Light-Weight Implementation Guidance
Internet-Draft
Intended status: Informational
Expires: April 18, 2013

M. Kovatsch
ETH Zurich

October 15, 2012

Implementing CoAP for Class 1 Devices
draft-kovatsch-lwig-class1-coap-00

Abstract

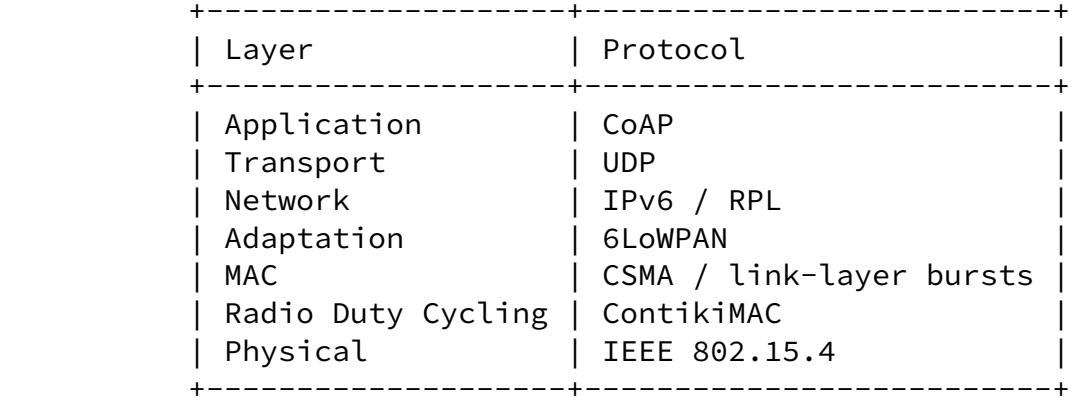
The Constrained Application Protocol (CoAP) is designed for resource-constrained nodes and networks, e.g., sensor nodes in low-power lossy networks (LLNs). Still, to implement this Internet protocol on Class 1 devices, i.e., ~10KiB of RAM and ~100KiB of ROM, light-weight implementation techniques are necessary. This document provides the lessons learned from implementing CoAP for Contiki, an operating system for tiny, battery-operated networked embedded systems. The information may become part of the Light-Weight Implementation Guidance document planned by the IETF working group LWIG.

Table of Contents

1. Introduction 3
2. Implementing CoAP 3
2.1. Memory Management 4
2.2. Message Buffers 4
2.3. Retransmissions 5
2.4. Separate Responses 5
2.5. Deduplication 5
2.6. Observing 6
2.7. Blockwise Transfers 6
2.8. Developer API 7
3. Low-power Wireless 7
3.1. Radio Duty Cycling 8
3.2. Sleepy Nodes 8
4. Security Considerations 9
5. Informative References 9
Author's Address 10

1. Introduction

The Internet protocol suite 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---seamlessly from the existing IT infrastructure. The layered architecture helps to manage the complexity, as multiple aspects such as routing over lossy links, link layer adaption, and low-power communication have to be addressed. Nonetheless, attention has to be given to achieve light-weight implementations that can run on resource-constrained devices such as sensor nodes with only microcontroller units (MCUs), ~10KiB of RAM, and ~100KiB of ROM [I-D.ietf-lwig-guidance]. Figure 1 depicts a typical stack configuration for such Class 1 devices. This document discusses a light-weight implementation of CoAP at the application layer in Section 2 and techniques for energy-efficiency such as radio duty cycling in Section 3.



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

A typical stack configuration for Class 1 devices.

Figure 1

2. Implementing CoAP

The following experience stems from implementing CoAP for the Contiki operating system [ERBIUM], but is generalized for any embedded OS. The information is not meant to be a final solution, but a first step towards a Light-Weight Implementation Guidance contribution. Alternatives will be incorporated throughout the merging process. The document assumes detailed knowledge of CoAP, its message format and interaction model. For more information, please refer to to [I-D.ietf-core-coap], [I-D.ietf-core-block], and [I-D.ietf-core-observe].

2.1. Memory Management

For embedded systems, it is common practice to allocate memory statically to ensure stable behavior, as no memory management unit (MMU) or other abstractions are available. For a CoAP node, the two key parameters are the number of (re)transmission buffers and the maximum message size that must be supported by each buffer. It is common practice to set the maximum message size far below the 1280-byte MTU of 6LoWPAN to allow more than one open confirmable transmissions at a time (in particular for observe notifications). Note that implementations on constrained platforms often not even support the full MTU. Larger messages must then use blockwise transfers [I-D.ietf-core-block], while a good trade-off between 6LoWPAN fragmentation and CoAP header overhead must be found. Usually the amount of available free RAM dominates this decision, on current platforms ending up at a maximum message size of 128 or 256 bytes plus maximum estimated header size.

2.2. Message Buffers

Class 1 devices usually run an OS or event loop system with cooperative multi-threading. This allows to optimize memory usage through in-place processing and reuse of buffers. Incoming payload and byte strings of the header can be directly accessed in the IP buffer, which is provided by the OS, using pointers. For numeric options, there are two alternatives: Either process the header on the fly when an option is accessed or initially parse/allocate all values into a local data structure. Although the latter choice requires an additional amount of memory, it is preferable. First, local

processing anyway requires integers in host byte order and stored in a variable of corresponding type. Second, on-the-fly processing might force developers to set options for outgoing messages in a specific order or cause extensive memmove operations due to CoAP's delta encoding.

CoAP servers can significantly benefit from in-place processing, as they can create responses directly in the incoming IP buffer. When a CoAP server only sends piggy-backed or non-confirmable responses, no additional buffer is required in the application layer. This, however, requires an elaborated timing so that no incoming data is overwritten before it was processed. Note that an embedded OS usually reuses a single buffer for incoming and outgoing IP packets. So, either care or a buffer to save the incoming data has to be spent in any case.

For clients, this is only an option for non-reliable requests that do not need to be kept for retransmission. Using the IP also for retransmissions would require to forbid any packet reception during

an open request, but could be applied in some cases.

Empty ACKs and RST messages can promptly be assembled and sent using the IP buffer. The first few bytes are usually parsed into the local data structure and can be overwritten without harm.

2.3. Retransmissions

CoAP's reliable transmissions require the before-mentioned retransmission buffers. For clients, obviously the request has to be stored, preferably already serialized. For servers, retransmissions apply for confirmable separate responses and confirmable notifications [I-D.ietf-core-observe]. As separate responses stem from long-lasting resource handlers, the response should be stored for retransmission instead of re-dispatching a stored request (which would allow for updating the representation). For confirmable notifications, please see Section 2.6, as simply storing the response can break the concept of eventual consistency.

String payloads such as JSON require a buffer to print to. By splitting the retransmission buffer into header and payload part, it can be reused. First to generate the payload and then storing the

CoAP message by serializing into the same memory. Thus, providing a retransmission for any message type can save the need for a separate application buffer. This, however, requires an estimation about the maximum expected header size to split the buffer and a memmove to concatenate the two parts.

2.4. Separate Responses

Separate responses are required for long-lasting resource handlers that are too expensive to continuously update in the background to instantly answer from a fresh cache. If possible, those handlers should be realized with split phase execution (e.g., enable a slow sensor, return, and wait for a callback) to not fully block the server during that time. A convenient mechanism to store required data such as the client address and to automatically send the empty ACK could be provided by the implementation. This avoids code duplication when the server has multiple separate resource handlers.

2.5. Deduplication

Deduplication is heavy for Class 1 devices, as the number of peer addresses can be vast. Servers should be kept stateless, i.e., the REST API should be designed idempotent whenever possible. When this is not the case, the resource handler could perform an optimized deduplication by exploiting knowledge about the application. Another, server-wide strategy is to only keep track of non-idempotent

requests.

2.6. Observing

At the server, the list of observers should be stored per resource to only have a handle per observable resource in a superordinate list instead of one resource handle per observer entry. Then for each observer, at least address, port, token, and the last outgoing message ID has to be stored. The latter is needed to match incoming RST messages and cancel the observe relationship.

Besides the list of observers, it is best to have one retransmission buffer per observable resource. Each notification is serialized once into this buffer and only address, port, and token are changed when iterating over the observer list (note that different token lengths

might require realignment). The advantage becomes clear for confirmable notifications: Instead of one retransmission buffer per observer, only one buffer and only individual retransmission counters and timers in the list entry need to be stored. When the notifications can be sent fast enough, even a single timer would suffice. Furthermore, per-resource buffers simplify the update with a new resource state during open deliveries.

2.7. Blockwise Transfers

Blockwise transfers have the main purpose of providing fragmentation at the application layer, where partial information can be processed. This is not possible at lower layers such as 6LoWPAN, as only assembled packets can be passed up the stack. While [I-D.ietf-core-block] also anticipates atomic handling of blocks, i.e., only fully received CoAP messages, this is not possible on Class 1 devices.

When receiving a blockwise transfer, each blocks is usually passed to a handler function that for instance performs stream processing or writes the blocks to external memory such as flash. Although there are no restrictions in [I-D.ietf-core-block], it is beneficial for Class 1 devices to only allow ordered transmission of blocks. Otherwise on-the-fly processing would not be possible.

When sending a blockwise transfer, Class 1 devices usually do not have sufficient memory to print the full message into a buffer, and slice and send it in a second step. When transferring the CoRE Link Format from /.well-known/core for instance, a generator function is required that generates slices of a large string with a specific offset length (a 'sonprintf()'). This functionality is required recurrently and should be included in a library.

2.8. Developer API

Bringing a Web transfer protocol to constrained environments does not only change the networking of the corresponding systems, but also the way they should be programmed. A CoAP implementation should provide a developer API similar to REST frameworks in traditional computing. A server should not be created around an event loop with several function calls, but rather by implementing handlers following the

resource abstraction.

So far, the following types of RESTful resources were identified:

NORMAL A normal resource defined by a static Uri-Path that is associated with a resource handler function. Allowed methods could already be filtered by the implementation based on flags. This is the basis for all other resource types.

PARENT A parent resource manages several sub-resources by programmatically evaluating the Uri-Path, which may be longer than that of the parent resource. Defining a URI templates (see [RFC6570]) would be a convenient way to pre-parse arguments given in the Uri-Path.

PERIODIC A resource that has an additional handler function that is triggered periodically by the CoAP implementation with a resource-defined interval. It can be used to sample a sensor or perform similar periodic updates. Usually, a periodic resource is observable and sends the notifications in the periodic handler function. These periodic tasks are quite common for sensor nodes, thus it makes sense to provide this functionality in the CoAP implementation and avoid redundant code in every resource.

EVENT An event resource is similar to an periodic resource, only that the second handler is called by an irregular event such as a button.

3. Low-power Wireless

The Internet of wireless things needs power-efficient protocols, but existing protocols have typically been designed without explicit power-efficiency. CoAP is optimized to run over low-power link layers such IEEE 802.15.4, but in low-power wireless systems, ultimate power-efficiency translates into the ability to keep the radio off as much as possible, as the radio transceiver is typically the most power-consuming component. This can be achieved in two ways: So called radio duty cycling (RDC) aims to keep the radio off as much as possible, but performs periodic channel checks to realize

a virtual always-on link. Sleepy nodes instead put the radio into

hibernation for a long period during which the node is fully disconnected from the network.

3.1. Radio Duty Cycling

RDC can be achieved through a separate, independent layer between PHY and MAC as depicted in Figure 1. The upper layers can remain more or less untouched and only experience a higher latency, which might require tweaking the timeout parameters. State-of-the-art RDC layers can achieve an idle duty cycling way below 1% while checking the channel several times per second. ContikiMAC for instance achieves a 0.3% cycle with a channel check rate of 4 Hz, which results in a worst-case delay of 250ms per hop. While saving energy, ContikiMAC also makes link-layer transmissions more robust due to its retransmission policy. Please refer to [CONMAC] for details.

In general, RDC can be divided into two approaches: sender initiated (e.g., ContikiMAC) and receiver initiated (e.g., A-MAC [AMAC]). In the first approach, the sender enables the radio first and continuously transmits its message in a strobe until a link-layer ACK is received (note that for IEEE 802.15.4 transceivers, transmitting consumes less energy than receiving). Receivers turn on their radio only periodically to check for these announcements. If they sense a busy channel, the radio is kept on to receive a potential message and finally acknowledge it. In the other approach, the receiver periodically announces that it will keep the radio on for receiving for a while. The senders turns on its radio and listens for an announcement of the recipient. When that is received, it transmits the message (following the scheme of the above MAC layer of course, while back-offs must match the awake time after announcements). Which approach is optimal mainly depends on the communication pattern of the application. Sender initiated RDCs are more efficient for IEEE 802.15.4, but the strobes can congest a busy channel.

3.2. Sleepy Nodes

Going to sleep for a longer time is not transparent for the application layer, as nodes need to re-synchronize and maybe re-associate with the network. Several drafts in the IETF CoRE working group cover this strategy for low-power wireless networking (cf. [I-D.vial-core-mirror-proxy], [I-D.fossati-core-publish-option], [I-D.fossati-core-monitor-option], and [I-D.rahman-core-sleepy]). Such features will have to be integrated into the nodes CoAP implementation as well as the back-end systems. In addition, alternatives to standard diagnosis tools such as ICMP ping will have to be provided, e.g., heartbeats by the application.

This strategy is particular useful for communications other than IEEE 802.15.4. Low-power Wi-Fi for instance is mainly based on long sleeping periods with short wake-up cycles. Although the data rate would be high enough for HTTP over TCP, low-power Wi-FI can greatly benefit from CoAP and its shorter round trip times. For further information about sleepy nodes based on low-power Wi-Fi see [LPWIFI].

4. Security Considerations

T.B.D.

5. Informative References

[AMAC] Dutta, P., Dawson-Haggerty, S., Y., A., Liang, C., and A. Terzis, "Designand Evaluation of a Versatile and Efficient Receiver-Initiated Link Layer for Low-Power Wireless", In Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys 2010). Zurich, Switzerland, November 2010.

[CONMAC] Dunkels, A., "The ContikiMAC Radio Duty Cycling Protocol", SICS Technical Report T2011:13, ISSN 1100-3154, December 2011.

[ERBIUM] Kovatsch, M., Duquennoy, S., and A. Dunkels, "A Low-Power CoAP for Contiki", In Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011). Valencia, Spain, October 2011.

[I-D.fossati-core-monitor-option] Fossati, T., Giacomin, P., and S. Loreto, "Monitor Option for CoAP", draft-fossati-core-monitor-option-00 (work in progress), July 2012.

[I-D.fossati-core-publish-option] Fossati, T., Giacomin, P., and S. Loreto, "Publish Option for CoAP", draft-fossati-core-publish-option-00 (work in progress), July 2012.

[I-D.ietf-core-block] Bormann, C. and Z. Shelby, "Blockwise transfers in CoAP", draft-ietf-core-block-09 (work in progress), August 2012.

[I-D.ietf-core-coap]

Shelby, Z., Hartke, K., Bormann, C., and B. Frank,
"Constrained Application Protocol (CoAP)",

draft-ietf-core-coap-12 (work in progress), October 2012.

[I-D.ietf-core-observe]
Hartke, K., "Observing Resources in CoAP",
draft-ietf-core-observe-06 (work in progress),
September 2012.

[I-D.ietf-lwig-guidance]
Bormann, C., "Guidance for Light-Weight Implementations of the Internet Protocol Suite", draft-ietf-lwig-guidance-02 (work in progress), August 2012.

[I-D.rahman-core-sleepy]
Rahman, A., "Enhanced Sleepy Node Support for CoAP",
draft-rahman-core-sleepy-00 (work in progress), July 2012.

[I-D.vial-core-mirror-proxy]
Vial, M., "CoRE Mirror Server",
draft-vial-core-mirror-proxy-01 (work in progress),
July 2012.

[LPWIFI] Ostermaier, B., Kovatsch, M., and S. Santini, "Connecting Things to the Web using Programmable Low-power WiFi Modules", In Proceedings of the 2nd International Workshop on the Web of Things (WoT 2011). San Francisco, CA, USA, June 2011.

[RFC6550] Winter, T., Thubert, P., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, JP., and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", RFC 6550, March 2012.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, March 2012.

Author's Address

Matthias Kovatsch

ETH Zurich
Universitaetstrasse 6
Zurich, CH-8092
Switzerland

Phone: +41 44 632 06 87
Email: kovatsch@inf.ethz.ch