A Low-Power CoAP for Contiki

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

Simon Duquennoy, Adam Dunkels
Swedish Institute of Computer Science
Kista, Sweden
Email: {simonduq,adam}@sics.se

***Abstract*—Internet of Things devices will by and large be battery-operated, but existing application protocols have typically not been designed with power-efficiency in mind. In low-power wireless systems, power-efficiency is determined by the ability to maintain a low radio duty cycle: keeping the radio off as much as possible. We present an implementation of the IETF Constrained Application Protocol (CoAP) for the Contiki operating system that leverages the ContikiMAC low-power duty cycling mechanism to provide power efficiency. We experimentally evaluate our low-power CoAP, demonstrating that an existing application layer protocol can be made power-efficient through a generic radio duty cycling mechanism. To the best of our knowledge, our CoAP implementation is the first to provide power-efficient operation through radio duty cycling. Our results question the need for specialized low-power mechanisms at the application layer, instead providing low-power operation only at the radio duty cycling layer.**

***Keywords*-Internet of Things; Web of Things; CoAP; embedded Web services; energy; radio duty cycling;**

I. Introduction

The Internet of wireless things needs power-efficient protocols, but existing protocols have typically been designed without power-efficiency in mind. In low-power wireless systems, the radio transceiver is typically the most power-consuming component, so power-efficiency translates into efficient radio duty cycling: the ability to keep the radio off as much as possible.

The Internet protocol stack is a suitable solution to realize an Internet of Things (IoT), a network of tiny networked embedded devices that create a link to the physical world. The narrow waist of IP can be used to directly access sensor readings throughout a sustainable city, acquire the necessary information for the smart grid, or control smart homes, buildings, and factories. The stack's layered architecture helps to manage the complexity.

We have implemented the IETF Constrained Application Protocol (CoAP) [11] for Contiki, which enables interoperability at the application layer through RESTful Web services. As depicted in Figure 1, we have integrated a full protocol stack necessary for an IoT and evaluated the system performance from an application layer perspective.

Layer	*Protocol*
Application	IETF CoAP / REST Engine
Transport	UDP
Network	IPv6 / RPL
Adaptation	6LoWPAN
MAC	CSMA / link-layer bursts
Radio Duty Cycling	ContikiMAC
Physical	IEEE 802.15.4

Figure 1. Low-power operation is done only in the Radio Duty Cycling (RDC) layer, thereby separating low-power operation from the application layer. This reduces complexity and follows the layered architecture that allowed the Internet to evolve.

The contribution of this paper is that we are the first to demonstrate power-efficient CoAP operation through radio duty cycling. These results challenge the need for specialized power-management at the application layer: by constraining power management to the radio duty cycling layer, complexity can be removed from the application layer.

II. Background

A. Power Efficiency through Duty Cycling

On typical IoT platforms, the radio transceiver is one of the most power-consuming components. Listening is as expensive as receiving packets. To conserve energy, the radio transceiver must be switched completely off for most of the time. Several radio duty cycling (RDC) algorithms have been designed, allowing nodes to keep the radio chip off for more than 99% of the time while still being able to send and receive messages [4], [7].

In this work, we use the ContikiMAC RDC protocol [4]. ContikiMAC is a low-power listening MAC protocol that uses an efficient wake-up mechanism to attain a high power efficiency: with a wake-up frequency of 8 Hz, the idle radio duty cycle is only 0.6% [4].

The ContikiMAC principle of operation is shown in Figure 2. Nodes periodically wake up to check the radio channel for a transmission from a neighbor. If a radio signal is sensed, the node keeps the radio on to listen for the packet. When the data frame is received, the receiver sends an acknowledgment frame. To send a packet, the

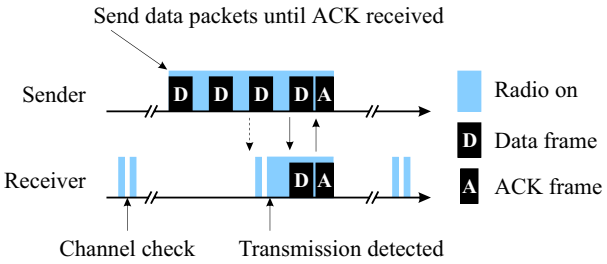


Figure 2. A ContikiMAC sender wakes its neighbors up by sending a strobe of data frames until it gets an acknowledgment.

sender repeatedly sends the data frame until it receives an acknowledgment, or until the packet was sent for an entire wake-up period without an acknowledgment being received.

B. The Constrained Application Protocol

The IETF Constrained Application Protocol (CoAP) is an application-layer protocol designed to provide a REST-like interface [5], but with a lower cost in terms of bandwidth and implementation complexity than HTTP-based REST interfaces. CoAP adopts patterns from HTTP such as resource abstraction, URIs, RESTful interaction, and extensible header options, but uses a compact binary representations that are designed to be easy to parse. Unlike HTTP over TCP, CoAP uses UDP. This makes it possible to use CoAP in one-to-many and many-to-one communication patterns.

Central CoAP mechanisms are:

1) Applications can send CoAP messages reliably ("confirmable") or non-reliably ("non-confirmable"). Confirmables are retransmitted with exponential timeouts until acknowledged by the receiver or reaching the maximum number of retransmissions.
2) CoAP is intended to provide group communication via IP multicast, but this mechanism has not yet been specified.
3) CoAP features native push notifications through a publish/subscribe mechanism called "observing resources" [6]. Clients can send a request with an observe header option to a CoAP resource. The server keeps track of these subscribers and sends a response whenever the observed resource changes.
4) For resource discovery, CoAP follows RFC 5785 by using the `/.well-known/core` path to provide resource descriptions in its CoRE Link Format [10]. This format extends Web Linking [9] and defines attributes for a semantical type ("rt"), interface usage ("if"), content-type ("ct"), and the maximum expected size ("sz") of a resource. In addition, a directory service is intended.

When RAM for IP and application buffers is limited, devices can only process a specific amount of data at a time. Larger data can be handled by storing these "chunks" in flash memory, for instance to receive a new firmware or to provide a full datalog. To avoid the need of a secondary protocol to exchange these data, CoAP specifies a simple stop-and-wait mechanism called "blockwise transfers" [1].

III. A Low-power CoAP for Contiki

We have implemented CoAP for the Contiki operating system, taking advantage of the Contiki REST layer abstraction. This layer provides a generic abstraction for RESTful applications [12]. Our CoAP implementation is available from a public Git repository[1]. At the time of writing, the code implements the CoAP draft specification version 07, released on 8 Jul 2011.

A. The Contiki REST Engine

Our REST Engine is an improvement of Contiki's REST layer by Yazar and Dunkels [12]. It provides macros to define and automatically instantiate RESTful Web service resources. The layering now follows Contiki's network stack model. This way, the application code is fully decoupled from the underlying protocol, and either CoAP-03, CoAP-07, or HTTP can be linked to implement the RESTful Web services. We provide the necessary mapping of RESTful methods, status codes, header options, query variables, and so forth to their CoAP and HTTP representations.

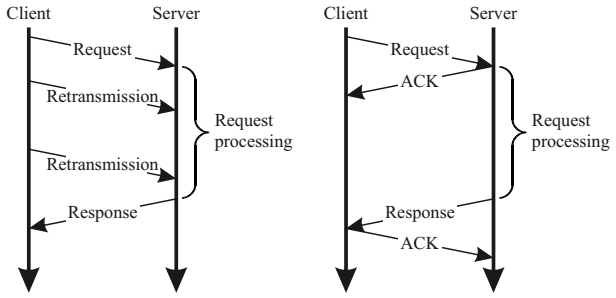
The new REST Engine offers three abstractions to create RESTful resources:

RESOURCE: A basic REST resource is defined by URI-path, allowed methods, and a string for the Web Linking [9] information. For every resource, the application must provide a resource handler function, which receives the request and generates the corresponding response. Both messages are accessed through the REST Engine API, which hides the actual implementation (e.g., `REST.set_header_etag(response, etag_buf, etag_len)`).

EVENT_RESOURCE: This abstraction requires a second handler function to be implemented by the application developer. A user-defined event triggers this handler, which can be a button press or a PUT to another resource that caused a status update.

PERIODIC_RESOURCE: Additional to the signature of the basic `RESOURCE`, the last macro takes a time interval. With it, the REST Engine periodically calls a second handler function similar to the one for events. This function can be used to poll on-board sensors and for instance perform a threshold check whether the resource should be considered as changed.

[1] https://github.com/mkovatsc/SmartAppContiki



(a) Without separate response (b) With separate response

Figure 3. With separate responses, the server can notify the client that it received the request, enabling long processing times and avoiding unnecessary retransmissions.

In the end, a typical RESTful Web service application consists of a single C-file. It contains the resource macros together with their handler functions and one Contiki process that initializes the REST Engine, activates the resources, and optionally waits for user events for any `EVENT_RESOURCE`.

B. Blockwise Transfers

Our CoAP implementation supports blockwise transfers [1]. If a response generated by a resource handler exceeds the client's requested block size, the REST Engine automatically divides the response without involvement of the handler.

Resources can also process larger data in a chunk-wise manner. For this, we manage the byte offset and preferred chunk size for the handlers and map their chunks to the blockwise transfer.[2]

C. Observing Resources

We implemented CoAP's publish/subscribe mechanism [6] as a resource post-handler that automatically registers subscriptions when a request was successful. This post-handler is configured by default for `EVENT_` and `PERIODIC_RESOURCE`s. Their event and polling handlers then can use `REST.notify_subscribers()` to publish changed resource representation to all subscribers.

D. Separate Responses

Some resources may require a long processing time or wait for hardware resources, such as a slow sensor, for an unknown time. This could cause several unnecessary request retransmissions or even a request timeout at the client before the server can send the response. Thus, CoAP provides "separate responses", for which the server sends an empty ACK instantly and a confirmable message when finished (see Figure 3). Our implementation provides a pre-handler that takes care of the ACK and the configuration of the actual response with new message ID and type. We envisioned to implement a benchmarking mechanism that automatically determines response separation for resources. Due to our findings discussed in Section IV-E, it would, however, only consume precious memory. We recommend to configure the pre-handler simply for every resource that might have a processing delay, for instance because of a slow sensor.

E. Resource Discovery

The CoRE Link Format [10] is generated automatically for all resources. Our handler for `/.well-known/core` also respects chunk-wise processing and generates the required substrings[3] within the bounds of the buffer provided by the REST Engine.

F. CoAP Clients

The REST Engine makes implementation of CoAP clients easy. We provide a blocking function call implemented with protothreads to issue a request. This linear programming model can also hide blockwise transfers, as it continues first when all data were received.

G. Message Deduplication

The current implementation follows the consideration of the CoAP specification to relax duplicate filtering. Contiki is meant for memory-constrained devices. Here, the management of several Internet IPv6 addresses would pose a massive overhead while requests are typically idempotent.

H. Memory Management

Contiki's cooperative multi-threading allows us to provide access to incoming payloads and byte strings (e.g., the ETag, but also parsed query variables) directly in the IP buffer. This in-place processing saves an additional application layer buffer. For the response generation, our REST Engine organizes buffers for the resource handlers. The buffers are reused to serialize the CoAP message in-place and store confirmables for retransmission. Our REST Engine lets developers define the required chunk size for their application and then will dimension the required buffers accordingly. The required IP buffer size is checked as well.

Numeric header options are parsed and set with additional integer variables. This eases read and write access for the application, as CoAP uses a special differential and run-length encoding for header options.

[2]For an HTTP implementation, the REST Engine maps the chunks to TCP segments to respect the IP buffer constraints.

[3]The Link Format is basically a long, easily parsable string consisting of all the resource paths and their Web Linking attributes.

	ROM (kB)	*RAM (kB)*
CoAP REST Engine total	8.5	1.5
Measured stack usage	-	0.1
REST Engine	0.7	0.0
CoAP-07 base	4.5	0.0
CoAP-07 server	1.9	0.3
CoAP-07 transport	0.4	0.9
CoAP-07 observing	0.9	0.2
CoAP-07 separate	0.1	0.0

Table I. Our CoAP implementation including the REST abstraction consumes 8.5 kB of ROM and 1.5 kB of RAM. The required stack is small due to in-place processing and statically allocated buffers.

IV. Evaluation

We evaluate both static and dynamic properties of our low-power CoAP: memory footprint, energy consumption, and data throughput.

A. Experimental Setup

We run all our experiments on Tmote Sky sensor motes. The platform is based on a MSP430 16-bit CPU running at 3.9 MHz. It provides a CC2420 radio chip, 48 kB of program flash and 10 kB of RAM. We use a small, linear 4-hop network with static routes. One Tmote Sky implements the 6LoWPAN border router connected to a computer running Linux. The 802.15.4 radio is configured to channel 15, which underlies WiFi interference. Unless explicitly mentioned, we always use ContikiMAC and set the listener wake-up frequency to 8 Hz, which corresponds to a 0.6% idle duty cycle.

The results displayed are averaged over 100 runs and error bars show the standard deviations. We characterize the motes energy consumption via Contiki's built-in energy profiler Powertrace [3].

B. Memory Footprint

Table I shows the detailed memory footprint of our CoAP implementation for Contiki. The code is compiled with msp430-gcc (GCC) 3.2.3 for the Tmote Sky.

In total, our implementation requires 8.5 kB of ROM and 1.5 kB of RAM (heap plus measured maximum stack for all CoAP and REST related functions). The CoAP-07 base, which handles parsing, manipulation, and serialization of CoAP messages, contains the largest amount of program code, followed by the server module, which connects to the IP stack. The CoAP-07 transport module requires the largest amount of RAM, as it provides the mentioned message buffers. With the used default settings, four confirmable messages with a chunk size of 128 bytes can be stored at a time.

Being decoupled from the protocol and only using the REST Engine API, the application code is compact. A RESTful application with five resources as depicted in Table II (in addition to the `/.well-known/core` resource) only consumes 1.5 kB of ROM and 160 B of heap.

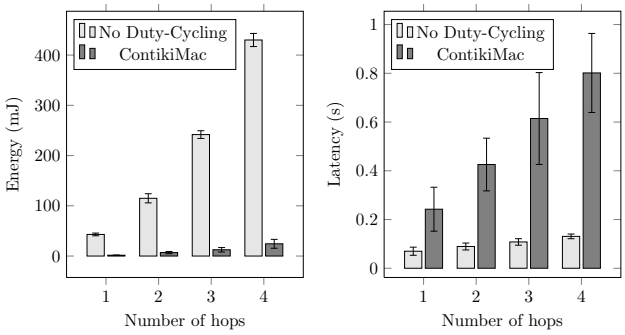
URI-Path	*Functionality*
`/battery`	Voltage level as text or JSON
`/event`	Button press notifications
`/leds`	Control per key-value pairs in query and payload
`/light`	Both light sensor readings as text or JSON
`/push`	Periodic notifications

Table II. Our measured example application provides five non-trivial resources with Link Format descriptions between 32 and 85 bytes. As the CoAP REST Engine provides most functionality for RESTful Web services, typical applications only use 1–2 kB of ROM.

For the implementation of an IoT system on memory-constrained devices, the maximum supported chunk size, i.e. CoAP block size, is a central design parameter. It decides over the IP buffer size as well as the required buffers for resource handlers and retransmissions. Due to CoAP's exponential block sizes between 16 and 1024 bytes, the steps above 128 bytes appeared to be too coarse-grained and hence limit the design space. It becomes hard to optimize the memory distribution among different system components and to adjust to different hardware resources.

C. Energy Consumption

To evaluate the trade-off between energy consumption and latency, we set up an experiment in which we issue a request to different motes in our 4-hop network, with ContikiMAC disabled (best case latency) and enabled (0.6% RDC). The requested CoAP resource responds with an echoed 2-byte token and a fixed payload of 64 bytes. Figure 4(a) shows the cumulated consumption of all motes involved in the process. This includes forwarding nodes as well as the targeted CoAP server. As expected, ContikiMAC saves a lot of energy, reaching an improvement by a factor of 26 compared to no duty cycling. Figure 4(b) shows that the penalty of duty cycling in terms of latency is acceptable with



(a) Energy (b) Latency

Figure 4. The overall energy consumption and latency for a request with 64 bytes payload in the response and no fragmentation. ContikiMAC substantially reduces the motes energy consumption while keeping a reasonable end-to-end latency.

a duty cycle well below the 1% mark.[4] The maximum measured slow-down was about factor 6, keeping the latency under 1 second over 4 hops. We argue that the substantial lifetime increase offered by ContikiMAC is in many cases worth paying this latency overhead.

D. Transmitting Large Data

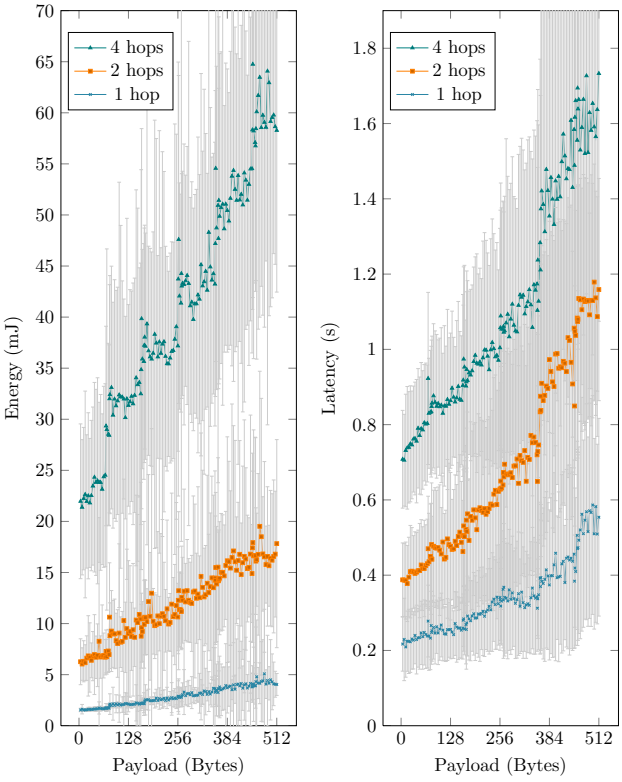
Not all CoAP resource representations can fit into a single 802.15.4 frame, so that either 6LoWPAN fragmentation or blockwise transfer is required. We focus on fragmentation to analyze the energy consumption as a fine-grained function of the payload size using RDC. The energy cost of blockwise transfers would simply correspond to multiple requests with the payload size adjusted for the additional Block2 header option.

To enable energy-efficient transmission of consecutive frames, we use link-layer bursts. When a sender has several frames to send, it first wakes up its neighbors with a ContikiMAC strobe and sets the "frame pending" bit in the 802.15.4 frame header to tell the receiver that another frame will follow. Remaining frames are sent consecutively and acknowledged by the receiver until the frame pending bit is unset.

We ran an experiment in which a client on the computer requests a payload ranging between 1 and 512 bytes from a CoAP server, which also echoes a 2-byte token. The client targets motes at 1, 2, and 4 hops from the border router. We expected to see a step-wise increment of the energy consumption as the number of fragments increases.

Figure 5 shows the cumulative energy consumption of all nodes involved in the request (forwarding and server) and the request-response latency. The latter is almost unaffected by 6LoWPAN fragmentation due to the frame bursts. The increase in energy consumption is, however, clearly noticeable for the first additional fragment (in our configuration at 70 bytes for 4 hops and 79 bytes for 1 and 2 hops). The following per-fragment steps become less distinguishable. This is also due to the link-layer bursts, which, besides time, save the ContikiMAC strobes for every additional fragment. Furthermore, we have a large standard error deviation due to highly varying link quality: the experiment was run in offices and the motes were subject to WiFi interference. When packet loss occurs, ContikiMAC simply transmits additional data frames until getting an acknowledgment. Because of this lossy environment, the total number of frames sent depends more on the local link quality than on fragmentation. As a result, the total energy cost and latency are roughly proportional to the payload size and not discretely to the number of 6LoWPAN fragments.

[4]Note that the actual duty cycle depends on the data traffic.



(a) Energy (b) Latency

Figure 5. In lossy environments, the energy cost and latency when sending large CoAP payloads depends more on the payload size than the number of required 6LoWPAN fragments.

E. Optimizing Separate Responses

We examined the ability of separate responses to save energy by avoiding request retransmissions. This applies for resources that have a significant processing time, which stays below the overall retransmission timeout, though. We targeted a mote 4 hops from the border router. Its CoAP resource takes a variable amount of time to process the request, ranging between 1 and 24 s. We tested two different CoAP configurations: the first does not use separate responses, the second does.

Figure 6 shows the energy consumed by the motes depending on the server-side response delay, without and with separate response. The consumption graphs increase linearly with the delay due to radio idle listening, but also show offsets due to the forwarded (re)transmissions. After about 10 seconds, the two curves cross: at this point, separate responses start saving energy. Before this point, the mechanism spends more energy than it saves because of the extra ACK. In contrast to our expectation, the energy savings by optimizing when to use separate responses are marginal, even for very long-lasting requests. Note, however, that for even longer requests, the mechanism is necessary to avoid request cancellation.

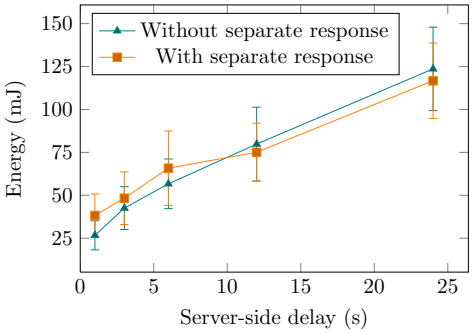


Figure 6. The separate response mechanism can save a little amount of energy for resources that have a long processing time, but would not exceed the overall request timeout.

V. Related Work

There is a body of work in the wireless sensor network field on radio duty cycling mechanisms [4], [7]. We are not the first to argue the benefit of isolating low-power mechanisms to a single layer, but we are the first to demonstrate the implications of duty cycling for IoT application layers, as exemplified by CoAP.

There are many implementations of CoAP, such as libcoap [8], CoAPy[5], jCoAP[6], and Californium[7], but none has been evaluated in terms of low-power behavior in a multi-hop network. Kuladinithi et al. [8] measure the latency of their CoAP implementation, but without any duty cycling and only in a single-hop network. Colitti et al. [2] take a first step towards analyzing the power consumption of the previous CoAP implementation for Contiki [12], but with a simplified power model that only considers application data size and that does not include the full energy consumption by the system. By contrast, we are the first to experimentally evaluate the full system power consumption of a multi-hop IPv6/CoAP network.

VI. Conclusion and Future Work

We presented our low-power CoAP implementation for Contiki that leverages a generic radio duty cycling mechanism to achieve a high energy efficiency. We experimentally evaluated our implementation in a multi-hop network and showed that the use of a duty cycle results in a low power consumption, at the cost of a higher latency. Our protocol-independent REST Engine provides an abstraction to create RESTful Web services. Our CoAP implementation addresses the trade-off between memory-efficiency and a convenient API for developers. In-place processing and the reuse of buffers allows for a small memory footprint. We evaluated our implementation and the general mechanisms of CoAP in a realistic setting: a 0.6% idle RDC, link-layer bursts for fragmented packets, and IP multihop routing on a 802.15.4 channel underlying WiFi interference.

Our experiments confirm that CoAP request/response cycles are most energy-efficient when each message fits into a single 802.15.4 frame. Once 6LoWPAN fragmentation is performed, however, there is no need to optimize the number of fragments. The number of transmitted frames is dominated by the link quality and clock synchronization, which affects the length of the RDC strobe. Consequently, an optimization of the CoAP block size definitions for 6LoWPAN fragments has no significant benefit, at least when link-layer bursts and a sender-initiated RDC layer are used.

Our implementation is available in the official Contiki repository. In future work, we plan to evaluate the possibilities and limitations of the RESTful approach and how the IP-based IoT performs in terms of latency, reliability, and battery-lifetime.

[5]http://coapy.sourceforge.net/
[6]http://code.google.com/p/jcoap/
[7]https://github.com/mkovatsc/Californium/

Acknowledgment

This work has been supported by the Swedish Foundation for Strategic Research (SSF) through the Promos project and the European Commission with contract FP7-2007-2-224053 (CONET).

References

[1] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. draft-ietf-core-block-04, 2011.
[2] W. Colitti, K. Steenhaut, and N. De Caro. Integrating Wireless Sensor Networks with the Web. In *Proc. IP+SN*, Chicago, IL, USA, 2011.
[3] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes. Powertrace: Network-Level Power Profiling for Low-power Wireless Networks. Technical Report T2011:05, SICS, 2011.
[4] A. Dunkels, L. Mottola, N. Tsiftes, F. Österlind, J. Eriksson, and N. Finne. The Announcement Layer: Beacon Coordination for the Sensornet Stack. In *Proc. EWSN*, Bonn, Germany, 2011.
[5] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *TOIT*, 2, May 2002.
[6] K. Hartke and Z. Shelby. Observing Resources in CoAP. draft-ietf-core-observe-02, 2011.
[7] J. Hui and D. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proc. SenSys*, Raleigh, NC, USA, 2008.
[8] K. Kuladinithi, O. Bergmann, T. Ptsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proc. IP+SN*, Chicago, IL, USA, 2011.
[9] M. Nottingham. Web Linking. RFC5988, 2010.
[10] Z. Shelby. CoRE Link Format. draft-ietf-core-link-format-06, 2011.
[11] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). draft-ietf-core-coap-07, 2011.
[12] D. Yazar and A. Dunkels. Efficient Application Integration in IP-Based Sensor Networks. In *Proc. BuildSys*, Berkeley, CA, USA, 2009.