good implementability of the CoAP specification.

However, only a few implementations currently support optional features. But it can be assumed that this is due to the novelty of the specification and many implementations will support optional features in near future.

Acknowledgment

The authors would like to thank the organizers from ETSI and the Plugtest attendees who participated in our survey and made the test results available.

References

[1] C. Bormann. 6LoWPAN Generic Compression of Headers and Header-like Payloads. draft-bormann-6lowpan-ghc-04, 2012.

[2] C. Bormann. Guidance for Light-Weight Implementations of the Internet Protocol Suite. draft-bormann-lwig-guidance-01, 2012.

[3] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. draft-ietf-core-block-08, 2012.

[4] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *Proc. MobiSys*, San Francisco, CA, USA, 2003.

[5] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Smews: Smart and Mobile Embedded Web Server. In *Proc. CISIS*, Fukuoka, Japan, 2009.

[6] ETSI. 1st CoAP Plugtest. Technical Report CTI Plugtest Report 1.1.1 (2012-03), ETSI, 2012.

[7] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *Trans. Internet Technology*, 2(2):115–150, 2002.

[8] T. Fossati, P. Giacomin, and S. Loreto. Publish and Monitor Options for CoAP. draft-fossati-core-publish-monitor-options-01, 2012.

[9] K. Hartke. Observing Resources in CoAP. draft-ietf-core-observe-05, 2012.

[10] J. Hui and D. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proc. SenSys*, Raleigh, NC, USA, 2008.

[11] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC6282, 2011.

[12] M. Kovatsch. Demo Abstract: Human–CoAP Interaction with Copper. In *Proc. DCOSS*, Barcelona, Spain, 2011.

[13] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proc. MASS*, Valencia, Spain, 2011.

[14] M. Kovatsch, S. Mayer, and B. Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Proc. IMIS*, Palermo, Italy, 2012.

[15] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proc.IP+SN*, Chicago, IL, USA, 2011.

[16] M. Nottingham. Web Linking. RFC5988, 2010.

[17] M. Nottingham and E. Hammer-Lahav. Defining Well-Known Uniform Resource Identifiers (URIs). RFC5785, 2010.

[18] A. Rahman and E. Dijk. Group Communication for CoAP. draft-ietf-core-groupcomm-01, 2012.

[19] Z. Shelby. CoRE Link Format. draft-ietf-core-link-format-11, 2012.

[20] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). draft-ietf-core-coap-09, 2012.

[21] G. Tolle. Embedded Binary HTTP (EBHTTP). draft-tolle-core-ebhttp-00, 2010.

[22] UPnP Forum. UPnP Device Architecture 1.0. Document Revision Date 15 October 2008.

[23] B. Villaverde, D. Pesch, R. Alberola, S. Fedor, and M. Boubekeur. Constrained application protocol for low power embedded networks: A survey. In *Proc. IMIS*, Palermo, Italy, 2012.

[24] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC6550, 2012.