LWIG Working Group   M. Kovatsch
Internet-Draft   ETH Zurich
Intended status: Informational   O. Bergmann
Expires: January 5, 2015   Universitaet Bremen TZI
                                                               E. Dijk
                                               Philips Research
                                                                 X. He
                                            Hitachi (China) R&D Corp.
                                                     C. Bormann, Ed.
                                               Universitaet Bremen TZI
                                                         July 04, 2014

                        CoAP Implementation Guidance
                          draft-ietf-lwig-coap-01

Abstract

   The Constrained Application Protocol (CoAP) is designed for resource-
   constrained nodes and networks, e.g., sensor nodes in a low-power
   lossy network (LLN). Yet to implement this Internet protocol on
   Class 1 devices (as per RFC 7228, ~ 10 KiB of RAM and ~ 100 KiB of
   ROM) also lightweight implementation techniques are necessary. This
   document provides lessons learned from implementing CoAP for tiny,
   battery-operated networked embedded systems. In particular, it
   provides guidance on correct implementation of the CoAP specification
   RFC 7252, memory optimizations, and customized protocol parameters.

Copyright Notice

Table of Contents

[1](#). Introduction

   The Constrained Application Protocol [RFC7252] has been designed
   specifically for machine-to-machine communication in networks with
   very constrained nodes. Typical application scenarios therefore
   include building automation, process optimization, and the Internet
   of Things. The major design objectives have been set on small
   protocol overhead, robustness against packet loss, and against high
   latency induced by small bandwidth shares or slow request processing
   in end nodes. To leverage integration of constrained nodes with the
   world-wide Internet, the protocol design was led by the REST
   architectural style that accounts for the scalability and robustness
   of the Hypertext Transfer Protocol [RFC7230].

   Lightweight implementations benefit from this design in many
   respects: First, the use of Uniform Resource Identifiers (URIs) for
   naming resources and the transparent forwarding of their
   representations in a server-stateless request/response protocol make
   protocol translation to HTTP a straightforward task. Second, the set
   of protocol elements that are unavoidable for the core protocol and
   thus must be implemented on every node has been kept very small,
   minimizing the unnecessary accumulation of "optional" features.
   Options that - when present - are critical for message processing are
   explicitly marked as such to force immediate rejection of messages
   with unknown critical options. Third, the syntax of protocol data
   units is easy to parse and is carefully defined to avoid creation of
   state in servers where possible.

   Although these features enable lightweight implementations of the
   Constrained Application Protocol, there is still a tradeoff between
   robustness and latency of constrained nodes on one hand and resource
   demands on the other. For constrained nodes of Class 1 or even
   Class 2 [RFC7228], the most limiting factors usually are dynamic
   memory needs, static code size, and energy. Most implementations
   therefore need to optimize internal buffer usage, omit idle protocol
   feature, and maximize sleeping cycles.

   The present document gives possible strategies to solve this tradeoff
   for very constrained nodes (i.e., Class 1). For this, it provides
   guidance on correct implementation of the CoAP specification
   [RFC7252], memory optimizations, and customized protocol parameters.

2. **Protocol Implementation**

   In the programming styles supported by very simple operating systems
   as found on constrained nodes, preemptive multi-threading is not an
   option. Instead, all operations are triggered by an event loop
   system, e.g., in a send-receive-dispatch cycle. It is also 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 for memory usage are the
   number of (re)transmission buffers and the maximum message size that
   must be supported by each buffer. Often the maximum message size is
   set far below the 1280-byte MTU of 6LoWPAN to allow more than one
   open Confirmable transmission at a time (in particular for parallel
   observe notifications [I-D.ietf-core-observe]). 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 tradeoff between 6LoWPAN
   fragmentation and CoAP header overhead must be found. Usually the
   amount of available free RAM dominates this decision. For Class 1
   devices, the maximum message size is typically 128 or 256 bytes
   (blockwise) payload plus an estimate of the maximum header size with
   a worst case option setting.

2.1. **Client/Server Model**

   In general, CoAP servers can be implemented more efficiently than
   clients. REST allows them to keep the communication stateless and
   piggy-backed responses are not stored for retransmission, saving
   buffer space. The use of idempotent requests also allows to relax
   deduplication, which further decreases memory usage. It is also easy
   to estimate the required maximum size of message buffers, since URI
   paths, supported options, and maximum payload sizes of the
   application are known at compile time. Hence, when the application
   is distributed over constrained and unconstrained nodes, the
   constrained ones should preferably have the server role.

   HTTP-based applications have established an inverse model because of
   the need for simple push notifications: A constrained client uses
   POST requests to update resources on an unconstrained server whenever
   an event, e.g., a new sensor reading, is triggered. This requirement
   is solved by the Observe option [I-D.ietf-core-observe] of CoAP. It
   allows servers to initiate communication and send push notifications
   to interested client nodes. This allows a more efficient and also
   more natural model for CoAP-based applications, where the information
   source is in server role and can benefit from caching.

2.2. Message Processing

Apart from the required buffers, message processing is symmetric for
clients and servers. First the 4-byte base header has to be parsed
and thereby checked if it is a CoAP message. Since the encoding is
very dense, only a wrong Version or a datagram size smaller than four
bytes identify non-CoAP datagrams. These need to be silently
ignored. Other message format errors, such as an incomplete datagram
length or the usage of reserved values, may need to be rejected with
a Reset (RST) message (see Section 4.2 and 4.3 of [RFC7252] for
details). Next the Token is read based on the TKL field. For the
following header options, there are two alternatives: Either process
the header on the fly when an option is accessed or initially parse
all values into an internal data structure.

2.2.1. On-the-fly Processing

The advantage of on-the-fly processing is that no additional memory
needs to be allocated to store the option values, which are stored
efficiently inline in the buffer for incoming messages. Once the
message is accepted for further processing, the set of options
contained in the received message must be decoded to check for
unknown critical options. To avoid multiple passes through the
option list, the option parser might maintain a bit-vector where each
bit represents an option number that is present in the received
request. With the wide and sparse range of option numbers, the
number itself cannot be used to indicate the number of left-shift
operations to mask the corresponding bit. Hence, an implementation-
specific enum of supported options should be used to mask the present
options of a message in the bitmap. In addition, the byte index of
every option (a direct pointer) can be added to a sparse list (e.g.,
a one-dimensional array) for fast retrieval.

This particularly enables efficient handling of options that might
occur more than once such as Uri-Path. In this implementation
strategy, the delta is zero for any subsequent path segment, hence
the stored byte index for this option (e.g., 11 for Uri-Path) would
be overwritten to hold a pointer to only the last occurrence of that
option. The Uri-Path can be resolved on the fly, though, and a
pointer to the targeted resource stored directly in the sparse list.
In simpler cases, conditionals can preselect one of the repeated
option values.

Once the option list has been processed, all known critical option
and all elective options can be masked out in the bit-vector to
determine if any unknown critical option was present. If this is the
case, this information can be used to create a 4.02 response
accordingly. Note that full processing must only be done up to the

highest supported option number. Beyond that, only the least
significant bit (Critical or Elective) needs to be checked.
Otherwise, if all critical options are supported, the sparse list of
option pointers is used for further handling of the message.

2.2.2. Internal Data Structure

Using an internal data structure for all parsed options has an
advantage when working on the option values, as they are already in a
variable of corresponding type, e.g., an integer in host byte order.
The incoming payload and byte strings of the header can be accessed
directly in the buffer for incoming messages using pointers (similar
to on-the-fly processing). This approach also benefits from a
bitmap. Otherwise special values must be reserved to encode an unset
option, which might require a larger type than required for the
actual value range (e.g., a 32-bit integer instead of 16-bit).

The byte strings (e.g., the URI) are usually not required when
generating the response. And since all important values were copied,
this alternative facilitates using the buffer for incoming messages
also for the assembly of outgoing messages - which can be the shared
IP buffer provided by the OS.

Setting options for outgoing messages is also easier with an internal
data structure. Application developers can set options independent
from the option number, whose order is required for the delta
encoding. The CoAP encoding is then applied in a serialization step
before sending. In contrast, assembling outgoing messages with on-
the-fly processing requires either extensive memmove operations to
insert new header options or restrictions for developers to set
options in their correct order.

2.3. Duplicate Rejection

If CoAP is used directly on top of UDP (i.e., in NoSec mode), it
needs to cope with the fact that the UDP datagram transport can
reorder and duplicate messages. (In contrast to UDP, DTLS has its
own duplicate detection.) CoAP has been designed with protocol
functionality such that rejection of duplicate messages is always
possible. It is at the discretion of the receiver if it actually
wants to make use of this functionality. Processing of duplicate
messages comes at a cost, but so does the management of the state
associated with duplicate rejection. The number of remote endpoints
that need to be managed might be vast. This can be costly in
particular for unconstrained nodes that have throughput in the order
of one hundred thousand requests per second (which might need about
16 GiB of RAM just for duplicate rejection). Deduplication is also
heavy for servers on Class 1 devices, as also piggy-backed responses

need to be stored for the case that the ACK message is lost. Hence,
a receiver may have good reasons to decide not to do the
deduplication.

If duplicate rejection is indeed necessary, e.g., for non-idempotent
requests, it is important to control the amount of state that needs
to be stored. It can be reduced for instance by deduplication at
resource level: Knowledge of the application and supported
representations can minimize the amount of state that needs to be
kept. Duplicate rejection on the client side can be simplified by
choosing clever Tokens and only filter based on this information
(e.g., a list of Tokens currently in use or an obscured counter in
the Token value).

2.4. Token Usage

Tokens are chosen by the client and help to identify request/response
pairs that span several message exchanges (e.g., a separate response,
which has a new MID). Servers do not generate Tokens and only mirror
what they receive from the clients. Tokens must be unique within the
namespace of a client throughout their lifetime. This begins when
being assigned to a request and ends when the open request is closed
by receiving and matching the final response. Neither empty ACKs nor
notifications (i.e., responses carrying the Observe option) terminate
the lifetime of a Token.

As already mentioned, a clever assignment of Tokens can help to
simplify duplicate rejection. Yet this is also important for coping
with client crashes. When a client restarts during an open request
and (unknowingly) re-uses the same Token, it might match the response
from the previous request to the current one. Hence, when only the
Token is used for matching, which is always the case for separate
responses, randomized Tokens with enough entropy should be used. The
8-byte range for Tokens even allows for one-time usage throughout the
lifetime of a client node. When DTLS is used, client crashes/
restarts will lead to a new security handshake, thereby solving the
problem of mismatching responses and/or notifications.

2.4.1. Tokens for Observe

In the case of Observe [I-D.ietf-core-observe], a request will be
answered with multiple notifications and it can become hard to
determine the end of a Token lifetime. When establishing an Observe
relationship, the Token is registered at the server. Hence, the
client partially loses control of the used Token. A client can
attempt to cancel the relationship, which frees the Token upon
success (i.e., the message with an Observe Option with the value set
to 'deregister' (1) is acknowledged; see [I-D.ietf-core-observe]

section 3.6). However, the client might never receive the ACK due to
a temporary network outages or worse, a server crash. Although a
network outage will also affect notifications so that the Observe
garbage collection could apply, the server might simply not send CON
notifications during that time. Alternative Observe lifetime models
such as Stubbornness(tm) might also keep relationships alive for
longer periods.

Thus, Observe requests should carefully chose the value (and the
empty value will rarely be applicable). One option is to assign and
re-use dedicated Tokens for each Observe relationship the client will
establish. This is, however, critical for spoofing attacks in NoSec
mode. The recommendation is to use randomized Tokens with a length
of at least four bytes (see Section 5.3.1 of [RFC7252]). Thus,
dedicated ranges within the 8-byte Token space should be used when in
NoSec mode. This also solves the problem of mismatching
notifications after a client crash/restart.

2.4.2. Tokens for Blockwise Transfers

In general, blockwise transfers are independent from the Token and
are correlated through client endpoint address and server address and
resource path (destination URI). Thus, each block may be transferred
using a different Token. Still it can be beneficial to use the same
Token (it is freed upon reception of a response block) for all
blocks, e.g., to easily route received blocks to the same response
handler.

When Block2 is combined with Observe, notifications only carry the
first block and it is up to the client to retrieve the remaining
ones. These GET requests do not carry the Observe option and need to
use a different Token, since the Token from the notification is still
in use.

2.5. Transmission States

CoAP endpoints must keep transmission state to manage open requests,
to handle the different response modes, and to implement reliable
delivery at the message layer. The following finite state machines
(FSMs) model the transmissions of a CoAP exchange at the request/
response layer and the message layer. These layers are linked
through actions. The M_CMD() action triggers a corresponding
transition at the message layer and the RR_EVT() action triggers a
transition at the request/response layer. The FSMs also use guard
conditions to distinguish between information that is only available
through the other layer (e.g., whether a request was sent using a CON
or NON message).

2.5.1. **Request/Response Layer**

   Figure 1 depicts the two states at the request/response layer of a
   CoAP client. When a request is issued, a "reliable_send" or
   "unreliable_send" is triggered at the message layer. The WAITING
   state can be left through three transitions: Either the client
   cancels the request and triggers cancellation of a CON transmission
   at the message layer, the client receives a failure event from the
   message layer, or a receive event containing a response.

```
     +------------CANCEL-------------------------------+
     |          / M_CMD(cancel)                        |
     |                                                 V
     |                                          +------+
 +-------+ -------RR_EVT(fail)------------------> |      |
 |WAITING|                                      | IDLE |
 +-------+ -------RR_EVT(rx)[is Response]--------> |      |
     ^                / M_CMD(accept)           +------+
     |                                             |
     +--------------------REQUEST---------------------+
                 / M_CMD((un)reliable_send)
```
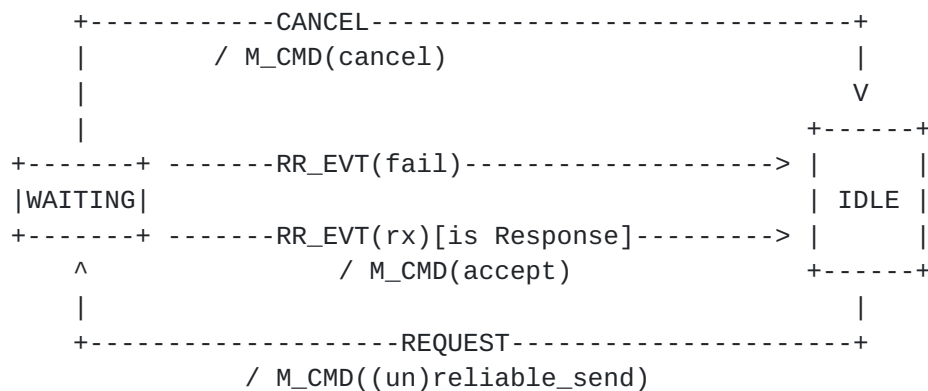
                Figure 1: CoAP Client Request/Response Layer FSM

   A server resource can decide at the request/response layer whether to
   respond with a piggy-backed or a separate response. Thus, there are
   two busy states in Figure 2, SERVING and SEPARATE. An incoming
   receive event with a NON request directly triggers the transition to
   the SEPARATE state.

```
 +--------+ <----------RR_EVT(rx)[is NON]---------- +------+
 |SEPARATE|                                         |      |
 +--------+ ----------------RESPONSE--------------> | IDLE |
     ^              / M_CMD((un)reliable_send)       |      |
     |                                          +---> +------+
     |EMPTY_ACK                                 |        |
     |/M_CMD(accept)                            |        |
     |                                          |        |
     |                                          |        |
 +--------+                                     |        |
 |SERVING | --------------RESPONSE------------+        |
 +--------+          / M_CMD(accept)                   |
     ^                                                 |
     +-----------------------RR_EVT(rx)[is CON]--------+
```

                Figure 2: CoAP Server Request/Response Layer FSM

2.5.2. Message Layer

   Figure 3 shows the different states of a CoAP endpoint per message
   exchange. Besides the linking action RR_EVT(), the message layer has
   a TX action to send a message. For sending and receiving NONs, the
   endpoint remains in its CLOSED state. When sending a CON, the
   endpoint remains in RELIABLE_TX and keeps retransmitting until the
   transmission times out, it receives a matching RST, the request/
   response layer cancels the transmission, or the endpoint receives an
   implicit acknowledgement through a matching NON or CON. Whenever the
   endpoint receives a CON, it transitions into the ACK_PENDING state,
   which can be left by sending the corresponding ACK.

```
+-----------+ <-------M_CMD(reliable_send)-----+
|           |              / TX(con)            \
|           |                           +--------------+
|           | ---TIMEOUT(RETX_WINDOW)------> |              |
|RELIABLE_TX|     / RR_EVT(fail)             |              |
|           | --------------------RX_RST--> |              | <----+
|           |              / RR_EVT(fail)   |              |      |
+-----------+ ----M_CMD(cancel)------------> |    CLOSED    |      |
  ^  |  |  \  \                              |              | --+  |
  |  |  |   \  +------------------RX_ACK---> |              |  |  |
 +*1+ |   \           / RR_EVT(rx)          |              |  |  |
      |    +----RX_NON------------------> +--------------+  |  |
      |       / RR_EVT(rx)                 ^ ^ ^ ^  | | | |  |  |
      |                                    | | | |  | | | |  |  |
      |                                    | | | +*2+ | | |  |  |
      |                                    | | +--*3--+ | |  |  |
      |                                    | +----*4----+ |  |  |
      |                                    +------*5------+  |  |
      |              +---------------+                       |  |
      |              | ACK_PENDING   | <--RX_CON-------------+  |
     +----RX_CON----> |              |   / RR_EVT(rx)           |
       / RR_EVT(rx)   +---------------+ --------M_CMD(accept)---+
                                                   / TX(ack)
```
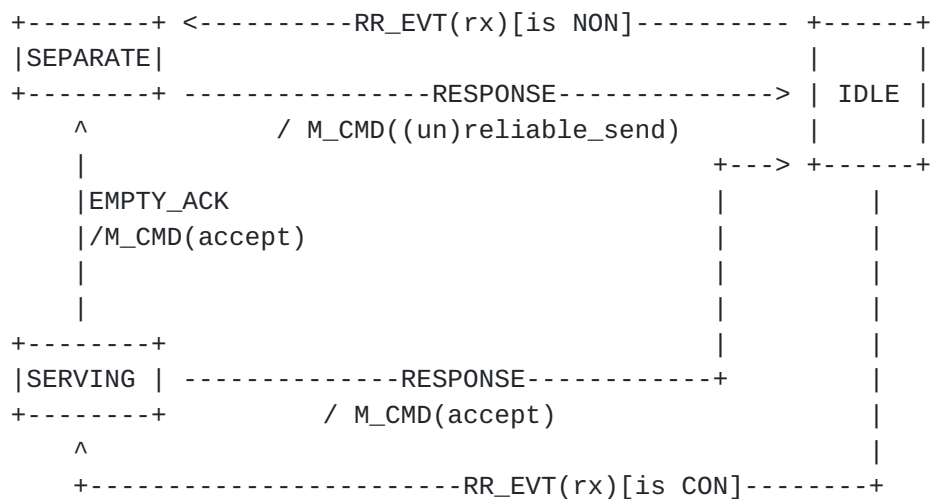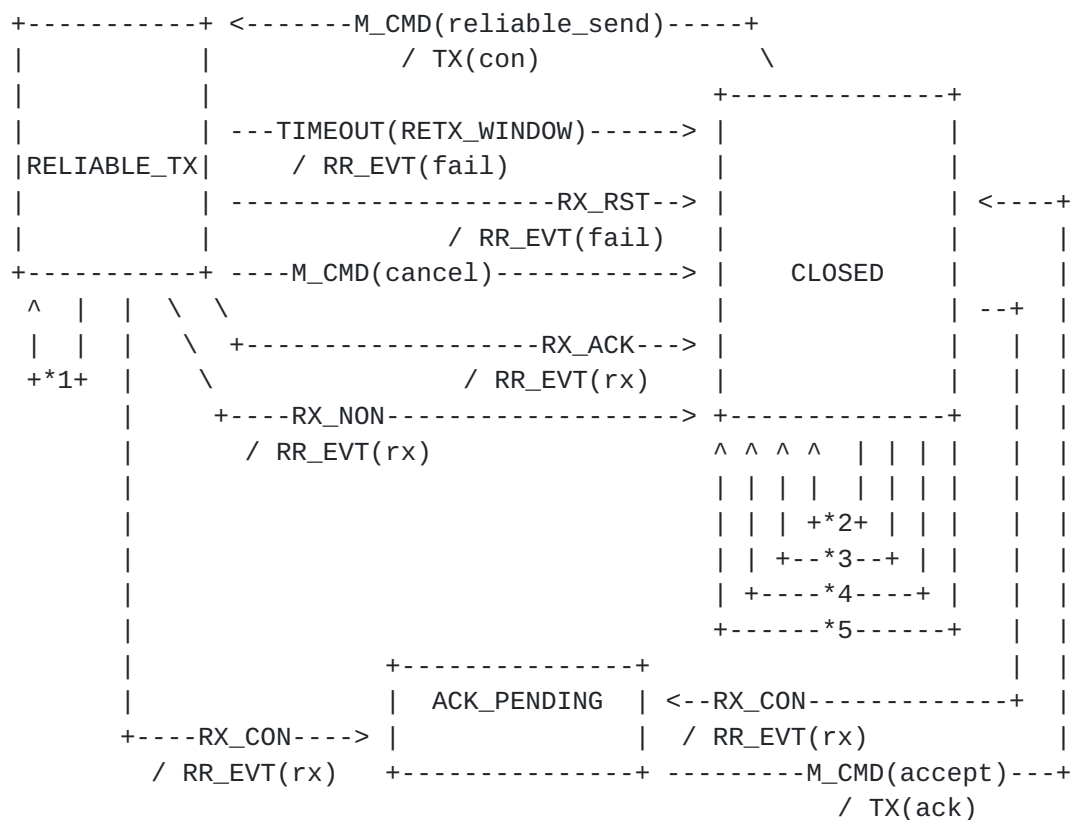
   *1: TIMEOUT(RETX_TIMEOUT) / TX(con)
   *2: M_CMD(unreliable_send) / TX(non)
   *3: RX_NON / RR_EVT(rx)
   *4: RX_RST / REMOVE_OBSERVER
   *5: RX_ACK

                     Figure 3: CoAP Message Layer FSM

   T.B.D.: (i) Rejecting messages (can be triggered at message and
   request/response layer). (ii) ACKs can also be triggered at both
   layers.

2.6. Out-of-band Information

   The CoAP implementation can also leverage out-of-band information,
   that might also trigger some of the transitions shown in Section 2.5.
   In particular ICMP messages can inform about unreachable remote
   endpoints or whole network outages. This information can be used to
   pause or cancel ongoing transmission to conserve energy. Providing
   ICMP information to the CoAP implementation is easier in constrained
   environments, where developers usually can adapt the underlying OS
   (or firmware). This is not the case on general purpose platforms
   that have full-fledged OSes and make use of high-level programming
   frameworks.

   The most important ICMP messages are host, network, port, or protocol
   unreachable errors. After appropriate vetting (cf. [RFC5927]), they
   should cause the cancellation of ongoing CON transmissions and
   clearing (or deferral) of Observe relationships. Requests to this
   destination should be paused for a sensible interval. In addition,
   the device could indicate of this error through a notification to a
   management endpoint or external status indicator, since the cause
   could be a misconfiguration or general unavailability of the required
   service. Problems reported through the Parameter Problem message are
   usually caused through a similar fundamental problem.

   The CoAP specification recommends to ignore Source Quench and Time
   Exceeded ICMP messages, though. Source Quench messages were
   originally intended to inform the sender to reduce the rate of
   packets. However, this mechanism is deprecated through [RFC6633].
   CoAP also comes with its own congestion control mechanism, which is
   already designed conservatively. One advanced mechanism that can be
   employed for better network utilization is CoCoA,
   [I-D.bormann-core-cocoa]. Time Exceeded messages often occur during
   transient routing loops (unless they are caused by a too small
   initial Hop Limit value).

2.7. Programming Model

   The event-driven approach, which is common in event-loop-based
   firmware, has also proven very efficient for embedded operating
   systems [TinyOS], [Contiki]. Note that an OS is not necessarily
   required and a traditional firmware approach can suffice for Class 1
   devices. Event-driven systems use split-phase operations (i.e.,
   there are no blocking functions, but functions return and an event
   handler is called once a long-lasting operation completes) to enable
   cooperative multi-threading with a single stack.

   Bringing a Web transfer protocol to constrained environments does not
   only change the networking of the corresponding systems, but also the

programming model. The complexity of event-driven systems can be
hidden through APIs that resemble classic RESTful Web service
implementations.

2.7.1. Client

An API for asynchronous requests with response handler functions goes
hand-in-hand with the event-driven approach. Synchronous requests
with a blocking send function can facilitate applications that
require strictly ordered, sequential request execution (e.g., to
control a physical process) or other checkpointing (e.g., starting
operation only after registration with the resource directory was
successful). However, this can also be solved by triggering the next
operation in the response handlers. Furthermore, as mentioned in
Section 2.1, it is more like that complex control flow is done by
more powerful devices and Class 1 devices predominantly run a CoAP
server (which might include a minimal client to communicate with a
resource directory).

2.7.2. Server

On CoAP servers, the event-driven nature can be hidden through
resource handler abstractions as known from traditional REST
frameworks. The following types of RESTful resources have proven
useful to provide an intuitive API on constrained event-driven
systems:

NORMAL A normal resource defined by a static Uri-Path and an
   associated 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 under a given
   base path by programmatically evaluating the Uri-Path. Defining a
   URI template (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-
   specific interval. It can be used to sample a sensor or perform
   similar periodic updates of its state. Usually, a periodic
   resource is observable and sends the notifications by triggering
   its normal resource handler from the periodic handler. 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.

   SEPARATE Separate responses are usually used when handling a request
      takes more time, e.g., due to a slow sensor or UART-based
      subsystems. To not fully block the system during this time, the
      handler should also employ split-phase execution: The resource
      handler returns as soon as possible and an event handler resumes
      responding when the result is ready. The separate resource type
      can abstract from the split-phase operation and take care of
      temporarily storing the request information that is required later
      in the result handler to send the response (e.g., source address
      and Token).

3. Optimizations

3.1. Message Buffers

   The cooperative multi-threading of an event loop system allows to
   optimize memory usage through in-place processing and reuse of
   buffers, in particular the IP buffer provided by the OS or firmware.

   CoAP servers can significantly benefit from in-place processing, as
   they can create responses directly in the incoming IP buffer. Note
   that an embedded OS usually only has a single buffer for incoming and
   outgoing IP packets. The first few bytes of the basic header are
   usually parsed into an internal data structure and can be overwritten
   without harm. Thus, empty ACKs and RST messages can promptly be
   assembled and sent using the IP buffer. Also when a CoAP server only
   sends piggy-backed or Non-confirmable responses, no additional buffer
   is required at the application layer. This, however, requires
   careful timing so that no incoming data is overwritten before it was
   processed. Because of cooperative multi-threading, this requirement
   is relaxed, though. Once the message is sent, the IP buffer can
   accept new messages again. This does not work for Confirmable
   messages, however. They need to be stored for retransmission and
   would block any further IP communication.

   Depending on the number of requests that can be handled in parallel,
   an implementation might create a stub response filled with any option
   that has to be copied from the original request to the separate
   response, especially the Token option. The drawback of this
   technique is that the server must be prepared to receive
   retransmissions of the previous (Confirmable) request to which a new
   acknowledgement must be generated. If memory is an issue, a single
   buffer can be used for both tasks: Only the message type and code
   must be updated, changing the message id is optional. Once the

resource representation is known, it is added as new payload at the end of the stub response. Acknowledgements still can be sent as described before as long as no additional options are required to describe the payload.

3.2. Retransmissions

CoAP's reliable transmissions require the before-mentioned retransmission buffers. Messages, such as the requests of a client, should be stored in serialized form. 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.

For platforms that disable clock tick interrupts in sleep states, the application must take into consideration the clock deviation that occurs during sleep (or ensure to remain in idle state until the message has been acknowledged or the maximum number of retransmissions is reached). Since CoAP allows up to four retransmissions with a binary exponential back-off it could take up to 45 seconds until the send operation is complete. Even in idle state, this means substantial energy consumption for low-power nodes. Implementers therefore might choose a two-step strategy: First, do one or two retransmissions and then, in the later phases of back-off, go to sleep until the next retransmission is due. In the meantime, the node could check for new messages including the acknowledgement for any Confirmable message to send.

3.3. Observable Resources

For each observer, the server needs to store at least address, port, token, and the last outgoing message ID. The latter is needed to match incoming RST messages and cancel the observe relationship.

It is favorable to have one retransmission buffer per observable
resource that is shared among all observers. 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.

3.4. 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 block 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 out of dynamically generated
information, 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. For instance, if the CoRE Link Format at /.well-known/
core is dynamically generated, 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.

3.5. Deduplication with Sequential MIDs

CoAP's duplicate rejection functionality can be straightforwardly
implemented in a CoAP endpoint by storing, for each remote CoAP
endpoint ("peer") that it communicates with, a list of recently
received CoAP Message IDs (MIDs) along with some timing information.
A CoAP message from a peer with a MID that is in the list for that
peer can simply be discarded.

The timing information in the list can then be used to time out
entries that are older than the _expected extent of the re-ordering_,
an upper bound for which can be estimated by adding the _potential
retransmission window_ ([RFC7252] section "Reliable Messages") and
the time packets can stay alive in the network.

Such a straightforward implementation is suitable in case other CoAP
endpoints generate random MIDs. However, this storage method may
consume substantial RAM in specific cases, such as:

o many clients are making periodic, non-idempotent requests to a
   single CoAP server;

o one client makes periodic requests to a large number of CoAP
   servers and/or requests a large number of resources; where servers
   happen to mostly generate separate CoAP responses (not piggy-
   backed);

For example, consider the first case where the expected extent of re-
ordering is 50 seconds, and N clients are sending periodic POST
requests to a single CoAP server during a period of high system
activity, each on average sending one client request per second. The
server would need 100 * N bytes of RAM to store the MIDs only. This
amount of RAM may be significant on a RAM-constrained platform. On a
number of platforms, it may be easier to allocate some extra program
memory (e.g. Flash or ROM) to the CoAP protocol handler process than
to allocate extra RAM. Therefore, one may try to reduce RAM usage of
a CoAP implementation at the cost of some additional program memory
usage and implementation complexity.

Some CoAP clients generate MID values by a using a Message ID
variable [RFC7252] that is incremented by one each time a new MID
needs to be generated. (After the maximum value 65535 it wraps back
to 0.) We call this behavior "sequential" MIDs. One approach to
reduce RAM use exploits the redundancy in sequential MIDs for a more
efficient MID storage in CoAP servers.

Naturally such an approach requires, in order to actually reduce RAM
usage in an implementation, that a large part of the peers follow the
sequential MID behavior. To realize this optimization, the authors
therefore RECOMMEND that CoAP endpoint implementers employ the
"sequential MID" scheme if there are no reasons to prefer another
scheme, such as randomly generated MID values.

Security considerations might call for a choice for
(pseudo)randomized MIDs. Note however that with truly randomly
generated MIDs the probability of MID collision is rather high in use
cases as mentioned before, following from the Birthday Paradox. For

example, in a sequence of 52 randomly drawn 16-bit values the
probability of finding at least two identical values is about 2
percent.

From here on we consider efficient storage implementations for MIDs
in CoAP endpoints, that are optimized to store "sequential" MIDs.
Because CoAP messages may be lost or arrive out-of-order, a solution
has to take into account that received MIDs of CoAP messages are not
actually arriving in a sequential fashion, due to lost or reordered
messages. Also a peer might reset and lose its MID counter(s) state.
In addition, a peer may have a single Message ID variable used in
messages to many CoAP endpoints it communicates with, which partly
breaks sequentiality from the receiving CoAP endpoint's perspective.
Finally, some peers might use a randomly generated MID values
approach. Due to these specific conditions, existing sliding window
bitfield implementations for storing received sequence numbers are
typically not directly suitable for efficiently storing MIDs.

Table 1 shows one example for a per-peer MID storage design: a table
with a bitfield of a defined length _K_ per entry to store received
MIDs (one per bit) that have a value in the range [MID_i + 1 , MID_i
+ K].

```
          +----------+----------------+-----------------+
          | MID base | K-bit bitfield | base time value |
          +----------+----------------+-----------------+
          | MID_0    | 010010101001   | t_0             |
          |          |                |                 |
          | MID_1    | 111101110111   | t_1             |
          |          |                |                 |
          | ... etc. |                |                 |
          +----------+----------------+-----------------+
```

          Table 1: A per-peer table for storing MIDs based on MID_i

The presence of a table row with base MID_i (regardless of the
bitfield values) indicates that a value MID_i has been received at a
time t_i. Subsequently, each bitfield bit k (0...K-1) in a row i
corresponds to a received MID value of MID_i + k + 1. If a bit k is
0, it means a message with corresponding MID has not yet been
received. A bit 1 indicates such a message has been received already
at approximately time t_i. This storage structure allows e.g. with
k=64 to store in best case up to 130 MID values using 20 bytes, as
opposed to 260 bytes that would be needed for a non-sequential
storage scheme.

The time values t_i are used for removing rows from the table after a
preset timeout period, to keep the MID store small in size and enable

these MIDs to be safely re-used in future communications. (Note that
the table only stores one time value per row, which therefore needs
to be updated on receipt of another MID that is stored as a single
bit in this row. As a consequence of only storing one time value per
row, older MID entries typically time out later than with a simple
per-MID time value storage scheme. The endpoint therefore needs to
ensure that this additional delay before MID entries are removed from
the table is much smaller than the time period after which a peer
starts to re-use MID values due to wrap-around of a peer's MID
variable. One solution is to check that a value t_i in a table row
is still recent enough, before using the row and updating the value
t_i to current time. If not recent enough, e.g. older than N
seconds, a new row with an empty bitfield is created.) [Clearly,
these optimizations would benefit if the peer were much more
conservative about re-using MIDs than currently required in the
protocol specification.]

The optimization described is less efficient for storing randomized
MIDs that a CoAP endpoint may encounter from certain peers. To solve
this, a storage algorithm may start in a simple MID storage mode,
first assuming that the peer produces non-sequential MIDs. While
storing MIDs, a heuristic is then applied based on monitoring some
"hit rate", for example, the number of MIDs received that have a Most
Significant Byte equal to that of the previous MID divided by the
total number of MIDs received. If the hit rate tends towards 1 over
a period of time, the MID store may decide that this particular CoAP
endpoint uses sequential MIDs and in response improve efficiency by
switching its mode to the bitfield based storage.

4. Alternative Configurations

4.1. Transmission Parameters

When a constrained network of CoAP nodes is not communicating over
the Internet, for instance because it is shielded by a proxy or a
closed deployment, alternative transmission parameters can be used.
Consequently, the derived time values provided in [RFC7252] section
4.8.2 will also need to be adjusted, since most implementations will
encode their absolute values.

Static adjustments require a fixed deployment with a constant number
or upper bound for the number of nodes, number of hops, and expected
concurrent transmissions. Furthermore, the stability of the wireless
links should be evaluated. ACK_TIMEOUT should be chosen above the
xx% percentile of the round-trip time distribution.
ACK_RANDOM_FACTOR depends on the number of nodes on the network.
MAX_RETRANSMIT should be chosen suitable for the targeted
application. A lower bound for LEISURE can be calculated as

    lb_Leisure = S * G / R

    where S is the estimated response size, G the group size, and R the
    target data transfer rate (see [RFC7252] section 8.2). NSTART and
    PROBING_RATE depend on estimated network utilization. If the main
    cause for loss are weak links, higher values can be chosen.

    Dynamic adjustments will be performed by advanced congestion control
    mechanisms such as [I-D.bormann-core-cocoa]. They are required if
    the main cause for message loss is network or endpoint congestion.
    Semi-dynamic adjustments could be implemented by disseminating new
    static transmission parameters to all nodes when the network
    configuration changes (e.g., new nodes are added or long-lasting
    interference is detected).

4.2. CoAP over IPv4

    CoAP was designed for the properties of IPv6, which is dominating in
    constrained environments because of the 6LoWPAN adaption layer
    [RFC6282]. In particular, the size limitations of CoAP are tailored
    to the minimal MTU of 1280 bytes. Until the transition towards IPv6
    converges, CoAP nodes might also communicate over IPv4, though.
    Sections 4.2 and 4.6 of the base specification [RFC7252] already
    provide guidance and implementation notes to handle the smaller
    minimal MTUs of IPv4.

5. IANA considerations

    This document has no actions for IANA.

6. Security considerations

    TBD

7. References

7.1. Normative References

    [I-D.bormann-core-cocoa]
               Bormann, C., Betzler, A., Gomez, C., and I. Demirkol,
               "CoAP Simple Congestion Control/Advanced", draft-bormann-
               core-cocoa-02 (work in progress), July 2014.

    [I-D.ietf-core-block]
               Bormann, C. and Z. Shelby, "Blockwise transfers in CoAP",
               draft-ietf-core-block-14 (work in progress), October 2013.

    [I-D.ietf-core-observe]
               Hartke, K., "Observing Resources in CoAP", draft-ietf-
               core-observe-14 (work in progress), June 2014.

    [RFC6282] Hui, J. and P. Thubert, "Compression Format for IPv6
               Datagrams over IEEE 802.15.4-Based Networks", RFC 6282,
               September 2011.

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

    [RFC6633] Gont, F., "Deprecation of ICMP Source Quench Messages",
               RFC 6633, May 2012.

    [RFC7230] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol
               (HTTP/1.1): Message Syntax and Routing", RFC 7230, June
               2014.

    [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained
               Application Protocol (CoAP)", RFC 7252, June 2014.

7.2. Informative References

    [Contiki] Dunkels, A., Groenvall, B., and T. Voigt, "Contiki - a
               Lightweight and Flexible Operating System for Tiny
               Networked Sensors", Proceedings of the First IEEE Workshop
               on Embedded Networked Sensors , November 2004.

    [RFC5927] Gont, F., "ICMP Attacks against TCP", RFC 5927, July 2010.

    [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for
               Constrained-Node Networks", RFC 7228, May 2014.

    [TinyOS]   Levis, P., Madden, S., Polastre, J., Szewczyk, R.,
               Whitehouse, K., Woo, A., Gay, D., Woo, A., Hill, J.,
               Welsh, M., Brewer, E., and D. Culler, "TinyOS: An
               Operating System for Sensor Networks", Ambient
               intelligence, Springer (Berlin Heidelberg), ISBN
               978-3-540-27139-0 , 2005.

Authors' Addresses

Matthias Kovatsch
ETH Zurich
Universitaetstrasse 6
CH-8092 Zurich
Switzerland

Email: kovatsch@inf.ethz.ch

Olaf Bergmann
Universitaet Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

Email: bergmann@tzi.org

Esko Dijk
Philips Research

Email: esko.dijk@philips.com

Xuan He
Hitachi (China) R&D Corp.
301, Tower C North, Raycom, 2 Kexuyuan Nanlu
Beijing, 100190
P.R.China

Email: xhe@hitachi.cn

Carsten Bormann (editor)
Universitaet Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

Phone: +49-421-218-63921
Email: cabo@tzi.org