TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Distributed and Networked Systems

unicoap: A Unified and Modular CoAP Suite in RIOT

Carl Seifert

Bachelor's Thesis

to achieve the academic degree

Bachelor of Science (B.Sc.)

Supervisors

Mikolai Gütschow, Martine Lenders

Supervising professor

Prof. Dr. rer. nat. Matthias Wählisch

Submitted: January 29, 2025

Statement of Authorship

I hereby certify that I have authored this document entitled *unicoap: A Unified and Modular CoAP Suite in RIOT* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, January 29, 2025

Carl Seifert

Contents

List of Figures

List of Tables

Listings

1 Introduction

The Internet of Things (IoT) encompasses a broad spectrum of devices, covering consumer-grade applications to industrial uses. Deployments range from smart home actuators such as connected thermostats and personal devices like item trackers, to environmental sensors collecting weather data.

Most of these small, interconnected devices are inherently limited in terms of power and computational resources [1]. IoT nodes are often battery-powered and must operate with minimal power consumption across prolonged periods of time. IoT deployments typically involve exchanging data wirelessly to enable flexibility, scalability, and easy installation. Low-power IoT networks frequently experience packet loss and commonly have low data rates (e.g., 250 $\frac{\mathrm{kbit}}{s}$ in IEEE 802.15.4 networks [2]) [3]. HTTP as a general-purpose application protocol carries a significant overhead, particularly in terms of header size. HTTP would be problematic for IoT networks with limited bandwidth and nodes with little memory [4, 5].

The Constrained Application Protocol (CoAP) defines a transactional communication model for the IoT and is specifically optimized for embedded, constrained IoT deployments [6]. CoAP also covers a range of features needed in the IoT, such as resource discovery, message fragmentation, and end-to-end message protection [7, 8, 9]. CoAP is versatile as it can operate over various transport protocols, such as UDP, DTLS, TCP, TLS, and WebSockets [1, 10]. There are also ongoing efforts in the IETF to add more transports to CoAP [11].

To reduce manufacturing costs and drive energy efficiency, IoT hardware is commonly kept simple, providing only little memory. Many IoT devices are not equipped to run general-purpose operating systems such as Linux, or BSD derivatives [1]. RIOT is a well-known, free, and open-source operating system for these resource-constrained devices. RIOT is tailored to fit low-memory and real-time requirements [12], and is thus an ideal environment to implement a CoAP library.

In this thesis, we propose `unicoap` as a unified, high-level CoAP suite for the constrained IoT. `unicoap` unifies support for different transports and several CoAP features. Abstracting different CoAP transports is challenging: First, messaging properties, including reliability mechanisms and PDU formats, vary between transports [6, 10]. Second, various protocol features influence or build on top of each other. `unicoap` seeks to overcome these challenges by separating protocol mechanisms and features into layers. This approach is nontrivial due to memory demands in a constrained environment and state management across layers. Our design emphasizes extensibility, ensuring adaptability to evolving requirements such as support for future CoAP transports. Our key contributions consist of a layered design abstracting transport and protocol features, a modular implementation in RIOT for low-end hardware and an evaluation of our implementation.

This thesis is structured as follows. We begin by outlining CoAP and advanced protocol features in Chapter 2, underlining the complexity of a variable transport layer. Then, we discuss existing CoAP libraries in RIOT and highlight limitations in their design. In Chapter 3, we present the design of `unicoap`, including high-level transport abstractions and our approach to advanced CoAP features. We establish our strategy for layer separation and modularization. In Chapter 4, we detail our modular implementation of `unicoap` in RIOT. In Chapter 5, we evaluate the memory and timing properties of our implementation and assess the suitability of the design concepts for the constrained IoT. The thesis concludes with Chapter 6, where we summarize our findings.

2 Background and Related Work

2.1 The Constrained Application Protocol

The Constrained Application Protocol (CoAP) [6] implements a generic REST model for resource-constrained deployments involving low-end hardware with very little computational resources. RAM and ROM sizes of constrained IoT devices commonly range in the order of kilobytes [1].

CoAP facilitates data transfer through an asynchronous exchange of compact request and response messages. CoAP works as follows. A client sends a request with a certain request method to a server that replies by sending a response with a status code. CoAP can be easily mapped to the Hypertext Transfer Protocol (HTTP). Akin to HTTP headers, messages are designed to accommodate options such as the media type. The particular service a request is directed at, called *resource*, is identified by a Uniform Resource Identifier (URI). CoAP nodes may act as a client and server in parallel since client/server semantics are communicated individually per message. This section presents a summary of CoAP based on RFC 7252 [6].

2.1.1 Interaction Model

Each request sent by a client carries a *token*, an opaque byte array containing up to eight bytes. This token value is used to associate a response to a given request. Whenever a client initiates a new message exchange by sending a request, it generates a token and includes it in the message header. The corresponding response carries the same token, enabling the client to correlate the response to the pending request. This design allows for asynchronous communication with concurrent, interlaced message interactions. Clients carrying out interactions sequentially are allowed to use a zero-length token.

2.1.2 Message Format

CoAP request/response semantics are conveyed by a *code* field in the CoAP PDU. The *code* is a single byte divided into a 3-bit class and 5-bit detail field, expressed as `class.detail`. For instance, a code of `0.02` denotes a `POST` request (class 0, method 2). While requests always bear a code with class 0, responses are categorized into three distinct groups. Responses marked with the `2.xx` code class indicate success, while responses with a `4.xx` or `5.xx` code class denote an error, either due to a client or server failure, respectively. The special `0.00` code applies to empty messages only. In these cases, special logic applies in the messaging sublayer described in the next section.

The CoAP header is followed by options, analogous to HTTP headers such as `Accept` and `ETag`. Akin to HTTP/2 pseudo-headers, the host, path, and queries in a request take the form of dedicated `Uri-Host`, `Uri-Path`, and `Uri-Query` options. Some options are repeatable, i.e., a CoAP message can encompass more than one instance of that option with the same option number. A prominent example illustrating this mechanism is the `Uri-Path` option. The request path is not expressed as a contiguous string, but rather each path component is individually encoded as a `Uri-Path` CoAP option.

CoAP applies several techniques to create compact *Packet Data Units* (PDUs). In contrast to HTTP, each option is assigned a number, and their values such as media types are expressed as integer values, similar to TLV structures (type-length-value). Furthermore, unsigned integers are encoded with variable width, thus lack leading zero bytes for padding and therefore reduce the overall PDU size. Additionally, rather than specifying each option number separately, options in the PDU follow the order dictated by their number and only the difference to the previous option number is encoded. Figure 1 illustrates the delta encoding of option numbers.

```
Index     0    1    2    3    4    5    6    7    8    9    10   11   12   13
        +----+----+----+----+----+----+----+----+----+----+----+----+----+----+
Hex     | B3 | 6C | 65 | 64 | 05 | 63 | 6F | 6C | 6F | 72 | 11 | 32 | 51 | 32 |
        +----+----+----+----+----+----+----+----+----+----+----+----+----+----+
Comment   ①    l    e    d    ②    c    o    l    o    r    ③   50    ④   50
```

Figure 1: Example: delta-encoded options of a JSON request to an `/led/color` resource
Start of first option ①: `0xB3` indicates a delta of `0xB` $= 11$ and a value length of 3. Option number 11 corresponds to `Uri-Path`. Second option ②: `0x05` encodes a number delta of zero, i.e., the option number is also 11, and the value size is 5 bytes. Third option ③: `0x11` indicates a delta of 1 and a value length of 1 byte. The option number $11 + 1 = 12$ stands for `Content-Format`. `50` is the integer value of `application/json`. Fourth option ④: `0x51` starts an option with a delta of 5 and a single-byte value. $12 + 5 = 17$ corresponds to the `Accept` option.

To circumvent the need for a payload length field, a `0xff` byte precedes the payload, following the options part. UDP datagrams and IP PDUs already possesses payload length information from which the total size of a message payload can be inferred.

2.1.3 Layers

CoAP follows a layered approach to facilitate the use of different underlying transport protocols. This is illustrated in Figure 2. CoAP is mostly independent from the transport. The CoAP PDU header varies, but the exchange model and application layer features operating on this request/response layer are shared between CoAP variants.

CoAP was originally specified in RFC 7252 [6] and could only be used in combination with UDP, with support for optionally securing CoAP with DTLS. CoAP over UDP incorporates a thin messaging sublayer for datagram-oriented communication that provides optional reliability and congestion control. Hence, the messaging layer adds additional transport features without requiring a transport protocol like TCP that would entail a larger overhead. With CoAP over UDP, less traffic is produced for reliable retransmission than with a TCP-based protocol like MQTT [5].

RFC 8323 [10] extends the CoAP suite by communication over reliable transports such as TCP, TLS, and WebSockets. There is also an active Internet Draft focused on enabling CoAP over GATT [11]. This section gives an overview of the transport-layer and messaging-layer integration with RFC 7252 [6] and RFC 8323 [10].

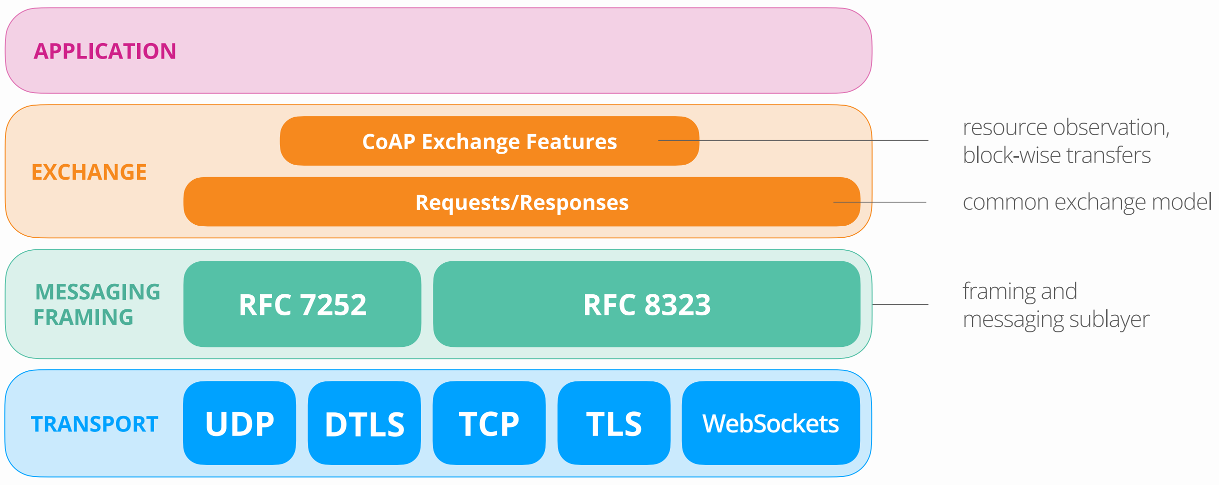


Figure 2: CoAP layers
Located beneath the application, the exchange layer takes responsibility for features that leverage request/response semantics, such as block-wise transfer and resource discovery. Messaging properties and framing vary depending on the transport and are largely independent of exchange-layer functionality.

2.1.3.1 Messaging Layer

CoAP over UDP

As UDP does not provide reliability, RFC 7252 introduces a messaging sublayer that uses four distinct message types: Non-confirmable (`NON`), confirmable (`CON`), acknowledgement (`ACK`), and reset (`RST`). This messaging layer is located between the exchange layer handling requests and responses and the transport layer responsible for UDP and DTLS transmission. Confirmable and non-confirmable messages embody transmission characteristics and are thus independent from the exchange layer, i.e. both message types can carry a request or response.

To uniquely identify transmitted messages, each message carries a 16-bit message ID, accompanying the message type and code in the PDU, as shown in Figure 3. Conceptually, a confirmable message must either be acknowledged by the recipient by sending an `ACK` or be rejected by responding with an `RST` message. Non-confirmable messages are not acknowledged, but rejecting such a message may involve sending an `RST` message. A message ID is only generated by the sender. Acknowledgement and reset messages replicate the ID of the corresponding `CON` or `NON` message. Each CoAP endpoint picks message IDs from its message ID space and must guarantee the ID is not reused within a certain timeframe. Hence, recipients of RFC 7252 CoAP messages are obliged to also check the originating endpoint when correlating message IDs of `CON` and `NON` messages to `RST` and `ACK` messages.

```
 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: Message format for CoAP over UDP [6]

Reliable Transmission. CoAP over UDP involves a simple stop-and-wait protocol with an exponential back-off mechanism. Confirmable messages are retransmitted in exponentially increasing intervals until either an acknowledgement message or a reset message is received. An acknowledgement message indicates the node has successfully processed the message on the messaging layer, i.e., the message may still cause an error on higher protocol levels. When unable to properly handle the message, e.g., due to parsing failures, the message being empty, or receiving a response that does not belong to a pending request, nodes must respond using a reset message. By receiving an `ACK` or `RST`, the sender of the confirmable message stops retransmitting. Neither acknowledgements nor reset messages elicit another `ACK` or `RST` message. Confirmable messages must not be empty to be valid, thus empty `CON` messages will always trigger a reset message. The technique of intentionally sending empty `CON` messages, i.e., with code `0.00`, and waiting for an `RST` in response, is considered a *CoAP ping*.

Retransmission is facilitated through a counter and a timer. To prevent network synchronization effects, timeout values are initially randomly chosen in a timeout range. Whenever the current retransmission timeout elapses, the sender retransmits the message, increments the retransmission counter, and doubles the timeout value. Reaching the maximum retransmission counter value (RFC 7252 [6] suggests 4) indicates packet loss or a processing error, forcing the sender to give up retransmitting the message. Cross-layer events, such as directly receiving a response to a confirmable request or ICMP errors, may also lead the sender to relinquish resending the message before having processed an `ACK`.

Acknowledgements are usually empty, i.e., do not have a payload and carry code `0.00`. However, a response to a confirmable request can be piggybacked onto the acknowledgement of the request, thereby eliminating the overhead caused by sending an empty `ACK` followed by the response message itself.

Unreliable Transmission. Non-confirmable messages provide a means of stateless, unreliable transmission ideal for multicast and repeated operations where the overhead of reliable transmission may be undesirable. In contrast to confirmable messages, non-confirmable messages do not elicit an acknowledgement when received. Provided the node is unable to handle a non-confirmable message, e.g., given a malformed PDU or reserved values being used in the header, the node may optionally send a reset message. Consequently, senders must be prepared to receive `RST` messages during non-confirmable transmissions. Table 1 gives an overview of permissible combinations of message type and code.

Type	Empty Message code `0.00`	Request Message `0.xx` codes, `xx` $\neq$ `00`	Response Message `2.xx`, `4.xx`, and `5.xx` codes	Signaling Message `7.xx` codes not used in RFC 7252
Confirmable `CON`	✓ ping	✓	✓	—
Non-confirmable `NON`	—	✓	✓	—
Acknowledgement `ACK`	✓ following `CON` only	—	✓ in response to `CON` request only	—
Reset `RST`	✓	—	—	—

Table 1: Overview of code and type combinations in RFC 7252 transmissions

Message deduplication. CoAP receivers must be prepared to repeatedly face the same `CON` or `NON` message within a predefined timeframe. A confirmable message may reach a node multiple times if the corresponding acknowledgement has been lost or is still in transit. Unaware of the acknowledgement, the client will continue to retransmit in exponentially increasing intervals. Furthermore, a server handling a confirmable request may not immediately send an empty `ACK` for the request, but instead wait until it has finished processing, ultimately responding with a piggybacked `ACK`. Similarly, more than one datagram containing the same non-confirmable message might be delivered to the node.

Servers can decide whether to handle duplicates again or to ignore them. In the latter case, RFC 7252 [6] recommends the same *kind* of acknowledgement is to be sent, i.e., when the request has been acknowledged using an empty `ACK`, the same empty `ACK` should be sent. If the server decided to respond while simultaneously acknowledging the `CON` request, the same piggybacked `ACK` ought to be sent when encountering further duplicates. Optionally, a server might decide to handle non-confirmable requests only once, ignoring any duplicated `NON` datagrams.

CoAP over TCP, TLS, and WebSockets

CoAP over TCP [10] offers flow and congestion control features that aid in exchanging larger payloads. Relying on TCP instead of UDP also reduces the frequency of sending keepalive messages needed to maintain NAT bindings as TCP connections typically convey more information about the exchange between nodes [13]. Still, TCP implementations require session state to be kept, thus binary sizes and memory usage are increased compared to UDP.

On account of the reliable nature of TCP, CoAP over TCP removes the necessity for an additional reliability mechanism. Thus, retransmission as well as deduplication, and consequently message IDs and the four message types are ommitted, as shown in Figure 4.

TCP being a stream-oriented transport protocol, CoAP message boundaries need to be made apparent. Therefore, a length field was introduced at the start of CoAP messages sent over TCP and TLS. Following the same rationale of compressing unsigned integers, the length indicator takes up 4 bits, with the potential to grow up to 4 bytes.

```
 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Len  |  TKL  | Extended Length (if any, as chosen by Len) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Code     | Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: Message format for CoAP over TCP and TLS [10]
For CoAP over WebSockets, the length field is always set to zero due the length field in the WebSocket frame.

Signaling. Signaling is a concept introduced for CoAP over reliable transports in RFC 8323 [10]. Signaling messages facilitate the exchange of connection parameters, and can be employed to maintain or close the transport session. CoAP control messages used for signaling can be sent by either side and bear codes in the `7.xx` range. Regular CoAP options as in requests and responses must not be embedded in signaling messages in favor of options unique to specific signals.

2.1.3.2 Exchange Layer

This section gives an overview of CoAP features that build upon the request/response exchange model.

Observing Resources

Resource observation [14] augments the transfer model to allow servers to inform clients about subsequent changes to a resource. For instance, a server may want to asynchronously notify clients whenever a sensor measurement or device state changes. The goal is to keep resource representations between clients and a given server in sync. With HTTP, a node would enroll in long polling or switch to WebSockets. CoAP on the other hand offers a feature that enables clients to *observe* a resource by initiating a primal request, expressing interest in future changes, and servers to send *notifications* at a later point in time. The message flow described in [14] is illustrated in Figure 5.

Conceptually, a resource located on a server represents an event source to which interested observers can subscribe. CoAP permits multiple client observers to be subscribed to a given resource and a single observer can subscribe to multiple resources. A server is required to store the client intent to receive *notifications* from a resource, called a *registration*. The registration is facilitated through a regular CoAP `GET` request to the resource, followed by a regular response. The observation intent is communicated through the `Observe` option, which must be set to `0`. The server keeps a list of observing endpoints for each resource.

Each time the server encounters a change in the state of a resource, it sends an additional CoAP response message. This *notification* is marked with the same token as in the original request and the initial response. By including an `Observe` option in each notification message, it provides an increasing sequence number to enable the ordering of notifications when CoAP is employed over unreliable transports. CoAP over reliable transports omits this sequence number due to the guarantee that message order is maintained [10, Section 7.1]. To deregister from receiving notifications, nodes send another request with `Observe` set to `1`. In CoAP over UDP, a client may alternatively simply drop state associated with the outstanding observation, prompting a reset message to be sent when another notification arrives. Receiving this `RST` message, the server will cancel the registration [14, Section 3.6], i.e., remove the client endpoint from its list of resource observers. Naturally, this method is not applicable for CoAP over reliable transports, hence the registration must be cancelled explicitly

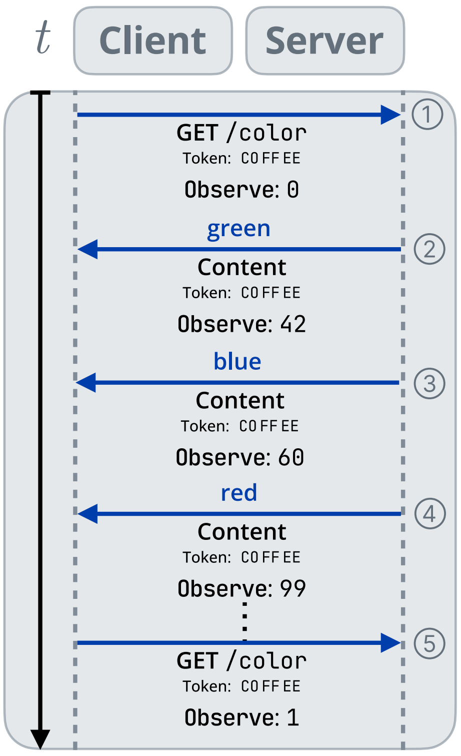


Figure 5: Observing a CoAP resource
First, the client node sends its initial request ①, expressing its intent to receive further notifications. The initial response ② contains a payload of *'green'*. Further responses ③ ④ are notifications with different sequence numbers. Finally, the client unsubscribes from the `/color` resource ⑤.

Block-Wise Transfer

Network-layer and link-layer fragmentation can result in increased packet loss and should therefore be avoided in lossy IoT networks [3]. To avoid fragmentation on lower levels, such as due to the 802.15.4 maximum frame size of 128 bytes or the IPv6 MTU of 1280 bytes, RFC 7959 [8] defines *block-wise transfer*, a CoAP extension that enables exchanging large payloads in application layer *blocks*. This section outlines block-wise transfers in CoAP.

Block-wise transfers are facilitated through a set of CoAP options, through which bigger request or response *representations* can be exchanged across a series of request-response pairs. These elementary requests and responses transfer a single payload block each, leveraging regular message semantics. In block-wise transfers, the block size can be negotiated. Additionally, block-wise transfers can be implemented in servers in a stateless fashion. Hereinafter, *representation* refers to the content an endpoint wants to exchange with another, i.e., either as a request (client) or response (server). The term *payload* will be used for the data inside a single CoAP message. The total representation is divided into blocks. A client that aims to send a large representation, e.g., in a `POST` or `PUT` request, uses the `Block1` option. A large response representation can be sent in a series of responses with the `Block2` option. Block sizes must be a power of two and fall in the range of $[16, 1024]$. Furthermore, messages transmitted over reliable transport connections are allowed to contain multiple blocks of size 1024 (This feature is called BERT).

Transferring request representations. To transfer a large request representation, a client must split it into chunks of the chosen block size. The attached `Block1` option indicates the block size in the form of the exponent of the power of two. It also states whether the sent block is the last block in the block-wise exchange. Moreover, the `Block1` option contains the index of the transmitted block in the sequence of all previous and future blocks. Setting the `Block1` option in a request is considered its *descriptive use*, i.e., it describes the attached block payload. Upon receipt of the block request, the server can start processing the block in a streaming fashion or reassemble blocks, to ultimately handle the entire representation at once. If the aggregate representation size is too large or missing a block, a server can use the status code `4.08` *Request Entity Incomplete* or `4.13` *Request Entity Too Large* respectively. Unless the final block has arrived, where the server would send its regular response, it must respond with status `2.31` Continue to ask for further blocks to be sent. The `Block1` option in a response, i.e., in *control usage* signifies the number of the block being acknowledged and the block size suggested by the server. Generally, after the last block has been received, the server responds with its resource. Only a single `Block1` transfer can be active at a time per resource and client endpoint.

Transferring response representations. The `Block2` option works similarly to `Block1`. In a response, a server can set the `Block2` option to indicate the response is transferred block-wise. In this case, which is considered the block option *descriptive usage*, the block option indicates whether more blocks can be requested, and the block index and size. A client can suggest a block size by including a `Block2` value in the initial request (*control usage*). In any further request, the block option contains the requested block number. The block size in the `Block2` option in any further request must match the negotiated size. Whenever a subsequent `Block2` request is received, the server returns a response with the `Block2` option. Like `Size1`, a requester can supply the `Size2` option to demand an estimated resource size. In block responses, the `Size2` option denotes the current size estimate of the resource representation. Effectively, this design also permits requesting specific regions of a resource representation provided the block offset is a power of two.

OSCORE

Object Security for Constrained RESTful Environments (OSCORE) [9] is a security extension used to protect CoAP messages on the application layer. OSCORE is independent of the chosen transport protocol. OSCORE provides end-to-end message protection by encrypting and signing a CoAP message using CBOR Object Signing and Encryption (COSE). The resulting COSE object acts as the payload of an outer CoAP or HTTP message carrying an OSCORE CoAP option or HTTP header, signaling the use of OSCORE. As only the inner message is protected, the CoAP messaging layer and token are not secured by OSCORE. Cryptographic operations depend on pre-shared keys, i.e., keys must be established out of band. The data necessary to perform cryptographic operations on OSCORE-protected messages is called *security context*. Each party in an OSCORE exchange possesses a shared security context [9].

2.2 CoAP Support in RIOT

At present, RIOT features multiple CoAP libraries: nanoCoAP [15], nanoCoAP Sock [16], and GCoAP [17]. CoAP support originated in nanoCoAP, a small CoAP implementation, providing a message parser and option manipulation functions, as well as helper functions for blockwise transfers. nanoCoAP Sock offers a simple CoAP client and server API for CoAP over UDP and DTLS. Messages are handled synchronously, preventing interlaced message exchanges. In contrast to nanoCoAP Sock, GCoAP implements a high-level CoAP interface with asynchronous messaging. Still, it relies on the nanoCoAP parser, and using GCoAP requires calling nanoCoAP APIs. In Table 2, we compare the feature set of nanoCoAP Sock and GCoAP. Several modular CoAP extension implementations exist in RIOT, such as a forward proxy and resource observation implementation for GCoAP and a virtual filesystem module for nanoCoAP Sock. However, CoAP support in RIOT is currently limited to RFC 7252, thus varying the underlying transport protocol is not possible. Moreover, both libraries require in-depth protocol knowledge.

Feature	nanoCoAP Sock	GCoAP
High-level transport abstraction	×	×
CoAP over UDP	✓	✓
CoAP over DTLS	✓	✓
CoAP over reliable transports	×	×
OSCORE	×	×
Parallel message exchanges	×	✓
Asynchronous client API	×	✓
Synchronous client API	✓	×
Resource observation	×	✓
Automatic block-wise transfers	×	×
Separate responses (RFC 7252)	✓	×
Message deduplication (RFC 7252)	×	×
Caching	✓	✓
Convenient options API	×	×

Table 2: Feature comparison of nanoCoAP Sock and GCoAP [16, 17]

2.2.1 nanoCoAP

nanoCoAP [15] is a small library for parsing and serializing CoAP messages and interacting with options. It also lets applications define resources and provides helper functions for block-wise transfers.

nanoCoAP represents messages using a `coap_pkt_t` structure as illustrated in Listing 1. The packet structure provides a *view* into a CoAP PDU backing buffer. The creation of a new message spans across multiple steps. At first, the process involves initializing the structure with a backing buffer using `coap_pkt_init`. This sets the header and payload pointers accordingly. The application must prepare the header by invoking `coap_build_hdr` which sets the message type, message ID, code, and token. The memory layout of the `coap_hdr_t` structure directly models the header of a CoAP PDU. However, this design is incompatible with CoAP PDU headers of other transport variants. Substituting the header with a union of various header formats would still mandate knowledge of the underlying protocol in each code path to access fields like the `code`, placing an additional burden on the implementation to always check the transport type.

```
typedef struct {
  coap_hdr_t* hdr;
  uint8_t* payload;
  // [...]
  uint16_t payload_len;
  uint16_t options_len;
  coap_optpos_t options[CONFIG_NANOCOAP_NOPTS_MAX];
  // [...]
} coap_pkt_t;
```

Listing 1: `coap_pkt_t` structure [15]

```
typedef struct __attribute__((packed)) {
  uint8_t ver_t_tkl;
  uint8_t code;
  uint16_t id;
} coap_hdr_t;
```

Listing 2: `coap_hdr_t` structure [15]

nanoCoAP implements several methods for configuring options, both for predefined options such as `Content-Format` and, e.g., opaque string and integer options. However, adding options requires calling methods in the order dictated by option numbers, forcing developers to be aware of the number of each option. Once options are added, an application must call `coap_opt_finish` which optionally adds the payload marker. After options and the payload has been added, no further options can be added. Besides, once the payload is set, the stack beneath is unable to add options whose number may precede the option added last. This would complicate the integration of advanced CoAP features in the stack that involve options such as `Observe` or `Block1`. These features would act on the message after the application has added its payload.

2.2.2 nanoCoAP Sock

nanoCoAP Sock [16] builds upon nanoCoAP by implementing a synchronous interface for clients and servers. Clients first need to initialize a socket object shown in Listing 3 by invoking `nanocoap_sock_connect` (UDP), `nanocoap_sock_dtls_connect` (DTLS), or `nanocoap_sock_url_connect`. The latter allows specifying a scheme, host, and port. Each of these initializers sets up the UDP socket, and optionally establishes a DTLS session if the DTLS initializer was used or the URI scheme is equal to `coaps`. The transport employed is stored in the socket in the `type` property. Like nanoCoAP, nanoCoAP Sock supports only RFC 7252 through APIs like `nanocoap_sock_get` and `nanocoap_sock_put`, which send confirmable requests. Unreliable transmission is facilitated through functions with a `_non` suffix, such as `nanocoap_sock_get_non`. Moreover, nanoCoAP Sock offers convenience functions for each request method that avoid creating a socket first. These APIs carry a `_url` suffix. Yet, nanoCoAP Sock lacks APIs that accept a URL for sending non-confirmable requests. This exemplifies the difficulties induced by the variety of combinations of request parameters, such as the message type and the way of specifying the request destination.

```c
typedef struct {
  sock_udp_t udp;
  sock_dtls_t dtls;
  sock_dtls_session_t dtls_session;
  nanocoap_socket_type_t type;
  uint16_t msg_id;
} nanocoap_sock_t;
```

Listing 3: `nanocoap_sock_t` structure [16]

The nanoCoAP Sock client has support for block-wise requests. Responses split into blocks (Block2) can be received individually using `nanocoap_sock_get_blockwise` or reassembled with `nanocoap_sock_get_slice`. Requests may also be transmitted block-wise (Block1) using `nanocoap_sock_block_request`.

Server functionality in nanoCoAP sock involves defining resources at compile time, each accepting a set of methods. Upon receiving a request destined for an application-defined resource, the library calls the provided callback. Said callback may either immediately modify the given response buffer and return or invoke `nanocoap_server_prepare_separate` to delay response after the handler has returned. In this case, the deferred response can be sent using `nanocoap_server_send_separate` and a request context created by the preparation function. This technique avoids blocking the event loop while performing expensive operations in the request handler. The nanoCoAP Sock library does not support deduplication, hence handlers must be prepared to be invoked multiple times with the same message.

2.2.3 GCoAP

GCoAP [17] is a high-level CoAP stack in RIOT that enables multiple, interlaced asynchronous message exchanges. GCoAP enables asynchrony through a dedicated background thread that processes inbound messages. It depends on the nanoCoAP packet interface, but wraps these APIs in dedicated `gcoap_req_init` and `gcoap_resp_init` initializers. Populating a message involves calling several functions. Once populated, requests can be sent using `gcoap_req_send`, with the supplied callback being called asynchronously when the response is received. While the client API accepts an argument for transport layer selection, the type representing an endpoint is limited to UDP and DTLS over UDP.

Internally, GCoAP retains an array of request *memos* responsible for tracking outstanding requests and retransmissions. A memo is equipped to hold a PDU copy for retransmissions, a timer and a counter for the exponential back-off mechanism, and the response callback. The memo also stores the server endpoint and a potential DTLS session context. This part of the design moves the responsibility of tracking exchange state into the stack and permits multiple exchanges to be active in parallel. GCoAP also allows control over the number of memos through compile-time configuration in RIOT.

A GCoAP server leverages the same callback API introduced in nanoCoAP. However, while nanoCoAP Sock allows sending separate responses, GCoAP does not permit such behavior, i.e., responses take the form of either a non-confirmable message or a piggybacked acknowledgement. This approach complicates scenarios where asynchronous work is performed in a request handler, such as in proxies delegating incoming requests. Like nanoCoAP Sock, GCoAP does not support deduplication.

Moreover, GCoAP implements server features for observing resources. Like request memos, state management necessary for resource observation is integrated into the stack. A list is subscribed endpoints is maintained and notification packets may be constructed by the application using `gcoap_obs_init` and sent to observers by calling `gcoap_obs_send`. In summary, GCoAP implements several features that would otherwise entail keeping state in the application but is still limited to UDP and DTLS, partly due to the nanoCoAP packet API.

3 Design

CoAP is a versatile protocol. A CoAP suite should address the entire spectrum of transports and extensions. A high-level interface to CoAP is necessary to provide consistent, low-threshold access to CoAP features across applications. Removing the burden of implementing extensions such as block-wise transfer in applications centralizes maintenance work and lowers entrance barriers for developers. The `unicoap` design tries to hide implementation constraints imposed by CoAP to be more accessible to users while still offering means for full control for advanced users. `unicoap` integrates client and server functionality and focuses on flexibility, accounting for a variety of scenarios, such as asynchronous and synchronous API usage.

A key incentive behind the design of `unicoap` is to enable the seamless use of different underlying transport protocols. It aims to minimize changes necessary to switch to another transport. By grouping features and mechanisms in layers, thereby modularizing CoAP, `unicoap` defines a path for incorporating new transports in the future. In `unicoap`, *a CoAP combination* denotes the slightly modified version of CoAP for each transport protocol. Information with semantics tied to a particular CoAP combination or a set of CoAP combinations, such as the message ID, is hereinafter denoted as *transport-specific* data. The terminology used throughout the following chapters is defined Appendix A.

The design of `unicoap` involves three distinct layers that reflect the layered approach of CoAP, as shown in Figure 6. Conceptually, newly received message traverse these layers up to the application, and data sent by the application travels in the opposite direction. Located beneath the application, the *exchange* layer embodies the REST model of CoAP. It is responsible for handling advanced CoAP features operating above the request-response model, such as resource observation and block-wise transfer. This layer is shared between CoAP combinations. Since framing and messaging differ between CoAP combinations [6, 10, 11], a modular design to ease the addition of new CoAP combinations is necessary. The layer dedicated to CoAP messaging covers framing and can accommodate a custom reliability mechanism, such as the one specified in RFC 7252 [6]. Serializing messages and parsing PDUs received from the network are also handled by the messaging layer. The transport layer at the bottom manages different transport protocols. Here, `unicoap` coordinates with the operating system networking interface.

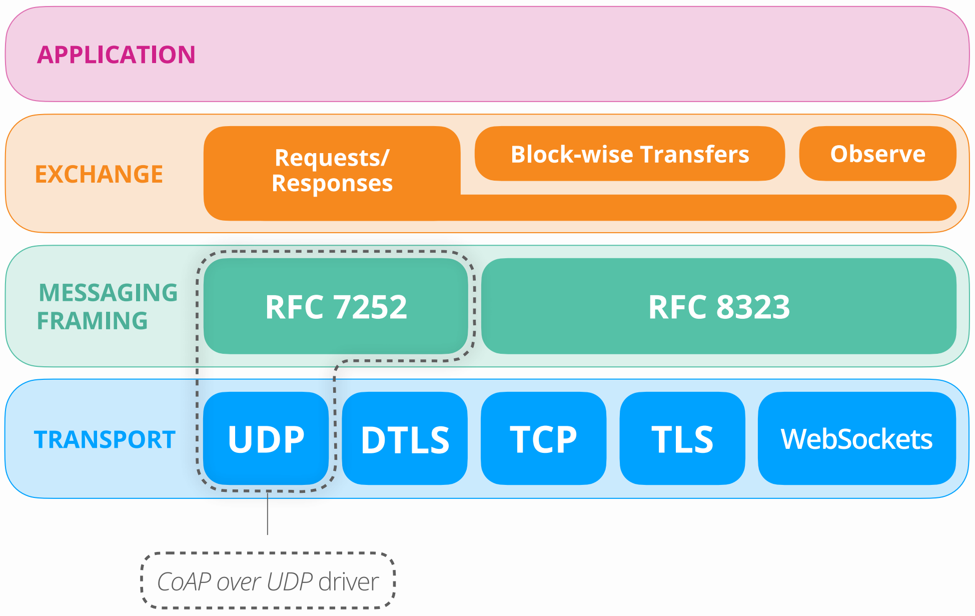


Figure 6: CoAP layers

This chapter is structured as follows. First, the user-facing API is described on a high level, alongside client and server features. Then we outline functionality for automatic block-wise transfers. Finally, we derive a design for state management.

Messages. The primary incentive for revising the message container from nanoCoAP is flexibility. The new container aims to provide an interface suitable for every CoAP combination and PDU format. The design of the object representing CoAP messages follows a compartmentalized approach. As the CoAP header differs between CoAP combinations, `unicoap` separates common message contents from transport-specific data. Additionally, information only relevant to the stack to retain state is detached from the representation of a message. As a result, the public data structure embodying a generic CoAP message comprises only the message code, its payload, and options. Options themselves are stored in a separate object. Detaching them from the main message object allows adding options after the payload has been set and therefore avoids the need for building a PDU step by step.

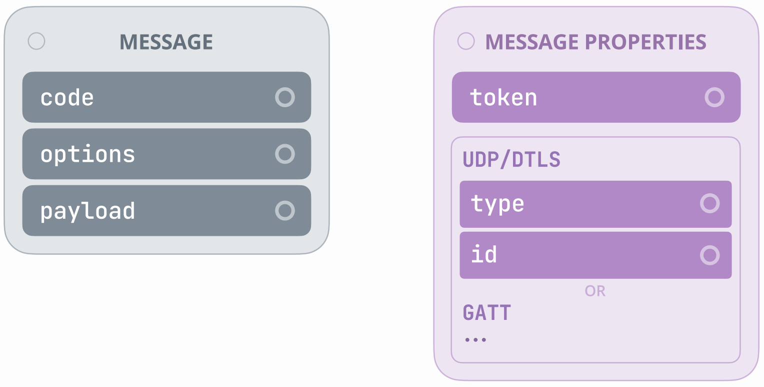


Figure 7: Message and message properties

Message properties. `unicoap` refers to the token, message ID, and message type as *message properties* which are stored in a dedicated object, as demonstrated in Figure 7. These properties are only set by `unicoap` internals. This aids in only exposing an interface to strictly necessary data like payload and options in the public client API while still permitting auxiliary message properties to be provided as a supplement when emitting data into the application. Transport-specific message properties in the properties object are stored in a union type. This approach solves the problem of varying header formats. For instance, the message type and ID are represented differently in CoAP over UDP [6] compared to CoAP over GATT [11] and are not present in CoAP over reliable transports. This design avoids the previously required awareness of transport headers and removes the need for a backing buffer for the header.

Options. The options API offers means for listing all options, reading specific options, adding, modifying, and removing options. It defines methods for setting options in a strongly typed fashion while offering functions for manipulating generic options in parallel.

`unicoap` supports a set of option operations to account for repeatable and non-repeatable options. The latter support *get*, *set*, and *remove* accessors. Repeatable options can be manipulated using *add* and *remove* methods and may be read using an *options iterator*. The iterator supports a *next* method for enumerating all options and a *next by number* method that facilitates iterating over values of a specific repeatable option, such as multiple `Uri-Path` components. In addition to the iterator, convenience methods are provided that automatically build the aggregate representation from repeatable options, such as the full path from multiple `Uri-Path` options. This functionality is abstracted into a generic method that accepts a separator byte used to join individual option values.

For most options registered by the IANA, `unicoap` defines typed accessors. Moreover, generic accessors are provided for custom options and common formats, such as strings, integers, bytes, and booleans (A boolean option encodes its values via its presence and has a zero-length value).

3.1 Client

The client API of `unicoap` aims to solve issues in the GCoAP and nanoCoAP Sock client interface. The client API aims to be flexible in a variety of dimensions. First, applications must be able to use all request methods in combination with most other settings. Additionally, the destination of a request allows for different representations. The most straightforward mode of addressing a server is by passing in a network and transport layer address, such as an IPv6 address literal and a UDP port number. In this case, the caller would set the request path manually by adding individual `Uri-Path` options, if applicable. An application might also want to supply a URI instead. This method of expressing the destination has the added benefit of providing the protocol, the network and transport address, the path, and queries in a single string. In the future, applications may want to use Concise Resource Identifiers (CRIs) instead, or just present a host string, relying on Service Binding Records (SVCB) in the DNS [18]. Furthermore, provided the request is transmitted over UDP or DTLS, callers should be able to specify the message type to use. The API must be capable of accommodating further extensions such as OSCORE.

Finally, various use cases substantiate the need for a synchronous API in parallel to asynchronous methods. Feedback from the community indicated that, while more versatile, the asynchronous nature of the client interface sometimes hinders the development of readable code. Applications may not perform any work until the response has arrived. This frequently leads to application logic being nested in response handlers or requires working with semaphores.

Among all dimensions of variety in the API, `unicoap` only exposes the concurrency axis as individual functions to reduce the overall number of methods. An alternative approach would have been to model concurrency behavior as parameters. However, this technique would not adequately capture the extent of variation in behavior and might even be infeasible in languages with support for async/await keywords. The remaining settings and extensions take the form of function parameters.

The request method is passed as an argument. To allow specifying an endpoint object or a URI without creating overloads, the request methods accept a destination object that can take different representations using a tagged union. Given a destination object, `unicoap` will check if it needs to resolve the destination object into an endpoint. For CoAP URIs, `unicoap` will also attempt DNS resolution, given a host.

Furthermore, a boolean reliability setting is introduced, supporting the specification of the message type while abstracting the underlying transport mechanism. When used with inherently reliable transports, the reliability flag has no effect. In combination with a UDP or DTLS endpoint or URI, it turns on the optional reliability mechanism in RFC 7252 [6], forcing `unicoap` to use confirmable messages. To reduce memory occupied by a boolean flag, the reliability flag is integrated into a bitfield that can accommodate further flags for resource observation. A registration and cancellation flag is introduced to manage resource subscription memberships. On account of the asynchronous properties of CoAP notifications, these flags are only compatible with the asynchronous request version. The same provided callback is then invoked for each notification received, reflecting the similarity of the original response and additional notification responses.

To enable support for OSCORE and potential future extensions akin to OSCORE without entailing source-breaking changes, a *profile* data structure is introduced. A profile implies special treatment is to be applied to request and response messages, e.g., encryption and decryption with OSCORE. For OSCORE, the profile would contain the security context.

While processing a response, additional contextual information might be necessary. For instance, if the same code path is used with different transports, the application should be notified of the remote and local endpoint. For debugging purposes, the application might also want to inspect the message token or message ID. To forward these additional values to the application, an auxiliary information object (`aux`) is emitted alongside the response. The current profile, e.g., the OSCORE security context, is also passed as part of the auxiliary object. This design also permits further additions without breaking source compatibility or elongating parameter lists. Figure 8 shows the final list of parameters, with the `throws` keyword indicating that the given function can fail. Objects listed behind the → symbol are to be interpreted as return values.

```
unicoap_send_request_ sync  async (
  request , destination , flags , profile ) throws → response , aux
```

Figure 8: Request methods

3.2 Server

The design of the server API builds on the GCoAP interface. Like nanoCoAP and GCoAP, `unicoap` explicitly represents CoAP resources as data structures. Each resource is bound to a path and a set of methods. Whenever a request is received, it is first matched against a resource. `unicoap` augments the resource object with a set of allowed protocols to ease restricting the reachability of a given resource to only a subset of transport protocols. For instance, a resource processing sensitive information may only want to accept requests received over DTLS and TLS. Functionality from GCoAP is retained, e.g., resources can be configured with flags to match a path prefix. Like GCoAP, `unicoap` also supports grouping resources into *listeners* as illustrated in Figure 9. This technique simplifies the development of modularized CoAP applications for accessories that, e.g., share resources for pairing, but integrate distinct sets of resources for controlling actuators or reading sensor data. In light of different transports, `unicoap` furthermore lets developers assign a set of protocols to each listener. This removes the need to explicitly constrain each resource to a number of transports in favor of a transport policy that applies to the entire listener. Additionally, rules for determining whether a request matches a resource can be customized by providing a *matcher* as part of the listener, which has also been present in GCoAP.

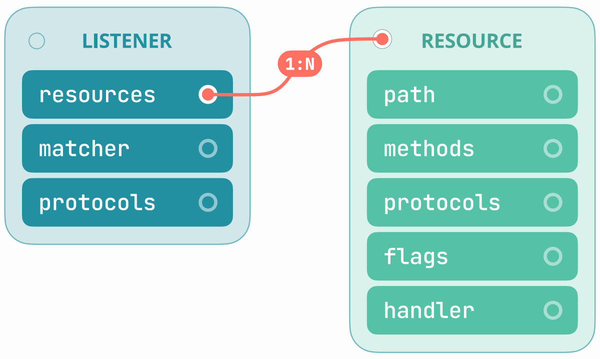


Figure 9: CoAP listener and resource objects

`unicoap` supports deduplication of requests transmitted over UDP and DTLS through a deduplication flag that can be set when defining a resource. Provided a resource matches an incoming request and is not filtered, the associated function prototype shown in Figure 10 is invoked, processes the request, and may send a response. The handler is given the request and auxiliary data, establishing parity with the client API. The handler function is also passed a context the application can use to send a response. Calling `send_response`, illustrated in Figure 11, consumes the context. Response transmission characteristics can be configured with flags. To align the design with the client API, a reliability flag can be set on a resource that instructs `unicoap` to choose confirmable messages for the response, given an RFC 7252 endpoint. This approach also fits other reliability mechanisms, such as the one in CoAP over GATT [11].

```
handler(request, aux, context) throws
```

Figure 10: Request handler parameters

```
send_response(response, context) throws
```

Figure 11: Response function

Conceptually, there are two modes of responding. A server may be capable of immediately providing the response—which may involve brief, blocking, synchronous work. Alternatively, the server could perform an additional, more expensive operation before replying. This is especially true for proxies that delegate requests by initiating a second request as a client of the origin server. `unicoap` provides a means for the application to signal that processing will take longer, granting `unicoap` the chance to send an empty acknowledgement in the meantime. `unicoap` ensures a response is sent eventually using a timeout.

Whenever a request with the `Observe` option reaches the server, `unicoap` automatically registers or unsubscribes the client from future notifications. Therefore, it appends or removes a new registration object to its list of registered clients. Each list element contains the respective client endpoint, the resource to send notifications from, the token to use for notifications, and the profile. Keeping the profile is necessary to encrypt notifications if the original request was also protected by OSCORE [9]. In its server API, `unicoap` exposes a `send_notification` method, which sends a provided message as a notification to all clients subscribed to the given resource. Akin to the `send_response` API and the reliability flag, a *notification reliability* flag can be set on the resource for precise control over the message type used for notifications.

3.3 Block-Wise Transfer

`unicoap` provides a means for automatically performing block-wise transfers [6] to mitigate the extent of work demanded from the application. Both the client and server API offer support for letting the stack beneath manage the block-wise transfer. On one hand, an application might want to send a large `POST` payload using several `Block1` requests. On the other hand, a client should be able to instruct `unicoap` to automatically ask for `Block2` responses. In the latter case, the extent of support required from the CoAP stack may vary. To start, an application may want to process just the reassembled, contiguous representation of the response present on the server. In scenarios where maintaining a buffer large enough to fit the entirety of the reassembled payload is infeasible, letting the application process each block individually is more appropriate. Furthermore, a server may want to act on the entirety of a client request instead of processing each block separately. Hence, the server API should facilitate reassembling multiple `Block1` requests.

`unicoap` refers to the entirety of multiple block payloads as a *representation*. This may be a request representation transmitted in a `Block1` transfer or a response *representation* in a `Block2` exchange. `unicoap` applies the following terminology to the degree of support for automatic block-wise transfers.

— **Slicing** is the process of dividing a larger representation into smaller chunks and sending them to the peer.
— **Collecting** involves requesting more blocks from the peer and tracking progress, acting on each block individually.

— **Reassembling** denotes the process of requesting more blocks, tracking progress, and copying blocks to a larger buffer, thereby recreating and processing just the original, contiguous representation.

Each of these *modes* is applicable for `Block1` and `Block2` transfer *stages*, as shown in Table 3. The stage dictates the kind of representation that is transmitted, i.e., the request in a `Block1` and the response in a `Block2` transfer. Slicing and collecting is facilitated through flags passed as parameters in the client API or as part of resource definition on a server. Given the *auto-slice* flag, `unicoap` obtains the full representation from the application above and elicits multiple request-response pairs that are exchanged. The *auto-reassemble* flag causes `unicoap` to recreate the full representation before it is emitted into the application as a response or new request. In each of these cases, the application can be unaware of the block-wise transfer, treating the aggregated message as usual.

Stage	Client	Server
`Block1` Transfer of request representation	Slicing	Reassembling
`Block2` Transfer of response representation	Collecting/ reassembling	Slicing

Table 3: Block-wise terminology and `Block` headers

Unlike slicing and reassembling, collecting requires application awareness. Thus, this feature cannot be expressed by merely a flag. `unicoap` defines an additional client method operating under a special contract, which enforces awareness using a callback invoked for each new response block. Depending on the language, this could also be modeled using polling, asynchronous collection types, or asynchronous iterators. The per-block callback allows handling incoming data in a streaming fashion and also permits aborting the transfer partway through. To lift the burden of parsing the `Block2` value, `unicoap` propagates a *block info* object to the application, providing access to the block size, offset, its number, whether more blocks follow, and whether the transfer uses BERT. Akin to the regular request methods, two versions are provided for block-wise collecting: a synchronous and an asynchronous API, as depicted in Figure 12. The same approach could be applied to the server API but would necessitate keeping additional state in the application since more than one block-wise transfer can be in progress per resource. As each resource handler is already invoked for each request, the added complexity of automatically collecting request blocks seems disproportionate and would effectively only remove the need to set the block option and call `send_response` for intermediate blocks.

```
send_request_with_block_callback_ sync  async (
   request , destination , block callback , flags , profile ) throws

block callback ( response , aux , block info ) throws
```

Figure 12: Additional block-wise request methods for collecting response blocks

`unicoap` exposes the underlying functionality for the cases detailed in Table 3 as public APIs. This also ensures consistent behavior regardless of whether transfers are managed by the stack. These APIs leverage a common *block iterator* type that can be configured in one of the three modes, i.e., the iterator acts either as a slicer, collector, or reassembler.

Figure 13 demonstrates the communication between a `unicoap` client and a `unicoap` server with automatic reassembly and slicing enabled on both sides.

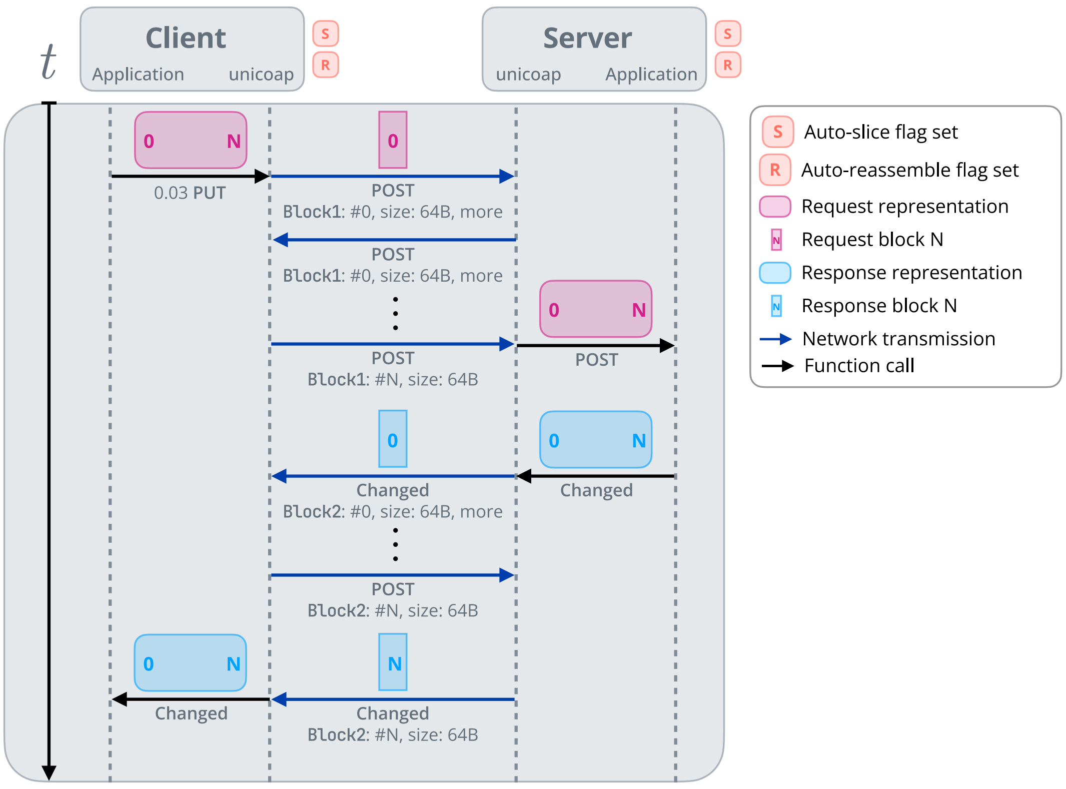


Figure 13: Block-wise transfer of a request and response, coordinated by `unicoap`

3.4 Drivers

Transport endpoints. `unicoap` assigns each underlying transport a protocol number, similar to POSIX sockets. This protocol number instructs `unicoap` to select the corresponding CoAP combination. CoAP endpoints are communicated to `unicoap` using a tagged union type representing the transport type and transport-specific address, such as an IP address and UDP port number for UDP endpoints or a MAC address for GATT nodes. `unicoap` treats this as an opaque object until it is inspected by the messaging and transport layers. Located beneath the messaging layer, the transport layer unwraps the endpoint object when prompted. Usually, the unwrapped representation will leverage the networking library or peripheral objects. Consequently, after obtaining PDU data from the network, the transport layer wraps the source address obtained from the networking API in an endpoint object.

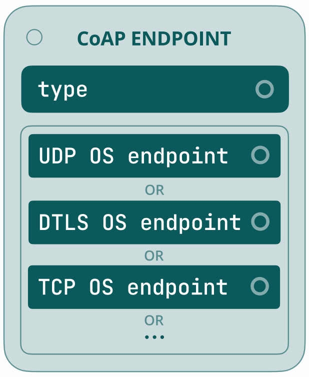


Figure 14: Endpoint object

The RFC 7252 messaging model could theoretically also be applied to CoAP over reliable transports if messages were treated like non-confirmable messages. In light of future CoAP combinations, such as CoAP over GATT, this design proved to be infeasible as further CoAP combinations are not guaranteed to be compatible with a subset or superset of the original RFC 7252 messaging layer[1]. Hence, custom messaging functionality is needed for each CoAP combination.

Furthermore, a *single* implementation of the messaging layer may be used in combination with *multiple* transport protocols. For instance, CoAP over UDP involves the same, shared RFC 7252 messaging sublayer, but does not require session management needed for CoAP over DTLS. Both CoAP over UDP and DTLS build upon the same PDU format and messaging sublayer. Similarly, CoAP over TCP and TLS utilize the same PDU format, yet the correspond-

[1]according to personal discussions with Christian Amsüss, one of the active IETF CoRE working group members, during the RIOT Summit 2024

ing transport components are separate. As a result, `unicoap` divides functionality that is specific to a CoAP combination into a messaging and a transport layer.

To integrate new CoAP combinations, functionality for these layers must be added. The `unicoap` design refers to these integrations collectively as a *driver*, representing a CoAP combination, such as CoAP over DTLS. Drivers themselves can in turn consist of a shared module for messaging and a specific transport support module. For example, the CoAP over DTLS driver encompasses a transport module for DTLS networking; and depends on the common RFC 7252 messaging module also employed by the CoAP over UDP driver. In contrast, CoAP combinations such as CoAP over GATT whose messaging layer is in a one-to-one relationship with the underlying transport protocol do not entail a mandatory separation into a messaging and transport module.

Each driver must comply with the following interface.

— `init`, `deinit`: Drivers must provide an initialization and teardown method. These may be used for setup work in the transport and messaging layer such as for creating sockets or establishing connections to peripherals, alongside allocating objects required for messaging.

— `send`: A driver must expose a standardized API for sending from the messaging layer. The exchange layer will call into this functionality, prompting the driver to perform any due work in the messaging layer like attempting to retransmit the message. In parallel to the message, as well as the remote and local endpoint, this procedure accepts flags that customize transmission behavior. The RFC 7252 message type is abstracted into a *reliability* flag the messaging layer in the CoAP over UDP and DTLS drivers interpret as an instruction to send a confirmable message. When finished, the messaging layer serializes the message and forwards it to the transport implementation.

— `ping`: Due to the variability in ping mechanisms (empty `CON` in CoAP over UDP and `7.03` message in CoAP over reliable transports), each driver can implement a ping function. `unicoap` bundles these APIs and provides a single, generic ping method that multiplexes between the driver implementations.

When the transport layer of a driver receives data, it propagates it to the messaging layer for further processing. Ultimately, the messaging layer will invoke the exchange-layer message `process` function. Processing is divided into two steps. A preprocessing step must be completed, ensuring the exchange layer *is capable* of properly handling the newly received message. This includes checking if a response matches an outstanding request, affecting messaging layer behavior. For instance, according to RFC 7252, unknown confirmable responses must be countered with a reset message. In contrast, a confirmable response awaited by `unicoap` is to be acknowledged. This also applies to CoAP notifications transmitted over UDP. The design of `unicoap` ensures such checks are completed *before* the client inspects the response to obviate the risk of a timing attack.

The functionality for encoding and decoding messages is modularized. In general, PDU processing is divided into a common options-and-payload step and a step unique to the header, depending on the transport. This involves a common function for serializing and parsing the message part that follows the transport-specific header, i.e., the PDU excluding the options section and the payload. Each transport-specific parser and serializer first handles the corresponding header and then calls into the provided, shared function for the remainder of the PDU. Moreover, the signatures of transport-specific parsing and serializing methods are standardized in `unicoap`. A parser always accepts raw PDU data and conceptually returns a generic CoAP message object.

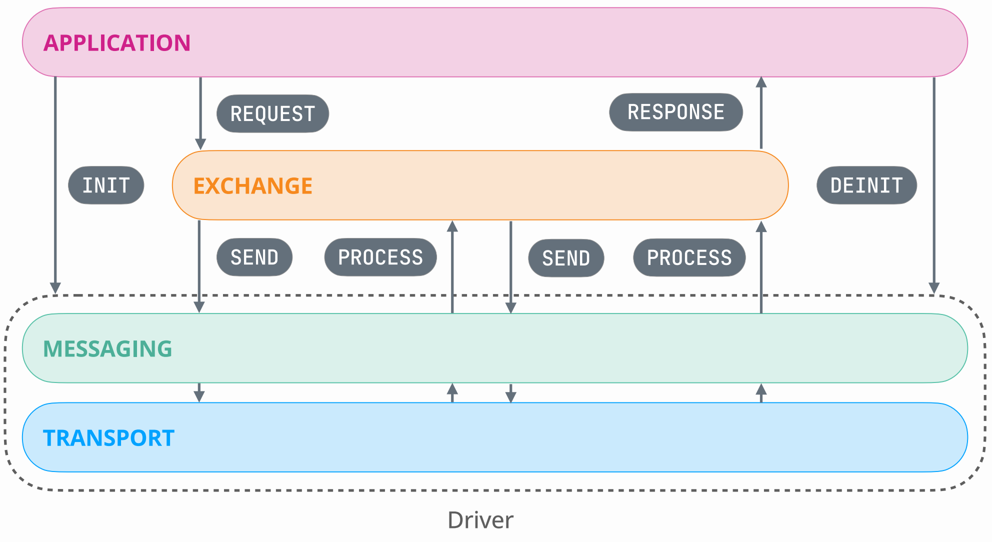


Figure 15: Communication between layers
In a block-wise transfer, a client request from the application may result in multiple `send` and `process` calls between the exchange and messaging layer.

3.5 State Management and Timers

`unicoap` adopts the notion of *exchanges*. An exchange spans the entire duration needed to convey a request representation to a server and the response representation back to the client. Exchanges are coordinated by the exchange layer and can involve multiple requests and responses, e.g., in a block-wise transfer, or additional notification responses when resource observation is used. An exchange is not considered complete until the last response block or notification has been transferred to the client.

The `unicoap` client needs to retain a callback for outstanding exchanges to delegate responses to the application. Furthermore, an exchange is made up of phases, each using a unique token. For each request in that exchange and associated responses, e.g., with additional notification responses, the token must be stored to match incoming responses to the ongoing exchange. This also enables `unicoap` to propagate an error to the messaging layer if the response does not belong to an existing client exchange. The server component must keep information on deferred requests in order to automatically send a server error response if the application fails to respond within a specified timeframe. Furthermore, both the client and server part must store state to manage a block-wise transfer, including the current block number, size, and perhaps also a buffer for reassembling.

In contrast, a *transmission* denotes the process of sending a single message, which may entail retransmitting it. As an exchange encompasses multiple messages, an exchange can be divided into individual transmissions that end once the messaging layer has determined the message has arrived at the receiving node. With reliable transports, no messaging state must be kept. However, in RFC 7252 transmissions, a copy of the confirmable message sent must be available for retransmissions, alongside a counter. Furthermore, for filtering duplicate requests, the message ID must be kept. According to RFC 7252 [6], each duplicated confirmable request message should be acknowledged the same way, i.e., using an empty `ACK` or a piggybacked `ACK` containing the response. In the latter case, `unicoap` should maintain a copy of the piggybacked `ACK`.

Timers are also required for several purposes, including the maximum response wait time, the interval for the next block request in a block-wise transfer, retransmissions, and specific durations for deduplication.

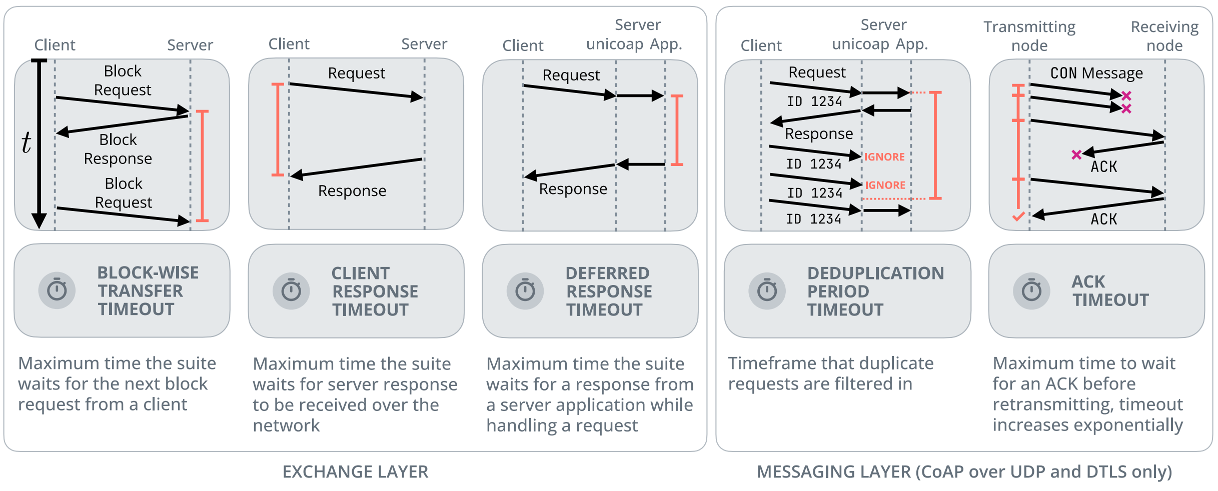

Client
Server
Block Request
t
Block Response
Block Request
BLOCK-WISE TRANSFER TIMEOUT
Maximum time the suite waits for the next block request from a client
Client
Server
Request
Response
CLIENT RESPONSE TIMEOUT
Maximum time the suite waits for server response to be received over the network
Client
Server
unicoap
App.
Request
Response
DEFERRED RESPONSE TIMEOUT
Maximum time the suite waits for a response from a server application while handling a request
Client
Server
unicoap
App.
Request
ID 1234
Response
ID 1234
IGNORE
ID 1234
IGNORE
ID 1234
DEDUPLICATION PERIOD TIMEOUT
Timeframe that duplicate requests are filtered in
Transmitting node
Receiving node
CON Message
ACK
ACK
ACK TIMEOUT
Maximum time to wait for an ACK before retransmitting, timeout increases exponentially
EXCHANGE LAYER
MESSAGING LAYER (CoAP over UDP and DTLS only)

Figure 16: Timeouts

As demonstrated by Figure 16, exchange-layer information is tied to lifetimes different than those of messaging-layer state. For instance, the client response timeout applies to responses in a client exchange while the `ACK` timeout runs in parallel and pertains to every transmission. In a block-wise transfer on a server, the block-wise transfer timeout for the next request may be active concurrently with the `ACK` timeout for the response that was sent last, provided it was confirmable. The deduplication period may even span across multiple exchanges, depending on its configured timeout value. As a result, state for the exchange and messaging layer is saved separately, as shown in Figure 17 and Figure 18. Transmission data must also be viewed as being detached from deduplication data since transmissions are short-lived whereas a duplicate filter covers multiple requests that are received.

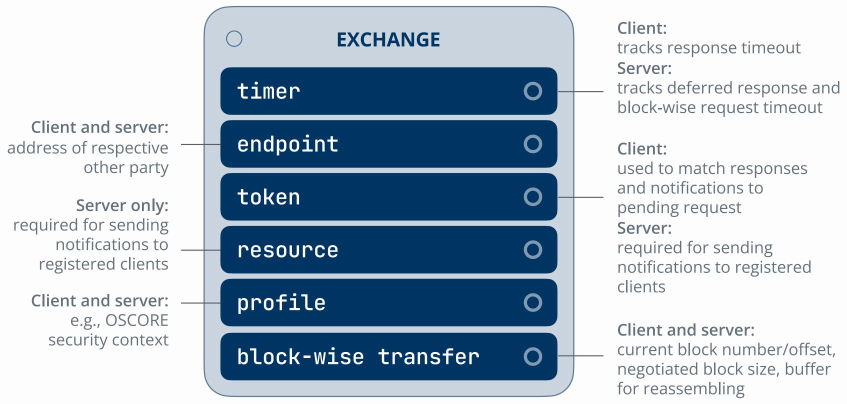

EXCHANGE
timer
endpoint
token
resource
profile
block-wise transfer
Client: tracks response timeout
Server: tracks deferred response and block-wise request timeout
Client and server: address of respective other party
Client: used to match responses and notifications to pending request
Server: required for sending notifications to registered clients
Server only: required for sending notifications to registered clients
Client and server: e.g., OSCORE security context
Client and server: current block number/offset, negotiated block size, buffer for reassembling

Figure 17: Exchange-layer state

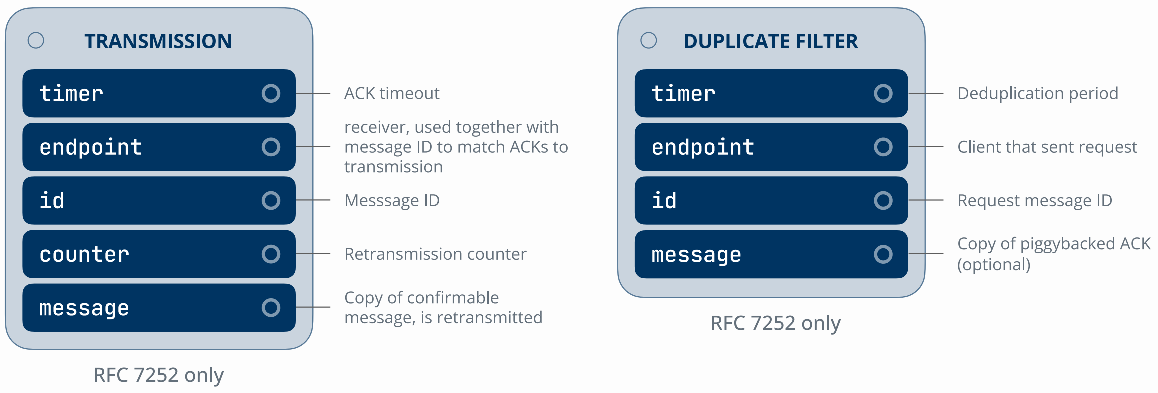


Figure 18: Messaging-layer state for CoAP over UDP and DTLS (RFC 7252)

3.5.1 Piggybacked and Separate Responses

The CoAP over UDP and DTLS messaging layer permits two styles of responding to confirmable requests, depicted in Figure 19. Provided the application responds immediately before returning from the resource handler, a piggybacked response can be sent, reducing airtime and I/O operations. Sending an empty ACK first prompts the client to stop retransmitting while the server is busy processing the request. A tradeoff decision must be made between the two options.

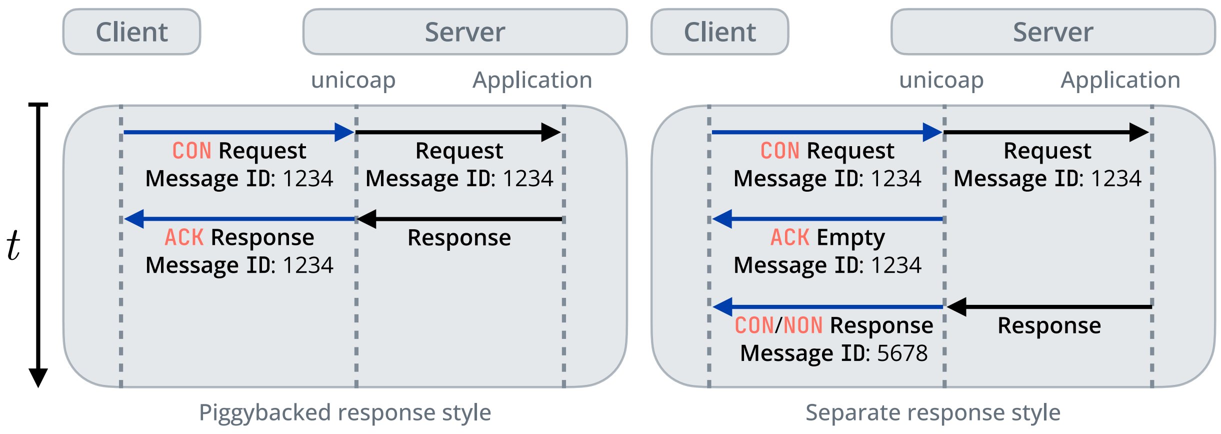


Figure 19: Piggybacked and separate response in RFC 7252 messaging

In cases where the application indicates processing will require more time by invoking `defer_response`, the messaging layer must decide which response style to choose. How this decision is made depends on software behavior exhibited when a retransmitted request arrives. On one hand, `unicoap` could ignore retransmissions and instead rely on a timer to determine if a piggybacked response can still be sent. Provided the response is ready before the automatic `ACK` timeout elapses, `unicoap` can package the response in a piggybacked acknowledgement. If the timer fires prior to that, an empty acknowledgement is transmitted, preventing a piggybacked response from being sent later. These cases are illustrated in Figure 20.

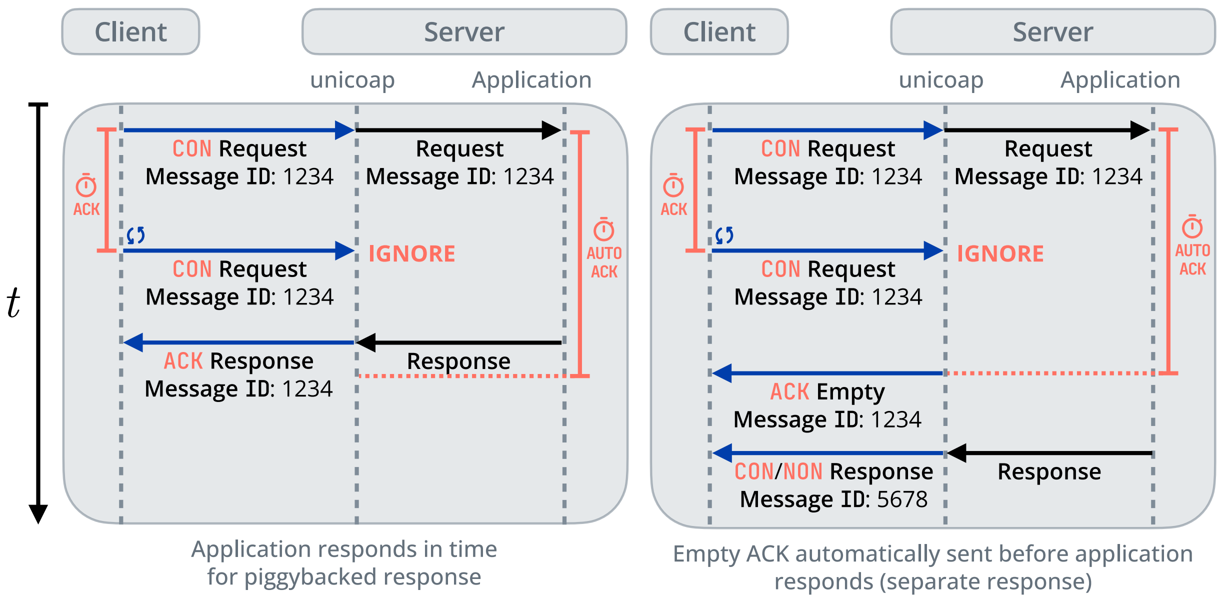


Figure 20: Message sequence of responses for ignoring retransmissions

Given a retransmission, an implementation may alternatively engage in directly acknowledging it. The approach in Figure 21 would cause the client to stop retransmitting early, but would also prevent `unicoap` itself from using a single piggybacked `ACK`.

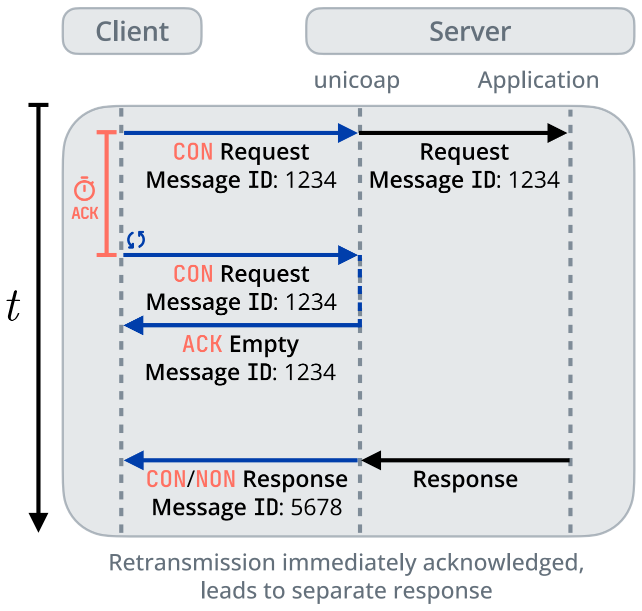


Figure 21: Message sequence of response for acknowledging retransmissions

After the first retransmission, the server can afford to take approximately twice as long to prepare the response due to the exponential back-of mechanism in place at the client. Directly acknowledging the duplicate request in this second `ACK` timeout period would yield no significant advantage compared to acknowledging it close to the end of that period. In fact, waiting for the application can enable the messaging layer to still use a piggybacked response. Additionally, the timeout value can be configured by the application. Consequently, we choose to ignore retransmissions in favor of a timer controlling the response style.

4 Implementation

`unicoap` is implemented as a C library in RIOT. `unicoap` builds upon the foundational principles of GCoAP, though it evolved into nearly a complete rewrite. `unicoap` unifies nanoCoAP, nanoCoAP Sock, and GCoAP. It addresses all key components of the design outlined in the previous chapter, with a focus on extensibility and convenience. The C implementation aims to reduce boilerplate code and to streamline development using beginner-friendly APIs. `unicoap` adheres to a philosophy that minimizes demands on the application while maximizing the benefits and functionality it provides in return. We expose as much of the functionality used internally as possible to reduce redundant implementation of features. Similarly, any message propagated to applications is accompanied by auxiliary data that was generated while processing it.

This chapter is structured as follows. First, we expound general approaches and constraints of the implementation, aligning `unicoap` with existing libraries in RIOT. In the first section, we detail the implementation of our high-level interface. In that section, we also derive a message structure and a data structure for storing options. Sections 4.2 and 4.3 introduce the client and server API. The subsequent section describes our approach to state management, focusing on exchange-layer data, block-wise transfers, and timers. Thereafter, we explain how communication between layers is facilitated and outline the set of functions exposed by each layer. We also explain the approach to modular drivers in RIOT and additionally expand on the implementation of the CoAP-over-UDP and DTLS drivers. Ultimately, we present several features aiding in the development of `unicoap` applications and further drivers, including a CoAP client shell.

Like in GCoAP, processing inbound messages is implemented in an asynchronous fashion. The respective functionality runs on a background thread dedicated to processing state. Jobs executed on this thread, hereinafter called the *`unicoap` thread*, are coordinated by an event queue. `unicoap` maintains an `event_queue_t` structure which functions like a FIFO of events that are posted and then synchronously executed on the thread [19]. Callbacks for messages delivered over the network are executed on the `unicoap` thread.

We also leverage the compile-time configuration mechanism employed widely throughout RIOT. Timing constants and buffer sizes can be adjusted by setting macro values. This approach is also applied to state management due to the lack of a heap. RIOT relies on static memory allocations, requiring fixed-size arrays of state objects. The size of these state arrays can be configured by modifying configuration macros. Modularization is facilitated through the RIOT build system. Advanced features can be conditionally included in the application binary by using submodules. Furthermore, our implementation has test coverage for block-wise transfer functions, options, the parser, and URI support.

4.1 Message

In `unicoap`, messages are represented using a generic `unicoap_message_t` message structure. This approach avoids casting between dedicated request, response, and signaling message types. `unicoap` ensures type-safe access to the request method, status code, or signal number using dedicated enumerations, as illustrated in Listing 4. `unicoap` also defines several message initializers to allow message creation in a single line of code. Listing 5 exemplarily shows convenience initializers for a request message.

```c
typedef struct {
  unicoap_options_t* options;
  uint8_t* payload;
  size_t payload_size;
  union {
    uint8_t code;
    unicoap_method_t method;
    unicoap_status_t status;
    unicoap_signal_t signal;
  };
} unicoap_message_t;
```

Listing 4: Message structure and code accessors

```c
static inline void unicoap_request_init_string(
  unicoap_message_t* request, unicoap_method_t method,
  const char* payload);

static inline void unicoap_request_init_string_options(
  unicoap_message_t* request, unicoap_method_t method,
  const char* payload, unicoap_options_t* options);
```

Listing 5: Convenience initializers for a request message

```c
typedef struct {
  uint8_t* token;
  uint8_t token_length : UNICOAP_TOKEN_LENGTH_FIXED_WIDTH;

  union {
    struct {
        uint16_t id;
        unicoap_message_type_t type : UNICOAP_MESSAGE_TYPE_FIXED_WIDTH;
    } rfc7252;

    // ...
  };
} unicoap_message_properties_t;
```

Listing 6: Message properties

The token of a message is hidden from the application by default, as are the message type and the ID. `unicoap` stores these properties in a separate `unicoap_message_properties_t` structure depicted in Listing 6. Transport-specific PDU properties are stored in a substructure inside a union which could accommodate further transport-specific properties of other CoAP combinations. The `union` ensures extensibility for different versions of properties, e.g., CoAP over TCP does not use a message type and ID, but CoAP over GATT does, yet in a different respresentation.

4.1.1 Parsing

To help users parse messages manually, `unicoap` exposes parser functionality. Each message parser adheres to the function prototype shown in Listing 7. Parsers implement logic for reading the PDU header, i.e., message properties and the code. Parsing options is shared functionality, hence all parsers will eventually invoke the function in Listing 8.

```c
typedef ssize_t (* unicoap_parser_t)(
  const uint8_t* pdu, size_t size,
  unicoap_message_t* message, unicoap_message_properties_t* properties
);
```

Listing 7: Parser function prototype

```
ssize_t unicoap_parse_options_and_payload(
  uint8_t* cursor, const uint8_t* end,
  unicoap_message_t* message);
```

Listing 8: Common payload and options parser
When calling this function, the `cursor` points to the first byte after the header, i.e., the options or payload part. The `end` argument is used to detect out-of-bounds errors. The `message` argument points to an allocated `unicoap_message_t` structure whose `payload`, `payload_size`, and `options` will be set.

Serializing a message works by first invoking a header builder, i.e., a function that writes the transport-specific header into a buffer. To embrace the capability of the socket API in RIOT accepting chunked data, `unicoap` does not write the PDU contiguously into a buffer. Instead, we leverage the `iolist_t` type which stores a list of buffers the socket API can use. Normally, when building a contiguous PDU buffer, the payload and options parts would need to be copied into a larger buffer, which may then in turn be copied again into another buffer, depending on the network backend. To prevent this duplicate copy operation, `unicoap` directly passes a list consisting of the header, the payload, and the options buffer to the socket API. This enables the network backend to directly copy the contents of each of these buffers into a larger buffer, if necessary. Listing 9 shows the function populating a buffer with the RFC 7252 PDU header and the common builder function shared across PDU formats. This design separates the serialization of the header from the remainder of the PDU.

```
ssize_t unicoap_header_build_rfc7252(
  uint8_t* header, size_t capacity,
  const unicoap_message_t* message,
  const unicoap_message_properties_t* properties);
  unicoap_message_t* message);

int unicoap_pdu_build_iolist(
  uint8_t* header, size_t header_size,
  unicoap_message_t* message,
  iolist_t* iolists);
```

Listing 9: RFC 7252 header serializer and common PDU builder

4.1.2 Option Handling

The aim of this section is to deduce a design for the options data structure and the mechanism used for manipulating options. A key obstacle for the options API lies in the mandated order of options. Currently, developers relying on nanoCoAP and GCoAP are required to organize the control flow of their applications based on the order of options. The goal of the options API is to enable setting and inserting options in any application-defined order while still guaranteeing options in the final PDU adhere to the order dictated by their number. Designing a mechanism that can tolerate any calling order is complicated by the potential absence of a heap.

Potential implementations. To meet the beforementioned requirements, we considered three potential general approaches, including

1. storing options as designated, strongly typed members,
2. operating on options serialized according to [6], and
3. maintaining a list of option numbers and their values.

The first technique defers the serializing work to the point of building PDU data. An implementation would only compute the delta-encoding and variable integers according to the options format at times the raw PDU data is needed. Yet, designated members would induce a memory overhead as the spectrum of all options specified by the IANA is unlikely to be present at the same time. Moreover, relying on strongly typed members removes the possibility of setting custom options.

With the second approach, an application would need to iterate and parse the options data each time a specific option is requested, which would entail a performance overhead that grows linearly with the number of options.

The third technique enables faster lookup times and is not bound to predefined types like approach 1 is. This technique allows for multiple implementations. As a starting point, each list entry would hold the option number and a value. As a convenience measure, various value types may be stored in a union or as an opaque type if the language environment retains dynamic type information. In the absence of such features in C, the design is limited to fixed-size buffers for potentially variable-length values. However, this imposes the risk of utilizing memory inefficiently, especially in light of heterogeneous values, options with small integer values, and string options like `Uri-Path` and `Uri-Host`. Consequently, storing values in a *backing storage buffer* is more suitable. In this case, each entry would reference the corresponding value location in the storage buffer, i.e., using a pointer or offset. We concluded a combination of approaches 2 and 3 offers the best tradeoff between flexibility and performance. The combination of an option numbers list and a backing buffer is illustrated in Figure 22.

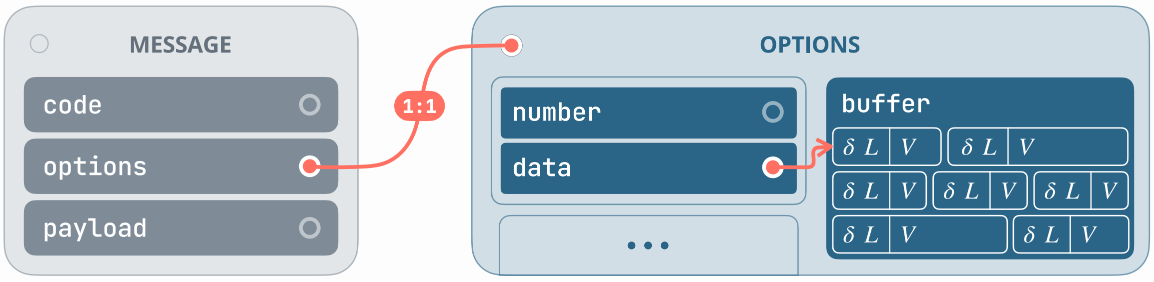


Figure 22: Conceptual illustration of storing options in a backing buffer
In this graphic, options are depicted by their number delta (δ), value length (L), and their value (V).

The storage buffer either contains serialized options with deltas and size indicators or option values only. A buffer of serialized options is advantageous to reduce copy operations. In contrast, storing just option values leads to a simpler implementation but requires copying values when serializing options. When inserting options, both storage buffer approaches necessitate moving trailing options or values. As directly using serialized options avoids the additional copying overhead at the end, we chose this version of the storage buffer. Hence, the options structure in `unicoap` encompasses a list of option entries that each hold the option number and a reference to the option in the storage buffer. This approach is very similar to the structure used in nanoCoAP. The final implementation is depicted in Listing 10.

`unicoap` tolerates any calling order of option manipulation methods using the algorithms in Appendix B. Since inserting options in between adjacent options is more complex, the best performance can still be achieved by adding options in the order dictated by their number.

```
typedef struct {
  uint8_t* data;
  size_t size;
  unicoap_option_number_t number;
} unicoap_option_entry_t;

typedef struct {
    size_t storage_size;
    size_t storage_capacity;
    unicoap_option_entry_t entries[CONFIG_UNICOAP_OPTIONS_MAX];
    size_t entry_count;
} unicoap_options_t;
```

Listing 10: Options structure
The `storage_size` and `storage_capacity` members track the total number of bytes in the options buffer and bytes occupied by options. The `data` pointer references the start of the delta-encoded option in the storage buffer.

4.2 Client

This section outlines the client implementation, with particular attention to the approach to concurrency, support for URIs and endpoint addresses, and OSCORE. Listing 11 shows a sample `GET` request carried out using `unicoap`.

```c
static int handle_response(
  const unicoap_message_t* response, const unicoap_aux_t* aux,
  int error, void* arg) {

  printf("status=%s (%zu bytes)\n",
    unicoap_label_from_status(response->status), response->payload_size);

  return 0;
}

static int main(void) {
  unicoap_message_t request = { .method = UNICOAP_METHOD_GET };
  unicoap_destination_t destination =
      unicoap_destination_uri("coap://example.org:10123/led/color?a=1&b=2");
  /* or
  sock_udp_ep_t udp_ep = { ... };
  unicoap_destination_t destination =
    unicoap_destination_endpoint(&udp_ep);
  ... */

  int res = unicoap_send_request_async(&request, &destination,
      &handle_response, UNICOAP_CLIENT_FLAG_RELIABLE, NULL);

  /* or
  int res = unicoap_send_request_sync(&request, &destination,
      handle_response, UNICOAP_CLIENT_FLAG_RELIABLE, NULL); */

  // ...
  return 0;
}
```

Listing 11: Sample asynchronous `GET` request using a URI
Using the `UNICOAP_CLIENT_FLAG_RELIABLE` instructs the stack to use a `CON` message over UDP/DTLS.

To provide ease of use in synchronous and asynchronous scenarios, we offer several versions of our request method. The asynchronous client API is implemented using callbacks. For synchronous API usage, we define a blocking, callback-based, and a blocking, copying version of the request method. Due to the stack allocation of the response message and buffers, synchronous callbacks are advantageous as they do not necessitate creating copies of buffers. Access to result data is limited to logic within the callback. Provided response data must outlive the scope of a callback, copying is imperative. In these cases, we offer a blocking request method.

The most direct way of specifying the request destination is by supplying an endpoint structure such as `sock_udp_ep_t`. In light of future additions to `unicoap`, different endpoints like `sock_tcp_ep_t` or even MAC addresses for CoAP over GATT must be supported. To account for the different representations, we introduce a `unicoap_proto_t` enumeration to identify each CoAP combination and the `unicoap_endpoint_t` in Listing 12 that functions as a generic CoAP endpoint. Internally, `unicoap` always uses this endpoint to address CoAP nodes.

However, there exist more elegant techniques for addressing a CoAP server or resource in an application. Developers may want to specify purely a URI or host. For instance, URIs additionally contain values for `Uri-Path`, `Uri-Host`, and `Uri-Query` options, which would otherwise need to be set manually. As a result, we define a `unicoap_destination_spec_t` container capable of representing either just a URI, just a host, an endpoint, or a CRI in the future. Internally, `unicoap` resolves destination specifications to an endpoint object. To provide access to the resolved endpoint even when a URI has been supplied, we pass an instance of the auxiliary information container depicted in Listing 14 as a callback argument.

```c
typedef struct {
  unicoap_proto_t proto;

  union {
      struct _sock_tl_ep _tl_ep;
      sock_udp_ep_t udp_ep;
      sock_udp_ep_t dtls_ep;
      sock_tcp_ep_t tcp_ep;

     // MARK: unicoap_driver_extension_point
  };
} unicoap_endpoint_t;
```

Listing 12: Endpoint capable of representing a multitude of transport addresses

`unicoap_proto_t` is an `enum` of all possible CoAP combinations and serves for determining which endpoint version to read from the `union`. Both `sock_udp_ep_t` and `sock_tcp_ep_t` are type aliases for `struct _sock_tl_ep`. As union members are guaranteed to start at offset zero, the `_tl_ep` aids in populating the endpoint when parsing a URI. Symbols used internally are prepended with a leading underscore.

Listing 13 exhibits the set of request functions exposed by `unicoap`. The synchronous versions internally leverage the asynchronous request method and block the caller using a semaphore.

```
int unicoap_send_request_async(
  unicoap_message_t* request,
  unicoap_destination_t* destination_spec,
  unicoap_response_callback_t callback, void* callback_arg,
  unicoap_client_flags_t flags,
  unicoap_profile_t* profile);

int unicoap_send_request_sync(
  unicoap_message_t* request,
  unicoap_destination_t* destination_spec,
  unicoap_response_callback_t callback, void* callback_arg,
  unicoap_client_flags_t flags,
  unicoap_profile_t* profile);

int unicoap_send_request_sync_copy(
  unicoap_message_t* request,
  unicoap_destination_t* destination_spec,
  unicoap_message_t* response,
  uint8_t* buffer, size_t buffer_capacity,
  unicoap_client_flags_t flags,
  unicoap_profile_t* profile,
  unicoap_aux_t* aux);
```

Listing 13: Synchronous and asynchronous request functions offered by `unicoap`

To enable support for OSCORE while leaving room for future CoAP extensions akin to OSCORE, we define `unicoap_profile_t` in Listing 15. It follows the same tagged `union` approach we also use for `unicoap_endpoint_t` and `unicoap_destination_t`. Moreover, wrapping the security context in a separate structure avoids creating more request functions in the future.

```
typedef struct {
  const unicoap_endpoint_t* remote;
  const unicoap_endpoint_t* local;
  const unicoap_message_properties_t* properties;
  unicoap_profile_t* profile;
} unicoap_aux_t;
```

Listing 14: Auxiliary information structure

```
typedef struct {
  unicoap_profile_type_t type;
  union {
      void* oscore_security_context;
  };
} unicoap_profile_t;
```

Listing 15: Profile structure for OSCORE

unicoap_profile_type_t is an enum. The oscore_security_context preliminarily bears a type of void* until OSCORE support has been integrated into unicoap.

4.3 Server

The server API in unicoap is derived from GCoAP and has been extended to be more flexible. Listing 16 shows a sample server application leveraging unicoap.

```
UNICOAP_RESOURCE(my_resource) {
  .path = "/led/color",
  .handler = handle_color_request,
  .methods = UNICOAP_METHODS(UNICOAP_METHOD_GET),
};

static int handle_color_request(
  unicoap_message_t* message, const unicoap_aux_t* aux,
  unicoap_request_context_t* context, void* arg) {

  unicoap_response_init_string(&message, UNICOAP_STATUS_CONTENT, "rebeccapurple");
  return unicoap_send_response(&message, context);
}
```

Listing 16: Sample resource and request handler

4.3.1 Resource Definitions

Resources may be defined in two ways. We implement the same static resource definition technique from nanoCoAP and GCoAP that uses cross-file arrays (XFA). Cross-file arrays are stored in a dedicated section in the binary and read by unicoap on startup. Similar to the existing CoAP libraries in RIOT, we define a UNICOAP_RESOURCE macro encapsulating the cross-file array definition. The resource structure and usage of XFA resource definitions can be seen in Listing 17 and Listing 18, respectively. Resources can be grouped into listeners according to the design introduced in Chapter 3. Moreover, listeners can be added or removed at runtime using the functions in Listing 19. Upon initialization, XFA resources defined using the `UNICOAP_RESOURCE` macro are automatically read by `unicoap` and added to a default listener solely responsible for `XFA` resources.

```
typedef struct {
  const char* path;
  unicoap_resource_handler_t handler;
  void* handler_arg;
  unicoap_resource_flags_t flags;
  unicoap_methods_t methods;
  unicoap_proto_set_t protocols;
} unicoap_resource_t;
```

Listing 17: Resource structure
The `handler` property points to the function invoked for each request directed toward the given `path`. The `handler_arg` is an optional, opaque argument passed to the handler allowing the application to provide additional context. The `methods` and `protocols` members contain bitfields that serve to restrict the resource to certain CoAP method codes and transport protocols.

Accepted request methods in a resource definition previously required additional flags in nanoCoAP, e.g., `COAP_GET`, while regular methods were defined as `enum` cases such as `COAP_METHOD_GET`. This ambiguity has been removed in `unicoap`. We define a single `enum` encompassing all request methods, with each method directly corresponding to the code value assigned by IANA. As these values are not powers of two, we introduce a `UNICOAP_METHODS` macro that converts methods to flags, creating a bitfield. We consolidate this feature into an internal `UNICOAP_BITFIELD` macro and can thus apply the same technique to the list of eligible protocols using a `UNICOAP_PROTOCOLS` macro. Figure 23 demonstrates the expansion of a `UNICOAP_METHODS` invocation.

```c
UNICOAP_RESOURCE(my_resource) {
  .path = "/led/color",
  .handler = handle_color_request,
  .methods = UNICOAP_METHODS(UNICOAP_METHOD_GET, UNICOAP_METHOD_PUT),
  .protocols = UNICOAP_PROTOCOLS(UNICOAP_PROTO_DTLS),
  .flags =
    UNICOAP_RESOURCE_FLAG_MATCH_SUBTREE |
    UNICOAP_RESOURCE_FLAG_DEDUPLICATE |
    UNICOAP_RESOURCE_FLAG_RELIABLE,
};
```

Listing 18: Sample resource defined using the `UNICOAP_RESOURCE` macro (`XFA`)

The behavior of a resource can be customized with flags.

`UNICOAP_RESOURCE_FLAG_MATCH_SUBTREE`: The given code snippet defines a resource reachable under `/led/color` and subpaths. For instance, the user-supplied callback `handle_color_request` would also be invoked for `/led/color/rgb` or `/led/color/hsl`.

`UNICOAP_RESOURCE_FLAG_DEDUPLICATE`: Requests are automatically deduplicated.

`UNICOAP_RESOURCE_FLAG_RELIABLE`: Responses sent from the handler will use a message type of `CON` when communication takes place over DTLS. Note that the resource name (`my_resource`) is solely necessary for technical reasons involving the implementation of cross-file arrays.

```c
typedef struct {
  const unicoap_resource_t* resources;
  size_t resource_count;
  unicoap_request_matcher_t request_matcher;
  unicoap_link_encoder_t link_encoder;
  unicoap_listener_t* next;
  unicoap_proto_set_t protocols;
} unicoap_listener_t;

void unicoap_listener_register(unicoap_listener_t* listener);
void unicoap_listener_deregister(unicoap_listener_t* listener);
```

Listing 19: Listener structure and registration functions

Listeners form a linked list with storing listeners being facilitated by the application. The registration functions modify this linked list. The `handler_arg` is an optional, opaque argument passed to the handler.

```
UNICOAP_METHODS(UNICOAP_METHOD_GET, UNICOAP_METHOD_PUT, UNICOAP_METHOD_POST)
↓
UNICOAP_BITFIELD(UNICOAP_METHOD_GET, UNICOAP_METHOD_PUT, UNICOAP_METHOD_POST)
↓
( (1 << (UNICOAP_METHOD_GET)) | (1 << (UNICOAP_METHOD_PUT)) | (1 << (UNICOAP_METHOD_POST)) )
↓
( (1 << (1)) | (1 << (3)) ) | ( (1 << (2)) )
↓
0x0E
```

Figure 23: Sample macro expansion of a `UNICOAP_METHODS` invocation
`UNICOAP_METHODS` is variadic in the number of macro arguments.

4.3.2 Request Handlers

Request handlers are invoked asynchronously on the `unicoap` thread. Into each request handler, we pass the request, auxiliary information, and a context pointer. Auxiliary data is carried through the `unicoap_aux_t` structure. The request context serves as a handle to the application to respond to the given request. The handler can respond by calling `unicoap_send_response`, as shown in Listing 16. Alternatively, we support directly returning a CoAP status code when no payload is supposed to be attached to the response, removing boilerplate. In this case, the request context must not be acted on, as illustrated in Listing 20.

```c
static int handle_color_request(
  unicoap_message_t* request, const unicoap_aux_t* aux,
  unicoap_request_context_t* context, void* arg) {

  return UNICOAP_STATUS_NOT_IMPLEMENTED;
}
```

Listing 20: Sample request handler returning a status code

Request context. Conceptually, calling `unicoap_send_response` consumes the context. C does not support explicitly ending the lifetime of an object, and hence, we mark the context as being redeemed, preventing further calls to `unicoap_send_response`. Internal data is referenced by request context members, permitting future changes to the number and layout of internal objects since they are not reflected in the handler prototype. For instance, the request token is passed to the response function using this technique. Respective members of the request context are marked private, i.e., begin with an underscore, and have been documented to be private. While a type of `void*` would suffice for the request context to merely channel internal data to the response function, we use it to convey additional data that might be relevant to processing the request such as the resource. For instance, a server might use the same request handler for several resources, which would otherwise force the server to reconstruct URI paths from options. `unicoap_aux_t` is a shared interface between the client and server implementation, and is thus not well-suited for storing data only relevant to servers.

Deferred responses. Some programming languages permit suspending execution, e.g., using an `await` keyword. Without language support for concurrency, management of asynchronous tasks, and suspension points in C, delaying the return to the caller requires blocking the handler. This would also block the internal processing queue, potentially leading to buffers for inbound packets filling up and eventually packet loss. In RFC 7252 transmissions, the client will also continue retransmitting, as the server is unable to acknowledge the request. Consequently, `unicoap` needs to provide a way for the application to indicate processing is going to take longer. `unicoap` offers a `unicoap_defer_response` method, signaling to the suite that processing will take longer. This technique lets servers *defer* the response, allowing them to respond asynchronously after the handler has already returned. Additionally, it ensures that `unicoap` responds eventually with a server error response after a configurable timeout. We define alternative versions of the `unicoap_defer_response` function, letting users customize the timeout and timeout response message. Like `unicoap_send_response`, the `unicoap_defer_response` function consumes the request context, while initializing a new, deferred response context. This is necessary to copy stack-allocated data in the caller frame that would otherwise not be available when the application intends to respond asynchronously. This ensures developers explicitly need to opt-in to the copy overhead. Furthermore, applications will likely limit the number of outstanding deferred requests and already have a system for tracking operations. Thus, we place the responsibility for storing contexts on the application. Listing 21 demonstrates the process of deferring a response to account for an asynchronous operation.

```
static bool reading_color = false;
static unicoap_request_deferred_context_t deferred_context; // ①

static int handle_color_request(
  unicoap_message_t* request, const unicoap_aux_t* aux,
  unicoap_request_context_t* context, void* arg) {

  if (reading_color) {
    return UNICOAP_STATUS_SERVICE_UNAVAILABLE;
  }

  int res = unicoap_defer_response_with_timeout(
    context, &deferred_context, 5000 /* 5 seconds */); // ②

  if (res < 0) {
    printf("error: code=%i reason='%s\n'", res, strerror(-res));
  }

  reading_color = true;
  start_reading_color_with_completion_handler(on_color_ready); // ③
  return res;
}

int on_color_ready(const char* color) { // ④
  reading_color = false;

  int res = 0;
  unicoap_message_t response = { 0 };
  unicoap_response_init_string(&response, UNICOAP_STATUS_CONTENT, color);

  // ⑤
  if ((res = unicoap_send_response_deferred(&response, &deferred_context)) < 0) {
    printf("error: code=%i reason='%s'\n", res, strerror(-res));
  }

  return res;
}
```

Listing 21: Sample code of a resource handler deferring a response

The application provides storage for deferred request contexts based on knowledge of the duration and number of permissible operations ①. It also implements fallback behavior when the number of operations exceeds the configured limit. Upon being prompted to handle a request that is known to entail an asynchronous (or expensive) operation, the handler tells `unicoap` it is going to take longer and sets a maximum response time ②. The asynchronous operation is then started ③. Once the operation has completed, the application sends a callback by preparing a response message ④ and ultimately calling `unicoap_send_response_deferred` ⑤. `handle_color_request` is executed on the `unicoap` thread to which access is synchronized.

4.4 State Management

In this section, we present the implementation of state management in `unicoap` based on the design introduced in Section 3.5.

To track state in exchanges, `unicoap` maintains a pool of *memos*. The information contained in a memo, shown in Listing 22, is bound to the lifetime of the corresponding exchange, such as of a block-wise transfer. Memos can be used by the server or client implementation, as indicated by the `kind` property (`UNICOAP_MEMO_CLIENT_EXCHANGE`, `UNICOAP_MEMO_SERVER_EXCHANGE`). The memo stores exchange-layer information, such as the token, callback, and flags in client exchanges. To map inbound messages to an existing memo, we retain an endpoint in each memo. Exchange-layer timeouts detailed in Section 3.5 are also tracked by each memo, e.g., the response timeout on clients, as well as the block-wise request timeout and deferred response timeout on servers.

To react to errors that occur at the exchange layer, we maintain a reference to potential messaging state in each memo. For instance, after the response timeout has elapsed, state related to retransmissions in the messaging layer must be released. Messaging state is not directly stored in the memo but is instead only referenced since the size of messaging state per sent message varies. For instance, considering a `unicoap` application that primarily engages in CoAP over TCP interchanges, and rarely uses CoAP over UDP, storing the rather large UDP messaging state compared to TCP in the memo would create an unnecessary overhead. Hence, drivers accommodate their messaging state that can, in turn, be referenced by each memo. As a result of this separation, the number of state objects per CoAP combination can be configured according to the estimated usage, thereby reducing memory needs.

```
typedef struct {
  struct {
      unicoap_scheduled_event_t timeout;
      union {
          unicoap_client_exchange_t client;
          unicoap_server_exchange_t server;
      };
      unicoap_blockwise_transfer_t* blockwise_transfer;
  } exchange;

  unicoap_endpoint_t endpoint;
  unicoap_memo_kind_t kind : UNICOAP_MEMO_KIND_FIXED_WIDTH;

  union {
   #if IS_USED(MODULE_UNICOAP_DRIVER_RFC7252_COMMON)
     void* rfc7252;
   #endif
  } messaging;
} unicoap_memo_t;
```

Listing 22: Memo structure
The `IS_USED` macro checks if the given module is used when building the application [20]. `MODULE_UNICOAP_DRIVER_RFC7252_COMMON` is the auto-generated name for the internal `unicoap_driver_rfc7252_common` submodule that implements RFC 7252 messaging.

4.4.1 Resource Observation

We separate state related to resource observation from the memo structure to further reduce our memory footprint. First, notification registrations are not inherently bound to a lifetime. Thus, the timer in each memo serves no purpose in resource observation. Secondly, the token varies in length. The `unicoap` client only needs to keep track of the self-generated 2-byte token (*internal token length*) while a server must be prepared to retain a token up to 8 bytes in length (*external token length*) in notification registrations. Finally, the same client might subscribe to several resources on the same server in parallel. Repeatedly storing the client endpoint by value would entail a significant memory overhead. As a result, we provide a structure disconnected from memos for observation state on `unicoap` servers. As in GCoAP, the corresponding client endpoints are stored in a separate, statically allocated array. Registrations, shown in Listing 23, reference the client endpoint, saving memory if a client subscribes to multiple resources. Client endpoints themselves are stored in a separate endpoint array.

```c
typedef struct {
  unicoap_endpoint_t* client;
  const unicoap_resource_t* resource;
  unicoap_profile_t* profile;
  uint8_t token[UNICOAP_EXTERNAL_TOKEN_LENGTH_MAX];
  uint16_t last_notification_id;
  uint8_t token_length : UNICOAP_TOKEN_LENGTH_FIXED_WIDTH;
} unicoap_observe_registration_t;
```

Listing 23: Resource observation registration structure

`unicoap` defines an internal interface for registering clients for future resource notifications. Whenever the `unicoap` server encounters an `Observe` option, it will call `unicoap_observe_register` or `unicoap_observe_deregister`. We also support terminating a notification subscription through the message ID in RFC 7252 transmissions as a special case using `unicoap_observe_deregister_by_id`. This API is needed to act on `RST` messages elicited by a client in response to a notification when the subscription has been intentionally forgotten or state has been lost. The new API lifts the restriction present in GCoAP where only a single registration per resource was possible, though this legacy behavior can still be enabled in `unicoap` by modifying the compile-time configuration.

4.4.2 Block-Wise Transfer

As outlined in Section 3.3, `unicoap` supports automatic block-wise transfers in the client and server API. For instance, developers can specify `UNICOAP_CLIENT_FLAG_REASSEMBLE` when initiating a client request, or `UNICOAP_SERVER_FLAG_SLICE` when defining a resource. Normally, only two flags for slicing and reassembling would be necessary. However, to provide further means for minimizing memory consumption, we introduce `SLICE_NO_COPY` flags. A client application instructing `unicoap` to slice the given request message can pass `UNICOAP_CLIENT_FLAG_SLICE_NO_COPY`, indicating that the payload buffer pointer is guaranteed to stay valid throughout the transfer. Without this flag, `unicoap` must assume the buffer stems from a stack allocation and must thus be copied into an internal buffer. This also applies to servers wanting to send a sliced response. The `UNICOAP_SERVER_FLAG_SLICE_NO_COPY` flag might be especially useful if the payload is statically known or maintained as global state in the application. The `SLICE_NO_COPY` flags help avoid the copying overhead.

Internally, each managed block-wise transfer is tracked using a dedicated structure, shown in Figure 24. It stores the message that is being sliced or reassembled and furthermore optionally references a buffer for the reconstructed representation. In the unified CoAP suite, this buffer is detached from the transfer structure, i.e., must be allocated separately. As unicoap is designed under the contraint of static memory allocation, it maintains a pool for block-wise transfer structures and representation buffers, respectively. These pools reside in the RAM. The choice of separating buffers from transfers allows developers to reduce the number of buffers available below the number of transfers, reducing the memory footprint of the suite. Provided, the application does not depend on unicoap reconstructing representations and only uses the SLICE_NO_COPY flags, it might even set the buffer pool size to zero. To prevent collecting or reassembling blocks with misaligned formats or from different cached versions, the Content-Format and ETag of the first received message are saved and then compared against subsequent messages.

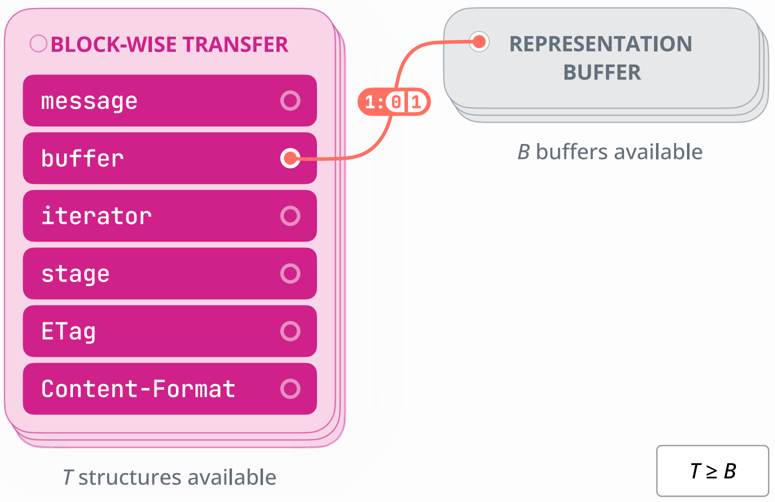


Figure 24: Block-wise transfer pool and representation buffer pool
A transfer can optionally have buffer from the representation buffer pool. The buffer pool maintains as many as or fewer buffers than transfers in the transfer pool.

We package automatic block-wise functionality in a unicoap_blockwise submodule, i.e., binary sizes and memory usage remain unaffected if block-wise support is not needed. Independently of the unicoap_blockwise module, we expose helper functions used by the automatic block-wise transfer implementation publicly. We provide a block-wise iterator structure alongside methods for each of the block-wise modes outlined in Section 3.3. The unified iterator in Listing 24 extends the `coap_block_slicer_t` from nanoCoAP to support collecting and reassembling blocks.

```
typedef struct {
  uint8_t* representation;
  size_t representation_size;
  size_t offset;
  unicoap_block_option_t block_option;
} unicoap_blockwise_iterator_t;
```

Listing 24: Block-wise iterator structure
unicoap_block_option_t is a type alias for uint32_t, the value type of the Block1 and Block2 options.

Once the iterator has been configured to a particular block-wise mode using `unicoap_blockwise_iterator_init`, one of the methods shown in Listing 25 can be used for slicing, collecting, or reassembling blocks. If a message is to be sliced, the initial representation must be supplied as an argument during initialization. Similarly, to reassemble blocks, a representation buffer is required when calling `unicoap_blockwise_iterator_init`. By passing a block size of `UNICOAP_BLOCK_SZX_1024_BERT` to the initialization function, unicoap will use the Block-wise Extension for Reliable Transports (BERT), i.e., fit multiple payload chunks of size 1024 into a message.

```
int unicoap_blockwise_collect_block2(unicoap_blockwise_iterator_t* collector,
  unicoap_block_option_t block, uint8_t* chunk, size_t chunk_size);

ssize_t unicoap_blockwise_slice_block2(unicoap_blockwise_iterator_t* slicer,
  unicoap_block_option_t block, uint8_t** chunk);

ssize_t unicoap_blockwise_slice_block1(unicoap_blockwise_iterator_t* slicer,
  unicoap_block_option_t block, uint8_t** chunk);

int unicoap_blockwise_collect_block1(unicoap_blockwise_iterator_t* collector,
  unicoap_block_option_t block, uint8_t* chunk, size_t chunk_size);
```

Listing 25: Block-wise helper functions
In this listing, chunk refers to the payload of the block message, i.e., a part of the representation. We chose 'chunk' over 'block' to avoid ambiguity with the Block1/Block2 options. The unicoap_blockwise_collect_* functions also perform block reassembly if the iterator has been configured accordingly during initialization.

4.4.3 Timers

Multiple timers are running in parallel in a single exchange, such as the retransmission timer and response timer. unicoap is capable of managing several exchanges concurrently, and thus also multiple timers. As the underlying timer mechanism, we rely on `ztimer_t`. On the unicoap thread, we use an `event_queue_t` to manage outstanding jobs and synchronize state access by scheduling events (`event_t`). Accordingly, we must implement a strategy to execute jobs on the queue, i.e., post an event on the queue after a specified amount of time. The `ztimer_t` and `event_t` definitions are shown in Listing 26, and the general sequence of steps is depicted by Figure 25. In the following, we analyze three distinct approaches for implementing timeouts. The goal is to formulate a strategy for timeouts, with the key incentive being a reduction in the implementation's memory footprint.

```c
typedef struct {
  ztimer_base_t base;
  ztimer_callback_t callback;
  void* arg;
} ztimer_t;
```

```c
typedef struct event {
  clist_node_t list_node;
  event_handler_t handler;
} event_t;
```

Listing 26: ztimer and event structure [21, 22]

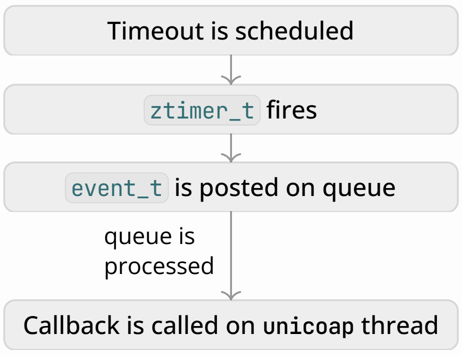

Timeout is scheduled
ztimer_t fires
event_t is posted on queue
queue is processed
Callback is called on unicoap thread

Figure 25: Steps from scheduling a timeout to executing the callback

Approach ①: `event_timeout_t` and `event_callback_t`. RIOT has an existing implementation of performing work on a queue after a timeout. By calling `event_timeout_set` on an `event_timeout_t`, an event can be scheduled on a given event queue. The event that is posted on the queue is wrapped in an `event_callback_t` structure. Listing 27 shows the definition of these structures.

```c
typedef struct {
  ztimer_clock_t* clock;
  ztimer_t timer;
  event_queue_t* queue;
  event_t* event;
} event_timeout_t;
```

```c
typedef struct {
  event_t super;
  void (*callback)(void*);
  void* arg;
} event_callback_t;
```

Listing 27: Event timeout and event callback structures [19]

By combining `event_timeout_t` and `event_callback_t`, several callbacks are invoked.

1. **Timer fires**: The internal `ztimer_t` callback calls `event_post` on the queue referenced in the `event_timeout_t` structure.

   ```
   static void _event_timeout_callback(void* arg) {
       event_timeout_t* event_timeout = (event_timeout_t*)arg;
       event_post(event_timeout->queue, event_timeout->event);
   }
   ```

2. **Queue processes event**: The `event_t` callback is executed after all events previously posted on the queue have been processed. The callback in `event_t` always receives a pointer to this event as its argument. This callback function is located in the `event_callback_t` implementation and executes the user-supplied callback with the customized argument.

   ```
   void _event_callback_handler(event_t* event) {
     event_callback_t* event_callback = (event_callback_t*)event;
     event_callback->callback(event_callback->arg);
   }
   ```

3. **Event callback executes actual callback**: The user-defined callback is executed on the queue.

There are several issues with this approach. On the one hand, `unicoap` uses the same clock and tick precision (`ZTIMER_MSEC`) on the same queue. This information is statically known. Hence, using an `event_timeout_t` structure for the timeout in each memo would entail redundant storage of the clock and queue, which would amount to the size of two words on most platforms. On the other hand, `unicoap` always acts on the enclosing struct when the timer fires, i.e., the the memo containing the timer. Storing an additional argument and callback as part of the event just to modify the argument is unnecessary. Instead of an `event_callback_t`, we can leverage the `container_of` macro to get the enclosing struct, i.e., the memo in this case.

Approach ②: Combination of `ztimer_t` and `event_t`. To eliminate the additional callback, a `ztimer_t` and `event_t` can be combined directly, as shown in Listing 28. The statically known queue and clock are omitted and the additional callback and argument from `event_callback_t` are elided as well.

```c
typedef struct {
  event_t super;
  ztimer_t ztimer;
} unicoap_scheduled_event_t;

static void _ztimer_callback(void* arg) {
  event_post(&_internal_queue, (event_t*)arg);
}

void unicoap_event_schedule(
  unicoap_scheduled_event_t* event, event_handler_t callback, uint32_t duration
) {
  event->ztimer.callback = _ztimer_callback;
  event->ztimer.arg = (void*)&event->super;
  event->super.handler = callback;
  ztimer_set(ZTIMER_MS, &event->ztimer, duration);
}
```

```c
typedef struct {
  unicoap_scheduled_event_t timeout;
  // ...
} unicoap_memo_t;

static void on_sample_timeout(event_t* e) {
  unicoap_memo_t* memo =
      container_of(e, unicoap_memo_t, timeout.super);
  // ...
}
```

Listing 28: Using a combination of `ztimer_t` and `event_t`
In this listing, `_internal_queue` denotes the `event_queue_t` running on the `unicoap` thread.

Approach 3: Using `evtimer_t`. The `evtimer_t` allows scheduling multiple timeouts with the same callback [23] which would be especially beneficial as there can be more than one timeout running per exchange, e.g., the retransmission timeout in parallel to the response timeout. However, in contrast to approach ①, `evtimer_t` does not support posting events on an event queue by default [23]. The `evtimer_event_t` shown in Listing 29 would need to be wrapped in a modified event structure. This is illustrated in Listing 30.

```c
typedef struct {
  ztimer_t timer;
  uint32_t base;
  evtimer_callback_t callback;
  evtimer_event_t* events;
} evtimer_t;
```

```c
typedef struct evtimer_event {
  struct evtimer_event* next;
  uint32_t offset;
} evtimer_event_t;
```

Listing 29: Event timer structures [23]

```c
typedef struct {
  evtimer_event_t evtimer_event;
  event_t queue_event;
} unicoap_scheduled_event_t;

static void _evtimer_callback(evtimer_event_t* evtimer_event) {
  unicoap_scheduled_event_t* e = (unicoap_scheduled_event_t*)evtimer_event;
  event_post(&_internal_queue, e->queue_event);
}
```

```c
typedef struct {
  evtimer_t timer;
  unicoap_scheduled_event_t timeout;
  // ...
} unicoap_memo_t;

static void on_sample_timeout(event_t* e) {
  unicoap_memo_t* memo =
      container_of(e, unicoap_memo_t, timeout.queue_event);
  // ...
}
```

Listing 30: Usage of modified `evtimer_event_t` to support event queues
In this listing, `_internal_queue` denotes the `event_queue_t` running on the `unicoap` thread.

Evaluating approaches. In Figure 26, we compare options ① to ③ with respect to required memory. We evaluate the size in RAM for different memory address sizes and for a single and two timeouts scheduled in parallel. Approach ① entails the largest memory consumption due to the clock, queue, and additional callback and argument. While approach ③ requires the least number of bytes for each additional scheduled timeout, this advantage is never displayed since no more than two timeouts run in parallel. Approach ② occupies at least 42 % fewer bytes than approach ① and between 20 % and 50 % less memory than approach ③. In addition, separate structs for each timeout are easier to handle given the layer separation in `unicoap`. As a result of this analysis, we chose approach ② for use in `unicoap`.

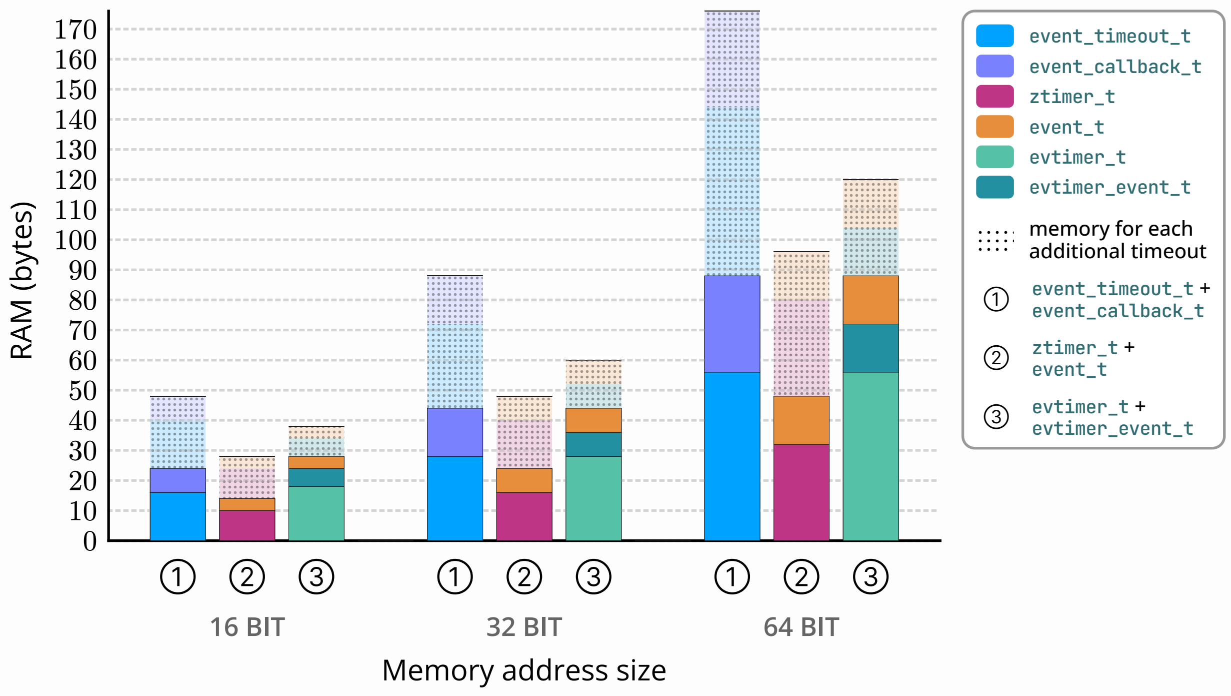


Figure 26: Memory consumption of different timeout containers
Data was acquired with `avr-gcc` and `msp430-elf-gcc` for 16-bit memory addresses, and `clang` for 32 and 64-bit addresses under the `-m32` and `-m64` flags.

4.5 Drivers and Modularization

Driver support in RIOT is modularized. In `unicoap`, we model support for different CoAP combinations using RIOT build system modules, e.g., including `USEMODULE += unicoap_driver_udp` in an application Makefile will import the CoAP over UDP driver. Internally, The `unicoap_driver_udp` and `unicoap_driver_dtls` depend on a common `unicoap_driver_rfc7252_common` submodule that implements the RFC 7252 messaging layer, as illustrated in Figure 27. Dependencies of drivers are automatically imported, except the network backed which must be supplied by the user (e.g., the Generic Network Stack (GNRC)). While all current CoAP combinations we support use the RIOT socket API, future drivers are not bound to the socket API which requires a network backend. Dependencies required for socket support and features exclusive to socket-based drivers are bundled in a `unicoap_sock_support` submodule that a new driver can declare as a dependency. Locations in the codebase that require driver-specific additions are annotated with a `/* MARK: unicoap_driver_extension_point */` comment.

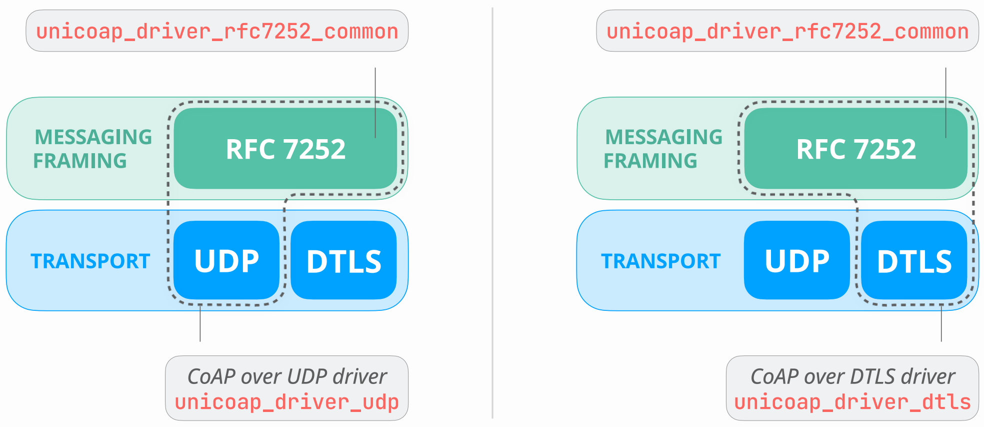


Figure 27: CoAP driver modules

4.5.1 Layer APIs

In this section, we explain the interface between the messaging, exchange and transport layer. Figure 28 illustrates the API calls involved in sending and processing messages.

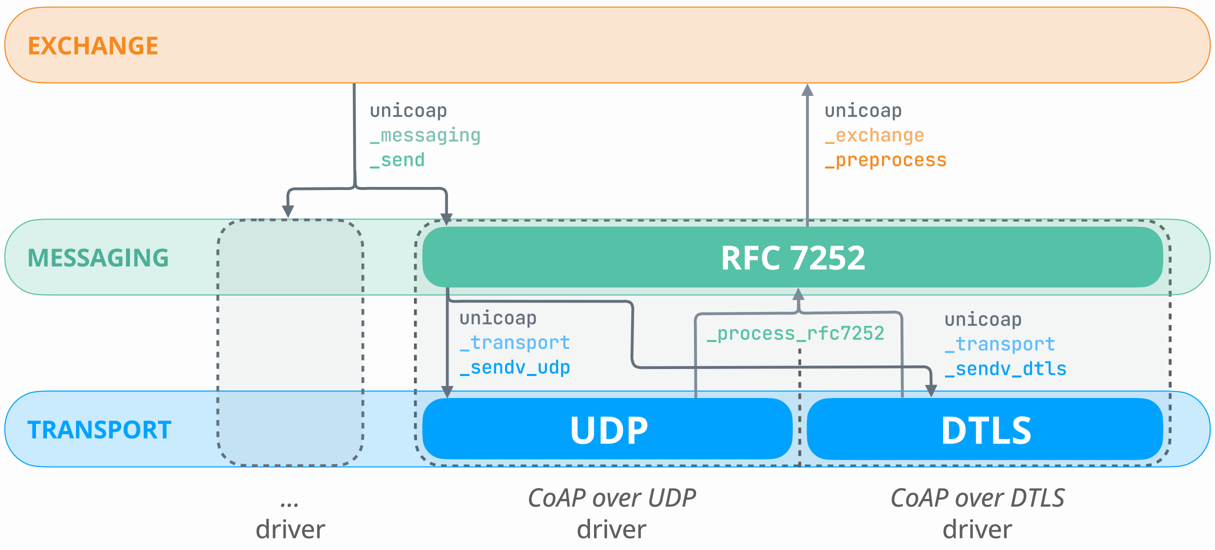


Figure 28: Exchange-, messaging-, and transport-layer APIs

Outbound. Sending messages is facilitated through the `unicoap_messaging_send` API implemented by the messaging layer that forwards the packet to the correct driver. The exchange-layer call to `unicoap_messaging_send` is accompanied by messaging flags originating from the original client API call or resource definition. Similar to block-wise flags, we also reserve a bit region in client and resource flags for messaging.

Figure 29 depicts the bit field region we designate to messaging flags such as `UNICOAP_MESSAGING_FLAG_RELIABLE`. The send function in the messaging layer takes flags in the form of `unicoap_messaging_flags_t`, ensuring that the messaging layer does not need to distinguish between messages in client and server exchanges.

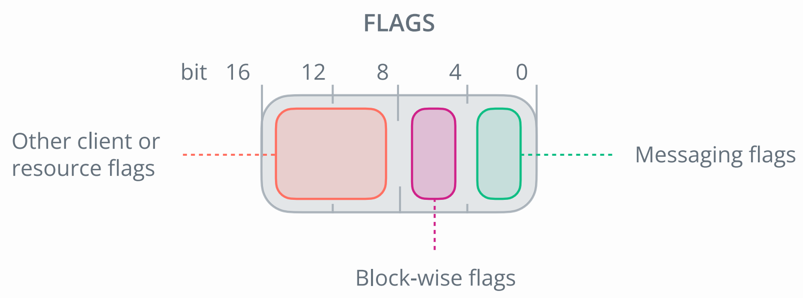


Figure 29: Regions in flag bit fields reserved for block-wise flags and messaging flags `unicoap_client_flags_t` and `unicoap_resource_flags_t` follow this scheme. We define the internal enumerations `unicoap_blockwise_flags_t` and `unicoap_messaging_flags_t` that map to the corresponding bit field region.

Inbound. Inbound message processing message processing is asynchronous. The UDP and DTLS transport-layer implementations leverage the asynchronous `sock` API in combination with the event queue on the `unicoap` thread. Once the message has been parsed and handled by the messaging layer, the driver must call `unicoap_exchange_preprocess`, i.e., engage in the two-step processing procedure outlined in Section 3.4. In the first step, the exchange layer verifies the message has not been truncated and is an acceptable request or expected response or notification. In this step, incoming responses are matched against the outstanding exchange, i.e., the corresponding memo is looked up. Otherwise, the specific error reason is returned to the messaging layer, giving it the opportunity to react accordingly. As part of the preprocessing result, we return messaging flags relevant to this exchange as illustrated in Listing 31. Furthermore, to remove the need for distinguishing requests and responses, we also return a function pointer to the exchange-layer processing function, i.e., either `unicoap_client_process_response` or `unicoap_server_process_request`. After the initial step, before the application is going to process the message, the messaging layer can also elicit an acknowledgement to prevent the risk of the beforementioned timing attack. Finally, the driver must call the processing function.

```
unicoap_preprocessing_result_t unicoap_exchange_preprocess(
  unicoap_packet_t* packet,
  unicoap_messaging_flags_t* flags,
  unicoap_processor_t* processor,
  unicoap_processor_arg_t* arg,
  bool truncated);
```

Listing 31: Exchange-layer processing function

`unicoap_packet_t` is a container type encompassing the remote and local endpoint, message, message properties, and a potential transport session. The packet container reduces the number of parameters pushed onto the stack. `*processor` is the function pointer to either `unicoap_client_process_response` or `unicoap_server_process_request`.

4.5.2 RFC 7252 Messaging Module

In the RFC 7252 messaging layer implementation, we track sent confirmable and non-confirmable messages using a `transmission_t` structure allocated in `unicoap_messaging_send`. As part of each `transmission_t`, we store a back-reference to the memo to allow the messaging layer to signal transmission errors to the exchange layer, prompting it to cancel the ongoing exchange. For instance, upon receiving a `RST` message in response to a `NON` or `CON` request, the application callback will be invoked with an error code indicating a request failure.

Unlike non-confirmable transmissions, confirmable transmissions contain a copy of the PDU for retransmission. For this purpose, we maintain a statically allocated list of *carbon copies* in the RFC 7252 messaging module, i.e., PDU copies of messages that have already been sent. The number of transmissions and carbon copies can be configured, and thus also the ratio between expected outbound confirmable and non-confirmable messages. This design permits application developers to lower the suite's memory consumption by restricting transmissions to be non-confirmable only. We also utilize the array of carbon copy buffers shown in Figure 30 for copies of piggybacked request acknowledgments when deduplication is enabled in order to send the same acknowledgement for each duplicate request.

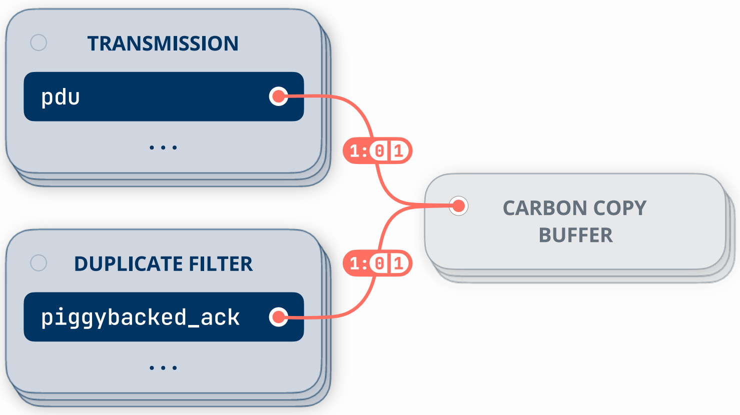


Figure 30: Carbon copies in the RFC 7252 messaging module
Transmissions and duplicate filters can *optionally* use buffers from the *same* carbon copy buffer pool.

Message deduplication. When processing new inbound messages, the RFC 7252 messaging module filters duplicate requests. Figure 31 provides an overview of the steps involved in filtering and acknowledging confirmable duplicates. Provided a resource has the `UNICOAP_RESOURCE_FLAG_DEDUPLICATE` set, `unicoap_exchange_preprocess` will emit the `UNICOAP_MESSAGING_FLAG_DEDUPLICATE` into the messaging layer. Seeing this flag, the RFC 7252 implementation checks if a deduplication filter has already been configured for the given message ID. In case a filter exists, a request with the same ID has been processed before. Given a message ID is unknown to the messaging layer, it will allocate a `deduplication_filter_t` structure that retains the message ID for a given, preconfigured time. Once the timeout has elapsed, the filter is deactivated.

Provided a matching filter has been found, a *non-confirmable* duplicate request is ignored and not forwarded to the exchange layer. Conversely, *confirmable* duplicates need to be acknowledged. Depending on the response style, the original piggybacked `ACK` or an empty `ACK` is retransmitted. However, deferred responses interfere with message deduplication. The `Auto-ACK` timeout introduced in Section 3.5.1 determines the maximum duration a piggybacked `ACK` can be sent in response to confirmable requests. Once the `Auto-ACK` timespan has elapsed, an empty `ACK` is sent automatically, forcing the response to be sent as a separate message. If a duplicate request reaches the node before the `Auto-ACK` timeout, an empty `ACK` would be sent due to the deduplication filter, thereby forcing the server to send a separate response later.

Eliciting an `ACK` for duplicates while neither the deferred response (in a piggybacked `ACK`) nor an empty `ACK` has been sent yet diminishes the chance to send a piggybacked `ACK`. On account of this conflict, we introduce a `sent_ack` property of the deduplication filter, indicating whether an `ACK` has already been sent. For deferred responses, `sent_ack` is `false` until either the `Auto-ACK` timeout elapses and an empty `ACK` is sent or a piggybacked `ACK` with the response attached is sent before the timeout. This ensures acknowledgements for duplicate requests do not obstruct the deferred response mechanism. Only if a regular `ACK` has been sent before, we allow the deduplication filter to elicit an empty `ACK` for retransmitted confirmable requests.

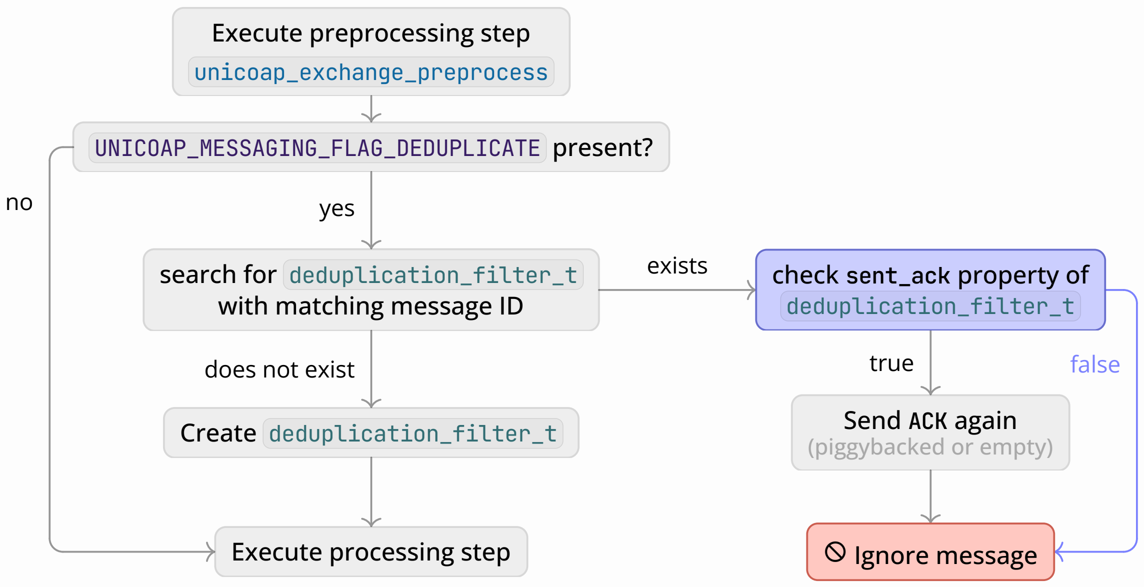

Execute preprocessing step
unicoap_exchange_preprocess
UNICOAP_MESSAGING_FLAG_DEDUPLICATE present?
no
yes
search for deduplication_filter_t with matching message ID
exists
check sent_ack property of deduplication_filter_t
does not exist
true
false
Create deduplication_filter_t
Send ACK again
(piggybacked or empty)
Execute processing step
Ignore message

Figure 31: Duplicate `CON` message filtering flow chart for RFC 7252
In the figure above, nodes and edges colored purple relate to special treatment for deferred responses.

4.6 Convenience Features

4.6.1 Shell Application

`unicoap` ships with a shell application that supports block-wise transfers and resource observation. For instance, `coap post -bs -br -r coap://rocket.tld/comms HelloThisIsHouston` will initiate a block-wise `POST` request with payload `HelloThisIsHouston` destined to the resource `comms` located on a host called `rocket.tld`. The `Content-Format` is set to `text/plain`.

- The `-bs` command line flag instructs the client to slice the payload into blocks, equivalent to the `UNICOAP_CLIENT_FLAG_SLICE` flag.
- The `-br` flag ensures all block responses are reassembled. This is equivalent to the `UNICOAP_CLIENT_FLAG_REASSEMBLE` flag.

— Requests are transmitted reliably over UDP using confirmable messages due to the `-r` flag being present.

Beyond the `coap` command, the shell enables introspection of the entirety of the suite's internal state using the `coap-state` instruction. Listeners, resources, memos, block-wise transfers, `Observe` registrations, and messaging state will be printed. Appendix C shows a sample state dump. The APIs for dumping internal state can alternatively be invoked programmatically from any application. We also provide convenience functions to create human-readable descriptions from option numbers, request methods, status codes, signals, and flags. The shell folder also includes scripts to test interoperability with `aiocoap`.

4.6.2 Safeguards

We implement several safeguard measures that catch improper API usage and obvious misconfigurations. Static misconfigurations like out-of-bounds error when setting the token length are caught using `static_assert` invocations. Furthermore, logical misconfigurations are detected, such as setting the `Auto-ACK` timeout to occur after the application response timeout.

Throughout the codebase, we use a custom `unicoap_assist` macro to emit a warning for missing modules at runtime. For instance, passing any of the block-wise flags into a public API will cause the warning in Listing 32 to be printed if the block-wise support module is missing.

```
WARNING: block-wise flag used, but module is missing
+++ FIXIT: add USEMODULE += unicoap_blockwise
```

Listing 32: Sample warning with instructions for adding block-wise module

If the documented contract of a type is violated, we inform users about the API being misused and offer advice on how to fix the given issue, such as when forgetting to send a response from a resource handler, as shown in Listing 33.

```
API MISUSE: handler did not respond
+++ FIXIT: add USEMODULE += unicoap_deferred_response and call unicoap_defer_response
+++ FIXIT: ignore request by returning UNICOAP_IGNORING_REQUEST
```

Listing 33: Sample warning with instructions for handling response

The logic implementing assistance messages can be stripped from the application binary by disabling the configuration setting `CONFIG_UNICOAP_ASSIST` and enabling link-time optimization.

5 Evaluation

In this chapter, we evaluate `unicoap` in terms of build size, memory usage, and processing time and compare `unicoap` with GCoAP and nanoCoAP.

5.1 Measurement Setup

The measurements in this section have been conducted on an nRF52840 SoC with a Cortex-M4 64 MHz MCU, 256 KiB of RAM, and 1 MiB of ROM [24]. We use the master branch of RIOT (commit `9a5ef12`) for our experiments, with the parameters listed in Table 4 and the `-O3` compiler optimization level. We also deactivate safety checks in RIOT using `DEVELHELP` where possible and enable the `NDEBUG` macro. For our tests, we disable any debug logging and serial output. Experiments that involve exchanging data wirelessly use the GNRC network stack in RIOT, 6LoWPAN, and the IEEE 802.15.4 on-board radio.

Parameter	Value	Description
`CONFIG_UNICOAP_PDU_SIZE_MAX`	64	Size of retransmission buffer
`CONFIG_UNICOAP_OPTIONS_BUFFER_DEFAULT_CAPACITY`	24	Maximum size of options data
`CONFIG_UNICOAP_BLOCK_SIZE`	32	Size of payload per block message
`CONFIG_UNICOAP_MEMOS_MAX`	2	Number of active parallel exchanges
`CONFIG_UNICOAP_RFC7252_TRANSMISSIONS_MAX`	2	Maximum number of PDU copies for retransmission
`CONFIG_UNICOAP_BLOCKWISE_TRANSFERS_MAX`	2	Number of active parallel block-wise transfers
`CONFIG_UNICOAP_BLOCKWISE_BUFFERS_MAX`	2	Number of buffers for slicing and reassembling
`CONFIG_UNICOAP_BLOCKWISE_REPRESENTATION_SIZE_MAX`	512	Maximum size of reassembled payload
`CONFIG_UNICOAP_DEBUG_LOGGING`	`0`	Warnings, error messages
`CONFIG_UNICOAP_ASSIST`	`0`	Helpful logs and tips
`CONFIG_GCOAP_PDU_BUF_SIZE`	64	Size of retransmission buffer, influences total size of options
`CONFIG_GCOAP_REQ_WAITING_MAX`	2	Number of active parallel client exchanges
`CONFIG_GCOAP_RESEND_BUFS_MAX`	2	Maximum number of PDU copies for retransmission

Table 4: Compile-time parameters in RIOT
We translated features in `unicoap` to their equivalent in GCoAP and configured parameters accordingly.

5.2 Memory

5.2.1 ROM and RAM

First, we analyze memory requirements of `unicoap` and compare them with GCoAP. We choose GCoAP over nanoCoAP Sock due to GCoAP covering a similar feature set and architecture, i.e., both feature an asynchronous event queue. Functionally equivalent applications were implemented using `unicoap` and GCoAP. The applications incorporate client and server functionality. The `unicoap` application can be optionally compiled with support for automatic block-wise transfers (client and server), deduplication (server), and URI support (client). In Figure 32, we measure the RAM and ROM consumption in the RIOT binary relevant to `unicoap` and GCoAP. Using `cosy`, we calculate the ROM size by summing up respective symbols in the `.text` and `.data` sections and the RAM size from the `.bss` and `.data` sections.

Size groups. Memory sizes are grouped into the following constituents. *Core* refers to essential features, including message-related and option APIs, exchange-layer state management as well as client and server functionality. As GCoAP depends on the nanoCoAP parser and option APIs, we added these symbols to the *Core* group of GCoAP. Due to the layer separation and modularization in `unicoap`, we display the implementation of *RFC 7252* messaging in a distinct category. The *Core* group of GCoAP includes RFC 7252 instructions and state variables. In the *UDP* and *DTLS* groups, we summarize the respective driver implementations. To visualize the difference in impact on memory consumption, we include the TinyDTLS sock implementation in the *DTLS* group. The *Application* group contains all instructions and variables of the beforementioned sample application. Under the *Block-wise*, *URI* and *Deferred response* categories, we group symbols that are only present in the binary when the respective modules are present at compile-time.

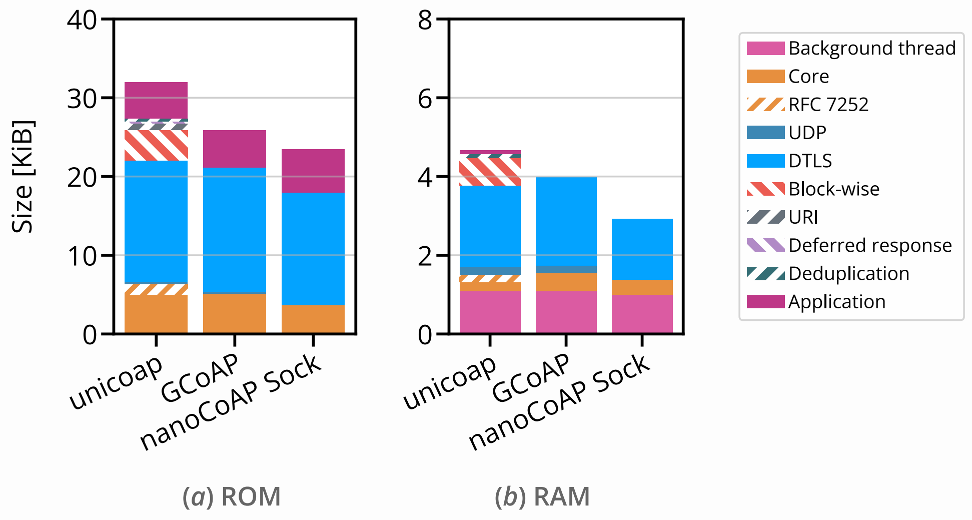


Figure 32: Memory requirements of `unicoap` and GCoAP
Support for URIs, automatic block-wise transfers, deferred responses, and deduplication is optional. The GCoAP core includes RFC 7252 messaging. In `unicoap`, RFC 7252 messaging is modularized.

Results. Measurements of the application size yield similar results. The `unicoap` application occupies ≈ 4.6 KiB of RAM and the GCoAP application requires about 4.8 KiB. The small discrepancy could be attributed to `unicoap` not mandating the allocation of a PDU buffer. *DTLS* and *UDP* sizes are approximately equal. Adding the *RFC 7252* and *Core* memory consumption, `unicoap` needs ≈ 0.4 KiB of RAM and approximately 6.3 KiB of ROM. The

GCoAP core also adds about 0.5 KiB to the overall RAM consumption and ≈ 5.1 KiB to the ROM size. We conclude that layer separation plays only a minor role in RAM consumption. Without DTLS support, `unicoap` contributes about 12 % to the ROM of the entire RIOT image. GCoAP adds about 10 % to the total ROM consumption. `unicoap` and GCoAP consistute ≈ 3 % and ≈ 4 % of the total RAM consumption, respectively. However, it is important to note that other compile-time configurations will lead to different results.

The built-in block-wise transfer support contributes another ≈ 0.7 KiB. However, this particular result needs to be interpreted with caution as the block-wise memory impact can be drastically reduced by modifying compile-time parameters. Our sample application was configured to support two parallel block-wise transfers, each with a 512-byte buffer. GCoAP does not feature automatic block-wise transfers.

5.2.2 Stack Usage

Next, we measure the maximum stack usage of the `unicoap` thread compared to the GCoAP background thread. For our measurements, we use the `test_utils_print_stack_usage` module in RIOT. We prepare a node running the same application as in the previous section and measure the stack usage on the `unicoap` thread in two phases. First, we initiate a request on the node being tested and wait until after the response has been processed by the application to measure the stack usage (client). Then, we send a request from another node and measure the stack usage after the application has responded to the request (server).

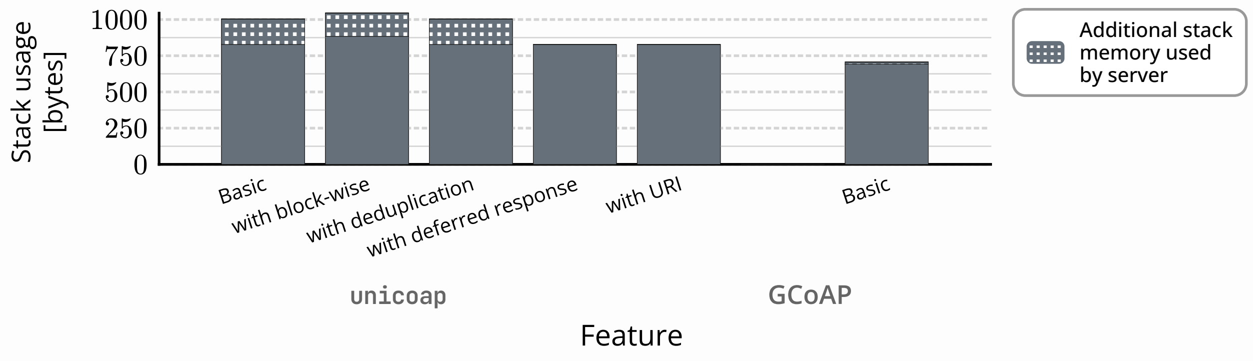


Figure 33: Stack usage of `unicoap` and GCoAP

Results. As visualized in Figure 33, the stack usage is 136 bytes higher with `unicoap` than with GCoAP. This corresponds to less than 0.1 % of the available RAM on our platform. Server functionality contributes another 176 bytes to the stack usage of `unicoap`. GCoAP has a comparatively low overhead of server functionality of just 16 bytes. This discrepancy is likely caused by the layer separation and, hence, deeper calls stacks in `unicoap`. The effect of an increased number of nested function calls is amplified when using server features: From the client side, no message is sent from the response handler. On the server side, a response message is sent in addition to processing the request. Thus, the layer separation overhead affects the stack usage twice, i.e., once while the request traverses the layers from the network to the application and once more while passing the response from the application to the lower layers.

Moreover, additional features have little to no effect on the stack usage: Automatic block-wise transfers add only 40 bytes to the stack usage. Deduplication and URI support have no implications on the peak stack usage of the `unicoap` thread.

5.3 Timing

In this section, we evaluate timing properties of `unicoap`.

5.3.1 Stack Traversal Times

First, we determine the time needed for a message to traverse the different layers of `unicoap`. For this, we use the server features of the application we also employed for the memory measurements. We observe each message passing from the network to the request handler in the application and the response back to the network. We measure the time needed for each layer using the `ztimer_usec` clock. We map the logic of each layer in `unicoap` to GCoAP and measure the delay in GCoAP at the respective points. In Figure 34, we measure the average traversal times using two approaches. First, we inject a UDP datagram containing a CoAP message into the UDP driver to analyze the overhead of the driver implementation without delays introduced by the network backend. Then, we inspect traversal times by sending a CoAP request over the IEEE 802.15.4 network to the node running the sample application. We also measure the processing delay of the block-wise implementation for every subsequent block message that is handled by `unicoap`, i.e., when `unicoap` automatically sends a response asking for the next request block or sends the next response block.

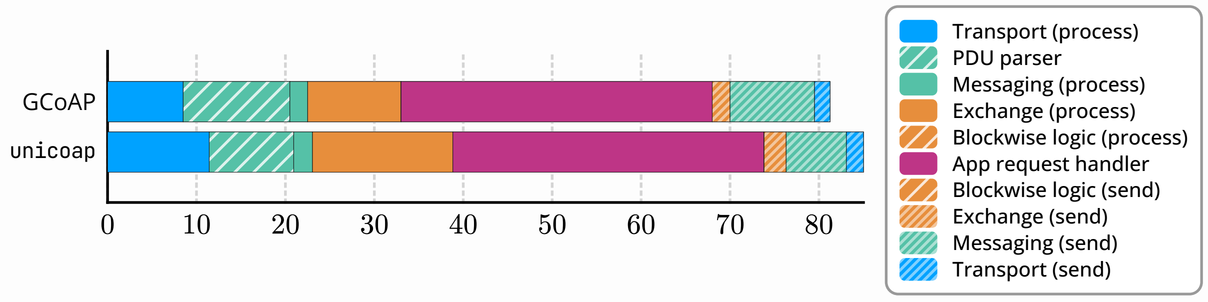


Figure 34: CoAP stack traversal times for a server application without networking for $N = 500$ requests.
In this figure, *'process'* denotes inbound message handling. *'Send'* refers to the logic responsible for sending an outbound message.

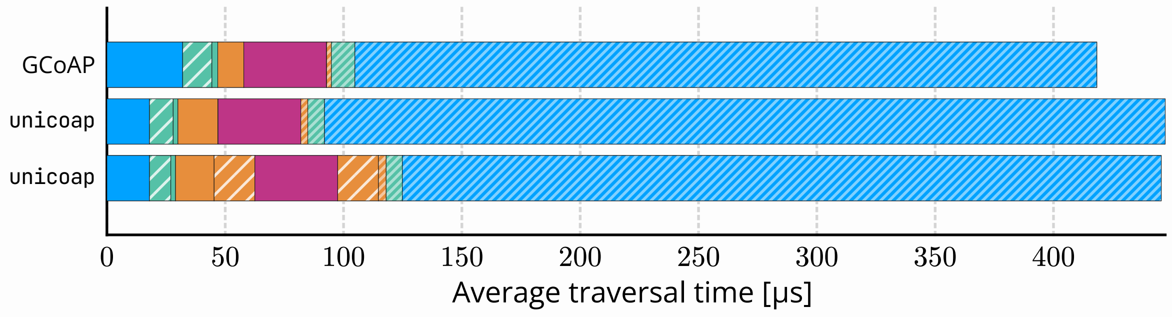


Figure 35: CoAP stack traversal times of a server in an IEEE 802.15.4 network for $N = 20$ requests

Results. In our experiments, the UDP transport implementation of GCoAP is marginally faster than GCoAP by ≈ 3 μs. Parsing took about 21 % less time with `unicoap` than with GCoAP/nanoCoAP. Exchange-layer logic in the inbound direction amounted to 16 μs compared to 11 μs in the GCoAP example. However, the results obtained when also measuring the delay introduced by the network backend indicate that differences in traversal times above the transport layer are negligible. In our tests, the transport-layer sending operation took more than 300 μs, which is between four and five times as long as the sum of all remaining delays. The observed, slight decrease in transport-layer processing times when using a real network backend compared to GCoAP may be the result of `unicoap` leveraging the zero-copy sock API, avoiding an additional PDU copy operation. The block-wise processing times amount to approximately 34 μs. Consequently, the block-wise transfer implementation has only a little performance overhead, with the additional benefit of reducing boilerplate and providing ease of use.

5.3.2 Options API and Parser

Finally, we benchmark our parser and options implementation by contrasting it with nanoCoAP. For this experiment, we prepare a message with a specified number of options and measure the time needed for several operations. In Figure 36, we show the average time needed for these operations relative to the number of existing options. For each of the operations *get*, *add/insert*, and *remove*, we additionally differentiate between a trivial case (the option in question is located at the end of the options buffer) and a more complex case (the option is located in between other options). This allows us to reveal performance impacts due to the location of the option. In addition to analyzing the option algorithms, we benchmark our parser.

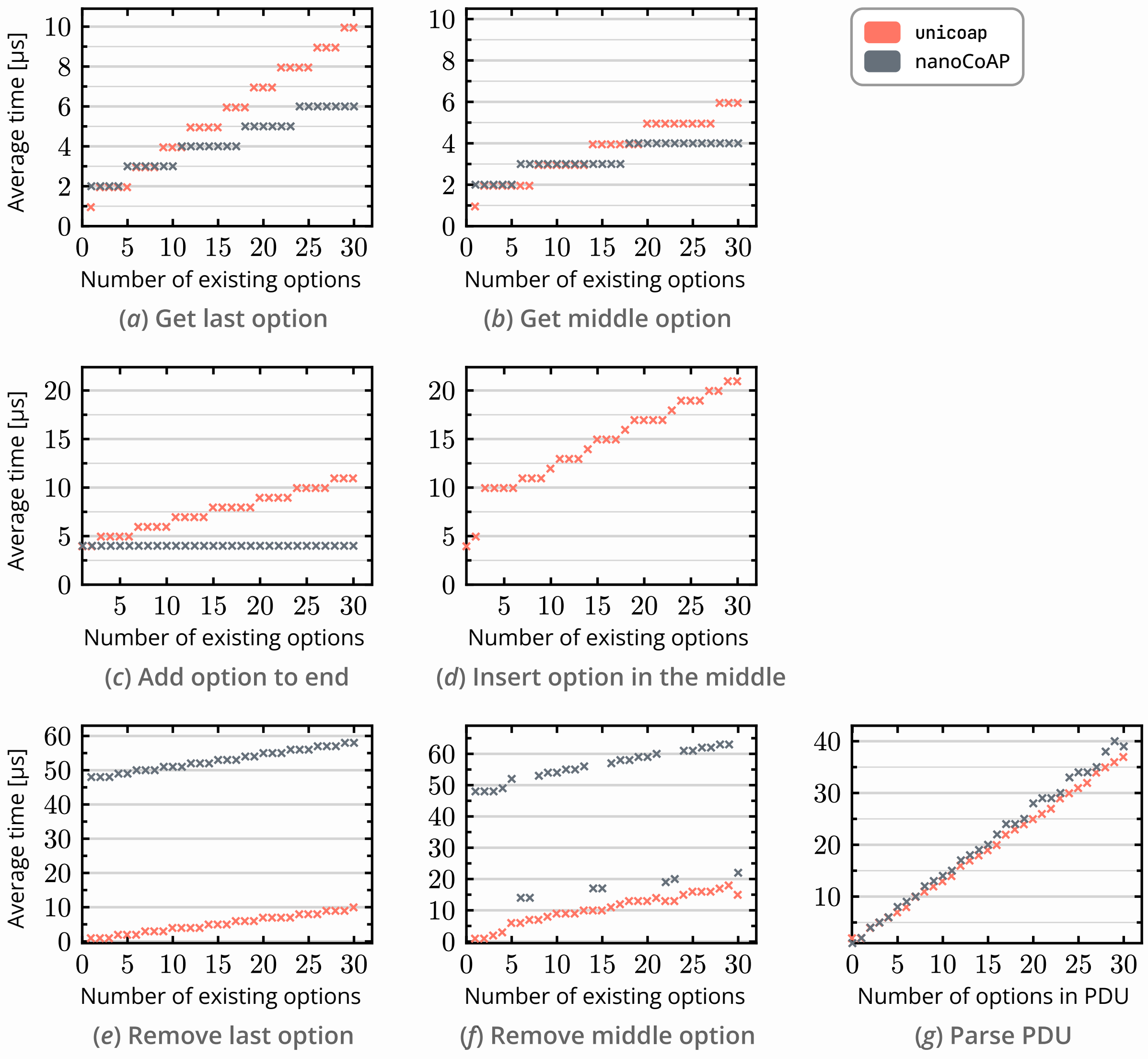


Figure 36: CoAP options and parser benchmark
In this figure, we compare the `unicoap` and nanoCoAP parser and option accessors. We sample each function in $N = 1000$ rounds. The step effects occurring in (*a*) – (*d*) are artifacts of the 1 µs visualization resolution.

Getter. As illustrated in (*a*) and (*b*), the average time needed to retrieve an option value grows linearly with the number of options present in the buffer. The lower ascent in (*b*) with both `unicoap` and GCoAP is likely due to the linear search, i.e., the algorithm finds an option in the middle earlier than at the end. The marginally larger growth rate of the `unicoap` algorithm in both diagrams might be caused by a higher number of sanity checks in `unicoap`.

Setter. Diagram (*c*) indicates a complexity of $O(1)$ of the nanoCoAP *add option* method while our version grows linearly with the number of options already present. nanoCoAP requires options to be inserted in order and thus does not need to check whether there are adjacent options. However, `unicoap` first iterates over the list of options until the right insertion slot has been found. Depending on whether the trivial case applies, the average processing time varies. Moreover, we employ more safeguard checks to prevent adding options without sufficient buffer capacity. Plot (*c*) depicts the trivial case whereas diagram (*d*) shows the complex case where trailing options change and must be moved. The latter plot does not contain data for nanoCoAP due to the lack of support for inserting options out of order.

Remove. Plots (*e*) and (*f*) compare our version of the *remove* operation with the nanoCoAP implementation. On average, our implementation is faster by more than 45 µs. The cause of the outliers in diagram (*f*) has not been identified yet.

Parser. As shown in plot (*g*), our parser performs slightly better than nanoCoAP does. This effect has already been shown in Figure 34 and Figure 35.

5.3.3 Inter-Request Time

Next, we compare the inter-request request time of a node running `unicoap` with a node running GCoAP. Therefore, we deploy a requester node with a RIOT application that repeatedly sends a prepared CoAP PDU by directly invoking the sock API. The tester node does not wait for responses. This way, we avoid any overhead introduced by a CoAP stack. The nodes are located approximately 10 cm apart from each other in an office environment. On the server node being tested, we measure the time between two subsequent requests from the application. Figure 37 shows the data collected as part of this experiment.

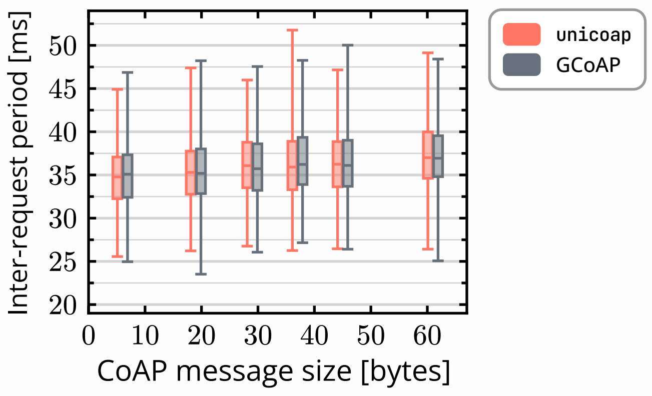


Figure 37: CoAP inter-request period of a server over 6LoWPAN over IEEE 802.15.4 for $N = 20$ requests per message size group.

Results. We measured inter-request periods between ≈ 25 ms and 50 ms, with most requests arriving within a range of 32 ms and 40 ms after the previous one. Half of all measured inter-request periods spread by no more than ≈ 5 ms around the median. Moreover, 50% of all inter-request periods measured for `unicoap` align with GCoAP within a margin of 1 ms (noise). Furthermore, the inter-request periods we identified in this experiment are between three and four orders of magnitude larger than the traversal times evaluated in a previous experiment. We conclude that traversal times and processing delays do not have a significant effect on message inter-request times on our platform.

6 Conclusion and Outlook

In this thesis, we proposed `unicoap`, a unified and modular CoAP suite for the constrained IoT. `unicoap` aims to address the need for an intuitive, high-level CoAP interface and an extensible library design that can adjust to different transport requirements.

Layered architecture. Previously, CoAP support in RIOT lacked layer separation, hindering the addition of other transports. In this thesis, we exhibited the challenges of designing a layered implementation of CoAP. We generalized features into layers and defined common interfaces between these layers. In particular, we highlighted the complexity of managing state across layers. With our implementation in RIOT, we demonstrated that layer separation in a CoAP implementation is feasible in a constrained environment. In Chapter 5, we also showed that layer abstractions in `unicoap` have only a very small effect on memory requirements and no significant implications on timing in our experiment. Furthermore, we revealed that similar memory and timing properties can be achieved compared to an implementation without a layered architecture.

Modularization. We group support for different CoAP transports into modules, providing a clear encapsulation of different messaging and transport properties and easing dependency management. Furthermore, CoAP supports a spectrum of extensions including block-wise transfer that influence memory metrics to different degrees. By modularizing CoAP features into components, we give developers tight control over binary sizes and memory usage. Our evaluation in Chapter 5 confirmed that modularization of CoAP drivers helps reduce RAM and ROM demands.

High-level interface. Furthermore, we have shown that designing an API that balances flexibility and scalability in terms of future transport-layer variations is possible. We illustrated the benefit of an extensible API: By abstracting features like support for different types of destination addresses, the API can adapt to future additions without breaking source compatibility. OSCORE support has been anticipated in our API design, enabling easy future integration of OSCORE functionality. Moreover, `unicoap` abstracts advanced features like block-wise transfer and deduplication away from the application, thereby reducing application complexity and increasing the code readability. Built-in functionality does not significantly affect performance, as we have shown in Chapter 5. Features such as deferred responses could ease the development of more sophisticated server applications. In the future, the integration of automatic block-wise transfers in `unicoap` may also lead to a lower entry barrier for using these features in applications.

Optimizations. In this thesis, we analyzed different versions of data structures that offer the best properties for constrained devices to maintain a small memory footprint. Additionally, our implementation in RIOT allows for a high level of customization, with configuration parameters that can further reduce the memory consumption of `unicoap` tailored to application needs.

Outlook. Future work shall migrate modules in RIOT that currently build upon GCoAP or nanoCoAP, such as the DNS over CoAP and proxy implementations. Future work could focus on further optimizing the memory consumption of `unicoap`. Furthermore, the profiles feature of our API provides a solid foundation for implementing OSCORE. Moreover, thanks to the layered architecture of `unicoap`, support CoAP over TCP can be added.

In summary, `unicoap` could foster a faster adoption of current and future CoAP transports and may, as a result, contribute to increased interoperability in the constrained IoT.

Bibliography

[1] C. Bormann, M. Ersue, and A. Keränen, “Terminology for Constrained-Node Networks.” [Online]. Available: https://www.rfc-editor.org/info/rfc7228

[2] “IEEE Standard for Low-Rate Wireless Networks,” *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, vol. 0, no. , pp. 1–800, 2020, doi: 10.1109/IEEESTD.2020.9144691.

[3] M. S. Lenders, T. C. Schmidt, and M. Wählisch, “Fragment Forwarding in Lossy Networks,” *IEEE Access*, vol. 9, no. , pp. 143969–143987, 2021, doi: 10.1109/ACCESS.2021.3121557.

[4] N. Naik, “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP,” in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7. doi: 10.1109/SysEng.2017.8088251.

[5] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, “Performance evaluation of MQTT and CoAP via a common middleware,” in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, 2014, pp. 1–6. doi: 10.1109/ISSNIP.2014.6827678.

[6] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP).” [Online]. Available: https://www.rfc-editor.org/info/rfc7252

[7] C. Amsüss, Z. Shelby, M. Koster, C. Bormann, and P. V. der Stok, “Constrained RESTful Environments (CoRE) Resource Directory.” [Online]. Available: https://www.rfc-editor.org/info/rfc9176

[8] C. Bormann and Z. Shelby, “Block-Wise Transfers in the Constrained Application Protocol (CoAP).” [Online]. Available: https://www.rfc-editor.org/info/rfc7959

[9] G. Selander, J. P. Mattsson, F. Palombini, and L. Seitz, “Object Security for Constrained RESTful Environments (OSCORE).” [Online]. Available: https://www.rfc-editor.org/info/rfc8613

[10] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, and B. Raymor, “CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets.” [Online]. Available: https://www.rfc-editor.org/info/rfc8323

[11] C. Amsüss, "CoAP over GATT (Bluetooth Low Energy Generic Attributes)," Internet Engineering Task Force, Sep. 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-amsuess-core-coap-over-gatt/07/

[12] E. Baccelli *et al.*, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018, doi: 10.1109/JIOT.2018.2815038.

[13] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An experimental study of home gateway characteristics," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, in IMC '10. Melbourne, Australia: Association for Computing Machinery, 2010, pp. 260–266. doi: 10.1145/1879141.1879174.

[14] K. Hartke, "Observing Resources in the Constrained Application Protocol (CoAP)." [Online]. Available: https://www.rfc-editor.org/info/rfc7641

[15] RIOT, "nanoCoAP small CoAP library." Accessed: Jan. 29, 2024. [Online]. Available: https://doc.riot-os.org/group__net__nanocoap.html

[16] RIOT, "nanoCoAP Sock." Accessed: Jan. 29, 2024. [Online]. Available: https://doc.riot-os.org/group__net__nanosock.html

[17] RIOT, "GCoAP." Accessed: Jan. 29, 2024. [Online]. Available: https://doc.riot-os.org/group__net__gcoap.html

[18] C. Amsüss and M. S. Lenders, "CoAP Transport Indication," Internet Engineering Task Force, Oct. 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-core-transport-indication/07/

[19] RIOT, "Event Queue." Accessed: Dec. 17, 2024. [Online]. Available: https://doc.riot-os.org/group__sys__event.html

[20] R. Kijewski and M. Rottleuthner, "modules.h." [Online]. Available: https://doc.riot-os.org/modules_8h.html

[21] RIOT, "ztimer high level timer abstraction." Accessed: Dec. 19, 2024. [Online]. Available: https://doc.riot-os.org/ztimer_8h.html

[22] RIOT, "Event Queue." Accessed: Dec. 19, 2024. [Online]. Available: https://doc.riot-os.org/group__sys__event.html

[23] RIOT, "Millisecond interval event timers." Accessed: Dec. 19, 2024. [Online]. Available: https://doc.riot-os.org/group__sys__evtimer.html

[24] "nRF52840 Product Specification v1.11." 2024. [Online]. Available: https://docs-be.nordicsemi.com/bundle/ps_nrf52840/attach/nRF52840_PS_v1.11.pdf?_LANG=enus

A Terminology

To simplify explanations, we assign names to some concepts and processes. The respective terms are listed below.

TRANSPORTS

CoAP combination A version of the Constrained Application protocol adapted to a certain transport protocol. The PDU header and messaging behavior varies between CoAP combinations. *Examples: CoAP over UDP [6], CoAP over DTLS [6], CoAP over TCP [10], CoAP over TLS, [10], CoAP over WebSockets [10], CoAP over WebSockets over TLS [10], CoAP over GATT [11].*

Transport-specific characteristics Features unique to a certain a CoAP combination.

Driver Modular implementation of a CoAP combination.

Endpoint A CoAP node in a network.

Remote endpoint Distant CoAP node in a network. Equivalent to the server endpoint from a client perspective or the client endpoint from a server perspective. The destination of a sent message or the source of message received.

Local endpoint The node `unicoap` runs on. The source of messages sent and the destination of messages received.

BLOCK-WISE TRANSFER

Block-wise transfer stage Phase a block-wise transfer is in. The transfer is in the `Block1` stage when the request is being transmitted and in the `Block2` stage when the response is transferred.

Representation The entire, original payload transmitted in individual blocks in a block-wise transfer.

Slice The operation a party in a block-wise performs when dividing a larger payload into chunks.

Collect The act of instructing the other party in a block-wise transfer to send more blocks while processing each block.

Reassemble The process of accumulating blocks and reconstructing the entire, original payload.

LAYER SEPARATION

Exchange Series of requests and responses needed to convey a representation to a server and back to the client. May involve successive notification responses.

Messaging Mechanism by which the transmission of messages is managed.

Message properties Messaging-related fields in a CoAP PDU not directly relevant to an application.

EXCHANGE LAYER

Profile Set of characteristics in an exchange that dictate special treatment must be applied to messages sent and/or received. *Example: OSCORE security context.*

`Observe` registration, notification registration Piece of information present on a server that is used to send notification responses to interested clients.

Internal token length Length of CoAP token generated by `unicoap` for client exchanges.

External token length Maximum length of a token `unicoap` can receive from another CoAP node.

MESSAGING LAYER

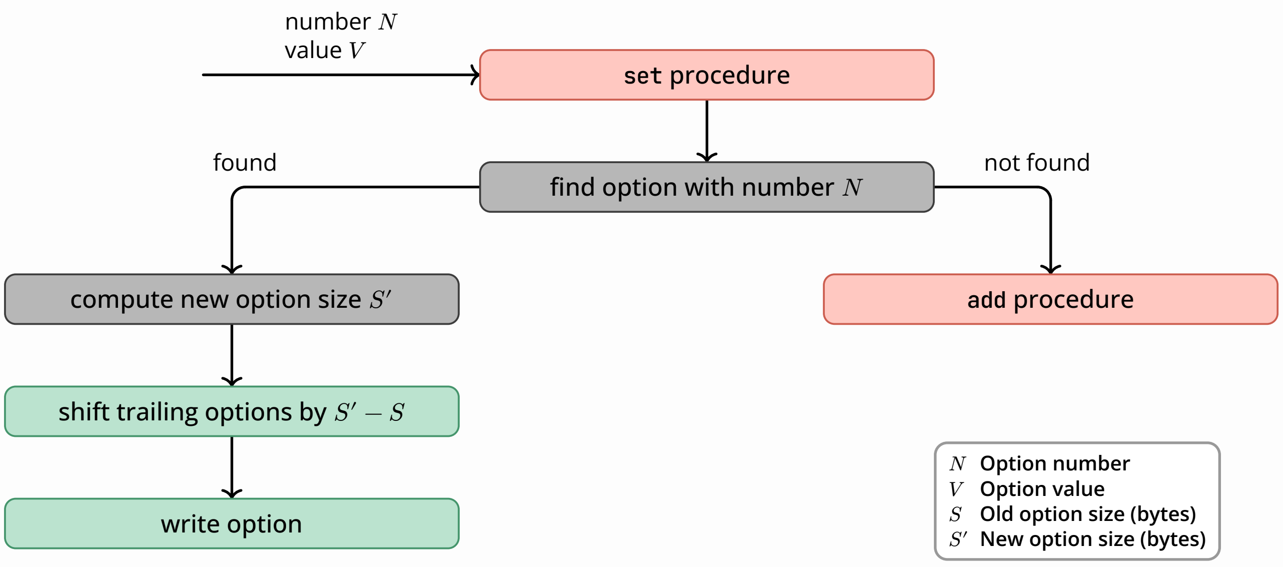
RFC 7252 messaging The messaging protocol used in CoAP over UDP and CoAP over DTLS.

Transmission The effort of sending a message to another endpoint

Auto-`ACK` timeout Timespan that has to elapse before a `unicoap` server sends an empty acknowledgement in response to a request. Applies to RFC 7252 messaging only. Before the timeout will have elapsed, a response will be sent as a piggybacked `ACK`. After the timeout, `unicoap` sends a separate response (`CON` or `NON`).

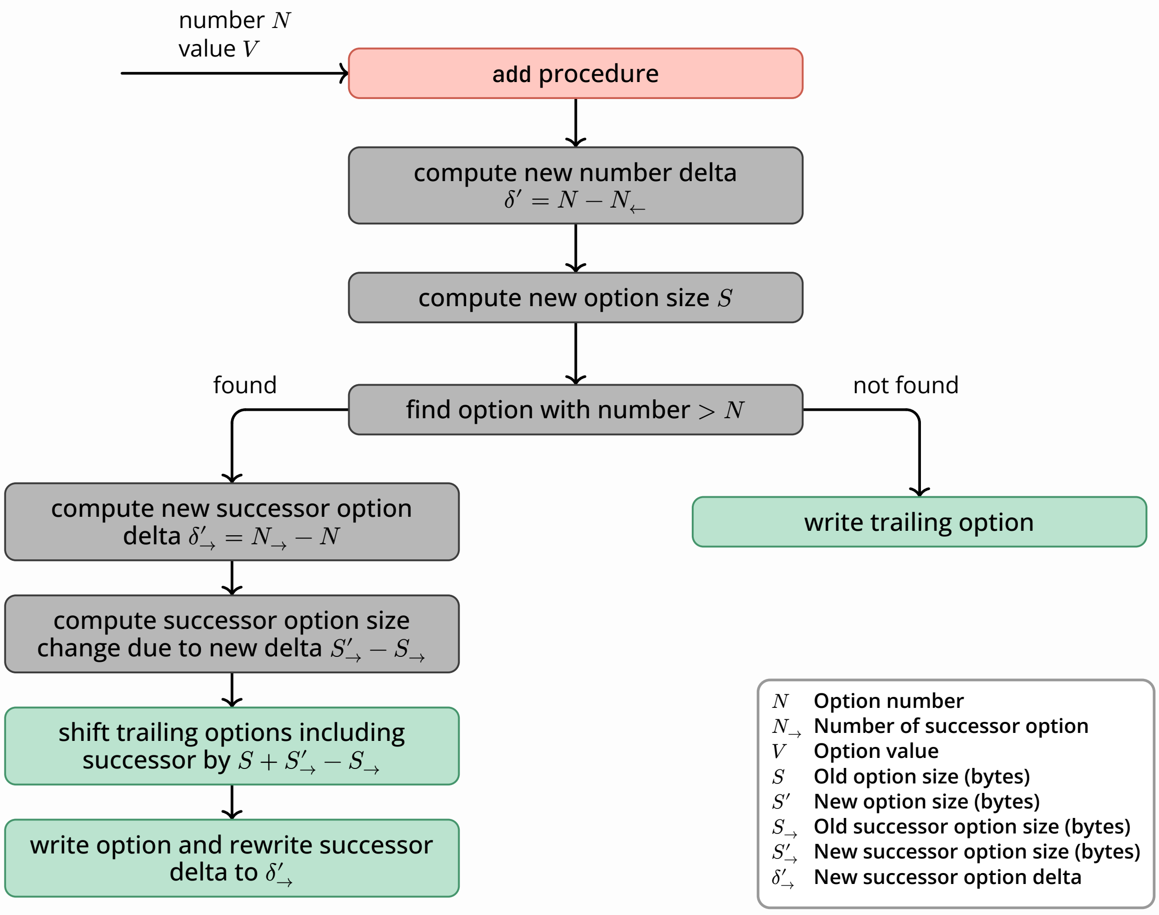
Deferred response A response sent after the application indicated to `unicoap` it is going to take longer to prepare the response.

B Option Manipulation Algorithms



number N
value V
set procedure
found
find option with number N
not found
compute new option size S'
add procedure
shift trailing options by $S' - S$
write option
N Option number
V Option value
S Old option size (bytes)
S' New option size (bytes)

Figure 38: Control flow of the `set` option operation



number N
value V
add procedure
compute new number delta $\delta' = N - N_{\leftarrow}$
compute new option size S
find option with number $> N$
found
not found
compute new successor option delta $\delta'_{\rightarrow} = N_{\rightarrow} - N$
compute successor option size change due to new delta $S'_{\rightarrow} - S_{\rightarrow}$
shift trailing options including successor by $S + S'_{\rightarrow} - S_{\rightarrow}$
write option and rewrite successor delta to $\delta'_{\rightarrow}$
write trailing option
N Option number
$N_{\rightarrow}$ Number of successor option
V Option value
S Old option size (bytes)
S' New option size (bytes)
$S_{\rightarrow}$ Old successor option size (bytes)
$S'_{\rightarrow}$ New successor option size (bytes)
$\delta'_{\rightarrow}$ New successor option delta

Figure 39: Control flow of the `add` option operation

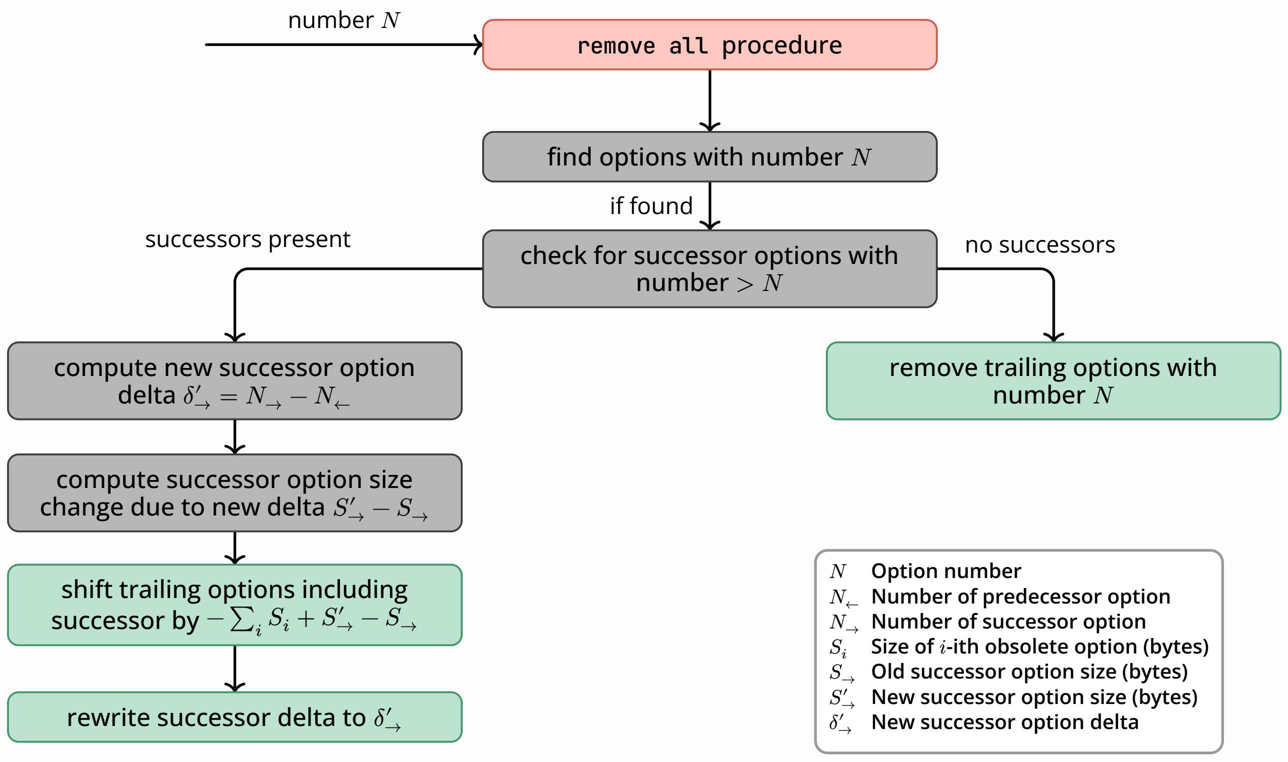

number N
remove all procedure
find options with number N
if found
check for successor options with number $> N$
successors present
no successors
compute new successor option delta $\delta'_{\rightarrow} = N_{\rightarrow} - N_{\leftarrow}$
compute successor option size change due to new delta $S'_{\rightarrow} - S_{\rightarrow}$
shift trailing options including successor by $-\sum_i S_i + S'_{\rightarrow} - S_{\rightarrow}$
rewrite successor delta to $\delta'_{\rightarrow}$
remove trailing options with number N
N Option number
$N_{\leftarrow}$ Number of predecessor option
$N_{\rightarrow}$ Number of successor option
S_i Size of i-ith obsolete option (bytes)
$S_{\rightarrow}$ Old successor option size (bytes)
$S'_{\rightarrow}$ New successor option size (bytes)
$\delta'_{\rightarrow}$ New successor option delta

Figure 40: Control flow of the `remove all` option operation

C Sample State Dump

```
  - listeners:
    - listener #0
      - request_matcher=<default>
      - link_encoder=<func at 0x15309>
      - protocols=<all>
      - resources (9):
        - resource #0 /led/color
          - flags=[ RELIABLE OBSERVABLE NOTIFY_RELIABLY SLICE SLICE_NO_COPY REASSEMBLE ]
          - methods=[ GET PUT ]
          - protocols=<all>
          - handler=<func at 0x15c5>
          - argument=<obj at 0>
    - [...]

  - memos (2 total)
    - memo #0
      - endpoint=UDP <sock_tl_ep port=5683 netif=0 ipv6=fe80::1234:5678:abcd:ef90>
      - exchange kind=client
      - client details:
        - token=719c
        - flags=[ RELIABLE ]
        - callback=<func at 0x1411>
        - argument=<obj at 0>
    - memo #1

- block-wise transfers (2 total)
  - block-wise transfer #0
    - stage=slice
    - blockwise_buffer=#0
    - iterator=<blockwise_iterator
        block_size=64 block_number=8 offset=563 representation=<buffer at 0x44e720 size=563>>
  - block-wise transfer #1

- block-wise buffers (2 total)
  - buffer #0 capacity=1400B used=1
  - buffer #1 capacity=1400B used=0


  - RFC 7252 transmissions (2 total):
    - transmission #0
      - MID=33378
      - remaining_retransmissions=3
      - pdu=<carbon_copy at 0x20006edc>
      - pdu_size=6
    - transmission #1

  - RFC 7252 carbon copies (2 total):
    - carbon_copy #0 capacity=200B used=1
    - carbon_copy #1 capacity=200B used=0

  - RFC 7252 deduplication filters (2 total):
    - filter #0
      - filtered MID=33380
      - endpoint=UDP <sock_tl_ep port=5683 netif=0 ipv6=::1>
      - sent_ack=1
      - confirmable=1
      - piggybacked_ack=<carbon_copy at 0x20006fa4>
      - ack_size=6
    - filter #1
```

Listing 34: Sample state dump