aiocoap

Release 0.4.5

Nov 22, 2022

Contents

1 Usage 3

2 Features / Standards 5

3 Dependencies 7

4 Development 9

5 Relevant URLs 11

6 Licensing 13

Python Module Index 111

Index 113

The aiocoap package is an implementation of CoAP, the Constrained Application Protocol.

It is written in Python 3 using its native asyncio methods to facilitate concurrent operations while maintaining an easy to use interface.

aiocoap is originally based on txThings. If you want to use CoAP in your existing Twisted application, or can not migrate to Python 3 yet, that is probably more useful to you than aiocoap.

CHAPTER 1

Usage

For how to use the aiocoap library, have a look at the *Guided Tour through aiocoap*, or at the *Usage Examples* and *CoAP tools* provided.

A full reference is available in the *API documentation*.

All examples can be run directly from a source code copy. If you prefer to install it, the usual Python mechanisms apply (see *Installing aiocoap*).

CHAPTER 2

Features / Standards

This library supports the following standards in full or partially:

- RFC7252 (CoAP): Supported for clients and servers. Multicast is supported on the server side, and partially for clients. DTLS is supported but experimental, and lacking some security properties. No caching is done inside the library.
- RFC7641 (Observe): Basic support for clients and servers. Reordering, re-registration, and active cancellation are missing.
- RFC7959 (Blockwise): Supported both for atomic and random access.
- RFC8323 (TCP, WebSockets): Supports CoAP over TCP, TLS, and WebSockets (both over HTTP and HTTPS). The TLS parts are server-certificate only; preshared, raw public keys and client certificates are not supported yet.
- RFC7967 (No-Response): Supported.
- RFC8132 (PATCH/FETCH): Types and codes known, FETCH observation supported.
- RFC9176: A standalone resource directory server is provided along with a library function to register at one. They lack support for groups and security considerations, and are generally rather simplistic.
- RFC8613 (OSCORE): Full support client-side; protected servers can be implemented based on it but are not automatic yet.
- draft-ietf-core-oscore-groupcomm-11 (Group OSCORE): Supported for both group and pairwise mode in groups that are fully known. (The lack of an implemented joining or persistence mechanism makes this impractical for anything but experimentation.)

If something described by one of the standards but not implemented, it is considered a bug; please file at the github issue tracker. (If it's not on the list or in the excluded items, file a wishlist item at the same location).

CHAPTER 3

Dependencies

Basic aiocoap works out of the box on Python 3.7 or newer (also works on PyPy3). For full support (DTLS, OSCORE and link-format handling) follow the *Installing aiocoap* instructions as these require additional libraries.

aiocoap provides different network backends for different platforms. The most featureful backend is available for Linux, but most operations work on BSDs, Windows and macOS as well. See the *FAQ* for more details.

If your library depends on aiocoap, it should pick the required extras (as per *Installing aiocoap*) and declare a dependency like `aiocoap[linkheader,oscore] >= 0.4b2`.

CHAPTER 4

Development

aiocoap tries to stay close to PEP8 recommendations and general best practice, and should thus be easy to contribute to.

Bugs (ranging from "design goal" and "wishlist" to typos) are currently tracked in the github issue tracker. Pull requests are welcome there; if you start working on larger changes, please coordinate on the issue tracker.

Documentation is built using sphinx with `./setup.py build_sphinx`; hacks used there are described in `./doc/README.doc`.

Unit tests are implemented in the `./tests/` directory and easiest run using tox (though still available through `./setup.py test` for the time being); complete test coverage is aimed for, but not yet complete (and might never be, as the error handling for pathological network partners is hard to trigger with a library designed not to misbehave). The tests are regularly run at the CI suite at gitlab, from where coverage reports are available.

CHAPTER 5

Relevant URLs

- https://github.com/chrysn/aiocoap

  This is where the latest source code can be found, and bugs can be reported. Generally, this serves as the project web site.

- http://aiocoap.readthedocs.org/

  Online documentation built from the sources.

- http://coap.technology/

  Further general information on CoAP, the standard documents involved, and other implementations and tools available.

CHAPTER 6

Licensing

aiocoap is published under the MIT License, see *LICENSE* for details.

When using aiocoap for a publication, please cite it according to the output of `./setup.py cite [--bibtex]`.

6.1 Installing aiocoap

Note: The commands here will install aiocoap in your current environment. By default, that is your platform's user install directory.

To keep that clean, or to use different sets or versions of libraries for different purposes, you may want to look into the venv documentation, which explains both the concept of virtual environments and how they are used on different platforms.

In most situations, it is recommended to install the latest released version of aiocoap. This is done using a simple:

```
$ pip3 install --upgrade "aiocoap[all]"
```

(In some cases, the program is called `pip` only).

If `pip3` is not available on your platform, you can manually download and unpack the latest `.tar.gz` file from the Python package index and run

```
$ ./setup.py install
```

6.1.1 Development version

If you want to play with aiocoap's internals or consider contributing to the project, the suggested way of operation is getting a Git checkout of the project:

```
$ git clone https://github.com/chrysn/aiocoap
$ cd aiocoap
```

You can then use the project from that location, or install it with

```
$ pip3 install --upgrade ".[all,docs]"
```

If you need to install the latest development version of aiocoap but do not plan on editing (eg. because you were asked in the course of a bug report to test something against the latest aiocoap version), you can install it directly from the web:

```
$ pip3 install --upgrade "git+https://github.com/chrysn/aiocoap#egg=aiocoap[all]"
```

With the `-e` option, that is also a viable option if you want to modify aiocoap and pip's choice of checkout directories is suitable for you.

6.1.2 Common errors

When upstream libraries change, or when dependencies of used libraries are not there (eg. no C compiler, C libraries missing), the installation process can fail.

On Debian based systems, it helps to install the packages `python3-dev`, `build-essential` and `autoconf`; generally, the error output will contain some hints as to what is missing.

As a workaround, it can be helpful to not install with all extras, but replace the `all` with the extras you actually want from the list below. For example, if you see errors from DTLSSocket, rather than installing with `[all,docs]`, you can leave out the `tinydtls` extra and install with `[linkheader,oscore,prettyprint,docs]`.

6.1.3 Slimmer installations

As aiocoap does not strictly depend on many of the libraries that are installed when following the above recommendations, a setup can be stripped down by entering any combination of the below "extras" in the place of the `all` in the above lines, or leaving out the `[all]` expression for a minimal installation.

The extras currently supported are:

- `linkheader`: Needed for generating and parsing files in RFC6690 link format, eg. `.well-known/core` files. Running or interacting with a Resource Directory is impossible without this module, as are many other discovery steps that applications will want to do.
- `oscore`: Required for the *`aiocoap.transports.oscore`* transport.
- `tinydtls`: Required for using CoAP over DTLS.
- `ws`: Required for using CoAP over WebSockets.
- `prettyprint`: Allows using the `--color` and `--pretty-print` options of *aiocoap-client*.
- `docs`: Installs tools needed to build the documentation (not part of `all`).

Which libraries and versions are pulled in by this exactly is documented in the `setup.py` file.

6.2 Guided Tour through aiocoap

This page gets you started on the concepts used in aiocoap; it will assume rough familiarity with what CoAP is, and a working knowledge of Python development, but introduce you to asynchronous programming and explain some CoAP

concepts along with the aiocoap API.

If you are already familiar with asynchronous programming and/or some other concepts involved, or if you prefer reading code to reading tutorials, you might want to go after the *Usage Examples* instead.

6.2.1 First, some tools

Before we get into programming, let's establish tools with which we can probe a server, and a server itself. If you have not done it already, *install aiocoap for development*.

Start off with the sample server by running the following in a terminal inside the aiocoap directory:

```
$ ./server.py
```

Note: The `$` sign indicates the prompt; you enter everything after it in a terminal shell. Lines not starting with a dollar sign are the program output, if any. Later on, we'll see lines starting with `>>>`; those are run inside a Python interpreter.

I recommend that you use the IPython interpreter. One useful feature for following through this tutorial is that you can copy full lines (including any `>>>` parts) to the clipboard and use the `%paste` IPython command to run it, taking care of indentation etc.

This has started a CoAP server with some demo content, and keeps running until you terminate it with Ctrl-C.

In a separate terminal, use *the aiocoap-client tool* to send a GET request to the server:

```
$ ./aiocoap-client coap://localhost/.well-known/core
application/link-format content was re-formatted
</.well-known/core>; ct="40",
</time>; obs,
</other/block>,
</other/separate>; title="A large resource",
</whoami>,
<https://christian.amsuess.com/tools/aiocoap/#version-0.4.3.post0>; rel="impl-info"
```

The address we're using here is a resource on the local machine (`localhost`) at the well-known location `.well-known/core`, which in CoAP is the go-to location if you don't know anything about the paths on the server beforehand. It tells that there is a resource at the path `/time` that has the `obs`ervable attribute, a resource at the path `/.well-known/core`, and more at `/other/...` and `/whoami`.

Note: Getting "5.00 Internal Server Error" instead, all lines in a single row or no color? Then there are third party modules missing. Run `python3 -m aiocoap.cli.defaults` to see which they are, or just go back to the *installation step* and make sure to include the "`[all]`" part.

Note: There can be a "(No newline at end of message)" line below your output. This just makes sure your prompt does not start in the middle of the screen. I'll just ignore that.

Let's see what `/time` gives us:

```
$ ./aiocoap-client coap://localhost/time
2021-12-07 10:08
```

The response should have arrived immediately: The client sent a message to the server in which it requested the resource at `/time`, and the server could right away send a message back. In contrast, `/other/separate` is slower:

```
$ ./aiocoap-client coap://localhost/other/separate
Three rings for the elven kings [abbreviated]
```

The response to this message comes back with a delay. Here, it is simulated by the server; in real-life situations, this delay can stem from network latency, servers waiting for some sensor to read out a value, slow hard drives etc.

6.2.2 A request

In order to run a similar request programmatically, we'll need a request message.

```
>>> from aiocoap import *
>>> msg = Message(code=GET, uri="coap://localhost/other/separate")
>>> print(msg)
<aiocoap.Message at 0x0123deadbeef: no mtype, GET (no MID, empty token) remote None,␣
↪2 option(s)>
```

The message consists of several parts. The non-optional ones are largely handled by aiocoap (message type, ID, token and remote are all None or empty here and will be populated when the message is sent). The options are roughly equivalent to what you might know as HTTP headers:

```
>>> msg.opt
<aiocoap.options.Options at 0x0123deadbef0: URI_HOST: localhost, URI_PATH: other /␣
↪separate>
```

You might have noticed that the Uri-Path option is shown with some space around the slash. This is because paths in CoAP are not a structured byte string with slashes in it (as they are in HTTP), but actually repeated options of a (UTF-8) string, which are represented as a tuple in Python:

```
>>> msg.opt.uri_path
('other', 'separate')
```

Now to send that network as a request over the network, we'll need a network protocol object. That has a request method, and can give a response (**bear with me, these examples don't actually work**):

```
>>> protocol.request(msg).response
<Future pending cb=[Request._response_cancellation_handler()]>
```

That is obviously not a proper response – yet. If the protocol returned a finished response, the program couldn't do any work in the meantime. Instead, it returns a Future – an object that will (at some time in the *future*) contain the response. Because the Future is returned immediately, the user can start other requests in parallel, or do other processing in the meantime. For now, all we want is to wait until the response is ready:

```
>>> await protocol.request(msg).response
<aiocoap.Message at 0x0123deadbef1: Type.CON 2.05 Content (MID 51187, token 00008199)␣
↪remote <UDP6EndpointAddress [::ffff:127.0.0.1]:5683 with local address>, 186␣
↪byte(s) payload>
```

Here, we have a successful message ("2.05 Content" is the rough equivalent of HTTP's "200 OK", and the 186 bytes of payload look promising). Until we can dissect that, we'll have to get those asynchronous things to work properly, though.

6.2.3 Asynchronous operation

To work interactively with asynchronous Python, start your Python interpreter like this:

```
$ python3 -m asyncio
>>>
```

Users of the highly recommended IPython can continue in their existing session, as support for the asynchronous shell is always available there.

```
>>> protocol = await Context.create_client_context()
>>> msg = Message(code=GET, uri="coap://localhost/other/separate")
>>> response = await protocol.request(msg).response
>>> print(response)
<aiocoap.Message at 0x0123deadbef1: Type.CON 2.05 Content (MID 51187, token 00008199)␣
↪remote <UDP6EndpointAddress [::ffff:127.0.0.1]:5683 with local address>, 186␣
↪byte(s) payload>
```

That's better!

Now the `protocol` object could also be created – we need to start that once to prepare a socket for all the requests we're sending later. That doesn't actually take a long time, but could, depending on the operating system.

Note: If you want to pack any of the code into functions, these functions need to be asynchronous functions. When working in a `.py` file, the `await` keyword is not available outside, and you'll need to kick off your program using asyncio.run.

The same code as above packed up in a file would look like this:

```
import asyncio
from aiocoap import *

async def main():
    protocol = await Context.create_client_context()
    msg = Message(code=GET, uri="coap://localhost/other/separate")
    response = await protocol.request(msg).response
    print(response)

asyncio.run(main())
```

6.2.4 The response

The response obtained in the main function is a message like the request message, just that it has a different code (2.05 is of the successful 2.00 group), incidentally no options (because it's a very simple server), and actual data.

The response code is represented in Python by an enum with some utility functions; the remote address (actually remote-local address pair) is an object too:

```
>>> response.code
<Successful Response Code 69 "2.05 Content">
>>> response.code.is_successful()
True
>>> response.remote.hostinfo
'[::ffff:127.0.0.1]'
```

(continues on next page)

(continued from previous page)

```
>>> response.remote.is_multicast
False
```

The actual response message, the body, or the payload of the response, is accessible in the payload property, and is always a bytestring:

```
>>> response.payload
b'Three rings for the elven kings [ abbreviated ]'
```

aiocoap does not yet provide utilities to parse the message according to its content format (which would be accessed as `response.opt.content_format`).

More asynchronous fun

The other examples don't show simultaneous requests in flight, so let's have one with parallel requests:

```
>>> async def main():
...     responses = [
...         protocol.request(Message(code=GET, uri=u)).response
...         for u
...         in ("coap://localhost/time", "coap://vs0.inf.ethz.ch/obs",
↪"coap://coap.me/test")
...     ]
...     for f in asyncio.as_completed(responses):
...         response = await f
...         print("Response from {}: {}".format(response.get_request_uri(),
↪ response.payload))
>>> run(main())
Response from coap://localhost/time: b'2016-12-07 18:16'
Response from coap://vs0.inf.ethz.ch/obs: b'18:16:11'
Response from coap://coap.me/test: b'welcome to the ETSI plugtest! last␣
↪change: 2016-12-06 16:02:33 UTC'
```

This also shows that the response messages do keep some information of their original request (in particular, the request URI) with them to ease further parsing.

This is currently the end of the guided tour; see the *aiocoap.resource* documentation for the server side until the tour covers that is complete.

6.3 OSCORE in aiocoap

6.3.1 Introducing OSCORE

OSCORE (RFC8613) is an end-to-end security mechanism available for CoAP and implemented in aiocoap.

Its main advantage over lower-layer protection (IPsec, (D)TLS) is that it can leverage any CoAP transport (as well as HTTP), can traverse proxies preserving some of their features (like block-wise fragmentation and retransmission) and supports multicast and other group communication scenarios (implemented, but not covered here as it needs even more manual actions so far).

By itself, OSCORE has no key exchange protocol; it relies on other protocols to establidsh keys (there is ongoing work on a lightweight key exchange named EDHOC, and the ACE-OSCORE profile goes some way). Until those are implemented and wide-spread, OSCORE contexts can be provisioned manually to devices.

6.3.2 OSCORE state

Unless an add-on mode (sometimes called B2 mode as it's describe in OSCORE's Appendix B.2) is used, some run-time information needs to be stored along with an OSCORE key.

This allows instantaneous zero-round-trip trusted requests with just a single round-trip (ie. a client can shut down, wake up with a different network address, and still the first UDP package it sends to the server can be relied and acted upon immediately). In this mode, there is no need for the device to have a reliable source of entropy.

In practice, this means that OSCORE keys need to reside in writable directories, are occasionally written to (the mechanisms of Appendix B.1 ensure that writes are rare: they happen at startup, shutdown, and only occasionally at runtime).

> **Warning:** This also means that stored OSCORE contexts must never be copied, only moved (or have the original deleted right after a copy).
>
> Where copies are unavoidable (eg. as part of a system backup), they must not be used unless it can be proven that the original was not written to at all after the backup was taken.
>
> When that can not be proven, the context must be deemed lost and reestablished by different means.

6.3.3 OSCORE credentials

As an experimental format, OSCORE uses JSON based credentials files that describes OSCORE or (D)TLS credentials.

For client, they indicate which URIs should be accessed using which OSCORE context. For servers, they indicate the available OSCORE contexts clients could use, and provide labels for them.

The keys and IDs themselves are stored in a directory referenced by the credentials file; this allows the state writes to be performed independently.

6.3.4 OSCORE example

This example sets up encrypted access to the file server demo from the generic command line client.

Note: Manual provisioning of OSCORE contexts is not expected to be a long-term solution, and meant primarily for initial experimentation.

Do not expect the security contexts set up here to be usable indefinitely, as the credentials and security context format used by aiocoap is still in flux. Moreover, the expample will change over time to reflect the best use of OSCORE possible with the current implementation.

First, create a pair of security contexts:

`client1/for-fileserver/settings.json`:

```
{
  "sender-id_hex": "01",
  "recipient-id_ascii": "file",

  "secret_ascii": "Correct Horse Battery Staple"
}
```

`server/from-client1/settings.json`:

```
{
  "recipient-id_hex": "01",
  "sender-id_ascii": "file",

  "secret_ascii": "Correct Horse Battery Staple"
}
```

A single secret must only be used once – please use something more unique than the standard passphrase.

With each of those goes a credentials map:

`client1.json`:

```
{
  "coap://localhost/*": { "oscore": { "contextfile": "client1/for-fileserver/" } }
}
```

`server.json`:

```
{
  ":client1": { "oscore": { "contextfile": "server/from-client1/" } }
}
```

Then, the server can be started:

```
$ ./aiocoap-fileserver data-to-be-served/ --credentials server.json
```

And queried using the client:

```
$ ./aiocoap-client coap://localhost/ --credentials client1.json
<subdirectory/>; ct="40",
<other-directory/>; ct="40",
<README>
```

Note that just passing in those credentials does not on its own make the server require encrypted communication, let alone require authorization. Requests without credentials still work, and in this very example it'd need a network sniffer (or increased verbosity) to even be sure *that* the request was protected.

Ways of implementing access controls, mandatory encryption and access control are being explored - as are extensions that simplify the setup process.

6.4 The aiocoap API

This is about the Python API of the aiocoap library; see *CoAP API design notes* for notes on how CoAP concepts play into the API.

6.4.1 API stability

In preparation for a semantically versioned 1.0 release, some parts of aiocoap are described as stable.

The library does not try to map the distinction between "public API" and internal components in the sense of semantic versioning to Python's "public" and "private" (_-prefixed) interaces – tying those together would mean intrusive refactoring every time a previously internal mechanism is stabilized.

Neither does it only document the public API, as that would mean that library development would need to resort to working with code comments; that would also impede experimentation, and migrating comments to docstrings would be intrusive again. All modules' documentation can be searched, and most modules are listed below.

Instead, functions, methods and properties in the library should only be considered public (in the semantic versioning sense) if they are described as "stable" in their documentation. The documentation may limit how an interface may used or what can be expected from it. (For example, while a method may be typed to return a particular class, the stable API may only guarantee that an instance of a particular abstract base class is returned).

The `__all__` attributes of aiocoap modules try to represent semantic publicality of its members (in accordance with PEP8); however, the documentation is the authoritative source.

6.4.2 Modules with stable components

aiocoap module

The aiocoap package is a library that implements CoAP, the Constrained Application Protocol.

If you are reading this through the Python documentation, be aware that there is additional documentation available online and in the source code's `doc` directory.

Module contents

This root module re-exports the most commonly used classes in aiocoap: *`Context`*, *`Message`* as well as all commonly used numeric constants from *`numbers`*; see their respective documentation entries.

The presence of *`Message`* and *`Context`* in the root module is stable.

class aiocoap.**Type**

Bases: `enum.IntEnum`

An enumeration.

CON = 0

NON = 1

ACK = 2

RST = 3

class aiocoap.**Code**

Bases: *`aiocoap.util.ExtensibleIntEnum`*

Value for the CoAP "Code" field.

As the number range for the code values is separated, the rough meaning of a code can be determined using the *`is_request()`*, *`is_response()`* and *`is_successful()`* methods.

EMPTY = <Code 0 "EMPTY">

GET = <Request Code 1 "GET">

POST = <Request Code 2 "POST">

PUT = <Request Code 3 "PUT">

DELETE = <Request Code 4 "DELETE">

FETCH = <Request Code 5 "FETCH">

PATCH = <Request Code 6 "PATCH">

iPATCH = <Request Code 7 "iPATCH">

CREATED = <Successful Response Code 65 "2.01 Created">

DELETED = <Successful Response Code 66 "2.02 Deleted">

VALID = <Successful Response Code 67 "2.03 Valid">

CHANGED = <Successful Response Code 68 "2.04 Changed">

CONTENT = <Successful Response Code 69 "2.05 Content">

CONTINUE = <Successful Response Code 95 "2.31 Continue">

BAD_REQUEST = <Response Code 128 "4.00 Bad Request">

UNAUTHORIZED = <Response Code 129 "4.01 Unauthorized">

BAD_OPTION = <Response Code 130 "4.02 Bad Option">

FORBIDDEN = <Response Code 131 "4.03 Forbidden">

NOT_FOUND = <Response Code 132 "4.04 Not Found">

METHOD_NOT_ALLOWED = <Response Code 133 "4.05 Method Not Allowed">

NOT_ACCEPTABLE = <Response Code 134 "4.06 Not Acceptable">

REQUEST_ENTITY_INCOMPLETE = <Response Code 136 "4.08 Request Entity Incomplete">

CONFLICT = <Response Code 137 "4.09 Conflict">

PRECONDITION_FAILED = <Response Code 140 "4.12 Precondition Failed">

REQUEST_ENTITY_TOO_LARGE = <Response Code 141 "4.13 Request Entity Too Large">

UNSUPPORTED_CONTENT_FORMAT = <Response Code 143 "4.15 Unsupported Content Format">

UNSUPPORTED_MEDIA_TYPE

UNPROCESSABLE_ENTITY = <Response Code 150 "4.22 Unprocessable Entity">

TOO_MANY_REQUESTS = <Response Code 157 "4.29 Too Many Requests">

INTERNAL_SERVER_ERROR = <Response Code 160 "5.00 Internal Server Error">

NOT_IMPLEMENTED = <Response Code 161 "5.01 Not Implemented">

BAD_GATEWAY = <Response Code 162 "5.02 Bad Gateway">

SERVICE_UNAVAILABLE = <Response Code 163 "5.03 Service Unavailable">

GATEWAY_TIMEOUT = <Response Code 164 "5.04 Gateway Timeout">

PROXYING_NOT_SUPPORTED = <Response Code 165 "5.05 Proxying Not Supported">

HOP_LIMIT_REACHED = <Response Code 168 "5.08 Hop Limit Reached">

CSM = <Code 225 "7.01 Csm">

PING = <Code 226 "7.02 Ping">

PONG = <Code 227 "7.03 Pong">

RELEASE = <Code 228 "7.04 Release">

ABORT = <Code 229 "7.05 Abort">

is_request()

True if the code is in the request code range

is_response()
True if the code is in the response code range

is_signalling()

is_successful()
True if the code is in the successful subrange of the response code range

can_have_payload()
True if a message with that code can carry a payload. This is not checked for strictly, but used as an indicator.

class_
The class of a code (distinguishing whether it's successful, a request or a response error or more).

```
>>> Code.CONTENT
<Successful Response Code 69 "2.05 Content">
>>> Code.CONTENT.class_
2
>>> Code.BAD_GATEWAY
<Response Code 162 "5.02 Bad Gateway">
>>> Code.BAD_GATEWAY.class_
5
```

dotted
The numeric value three-decimal-digits (c.dd) form

name_printable
The name of the code in human-readable form

name
The constant name of the code (equals name_printable readable in all-caps and with underscores)

class aiocoap.**OptionNumber**
Bases: *aiocoap.util.ExtensibleIntEnum*

A CoAP option number.

As the option number contains information on whether the option is critical, and whether it is safe-to-forward, those properties can be queried using the *is_** group of methods.

Note that whether an option may be repeated or not does not only depend on the option, but also on the context, and is thus handled in the *Options* object instead.

IF_MATCH = <OptionNumber 1 "IF_MATCH">

URI_HOST = <OptionNumber 3 "URI_HOST">

ETAG = <OptionNumber 4 "ETAG">

IF_NONE_MATCH = <OptionNumber 5 "IF_NONE_MATCH">

OBSERVE = <OptionNumber 6 "OBSERVE">

URI_PORT = <OptionNumber 7 "URI_PORT">

LOCATION_PATH = <OptionNumber 8 "LOCATION_PATH">

OSCORE = <OptionNumber 9 "OBJECT_SECURITY">

OBJECT_SECURITY = <OptionNumber 9 "OBJECT_SECURITY">

URI_PATH = <OptionNumber 11 "URI_PATH">

CONTENT_FORMAT = <OptionNumber 12 "CONTENT_FORMAT">

MAX_AGE = <OptionNumber 14 "MAX_AGE">

URI_QUERY = <OptionNumber 15 "URI_QUERY">

HOP_LIMIT = <OptionNumber 16 "HOP_LIMIT">

ACCEPT = <OptionNumber 17 "ACCEPT">

Q_BLOCK1 = <OptionNumber 19 "Q_BLOCK1">

LOCATION_QUERY = <OptionNumber 20 "LOCATION_QUERY">

EDHOC = <OptionNumber 21 "EDHOC">

BLOCK2 = <OptionNumber 23 "BLOCK2">

BLOCK1 = <OptionNumber 27 "BLOCK1">

SIZE2 = <OptionNumber 28 "SIZE2">

Q_BLOCK2 = <OptionNumber 31 "Q_BLOCK2">

PROXY_URI = <OptionNumber 35 "PROXY_URI">

PROXY_SCHEME = <OptionNumber 39 "PROXY_SCHEME">

SIZE1 = <OptionNumber 60 "SIZE1">

ECHO = <OptionNumber 252 "ECHO">

NO_RESPONSE = <OptionNumber 258 "NO_RESPONSE">

REQUEST_TAG = <OptionNumber 292 "REQUEST_TAG">

REQUEST_HASH = <OptionNumber 548 "REQUEST_HASH">

is_critical()

is_elective()

is_unsafe()

is_safetoforward()

is_nocachekey()

is_cachekey()

format

create_option(*decode=None*, *value=None*)

Return an Option element of the appropriate class from this option number.

An initial value may be set using the decode or value options, and will be fed to the resulting object's decode method or value property, respectively.

class aiocoap.**ContentFormat**

Bases: *aiocoap.util.ExtensibleIntEnum*

Entry in the CoAP Content-Formats registry of the IANA Constrained RESTful Environments (Core) Parameters group

Known entries have `.media_type` and `.encoding` attributes:

```
>>> ContentFormat(0).media_type
'text/plain; charset=utf-8'
>>> int(ContentFormat.by_media_type('text/plain;charset=utf-8'))
0
```

(continues on next page)

(continued from previous page)

```
>>> ContentFormat(60)
<ContentFormat 60, media_type='application/cbor', encoding='identity'>
>>> ContentFormat(11060).encoding
'deflate'
```

Unknown entries do not have these properties:

```
>>> ContentFormat(12345).is_known()
False
>>> ContentFormat(12345).media_type                    # doctest: +ELLIPSIS
Traceback (most recent call last):
    ...
AttributeError: ...
```

Only a few formats are available as attributes for easy access. Their selection and naming are arbitrary and biased. The remaining known types are available through the *by_media_type()* class method. >>> ContentFormat.TEXT <ContentFormat 0, media_type='text/plain; charset=utf-8', encoding='identity'>

A convenient property of ContentFormat is that any known content format is true in a boolean context, and thus when used in alternation with None, can be assigned defaults easily:

```
>>> requested_by_client = ContentFormat.TEXT
>>> int(requested_by_client) # Usually, this would always pick the default
0
>>> used = requested_by_client or ContentFormat.LINKFORMAT
>>> assert used == ContentFormat.TEXT
```

classmethod by_media_type(*media_type: str*, *encoding: str = 'identity'*) → aiocoap.numbers.contentformat.ContentFormat

Produce known entry for a known media type (and encoding, though 'identity' is default due to its prevalence), or raise KeyError.

is_known()

TEXT = <ContentFormat 0, media_type='text/plain; charset=utf-8', encoding='identity'>

LINKFORMAT = <ContentFormat 40, media_type='application/link-format', encoding='identi

OCTETSTREAM = <ContentFormat 42, media_type='application/octet-stream', encoding='iden

JSON = <ContentFormat 50, media_type='application/json', encoding='identity'>

CBOR = <ContentFormat 60, media_type='application/cbor', encoding='identity'>

SENML = <ContentFormat 112, media_type='application/senml+cbor', encoding='identity'>

class aiocoap.**Message**(*, *mtype=None*, *mid=None*, *code=None*, *payload=b''*, *token=b''*, *uri=None*, ***kwargs*)

Bases: object

CoAP Message with some handling metadata

This object's attributes provide access to the fields in a CoAP message and can be directly manipulated.

- Some attributes are additional data that do not round-trip through serialization and deserialization. They are marked as "non-roundtrippable".
- Some attributes that need to be filled for submission of the message can be left empty by most applications, and will be taken care of by the library. Those are marked as "managed".

The attributes are:

- payload: The payload (body) of the message as bytes.
- mtype: Message type (CON, ACK etc, see *numbers.types*). Managed unless set by the application.
- code: The code (either request or response code), see *numbers.codes*.
- opt: A container for the options, see *options.Options*.
- mid: The message ID. Managed by the *Context*.
- token: The message's token as bytes. Managed by the *Context*.
- remote: The socket address of the other side, managed by the *protocol.Request* by resolving the .opt.uri_host or unresolved_remote, or the Responder by echoing the incoming request's. Follows the *interfaces.EndpointAddress* interface. Non-roundtrippable.

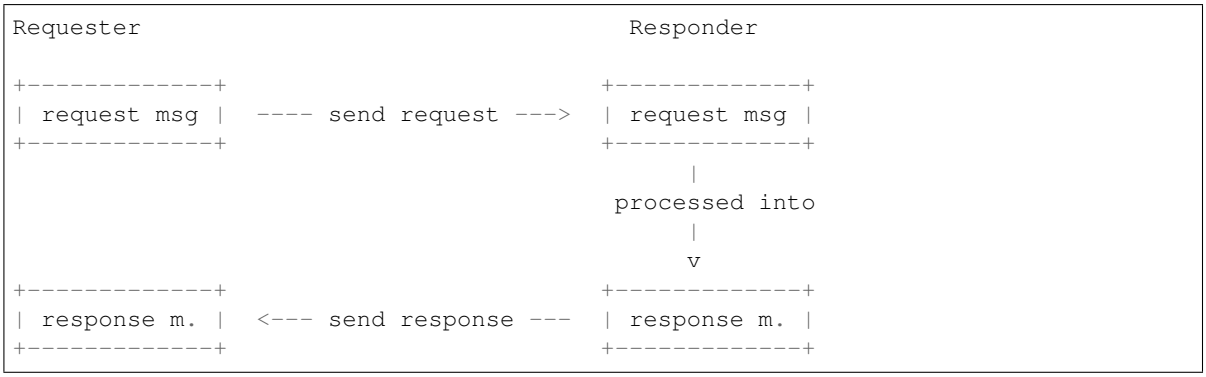
  While a message has not been transmitted, the property is managed by the *Message* itself using the *set_request_uri()* or the constructor *uri* argument.
- request: The request to which an incoming response message belongs; only available at the client. Managed by the *interfaces.RequestProvider* (typically a *Context*).

These properties are still available but deprecated:

- requested_*: Managed by the *protocol.Request* a response results from, and filled with the request's URL data. Non-roundtrippable.
- unresolved_remote: host[:port] (strictly speaking; hostinfo as in a URI) formatted string. If this attribute is set, it overrides .RequestManageropt.uri_host (and -_port) when it comes to filling the remote in an outgoing request.

  Use this when you want to send a request with a host name that would not normally resolve to the destination address. (Typically, this is used for proxying.)

Options can be given as further keyword arguments at message construction time. This feature is experimental, as future message parameters could collide with options.



The above shows the four message instances involved in communication between an aiocoap client and server process. Boxes represent instances of Message, and the messages on the same line represent a single CoAP as passed around on the network. Still, they differ in some aspects:

- The requested URI will look different between requester and responder if the requester uses a host name and does not send it in the message.
- If the request was sent via multicast, the response's requested URI differs from the request URI because it has the responder's address filled in. That address is not known at the responder's side yet, as it is typically filled out by the network stack.
- It is yet unclear whether the response's URI should contain an IP literal or a host name in the unicast case if the Uri-Host option was not sent.

- Properties like Message ID and token will differ if a proxy was involved.
- Some options or even the payload may differ if a proxy was involved.

copy(***kwargs*)
Create a copy of the Message. kwargs are treated like the named arguments in the constructor, and update the copy.

classmethod decode(*rawdata*, *remote=None*)
Create Message object from binary representation of message.

encode()
Create binary representation of message from Message object.

get_cache_key(*ignore_options=()*)
Generate a hashable and comparable object (currently a tuple) from the message's code and all option values that are part of the cache key and not in the optional list of ignore_options (which is the list of option numbers that are not technically NoCacheKey but handled by the application using this method).

```
>>> from aiocoap.numbers import GET
>>> m1 = Message(code=GET)
>>> m2 = Message(code=GET)
>>> m1.opt.uri_path = ('s', '1')
>>> m2.opt.uri_path = ('s', '1')
>>> m1.opt.size1 = 10 # the only no-cache-key option in the base spec
>>> m2.opt.size1 = 20
>>> m1.get_cache_key() == m2.get_cache_key()
True
>>> m2.opt.etag = b'000'
>>> m1.get_cache_key() == m2.get_cache_key()
False
>>> from aiocoap.numbers.optionnumbers import OptionNumber
>>> ignore = [OptionNumber.ETAG]
>>> m1.get_cache_key(ignore) == m2.get_cache_key(ignore)
True
```

get_request_uri(**, local_is_server=False*)
The absolute URI this message belongs to.

For requests, this is composed from the options (falling back to the remote). For responses, this is largely taken from the original request message (so far, that could have been trackecd by the requesting application as well), but – in case of a multicast request – with the host replaced by the responder's endpoint details.

This implements Section 6.5 of RFC7252.

By default, these values are only valid on the client. To determine a message's request URI on the server, set the local_is_server argument to True. Note that determining the request URI on the server is brittle when behind a reverse proxy, may not be possible on all platforms, and can only be applied to a request message in a renderer (for the response message created by the renderer will only be populated when it gets transmitted; simple manual copying of the request's remote to the response will not magically make this work, for in the very case where the request and response's URIs differ, that would not catch the difference and still report the multicast address, while the actual sending address will only be populated by the operating system later).

set_request_uri(*uri*, ***, *set_uri_host=True*)
Parse a given URI into the uri_* fields of the options.

The remote does not get set automatically; instead, the remote data is stored in the uri_host and uri_port options. That is because name resolution is coupled with network specifics the protocol will know better

by the time the message is sent. Whatever sends the message, be it the protocol itself, a proxy wrapper or an alternative transport, will know how to handle the information correctly.

When `set_uri_host=False` is passed, the host/port is stored in the `unresolved_remote` message property instead of the uri_host option; as a result, the unresolved host name is not sent on the wire, which breaks virtual hosts but makes message sizes smaller.

This implements Section 6.4 of RFC7252.

unresolved_remote

requested_scheme

requested_proxy_uri

requested_hostinfo

requested_path

requested_query

class aiocoap.**Context**(*loop=None*, *serversite=None*, *loggername='coap'*, *client_credentials=None*, *server_credentials=None*)

Bases: *aiocoap.interfaces.RequestProvider*

Applications' entry point to the network

A *Context* coordinates one or more network *transports* implementations and dispatches data between them and the application.

The application can start requests using the message dispatch methods, and set a `resources.Site` that will answer requests directed to the application as a server.

On the library-internals side, it is the prime implementation of the *interfaces.RequestProvider* interface, creates `Request` and `Response` classes on demand, and decides which transport implementations to start and which are to handle which messages.

Context creation and destruction

The following functions are provided for creating and stopping a context:

Note: A typical application should only ever create one context, even (or especially when) it acts both as a server and as a client (in which case a server context should be created).

A context that is not used any more must be shut down using *shutdown()*, but typical applications will not need to because they use the context for the full process lifetime.

---

classmethod create_client_context(**, loggername='coap', loop=None, transports: Optional[List[str]] = None*)

Create a context bound to all addresses on a random listening port.

This is the easiest way to get a context suitable for sending client requests.

classmethod create_server_context(*site, bind=None, *, loggername='coap-server', loop=None, _ssl_context=None, multicast=[], server_credentials=None, transports: Optional[List[str]] = None*)

Create a context, bound to all addresses on the CoAP port (unless otherwise specified in the `bind` argument).

This is the easiest way to get a context suitable both for sending client and accepting server requests.

The `bind` argument, if given, needs to be a 2-tuple of IP address string and port number, where the port number can be None to use the default port.

If `multicast` is given, it needs to be a list of (multicast address, interface name) tuples, which will all be joined. (The IPv4 style of selecting the interface by a local address is not supported; users may want to use the netifaces package to arrive at an interface name for an address).

As a shortcut, the list may also contain interface names alone. Those will be joined for the 'all CoAP nodes' groups of IPv4 and IPv6 (with scopes 2 and 5) as well as the respective 'all nodes' groups in IPv6.

Under some circumstances you may already need a context to pass into the site for creation; this is typically the case for servers that trigger requests on their own. For those cases, it is usually easiest to pass None in as a site, and set the fully constructed site later by assigning to the `serversite` attribute.

shutdown()

Take down any listening sockets and stop all related timers.

After this coroutine terminates, and once all external references to the object are dropped, it should be garbage-collectable.

This method takes up to *aiocoap.numbers.constants.SHUTDOWN_TIMEOUT* seconds, allowing transports to perform any cleanup implemented in them (such as orderly connection shutdown and cancelling observations, where the latter is currently not implemented).

Dispatching messages

CoAP requests can be sent using the following functions:

request(*request_message*, *handle_blockwise=True*)

Create and act on a a `Request` object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

If more control is needed, you can create a `Request` yourself and pass the context to it.

Other methods and properties

The remaining methods and properties are to be considered unstable even when the project reaches a stable version number; please file a feature request for stabilization if you want to reliably access any of them.

(Sorry for the duplicates, still looking for a way to make autodoc list everything not already mentioned).

request(*request_message*, *handle_blockwise=True*)

Create and act on a a `Request` object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

render_to_pipe(*pipe*)

Fill a pipe by running the site's render_to_pipe interface and handling errors.

classmethod create_client_context(*, *loggername='coap'*, *loop=None*, *transports: Optional[List[str]] = None*)

Create a context bound to all addresses on a random listening port.

This is the easiest way to get a context suitable for sending client requests.

classmethod create_server_context(*site*, *bind=None*, *, *loggername='coap-server'*, *loop=None*, *_ssl_context=None*, *multicast=[]*, *server_credentials=None*, *transports: Optional[List[str]] = None*)

Create a context, bound to all addresses on the CoAP port (unless otherwise specified in the `bind` argument).

This is the easiest way to get a context suitable both for sending client and accepting server requests.

The `bind` argument, if given, needs to be a 2-tuple of IP address string and port number, where the port number can be None to use the default port.

If `multicast` is given, it needs to be a list of (multicast address, interface name) tuples, which will all be joined. (The IPv4 style of selecting the interface by a local address is not supported; users may want to use the netifaces package to arrive at an interface name for an address).

As a shortcut, the list may also contain interface names alone. Those will be joined for the 'all CoAP nodes' groups of IPv4 and IPv6 (with scopes 2 and 5) as well as the respective 'all nodes' groups in IPv6.

Under some circumstances you may already need a context to pass into the site for creation; this is typically the case for servers that trigger requests on their own. For those cases, it is usually easiest to pass None in as a site, and set the fully constructed site later by assigning to the `serversite` attribute.

find_remote_and_interface(*message*)

shutdown()

Take down any listening sockets and stop all related timers.

After this coroutine terminates, and once all external references to the object are dropped, it should be garbage-collectable.

This method takes up to *aiocoap.numbers.constants.SHUTDOWN_TIMEOUT* seconds, allowing transports to perform any cleanup implemented in them (such as orderly connection shutdown and cancelling observations, where the latter is currently not implemented).

aiocoap.protocol module

This module contains the classes that are responsible for keeping track of messages:

- *Context* roughly represents the CoAP endpoint (basically a UDP socket) – something that can send requests and possibly can answer incoming requests.
- a *Request* gets generated whenever a request gets sent to keep track of the response
- a `Responder` keeps track of a single incoming request

Logging

Several constructors of the Context accept a logger name; these names go into the construction of a Python logger.

Log events will be emitted to these on different levels, with "warning" and above being a practical default for things that should may warrant reviewing by an operator:

- DEBUG is used for things that occur even under perfect conditions.
- INFO is for things that are well expected, but might be interesting during testing a network of nodes and not just when debugging the library. (This includes timeouts, retransmissions, and pings.)
- WARNING is for everything that indicates a malbehaved peer. These don't *necessarily* indicate a client bug, though: Things like requesting a nonexistent block can just as well happen when a resource's content has changed between blocks. The library will not go out of its way to determine whether there is a plausible explanation for the odd behavior, and will report something as a warning in case of doubt.
- ERROR is used when something clearly went wrong. This includes irregular connection terminations and resource handler errors (which are demoted to error responses), and can often contain a backtrace.

class aiocoap.protocol.**Context**(*loop=None*, *serversite=None*, *loggername='coap'*, *client_credentials=None*, *server_credentials=None*)

Bases: *aiocoap.interfaces.RequestProvider*

Applications' entry point to the network

A *Context* coordinates one or more network *transports* implementations and dispatches data between them and the application.

The application can start requests using the message dispatch methods, and set a `resources.Site` that will answer requests directed to the application as a server.

On the library-internals side, it is the prime implementation of the `interfaces.RequestProvider` interface, creates *Request* and `Response` classes on demand, and decides which transport implementations to start and which are to handle which messages.

Context creation and destruction

The following functions are provided for creating and stopping a context:

Note: A typical application should only ever create one context, even (or especially when) it acts both as a server and as a client (in which case a server context should be created).

A context that is not used any more must be shut down using *shutdown()*, but typical applications will not need to because they use the context for the full process lifetime.

---

classmethod create_client_context(*, *loggername='coap'*, *loop=None*, *transports: Optional[List[str]] = None*)

Create a context bound to all addresses on a random listening port.

This is the easiest way to get a context suitable for sending client requests.

classmethod create_server_context(*site*, *bind=None*, *, *loggername='coap-server'*, *loop=None*, *_ssl_context=None*, *multicast=[]*, *server_credentials=None*, *transports: Optional[List[str]] = None*)

Create a context, bound to all addresses on the CoAP port (unless otherwise specified in the `bind` argument).

This is the easiest way to get a context suitable both for sending client and accepting server requests.

The `bind` argument, if given, needs to be a 2-tuple of IP address string and port number, where the port number can be None to use the default port.

If `multicast` is given, it needs to be a list of (multicast address, interface name) tuples, which will all be joined. (The IPv4 style of selecting the interface by a local address is not supported; users may want to use the netifaces package to arrive at an interface name for an address).

As a shortcut, the list may also contain interface names alone. Those will be joined for the 'all CoAP nodes' groups of IPv4 and IPv6 (with scopes 2 and 5) as well as the respective 'all nodes' groups in IPv6.

Under some circumstances you may already need a context to pass into the site for creation; this is typically the case for servers that trigger requests on their own. For those cases, it is usually easiest to pass None in as a site, and set the fully constructed site later by assigning to the `serversite` attribute.

shutdown()

Take down any listening sockets and stop all related timers.

After this coroutine terminates, and once all external references to the object are dropped, it should be garbage-collectable.

This method takes up to *aiocoap.numbers.constants.SHUTDOWN_TIMEOUT* seconds, allowing transports to perform any cleanup implemented in them (such as orderly connection shutdown and cancelling observations, where the latter is currently not implemented).

Dispatching messages

CoAP requests can be sent using the following functions:

request(*request_message*, *handle_blockwise=True*)
Create and act on a a `Request` object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

If more control is needed, you can create a `Request` yourself and pass the context to it.

Other methods and properties

The remaining methods and properties are to be considered unstable even when the project reaches a stable version number; please file a feature request for stabilization if you want to reliably access any of them.

(Sorry for the duplicates, still looking for a way to make autodoc list everything not already mentioned).

request(*request_message*, *handle_blockwise=True*)
Create and act on a a `Request` object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

render_to_pipe(*pipe*)
Fill a pipe by running the site's render_to_pipe interface and handling errors.

classmethod create_client_context(*, *loggername='coap'*, *loop=None*, *transports: Optional[List[str]] = None*)
Create a context bound to all addresses on a random listening port.

This is the easiest way to get a context suitable for sending client requests.

classmethod create_server_context(*site*, *bind=None*, *, *loggername='coap-server'*, *loop=None*, *_ssl_context=None*, *multicast=[]*, *server_credentials=None*, *transports: Optional[List[str]] = None*)
Create a context, bound to all addresses on the CoAP port (unless otherwise specified in the `bind` argument).

This is the easiest way to get a context suitable both for sending client and accepting server requests.

The `bind` argument, if given, needs to be a 2-tuple of IP address string and port number, where the port number can be None to use the default port.

If `multicast` is given, it needs to be a list of (multicast address, interface name) tuples, which will all be joined. (The IPv4 style of selecting the interface by a local address is not supported; users may want to use the netifaces package to arrive at an interface name for an address).

As a shortcut, the list may also contain interface names alone. Those will be joined for the 'all CoAP nodes' groups of IPv4 and IPv6 (with scopes 2 and 5) as well as the respective 'all nodes' groups in IPv6.

Under some circumstances you may already need a context to pass into the site for creation; this is typically the case for servers that trigger requests on their own. For those cases, it is usually easiest to pass None in as a site, and set the fully constructed site later by assigning to the `serversite` attribute.

find_remote_and_interface(*message*)

shutdown()
Take down any listening sockets and stop all related timers.

After this coroutine terminates, and once all external references to the object are dropped, it should be garbage-collectable.

This method takes up to *aiocoap.numbers.constants.SHUTDOWN_TIMEOUT* seconds, allowing transports to perform any cleanup implemented in them (such as orderly connection shutdown and cancelling observations, where the latter is currently not implemented).

class aiocoap.protocol.**BaseRequest**
Bases: object

Common mechanisms of *Request* and MulticastRequest

class aiocoap.protocol.**BaseUnicastRequest**
Bases: *aiocoap.protocol.BaseRequest*

A utility class that offers the *response_raising* and *response_nonraising* alternatives to waiting for the response future whose error states can be presented either as an unsuccessful response (eg. 4.04) or an exception.

It also provides some internal tools for handling anything that has a response future and an observation

response_nonraising
An awaitable that rather returns a 500ish fabricated message (as a proxy would return) instead of raising an exception.

Experimental Interface.

response_raising
An awaitable that returns if a response comes in and is successful, otherwise raises generic network exception or a *error.ResponseWrappingError* for unsuccessful responses.

Experimental Interface.

class aiocoap.protocol.**Request**(*pipe*, *loop*, *log*)
Bases: *aiocoap.interfaces.Request*, *aiocoap.protocol.BaseUnicastRequest*

class aiocoap.protocol.**BlockwiseRequest**(*protocol*, *app_request*)
Bases: *aiocoap.protocol.BaseUnicastRequest*, *aiocoap.interfaces.Request*

class aiocoap.protocol.**ClientObservation**
Bases: object

An interface to observe notification updates arriving on a request.

This class does not actually provide any of the observe functionality, it is purely a container for dispatching the messages via callbacks or asynchronous iteration. It gets driven (ie. populated with responses or errors including observation termination) by a Request object.

register_callback(*callback*)
Call the callback whenever a response to the message comes in, and pass the response to it.

register_errback(*callback*)
Call the callback whenever something goes wrong with the observation, and pass an exception to the callback. After such a callback is called, no more callbacks will be issued.

callback(*response*)
Notify all listeners of an incoming response

error(*exception*)
Notify registered listeners that the observation went wrong. This can only be called once.

cancel()
Cease to generate observation or error events. This will not generate an error by itself.

on_cancel(*callback*)

class aiocoap.protocol.**ServerObservation**

Bases: object

accept(*cancellation_callback*)

deregister(*reason=None*)

trigger(*response=None*, *, *is_last=False*)

Send an updated response; if None is given, the observed resource's rendering will be invoked to produce one.

is_last can be set to True to indicate that no more responses will be sent. Note that an unsuccessful response will be the last no matter what is_last says, as such a message always terminates a CoAP observation.

aiocoap.message module

class aiocoap.message.**Message**(*, *mtype=None*, *mid=None*, *code=None*, *payload=b''*, *token=b''*, *uri=None*, **kwargs*)

Bases: object

CoAP Message with some handling metadata

This object's attributes provide access to the fields in a CoAP message and can be directly manipulated.

- Some attributes are additional data that do not round-trip through serialization and deserialization. They are marked as "non-roundtrippable".
- Some attributes that need to be filled for submission of the message can be left empty by most applications, and will be taken care of by the library. Those are marked as "managed".

The attributes are:

- `payload`: The payload (body) of the message as bytes.
- `mtype`: Message type (CON, ACK etc, see *numbers.types*). Managed unless set by the application.
- `code`: The code (either request or response code), see *numbers.codes*.
- `opt`: A container for the options, see *options.Options*.
- `mid`: The message ID. Managed by the *Context*.
- `token`: The message's token as bytes. Managed by the *Context*.
- `remote`: The socket address of the other side, managed by the *protocol.Request* by resolving the `.opt.uri_host` or `unresolved_remote`, or the `Responder` by echoing the incoming request's. Follows the *interfaces.EndpointAddress* interface. Non-roundtrippable.

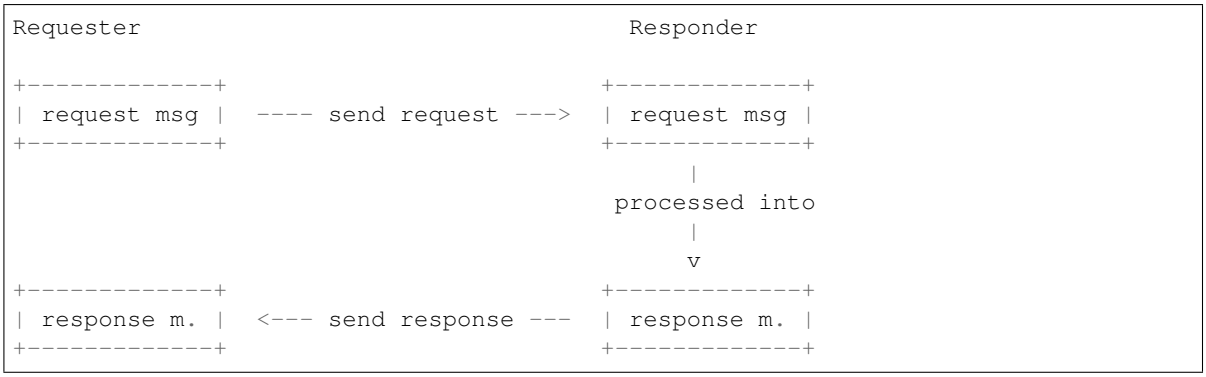
  While a message has not been transmitted, the property is managed by the *Message* itself using the *set_request_uri()* or the constructor *uri* argument.
- `request`: The request to which an incoming response message belongs; only available at the client. Managed by the *interfaces.RequestProvider* (typically a *Context*).

These properties are still available but deprecated:

- requested_*: Managed by the *protocol.Request* a response results from, and filled with the request's URL data. Non-roundtrippable.
- unresolved_remote: `host[:port]` (strictly speaking; hostinfo as in a URI) formatted string. If this attribute is set, it overrides `.RequestManageropt.uri_host` (and `-_port`) when it comes to filling the `remote` in an outgoing request.

Use this when you want to send a request with a host name that would not normally resolve to the destination address. (Typically, this is used for proxying.)

Options can be given as further keyword arguments at message construction time. This feature is experimental, as future message parameters could collide with options.



The above shows the four message instances involved in communication between an aiocoap client and server process. Boxes represent instances of Message, and the messages on the same line represent a single CoAP as passed around on the network. Still, they differ in some aspects:

- The requested URI will look different between requester and responder if the requester uses a host name and does not send it in the message.
- If the request was sent via multicast, the response's requested URI differs from the request URI because it has the responder's address filled in. That address is not known at the responder's side yet, as it is typically filled out by the network stack.
- It is yet unclear whether the response's URI should contain an IP literal or a host name in the unicast case if the Uri-Host option was not sent.
- Properties like Message ID and token will differ if a proxy was involved.
- Some options or even the payload may differ if a proxy was involved.

copy(***kwargs*)
Create a copy of the Message. kwargs are treated like the named arguments in the constructor, and update the copy.

classmethod decode(*rawdata*, *remote=None*)
Create Message object from binary representation of message.

encode()
Create binary representation of message from Message object.

get_cache_key(*ignore_options=()*)
Generate a hashable and comparable object (currently a tuple) from the message's code and all option values that are part of the cache key and not in the optional list of ignore_options (which is the list of option numbers that are not technically NoCacheKey but handled by the application using this method).

```
>>> from aiocoap.numbers import GET
>>> m1 = Message(code=GET)
>>> m2 = Message(code=GET)
>>> m1.opt.uri_path = ('s', '1')
>>> m2.opt.uri_path = ('s', '1')
>>> m1.opt.size1 = 10 # the only no-cache-key option in the base spec
>>> m2.opt.size1 = 20
>>> m1.get_cache_key() == m2.get_cache_key()
```

(continues on next page)

(continued from previous page)

```
True
>>> m2.opt.etag = b'000'
>>> m1.get_cache_key() == m2.get_cache_key()
False
>>> from aiocoap.numbers.optionnumbers import OptionNumber
>>> ignore = [OptionNumber.ETAG]
>>> m1.get_cache_key(ignore) == m2.get_cache_key(ignore)
True
```

get_request_uri(*, *local_is_server=False*)

The absolute URI this message belongs to.

For requests, this is composed from the options (falling back to the remote). For responses, this is largely taken from the original request message (so far, that could have been trackecd by the requesting application as well), but – in case of a multicast request – with the host replaced by the responder's endpoint details.

This implements Section 6.5 of RFC7252.

By default, these values are only valid on the client. To determine a message's request URI on the server, set the local_is_server argument to True. Note that determining the request URI on the server is brittle when behind a reverse proxy, may not be possible on all platforms, and can only be applied to a request message in a renderer (for the response message created by the renderer will only be populated when it gets transmitted; simple manual copying of the request's remote to the response will not magically make this work, for in the very case where the request and response's URIs differ, that would not catch the difference and still report the multicast address, while the actual sending address will only be populated by the operating system later).

set_request_uri(*uri*, *, *set_uri_host=True*)

Parse a given URI into the uri_* fields of the options.

The remote does not get set automatically; instead, the remote data is stored in the uri_host and uri_port options. That is because name resolution is coupled with network specifics the protocol will know better by the time the message is sent. Whatever sends the message, be it the protocol itself, a proxy wrapper or an alternative transport, will know how to handle the information correctly.

When `set_uri_host=False` is passed, the host/port is stored in the `unresolved_remote` message property instead of the uri_host option; as a result, the unresolved host name is not sent on the wire, which breaks virtual hosts but makes message sizes smaller.

This implements Section 6.4 of RFC7252.

unresolved_remote

requested_scheme

requested_proxy_uri

requested_hostinfo

requested_path

requested_query

aiocoap.message.**NoResponse = <NoResponse>**

Result that can be returned from a render method instead of a Message when due to defaults (eg. multicast link-format queries) or explicit configuration (eg. the No-Response option), no response should be sent at all. Note that per RFC7967 section 2, an ACK is still sent to a CON request.

Depercated; set the no_response option on a regular response instead (see *interfaces.Resource.render()* for details).

aiocoap.options module

class aiocoap.options.**Options**
Bases: object

Represent CoAP Header Options.

decode(*rawdata*)
Passed a CoAP message body after the token as rawdata, fill self with the options starting at the beginning of rawdata, an return the rest of the message (the body).

encode()
Encode all options in option header into string of bytes.

add_option(*option*)
Add option into option header.

delete_option(*number*)
Delete option from option header.

get_option(*number*)
Get option with specified number.

option_list()

uri_path
Iterable view on the URI_PATH option.

uri_query
Iterable view on the URI_QUERY option.

location_path
Iterable view on the LOCATION_PATH option.

location_query
Iterable view on the LOCATION_QUERY option.

block2
Single-value view on the BLOCK2 option.

block1
Single-value view on the BLOCK1 option.

content_format
Single-value view on the CONTENT_FORMAT option.

etag
Single ETag as used in responses

etags
List of ETags as used in requests

if_none_match
Presence of the IF_NONE_MATCH option.

observe
Single-value view on the OBSERVE option.

accept
Single-value view on the ACCEPT option.

uri_host
Single-value view on the URI_HOST option.

uri_port
Single-value view on the URI_PORT option.

proxy_uri
Single-value view on the PROXY_URI option.

proxy_scheme
Single-value view on the PROXY_SCHEME option.

size1
Single-value view on the SIZE1 option.

object_security
Single-value view on the OBJECT_SECURITY option.

max_age
Single-value view on the MAX_AGE option.

if_match
Iterable view on the IF_MATCH option.

no_response
Single-value view on the NO_RESPONSE option.

echo
Single-value view on the ECHO option.

request_tag
Iterable view on the REQUEST_TAG option.

hop_limit
Single-value view on the HOP_LIMIT option.

request_hash
Experimental property for draft-amsuess-core-cachable-oscore

edhoc
Presence of the EDHOC option.

size2
Single-value view on the SIZE2 option.

aiocoap.interfaces module

This module provides interface base classes to various aiocoap software components, especially with respect to request and response handling. It describes abstract base classes for messages, endpoints etc.

It is *completely unrelated* to the concept of "network interfaces".

class aiocoap.interfaces.**MessageInterface**
Bases: object

A MessageInterface is an object that can exchange addressed messages over unreliable transports. Implementations send and receive messages with message type and message ID, and are driven by a Context that deals with retransmission.

Usually, an MessageInterface refers to something like a local socket, and send messages to different remote endpoints depending on the message's addresses. Just as well, a MessageInterface can be useful for one single address only, or use various local addresses depending on the remote address.

send(*message*)
Send a given Message object

determine_remote(*message*)
Return a value suitable for the message's remote property based on its .opt.uri_host or .unresolved_remote.

May return None, which indicates that the MessageInterface can not transport the message (typically because it is of the wrong scheme).

shutdown()
Deactivate the complete transport, usually irrevertably. When the coroutine returns, the object must have made sure that it can be destructed by means of ref-counting or a garbage collector run.

class aiocoap.interfaces.**EndpointAddress**
Bases: object

An address that is suitable for routing through the application to a remote endpoint.

Depending on the MessageInterface implementation used, an EndpointAddress property of a message can mean the message is exchanged "with [2001:db8::2:1]:5683, while my local address was [2001:db8:1::1]:5683" (typical of UDP6), "over the connected <Socket at 0x1234>, whereever that's connected to" (simple6 or TCP) or "with participant 0x01 of the OSCAP key 0x..., routed over <another EndpointAddress>".

EndpointAddresses are only concstructed by MessageInterface objects, either for incoming messages or when populating a message's .remote in *MessageInterface.determine_remote()*.

There is no requirement that those address are always identical for a given address. However, incoming addresses must be hashable and hash-compare identically to requests from the same context. The "same context", for the purpose of EndpointAddresses, means that the message must be eligible for request/response, blockwise (de)composition and observations. (For example, in a DTLS context, the hash must change between epochs due to RFC7252 Section 9.1.2).

So far, it is required that hash-identical objects also compare the same. That requirement might go away in future to allow equality to reflect finer details that are not hashed. (The only property that is currently known not to be hashed is the local address in UDP6, because that is *unknown* in initially sent packages, and thus disregarded for comparison but needed to round-trip through responses.)

hostinfo
The authority component of URIs that this endpoint represents when request are sent to it

Note that the presence of a hostinfo does not necessarily mean that globally meaningful or even syntactically valid URI can be constructed out of it; use the *uri* property for this.

hostinfo_local
The authority component of URIs that this endpoint represents when requests are sent from it.

As with *hostinfo*, this does not necessarily produce sufficient input for a URI; use uri_local instead.

uri
Deprecated alias for uri_base

uri_base
The base URI for the peer (typically scheme plus .hostinfo).

This raises *error.AnonymousHost* when executed on an address whose peer coordinates can not be expressed meaningfully in a URI.

uri_base_local
The base URI for the local side of this remote.

This raises *error.AnonymousHost* when executed on an address whose local coordinates can not be expressed meaningfully in a URI.

is_multicast
True if the remote address is a multicast address, otherwise false.

is_multicast_locally
True if the local address is a multicast address, otherwise false.

scheme
The that is used with addresses of this kind

This is usually a class property. It is applicable to both sides of the communication. (Should there ever be a scheme that addresses the participants differently, a scheme_local will be added.)

maximum_block_size_exp = 6
The maximum negotiated block size that can be sent to this remote.

maximum_payload_size = 1124
The maximum payload size that can be sent to this remote. Only relevant if maximum_block_size_exp is 7. This will be removed in favor of a maximum message size when the block handlers can get serialization length predictions from the remote.

as_response_address()
Address to be assigned to a response to messages that arrived with this message

This can (and does, by default) return self, but gives the protocol the opportunity to react to create a modified copy to deal with variations from multicast.

authenticated_claims
Iterable of objects representing any claims (e.g. an identity, or generally objects that can be used to authorize particular accesses) that were authenticated for this remote.

This is experimental and may be changed without notice.

Its primary use is on the server side; there, a request handler (or resource decorator) can use the claims to decide whether the client is authorized for a particular request. Use on the client side is planned as a requirement on a request, although (especially on side-effect free non-confidential requests) it can also be used in response processing.

blockwise_key
A hashable (ideally, immutable) value that is only the same for remotes from which blocks may be combined. (With all current transports that means that the network addresses need to be in there, and the identity of the security context).

It does *not* just hinge on the identity of the address object, as a first block may come in an OSCORE group request and follow-ups may come in pairwise requests. (And there might be allowed relaxations on the transport under OSCORE, but that'd need further discussion).

class aiocoap.interfaces.**MessageManager**
Bases: object

The interface an entity that drives a MessageInterface provides towards the MessageInterface for callbacks and object acquisition.

dispatch_message(*message*)
Callback to be invoked with an incoming message

dispatch_error(*error: Exception*, *remote*)
Callback to be invoked when the operating system indicated an error condition from a particular remote.

client_credentials
A CredentialsMap that transports should consult when trying to establish a security context

class aiocoap.interfaces.**TokenInterface**
Bases: object

send_message(*message*, *messageerror_monitor*) → Optional[Callable[[], None]]
Send a message. If it returns a a callable, the caller is asked to call in case it no longer needs the message sent, and to dispose of if it doesn't intend to any more.

messageerror_monitor is a function that will be called at most once by the token interface: When the underlying layer is indicating that this concrete message could not be processed. This is typically the case for RSTs on from the message layer, and used to cancel observations. Errors that are not likely to be specific to a message (like retransmission timeouts, or ICMP errors) are reported through dispatch_error instead. (While the information which concrete message triggered that might be available, it is not likely to be relevant).

Currently, it is up to the TokenInterface to unset the no_response option in response messages, and to possibly not send them.

fill_or_recognize_remote(*message*)
Return True if the message is recognized to already have a .remote managedy by this TokenInterface, or return True and set a .remote on message if it should (by its unresolved remote or Uri-* options) be routed through this TokenInterface, or return False otherwise.

class aiocoap.interfaces.**TokenManager**
Bases: object

class aiocoap.interfaces.**RequestInterface**
Bases: object

request(*request: aiocoap.pipe.Pipe*)

fill_or_recognize_remote(*message*)

class aiocoap.interfaces.**RequestProvider**
Bases: object

request(*request_message*)
Create and act on a a *Request* object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

class aiocoap.interfaces.**Request**
Bases: object

A CoAP request, initiated by sending a message. Typically, this is not instanciated directly, but generated by a *RequestProvider.request()* method.

response = 'A future that is present from the creation of the object and fullfilled wi

class aiocoap.interfaces.**Resource**
Bases: object

Interface that is expected by a *protocol.Context* to be present on the serversite, which renders all requests to that context.

needs_blockwise_assembly(*request*)
Indicator to the protocol.Responder about whether it should assemble request blocks to a single request and extract the requested blocks from a complete-resource answer (True), or whether the resource will do that by itself (False).

render(*request*)
Return a message that can be sent back to the requester.

This does not need to set any low-level message options like remote, token or message type; it does however need to set a response code.

A response returned may carry a no_response option (which is actually specified to apply to requests only); the underlying transports will decide based on that and its code whether to actually transmit the response.

render_to_pipe(*request: aiocoap.pipe.Pipe*)
Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then *render()*, *needs_blockwise_assembly()* and *ObservableResource.add_observation()* are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

class aiocoap.interfaces.**ObservableResource**
Bases: *aiocoap.interfaces.Resource*

Interface the *protocol.ServerObservation* uses to negotiate whether an observation can be established based on a request.

This adds only functionality for registering and unregistering observations; the notification contents will be retrieved from the resource using the regular *render()* method from crafted (fake) requests.

add_observation(*request*, *serverobservation*)
Before the incoming request is sent to *render()*, the *add_observation()* method is called. If the resource chooses to accept the observation, it has to call the *serverobservation.accept(cb)* with a callback that will be called when the observation ends. After accepting, the ObservableResource should call *serverobservation.trigger()* whenever it changes its state; the ServerObservation will then initiate notifications by having the request rendered again.

render_to_pipe(*request: aiocoap.pipe.Pipe*)
Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then `render()`, `needs_blockwise_assembly()` and *ObservableResource.add_observation()* are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

aiocoap.error module

Common errors for the aiocoap library

exception aiocoap.error.**Error**
Bases: `Exception`

Base exception for all exceptions that indicate a failed request

exception aiocoap.error.**RenderableError**
Bases: *aiocoap.error.Error*

Exception that can meaningfully be represented in a CoAP response

to_message()
Create a CoAP message that should be sent when this exception is rendered

exception aiocoap.error.**ResponseWrappingError**(*coapmessage*)
Bases: *aiocoap.error.Error*

An exception that is raised due to an unsuccessful but received response.

A better relationship with *numbers.codes* should be worked out to do `except UnsupportedMediaType` (similar to the various `OSError` subclasses).

to_message()

exception aiocoap.error.**ConstructionRenderableError**(*message=None*)
Bases: *aiocoap.error.RenderableError*

RenderableError that is constructed from class attributes *code* and *message* (where the can be overridden in the constructor).

to_message()
Create a CoAP message that should be sent when this exception is rendered

code = <Response Code 160 "5.00 Internal Server Error">
Code assigned to messages built from it

message = ''
Text sent in the built message's payload

exception aiocoap.error.**NotFound**(*message=None*)
Bases: *aiocoap.error.ConstructionRenderableError*

code = <Response Code 132 "4.04 Not Found">

exception aiocoap.error.**MethodNotAllowed**(*message=None*)
Bases: *aiocoap.error.ConstructionRenderableError*

code = <Response Code 133 "4.05 Method Not Allowed">

exception aiocoap.error.**UnsupportedContentFormat**(*message=None*)
Bases: *aiocoap.error.ConstructionRenderableError*

code = <Response Code 143 "4.15 Unsupported Content Format">

exception aiocoap.error.**Unauthorized**(*message=None*)
Bases: *aiocoap.error.ConstructionRenderableError*

code = <Response Code 129 "4.01 Unauthorized">

exception aiocoap.error.**BadRequest**(*message=None*)
Bases: *aiocoap.error.ConstructionRenderableError*

code = <Response Code 128 "4.00 Bad Request">

exception aiocoap.error.**NoResource**
Bases: *aiocoap.error.NotFound*

Raised when resource is not found.

message = 'Error: Resource not found!'

exception aiocoap.error.**UnallowedMethod**(*message=None*)
Bases: *aiocoap.error.MethodNotAllowed*

Raised by a resource when request method is understood by the server but not allowed for that particular resource.

message = 'Error: Method not allowed!'

exception aiocoap.error.**UnsupportedMethod**(*message=None*)
Bases: *aiocoap.error.MethodNotAllowed*

Raised when request method is not understood by the server at all.

message = 'Error: Method not recognized!'

exception aiocoap.error.**NetworkError**
Bases: *aiocoap.error.Error*

Base class for all "something went wrong with name resolution, sending or receiving packages".

Errors of these kinds are raised towards client callers when things went wrong network-side, or at context creation. They are often raised from socket.gaierror or similar classes, but these are wrapped in order to make catching them possible independently of the underlying transport.

exception aiocoap.error.**ResolutionError**
Bases: *aiocoap.error.NetworkError*

Resolving the host component of a URI to a usable transport address was not possible

exception aiocoap.error.**MessageError**
Bases: *aiocoap.error.NetworkError*

Received an error from the remote on the CoAP message level (typically a RST)

exception aiocoap.error.**NotImplemented**
Bases: *aiocoap.error.Error*

Raised when request is correct, but feature is not implemented by library. For example non-sequential blockwise transfers

exception aiocoap.error.**RemoteServerShutdown**
Bases: *aiocoap.error.NetworkError*

The peer a request was sent to in a stateful connection closed the connection around the time the request was sent

exception aiocoap.error.**TimeoutError**
Bases: *aiocoap.error.NetworkError*

Base for all timeout-ish errors.

Like NetworkError, receiving this alone does not indicate whether the request may have reached the server or not.

exception aiocoap.error.**ConRetransmitsExceeded**
Bases: *aiocoap.error.TimeoutError*

A transport that retransmits CON messages has failed to obtain a response within its retransmission timeout.

When this is raised in a transport, requests failing with it may or may have been received by the server.

exception aiocoap.error.**RequestTimedOut**
Bases: *aiocoap.error.TimeoutError*

Raised when request is timed out.

This error is currently not produced by aiocoap; it is deprecated. Users can now catch error.TimeoutError, or newer more detailed subtypes introduced later.

exception aiocoap.error.**WaitingForClientTimedOut**
Bases: *aiocoap.error.TimeoutError*

Raised when server expects some client action:

- sending next PUT/POST request with block1 or block2 option
- sending next GET request with block2 option

but client does nothing.

This error is currently not produced by aiocoap; it is deprecated. Users can now catch error.TimeoutError, or newer more detailed subtypes introduced later.

exception aiocoap.error.**ResourceChanged**
Bases: *aiocoap.error.Error*

The requested resource was modified during the request and could therefore not be received in a consistent state.

exception aiocoap.error.**UnexpectedBlock1Option**
Bases: *aiocoap.error.Error*

Raised when a server responds with block1 options that just don't match.

exception aiocoap.error.**UnexpectedBlock2**
Bases: *aiocoap.error.Error*

Raised when a server responds with another block2 than expected.

exception aiocoap.error.**MissingBlock2Option**
Bases: *aiocoap.error.Error*

Raised when response with Block2 option is expected (previous response had Block2 option with More flag set), but response without Block2 option is received.

exception aiocoap.error.**NotObservable**
Bases: *aiocoap.error.Error*

The server did not accept the request to observe the resource.

exception aiocoap.error.**ObservationCancelled**
Bases: *aiocoap.error.Error*

The server claimed that it will no longer sustain the observation.

exception aiocoap.error.**UnparsableMessage**
Bases: *aiocoap.error.Error*

An incoming message does not look like CoAP.

Note that this happens rarely – the requirements are just two bit at the beginning of the message, and a minimum length.

exception aiocoap.error.**LibraryShutdown**
Bases: *aiocoap.error.Error*

The library or a transport registered with it was requested to shut down; this error is raised in all outstanding requests.

exception aiocoap.error.**AnonymousHost**
Bases: *aiocoap.error.Error*

This is raised when it is attempted to express as a reference a (base) URI of a host or a resource that can not be reached by any process other than this.

Typically, this happens when trying to serialize a link to a resource that is hosted on a CoAP-over-TCP or -WebSockets client: Such resources can be accessed for as long as the connection is active, but can not be used any more once it is closed or even by another system.

aiocoap.pipe module

class aiocoap.pipe.**Pipe**(*request*, *log*)
Bases: object

Low-level meeting point between a request and a any responses that come back on it.

A single request message is placed in the Pipe at creation time. Any responses, as well as any exception happening in the course of processing, are passed back to the requester along the Pipe. A response can carry an indication of whether it is final; an exception always is.

This object is used both on the client side (where the Context on behalf of the application creates a Pipe and passes it to the network transports that send the request and fill in any responses) and on the server side (where the Context creates one for an incoming request and eventually lets the server implementation populate it with responses).

This currently follows a callback dispatch style. (It may be developed into something where only awaiting a response drives the proces, though).

Currently, the requester sets up the object, connects callbacks, and then passes the Pipe on to whatever creates the response.

The creator of responses is notified by the Pipe of a loss of interest in a response when there are no more callback handlers registered by registering an on_interest_end callback. As the response callbacks need to be already in place when the Pipe is passed on to the responder, the absence event callbacks is signalled by callign the callback immediately on registration.

To accurately model "loss of interest", it is important to use the two-phase setup of first registering actual callbacks and then producing events and/or placing on_interest_end callbacks; this is not clearly expressed in type or state yet. (One possibility would be for the Pipe to carry a preparation boolean, and which prohibits event sending during preparation and is_interest=True callback creation afterwards).

This was previously named PlumbingRequest.

Stability

Sites and resources implemented by provinding a *render_to_pipe()* method can stably use the *add_response()* method of a Pipe (or something that quacks like it).

They should not rely on *add_exception()* but rather just raise the exception, and neither register *on_event()* handlers (being the sole producer of events) nor hook to *on_interest_end()* (instead, they can use finally clauses or async context managers to handle any cleanup when the cancellation of the render task indicates the peer's loss of interest).

class Event(*message*, *exception*, *is_last*)

Bases: tuple

exception

Alias for field number 1

is_last

Alias for field number 2

message

Alias for field number 0

poke()

Ask the responder for a life sign. It is up to the responder to ignore this (eg. because the responder is the library/application and can't be just gone), to issue a generic transport-dependent 'ping' to see whether the connection is still alive, or to retransmit the request if it is an observation over an unreliable channel.

In any case, no status is reported directly to the poke, but if whatever the responder does fails, it will send an appropriate error message as a response.

on_event(*callback*, *is_interest=True*)

Call callback on any event. The callback must return True to be called again after an event. Callbacks must not produce new events or deregister unrelated event handlers.

If is_interest=False, the callback will not be counted toward the active callbacks, and will receive a (None, None, is_last=True) event eventually.

To unregister the handler, call the returned closure; this can trigger on_interest_end callbacks.

on_interest_end(*callback*)
Register a callback that will be called exactly once – either right now if there is not even a current indicated interest, or at a last event, or when no more interests are present

add_response(*response*, *is_last=False*)

add_exception(*exception*)

aiocoap.pipe.**run_driving_pipe**(*pipe*, *coroutine*, *name=None*)
Create a task from a coroutine where the end of the coroutine produces a terminal event on the pipe, and lack of interest in the pipe cancels the task.

The coroutine will typically produce output into the pipe; that connection is set up by the caller like as in `run_driving_pipe(pipe, render_to(pipe))`.

The create task is not returned, as the only sensible operation on it would be cancellation and that's already set up from the pipe.

aiocoap.pipe.**error_to_message**(*old_pr*, *log*)
Given a pipe set up by the requester, create a new pipe to pass on to a responder.

Any exceptions produced by the responder will be turned into terminal responses on the original pipe, and loss of interest is forwarded.

class aiocoap.pipe.**IterablePipe**(*request*)
Bases: `object`

A stand-in for a Pipe that the requesting party can use instead. It should behave just like a Pipe to the responding party, but the caller does not register on_event handlers and instead iterates asynchronously over the events.

Note that the PR can be aitered over only once, and does not support any additional hook settings once asynchronous iteration is started; this is consistent with the usage pattern of pipes.

on_interest_end(*callback*)

add_response(*response*, *is_last=False*)

add_exception(*exception*)

class Iterator(*queue*, *on_interest_end*)
Bases: `object`

aiocoap.defaults module

This module contains helpers that inspect available modules and platform specifics to give sane values to aiocoap defaults.

All of this should eventually overridable by other libraries wrapping/using aiocoap and by applications using aiocoap; however, these overrides do not happen in the defaults module but where these values are actually accessed, so this module is considered internal to aiocoap and not part of the API.

The `_missing_modules` functions are helpers for inspecting what is reasonable to expect to work. They can influence default values, but should not be used in the rest of the code for feature checking (just raise the ImportErrors) unless it's directly user-visible ("You configured OSCORE key material, but OSCORE needs the following unavailable modules") or in the test suite to decide which tests to skip.

aiocoap.defaults.**get_default_clienttransports**(*, *loop=None*, *use_env=True*)
Return a list of transports that should be connected when a client context is created.

If an explicit `AIOCOAP_CLIENT_TRANSPORT` environment variable is set, it is read as a colon separated list of transport names.

By default, a DTLS mechanism will be picked if the required modules are available, and a UDP transport will be selected depending on whether the full udp6 transport is known to work.

aiocoap.defaults.**get_default_servertransports**(*, *loop=None*, *use_env=True*)
Return a list of transports that should be connected when a server context is created.

If an explicit AIOCOAP_SERVER_TRANSPORT environment variable is set, it is read as a colon separated list of transport names.

By default, a DTLS mechanism will be picked if the required modules are available, and a UDP transport will be selected depending on whether the full udp6 transport is known to work. Both a simple6 and a simplesocketserver will be selected when udp6 is not available, and the simple6 will be used for any outgoing requests, which the simplesocketserver could serve but is worse at.

aiocoap.defaults.**has_reuse_port**(*, *use_env=True*)
Return true if the platform indicates support for SO_REUSEPORT.

Can be overridden by explicitly setting AIOCOAP_REUSE_PORT to 1 or 0.

aiocoap.defaults.**dtls_missing_modules**()
Return a list of modules that are missing in order to use the DTLS transport, or a false value if everything is present

aiocoap.defaults.**oscore_missing_modules**()
Return a list of modules that are missing in order to use OSCORE, or a false value if everything is present

aiocoap.defaults.**ws_missing_modules**()
Return a list of modules that are missing in order to user CoAP-over-WS, or a false value if everything is present

aiocoap.defaults.**linkheader_missing_modules**()
Return a list of moudles that are missing in order to use link_header functionaity (eg. running a resource directory), of a false value if everything is present.

aiocoap.defaults.**prettyprint_missing_modules**()
Return a list of modules that are missing in order to use pretty printing (ie. full aiocoap-client)

aiocoap.transports module

Container module for transports

Transports are expected to be the modular backends of aiocoap, and implement the specifics of eg. TCP, WebSockets or SMS, possibly divided by backend implementations as well.

Transports are not part of the API, so the class descriptions in the modules are purely informational.

Multiple transports can be used in parallel in a single *Context*, and are loaded in a particular sequence. Some transports will grab all addresses of a given protocol, so they might not be practical to combine. Which transports are started in a given Context follows the *defaults.get_default_clienttransports()* function.

The available transports are:

aiocoap.transports.generic_udp module

class aiocoap.transports.generic_udp.**GenericMessageInterface**(*mman: aiocoap.interfaces.MessageManager*, *log*, *loop*)
Bases: *aiocoap.interfaces.MessageInterface*

GenericMessageInterface is not a standalone implementation of a message inteface. It does implement everything between the MessageInterface and a not yet fully specified interface of “bound UDP sockets”.

It delegates sending through the address objects (which persist through some time, given this is some kind of bound-socket scenario).

The user must: * set up a ._pool after construction with a shutdown and a connect method * provide their addresses with a send(bytes) method * pass incoming data to the _received_datagram and _received_exception methods

send(*message*)
Send a given `Message` object

determine_remote(*request*)
Return a value suitable for the message's remote property based on its .opt.uri_host or .unresolved_remote.

May return None, which indicates that the MessageInterface can not transport the message (typically because it is of the wrong scheme).

shutdown()
Deactivate the complete transport, usually irrevertably. When the coroutine returns, the object must have made sure that it can be destructed by means of ref-counting or a garbage collector run.

aiocoap.transports.oscore module

This module implements a RequestProvider for OSCORE. As such, it takes routing ownership of requests that it has a security context available for, and sends off the protected messages via another transport.

This transport is a bit different from the others because it doesn't have its dedicated URI scheme, but purely relies on preconfigured contexts.

So far, this transport only deals with outgoing requests, and does not help in building an OSCORE server. (Some code that could be used here in future resides in *contrib/oscore-plugtest/plugtest-server* as the *ProtectedSite* class.

In outgoing request, this transport automatically handles Echo options that appear to come from RFC8613 Appendix B.1.2 style servers. They indicate that the server could not process the request initially, but could do so if the client retransmits it with an appropriate Echo value.

Unlike other transports that could (at least in theory) be present multiple times in `aiocoap.protocol.Context.request_interfaces` (eg. because there are several bound sockets), this is only useful once in there, as it has no own state, picks the OSCORE security context from the CoAP `aiocoap.protocol.Context.client_credentials` when populating the remote field, and handles any populated request based ono its remote.security_context property alone.

class aiocoap.transports.oscore.**OSCOREAddress**
Bases: aiocoap.transports.oscore._OSCOREAddress, *aiocoap.interfaces.EndpointAddress*

Remote address type for **:cls:'TransportOSCORE'**.

hostinfo
The authority component of URIs that this endpoint represents when request are sent to it

Note that the presence of a hostinfo does not necessarily mean that globally meaningful or even syntactically valid URI can be constructed out of it; use the *uri* property for this.

hostinfo_local
The authority component of URIs that this endpoint represents when requests are sent from it.

As with *hostinfo*, this does not necessarily produce sufficient input for a URI; use `uri_local` instead.

uri_base
The base URI for the peer (typically scheme plus .hostinfo).

This raises *error.AnonymousHost* when executed on an address whose peer coordinates can not be expressed meaningfully in a URI.

uri_base_local
The base URI for the local side of this remote.

This raises *error.AnonymousHost* when executed on an address whose local coordinates can not be expressed meaningfully in a URI.

scheme
The that is used with addresses of this kind

This is usually a class property. It is applicable to both sides of the communication. (Should there ever be a scheme that addresses the participants differently, a scheme_local will be added.)

authenticated_claims
Iterable of objects representing any claims (e.g. an identity, or generally objects that can be used to authorize particular accesses) that were authenticated for this remote.

This is experimental and may be changed without notice.

Its primary use is on the server side; there, a request handler (or resource decorator) can use the claims to decide whether the client is authorized for a particular request. Use on the client side is planned as a requirement on a request, although (especially on side-effect free non-confidential requests) it can also be used in response processing.

is_multicast = False

maximum_payload_size = 1024

maximum_block_size_exp = 6

blockwise_key
A hashable (ideally, immutable) value that is only the same for remotes from which blocks may be combined. (With all current transports that means that the network addresses need to be in there, and the identity of the security context).

It does *not* just hinge on the identity of the address object, as a first block may come in an OSCORE group request and follow-ups may come in pairwise requests. (And there might be allowed relaxations on the transport under OSCORE, but that'd need further discussion).

class aiocoap.transports.oscore.**TransportOSCORE**(*context*, *forward_context*)
Bases: *aiocoap.interfaces.RequestProvider*

request(*request*)
Create and act on a a `Request` object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

fill_or_recognize_remote(*message*)

shutdown()

aiocoap.transports.rfc8323common module

Common code for the tcp and the ws modules, both of which are based on RFC8323 mechanisms, but differ in their underlying protocol implementations (asyncio stream vs. websockets module) far enough that they only share small portions of their code

exception aiocoap.transports.rfc8323common.**CloseConnection**
Bases: `Exception`

Raised in RFC8323 common processing to trigger a connection shutdown on the TCP / WebSocket side.

The TCP / WebSocket side should send the exception's argument on to the token manager, close the connection, and does not need to perform further logging.

class aiocoap.transports.rfc8323common.**RFC8323Remote**
Bases: object

Mixin for Remotes for all the common RFC8323 processing

Implementations still need the per-transport parts, especially a _send_message and an _abort_with implementation.

is_multicast = False

is_multicast_locally = False

hostinfo

hostinfo_local

uri_base

uri_base_local

maximum_block_size_exp

maximum_payload_size

blockwise_key

abort(*errormessage=None*, *bad_csm_option=None*)

release()
Send Release message, (not implemented:) wait for connection to be actually closed by the peer.

Subclasses should extend this to await closing of the connection, especially if they'd get into lock-up states otherwise (was would WebSockets).

aiocoap.transports.simple6 module

This module implements a MessageInterface for UDP based on the asyncio DatagramProtocol.

This is a simple version that works only for clients (by creating a dedicated unbound but connected socket for each communication partner) and probably not with multicast (it is assumed to be unsafe for multicast), which can be expected to work even on platforms where the *udp6* module can not be made to work (Android, OSX, Windows for missing `recvmsg` and socket options, or any event loops that don't have an add_reader method).

Note that the name of the module is a misnomer (and the module is likely to be renamed): Nothing in it is IPv6 specific; the socket is created using whichever address family the OS chooses based on the given host name.

One small but noteworthy detail about this transport is that it does not distinguish between IP literals and host names. As a result, requests and responses from remotes will appear to arrive from a remote whose netloc is the requested name, not an IP literal.

This transport is experimental, likely to change, and not fully tested yet (because the test suite is not yet ready to matrix-test the same tests with different transport implementations, and because it still fails in proxy blockwise tests).

For one particular use case, this may be usable for servers in a sense: If (and only if) all incoming requests are only ever sent from clients that were previously addressed as servers by the running instance. (This is generally undesirable as it greatly limits the usefulness of the server, but is used in LwM2M setups). As such a setup makes demands on the peer that are not justified by the CoAP specification (in particular, that it send requests from a particular port), this should still only be used for cases where the udp6 transport is unavailable due to platform limitations.

class aiocoap.transports.simple6.**MessageInterfaceSimple6**(*mman: aiocoap.interfaces.MessageManager*, *log*, *loop*)

Bases: *aiocoap.transports.generic_udp.GenericMessageInterface*

classmethod create_client_transport_endpoint(*ctx*, *log*, *loop*)

recognize_remote(*remote*)

aiocoap.transports.simplesocketserver module

This module implements a MessageInterface for UDP based on the asyncio DatagramProtocol.

This is a simple version that works only for servers bound to a single unicast address. It provides a server backend in situations when *udp6* is unavailable and *simple6* needs to be used for clients.

While it is in theory capable of sending requests too, it should not be used like that, because it won't receive ICMP errors (see below).

Shortcomings

- This implementation does not receive ICMP errors. This violates the CoAP standard and can lead to unnecessary network traffic, bad user experience (when used for client requests) or even network attack amplification.
- The server can not be used with the "any-address" (`::`, `0.0.0.0`). If it were allowed to bind there, it would not receive any indication from the operating system as to which of its own addresses a request was sent, and could not send the response with the appropriate sender address.

  (The *udp6* transport does not suffer that shortcoming, simplesocketserver is typically only used when that is unavailable).

  With simplesocketserver, you need to explicitly give the IP address of your server in the `bind` argument of *aiocoap.protocol.Context.create_server_context()*.
- This transport is experimental and likely to change.

class aiocoap.transports.simplesocketserver.**MessageInterfaceSimpleServer**(*mman: aiocoap.interfaces.MessageM*, *log*, *loop*)

Bases: *aiocoap.transports.generic_udp.GenericMessageInterface*

classmethod create_server(*bind*, *ctx: aiocoap.interfaces.MessageManager*, *log*, *loop*)

recognize_remote(*remote*)

aiocoap.transports.tcp module

class aiocoap.transports.tcp.**TcpConnection**(*ctx*, *log*, *loop*, ***, *is_server*)

Bases: asyncio.protocols.Protocol, *aiocoap.transports.rfc8323common.RFC8323Remote*, *aiocoap.interfaces.EndpointAddress*

scheme

The that is used with addresses of this kind

This is usually a class property. It is applicable to both sides of the communication. (Should there ever be a scheme that addresses the participants differently, a scheme_local will be added.)

connection_made(*transport*)
Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for data_received() calls. When the connection is closed, connection_lost() is called.

connection_lost(*exc*)
Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

data_received(*data*)
Called when some data is received.

The argument is a bytes object.

eof_received()
Called when the other end calls write_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

pause_writing()
Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()
Called when the transport's buffer drains below the low-water mark.

See pause_writing() for details.

class aiocoap.transports.tcp.**TCPServer**
Bases: aiocoap.transports.tcp._TCPPooling, *aiocoap.interfaces.TokenInterface*

classmethod create_server(*bind*, *tman: aiocoap.interfaces.TokenManager*, *log*, *loop*, *, *_server_context=None*)

fill_or_recognize_remote(*message*)
Return True if the message is recognized to already have a .remote managedy by this TokenInterface, or return True and set a .remote on message if it should (by its unresolved remote or Uri-* options) be routed through this TokenInterface, or return False otherwise.

shutdown()

class aiocoap.transports.tcp.**TCPClient**
Bases: aiocoap.transports.tcp._TCPPooling, *aiocoap.interfaces.TokenInterface*

classmethod create_client_transport(*tman: aiocoap.interfaces.TokenManager*, *log*, *loop*, *credentials=None*)

fill_or_recognize_remote(*message*)
Return True if the message is recognized to already have a .remote managedy by this TokenInterface, or

return True and set a .remote on message if it should (by its unresolved remote or Uri-* options) be routed through this TokenInterface, or return False otherwise.

shutdown()

aiocoap.transports.tinydtls module

This module implements a MessageInterface that handles coaps:// using a wrapped tinydtls library.

This currently only implements the client side. To have a test server, run:

```
$ git clone https://github.com/obgm/libcoap.git --recursive
$ cd libcoap
$ ./autogen.sh
$ ./configure --with-tinydtls --disable-shared --disable-documentation
$ make
$ ./examples/coap-server -k secretPSK
```

(Using TinyDTLS in libcoap is important; with the default OpenSSL build, I've seen DTLS1.0 responses to DTLS1.3 requests, which are hard to debug.)

The test server with its built-in credentials can then be accessed using:

```
$ echo '{"coaps://localhost/*": {"dtls": {"psk": {"ascii": "secretPSK"}, "client-
↪identity": {"ascii": "client_Identity"}}}}' > testserver.json
$ ./aiocoap-client coaps://localhost --credentials testserver.json
```

While it is planned to allow more programmatical construction of the credentials store, the currently recommended way of storing DTLS credentials is to load a structured data object into the client_credentials store of the context:

```
>>> c = await aiocoap.Context.create_client_context()          # doctest: +SKIP
>>> c.client_credentials.load_from_dict(
...     {'coaps://localhost/*': {'dtls': {
...         'psk': b'secretPSK',
...         'client-identity': b'client_Identity',
...         }}})                                                # doctest: +SKIP
```

where, compared to the JSON example above, byte strings can be used directly rather than expressing them as 'ascii'/'hex' (*{'hex': '30383135'}* style works as well) to work around JSON's limitation of not having raw binary strings.

Bear in mind that the aiocoap CoAPS support is highly experimental; for example, while requests to this server do complete, error messages are still shown during client shutdown.

exception aiocoap.transports.tinydtls.**CloseNotifyReceived**

Bases: Exception

The DTLS connection a request was sent on raised was closed by the server while the request was being processed

exception aiocoap.transports.tinydtls.**FatalDTLSError**

Bases: Exception

The DTLS connection a request was sent on raised a fatal error while the request was being processed

class aiocoap.transports.tinydtls.**DTLSClientConnection**(*host*, *port*, *pskId*, *psk*, *coap-transport*)

Bases: *aiocoap.interfaces.EndpointAddress*

is_multicast = False

is_multicast_locally = False

uri_base

uri_base_local

scheme = 'coaps'

hostinfo_local
The authority component of URIs that this endpoint represents when requests are sent from it.

As with *hostinfo*, this does not necessarily produce sufficient input for a URI; use `uri_local` instead.

blockwise_key
A hashable (ideally, immutable) value that is only the same for remotes from which blocks may be combined. (With all current transports that means that the network addresses need to be in there, and the identity of the security context).

It does *not* just hinge on the identity of the address object, as a first block may come in an OSCORE group request and follow-ups may come in pairwise requests. (And there might be allowed relaxations on the transport under OSCORE, but that'd need further discussion).

hostinfo = None

send(*message*)

log

shutdown()

class SingleConnection(*parent*)
Bases: `object`

classmethod factory(*parent*)

parent = None
DTLSClientConnection

connection_made(*transport*)

connection_lost(*exc*)

error_received(*exc*)

datagram_received(*data*, *addr*)

class aiocoap.transports.tinydtls.**MessageInterfaceTinyDTLS**(*ctx: aiocoap.interfaces.MessageManager*, *log*, *loop*)
Bases: *aiocoap.interfaces.MessageInterface*

classmethod create_client_transport_endpoint(*ctx: aiocoap.interfaces.MessageManager*, *log*, *loop*)

determine_remote(*request*)
Return a value suitable for the message's remote property based on its .opt.uri_host or .unresolved_remote.

May return None, which indicates that the MessageInterface can not transport the message (typically because it is of the wrong scheme).

recognize_remote(*remote*)

shutdown()
Deactivate the complete transport, usually irrevertably. When the coroutine returns, the object must have made sure that it can be destructed by means of ref-counting or a garbage collector run.

send(*message*)
Send a given `Message` object

aiocoap.transports.tinydtls_server module

This module implements a MessageInterface that serves coaps:// using a wrapped tinydtls library.

Bear in mind that the aiocoap CoAPS support is highly experimental and incomplete.

Unlike other transports this is *not* enabled automatically in general, as it is limited to servers bound to a single address for implementation reasons. (Basically, because it is built on the simplesocketserver rather than the udp6 server – that can change in future, though). Until either the implementation is changed or binding arguments are (allowing different transports to bind to per-transport addresses or ports), a DTLS server will only be enabled if the AIOCOAP_DTLSSERVER_ENABLED environment variable is set, or tinydtls_server is listed explicitly in AIOCOAP_SERVER_TRANSPORT.

class aiocoap.transports.tinydtls_server.**GoingThroughMessageDecryption**(*plaintext_interface: aiocoap.transports.generic_udp*

Bases: `object`

Warapper around GenericMessageInterface that puts incoming data through the DTLS context stored with the address

class aiocoap.transports.tinydtls_server.**SecurityStore**(*server_credentials*)
Bases: `object`

Wrapper around a CredentialsMap that makes it accessible to the dict-like object DTLSSocket expects.

Not only does this convert interfaces, it also adds a back channel: As DTLSSocket wouldn't otherwise report who authenticated, this is tracking access and storing the claims associated with the used key for later use.

Therefore, SecurityStore objects are created per connection and not per security store.

keys()

class aiocoap.transports.tinydtls_server.**MessageInterfaceTinyDTLSServer**(*mman: aiocoap.interfaces.MessageMa log, loop*)

Bases: *aiocoap.transports.simplesocketserver.MessageInterfaceSimpleServer*

classmethod create_server(*bind, ctx: aiocoap.interfaces.MessageManager, log, loop, server_credentials*)

shutdown()
Deactivate the complete transport, usually irrevertably. When the coroutine returns, the object must have made sure that it can be destructed by means of ref-counting or a garbage collector run.

aiocoap.transports.tls module

CoAP-over-TLS transport (early work in progress)

Right now this is running on self-signed, hard-coded certificates with default SSL module options.

To use this, generate keys as with:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 5 -nodes
```

and state your hostname (eg. localhost) when asked for the Common Name.

class aiocoap.transports.tls.**TLSServer**
Bases: aiocoap.transports.tls._TLSMixIn, *aiocoap.transports.tcp.TCPServer*

classmethod create_server(*bind*, *tman*, *log*, *loop*, *server_context*)

class aiocoap.transports.tls.**TLSClient**
Bases: aiocoap.transports.tls._TLSMixIn, *aiocoap.transports.tcp.TCPClient*

aiocoap.transports.udp6 module

This module implements a MessageInterface for UDP based on a variation of the asyncio DatagramProtocol.

This implementation strives to be correct and complete behavior while still only using a single socket; that is, to be usable for all kinds of multicast traffic, to support server and client behavior at the same time, and to work correctly even when multiple IPv6 and IPv4 (using V4MAPPED addresses) interfaces are present, and any of the interfaces has multiple addresses.

This requires using some standardized but not necessarily widely ported features: `AI_V4MAPPED` to support IPv4 without resorting to less standardized mechanisms for later options, `IPV6_RECVPKTINFO` to determine incoming packages' destination addresses (was it multicast) and to return packages from the same address, `IPV6_JOIN_GROUP` for multicast membership management and `recvmsg` to obtain data configured with the above options.

To the author's knowledge, there is no standardized mechanism for receiving ICMP errors in such a setup. On Linux, `IPV6_RECVERR` and `MSG_ERRQUEUE` are used to receive ICMP errors from the socket; on other platforms, a warning is emitted that ICMP errors are ignored. Using a *simple6* for clients is recommended for those when working as a client only.

Exceeding for the above error handling, no attempts are made to fall back to a kind-of-correct or limited-functionality behavior if these options are unavailable, for the resulting code would be hard to maintain ("`ifdef` hell") or would cause odd bugs at users (eg. servers that stop working when an additional IPv6 address gets assigned). If the module does not work for you, and the options can not be added easily to your platform, consider using the *simple6* module instead.

class aiocoap.transports.udp6.**InterfaceOnlyPktinfo**
Bases: `bytes`

A thin wrapper over bytes that represent a pktinfo built just to select an outgoing interface.

This must not be treated any different than a regular pktinfo, and is just tagged for better debug output. (Ie. if this is replaced everywhere with plain *bytes*, things must still work).

class aiocoap.transports.udp6.**UDP6EndpointAddress**(*sockaddr*, *interface*, ***, *pktinfo=None*)
Bases: *aiocoap.interfaces.EndpointAddress*

Remote address type for **:cls:'MessageInterfaceUDP6'**. Remote address is stored in form of a socket address; local address can be roundtripped by opaque pktinfo data.

For purposes of equality (and thus hashing), the local address is *not* checked. Neither is the scopeid that is part of the socket address.

```
>>> interface = type("FakeMessageInterface", (), {})
>>> if1_name = socket.if_indextoname(1)
>>> local = UDP6EndpointAddress(socket.getaddrinfo('127.0.0.1', 5683, type=socket.
↪SOCK_DGRAM, family=socket.AF_INET6, flags=socket.AI_V4MAPPED)[0][-1], interface)
>>> local.is_multicast
```

(continues on next page)

(continued from previous page)

```
False
>>> local.hostinfo
'127.0.0.1'
>>> all_coap_link1 = UDP6EndpointAddress(socket.getaddrinfo('ff02:0:0:0:0:0:0:fd%1
↪', 1234, type=socket.SOCK_DGRAM, family=socket.AF_INET6)[0][-1], interface)
>>> all_coap_link1.is_multicast
True
>>> all_coap_link1.hostinfo == '[ff02::fd%{}]:1234'.format(if1_name)
True
>>> all_coap_site = UDP6EndpointAddress(socket.getaddrinfo('ff05:0:0:0:0:0:0:fd',␣
↪1234, type=socket.SOCK_DGRAM, family=socket.AF_INET6)[0][-1], interface)
>>> all_coap_site.is_multicast
True
>>> all_coap_site.hostinfo
'[ff05::fd]:1234'
>>> all_coap4 = UDP6EndpointAddress(socket.getaddrinfo('224.0.1.187', 5683,␣
↪type=socket.SOCK_DGRAM, family=socket.AF_INET6, flags=socket.AI_V4MAPPED)[0][-
↪1], interface)
>>> all_coap4.is_multicast
True
```

scheme = 'coap'

interface

netif
Textual interface identifier of the explicitly configured remote interface, or the interface identifier reported in an incoming link-local message. None if not set.

hostinfo
The authority component of URIs that this endpoint represents when request are sent to it

Note that the presence of a hostinfo does not necessarily mean that globally meaningful or even syntactically valid URI can be constructed out of it; use the *uri* property for this.

hostinfo_local
The authority component of URIs that this endpoint represents when requests are sent from it.

As with *hostinfo*, this does not necessarily produce sufficient input for a URI; use `uri_local` instead.

uri_base
The base URI for the peer (typically scheme plus .hostinfo).

This raises *error.AnonymousHost* when executed on an address whose peer coordinates can not be expressed meaningfully in a URI.

uri_base_local
The base URI for the local side of this remote.

This raises *error.AnonymousHost* when executed on an address whose local coordinates can not be expressed meaningfully in a URI.

is_multicast
True if the remote address is a multicast address, otherwise false.

is_multicast_locally
True if the local address is a multicast address, otherwise false.

as_response_address()
Address to be assigned to a response to messages that arrived with this message

This can (and does, by default) return self, but gives the protocol the opportunity to react to create a modified copy to deal with variations from multicast.

blockwise_key
A hashable (ideally, immutable) value that is only the same for remotes from which blocks may be combined. (With all current transports that means that the network addresses need to be in there, and the identity of the security context).

It does *not* just hinge on the identity of the address object, as a first block may come in an OSCORE group request and follow-ups may come in pairwise requests. (And there might be allowed relaxations on the transport under OSCORE, but that'd need further discussion).

class aiocoap.transports.udp6.**SockExtendedErr**
Bases: aiocoap.transports.udp6._SockExtendedErr

classmethod load(*data*)

class aiocoap.transports.udp6.**MessageInterfaceUDP6**(*ctx: aiocoap.interfaces.MessageManager*, *log*, *loop*)
Bases: *aiocoap.util.asyncio.recvmsg.RecvmsgDatagramProtocol*, *aiocoap.interfaces.MessageInterface*

ready = None
Future that gets fullfilled by connection_made (ie. don't send before this is done; handled by create_...._context

send(*message*)
Send a given Message object

classmethod create_client_transport_endpoint(*ctx: aiocoap.interfaces.MessageManager*, *log*, *loop*)

classmethod create_server_transport_endpoint(*ctx: aiocoap.interfaces.MessageManager*, *log*, *loop*, *bind*, *multicast*)

determine_remote(*request*)
Return a value suitable for the message's remote property based on its .opt.uri_host or .unresolved_remote.

May return None, which indicates that the MessageInterface can not transport the message (typically because it is of the wrong scheme).

recognize_remote(*remote*)

shutdown()
Deactivate the complete transport, usually irrevertably. When the coroutine returns, the object must have made sure that it can be destructed by means of ref-counting or a garbage collector run.

connection_made(*transport*)
Implementation of the DatagramProtocol interface, called by the transport.

datagram_msg_received(*data*, *ancdata*, *flags*, *address*)
Implementation of the RecvmsgDatagramProtocol interface, called by the transport.

datagram_errqueue_received(*data*, *ancdata*, *flags*, *address*)
Called when some data is received from the error queue

error_received(*exc*)
Implementation of the DatagramProtocol interface, called by the transport.

connection_lost(*exc*)
Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

aiocoap.transports.ws module

This moduel implements a TokenInterface for CoAP over WebSockets.

As with CoAP-over-TCP, while the transport distinguishes a connection initiator ("WebSocket (and TCP) client") and a receiver ("WebSocket (and TCP) server"), both sides can take both roles in CoAP (ie. as a CoAP server and a CoAP client). As the WebSocket client can not possibly be connected to (even by the same server – once the connection is closed, it's gone and even a new one likely has a different port), aiocoap does not allow expressing their addresses in URIs (given they wouldn't serve their purpose as URLs and don't provide any stability either). Requests to a CoAP-over-WS client can be made by assigning the remote to an outgoing request.

Port choice

Unlike the other transports, CoAP-over-WS is specified with a privileged port (port 80) as the default port. This is impractical for aiocoap servers for two reasons:

- Unless explicitly configured, aiocoap is typically run as an unprivileged user (and has no provisions in place to receive a socket by other means than opening it).
- Where a CoAP-over-WS proxy is run, there is often a "proper" website running on the same port on a full HTTP server. That server is usually capable of forwarding requests, whereas the `websockets` module used by aiocoap is in no position to either serve websites nor to proxy to an underlying server.

The recommended setup is therefore to run a full web server at port 80, and configure it to proxy incoming requests for WebSockets at */.well-known/coap* to aiocoap's server, which defaults to binding to port 8683.

The port choice of outgoing connections, or the interpretation of the protocol's default port (ie. the port implied by `coap+ws://hostname/`) is of course unaffected by this.

Warning: Due to a shortcoming of aiocoap's way of specifying ports to bind to, if a port is explicitly stated to bind to, CoAP-over-WS will bind to that port plus 3000 (resulting in the abovementioned 8683 for 5683). If TLS server keys are given, the TLS server is launched on the next port after the HTTP server (typically 8684).

class aiocoap.transports.ws.**PoolKey**(*scheme*, *hostinfo*)
Bases: `tuple`

hostinfo
Alias for field number 1

scheme
Alias for field number 0

class aiocoap.transports.ws.**WSRemote**(*pool*, *connection*, *loop*, *log*, *, *scheme*, *local_hostinfo=None*, *remote_hostinfo=None*)
Bases: *aiocoap.transports.rfc8323common.RFC8323Remote*, *aiocoap.interfaces.EndpointAddress*

scheme = None

release()
Send Release message, (not implemented:) wait for connection to be actually closed by the peer.

Subclasses should extend this to await closing of the connection, especially if they'd get into lock-up states otherwise (was would WebSockets).

class aiocoap.transports.ws.**WSPool**(*tman*, *log*, *loop*)
Bases: *aiocoap.interfaces.TokenInterface*

classmethod create_transport(*tman: aiocoap.interfaces.TokenManager*, *log*, *loop*, ***, *client_credentials*, *server_bind=None*, *server_context=None*)

fill_or_recognize_remote(*message*)
Return True if the message is recognized to already have a .remote managedy by this TokenInterface, or return True and set a .remote on message if it should (by its unresolved remote or Uri-* options) be routed through this TokenInterface, or return False otherwise.

shutdown()

send_message(*message*, *messageerror_monitor*)
Send a message. If it returns a a callable, the caller is asked to call in case it no longer needs the message sent, and to dispose of if it doesn't intend to any more.

messageerror_monitor is a function that will be called at most once by the token interface: When the underlying layer is indicating that this concrete message could not be processed. This is typically the case for RSTs on from the message layer, and used to cancel observations. Errors that are not likely to be specific to a message (like retransmission timeouts, or ICMP errors) are reported through dispatch_error instead. (While the information which concrete message triggered that might be available, it is not likely to be relevant).

Currently, it is up to the TokenInterface to unset the no_response option in response messages, and to possibly not send them.

aiocoap.proxy module

Container module, see submodules:

- *client* – using CoAP via a proxy server
- *server* – running a proxy server

aiocoap.proxy.client module

class aiocoap.proxy.client.**ProxyForwarder**(*proxy_address*, *context*)
Bases: *aiocoap.interfaces.RequestProvider*

Object that behaves like a Context but only provides the request function and forwards all messages to a proxy.

This is not a proxy itself, it is just the interface for an external one.

proxy

request(*message*, ***kwargs*)
Create and act on a a `Request` object that will be handled according to the provider's implementation.

Note that the request is not necessarily sent on the wire immediately; it may (but, depend on the transport does not necessarily) rely on the response to be waited for.

aiocoap.proxy.server module

Basic implementation of CoAP-CoAP proxying

This is work in progress and not yet part of the API.

exception aiocoap.proxy.server.**CanNotRedirect**(*message=None*)
Bases: *aiocoap.error.ConstructionRenderableError*

message = 'Proxy redirection failed'

exception aiocoap.proxy.server.**NoUriSplitting**(*message=None*)
Bases: *aiocoap.proxy.server.CanNotRedirect*

code = <Response Code 161 "5.01 Not Implemented">

message = 'URI splitting not implemented, please use Proxy-Scheme.'

exception aiocoap.proxy.server.**IncompleteProxyUri**(*message=None*)
Bases: *aiocoap.proxy.server.CanNotRedirect*

code = <Response Code 128 "4.00 Bad Request">

message = 'Proxying requires Proxy-Scheme and Uri-Host'

exception aiocoap.proxy.server.**NotAForwardProxy**(*message=None*)
Bases: *aiocoap.proxy.server.CanNotRedirect*

code = <Response Code 165 "5.05 Proxying Not Supported">

message = 'This is a reverse proxy, not a forward one.'

exception aiocoap.proxy.server.**NoSuchHostname**(*message=None*)
Bases: *aiocoap.proxy.server.CanNotRedirect*

code = <Response Code 132 "4.04 Not Found">

message = ''

exception aiocoap.proxy.server.**CanNotRedirectBecauseOfUnsafeOptions**(*options*)
Bases: *aiocoap.proxy.server.CanNotRedirect*

code = <Response Code 130 "4.02 Bad Option">

aiocoap.proxy.server.**raise_unless_safe**(*request*, *known_options*)
Raise a BAD_OPTION CanNotRedirect unless all options in request are safe to forward or known

class aiocoap.proxy.server.**Proxy**(*outgoing_context*, *logger=None*)
Bases: *aiocoap.interfaces.Resource*

interpret_block_options = False

add_redirector(*redirector*)

apply_redirection(*request*)

render_to_pipe(*request: aiocoap.pipe.Pipe*)
Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then *render()*, *needs_blockwise_assembly()* and ObservableResource.add_observation() are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

needs_blockwise_assembly(*request*)
Indicator to the `protocol.Responder` about whether it should assemble request blocks to a single request and extract the requested blocks from a complete-resource answer (True), or whether the resource will do that by itself (False).

render(*request*)
Return a message that can be sent back to the requester.

This does not need to set any low-level message options like remote, token or message type; it does however need to set a response code.

A response returned may carry a no_response option (which is actually specified to apply to requests only); the underlying transports will decide based on that and its code whether to actually transmit the response.

class aiocoap.proxy.server.**ProxyWithPooledObservations**(*outgoing_context*, *logger=None*)
Bases: *aiocoap.proxy.server.Proxy*, *aiocoap.interfaces.ObservableResource*

add_observation(*request*, *serverobservation*)
As ProxiedResource is intended to be just the proxy's interface toward the Context, accepting observations is handled here, where the observations handling can be defined by the subclasses.

render(*request*)
Return a message that can be sent back to the requester.

This does not need to set any low-level message options like remote, token or message type; it does however need to set a response code.

A response returned may carry a no_response option (which is actually specified to apply to requests only); the underlying transports will decide based on that and its code whether to actually transmit the response.

class aiocoap.proxy.server.**ForwardProxy**(*outgoing_context*, *logger=None*)
Bases: *aiocoap.proxy.server.Proxy*

apply_redirection(*request*)

class aiocoap.proxy.server.**ForwardProxyWithPooledObservations**(*outgoing_context*, *logger=None*)
Bases: *aiocoap.proxy.server.ForwardProxy*, *aiocoap.proxy.server.ProxyWithPooledObservations*

class aiocoap.proxy.server.**ReverseProxy**(**args*, ***kwargs*)
Bases: *aiocoap.proxy.server.Proxy*

class aiocoap.proxy.server.**ReverseProxyWithPooledObservations**(**args*, ***kwargs*)
Bases: *aiocoap.proxy.server.ReverseProxy*, *aiocoap.proxy.server.ProxyWithPooledObservations*

class aiocoap.proxy.server.**Redirector**
Bases: `object`

apply_redirection(*request*)

class aiocoap.proxy.server.**NameBasedVirtualHost**(*match_name*, *target*, *rewrite_uri_host=False*, *use_as_proxy=False*)
Bases: *aiocoap.proxy.server.Redirector*

apply_redirection(*request*)

class aiocoap.proxy.server.**SubdomainVirtualHost**(**args*, ***kwargs*)
Bases: *aiocoap.proxy.server.NameBasedVirtualHost*

class aiocoap.proxy.server.**UnconditionalRedirector**(*target*, *use_as_proxy=False*)
Bases: *aiocoap.proxy.server.Redirector*

apply_redirection(*request*)

class aiocoap.proxy.server.**SubresourceVirtualHost**(*path*, *target*)
Bases: *aiocoap.proxy.server.Redirector*

apply_redirection(*request*)

aiocoap.numbers module

Module in which all meaningful numbers are collected. Most of the submodules correspond to IANA registries.

The contents of the *constants*, *types* and *codes* modules are accessible through this module directly; *contentformat*'s and *optionnumbers*' sole *contentformat.ContentFormat* and *optionnumbers.OptionNumber* classes are accessible in the same way.

aiocoap.numbers.codes module

List of known values for the CoAP "Code" field.

The values in this module correspond to the IANA registry "CoRE Parameters", subregistries "CoAP Method Codes" and "CoAP Response Codes".

The codes come with methods that can be used to get their rough meaning, see the *Code* class for details.

class aiocoap.numbers.codes.**Code**
Bases: *aiocoap.util.ExtensibleIntEnum*

Value for the CoAP "Code" field.

As the number range for the code values is separated, the rough meaning of a code can be determined using the *is_request()*, *is_response()* and *is_successful()* methods.

EMPTY = <Code 0 "EMPTY">

GET = <Request Code 1 "GET">

POST = <Request Code 2 "POST">

PUT = <Request Code 3 "PUT">

DELETE = <Request Code 4 "DELETE">

FETCH = <Request Code 5 "FETCH">

PATCH = <Request Code 6 "PATCH">

iPATCH = <Request Code 7 "iPATCH">

CREATED = <Successful Response Code 65 "2.01 Created">

DELETED = <Successful Response Code 66 "2.02 Deleted">

VALID = <Successful Response Code 67 "2.03 Valid">

CHANGED = <Successful Response Code 68 "2.04 Changed">

CONTENT = <Successful Response Code 69 "2.05 Content">

CONTINUE = <Successful Response Code 95 "2.31 Continue">

BAD_REQUEST = <Response Code 128 "4.00 Bad Request">

UNAUTHORIZED = <Response Code 129 "4.01 Unauthorized">

BAD_OPTION = <Response Code 130 "4.02 Bad Option">

FORBIDDEN = <Response Code 131 "4.03 Forbidden">

NOT_FOUND = <Response Code 132 "4.04 Not Found">

METHOD_NOT_ALLOWED = <Response Code 133 "4.05 Method Not Allowed">

NOT_ACCEPTABLE = <Response Code 134 "4.06 Not Acceptable">

REQUEST_ENTITY_INCOMPLETE = <Response Code 136 "4.08 Request Entity Incomplete">

CONFLICT = <Response Code 137 "4.09 Conflict">

PRECONDITION_FAILED = <Response Code 140 "4.12 Precondition Failed">

REQUEST_ENTITY_TOO_LARGE = <Response Code 141 "4.13 Request Entity Too Large">

UNSUPPORTED_CONTENT_FORMAT = <Response Code 143 "4.15 Unsupported Content Format">

UNSUPPORTED_MEDIA_TYPE

UNPROCESSABLE_ENTITY = <Response Code 150 "4.22 Unprocessable Entity">

TOO_MANY_REQUESTS = <Response Code 157 "4.29 Too Many Requests">

INTERNAL_SERVER_ERROR = <Response Code 160 "5.00 Internal Server Error">

NOT_IMPLEMENTED = <Response Code 161 "5.01 Not Implemented">

BAD_GATEWAY = <Response Code 162 "5.02 Bad Gateway">

SERVICE_UNAVAILABLE = <Response Code 163 "5.03 Service Unavailable">

GATEWAY_TIMEOUT = <Response Code 164 "5.04 Gateway Timeout">

PROXYING_NOT_SUPPORTED = <Response Code 165 "5.05 Proxying Not Supported">

HOP_LIMIT_REACHED = <Response Code 168 "5.08 Hop Limit Reached">

CSM = <Code 225 "7.01 Csm">

PING = <Code 226 "7.02 Ping">

PONG = <Code 227 "7.03 Pong">

RELEASE = <Code 228 "7.04 Release">

ABORT = <Code 229 "7.05 Abort">

is_request()
: True if the code is in the request code range

is_response()
: True if the code is in the response code range

is_signalling()

is_successful()
: True if the code is in the successful subrange of the response code range

can_have_payload()
: True if a message with that code can carry a payload. This is not checked for strictly, but used as an indicator.

class_
: The class of a code (distinguishing whether it's successful, a request or a response error or more).

```
>>> Code.CONTENT
<Successful Response Code 69 "2.05 Content">
>>> Code.CONTENT.class_
2
>>> Code.BAD_GATEWAY
<Response Code 162 "5.02 Bad Gateway">
>>> Code.BAD_GATEWAY.class_
5
```

dotted
The numeric value three-decimal-digits (c.dd) form

name_printable
The name of the code in human-readable form

name
The constant name of the code (equals name_printable readable in all-caps and with underscores)

aiocoap.numbers.constants module

Constants either defined in the CoAP protocol (often default values for lack of ways to determine eg. the estimated round trip time). Some parameters are invented here for practical purposes of the implementation (eg. DEFAULT_BLOCK_SIZE_EXP, EMPTY_ACK_DELAY).

aiocoap.numbers.constants.**COAP_PORT = 5683**
The IANA-assigned standard port for COAP services.

aiocoap.numbers.constants.**ACK_TIMEOUT = 2.0**
The time, in seconds, to wait for an acknowledgement of a confirmable message. The inter-transmission time doubles for each retransmission.

aiocoap.numbers.constants.**ACK_RANDOM_FACTOR = 1.5**
Timeout multiplier for anti-synchronization.

aiocoap.numbers.constants.**MAX_RETRANSMIT = 4**
The number of retransmissions of confirmable messages to non-multicast endpoints before the infrastructure assumes no acknowledgement will be received.

aiocoap.numbers.constants.**NSTART = 1**
Maximum number of simultaneous outstanding interactions that endpoint maintains to a given server (including proxies)

aiocoap.numbers.constants.**MAX_TRANSMIT_SPAN = 45.0**
Maximum time from the first transmission of a confirmable message to its last retransmission.

aiocoap.numbers.constants.**MAX_TRANSMIT_WAIT = 93.0**
Maximum time from the first transmission of a confirmable message to the time when the sender gives up on receiving an acknowledgement or reset.

aiocoap.numbers.constants.**MAX_LATENCY = 100.0**
Maximum time a datagram is expected to take from the start of its transmission to the completion of its reception.

aiocoap.numbers.constants.**PROCESSING_DELAY = 2.0**
“Time a node takes to turn around a confirmable message into an acknowledgement.

aiocoap.numbers.constants.**MAX_RTT = 202.0**
Maximum round-trip time.

aiocoap.numbers.constants.**EXCHANGE_LIFETIME = 247.0**
time from starting to send a confirmable message to the time when an acknowledgement is no longer expected, i.e. message layer information about the message exchange can be purged

aiocoap.numbers.constants.**DEFAULT_BLOCK_SIZE_EXP = 6**
Default size exponent for blockwise transfers.

aiocoap.numbers.constants.**EMPTY_ACK_DELAY = 0.1**
After this time protocol sends empty ACK, and separate response

aiocoap.numbers.constants.**REQUEST_TIMEOUT = 93.0**
Time after which server assumes it won't receive any answer. It is not defined by IETF documents. For human-operated devices it might be preferable to set some small value (for example 10 seconds) For M2M it's application dependent.

aiocoap.numbers.constants.**OBSERVATION_RESET_TIME = 128**
Time in seconds after which the value of the observe field are ignored.

This number is not explicitly named in RFC7641.

aiocoap.numbers.constants.**SHUTDOWN_TIMEOUT = 3**
Maximum time, in seconds, for which the process is kept around during shutdown

aiocoap.numbers.contentformat module

Module containing the CoRE parameters / CoAP Content-Formats registry

class aiocoap.numbers.contentformat.**ContentFormat**
Bases: *aiocoap.util.ExtensibleIntEnum*

Entry in the CoAP Content-Formats registry of the IANA Constrained RESTful Environments (Core) Parameters group

Known entries have `.media_type` and `.encoding` attributes:

```
>>> ContentFormat(0).media_type
'text/plain; charset=utf-8'
>>> int(ContentFormat.by_media_type('text/plain;charset=utf-8'))
0
>>> ContentFormat(60)
<ContentFormat 60, media_type='application/cbor', encoding='identity'>
>>> ContentFormat(11060).encoding
'deflate'
```

Unknown entries do not have these properties:

```
>>> ContentFormat(12345).is_known()
False
>>> ContentFormat(12345).media_type                    # doctest: +ELLIPSIS
Traceback (most recent call last):
    ...
AttributeError: ...
```

Only a few formats are available as attributes for easy access. Their selection and naming are arbitrary and biased. The remaining known types are available through the *by_media_type()* class method. >>> ContentFormat.TEXT <ContentFormat 0, media_type='text/plain; charset=utf-8', encoding='identity'>

A convenient property of ContentFormat is that any known content format is true in a boolean context, and thus when used in alternation with None, can be assigned defaults easily:

```
>>> requested_by_client = ContentFormat.TEXT
>>> int(requested_by_client) # Usually, this would always pick the default
0
>>> used = requested_by_client or ContentFormat.LINKFORMAT
>>> assert used == ContentFormat.TEXT
```

classmethod by_media_type(*media_type: str*, *encoding: str = 'identity'*) → aiocoap.numbers.contentformat.ContentFormat

Produce known entry for a known media type (and encoding, though 'identity' is default due to its prevalence), or raise KeyError.

is_known()

TEXT = <ContentFormat 0, media_type='text/plain; charset=utf-8', encoding='identity'>

LINKFORMAT = <ContentFormat 40, media_type='application/link-format', encoding='identi

OCTETSTREAM = <ContentFormat 42, media_type='application/octet-stream', encoding='iden

JSON = <ContentFormat 50, media_type='application/json', encoding='identity'>

CBOR = <ContentFormat 60, media_type='application/cbor', encoding='identity'>

SENML = <ContentFormat 112, media_type='application/senml+cbor', encoding='identity'>

aiocoap.numbers.optionnumbers module

Known values for CoAP option numbers

The values defined in *OptionNumber* correspond to the IANA registry "CoRE Parameters", subregistries "CoAP Method Codes" and "CoAP Response Codes".

The option numbers come with methods that can be used to evaluate their properties, see the *OptionNumber* class for details.

class aiocoap.numbers.optionnumbers.**OptionNumber**

Bases: *aiocoap.util.ExtensibleIntEnum*

A CoAP option number.

As the option number contains information on whether the option is critical, and whether it is safe-to-forward, those properties can be queried using the *is_** group of methods.

Note that whether an option may be repeated or not does not only depend on the option, but also on the context, and is thus handled in the *Options* object instead.

IF_MATCH = <OptionNumber 1 "IF_MATCH">

URI_HOST = <OptionNumber 3 "URI_HOST">

ETAG = <OptionNumber 4 "ETAG">

IF_NONE_MATCH = <OptionNumber 5 "IF_NONE_MATCH">

OBSERVE = <OptionNumber 6 "OBSERVE">

URI_PORT = <OptionNumber 7 "URI_PORT">

LOCATION_PATH = <OptionNumber 8 "LOCATION_PATH">

OSCORE = <OptionNumber 9 "OBJECT_SECURITY">

OBJECT_SECURITY = <OptionNumber 9 "OBJECT_SECURITY">

URI_PATH = <OptionNumber 11 "URI_PATH">

CONTENT_FORMAT = <OptionNumber 12 "CONTENT_FORMAT">

MAX_AGE = <OptionNumber 14 "MAX_AGE">

URI_QUERY = <OptionNumber 15 "URI_QUERY">

HOP_LIMIT = <OptionNumber 16 "HOP_LIMIT">

ACCEPT = <OptionNumber 17 "ACCEPT">

Q_BLOCK1 = <OptionNumber 19 "Q_BLOCK1">

LOCATION_QUERY = <OptionNumber 20 "LOCATION_QUERY">

EDHOC = <OptionNumber 21 "EDHOC">

BLOCK2 = <OptionNumber 23 "BLOCK2">

BLOCK1 = <OptionNumber 27 "BLOCK1">

SIZE2 = <OptionNumber 28 "SIZE2">

Q_BLOCK2 = <OptionNumber 31 "Q_BLOCK2">

PROXY_URI = <OptionNumber 35 "PROXY_URI">

PROXY_SCHEME = <OptionNumber 39 "PROXY_SCHEME">

SIZE1 = <OptionNumber 60 "SIZE1">

ECHO = <OptionNumber 252 "ECHO">

NO_RESPONSE = <OptionNumber 258 "NO_RESPONSE">

REQUEST_TAG = <OptionNumber 292 "REQUEST_TAG">

REQUEST_HASH = <OptionNumber 548 "REQUEST_HASH">

is_critical()

is_elective()

is_unsafe()

is_safetoforward()

is_nocachekey()

is_cachekey()

format

create_option(*decode=None*, *value=None*)

Return an Option element of the appropriate class from this option number.

An initial value may be set using the decode or value options, and will be fed to the resulting object's decode method or value property, respectively.

aiocoap.numbers.types module

List of known values for the CoAP "Type" field.

As this field is only 2 bits, its valid values are comprehensively enumerated in the *Type* object.

class aiocoap.numbers.types.**Type**
Bases: enum.IntEnum

An enumeration.

CON = 0

NON = 1

ACK = 2

RST = 3

aiocoap.optiontypes module

class aiocoap.optiontypes.**OptionType**(*number*, *value*)
Bases: object

Interface for decoding and encoding option values

Instances of *OptionType* are collected in a list in a Message.opt *Options* object, and provide a translation between the CoAP octet-stream (accessed using the *encode()*/*decode()* method pair) and the interpreted value (accessed via the value attribute).

Note that OptionType objects usually don't need to be handled by library users; the recommended way to read and set options is via the Options object'sproperties (eg. `message.opt.uri_path = ('.well-known', 'core')`).

encode()
Return the option's value in serialzied form

decode(*rawdata*)
Set the option's value from the bytes in rawdata

class aiocoap.optiontypes.**StringOption**(*number*, *value=''*)
Bases: *aiocoap.optiontypes.OptionType*

String CoAP option - used to represent string options. Always encoded in UTF8 per CoAP specification.

encode()
Return the option's value in serialzied form

decode(*rawdata*)
Set the option's value from the bytes in rawdata

class aiocoap.optiontypes.**OpaqueOption**(*number*, *value=b''*)
Bases: *aiocoap.optiontypes.OptionType*

Opaque CoAP option - used to represent options that just have their uninterpreted bytes as value.

encode()
Return the option's value in serialzied form

decode(*rawdata*)
Set the option's value from the bytes in rawdata

class aiocoap.optiontypes.**UintOption**(*number*, *value=0*)
Bases: *aiocoap.optiontypes.OptionType*

Uint CoAP option - used to represent integer options.

encode()
Return the option's value in serialzied form

decode(*rawdata*)
Set the option's value from the bytes in rawdata

class aiocoap.optiontypes.**TypedOption**(*number*, *value=None*)
Bases: *aiocoap.optiontypes.OptionType*

type
Checked type of the option

value

class aiocoap.optiontypes.**BlockOption**(*number*, *value=None*)
Bases: *aiocoap.optiontypes.TypedOption*

Block CoAP option - special option used only for Block1 and Block2 options. Currently it is the only type of CoAP options that has internal structure.

That structure (BlockwiseTuple) covers not only the block options of RFC7959, but also the BERT extension of RFC8323. If the reserved size exponent 7 is used for purposes incompatible with BERT, the implementor might want to look at the context dependent option number interpretations which will hopefully be in place for Signaling (7.xx) messages by then.

class BlockwiseTuple
Bases: aiocoap.optiontypes._BlockwiseTuple

size

start
The byte offset in the body indicated by block number and size.

Note that this calculation is only valid for descriptive use and Block2 control use. The semantics of block_number and size in Block1 control use are unrelated (indicating the acknowledged block number in the request Block1 size and the server's preferred block size), and must not be calculated using this property in that case.

is_bert
True if the exponent is recognized to signal a BERT message.

is_valid_for_payload_size(*payloadsize*)

reduced_to(*maximum_exponent*)
Return a BlockwiseTuple whose exponent is capped to the given maximum_exponent

```
>>> initial = BlockOption.BlockwiseTuple(10, 0, 5)
>>> initial == initial.reduced_to(6)
True
>>> initial.reduced_to(3)
BlockwiseTuple(block_number=40, more=0, size_exponent=3)
```

type
alias of *BlockOption.BlockwiseTuple*

encode()
Return the option's value in serialzied form

decode(*rawdata*)
Set the option's value from the bytes in rawdata

class aiocoap.optiontypes.**ContentFormatOption**(*number*, *value=None*)
Bases: *aiocoap.optiontypes.TypedOption*

Type of numeric options whose number has ContentFormat semantics

type
alias of *aiocoap.numbers.contentformat.ContentFormat*

encode()
Return the option's value in serialzied form

decode(*rawdata*)
Set the option's value from the bytes in rawdata

aiocoap.resource module

Basic resource implementations

A resource in URL / CoAP / REST terminology is the thing identified by a URI.

Here, a *Resource* is the place where server functionality is implemented. In many cases, there exists one persistent Resource object for a given resource (eg. a `TimeResource()` is responsible for serving the `/time` location). On the other hand, an aiocoap server context accepts only one thing as its serversite, and that is a Resource too (typically of the *Site* class).

Resources are most easily implemented by deriving from *Resource* and implementing `render_get`, `render_post` and similar coroutine methods. Those take a single request message object and must return a *aiocoap.Message* object or raise an *error.RenderableError* (eg. `raise UnsupportedMediaType()`).

To serve more than one resource on a site, use the *Site* class to dispatch requests based on the Uri-Path header.

aiocoap.resource.**hashing_etag**(*request*, *response*)
Helper function for render_get handlers that allows them to use ETags based on the payload's hash value

Run this on your request and response before returning from render_get; it is safe to use this function with all kinds of responses, it will only act on 2.05 Content messages (and those with no code set, which defaults to that for GET requests). The hash used are the first 8 bytes of the sha1 sum of the payload.

Note that this method is not ideal from a server performance point of view (a file server, for example, might want to hash only the stat() result of a file instead of reading it in full), but it saves bandwith for the simple cases.

```
>>> from aiocoap import *
>>> req = Message(code=GET)
>>> hash_of_hello = b'\xaa\xf4\xc6\x1d\xdc\xc5\xe8\xa2'
>>> req.opt.etags = [hash_of_hello]
>>> resp = Message(code=CONTENT)
>>> resp.payload = b'hello'
>>> hashing_etag(req, resp)
>>> resp                                                    # doctest: +ELLIPSIS
<aiocoap.Message at ... 2.03 Valid ... 1 option(s)>
```

class aiocoap.resource.**Resource**
Bases: aiocoap.resource._ExposesWellknownAttributes, *aiocoap.interfaces.Resource*

Simple base implementation of the `interfaces.Resource` interface

The render method delegates content creation to `render_$method` methods (`render_get`, `render_put` etc), and responds appropriately to unsupported methods. Those messages may return messages without a response code, the default render method will set an appropriate successful code ("Content" for GET/FETCH, "Deleted" for DELETE, "Changed" for anything else). The render method will also fill in the request's no_response code into the response (see *interfaces.Resource.render()*) if none was set.

Moreover, this class provides a `get_link_description` method as used by .well-known/core to expose a resource's `.ct`, `.rt` and `.if_` (alternative name for `if` as that's a Python keyword) attributes. Details can be added by overriding the method to return a more comprehensive dictionary, and resources can be hidden completely by returning None.

needs_blockwise_assembly(*request*)
Indicator to the `protocol.Responder` about whether it should assemble request blocks to a single request and extract the requested blocks from a complete-resource answer (True), or whether the resource will do that by itself (False).

render(*request*)
Return a message that can be sent back to the requester.

This does not need to set any low-level message options like remote, token or message type; it does however need to set a response code.

A response returned may carry a no_response option (which is actually specified to apply to requests only); the underlying transports will decide based on that and its code whether to actually transmit the response.

render_to_pipe(*request: aiocoap.pipe.Pipe*)
Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then *render()*, *needs_blockwise_assembly()* and *ObservableResource.add_observation()* are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

class aiocoap.resource.**ObservableResource**
Bases: *aiocoap.resource.Resource*, *aiocoap.interfaces.ObservableResource*

update_observation_count(*newcount*)
Hook into this method to be notified when the number of observations on the resource changes.

updated_state(*response=None*)
Call this whenever the resource was updated, and a notification should be sent to observers.

get_link_description()

add_observation(*request*, *serverobservation*)
Before the incoming request is sent to *render()*, the *add_observation()* method is called. If the resource chooses to accept the observation, it has to call the *serverobservation.accept(cb)* with a callback that will be called when the observation ends. After accepting, the ObservableResource should call *serverobservation.trigger()* whenever it changes its state; the ServerObservation will then initiate notifications by having the request rendered again.

render_to_pipe(*request: aiocoap.pipe.Pipe*)
Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then `render()`, `needs_blockwise_assembly()` and *ObservableResource.add_observation()* are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

aiocoap.resource.**link_format_to_message**(*request*, *linkformat*, *default_ct=<ContentFormat 40*, *media_type='application/link-format'*, *encoding='identity'>*)
Given a LinkFormat object, render it to a response message, picking a suitable conent format from a given request.

It returns a Not Acceptable response if something unsupported was queried.

It makes no attempt to modify the URI reference literals encoded in the LinkFormat object; they have to be suitably prepared by the caller.

class aiocoap.resource.**WKCResource**(*listgenerator*, *impl_info='https://christian.amsuess.com/tools/aiocoap/#version-0.4.5'*, ***kwargs*)

Bases: *aiocoap.resource.Resource*

Read-only dynamic resource list, suitable as .well-known/core.

This resource renders a link_header.LinkHeader object (which describes a collection of resources) as application/link-format (RFC 6690).

The list to be rendered is obtained from a function passed into the constructor; typically, that function would be a bound Site.get_resources_as_linkheader() method.

This resource also provides server implementation information link; server authors are invited to override this by passing an own URI as the *impl_info* parameter, and can disable it by passing None.

ct = '40'

render_get(*request*)

class aiocoap.resource.**PathCapable**

Bases: object

Class that indicates that a resource promises to parse the uri_path option, and can thus be given requests for *render()*-ing that contain a uri_path

class aiocoap.resource.**Site**

Bases: *aiocoap.interfaces.ObservableResource*, *aiocoap.resource.PathCapable*

Typical root element that gets passed to a Context and contains all the resources that can be found when the endpoint gets accessed as a server.

This provides easy registration of statical resources. Add resources at absolute locations using the *add_resource()* method.

For example, the site at

```
>>> site = Site()
>>> site.add_resource(["hello"], Resource())
```

will have requests to </hello> rendered by the new resource.

You can add another Site (or another instance of *PathCapable*) as well, those will be nested and integrally reported in a WKCResource. The path of a site should not end with an empty string (ie. a slash in the URI) – the child site's own root resource will then have the trailing slash address. Subsites can not have link-header attributes on their own (eg. *rt*) and will never respond to a request that does not at least contain a single slash after the the given path part.

For example,

```
>>> batch = Site()
>>> batch.add_resource(["light1"], Resource())
>>> batch.add_resource(["light2"], Resource())
>>> batch.add_resource([], Resource())
>>> s = Site()
>>> s.add_resource(["batch"], batch)
```

will have the three created resources rendered at </batch/light1>, </batch/light2> and </batch/>.

If it is necessary to respond to requests to </batch> or report its attributes in .well-known/core in addition to the above, a non-PathCapable resource can be added with the same path. This is usually considered an odd design, not fully supported, and for example doesn't support removal of resources from the site.

add_observation(*request*, *serverobservation*)
Before the incoming request is sent to *render()*, the *add_observation()* method is called. If the resource chooses to accept the observation, it has to call the *serverobservation.accept(cb)* with a callback that will be called when the observation ends. After accepting, the ObservableResource should call *serverobservation.trigger()* whenever it changes its state; the ServerObservation will then initiate notifications by having the request rendered again.

needs_blockwise_assembly(*request*)
Indicator to the `protocol.Responder` about whether it should assemble request blocks to a single request and extract the requested blocks from a complete-resource answer (True), or whether the resource will do that by itself (False).

render(*request*)
Return a message that can be sent back to the requester.

This does not need to set any low-level message options like remote, token or message type; it does however need to set a response code.

A response returned may carry a no_response option (which is actually specified to apply to requests only); the underlying transports will decide based on that and its code whether to actually transmit the response.

render_to_pipe(*request: aiocoap.pipe.Pipe*)
Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then *render()*, *needs_blockwise_assembly()* and *ObservableResource.add_observation()* are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

add_resource(*path*, *resource*)

remove_resource(*path*)

get_resources_as_linkheader()

aiocoap.util module

Tools not directly related with CoAP that are needed to provide the API

These are only part of the stable API to the extent they are used by other APIs – for example, you can use the type constructor of *ExtensibleEnumMeta* when creating an *aiocoap.numbers.optionnumbers.OptionNumber*, but don't expect it to be usable in a stable way for own extensions.

Most functions are available in submodules; some of them may only have components that are exclusively used internally and never part of the public API even in the limited fashion stated above.

aiocoap.util.asyncio module

Extensions to asyncio and workarounds around its shortcomings

`aiocoap.util.asyncio.`**py38args**(***kwargs*)
Wrapper around kwargs that replaces them with an empty list for Python versions earlier than 3.8.

This is used to assign a name in asyncio.create_task to pass in a name.

aiocoap.util.asyncio.recvmsg module

class aiocoap.util.asyncio.recvmsg.**RecvmsgDatagramProtocol**
Bases: asyncio.protocols.BaseProtocol

Callback interface similar to asyncio.DatagramProtocol, but dealing with recvmsg data.

datagram_msg_received(*data*, *ancdata*, *flags*, *address*)
Called when some datagram is received.

datagram_errqueue_received(*data*, *ancdata*, *flags*, *address*)
Called when some data is received from the error queue

error_received(*exc*)
Called when a send or receive operation raises an OSError.

class aiocoap.util.asyncio.recvmsg.**RecvmsgSelectorDatagramTransport**(*loop*, *sock*, *protocol*, *waiter*)
Bases: asyncio.transports.BaseTransport

A simple loop-independent transport that largely mimicks DatagramTransport but interfaces a RecvmsgSelectorDatagramProtocol.

This does not implement any flow control, based on the assumption that it's not needed, for CoAP has its own flow control mechanisms.

max_size = 4096

close()
Close the transport.

Buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's connection_lost() method will (eventually) called with None as its argument.

sendmsg(*data*, *ancdata*, *flags*, *address*)

aiocoap.util.asyncio.recvmsg.**create_recvmsg_datagram_endpoint**(*loop*, *factory*, *sock*)
Create a datagram connection that uses recvmsg rather than recvfrom, and a RecvmsgDatagramProtocol protocol type.

This is used like the create_datagram_endpoint method of an asyncio loop, but implemented in a generic way using the loop's add_reader method; thus, it's not a method of the loop but an independent function.

Due to the way it is used in aiocoap, socket is not an optional argument here; it could be were this module ever split off into a standalone package.

aiocoap.util.asyncio.timeoutdict module

class aiocoap.util.asyncio.timeoutdict.**TimeoutDict**(*timeout: float*)
Bases: object

A dict-ish type whose entries live on a timeout; adding and accessing an item each refreshes the timeout.

The timeout is a lower bound; items may live up to twice as long.

The container is implemented incompletely, with additions made on demand.

This is not thread safe.

timeout = None
Timeout set on any access

This can be changed at runtime, but changes only take effect

aiocoap.util.cli module

Helpers for creating server-style applications in aiocoap

Note that these are not particular to aiocoap, but are used at different places in aiocoap and thus shared here.

class aiocoap.util.cli.**ActionNoYes**(*option_strings*, *dest*, *default=True*, *required=False*, *help=None*)
Bases: argparse.Action

Simple action that automatically manages –{,no-}something style options

class aiocoap.util.cli.**AsyncCLIDaemon**(**args*, ***kwargs*)
Bases: object

Helper for creating daemon-style CLI prorgrams.

Note that this currently doesn't create a Daemon in the sense of doing a daemon-fork; that could be added on demand, though.

Subclass this and implement the `start()` method as an async function; it will be passed all the constructor's arguments.

When all setup is complete and the program is operational, return from the start method.

Implement the `shutdown()` coroutine and to do cleanup; what actually runs your program will, if possible, call that and await its return.

Two usage patterns for this are supported:

- Outside of an async context, run run `MyClass.sync_main()`, typically in the program's `if __name__ == "__main__":` section.

  In this mode, the loop that is started is configured to safely shut down the loop when SIGINT is received.

- To run a subclass of this in an existing loop, start it with `MyClass(...)` (possibly passing in the loop to run it on if not already in an async context), and then awaiting its `.initializing` future. To stop it, await its `.shutdown()` method.

  Note that with this usage pattern, the *`stop()`* method has no effect; servers that `.stop()` themselves need to signal their desire to be shut down through other channels (but that is an atypical case).

stop(*exitcode*)
Stop the operation (and exit sync_main) at the next convenience.

classmethod sync_main(**args*, ***kwargs*)
Run the application in an AsyncIO main loop, shutting down cleanly on keyboard interrupt.

aiocoap.util.contenttype module

Helpers around content types

This uses the terminology clarified in 1, and primarily deals with content types in their usual string representation.

Unless content types get used a lot more in aiocoap, this provides only accessors to some of their relevant properties, without aiming to build semantically accessible objects to encapsulate them.

aiocoap.util.contenttype.**categorize**(*contenttype: str*)
Return 'cbor', 'json' or 'link-format' if the content type indicates it is that format itself or derived from it.

aiocoap.util.cryptography_additions module

Workaround for https://github.com/pyca/cryptography/issues/5557

These functions could be methods to *cryptography.hazmat.primitives.asymmetric.ed25519.{Ed25519PrivateKey, Ed25519PublicKey}*, respectively, and are currently implemented manually or using ge25519.

These conversions are not too critical in that they do not run on data an attacker can send arbitrarily (in the most dynamic situation, the keys are distributed through a KDC aka. group manager).

aiocoap.util.cryptography_additions.**sk_to_curve25519**(*ed: cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519Pri* → cryptography.hazmat.primitives.asymmetric.x25519.X25519Priva

aiocoap.util.cryptography_additions.**pk_to_curve25519**(*ed: cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519Pu* → cryptography.hazmat.primitives.asymmetric.x25519.X25519Publi

aiocoap.util.linkformat module

This module contains in-place modifications to the LinkHeader module to satisfy RFC6690 constraints.

It is a general nursery for what aiocoap needs of link-format management before any of this is split out into its own package.

class aiocoap.util.linkformat.**LinkFormat**(*links=None*)
Bases: link_header.LinkHeader

class aiocoap.util.linkformat.**Link**(*href*, *attr_pairs=None*, ***kwargs*)
Bases: link_header.Link

aiocoap.util.linkformat.**parse**(*linkformat*)

aiocoap.util.linkformat_pygments module

class aiocoap.util.linkformat_pygments.**LinkFormatLexer**(***options*)
Bases: pygments.lexer.RegexLexer

name = 'LinkFormatLexer'

mimetypes = ['application/link-format']

tokens = {'attribute': [('([^,;=]+)((=)("[^"]*"|[^,;"]+))?', <function bygroups.<loca

aiocoap.util.prettyprint module

A pretty-printer for known mime types

aiocoap.util.prettyprint.**lexer_for_mime**(*mime*)
A wrapper around pygments.lexers.get_lexer_for_mimetype that takes subtypes into consideration and catches the custom hexdump mime type.

aiocoap.util.prettyprint.**pretty_print**(*message*)
Given a CoAP message, reshape its payload into something human-readable. The return value is a triple (infos, mime, text) where text represents the payload, mime is a type that could be used to syntax-highlight the text (not necessarily related to the original mime type, eg. a report of some binary data that's shaped like Markdown could use a markdown mime type), and some line of infos that give additional data (like the reason for a hex dump or the original mime type).

```
>>> from aiocoap import Message
>>> def build(payload, request_cf, response_cf):
...     response = Message(payload=payload, content_format=response_cf)
...     request = Message(accept=request_cf)
...     response.request = request
...     return response
>>> pretty_print(Message(payload=b"Hello", content_format=0))
([], 'text/plain;charset=utf8', 'Hello')
>>> print(pretty_print(Message(payload=b'{"hello":"world"}', content_format=50))[-
↪1])
{
    "hello": "world"
}
>>> # Erroneous inputs still go to the pretty printer as long as they're
>>> #Unicode
>>> pretty_print(Message(payload=b'{"hello":"world', content_format=50))
(['Invalid JSON not re-formated'], 'application/json', '{"hello":"world')
>>> pretty_print(Message(payload=b'<>,', content_format=40))
(['Invalid application/link-format content was not re-formatted'], 'application/
↪link-format', '<>,')
>>> pretty_print(Message(payload=b'a', content_format=60)) # doctest: +ELLIPSIS
(['Showing hex dump of application/cbor payload: CBOR value is invalid'], 'text/
↪vnd.aiocoap.hexdump', '00000000  61 ...
```

aiocoap.util.socknumbers module

This module contains numeric constants that would be expected in the socket module, but are not exposed there.

This gathers both socket numbers that can be present in the socket module (eg. the PKTINFO constants) but are not in some versions (eg. on macOS before <https://bugs.python.org/issue35569> is fixed) and platform dependent constants that are not generally available at all (the ERR constants).

Where available, the CPython-private IN module is used to obtain some platform specific constants.

Any hints on where to get them from in a more reliable way are appreciated; possible options are parsing C header files (at build time?) or interacting with shared libraries for obtaining the symbols. The right way would probably be including them in Python in a "other constants defined on this platform for sockets" module or dictionary.

aiocoap.util.socknumbers.**HAS_RECVERR = True**
Indicates whether the discovered constants indicate that the Linux *setsockopt(IPV6, RECVERR)* / *recvmsg(..., MSG_ERRQUEUE)* mechanism is available

aiocoap.util.uri module

Tools that I'd like to have in urllib.parse

aiocoap.util.uri.**unreserved = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456**
"unreserved" characters from RFC3986

aiocoap.util.uri.**sub_delims = "!$&'()*+,;="**
"sub-delims" characters from RFC3986

aiocoap.util.uri.**quote_factory**(*safe_characters*)
Return a quote function that escapes all characters not in the safe_characters iterable.

class aiocoap.util.**ExtensibleEnumMeta**(*name*, *bases*, *dict*)
Bases: type

Metaclass for ExtensibleIntEnum, see there for detailed explanations

class aiocoap.util.**ExtensibleIntEnum**
Bases: int

Similar to Python's enum.IntEnum, this type can be used for named numbers which are not comprehensively known, like CoAP option numbers.

aiocoap.util.**hostportjoin**(*host*, *port=None*)
Join a host and optionally port into a hostinfo-style host:port string

```
>>> hostportjoin('example.com')
'example.com'
>>> hostportjoin('example.com', 1234)
'example.com:1234'
>>> hostportjoin('127.0.0.1', 1234)
'127.0.0.1:1234'
```

This is lax with respect to whether host is an IPv6 literal in brackets or not, and accepts either form; IP-future literals that do not contain a colon must be already presented in their bracketed form:

```
>>> hostportjoin('2001:db8::1')
'[2001:db8::1]'
>>> hostportjoin('2001:db8::1', 1234)
'[2001:db8::1]:1234'
>>> hostportjoin('[2001:db8::1]', 1234)
'[2001:db8::1]:1234'
```

aiocoap.util.**hostportsplit**(*hostport*)
Like urllib.parse.splitport, but return port as int, and as None if not given. Also, it allows giving IPv6 addresses like a netloc:

```
>>> hostportsplit('foo')
('foo', None)
>>> hostportsplit('foo:5683')
('foo', 5683)
>>> hostportsplit('[::1%eth0]:56830')
('::1%eth0', 56830)
```

aiocoap.util.**quote_nonascii**(*s*)
Like urllib.parse.quote, but explicitly only escaping non-ascii characters.

This function is deprecated due to it use of the irrelevant "being an ASCII character" property (when instead RFC3986 productions like "unreserved" should be used), and due for removal when aiocoap's URI processing is overhauled the next time.

class aiocoap.util.**Sentinel**(*label*)
Bases: object

Class for sentinel that can only be compared for identity. No efforts are taken to make these singletons; it is up to the users to always refer to the same instance, which is typically defined on module level.

aiocoap.cli module

Container module for command line utilities bundled with aiocoap.

These modules are not considered to be a part of the aioCoAP API, and are thus subject to change even when the project reaches a stable version number. If you want to use any of that infrastructure, please file a feature request for stabilization in the project's issue tracker.

The tools themselves are documented in *CoAP tools*.

aiocoap.meta module

aiocoap.meta.**version = '0.4.5'**
Make library version internally

This is not supposed to be used in any decision-making process (use package dependencies for that) or workarounds, but used by command-line tools or the impl-info link to provide debugging information.

aiocoap.meta.**library_uri = 'https://christian.amsuess.com/tools/aiocoap/#version-0.4.5'**
URI used to describe the current version of the library

This is used the same way as *version* but when a URI is required, for example as a default value for .well-known/core's rel=impl-info link.

aiocoap.oscore module

This module contains the tools to send OSCORE secured messages.

It only deals with the algorithmic parts, the security context and protection and unprotection of messages. It does not touch on the integration of OSCORE in the larger aiocoap stack of having a context or requests; that's what `aiocoap.transports.osore` is for.'

exception aiocoap.oscore.**NotAProtectedMessage**(*message*, *plain_message*)
Bases: *aiocoap.error.Error*, ValueError

Raised when verification is attempted on a non-OSCORE message

exception aiocoap.oscore.**ProtectionInvalid**
Bases: *aiocoap.error.Error*, ValueError

Raised when verification of an OSCORE message fails

exception aiocoap.oscore.**DecodeError**
Bases: *aiocoap.oscore.ProtectionInvalid*

Raised when verification of an OSCORE message fails because CBOR or compressed data were erroneous

exception aiocoap.oscore.**ReplayError**
Bases: *aiocoap.oscore.ProtectionInvalid*

Raised when verification of an OSCORE message fails because the sequence numbers was already used

exception aiocoap.oscore.**ReplayErrorWithEcho**(*secctx*, *request_id*, *echo*)
Bases: *aiocoap.oscore.ProtectionInvalid*, *aiocoap.error.RenderableError*

Raised when verification of an OSCORE message fails because the recipient replay window is uninitialized, but a 4.01 Echo can be constructed with the data in the exception that can lead to the client assisting in replay window recovery

to_message()
Create a CoAP message that should be sent when this exception is rendered

exception aiocoap.oscore.**ContextUnavailable**
Bases: *aiocoap.error.Error*, ValueError

Raised when a context is (currently or permanently) unavailable for protecting or unprotecting a message

class aiocoap.oscore.**RequestIdentifiers**(*kid*, *partial_iv*, *nonce*, *can_reuse_nonce*)
Bases: object

A container for details that need to be passed along from the (un)protection of a request to the (un)protection of the response; these data ensure that the request-response binding process works by passing around the request's partial IV.

Users of this module should never create or interact with instances, but just pass them around.

get_reusable_nonce()
Return the nonce if can_reuse_nonce is True, and set can_reuse_nonce to False.

class aiocoap.oscore.**Algorithm**
Bases: object

encrypt(*plaintext*, *aad*, *key*, *iv*)
Return ciphertext + tag for given input data

decrypt(*ciphertext_and_tag*, *aad*, *key*, *iv*)
Reverse encryption. Must raise ProtectionInvalid on any error stemming from untrusted data.

class aiocoap.oscore.**AES_CCM**
Bases: *aiocoap.oscore.Algorithm*

AES-CCM implemented using the Python cryptography library

classmethod encrypt(*plaintext*, *aad*, *key*, *iv*)
Return ciphertext + tag for given input data

classmethod decrypt(*ciphertext_and_tag*, *aad*, *key*, *iv*)
Reverse encryption. Must raise ProtectionInvalid on any error stemming from untrusted data.

class aiocoap.oscore.**AES_CCM_16_64_128**
Bases: *aiocoap.oscore.AES_CCM*

value = 10

key_bytes = 16

tag_bytes = 8

iv_bytes = 13

class aiocoap.oscore.**AES_CCM_16_64_256**
Bases: *aiocoap.oscore.AES_CCM*

value = 11

key_bytes = 32

tag_bytes = 8

iv_bytes = 13

class aiocoap.oscore.**AES_CCM_64_64_128**
Bases: *aiocoap.oscore.AES_CCM*

value = 12

key_bytes = 16

tag_bytes = 8

iv_bytes = 7

class aiocoap.oscore.**AES_CCM_64_64_256**
Bases: *aiocoap.oscore.AES_CCM*

value = 13

key_bytes = 32

tag_bytes = 8

iv_bytes = 7

class aiocoap.oscore.**AES_CCM_16_128_128**
Bases: *aiocoap.oscore.AES_CCM*

value = 30

key_bytes = 16

tag_bytes = 16

iv_bytes = 13

class aiocoap.oscore.**AES_CCM_16_128_256**
Bases: *aiocoap.oscore.AES_CCM*

value = 31

key_bytes = 32

tag_bytes = 16

iv_bytes = 13

class aiocoap.oscore.**AES_CCM_64_128_128**
Bases: *aiocoap.oscore.AES_CCM*

value = 32

key_bytes = 16

tag_bytes = 16

iv_bytes = 7

class aiocoap.oscore.**AES_CCM_64_128_256**
Bases: *aiocoap.oscore.AES_CCM*

value = 33

key_bytes = 32

tag_bytes = 16

iv_bytes = 7

class aiocoap.oscore.**AES_GCM**
Bases: *aiocoap.oscore.Algorithm*

AES-GCM implemented using the Python cryptography library

iv_bytes = 12

classmethod encrypt(*plaintext*, *aad*, *key*, *iv*)
Return ciphertext + tag for given input data

classmethod decrypt(*ciphertext_and_tag*, *aad*, *key*, *iv*)
Reverse encryption. Must raise ProtectionInvalid on any error stemming from untrusted data.

class aiocoap.oscore.**A128GCM**
Bases: *aiocoap.oscore.AES_GCM*

value = 1

key_bytes = 16

tag_bytes = 16

class aiocoap.oscore.**A192GCM**
Bases: *aiocoap.oscore.AES_GCM*

value = 2

key_bytes = 24

tag_bytes = 16

class aiocoap.oscore.**A256GCM**
Bases: *aiocoap.oscore.AES_GCM*

value = 3

key_bytes = 32

tag_bytes = 16

class aiocoap.oscore.**ChaCha20Poly1305**
Bases: *aiocoap.oscore.Algorithm*

value = 24

key_bytes = 32

tag_bytes = 16

iv_bytes = 12

classmethod encrypt(*plaintext*, *aad*, *key*, *iv*)
Return ciphertext + tag for given input data

classmethod decrypt(*ciphertext_and_tag*, *aad*, *key*, *iv*)
Reverse encryption. Must raise ProtectionInvalid on any error stemming from untrusted data.

class aiocoap.oscore.**AlgorithmCountersign**
Bases: object

A fully parameterized COSE countersign algorithm

An instance is able to provide all the alg_countersign, par_countersign and par_countersign_key parameters taht go into the Group OSCORE algorithms field.

sign(*body*, *external_aad*, *private_key*)
Return the signature produced by the key when using CounterSignature0 as describe in draft-ietf-cose-countersign-01

verify(*signature*, *body*, *external_aad*, *public_key*)
Verify a signature in analogy to sign

generate()
Return a usable private key

public_from_private(*private_key*)
Given a private key, derive the publishable key

staticstatic(*private_key*, *public_key*)
Derive a shared static-static secret from a private and a public key

signature_length
The length of a signature using this algorithm

class aiocoap.oscore.**Ed25519**
Bases: *aiocoap.oscore.AlgorithmCountersign*

sign(*body*, *aad*, *private_key*)
Return the signature produced by the key when using CounterSignature0 as describe in draft-ietf-cose-countersign-01

verify(*signature*, *body*, *aad*, *public_key*)
Verify a signature in analogy to sign

generate()
Return a usable private key

public_from_private(*private_key*)
Given a private key, derive the publishable key

staticstatic(*private_key*, *public_key*)
Derive a shared static-static secret from a private and a public key

value_all_par = [-8, [[1], [1, 6]]]

signature_length = 64

class aiocoap.oscore.**ECDSA_SHA256_P256**
Bases: *aiocoap.oscore.AlgorithmCountersign*

from_public_parts(*x: bytes*, *y: bytes*)
Create a public key from its COSE values

from_private_parts(*x: bytes*, *y: bytes*, *d: bytes*)

sign(*body*, *aad*, *private_key*)
Return the signature produced by the key when using CounterSignature0 as describe in draft-ietf-cose-countersign-01

verify(*signature*, *body*, *aad*, *public_key*)
Verify a signature in analogy to sign

generate()
Return a usable private key

public_from_private(*private_key*)
Given a private key, derive the publishable key

staticstatic(*private_key*, *public_key*)
Derive a shared static-static secret from a private and a public key

value_all_par = [-7, [[2], [2, 1]]]

signature_length = 64

class aiocoap.oscore.**BaseSecurityContext**
Bases: object

external_aad_is_group = False

authenticated_claims = []

class aiocoap.oscore.**CanProtect**
Bases: *aiocoap.oscore.BaseSecurityContext*

is_signing = False

responses_send_kid = False

protect(*message*, *request_id=None*, ***, *kid_context=True*)
Given a plain CoAP message, create a protected message that contains message's options in the inner or outer CoAP message as described in OSCOAP.

If the message is a response to a previous message, the additional data from unprotecting the request are passed in as request_id. When request data is present, its partial IV is reused if possible. The security context's ID context is encoded in the resulting message unless kid_context is explicitly set to a False; other values for the kid_context can be passed in as byte string in the same parameter.

new_sequence_number()
Return a new sequence number; the implementation is responsible for never returning the same value twice in a given security context.

May raise ContextUnavailable.

post_seqnoincrease()
Ensure that sender_sequence_number is stored

context_from_response(*unprotected_bag*) → aiocoap.oscore.CanUnprotect
When receiving a response to a request protected with this security context, pick the security context with which to unprotect the response given the unprotected information from the Object-Security option.

This allow picking the right security context in a group response, and helps getting a new short-lived context for B.2 mode. The default behaivor is returning self.

class aiocoap.oscore.**CanUnprotect**
Bases: *aiocoap.oscore.BaseSecurityContext*

unprotect(*protected_message*, *request_id=None*)

context_for_response() → aiocoap.oscore.CanProtect
After processing a request with this context, with which security context should an outgoing response be protected? By default, it's the same context.

class aiocoap.oscore.**SecurityContextUtils**
Bases: *aiocoap.oscore.BaseSecurityContext*

derive_keys(*master_salt*, *master_secret*)
Populate sender_key, recipient_key and common_iv from the algorithm, hash function and id_context already configured beforehand, and from the passed salt and secret.

get_oscore_context_for(*unprotected*)
Return a sutiable context (most easily self) for an incoming request if its unprotected data (COSE_KID, COSE_KID_CONTEXT) fit its description. If it doesn't match, it returns None.

The default implementation just strictly checks for whether kid and any kid context match (not matching if a local KID context is set but none is given in the request); modes like Group OSCORE can spin up aspect objects here.

class aiocoap.oscore.**ReplayWindow**(*size*, *strike_out_callback*)
Bases: object

A regular replay window of a fixed size.

It is implemented as an index and a bitfield (represented by an integer) whose least significant bit represents the seqyence number of the index, and a 1 indicates that a number was seen. No shenanigans around implicit leading ones (think floating point normalization) happen.

```
>>> w = ReplayWindow(32, lambda: None)
>>> w.initialize_empty()
```

(continues on next page)

(continued from previous page)

```
>>> w.strike_out(5)
>>> w.is_valid(3)
True
>>> w.is_valid(5)
False
>>> w.strike_out(0)
>>> w.strike_out(1)
>>> w.strike_out(2)
>>> w.is_valid(1)
False
```

Jumping ahead by the window size invalidates older numbers:

```
>>> w.is_valid(4)
True
>>> w.strike_out(35)
>>> w.is_valid(4)
True
>>> w.strike_out(36)
>>> w.is_valid(4)
False
```

For every key, the replay window can only be initielized empty once. On later uses, it needs to be persisted by storing the output of self.persist() somewhere and loaded from that persisted data.

It is acceptable to store persistance data in the strike_out_callback, but that must then ensure that the data is written (flushed to a file or committed to a database), but that is usually inefficient.

This class is not considered for stabilization yet and an implementation detail of the SecurityContext implementation(s).

is_initialized()

initialize_empty()

initialize_from_persisted(*persisted*)

initialize_from_freshlyseen(*seen*)
Initialize the replay window with a particular value that is just being observed in a fresh (ie. generated by the peer later than any messages processed before state was lost here) message. This marks the seen sequence number and all preceding it as invalid, and and all later ones as valid.

is_valid(*number*)

strike_out(*number*)

persist()
Return a dict containing internal state which can be passed to init to recreated the replay window.

class aiocoap.oscore.**FilesystemSecurityContext**(*basedir*, *sequence_number_chunksize_start=10*, *sequence_number_chunksize_limit=10000*)

Bases: *aiocoap.oscore.CanProtect*, *aiocoap.oscore.CanUnprotect*, *aiocoap.oscore.SecurityContextUtils*

Security context stored in a directory as distinct files containing containing

- Master secret, master salt, sender and recipient ID, optionally algorithm, the KDF hash function, and replay window size (settings.json and secrets.json, where the latter is typically readable only for the user)
- sequence numbers and replay windows (sequence.json, the only file the process needs write access to)

The static parameters can all either be placed in settings.json or secrets.json, but must not be present in both; the presence of either file is sufficient.

> **Warning:** Security contexts must never be copied around and used after another copy was used. They should only ever be moved, and if they are copied (eg. as a part of a system backup), restored contexts must not be used again; they need to be replaced with freshly created ones.

An additional file named *lock* is created to prevent the accidental use of a context by to concurrent programs.

Note that the sequence number file is updated in an atomic fashion which requires file creation privileges in the directory. If privilege separation between settings/key changes and sequence number changes is desired, one way to achieve that on Linux is giving the aiocoap process's user group write permissions on the directory and setting the sticky bit on the directory, thus forbidding the user to remove the settings/secret files not owned by him.

Writes due to sent sequence numbers are reduced by applying a variation on the mechanism of RFC8613 Appendix B.1.1 (incrementing the persisted sender seqence number in steps of k). That value is automatically grown from sequence_number_chunksize_start up to sequence_number_chunksize_limit. At runtime, the receive window is not stored but kept indeterminate. In case of an abnormal shutdown, the server uses the mechanism described in Appendix B.1.2 to recover.

exception LoadError
Bases: ValueError

Exception raised with a descriptive message when trying to load a faulty security context

post_seqnoincrease()
Ensure that sender_sequence_number is stored

class aiocoap.oscore.**GroupContext**
Bases: object

is_signing = True

external_aad_is_group = True

responses_send_kid = True

private_key
Private key used to sign outgoing messages.

Contexts not designed to send messages may raise a RuntimeError here; that necessity may later go away if some more accurate class modelling is found.

recipient_public_key
Public key used to verify incoming messages.

Contexts not designed to receive messages (because they'd have aspects for that) may raise a RuntimeError here; that necessity may later go away if some more accurate class modelling is found.

class aiocoap.oscore.**SimpleGroupContext**(*algorithm*, *hashfun*, *alg_countersign*, *group_id*, *master_secret*, *master_salt*, *sender_id*, *private_key*, *peers*)
Bases: *aiocoap.oscore.GroupContext*, *aiocoap.oscore.CanProtect*, *aiocoap.oscore.CanUnprotect*, *aiocoap.oscore.SecurityContextUtils*

A context for an OSCORE group

This is a non-persistable version of a group context that does not support any group manager or rekeying; it is set up statically at startup.

It is intended for experimentation and demos, but aims to be correct enough to be usable securely.

private_key = None

recipient_public_key
Public key used to verify incoming messages.

Contexts not designed to receive messages (because they'd have aspects for that) may raise a RuntimeError here; that necessity may later go away if some more accurate class modelling is found.

derive_keys(*master_salt*, *master_secret*)
Populate sender_key, recipient_key and common_iv from the algorithm, hash function and id_context already configured beforehand, and from the passed salt and secret.

post_seqnoincrease()
No-op because it's ephemeral

context_from_response(*unprotected_bag*) → aiocoap.oscore.CanUnprotect
When receiving a response to a request protected with this security context, pick the security context with which to unprotect the response given the unprotected information from the Object-Security option.

This allow picking the right security context in a group response, and helps getting a new short-lived context for B.2 mode. The default behaivor is returning self.

get_oscore_context_for(*unprotected*)
Return a sutiable context (most easily self) for an incoming request if its unprotected data (COSE_KID, COSE_KID_CONTEXT) fit its description. If it doesn't match, it returns None.

The default implementation just strictly checks for whether kid and any kid context match (not matching if a local KID context is set but none is given in the request); modes like Group OSCORE can spin up aspect objects here.

pairwise_for(*recipient_id*)

for_sending_deterministic_requests(*deterministic_id, target_server: Optional[bytes]*)

aiocoap.oscore.**decode_dss_signature**()

aiocoap.oscore.**encode_dss_signature**()

aiocoap.oscore.**verify_start**(*message*)
Extract the unprotected COSE options from a message for the verifier to then pick a security context to actually verify the message. (Future versions may also report fields from both unprotected and protected, if the protected bag is ever used with OSCORE.).

Call this only requests; for responses, you'll have to know the security context anyway, and there is usually no information to be gained.

6.5 CoAP API design notes

This documentation chapter is not a full-fledged guide yet; rather, it highlights some points of how CoAP is expressed in aiocoap.

- Library as a proxy:

  A CoAP library and API can, to some extent, be viewed as a CoAP proxy and a CoAP transport protocol, respectively.

  This shapes what the library can do, and makes guidance on how to do it accessible – for CoAP specification describe what proxies may do but say nothing on APIs.

For example, splitting up large messages into block-wise chunks is something aiocoap does unless asked specificially not to; in its operation, it follows the guidance set out for proxies in RFC7959. Likewise, this is what justifies that aiocoap intermittently drops observe notifications. (Future releases might even take on intermediate proxies due to discovered alternative protocols).

On the flip side, going all the way with this would mean that the application gets no choice in properties lost across proxies: The application could not decide whether reliable transport should be used. Furthermore, applied in full, the application could not use any proxy-unsafe options not suported by the library.

In aiocoap, a balance is attempted. It behaves like a proxy for some convenience operations, which can be disabled as needed. It still allows the application author to set message properties for the first hop, and does not reject messages with proxy-unsafe options (trusting that no new proxy unsafe options are unsafe for the limited thing the library does).

- Messages as exchange objects:

  In aiocoap, requests and responses on the server and client side are handed to the application as CoAP messages.

  This gives the application a lot of flexibility in terms of setting and reacting to options; it allows application authors to explore extensions to CoAP. It is also the style of API used by libcoap and gcoap / nanocoap. On the other hand, it makes it relatively verbose to write applications that exclusively operate on predefined patterns (like objects that can be rendered into a representation depending on content format negotiation, or getter-setter patterns using GET and PUT). Simplified handlers for such cases can be built on aiocoap; the `contrib` directory contains some exploratory examples.

  In combination with the abovementioned proxy paradigm, this can lead to some weirdness when messages are represented differently on different transports. The general approach currenlty taken is to build the application level messages like a CoAP-over-UDP message was treated if UDP messages could be arbitrarily long (or possibly, with future changes to the internal block-wise mechanisms, using BERT). Notably, this means that applications that set Observe numbers manually should pack them into a 4-byte integer (which the TCP transport would then elide); transports may, however, do any deduplication and then just forward to the application that there *is* still an Observe number set. This is all not set in stone, though, and open for further development.

  Handling of properties outside of code, options and payload is currently still a bit mixed: Most resides in custom attributes of the message (like `aiocoap.Message.remote` or `aiocoap.Message.mtype`); thes are generally treated as hints and not always fully applicable. Some properties are also transported in options even though they are not exactly fitting here; for example, the No-Response option is used in responses to indicate to the stack that no response should be set. The latter should be cleaned up.

6.6 Usage Examples

These files can serve as reference implementations for a simplistic server and client. In order to test them, run `./server.py` in one terminal, and use `./clientGET.py` and `./clientPUT.py` to interact with it.

The programs' source code should give you a good starting point to get familiar with the library if you prefer reading code to reading tutorials. Otherwise, you might want to have a look at the *Guided Tour through aiocoap*, where the relevant concepts are introduced and explained step by step.

Note: These example programs are not shipped in library version of aiocoap. They are present if you followed the *Development version* section of the installation instructions; otherwise, you can download them from the project website.

6.6.1 Client

```
import logging
import asyncio

from aiocoap import *

logging.basicConfig(level=logging.INFO)

async def main():
    protocol = await Context.create_client_context()

    request = Message(code=GET, uri='coap://localhost/time')

    try:
        response = await protocol.request(request).response
    except Exception as e:
        print('Failed to fetch resource:')
        print(e)
    else:
        print('Result: %s\n%r'%(response.code, response.payload))

if __name__ == "__main__":
    asyncio.run(main())
```

```
import logging
import asyncio

from aiocoap import *

logging.basicConfig(level=logging.INFO)

async def main():
    """Perform a single PUT request to localhost on the default port, URI
    "/other/block". The request is sent 2 seconds after initialization.

    The payload is bigger than 1kB, and thus sent as several blocks."""

    context = await Context.create_client_context()

    await asyncio.sleep(2)

    payload = b"The quick brown fox jumps over the lazy dog.\n" * 30
    request = Message(code=PUT, payload=payload, uri="coap://localhost/other/block")

    response = await context.request(request).response

    print('Result: %s\n%r'%(response.code, response.payload))

if __name__ == "__main__":
    asyncio.run(main())
```

6.6.2 Server

```
import datetime
import logging

import asyncio

import aiocoap.resource as resource
import aiocoap


class BlockResource(resource.Resource):
    """Example resource which supports the GET and PUT methods. It sends large
    responses, which trigger blockwise transfer."""

    def __init__(self):
        super().__init__()
        self.set_content(b"This is the resource's default content. It is padded "
                b"with numbers to be large enough to trigger blockwise "
                b"transfer.\n")

    def set_content(self, content):
        self.content = content
        while len(self.content) <= 1024:
            self.content = self.content + b"0123456789\n"

    async def render_get(self, request):
        return aiocoap.Message(payload=self.content)

    async def render_put(self, request):
        print('PUT payload: %s' % request.payload)
        self.set_content(request.payload)
        return aiocoap.Message(code=aiocoap.CHANGED, payload=self.content)


class SeparateLargeResource(resource.Resource):
    """Example resource which supports the GET method. It uses asyncio.sleep to
    simulate a long-running operation, and thus forces the protocol to send
    empty ACK first. """

    def get_link_description(self):
        # Publish additional data in .well-known/core
        return dict(**super().get_link_description(), title="A large resource")

    async def render_get(self, request):
        await asyncio.sleep(3)

        payload = "Three rings for the elven kings under the sky, seven rings "\
                "for dwarven lords in their halls of stone, nine rings for "\
                "mortal men doomed to die, one ring for the dark lord on his "\
                "dark throne.".encode('ascii')
        return aiocoap.Message(payload=payload)

class TimeResource(resource.ObservableResource):
    """Example resource that can be observed. The `notify` method keeps
    scheduling itself, and calles `update_state` to trigger sending
    notifications."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()

        self.handle = None

    def notify(self):
        self.updated_state()
        self.reschedule()

    def reschedule(self):
        self.handle = asyncio.get_event_loop().call_later(5, self.notify)

    def update_observation_count(self, count):
        if count and self.handle is None:
            print("Starting the clock")
            self.reschedule()
        if count == 0 and self.handle:
            print("Stopping the clock")
            self.handle.cancel()
            self.handle = None

    async def render_get(self, request):
        payload = datetime.datetime.now().\
                strftime("%Y-%m-%d %H:%M").encode('ascii')
        return aiocoap.Message(payload=payload)

class WhoAmI(resource.Resource):
    async def render_get(self, request):
        text = ["Used protocol: %s." % request.remote.scheme]

        text.append("Request came from %s." % request.remote.hostinfo)
        text.append("The server address used %s." % request.remote.hostinfo_local)

        claims = list(request.remote.authenticated_claims)
        if claims:
            text.append("Authenticated claims of the client: %s." % ", ".join(repr(c)␣
↪for c in claims))
        else:
            text.append("No claims authenticated.")

        return aiocoap.Message(content_format=0,
                payload="\n".join(text).encode('utf8'))

# logging setup

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

async def main():
    # Resource tree creation
    root = resource.Site()

    root.add_resource(['.well-known', 'core'],
            resource.WKCResource(root.get_resources_as_linkheader))
    root.add_resource(['time'], TimeResource())
    root.add_resource(['other', 'block'], BlockResource())
```

(continues on next page)

(continued from previous page)

```
    root.add_resource(['other', 'separate'], SeparateLargeResource())
    root.add_resource(['whoami'], WhoAmI())

    await aiocoap.Context.create_server_context(root)

    # Run forever
    await asyncio.get_running_loop().create_future()

if __name__ == "__main__":
    asyncio.run(main())
```

6.7 CoAP tools

As opposed to the *Usage Examples*, programs listed here are not tuned to show the use of aiocoap, but are tools for everyday work with CoAP implemented in aiocoap. Still, they can serve as examples of how to deal with user-provided addresses (as opposed to the fixed addresses in the examples), or of integration in a bigger project in general.

6.7.1 aiocoap-client

aiocoap-client is a simple command-line tool for interacting with CoAP servers

```
usage: aiocoap-client [-h] [--non] [-m METHOD] [--observe]
                      [--observe-exec CMD] [--accept MIME] [--proxy URI]
                      [--payload X] [--payload-initial-szx SZX]
                      [--content-format MIME] [--no-set-hostname] [-v] [-q]
                      [--interactive] [--credentials CREDENTIALS] [--version]
                      [--color] [--pretty-print]
                      url
```

Positional Arguments

url CoAP address to fetch

Named Arguments

--non Send request as non-confirmable (NON) message

Default: False

-m, --method Name or number of request method to use (default: “GET”)

Default: “GET”

--observe Register an observation on the resource

Default: False

--observe-exec Run the specified program whenever the observed resource changes, feeding the response data to its stdin

--accept Content format to request

--proxy Relay the CoAP request to a proxy for execution

--payload Send X as request payload (eg. with a PUT). If X starts with an '@', its remainder is treated as a file name and read from; '@-' reads from the console. Non-file data may be recoded, see –content-format.

--payload-initial-szx Size exponent to limit the initial block's size (0 16 Byte, 6 1024 Byte)

--content-format Content format of the –payload data. If a known format is given and –payload has a non-file argument, conversion is attempted (currently only JSON/Python-literals to CBOR).

--no-set-hostname Suppress transmission of Uri-Host even if the host name is not an IP literal

Default: True

-v, --verbose Increase the debug output

-q, --quiet Decrease the debug output

--interactive Enter interactive mode

Default: False

--credentials Load credentials to use from a given file

--version show program's version number and exit

--color, --no-color Color output (default on TTYs if all required modules are installed)

--pretty-print, --no-pretty-print Pretty-print known content formats (default on TTYs if all required modules are installed)

6.7.2 aiocoap-proxy

a plain CoAP proxy that can work both as forward and as reverse proxy

```
usage: aiocoap-proxy [-h] [--forward] [--reverse] [--bind BIND]
                     [--credentials CREDENTIALS]
                     [--tls-server-certificate CRT] [--tls-server-key KEY]
                     [--register [RD-URI]] [--register-as EP[.D]]
                     [--register-proxy] [--namebased NAME:DEST]
                     [--subdomainbased NAME:DEST] [--pathbased PATH:DEST]
                     [--unconditional DEST]
```

mode

Required argument for setting the operation mode

--forward Run as forward proxy

Default: False

--reverse Run as reverse proxy

Default: False

details

Options that govern how requests go in and out

--bind Host and/or port to bind to (see –help-bind for details)

--credentials	JSON file pointing to credentials for the server's identity/ies.
--tls-server-certificate	TLS certificate (chain) to present to connecting clients (in PEM format)
--tls-server-key	TLS key to load that supports the server certificate
--register	Register with a Resource directory Default: False
--register-as	Endpoint name (with possibly a domain after a dot) to register as
--register-proxy	Ask the RD to serve as a reverse proxy. Note that this is only practical for –unconditional or –pathbased reverse proxies. Default: False

Rules

Sequence of forwarding rules that, if matched by a request, specify a forwarding destination. Destinations can be prefixed to change their behavior: With an '@' sign, they are treated as forward proxies. With a '!' sign, the destination is set as Uri-Host.

--namebased	If Uri-Host matches NAME, route to DEST
--subdomainbased	If Uri-Host is anything.NAME, route to DEST
--pathbased	If a requested path starts with PATH, split that part off and route to DEST
--unconditional	Route all requests not previously matched to DEST

6.7.3 aiocoap-rd

A plain CoAP resource directory according to draft-ietf-core-resource-directory-25

Known Caveats:

- It is very permissive. Not only is no security implemented.
- This may and will make exotic choices about discoverable paths whereever it can (see StandaloneResourceDirectory documentation)
- Split-horizon is not implemented correctly
- Unless enforced by security (ie. not so far), endpoint and sector names (ep, d) are not checked for their lengths or other validity.
- Simple registrations don't cache .well-known/core contents

```
usage: aiocoap-rd [-h] [--bind BIND] [--credentials CREDENTIALS]
                  [--tls-server-certificate CRT] [--tls-server-key KEY]
```

Named Arguments

--bind	Host and/or port to bind to (see –help-bind for details)
--credentials	JSON file pointing to credentials for the server's identity/ies.
--tls-server-certificate	TLS certificate (chain) to present to connecting clients (in PEM format)
--tls-server-key	TLS key to load that supports the server certificate

6.7.4 aiocoap-fileserver

A simple file server that serves the contents of a given directory in a read-only fashion via CoAP. It provides directory listings, and guesses the media type of files it serves.

It follows the conventions set out for the [kitchen-sink fileserver], optionally with write support, with some caveats:

- There are some time-of-check / time-of-use race conditions around the handling of ETags, which could probably only be resolved if heavy file system locking were used. Some of these races are a consequence of this server implementing atomic writes through renames.

  As long as no other processes access the working area, and aiocoap is run single threaded, the races should not be visible to CoAP users.

- ETags are constructed based on information in the file's (or directory's) *stat* output – this avoids reaing the whole file on overwrites etc.

  This means that forcing the MTime to stay constant across a change would confuse clients.

- While GET requests on files are served block by block (reading only what is being requested), PUT operations are spooled in memory rather than on the file system.

- Directory creation and deletion is not supported at the moment.

[kitchen-sink fileserver]: https://www.ietf.org/archive/id/draft-amsuess-core-coap-kitchensink-00.html#name-coap

```
usage: aiocoap-fileserver [-h] [-v] [--register [RD-URI]] [--write]
                          [--bind BIND] [--credentials CREDENTIALS]
                          [--tls-server-certificate CRT]
                          [--tls-server-key KEY]
                          [path]
```

Positional Arguments

path Root directory of the server

Default: .

Named Arguments

-v, --verbose Be more verbose (repeat to debug)

Default: 0

--register Register with a Resource directory

Default: False

--write Allow writes by any user

Default: False

--bind Host and/or port to bind to (see –help-bind for details)

--credentials JSON file pointing to credentials for the server's identity/ies.

--tls-server-certificate TLS certificate (chain) to present to connecting clients (in PEM format)

--tls-server-key TLS key to load that supports the server certificate

Those utilities are installed by *setup.py* at the usual executable locations; during development or when working from a git checkout of the project, wrapper scripts are available in the root directory. In some instances, it might be practical to access their functionality from within Python; see the *aiocoap.cli* module documentation for details.

All tools provide details on their invocation and arguments when called with the `--help` option.

6.7.5 contrib

Tools in the `contrib/` folder are somewhere inbetween *Usage Examples* and the tools above; the rough idea is that they should be generally useful but not necessarily production tools, and simple enough to be useful as an inspiration for writing other tools; none of this is set in stone, though, so that area can serve as a noncommittal playground.

These tools are currently present:

- `aiocoap-widgets`: Graphical software implementations of example CoAP devices as servers (eg. light bulb, switch). They should become an example of how CoRE interfaces and dynlinks can be used to discover and connect servers, and additionally serve as a playground for a more suitable Resource implementation.

  The GUI is implemented in Gtk3 using the gbulb asyncio loop.

- `aiocoap-kivy-widget`: A similar (and smaller) widget implemented in Kivy.

  As asyncio support is not merged in Kivy yet, be sure to build the library from the asyncio pull request.

- `oscore-plugtest`: Server and client for the interoperability tests conducted during the development of OSCORE.

  The programs in there are also used as part of the test suite.

- `rd-relay`: An experiment of how much a host must implement if it is to be discovered during a Resource Directory discovery process, but does not serve as the full resource directory itself and redirects the client there.

6.8 Frequently Answered Questions

(Not *actually* asked frequently – actually, this is a bunch of notes that users of the library should probably see at some point, while it is not clear where to better put them).

- **Which platforms are supported?**

  aiocoap requires Python 3.7 (or PyPy 3.7), and should run on all operating systems supported by Python.

  Development and automated tests run on Linux, and this is where all listed features are supported.

  aiocoap generally runs on FreeBSD, Windows and macOS as well. Tests on FreeBSD are conducted manually; for Windows and macOS it's all largely relying on user feedback tracked in the bug tracker for portability issues.

  Note that the main CoAP-over-UDP transport *udp6* is only on-by-default on Linux because other platforms have no way of receiving network errors from an unconnected socket. The simpler UDP transports used on the other platforms do not support all features, and in particular lack multicast support.

  aiocoap is agnostic of the backing asyncio implementation as long as it implements the functionality required by the transport (`add_reader` for udp6, `sockname` extra for role reversal on simple6). It is known to work with uvloop and gbulb.

- **How can a server be scaled up to use multiple cores?**

  Python is often considered weak around threading. While setups with multiple asyncio worker should conceptually work, the easiest way to parallelize is just to have multiple instances of your server running at the same time. This works when transports and platform support the SO_REUSEPORT option (this is the case on Linux

with the default transports, but never on Windows), with which incoming requests are dispatched to any of the processes serving the port by the operating system.

This requires an application design that has all its persistence managed outside the server process; that is typically the case with file system or database backed servers.

(aiocoap internally does hold some state, but that is always per client, and the load balancing typically ensures that requests from the same client wind up in the same process.)

- **Why do I get a "The transport can not be bound to any-address." error message?**

  For your platform, the `simplesocketserver` module was selected. See *the simplesocketserver documentation* for why it can not bind to that address.

- **How is multicast supported?**

  Support for multicast is currently limited.

  On the server side, things are mostly ready. Groups are joined *at server creation*.

  On the client side, requests to multicast addresses can be sent, and while they are treated adaequately on the protocol level (eg. will not send CON requests), the *request interface* only exposes the first response. Thus, it can be used in discovery situations as long as only one response is processed, but not yet to its full power of obtaining data from multiple devices.

  Note that multicast requests often require specification of an interface, as otherwise the request is underspecified. Thus, a typical command line request might look like this:

```
./aiocoap-client coap://'[ff02::fd%eth0]'/.well-known/core --non
```

- **aiocoap fails to start if IPv6 is disabled system-wide.**

  Yes. Don't do that It is not a supported mode of operation with the default implementation.

  Background details:

  The default transport of aiocoap uses APIs that are well specified for IPv6 and work there for both IPv4 and IPv6 packets. Explicitly re-implementing everything on v4 would not only be needless extra work, it would also be a portability problem as unlike for IPv6, the interfaces are not specified platform independenlty for IPv4. Moreover, that mode would be error prone because it wouldn't receive regular testing.

6.9 Change log

This summarizes the changes between released versions. For a complete change log, see the git history. For details on the changes, see the respective git commits indicated at the start of the entry.

6.9.1 Version 0.4.5

Behavioral changes

- RSTs are not sent on unrecognized responses any more unless the received message was a CON; the previous behavior was violating the specification.

Deprecations

- UNSUPPORTED_MEDIA_TYPE is now formally deprecated, use UNSUPPORTED_CONTENT_FORMAT instead.

Minor enhancements

- Fix tests for Python 3.11.
- Lower log level of "but could not match it to a running exchange" from warning to info.
- Shorten the string representation of message types (to "CON", "ACK" etc.)

6.9.2 Version 0.4.4

New features

- Content-Format / Accept option now use a dedicated ContentFormat type.

  Applications should be unaffected as the type is still derived from int.
- Non-traditional responses are now experimentally supported by implementing `.render_to_pipe()` on a resource.

Deprecations

- Building custom resources by inheriting from `interfaces.Resource` / `interfaces.ObservableResource` and implementing `.render()` etc. is deprecated. Instead, inherit from `resource.Resource` (recommended), or implement `.render_to_pipe()` (eg. when implementing a proxy).
- numbers.media_type and media_type_rev: Use the ContentFormat type's constructor and accessors instead.

Tools

- aiocoap-fileserver now has optiojnal write support, and ETag and If-* option handling.
- aiocoap-client now assembles and displays the Location-* options of responses.
- aiocoap-rd now has dedicated logging independent of aiocoap's.
- Various small fixes to aiocoap-rd.
- Help and error texts were improved.

Minor enhancements

- Documentation now uses `await` idiom, as it is available even inside the asyncio REPL.
- The default cut-off for block-wise fragmentation was increased from 1024 to 1124 bytes. This allows OSCORE to use the full inner block-wise size without inadvertently causing outer fragmentation, while still fitting within the IPv6 minimum MTU.
- Connection shutdown for TCP and WebSockets has been implemented, they now send Release messages and wait for the peer to close the connection.
- Type annotations are now used more widely.
- Library shutdown works more cleanly by not relying on the presence of the async loop.
- OSCORE contexts now only access the disk when necessary.
- OSCORE now supports inner block-wise transfer and observations.

- WebSocket servers can now pick an ephemeral port (when binding to port 0).
- Tasks created by the library are now named for easier debugging.
- Bugs fixed around handling of IP literals in proxies.

Internal refactoring

- Pipes (channels for asynchronously producing resposnes, previously called PlumbingResponse) are now used also for resource rendering. Block-wise and observation handling could thus be moved away from the core protocol and into the resource implementations.
- Exception chaining was started to be reworked into explicit re-raises.

6.9.3 Version 0.4.3

Compatibility

- Fix compatibility with websockets 10.1.

Minor enhancements

- Failure path fixes.

6.9.4 Version 0.4.2

New features

- Experimental support for DTLS server operation (PSK only).

Tools

- aiocoap-client reports responder address if different from requested.
- aiocoap-rd is aligned with draft version -27 (e.g. using .well-known/rd).
- aiocoap-proxy can be registered to an RD.

Compatibility

- Group OSCORE updated to -11.
- Fixes to support Python 3.10, including removal of some deprecated idioms and inconsistent loop handling.

Examples / contrib

- Demo for Deterministic OSCORE added.

Deprecations

- util.quote_nonascii
- error.{RequestTimedOut,WaitingForClientTimedOut}
- Direct use of AsyncCLIDaemon from asynchronous contexts (replacement not available yet).

Minor enhancements

- Resources can hide themselves from the listing in /.well-known/core.
- RD's built-in proxy handles block-wise better.
- Added __repr__ to TokenManager and MessageManager.
- Pretty printer errs gracefully.
- Failure path fixes.
- Documentation updates.
- Removed distutils dependency.

Internal refactoring

- CI testing now uses pytest.
- dispatch_error now passes on exceptions.
- DTLS client cleaned up.
- Build process now uses the build module.

6.9.5 Version 0.4.1

- Fix Python version reference to clearly indicate the 3.7 requirement everywhere.

  A Python requirement of ">= 3.6.9" was left over in the previous release's metadata from earlier intermediate steps that accomodated PyPy's pre-3.7 version.

6.9.6 Version 0.4

Multicast improvements

- Multicast groups are not joined by default any more. Instead, groups and interfaces on which to join need to be specified explicitly. The previous mechanism was unreliable, and only joined on one (more or less random) interface.
- Network interfaces can now be specified in remotes of larger than link-local scope.
- In udp6, network interface are selected via PKTINFO now. They used to be selected using the socket address tuple, but that was limited to link-local addresses, but PKTINFO worked just as well for link-local addresses.
- Remote addresses in udp6 now have a `netif` property.

New features

- The simple6 transport can now indicate the local address when supported by the platoforrm. This makes it a viable candidate for LwM2M clients as they often operate using role reversal.
- Servers (including the shipped examples) can now offer OSCORE through the OSCORE sitewrapper.

  Access control is only rudimentary in that the authorization information is not available in a convenient form yet.
- CoAP over WebSockets is now supported (in client and server role, with and without TLS). Please note that the default port bound to is not the HTTP default port but 8683.
- OSCORE group communication is now minimally supported (based on draft version 10). No automated ways of setting up a context are provided yet.

  This includes highly experimental support for deterministic requests.
- DTLS: Terminating connections are now handled correctly, and shut down when unused.

  The associated refactoring also reduces the resource usage of DTLS connections.

Tools updates

- aiocoap-client: New options to
  - set initial Block1 size (`--payload-initial-szx`), and to
  - elide the Uri-Host option from requests to named hosts.
- aiocoap-client: CBOR input now accepts Python literals or JSON automatically, and can thus produce numeric keys and byte strings.
- aiocoap-client: Preprocessed CBOR output now works for any CBOR-based content format.
- resource-directory: Updated to draft -25.
- resource-directory: Compatibility mode for LwM2M added.
- resource-directory: Proxying extension implemented. With this, and RD can be configured to allow access to endpoints behind a firewalls or NAT.
- Example server: Add /whoami resource.

Dependencies

- The minimum required Python version is now 3.7.
- The cbor library dependency was replaced with the cbor2 library.
- The dependency on the hkdf library was removed.
- The ge25519 library dependency was added to perform key conversion steps necessary for Group OSCORE.

Portability

- Several small adjustments were made to accomodate execution on Windows.
- FreeBSD was added to the list of supported systems (without any need for changes).

Fixes possibly breaking applications

- Some cases of OSError were previously raised in responses. Those are now all expressed as an aiocoap.error.NetworkError, so that an application only need to catch aiocoap.error.Error for anything that's expected to go wrong.

  The original error cause is available in a chained exception.

- Responses are not deduplicated any more; as a result, less state is kept in the library.

  As a result, separate responses whose ACKs get lost produce an RST the second time the CON comes. This changes nothing about the client-side handling (which is complete either way with the first response), but may upset servers that do not anticipate this allowed behavior.

Minor fixes

- The repr of udp6 addresses now shows all address components.
- Debug information output was increased in several spots.
- The `loop=` parameter was removed where it is deprecated by Python 3.8.
- asyncio Futures are created using create_future in some places.
- Binding to port 0 works again.
- The file server's registration at an RD was fixed.
- File server directories can now use block-wise transfer.
- Server errors from rendering exceptions to messages are now caught.
- Notifications now respect the block size limit.
- Several improvements to the test infrastructure.
- Refactoring around request processing internals (PlumbingRequest) alleviated potential memory leaks.
- Update option numbers from draft-ietf-echo-request-tag-10.
- Various proxying fixes and enhancements.
- TLS: Use SNI (Python >= 3.8), set correct hostinfo based on it.
- Internally used NoResponse options on responses are not leaked any more.
- Timeouts from one remote are now correctly propagated to all pending requests.
- Various logging improvements and changes.
- udp6: Show warnings when operating system fails to deliver pktinfo (happens with very old Linux kernels).
- Reduce installation clobber by excluding tests.
- Enhanced error reporting for erroneous `coap://2001:db8::1/` style URIs
- Improve OSCORE's shutdown robustness.
- Sending to IPv4 literals now does not send the Uri-Host automatically any more.

6.9.7 Version 0.4b3

Behavioral changes

- Responses to NON requests are now sent as NON.

Portability

- All uses of SO_REUSEPORT were changed to SO_REUSEADDR, as REUSEPORT is considered dangerous by some and removed from newer Python versions.

  On platoforms without support for that option, it is not set. Automatic load-balancing by running parallel servers is not supported there.

- The udp6 module is now usable on platforms without MSG_ERRQUEUE (ie. anything but Linux). This comes with caveats, so it is still only enabled by default on Linux.

  The required constants are now shipped with aiocoap for macOS for the benefit of Python versions less than 3.9.

Minor fixes

- More effort is made to sync OSCORE persistence files to disk.
- Memory leakage fixes on server and client side.
- Option numbers for Echo and Request-Tag were updated according to the latest draft version.

Other

- FAQ section started in the documentation.
- With `./setup.py test` being phased out, tests are now run via tox.

6.9.8 Version 0.4b2

New features

- OSCORE: Implement Appendix B.1 recovery. This allows the aiocoap program to run OSCORE without writing sequence numbers and replay windows to disk all the time. Instead, they write pessimistic values to disk that are rarely updated, write the last values on shutdown. In the event of an unclean shutdown, the sender sequence number is advanced by some, and the first request from a client is sent back for another roundtrip using the Echo option.

  An aiocoap client now also contains the code required to transparently resubmit requests if a server is in such a recovery situation.

- OSCORE: Security contexts are now protected against simultaneous use by multiple aiocoap processes. This incurs an additional dependency on the `filelock` package.

Breaking changes

- OSCORE: The file format of security context descriptions is changed. Instead of the previous roles concept, they now carry explicit sender and recipient IDs, and consequently do not take a role parameter in the credentials file any more.

  The sequence number format has changed incompatibly.

  No automatic conversion is available. It is recommended to replace old security contexts with new keys.

Minor fixes

- b4540f9: Fix workaround for missing definitions, restoring Python 3.5 support on non-amd64 platforms.
- b4b886d: Fix regression in the display of zone identifiers in IPv6 addresses.
- 5055bd5: The server now does not send RSTs in response to multicasts any more.
- OSCORE: The replay window used is now the prescribed 32bit large DTLS-like window.

6.9.9 Version 0.4b1

Tools

- aiocoap-client can now re-format binary output (hex-dumping binary files, showing CBOR files in JSON-like notation) and apply syntax highlighting. By default, this is enabled if the output is a terminal. If output redirection is used, data is passed on as-is.
- aiocoap-fileserver is now provided as a standalone tool. It provides directory listings in link format, guesses the content format of provided files, and allows observation.
- aiocoap-rd is now provided as a standalone tool and offers a simple CoRE Resource Directory server.

Breaking changes

- Client observations that have been requested by sending the Observe option must now be taken up by the client. The warning that was previously shown when an observation was shut down due to garbage collection can not be produced easily in this version, and will result in a useless persisting observation in the background. (See <https://github.com/chrysn/aiocoap/issues/104>)
- Server resources that expect the library to do handle blockwise by returning true to `needs_blockwise_assembly` do not allow random initial access any more; this this is especially problematic with clients that use a different source port for every package.

  The old behavior was prone to triggering an action twice on non-safe methods, and generating wrong results in block1+block2 scenarios when a later `FETCH block2:2/x/x` request would be treated as a new operation and return the result of an empty request body rather than being aligned with an earlier `FETCH block1:x/x/x` operation.
- fdc8b024: Support for Python 3.4 is dropped; minimum supported version is now 3.5.2.
- 0124ad0e: The network dumping feature was removed, as it would have been overly onerous to support it with the new more flexible transports.
- 092cf49f, 89c2a2e0: The content type mapped to the content format 0 was changed from “text/plain” (which was incorrect as it was just the bare media type) to the actual content of the IANA registry, ‘text/plain;charset=”utf8”’. For looking up the content format, text/plain is is still supported but deprecated.

- 17d1de5a: Handling of the various components of a remote was unified into the .remote property of messages. If you were previously setting unresolved addresses or even a tuple-based remote manualy, please set them using the `uri` pseudo-option now.
- 47863a29: Re-raise transport specific errors as aiocoap errors as aiocoap.error.ResolutionError or NetworkError. This allows API users to catch them independently of the underlying transport.
- f9824eb2: Plain strings as paths in add_resource are rejected. Applications that did this are very unlikely to have produced the intended behavior, and if so can be easily fixed by passing in `tuple(s)` rather than `s`.

New features

- 88f44a5d: TCP and TLS support added; TLS is currently limited to PKI certificates. This includes support for preserving the URI scheme in exchanges (0b0214db).
- a50da1a8: The credentials module was added to dispatch DTLS and OSCORE credentials
- f302da07: On the client side, OSCORE can now be used as a transport without any manual protection steps. It is automatically used for URIs for which a security context has been registered with the context's client credentials.
- 5e5388ae: Support for PyPy
- 0d09b2eb: NoResponse is now handled automatically. Handlers can override the default handling by setting a No-Response option on their response messages, whose value will them be examined by the library to decide whether the message is actually sent; the No-Response option is stripped from the outgoing message in the course of that (as it's actually not a response option).
- b048a50a: Some improvements on multicast handling. There is still no good support for sending a request to multicast and receiving the individual responses, but requests to multicast addresses are now unconditionally handled under the rules of multicast CoAP, even if they're used over the regular request interface (ie. sending to multicast but processing only the first response).
- c7ca0286: The software version used to run the server (by default, aiocoap's version) is now shown in .well-known/core using the impl-info relation.

Deprecations

- 0d09b2eb: Returning a NoResponse sentinel value is now deprecated.

Assorted changes

- Additions to the contrib/ collection of aiocoap based tools:
  - widgets, kivy-widgets
  - rd-relay
- 95c681a5 and others: Internal interfaces were introduced for the various CoAP sublayers. This should largely not affect operation (though it does change the choice of tokens or message IDs); where it does, it's noted above in the breaking changes.
- 5e5388ae, 9e17180e, 60137bd8: Various fixes to the OSCORE implementation, which is not considered experimental any more.
- Various additions to the test suite
- 61843d41: Asynchronous `recvmsg` calling (as used by the udp6 backend) was reworked from monkey-patching into using asyncio's `add_reader` method, and should thus now be usable on all asyncio implementations, including uvloop and gbulb.

- 3ab14c49: .well-known/core filtering will now properly filter by content format (ct=) in the presence of multiple supported content types.
- 9bd612de: Fix encoding of block size 16.
- 029a8f0e: Don't enforce V4MAPPED addresses in the simple6 backend. This makes the backend effectively a simple-any backend, as the address family can be picked arbitrarily by the operating system.
- 8e93eeb9: The simple6 backend now reuses the most recently used 64 sockets.
- cb8743b6: Resolve the name given as binding server name. This enables creating servers bound exclusively to a link-local address.
- d6aa5f8c: TinyDTLS now pulls in a more recent version of DTLSSocket that has its version negotiation fixed, and can thus interoperate with recent versions of libcoap and RIOT's the pending support for DTLS on Gcoap.
- 3d9613ab: Errors in URI encoding were fixed

6.9.10 Version 0.4a1

Security fixes

- 18ddf8c: Proxy now only creates log files when explicitly requested
- Support for secured protocols added (see Experimental Features)

Experimental features

- Support for OSCORE (formerly OSCOAP) and CoAP over DTLS was included

  These features both lack proper key management so far, which will be available in a 0.4 release.
- Added implementations of Resource Directory (RD) server and endpoint
- Support for different transports was added. The transport backends to enable are chosen heuristically depending on operating system and installed modules.
  - Transports for platforms not supporting all POSIX operations to run CoAP correctly were added (simple6, simplesocketserver). This should allow running aiocoap on Windows, MacOS and using uvloop, but with some disadvantages (see the the respective transport documentations).

Breaking changes

- 8641b5c: Blockwise handling is now available as stand-alone responder. Applications that previously created a Request object rather than using Protocol.request now need to create a BlockwiseRequest object.
- 8641b5c: The `.observation` property can now always be present in responses, and applications that previously checked for its presence should now check whether it is None.
- cdfeaeb: The multicast interface using queuewithend was replaced with asynchronous iterators
- d168f44: Handling of sub-sites changed, subsites' root resources now need to reside at path `("",)`

Deprecations

- e50e994: Rename UnsupportedMediaType to UnsupportedContentFormat

- 9add964 and others: The `.remote` message property is not necessarily a tuple any more, and has its own interface
- 25cbf54, c67c2c2: Drop support for Python versions < 3.4.4; the required version will be incremented to 3.5 soon.

Assorted changes

- 750d88d: Errors from predefined exceptions like BadRequest("...") are now sent with their text message in the diagnostic payload
- 3c7635f: Examples modernized
- 97fc5f7: Multicast handling changed (but is still not fully supported)
- 933f2b1: Added support for the No-Response option (RFC7967)
- baa84ee: V4MAPPED addresses are now properly displayed as IPv4 addresses

Tests

- Test suite is now run at Gitlab, and coverage reported
- b2396bf: Test suite probes for usable hostnames for localhost
- b4c5b1d: Allow running tests with a limited set of extras installed
- General improvements on coverage

6.9.11 Version 0.3

Features

- 4d07615: ICMP errors are handled
- 1b61a29: Accept 'fe80::...%eth0' style addresses
- 3c0120a: Observations provide modern `async for` interface
- 4e4ff7c: New demo: file server
- ef2e45e, 991098b, 684ccdd: Messages can be constructed with options, modified copies can be created with the `.copy` method, and default codes are provided
- 08845f2: Request objects have `.response_nonraising` and `.response_raising` interfaces for easier error handling
- ab5b88a, c49b5c8: Sites can be nested by adding them to an existing site, catch-all resources can be created by subclassing PathCapable

Possibly breaking changes

- ab5b88a: Site nesting means that server resources do not get their original Uri-Path any more
- bc76a7c: Location-{Path,Query} were opaque (bytes) objects instead of strings; disctinction between accidental and intentional opaque options is now clarified

Small features

- 2bb645e: set_request_uri allows URI parsing without sending Uri-Host
- e6b4839: Take block1.size_exponent as a sizing hint when sending block1 data
- 9eafd41: Allow passing in a loop into context creation
- 9ae5bdf: ObservableResource: Add update_observation_count
- c9f21a6: Stop client-side observations when unused
- dd46682: Drop dependency on obscure built-in IN module
- a18c067: Add numbers from draft-ietf-core-etch-04
- fabcfd5: .well-known/core supports filtering

Internals

- f968d3a: All low-level networking is now done in aiocoap.transports; it's not really hotpluggable yet and only UDPv6 (with implicit v4 support) is implemented, but an extension point for alternative transports.
- bde8c42: recvmsg is used instead of recvfrom, requiring some asyncio hacks

Package management

- 01f7232, 0a9d03c: aiocoap-client and -proxy are entry points
- 0e4389c: Establish an extra requirement for LinkHeader

6.10 LICENSE

Python Module Index

a

aiocoap, 21
aiocoap.cli, 81
aiocoap.defaults, 47
aiocoap.error, 42
aiocoap.interfaces, 38
aiocoap.message, 34
aiocoap.meta, 81
aiocoap.numbers, 64
aiocoap.numbers.codes, 64
aiocoap.numbers.constants, 66
aiocoap.numbers.contentformat, 67
aiocoap.numbers.optionnumbers, 68
aiocoap.numbers.types, 69
aiocoap.options, 37
aiocoap.optiontypes, 70
aiocoap.oscore, 81
aiocoap.pipe, 45
aiocoap.protocol, 30
aiocoap.proxy, 61
aiocoap.proxy.client, 61
aiocoap.proxy.server, 62
aiocoap.resource, 72
aiocoap.transports, 48
aiocoap.transports.generic_udp, 48
aiocoap.transports.oscore, 49
aiocoap.transports.rfc8323common, 50
aiocoap.transports.simple6, 51
aiocoap.transports.simplesocketserver, 52
aiocoap.transports.tcp, 52
aiocoap.transports.tinydtls, 54
aiocoap.transports.tinydtls_server, 56
aiocoap.transports.tls, 56
aiocoap.transports.udp6, 57
aiocoap.transports.ws, 60
aiocoap.util, 75
aiocoap.util.asyncio, 75
aiocoap.util.asyncio.recvmsg, 76
aiocoap.util.asyncio.timeoutdict, 76
aiocoap.util.cli, 77
aiocoap.util.contenttype, 77
aiocoap.util.cryptography_additions, 78
aiocoap.util.linkformat, 78
aiocoap.util.linkformat_pygments, 78
aiocoap.util.prettyprint, 78
aiocoap.util.socknumbers, 79
aiocoap.util.uri, 79

Index

A

A128GCM (*class in aiocoap.oscore*), 83
A192GCM (*class in aiocoap.oscore*), 84
A256GCM (*class in aiocoap.oscore*), 84
ABORT (*aiocoap.Code attribute*), 22
ABORT (*aiocoap.numbers.codes.Code attribute*), 65
abort() (*aiocoap.transports.rfc8323common.RFC8323Remote method*), 51
ACCEPT (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
ACCEPT (*aiocoap.OptionNumber attribute*), 24
accept (*aiocoap.options.Options attribute*), 37
accept() (*aiocoap.protocol.ServerObservation method*), 34
ACK (*aiocoap.numbers.types.Type attribute*), 70
ACK (*aiocoap.Type attribute*), 21
ACK_RANDOM_FACTOR (*in module aiocoap.numbers.constants*), 66
ACK_TIMEOUT (*in module aiocoap.numbers.constants*), 66
ActionNoYes (*class in aiocoap.util.cli*), 77
add_exception() (*aiocoap.pipe.IterablePipe method*), 47
add_exception() (*aiocoap.pipe.Pipe method*), 47
add_observation() (*aiocoap.interfaces.ObservableResource method*), 42
add_observation() (*aiocoap.proxy.server.ProxyWithPooledObservations method*), 63
add_observation() (*aiocoap.resource.ObservableResource method*), 73
add_observation() (*aiocoap.resource.Site method*), 75
add_option() (*aiocoap.options.Options method*), 37
add_redirector() (*aiocoap.proxy.server.Proxy method*), 62
add_resource() (*aiocoap.resource.Site method*), 75
add_response() (*aiocoap.pipe.IterablePipe method*), 47
add_response() (*aiocoap.pipe.Pipe method*), 47
AES_CCM (*class in aiocoap.oscore*), 82
AES_CCM_16_128_128 (*class in aiocoap.oscore*), 83
AES_CCM_16_128_256 (*class in aiocoap.oscore*), 83
AES_CCM_16_64_128 (*class in aiocoap.oscore*), 82
AES_CCM_16_64_256 (*class in aiocoap.oscore*), 82
AES_CCM_64_128_128 (*class in aiocoap.oscore*), 83
AES_CCM_64_128_256 (*class in aiocoap.oscore*), 83
AES_CCM_64_64_128 (*class in aiocoap.oscore*), 82
AES_CCM_64_64_256 (*class in aiocoap.oscore*), 83
AES_GCM (*class in aiocoap.oscore*), 83
aiocoap (*module*), 21
aiocoap.cli (*module*), 81
aiocoap.defaults (*module*), 47
aiocoap.error (*module*), 42
aiocoap.interfaces (*module*), 38
aiocoap.message (*module*), 34
aiocoap.meta (*module*), 81
aiocoap.numbers (*module*), 64
aiocoap.numbers.codes (*module*), 64
aiocoap.numbers.constants (*module*), 66
aiocoap.numbers.contentformat (*module*), 67
aiocoap.numbers.optionnumbers (*module*), 68
aiocoap.numbers.types (*module*), 69
aiocoap.options (*module*), 37
aiocoap.optiontypes (*module*), 70
aiocoap.oscore (*module*), 81
aiocoap.pipe (*module*), 45
aiocoap.protocol (*module*), 30
aiocoap.proxy (*module*), 61
aiocoap.proxy.client (*module*), 61
aiocoap.proxy.server (*module*), 62
aiocoap.resource (*module*), 72
aiocoap.transports (*module*), 48
aiocoap.transports.generic_udp (*module*), 48
aiocoap.transports.oscore (*module*), 49

aiocoap.transports.rfc8323common (*module*), 50
aiocoap.transports.simple6 (*module*), 51
aiocoap.transports.simplesocketserver (*module*), 52
aiocoap.transports.tcp (*module*), 52
aiocoap.transports.tinydtls (*module*), 54
aiocoap.transports.tinydtls_server (*module*), 56
aiocoap.transports.tls (*module*), 56
aiocoap.transports.udp6 (*module*), 57
aiocoap.transports.ws (*module*), 60
aiocoap.util (*module*), 75
aiocoap.util.asyncio (*module*), 75
aiocoap.util.asyncio.recvmsg (*module*), 76
aiocoap.util.asyncio.timeoutdict (*module*), 76
aiocoap.util.cli (*module*), 77
aiocoap.util.contenttype (*module*), 77
aiocoap.util.cryptography_additions (*module*), 78
aiocoap.util.linkformat (*module*), 78
aiocoap.util.linkformat_pygments (*module*), 78
aiocoap.util.prettyprint (*module*), 78
aiocoap.util.socknumbers (*module*), 79
aiocoap.util.uri (*module*), 79
Algorithm (*class in aiocoap.oscore*), 82
AlgorithmCountersign (*class in aiocoap.oscore*), 84
AnonymousHost, 45
apply_redirection() (*aiocoap.proxy.server.ForwardProxy method*), 63
apply_redirection() (*aiocoap.proxy.server.NameBasedVirtualHost method*), 63
apply_redirection() (*aiocoap.proxy.server.Proxy method*), 62
apply_redirection() (*aiocoap.proxy.server.Redirector method*), 63
apply_redirection() (*aiocoap.proxy.server.SubresourceVirtualHost method*), 64
apply_redirection() (*aiocoap.proxy.server.UnconditionalRedirector method*), 64
as_response_address() (*aiocoap.interfaces.EndpointAddress method*), 40
as_response_address() (*aiocoap.transports.udp6.UDP6EndpointAddress method*), 58
AsyncCLIDaemon (*class in aiocoap.util.cli*), 77
authenticated_claims (*aiocoap.interfaces.EndpointAddress attribute*), 40
authenticated_claims (*aiocoap.oscore.BaseSecurityContext attribute*), 85
authenticated_claims (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50

B

BAD_GATEWAY (*aiocoap.Code attribute*), 22
BAD_GATEWAY (*aiocoap.numbers.codes.Code attribute*), 65
BAD_OPTION (*aiocoap.Code attribute*), 22
BAD_OPTION (*aiocoap.numbers.codes.Code attribute*), 65
BAD_REQUEST (*aiocoap.Code attribute*), 22
BAD_REQUEST (*aiocoap.numbers.codes.Code attribute*), 64
BadRequest, 43
BaseRequest (*class in aiocoap.protocol*), 33
BaseSecurityContext (*class in aiocoap.oscore*), 85
BaseUnicastRequest (*class in aiocoap.protocol*), 33
BLOCK1 (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
BLOCK1 (*aiocoap.OptionNumber attribute*), 24
block1 (*aiocoap.options.Options attribute*), 37
BLOCK2 (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
BLOCK2 (*aiocoap.OptionNumber attribute*), 24
block2 (*aiocoap.options.Options attribute*), 37
BlockOption (*class in aiocoap.optiontypes*), 71
BlockOption.BlockwiseTuple (*class in aiocoap.optiontypes*), 71
blockwise_key (*aiocoap.interfaces.EndpointAddress attribute*), 40
blockwise_key (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50
blockwise_key (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
blockwise_key (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55
blockwise_key (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 59
BlockwiseRequest (*class in aiocoap.protocol*), 33
by_media_type() (*aiocoap.ContentFormat class method*), 25

by_media_type() (*aiocoap.numbers.contentformat.ContentFormat class method*), 68

C

callback() (*aiocoap.protocol.ClientObservation method*), 33
can_have_payload() (*aiocoap.Code method*), 23
can_have_payload() (*aiocoap.numbers.codes.Code method*), 65
cancel() (*aiocoap.protocol.ClientObservation method*), 33
CanNotRedirect, 62
CanNotRedirectBecauseOfUnsafeOptions, 62
CanProtect (*class in aiocoap.oscore*), 85
CanUnprotect (*class in aiocoap.oscore*), 86
categorize() (*in module aiocoap.util.contenttype*), 77
CBOR (*aiocoap.ContentFormat attribute*), 25
CBOR (*aiocoap.numbers.contentformat.ContentFormat attribute*), 68
ChaCha20Poly1305 (*class in aiocoap.oscore*), 84
CHANGED (*aiocoap.Code attribute*), 22
CHANGED (*aiocoap.numbers.codes.Code attribute*), 64
class_ (*aiocoap.Code attribute*), 23
class_ (*aiocoap.numbers.codes.Code attribute*), 65
client_credentials (*aiocoap.interfaces.MessageManager attribute*), 40
ClientObservation (*class in aiocoap.protocol*), 33
close() (*aiocoap.util.asyncio.recvmsg.RecvmsgSelectorDatagramTransport method*), 76
CloseConnection, 50
CloseNotifyReceived, 54
COAP_PORT (*in module aiocoap.numbers.constants*), 66
code (*aiocoap.error.BadRequest attribute*), 43
code (*aiocoap.error.ConstructionRenderableError attribute*), 43
code (*aiocoap.error.MethodNotAllowed attribute*), 43
code (*aiocoap.error.NotFound attribute*), 43
code (*aiocoap.error.Unauthorized attribute*), 43
code (*aiocoap.error.UnsupportedContentFormat attribute*), 43
code (*aiocoap.proxy.server.CanNotRedirectBecauseOfUnsafeOptions attribute*), 62
code (*aiocoap.proxy.server.IncompleteProxyUri attribute*), 62
code (*aiocoap.proxy.server.NoSuchHostname attribute*), 62
code (*aiocoap.proxy.server.NotAForwardProxy attribute*), 62
code (*aiocoap.proxy.server.NoUriSplitting attribute*), 62
Code (*class in aiocoap*), 21
Code (*class in aiocoap.numbers.codes*), 64
CON (*aiocoap.numbers.types.Type attribute*), 70
CON (*aiocoap.Type attribute*), 21
CONFLICT (*aiocoap.Code attribute*), 22
CONFLICT (*aiocoap.numbers.codes.Code attribute*), 65
connection_lost() (*aiocoap.transports.tcp.TcpConnection method*), 53
connection_lost() (*aiocoap.transports.tinydtls.DTLSClientConnection.SingleConnection method*), 55
connection_lost() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
connection_made() (*aiocoap.transports.tcp.TcpConnection method*), 52
connection_made() (*aiocoap.transports.tinydtls.DTLSClientConnection.SingleConnection method*), 55
connection_made() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
ConRetransmitsExceeded, 44
ConstructionRenderableError, 42
CONTENT (*aiocoap.Code attribute*), 22
CONTENT (*aiocoap.numbers.codes.Code attribute*), 64
CONTENT_FORMAT (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
CONTENT_FORMAT (*aiocoap.OptionNumber attribute*), 23
content_format (*aiocoap.options.Options attribute*), 37
ContentFormat (*class in aiocoap*), 24
ContentFormat (*class in aiocoap.numbers.contentformat*), 67
ContentFormatOption (*class in aiocoap.optiontypes*), 71
Context (*class in aiocoap*), 28
Context (*class in aiocoap.protocol*), 30
context_for_response() (*aiocoap.oscore.CanUnprotect method*), 86
context_from_response() (*aiocoap.oscore.CanProtect method*), 86
context_from_response() (*aiocoap.oscore.SimpleGroupContext method*), 89
ContextUnavailable, 81
CONTINUE (*aiocoap.Code attribute*), 22
CONTINUE (*aiocoap.numbers.codes.Code attribute*), 64
copy() (*aiocoap.Message method*), 27
copy() (*aiocoap.message.Message method*), 35
create_client_context() (*aiocoap.Context*

class method), 28, 29
create_client_context() (*aiocoap.protocol.Context class method*), 31, 32
create_client_transport() (*aiocoap.transports.tcp.TCPClient class method*), 53
create_client_transport_endpoint() (*aiocoap.transports.simple6.MessageInterfaceSimple6 class method*), 52
create_client_transport_endpoint() (*aiocoap.transports.tinydtls.MessageInterfaceTinyDTLS class method*), 55
create_client_transport_endpoint() (*aiocoap.transports.udp6.MessageInterfaceUDP6 class method*), 59
create_option() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
create_option() (*aiocoap.OptionNumber method*), 24
create_recvmsg_datagram_endpoint() (*in module aiocoap.util.asyncio.recvmsg*), 76
create_server() (*aiocoap.transports.simplesocketserver.MessageInterfaceSimpleServer class method*), 52
create_server() (*aiocoap.transports.tcp.TCPServer class method*), 53
create_server() (*aiocoap.transports.tinydtls_server.MessageInterfaceTinyDTLSServer class method*), 56
create_server() (*aiocoap.transports.tls.TLSServer class method*), 57
create_server_context() (*aiocoap.Context class method*), 28, 29
create_server_context() (*aiocoap.protocol.Context class method*), 31, 32
create_server_transport_endpoint() (*aiocoap.transports.udp6.MessageInterfaceUDP6 class method*), 59
create_transport() (*aiocoap.transports.ws.WSPool class method*), 61
CREATED (*aiocoap.Code attribute*), 22
CREATED (*aiocoap.numbers.codes.Code attribute*), 64
CSM (*aiocoap.Code attribute*), 22
CSM (*aiocoap.numbers.codes.Code attribute*), 65
ct (*aiocoap.resource.WKCResource attribute*), 74

D

data_received() (*aiocoap.transports.tcp.TcpConnection method*), 53
datagram_errqueue_received() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
datagram_errqueue_received() (*aiocoap.util.asyncio.recvmsg.RecvmsgDatagramProtocol method*), 76
datagram_msg_received() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
datagram_msg_received() (*aiocoap.util.asyncio.recvmsg.RecvmsgDatagramProtocol method*), 76
datagram_received() (*aiocoap.transports.tinydtls.DTLSClientConnection.SingleConnectio method*), 55
decode() (*aiocoap.Message class method*), 27
decode() (*aiocoap.message.Message class method*), 35
decode() (*aiocoap.options.Options method*), 37
decode() (*aiocoap.optiontypes.BlockOption method*), 71
decode() (*aiocoap.optiontypes.ContentFormatOption method*), 72
decode() (*aiocoap.optiontypes.OpaqueOption method*), 70
decode() (*aiocoap.optiontypes.OptionType method*), 70
decode() (*aiocoap.optiontypes.StringOption method*), 70
decode() (*aiocoap.optiontypes.UintOption method*), 70
decode_dss_signature() (*in module aiocoap.oscore*), 89
DecodeError, 81
decrypt() (*aiocoap.oscore.AES_CCM class method*), 82
decrypt() (*aiocoap.oscore.AES_GCM class method*), 83
decrypt() (*aiocoap.oscore.Algorithm method*), 82
decrypt() (*aiocoap.oscore.ChaCha20Poly1305 class method*), 84
DEFAULT_BLOCK_SIZE_EXP (*in module aiocoap.numbers.constants*), 67
DELETE (*aiocoap.Code attribute*), 21
DELETE (*aiocoap.numbers.codes.Code attribute*), 64
delete_option() (*aiocoap.options.Options method*), 37
DELETED (*aiocoap.Code attribute*), 22
DELETED (*aiocoap.numbers.codes.Code attribute*), 64
deregister() (*aiocoap.protocol.ServerObservation method*), 34
derive_keys() (*aiocoap.oscore.SecurityContextUtils method*), 86

derive_keys() (*aiocoap.oscore.SimpleGroupContext method*), 89
determine_remote() (*aiocoap.interfaces.MessageInterface method*), 38
determine_remote() (*aiocoap.transports.generic_udp.GenericMessageInterface method*), 49
determine_remote() (*aiocoap.transports.tinydtls.MessageInterfaceTinyDTLS method*), 55
determine_remote() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
dispatch_error() (*aiocoap.interfaces.MessageManager method*), 40
dispatch_message() (*aiocoap.interfaces.MessageManager method*), 40
dotted (*aiocoap.Code attribute*), 23
dotted (*aiocoap.numbers.codes.Code attribute*), 66
dtls_missing_modules() (*in module aiocoap.defaults*), 48
DTLSClientConnection (*class in aiocoap.transports.tinydtls*), 54
DTLSClientConnection.SingleConnection (*class in aiocoap.transports.tinydtls*), 55

E

ECDSA_SHA256_P256 (*class in aiocoap.oscore*), 85
ECHO (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
ECHO (*aiocoap.OptionNumber attribute*), 24
echo (*aiocoap.options.Options attribute*), 38
Ed25519 (*class in aiocoap.oscore*), 85
EDHOC (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
EDHOC (*aiocoap.OptionNumber attribute*), 24
edhoc (*aiocoap.options.Options attribute*), 38
EMPTY (*aiocoap.Code attribute*), 21
EMPTY (*aiocoap.numbers.codes.Code attribute*), 64
EMPTY_ACK_DELAY (*in module aiocoap.numbers.constants*), 67
encode() (*aiocoap.Message method*), 27
encode() (*aiocoap.message.Message method*), 35
encode() (*aiocoap.options.Options method*), 37
encode() (*aiocoap.optiontypes.BlockOption method*), 71
encode() (*aiocoap.optiontypes.ContentFormatOption method*), 72
encode() (*aiocoap.optiontypes.OpaqueOption method*), 70
encode() (*aiocoap.optiontypes.OptionType method*), 70
encode() (*aiocoap.optiontypes.StringOption method*), 70
encode() (*aiocoap.optiontypes.UintOption method*), 70
encode_dss_signature() (*in module aiocoap.oscore*), 89
encrypt() (*aiocoap.oscore.AES_CCM class method*), 82
encrypt() (*aiocoap.oscore.AES_GCM class method*), 83
encrypt() (*aiocoap.oscore.Algorithm method*), 82
encrypt() (*aiocoap.oscore.ChaCha20Poly1305 class method*), 84
EndpointAddress (*class in aiocoap.interfaces*), 39
eof_received() (*aiocoap.transports.tcp.TcpConnection method*), 53
Error, 42
error() (*aiocoap.protocol.ClientObservation method*), 33
error_received() (*aiocoap.transports.tinydtls.DTLSClientConnection.SingleConnection method*), 55
error_received() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
error_received() (*aiocoap.util.asyncio.recvmsg.RecvmsgDatagramProtocol method*), 76
error_to_message() (*in module aiocoap.pipe*), 47
ETAG (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
ETAG (*aiocoap.OptionNumber attribute*), 23
etag (*aiocoap.options.Options attribute*), 37
etags (*aiocoap.options.Options attribute*), 37
exception (*aiocoap.pipe.Pipe.Event attribute*), 46
EXCHANGE_LIFETIME (*in module aiocoap.numbers.constants*), 66
ExtensibleEnumMeta (*class in aiocoap.util*), 80
ExtensibleIntEnum (*class in aiocoap.util*), 80
external_aad_is_group (*aiocoap.oscore.BaseSecurityContext attribute*), 85
external_aad_is_group (*aiocoap.oscore.GroupContext attribute*), 88

F

factory() (*aiocoap.transports.tinydtls.DTLSClientConnection.SingleConnection class method*), 55
FatalDTLSError, 54
FETCH (*aiocoap.Code attribute*), 21
FETCH (*aiocoap.numbers.codes.Code attribute*), 64

FilesystemSecurityContext (*class in aiocoap.oscore*), 87
FilesystemSecurityContext.LoadError, 88
fill_or_recognize_remote() (*aiocoap.interfaces.RequestInterface method*), 41
fill_or_recognize_remote() (*aiocoap.interfaces.TokenInterface method*), 41
fill_or_recognize_remote() (*aiocoap.transports.oscore.TransportOSCORE method*), 50
fill_or_recognize_remote() (*aiocoap.transports.tcp.TCPClient method*), 53
fill_or_recognize_remote() (*aiocoap.transports.tcp.TCPServer method*), 53
fill_or_recognize_remote() (*aiocoap.transports.ws.WSPool method*), 61
find_remote_and_interface() (*aiocoap.Context method*), 30
find_remote_and_interface() (*aiocoap.protocol.Context method*), 32
for_sending_deterministic_requests() (*aiocoap.oscore.SimpleGroupContext method*), 89
FORBIDDEN (*aiocoap.Code attribute*), 22
FORBIDDEN (*aiocoap.numbers.codes.Code attribute*), 65
format (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
format (*aiocoap.OptionNumber attribute*), 24
ForwardProxy (*class in aiocoap.proxy.server*), 63
ForwardProxyWithPooledObservations (*class in aiocoap.proxy.server*), 63
from_private_parts() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85
from_public_parts() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85

G

GATEWAY_TIMEOUT (*aiocoap.Code attribute*), 22
GATEWAY_TIMEOUT (*aiocoap.numbers.codes.Code attribute*), 65
generate() (*aiocoap.oscore.AlgorithmCountersign method*), 84
generate() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85
generate() (*aiocoap.oscore.Ed25519 method*), 85
GenericMessageInterface (*class in aiocoap.transports.generic_udp*), 48
GET (*aiocoap.Code attribute*), 21
GET (*aiocoap.numbers.codes.Code attribute*), 64
get_cache_key() (*aiocoap.Message method*), 27
get_cache_key() (*aiocoap.message.Message method*), 35
get_default_clienttransports() (*in module aiocoap.defaults*), 47
get_default_servertransports() (*in module aiocoap.defaults*), 48
get_link_description() (*aiocoap.resource.ObservableResource method*), 73
get_option() (*aiocoap.options.Options method*), 37
get_oscore_context_for() (*aiocoap.oscore.SecurityContextUtils method*), 86
get_oscore_context_for() (*aiocoap.oscore.SimpleGroupContext method*), 89
get_request_uri() (*aiocoap.Message method*), 27
get_request_uri() (*aiocoap.message.Message method*), 36
get_resources_as_linkheader() (*aiocoap.resource.Site method*), 75
get_reusable_nonce() (*aiocoap.oscore.RequestIdentifiers method*), 82
GoingThroughMessageDecryption (*class in aiocoap.transports.tinydtls_server*), 56
GroupContext (*class in aiocoap.oscore*), 88

H

HAS_RECVERR (*in module aiocoap.util.socknumbers*), 79
has_reuse_port() (*in module aiocoap.defaults*), 48
hashing_etag() (*in module aiocoap.resource*), 72
HOP_LIMIT (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
HOP_LIMIT (*aiocoap.OptionNumber attribute*), 24
hop_limit (*aiocoap.options.Options attribute*), 38
HOP_LIMIT_REACHED (*aiocoap.Code attribute*), 22
HOP_LIMIT_REACHED (*aiocoap.numbers.codes.Code attribute*), 65
hostinfo (*aiocoap.interfaces.EndpointAddress attribute*), 39
hostinfo (*aiocoap.transports.oscore.OSCOREAddress attribute*), 49
hostinfo (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
hostinfo (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55
hostinfo (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58

hostinfo (*aiocoap.transports.ws.PoolKey attribute*), 60
hostinfo_local (*aiocoap.interfaces.EndpointAddress attribute*), 39
hostinfo_local (*aiocoap.transports.oscore.OSCOREAddress attribute*), 49
hostinfo_local (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
hostinfo_local (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55
hostinfo_local (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
hostportjoin() (*in module aiocoap.util*), 80
hostportsplit() (*in module aiocoap.util*), 80

I

IF_MATCH (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
IF_MATCH (*aiocoap.OptionNumber attribute*), 23
if_match (*aiocoap.options.Options attribute*), 38
IF_NONE_MATCH (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
IF_NONE_MATCH (*aiocoap.OptionNumber attribute*), 23
if_none_match (*aiocoap.options.Options attribute*), 37
IncompleteProxyUri, 62
initialize_empty() (*aiocoap.oscore.ReplayWindow method*), 87
initialize_from_freshlyseen() (*aiocoap.oscore.ReplayWindow method*), 87
initialize_from_persisted() (*aiocoap.oscore.ReplayWindow method*), 87
interface (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
InterfaceOnlyPktinfo (*class in aiocoap.transports.udp6*), 57
INTERNAL_SERVER_ERROR (*aiocoap.Code attribute*), 22
INTERNAL_SERVER_ERROR (*aiocoap.numbers.codes.Code attribute*), 65
interpret_block_options (*aiocoap.proxy.server.Proxy attribute*), 62
iPATCH (*aiocoap.Code attribute*), 21
iPATCH (*aiocoap.numbers.codes.Code attribute*), 64
is_bert (*aiocoap.optiontypes.BlockOption.BlockwiseTuple attribute*), 71
is_cachekey() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
is_cachekey() (*aiocoap.OptionNumber method*), 24
is_critical() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
is_critical() (*aiocoap.OptionNumber method*), 24
is_elective() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
is_elective() (*aiocoap.OptionNumber method*), 24
is_initialized() (*aiocoap.oscore.ReplayWindow method*), 87
is_known() (*aiocoap.ContentFormat method*), 25
is_known() (*aiocoap.numbers.contentformat.ContentFormat method*), 68
is_last (*aiocoap.pipe.Pipe.Event attribute*), 46
is_multicast (*aiocoap.interfaces.EndpointAddress attribute*), 39
is_multicast (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50
is_multicast (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
is_multicast (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 54
is_multicast (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
is_multicast_locally (*aiocoap.interfaces.EndpointAddress attribute*), 39
is_multicast_locally (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
is_multicast_locally (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 54
is_multicast_locally (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
is_nocachekey() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
is_nocachekey() (*aiocoap.OptionNumber method*), 24
is_request() (*aiocoap.Code method*), 22
is_request() (*aiocoap.numbers.codes.Code method*), 65
is_response() (*aiocoap.Code method*), 22
is_response() (*aiocoap.numbers.codes.Code*

method), 65
is_safetoforward() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
is_safetoforward() (*aiocoap.OptionNumber method*), 24
is_signalling() (*aiocoap.Code method*), 23
is_signalling() (*aiocoap.numbers.codes.Code method*), 65
is_signing (*aiocoap.oscore.CanProtect attribute*), 85
is_signing (*aiocoap.oscore.GroupContext attribute*), 88
is_successful() (*aiocoap.Code method*), 23
is_successful() (*aiocoap.numbers.codes.Code method*), 65
is_unsafe() (*aiocoap.numbers.optionnumbers.OptionNumber method*), 69
is_unsafe() (*aiocoap.OptionNumber method*), 24
is_valid() (*aiocoap.oscore.ReplayWindow method*), 87
is_valid_for_payload_size() (*aiocoap.optiontypes.BlockOption.BlockwiseTuple method*), 71
IterablePipe (*class in aiocoap.pipe*), 47
IterablePipe.Iterator (*class in aiocoap.pipe*), 47
iv_bytes (*aiocoap.oscore.AES_CCM_16_128_128 attribute*), 83
iv_bytes (*aiocoap.oscore.AES_CCM_16_128_256 attribute*), 83
iv_bytes (*aiocoap.oscore.AES_CCM_16_64_128 attribute*), 82
iv_bytes (*aiocoap.oscore.AES_CCM_16_64_256 attribute*), 82
iv_bytes (*aiocoap.oscore.AES_CCM_64_128_128 attribute*), 83
iv_bytes (*aiocoap.oscore.AES_CCM_64_128_256 attribute*), 83
iv_bytes (*aiocoap.oscore.AES_CCM_64_64_128 attribute*), 82
iv_bytes (*aiocoap.oscore.AES_CCM_64_64_256 attribute*), 83
iv_bytes (*aiocoap.oscore.AES_GCM attribute*), 83
iv_bytes (*aiocoap.oscore.ChaCha20Poly1305 attribute*), 84

J

JSON (*aiocoap.ContentFormat attribute*), 25
JSON (*aiocoap.numbers.contentformat.ContentFormat attribute*), 68

K

key_bytes (*aiocoap.oscore.A128GCM attribute*), 84
key_bytes (*aiocoap.oscore.A192GCM attribute*), 84
key_bytes (*aiocoap.oscore.A256GCM attribute*), 84
key_bytes (*aiocoap.oscore.AES_CCM_16_128_128 attribute*), 83
key_bytes (*aiocoap.oscore.AES_CCM_16_128_256 attribute*), 83
key_bytes (*aiocoap.oscore.AES_CCM_16_64_128 attribute*), 82
key_bytes (*aiocoap.oscore.AES_CCM_16_64_256 attribute*), 82
key_bytes (*aiocoap.oscore.AES_CCM_64_128_128 attribute*), 83
key_bytes (*aiocoap.oscore.AES_CCM_64_128_256 attribute*), 83
key_bytes (*aiocoap.oscore.AES_CCM_64_64_128 attribute*), 82
key_bytes (*aiocoap.oscore.AES_CCM_64_64_256 attribute*), 83
key_bytes (*aiocoap.oscore.ChaCha20Poly1305 attribute*), 84
keys() (*aiocoap.transports.tinydtls_server.SecurityStore method*), 56

L

lexer_for_mime() (*in module aiocoap.util.prettyprint*), 78
library_uri (*in module aiocoap.meta*), 81
LibraryShutdown, 45
Link (*class in aiocoap.util.linkformat*), 78
link_format_to_message() (*in module aiocoap.resource*), 73
LINKFORMAT (*aiocoap.ContentFormat attribute*), 25
LINKFORMAT (*aiocoap.numbers.contentformat.ContentFormat attribute*), 68
LinkFormat (*class in aiocoap.util.linkformat*), 78
LinkFormatLexer (*class in aiocoap.util.linkformat_pygments*), 78
linkheader_missing_modules() (*in module aiocoap.defaults*), 48
load() (*aiocoap.transports.udp6.SockExtendedErr class method*), 59
LOCATION_PATH (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
LOCATION_PATH (*aiocoap.OptionNumber attribute*), 23
location_path (*aiocoap.options.Options attribute*), 37
LOCATION_QUERY (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
LOCATION_QUERY (*aiocoap.OptionNumber attribute*), 24
location_query (*aiocoap.options.Options attribute*), 37

log (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55

M

MAX_AGE (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
MAX_AGE (*aiocoap.OptionNumber attribute*), 23
max_age (*aiocoap.options.Options attribute*), 38
MAX_LATENCY (*in module aiocoap.numbers.constants*), 66
MAX_RETRANSMIT (*in module aiocoap.numbers.constants*), 66
MAX_RTT (*in module aiocoap.numbers.constants*), 66
max_size (*aiocoap.util.asyncio.recvmsg.RecvmsgSelectorDatagramTransport attribute*), 76
MAX_TRANSMIT_SPAN (*in module aiocoap.numbers.constants*), 66
MAX_TRANSMIT_WAIT (*in module aiocoap.numbers.constants*), 66
maximum_block_size_exp (*aiocoap.interfaces.EndpointAddress attribute*), 40
maximum_block_size_exp (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50
maximum_block_size_exp (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
maximum_payload_size (*aiocoap.interfaces.EndpointAddress attribute*), 40
maximum_payload_size (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50
maximum_payload_size (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
message (*aiocoap.error.ConstructionRenderableError attribute*), 43
message (*aiocoap.error.NoResource attribute*), 43
message (*aiocoap.error.UnallowedMethod attribute*), 43
message (*aiocoap.error.UnsupportedMethod attribute*), 43
message (*aiocoap.pipe.Pipe.Event attribute*), 46
message (*aiocoap.proxy.server.CanNotRedirect attribute*), 62
message (*aiocoap.proxy.server.IncompleteProxyUri attribute*), 62
message (*aiocoap.proxy.server.NoSuchHostname attribute*), 62
message (*aiocoap.proxy.server.NotAForwardProxy attribute*), 62
message (*aiocoap.proxy.server.NoUriSplitting attribute*), 62
Message (*class in aiocoap*), 25
Message (*class in aiocoap.message*), 34
MessageError, 44
MessageInterface (*class in aiocoap.interfaces*), 38
MessageInterfaceSimple6 (*class in aiocoap.transports.simple6*), 51
MessageInterfaceSimpleServer (*class in aiocoap.transports.simplesocketserver*), 52
MessageInterfaceTinyDTLS (*class in aiocoap.transports.tinydtls*), 55
MessageInterfaceTinyDTLSServer (*class in aiocoap.transports.tinydtls_server*), 56
MessageInterfaceUDP6 (*class in aiocoap.transports.udp6*), 59
MessageManager (*class in aiocoap.interfaces*), 40
METHOD_NOT_ALLOWED (*aiocoap.Code attribute*), 22
METHOD_NOT_ALLOWED (*aiocoap.numbers.codes.Code attribute*), 65
MethodNotAllowed, 43
mimetypes (*aiocoap.util.linkformat_pygments.LinkFormatLexer attribute*), 78
MissingBlock2Option, 45

N

name (*aiocoap.Code attribute*), 23
name (*aiocoap.numbers.codes.Code attribute*), 66
name (*aiocoap.util.linkformat_pygments.LinkFormatLexer attribute*), 78
name_printable (*aiocoap.Code attribute*), 23
name_printable (*aiocoap.numbers.codes.Code attribute*), 66
NameBasedVirtualHost (*class in aiocoap.proxy.server*), 63
needs_blockwise_assembly() (*aiocoap.interfaces.Resource method*), 41
needs_blockwise_assembly() (*aiocoap.proxy.server.Proxy method*), 62
needs_blockwise_assembly() (*aiocoap.resource.Resource method*), 73
needs_blockwise_assembly() (*aiocoap.resource.Site method*), 75
netif (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
NetworkError, 43
new_sequence_number() (*aiocoap.oscore.CanProtect method*), 86
NO_RESPONSE (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
NO_RESPONSE (*aiocoap.OptionNumber attribute*), 24
no_response (*aiocoap.options.Options attribute*), 38
NON (*aiocoap.numbers.types.Type attribute*), 70
NON (*aiocoap.Type attribute*), 21

NoResource, 43
NoResponse (*in module aiocoap.message*), 36
NoSuchHostname, 62
NOT_ACCEPTABLE (*aiocoap.Code attribute*), 22
NOT_ACCEPTABLE (*aiocoap.numbers.codes.Code attribute*), 65
NOT_FOUND (*aiocoap.Code attribute*), 22
NOT_FOUND (*aiocoap.numbers.codes.Code attribute*), 65
NOT_IMPLEMENTED (*aiocoap.Code attribute*), 22
NOT_IMPLEMENTED (*aiocoap.numbers.codes.Code attribute*), 65
NotAForwardProxy, 62
NotAProtectedMessage, 81
NotFound, 43
NotImplemented, 44
NotObservable, 45
NoUriSplitting, 62
NSTART (*in module aiocoap.numbers.constants*), 66

O

OBJECT_SECURITY (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
OBJECT_SECURITY (*aiocoap.OptionNumber attribute*), 23
object_security (*aiocoap.options.Options attribute*), 38
ObservableResource (*class in aiocoap.interfaces*), 42
ObservableResource (*class in aiocoap.resource*), 73
OBSERVATION_RESET_TIME (*in module aiocoap.numbers.constants*), 67
ObservationCancelled, 45
OBSERVE (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
OBSERVE (*aiocoap.OptionNumber attribute*), 23
observe (*aiocoap.options.Options attribute*), 37
OCTETSTREAM (*aiocoap.ContentFormat attribute*), 25
OCTETSTREAM (*aiocoap.numbers.contentformat.ContentFormat attribute*), 68
on_cancel() (*aiocoap.protocol.ClientObservation method*), 33
on_event() (*aiocoap.pipe.Pipe method*), 46
on_interest_end() (*aiocoap.pipe.IterablePipe method*), 47
on_interest_end() (*aiocoap.pipe.Pipe method*), 46
OpaqueOption (*class in aiocoap.optiontypes*), 70
option_list() (*aiocoap.options.Options method*), 37
OptionNumber (*class in aiocoap*), 23
OptionNumber (*class in aiocoap.numbers.optionnumbers*), 68
Options (*class in aiocoap.options*), 37
OptionType (*class in aiocoap.optiontypes*), 70
OSCORE (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
OSCORE (*aiocoap.OptionNumber attribute*), 23
oscore_missing_modules() (*in module aiocoap.defaults*), 48
OSCOREAddress (*class in aiocoap.transports.oscore*), 49

P

pairwise_for() (*aiocoap.oscore.SimpleGroupContext method*), 89
parent (*aiocoap.transports.tinydtls.DTLSClientConnection.SingleConnec attribute*), 55
parse() (*in module aiocoap.util.linkformat*), 78
PATCH (*aiocoap.Code attribute*), 21
PATCH (*aiocoap.numbers.codes.Code attribute*), 64
PathCapable (*class in aiocoap.resource*), 74
pause_writing() (*aiocoap.transports.tcp.TcpConnection method*), 53
persist() (*aiocoap.oscore.ReplayWindow method*), 87
PING (*aiocoap.Code attribute*), 22
PING (*aiocoap.numbers.codes.Code attribute*), 65
Pipe (*class in aiocoap.pipe*), 45
Pipe.Event (*class in aiocoap.pipe*), 46
pk_to_curve25519() (*in module aiocoap.util.cryptography_additions*), 78
poke() (*aiocoap.pipe.Pipe method*), 46
PONG (*aiocoap.Code attribute*), 22
PONG (*aiocoap.numbers.codes.Code attribute*), 65
PoolKey (*class in aiocoap.transports.ws*), 60
POST (*aiocoap.Code attribute*), 21
POST (*aiocoap.numbers.codes.Code attribute*), 64
post_seqnoincrease() (*aiocoap.oscore.CanProtect method*), 86
post_seqnoincrease() (*aiocoap.oscore.FilesystemSecurityContext method*), 88
post_seqnoincrease() (*aiocoap.oscore.SimpleGroupContext method*), 89
PRECONDITION_FAILED (*aiocoap.Code attribute*), 22
PRECONDITION_FAILED (*aiocoap.numbers.codes.Code attribute*), 65
pretty_print() (*in module aiocoap.util.prettyprint*), 78
prettyprint_missing_modules() (*in module aiocoap.defaults*), 48
private_key (*aiocoap.oscore.GroupContext attribute*), 88

private_key (*aiocoap.oscore.SimpleGroupContext attribute*), 89
PROCESSING_DELAY (*in module aiocoap.numbers.constants*), 66
protect() (*aiocoap.oscore.CanProtect method*), 86
ProtectionInvalid, 81
proxy (*aiocoap.proxy.client.ProxyForwarder attribute*), 61
Proxy (*class in aiocoap.proxy.server*), 62
PROXY_SCHEME (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
PROXY_SCHEME (*aiocoap.OptionNumber attribute*), 24
proxy_scheme (*aiocoap.options.Options attribute*), 38
PROXY_URI (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
PROXY_URI (*aiocoap.OptionNumber attribute*), 24
proxy_uri (*aiocoap.options.Options attribute*), 38
ProxyForwarder (*class in aiocoap.proxy.client*), 61
PROXYING_NOT_SUPPORTED (*aiocoap.Code attribute*), 22
PROXYING_NOT_SUPPORTED (*aiocoap.numbers.codes.Code attribute*), 65
ProxyWithPooledObservations (*class in aiocoap.proxy.server*), 63
public_from_private() (*aiocoap.oscore.AlgorithmCountersign method*), 84
public_from_private() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85
public_from_private() (*aiocoap.oscore.Ed25519 method*), 85
PUT (*aiocoap.Code attribute*), 21
PUT (*aiocoap.numbers.codes.Code attribute*), 64
py38args() (*in module aiocoap.util.asyncio*), 75

Q

Q_BLOCK1 (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
Q_BLOCK1 (*aiocoap.OptionNumber attribute*), 24
Q_BLOCK2 (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
Q_BLOCK2 (*aiocoap.OptionNumber attribute*), 24
quote_factory() (*in module aiocoap.util.uri*), 80
quote_nonascii() (*in module aiocoap.util*), 80

R

raise_unless_safe() (*in module aiocoap.proxy.server*), 62
ready (*aiocoap.transports.udp6.MessageInterfaceUDP6 attribute*), 59
recipient_public_key (*aiocoap.oscore.GroupContext attribute*), 88
recipient_public_key (*aiocoap.oscore.SimpleGroupContext attribute*), 89
recognize_remote() (*aiocoap.transports.simple6.MessageInterfaceSimple6 method*), 52
recognize_remote() (*aiocoap.transports.simplesocketserver.MessageInterfaceSimpleServe method*), 52
recognize_remote() (*aiocoap.transports.tinydtls.MessageInterfaceTinyDTLS method*), 55
recognize_remote() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
RecvmsgDatagramProtocol (*class in aiocoap.util.asyncio.recvmsg*), 76
RecvmsgSelectorDatagramTransport (*class in aiocoap.util.asyncio.recvmsg*), 76
Redirector (*class in aiocoap.proxy.server*), 63
reduced_to() (*aiocoap.optiontypes.BlockOption.BlockwiseTuple method*), 71
register_callback() (*aiocoap.protocol.ClientObservation method*), 33
register_errback() (*aiocoap.protocol.ClientObservation method*), 33
RELEASE (*aiocoap.Code attribute*), 22
RELEASE (*aiocoap.numbers.codes.Code attribute*), 65
release() (*aiocoap.transports.rfc8323common.RFC8323Remote method*), 51
release() (*aiocoap.transports.ws.WSRemote method*), 60
RemoteServerShutdown, 44
remove_resource() (*aiocoap.resource.Site method*), 75
render() (*aiocoap.interfaces.Resource method*), 41
render() (*aiocoap.proxy.server.Proxy method*), 63
render() (*aiocoap.proxy.server.ProxyWithPooledObservations method*), 63
render() (*aiocoap.resource.Resource method*), 73
render() (*aiocoap.resource.Site method*), 75
render_get() (*aiocoap.resource.WKCResource method*), 74
render_to_pipe() (*aiocoap.Context method*), 29
render_to_pipe() (*aiocoap.interfaces.ObservableResource method*), 42
render_to_pipe() (*aiocoap.interfaces.Resource method*), 42
render_to_pipe() (*aiocoap.protocol.Context method*), 32

render_to_pipe() (*aiocoap.proxy.server.Proxy method*), 62
render_to_pipe() (*aiocoap.resource.ObservableResource method*), 73
render_to_pipe() (*aiocoap.resource.Resource method*), 73
render_to_pipe() (*aiocoap.resource.Site method*), 75
RenderableError, 42
ReplayError, 81
ReplayErrorWithEcho, 81
ReplayWindow (*class in aiocoap.oscore*), 86
Request (*class in aiocoap.interfaces*), 41
Request (*class in aiocoap.protocol*), 33
request() (*aiocoap.Context method*), 29
request() (*aiocoap.interfaces.RequestInterface method*), 41
request() (*aiocoap.interfaces.RequestProvider method*), 41
request() (*aiocoap.protocol.Context method*), 32
request() (*aiocoap.proxy.client.ProxyForwarder method*), 61
request() (*aiocoap.transports.oscore.TransportOSCORE method*), 50
REQUEST_ENTITY_INCOMPLETE (*aiocoap.Code attribute*), 22
REQUEST_ENTITY_INCOMPLETE (*aiocoap.numbers.codes.Code attribute*), 65
REQUEST_ENTITY_TOO_LARGE (*aiocoap.Code attribute*), 22
REQUEST_ENTITY_TOO_LARGE (*aiocoap.numbers.codes.Code attribute*), 65
REQUEST_HASH (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
REQUEST_HASH (*aiocoap.OptionNumber attribute*), 24
request_hash (*aiocoap.options.Options attribute*), 38
REQUEST_TAG (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
REQUEST_TAG (*aiocoap.OptionNumber attribute*), 24
request_tag (*aiocoap.options.Options attribute*), 38
REQUEST_TIMEOUT (*in module aiocoap.numbers.constants*), 67
requested_hostinfo (*aiocoap.Message attribute*), 28
requested_hostinfo (*aiocoap.message.Message attribute*), 36
requested_path (*aiocoap.Message attribute*), 28
requested_path (*aiocoap.message.Message attribute*), 36
requested_proxy_uri (*aiocoap.Message attribute*), 28
requested_proxy_uri (*aiocoap.message.Message attribute*), 36
requested_query (*aiocoap.Message attribute*), 28
requested_query (*aiocoap.message.Message attribute*), 36
requested_scheme (*aiocoap.Message attribute*), 28
requested_scheme (*aiocoap.message.Message attribute*), 36
RequestIdentifiers (*class in aiocoap.oscore*), 82
RequestInterface (*class in aiocoap.interfaces*), 41
RequestProvider (*class in aiocoap.interfaces*), 41
RequestTimedOut, 44
ResolutionError, 44
Resource (*class in aiocoap.interfaces*), 41
Resource (*class in aiocoap.resource*), 72
ResourceChanged, 44
response (*aiocoap.interfaces.Request attribute*), 41
response_nonraising (*aiocoap.protocol.BaseUnicastRequest attribute*), 33
response_raising (*aiocoap.protocol.BaseUnicastRequest attribute*), 33
responses_send_kid (*aiocoap.oscore.CanProtect attribute*), 85
responses_send_kid (*aiocoap.oscore.GroupContext attribute*), 88
ResponseWrappingError, 42
resume_writing() (*aiocoap.transports.tcp.TcpConnection method*), 53
ReverseProxy (*class in aiocoap.proxy.server*), 63
ReverseProxyWithPooledObservations (*class in aiocoap.proxy.server*), 63
RFC8323Remote (*class in aiocoap.transports.rfc8323common*), 51
RST (*aiocoap.numbers.types.Type attribute*), 70
RST (*aiocoap.Type attribute*), 21
run_driving_pipe() (*in module aiocoap.pipe*), 47

S

scheme (*aiocoap.interfaces.EndpointAddress attribute*), 40
scheme (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50
scheme (*aiocoap.transports.tcp.TcpConnection attribute*), 52
scheme (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55
scheme (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
scheme (*aiocoap.transports.ws.PoolKey attribute*), 60
scheme (*aiocoap.transports.ws.WSRemote attribute*), 60
SecurityContextUtils (*class in aiocoap.oscore*), 86

SecurityStore (*class in aiocoap.transports.tinydtls_server*), 56
send() (*aiocoap.interfaces.MessageInterface method*), 38
send() (*aiocoap.transports.generic_udp.GenericMessageInterface method*), 49
send() (*aiocoap.transports.tinydtls.DTLSClientConnection method*), 55
send() (*aiocoap.transports.tinydtls.MessageInterfaceTinyDTLS method*), 55
send() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
send_message() (*aiocoap.interfaces.TokenInterface method*), 40
send_message() (*aiocoap.transports.ws.WSPool method*), 61
sendmsg() (*aiocoap.util.asyncio.recvmsg.RecvmsgSelectorDatagramTransport method*), 76
SENML (*aiocoap.ContentFormat attribute*), 25
SENML (*aiocoap.numbers.contentformat.ContentFormat attribute*), 68
Sentinel (*class in aiocoap.util*), 80
ServerObservation (*class in aiocoap.protocol*), 33
SERVICE_UNAVAILABLE (*aiocoap.Code attribute*), 22
SERVICE_UNAVAILABLE (*aiocoap.numbers.codes.Code attribute*), 65
set_request_uri() (*aiocoap.Message method*), 27
set_request_uri() (*aiocoap.message.Message method*), 36
shutdown() (*aiocoap.Context method*), 29, 30
shutdown() (*aiocoap.interfaces.MessageInterface method*), 39
shutdown() (*aiocoap.protocol.Context method*), 31, 32
shutdown() (*aiocoap.transports.generic_udp.GenericMessageInterface method*), 49
shutdown() (*aiocoap.transports.oscore.TransportOSCORE method*), 50
shutdown() (*aiocoap.transports.tcp.TCPClient method*), 54
shutdown() (*aiocoap.transports.tcp.TCPServer method*), 53
shutdown() (*aiocoap.transports.tinydtls.DTLSClientConnection method*), 55
shutdown() (*aiocoap.transports.tinydtls.MessageInterfaceTinyDTLS method*), 55
shutdown() (*aiocoap.transports.tinydtls_server.MessageInterfaceTinyDTLSServer method*), 56
shutdown() (*aiocoap.transports.udp6.MessageInterfaceUDP6 method*), 59
shutdown() (*aiocoap.transports.ws.WSPool method*), 61
SHUTDOWN_TIMEOUT (*in module aiocoap.numbers.constants*), 67
sign() (*aiocoap.oscore.AlgorithmCountersign method*), 84
sign() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85
sign() (*aiocoap.oscore.Ed25519 method*), 85
signature_length (*aiocoap.oscore.AlgorithmCountersign attribute*), 84
signature_length (*aiocoap.oscore.ECDSA_SHA256_P256 attribute*), 85
signature_length (*aiocoap.oscore.Ed25519 attribute*), 85
SimpleGroupContext (*class in aiocoap.oscore*), 88
Site (*class in aiocoap.resource*), 74
size (*aiocoap.optiontypes.BlockOption.BlockwiseTuple attribute*), 71
SIZE1 (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
SIZE1 (*aiocoap.OptionNumber attribute*), 24
size1 (*aiocoap.options.Options attribute*), 38
SIZE2 (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
SIZE2 (*aiocoap.OptionNumber attribute*), 24
size2 (*aiocoap.options.Options attribute*), 38
sk_to_curve25519() (*in module aiocoap.util.cryptography_additions*), 78
SockExtendedErr (*class in aiocoap.transports.udp6*), 59
start (*aiocoap.optiontypes.BlockOption.BlockwiseTuple attribute*), 71
staticstatic() (*aiocoap.oscore.AlgorithmCountersign method*), 84
staticstatic() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85
staticstatic() (*aiocoap.oscore.Ed25519 method*), 85
stop() (*aiocoap.util.cli.AsyncCLIDaemon method*), 77
strike_out() (*aiocoap.oscore.ReplayWindow method*), 87
StructuringOption (*class in aiocoap.optiontypes*), 70
sub_delims (*in module aiocoap.util.uri*), 79
SubdomainVirtualHost (*class in aiocoap.proxy.server*), 63
SubresourceVirtualHost (*class in aiocoap.proxy.server*), 64
sync_main() (*aiocoap.util.cli.AsyncCLIDaemon class method*), 77

T

tag_bytes (*aiocoap.oscore.A128GCM attribute*), 84
tag_bytes (*aiocoap.oscore.A192GCM attribute*), 84

tag_bytes (*aiocoap.oscore.A256GCM attribute*), 84
tag_bytes (*aiocoap.oscore.AES_CCM_16_128_128 attribute*), 83
tag_bytes (*aiocoap.oscore.AES_CCM_16_128_256 attribute*), 83
tag_bytes (*aiocoap.oscore.AES_CCM_16_64_128 attribute*), 82
tag_bytes (*aiocoap.oscore.AES_CCM_16_64_256 attribute*), 82
tag_bytes (*aiocoap.oscore.AES_CCM_64_128_128 attribute*), 83
tag_bytes (*aiocoap.oscore.AES_CCM_64_128_256 attribute*), 83
tag_bytes (*aiocoap.oscore.AES_CCM_64_64_128 attribute*), 82
tag_bytes (*aiocoap.oscore.AES_CCM_64_64_256 attribute*), 83
tag_bytes (*aiocoap.oscore.ChaCha20Poly1305 attribute*), 84
TCPClient (*class in aiocoap.transports.tcp*), 53
TcpConnection (*class in aiocoap.transports.tcp*), 52
TCPServer (*class in aiocoap.transports.tcp*), 53
TEXT (*aiocoap.ContentFormat attribute*), 25
TEXT (*aiocoap.numbers.contentformat.ContentFormat attribute*), 68
timeout (*aiocoap.util.asyncio.timeoutdict.TimeoutDict attribute*), 76
TimeoutDict (*class in aiocoap.util.asyncio.timeoutdict*), 76
TimeoutError, 44
TLSClient (*class in aiocoap.transports.tls*), 57
TLSServer (*class in aiocoap.transports.tls*), 57
to_message() (*aiocoap.error.ConstructionRenderableError method*), 43
to_message() (*aiocoap.error.RenderableError method*), 42
to_message() (*aiocoap.error.ResponseWrappingError method*), 42
to_message() (*aiocoap.oscore.ReplayErrorWithEcho method*), 81
TokenInterface (*class in aiocoap.interfaces*), 40
TokenManager (*class in aiocoap.interfaces*), 41
tokens (*aiocoap.util.linkformat_pygments.LinkFormatLexer attribute*), 78
TOO_MANY_REQUESTS (*aiocoap.Code attribute*), 22
TOO_MANY_REQUESTS (*aiocoap.numbers.codes.Code attribute*), 65
TransportOSCORE (*class in aiocoap.transports.oscore*), 50
trigger() (*aiocoap.protocol.ServerObservation method*), 34
type (*aiocoap.optiontypes.BlockOption attribute*), 71
type (*aiocoap.optiontypes.ContentFormatOption attribute*), 71
type (*aiocoap.optiontypes.TypedOption attribute*), 71
Type (*class in aiocoap*), 21
Type (*class in aiocoap.numbers.types*), 69
TypedOption (*class in aiocoap.optiontypes*), 71

U

UDP6EndpointAddress (*class in aiocoap.transports.udp6*), 57
UintOption (*class in aiocoap.optiontypes*), 70
UnallowedMethod, 43
Unauthorized, 43
UNAUTHORIZED (*aiocoap.Code attribute*), 22
UNAUTHORIZED (*aiocoap.numbers.codes.Code attribute*), 64
UnconditionalRedirector (*class in aiocoap.proxy.server*), 63
UnexpectedBlock1Option, 45
UnexpectedBlock2, 45
UnparsableMessage, 45
UNPROCESSABLE_ENTITY (*aiocoap.Code attribute*), 22
UNPROCESSABLE_ENTITY (*aiocoap.numbers.codes.Code attribute*), 65
unprotect() (*aiocoap.oscore.CanUnprotect method*), 86
unreserved (*in module aiocoap.util.uri*), 79
unresolved_remote (*aiocoap.Message attribute*), 28
unresolved_remote (*aiocoap.message.Message attribute*), 36
UNSUPPORTED_CONTENT_FORMAT (*aiocoap.Code attribute*), 22
UNSUPPORTED_CONTENT_FORMAT (*aiocoap.numbers.codes.Code attribute*), 65
UNSUPPORTED_MEDIA_TYPE (*aiocoap.Code attribute*), 22
UNSUPPORTED_MEDIA_TYPE (*aiocoap.numbers.codes.Code attribute*), 65
UnsupportedContentFormat, 43
UnsupportedMethod, 43
update_observation_count() (*aiocoap.resource.ObservableResource method*), 73
updated_state() (*aiocoap.resource.ObservableResource method*), 73
uri (*aiocoap.interfaces.EndpointAddress attribute*), 39
uri_base (*aiocoap.interfaces.EndpointAddress attribute*), 39
uri_base (*aiocoap.transports.oscore.OSCOREAddress attribute*), 49

uri_base (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
uri_base (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55
uri_base (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
uri_base_local (*aiocoap.interfaces.EndpointAddress attribute*), 39
uri_base_local (*aiocoap.transports.oscore.OSCOREAddress attribute*), 50
uri_base_local (*aiocoap.transports.rfc8323common.RFC8323Remote attribute*), 51
uri_base_local (*aiocoap.transports.tinydtls.DTLSClientConnection attribute*), 55
uri_base_local (*aiocoap.transports.udp6.UDP6EndpointAddress attribute*), 58
URI_HOST (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
URI_HOST (*aiocoap.OptionNumber attribute*), 23
uri_host (*aiocoap.options.Options attribute*), 37
URI_PATH (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
URI_PATH (*aiocoap.OptionNumber attribute*), 23
uri_path (*aiocoap.options.Options attribute*), 37
URI_PORT (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 68
URI_PORT (*aiocoap.OptionNumber attribute*), 23
uri_port (*aiocoap.options.Options attribute*), 37
URI_QUERY (*aiocoap.numbers.optionnumbers.OptionNumber attribute*), 69
URI_QUERY (*aiocoap.OptionNumber attribute*), 24
uri_query (*aiocoap.options.Options attribute*), 37

V

VALID (*aiocoap.Code attribute*), 22
VALID (*aiocoap.numbers.codes.Code attribute*), 64
value (*aiocoap.optiontypes.TypedOption attribute*), 71
value (*aiocoap.oscore.A128GCM attribute*), 84
value (*aiocoap.oscore.A192GCM attribute*), 84
value (*aiocoap.oscore.A256GCM attribute*), 84
value (*aiocoap.oscore.AES_CCM_16_128_128 attribute*), 83
value (*aiocoap.oscore.AES_CCM_16_128_256 attribute*), 83
value (*aiocoap.oscore.AES_CCM_16_64_128 attribute*), 82
value (*aiocoap.oscore.AES_CCM_16_64_256 attribute*), 82
value (*aiocoap.oscore.AES_CCM_64_128_128 attribute*), 83
value (*aiocoap.oscore.AES_CCM_64_128_256 attribute*), 83
value (*aiocoap.oscore.AES_CCM_64_64_128 attribute*), 82
value (*aiocoap.oscore.AES_CCM_64_64_256 attribute*), 83
value (*aiocoap.oscore.ChaCha20Poly1305 attribute*), 84
value_all_par (*aiocoap.oscore.ECDSA_SHA256_P256 attribute*), 85
value_all_par (*aiocoap.oscore.Ed25519 attribute*), 85
verify() (*aiocoap.oscore.AlgorithmCountersign method*), 84
verify() (*aiocoap.oscore.ECDSA_SHA256_P256 method*), 85
verify() (*aiocoap.oscore.Ed25519 method*), 85
verify_start() (*in module aiocoap.oscore*), 89
version (*in module aiocoap.meta*), 81

W

WaitingForClientTimedOut, 44
WKCResource (*class in aiocoap.resource*), 74
ws_missing_modules() (*in module aiocoap.defaults*), 48
WSPool (*class in aiocoap.transports.ws*), 61
WSRemote (*class in aiocoap.transports.ws*), 60