aiocoap

Release 0.4b1

Sep 12, 2019

Contents

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): missing are a caching and cross proxy implementation, proper multicast (support is incomplete); DTLS support is client-side only so far, and lacking some security properties.
- RFC7641 (Observe): Reordering, re-registration, and active cancellation are missing.
- RFC7959 (Blockwise): Multicast exceptions missing.
- RFC8323 (TCP): Supports CoAP over TCP and TLS (certificate only, no preshared or raw public keys) but not CoAP over WebSockets.
- RFC7967 (No-Response): Supported.
- RFC8132 (PATCH/FETCH): Types and codes known, FETCH observation supported
- draft-ietf-core-resource-directory: 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, formerly OSCOAP): Full support client-side (except handling the Echo option); protected servers can be implemented based on it but are not automatic yet.

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 3.5.2 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 *udp6* module is most full-featured, but ties into the default asyncio loop and requires full POSIX network interfaces only available on Linux and possibly some BSDs. On Windows and macOS, more constrained *server* and *client* transports with some caveats of their own are used; for more details, see the currently open platform issues. Alternative main loops like uvloop or gbulb can be used without restriction.

If your library depends on aiocoap, it should pick the required extras (as per *Installing aiocoap*) and declare a dependency like `aiocoap[linkheader,oscore] >= 0.4a1`.

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 `./setup.py test`; 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

In most situations, it is recommended to install the latest released version of aiocoap. If you do not use a distribution that has aiocoap packaged, or if you use Python's virtual environments, this is done with

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

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.

In those cases, it is 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.
- `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
</time>; obs, </.well-known/core>; ct=40, </other/separate>, </other/block>
```

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 two more at `/other/separate` and `/other/block`.

Note: Getting "5.00 Internal Server Error" instead? Install the link_header module and restart the server, or trust me that the output would look like that if it were installed and proceed.

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
2016-12-07 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)>
```

(continues on next page)

(continued from previous page)

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 has whitespace 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. Because a Future is returned, 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

The interactive Python shell does not work in an asynchronous fashion (yet?) – it follows a strict "read, evaluate, print" loop (REPL), similar to how a Python program as a whole is executed. To launch asynchronous processing, we'll use the following shorthand:

```
>>> import asyncio
>>> run = asyncio.get_event_loop().run_until_complete
```

With that, we can run asynchronous functions; note that any function that `awaits` anything is itself asynchronous and has to be declared accordingly. Now we can run what did not work before:

```
>>> 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)
>>> run(main())
<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>
```

(continues on next page)

(continued from previous page)

That's better!

(Now the `protocol` object could also be created. That doesn't actually take a long time, but could, depending on the operating system).

6.2.4 The response

To dissect the response, let's make sure we have it available:

```
>>> protocol = run(Context.create_client_context())
>>> msg = Message(code=GET, uri="coap://localhost/other/separate")
>>> response = run(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>
```

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]'
>>> 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` and is numeric in CoAP).

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 %s: %r"%(response.remote.hostinfo,
↪ 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 The aiocoap API

6.3.1 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 or accedd via modindex.

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.

6.3.2 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.

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

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

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*)

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.

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 may take the time to inform communications partners of stopped observations (but currently does not).

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.

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).

render_to_plumbing_request(*plumbing_request*)
Satisfy a plumbing request from the full `render()` / `needs_blockwise_assembly()` / `add_observation()` interfaces provided by the site.

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

classmethod create_client_context(*, *loggername='coap'*, *loop=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*)
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.

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 may take the time to inform communications partners of stopped observations (but currently does not).

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**(*plumbing_request*, *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.

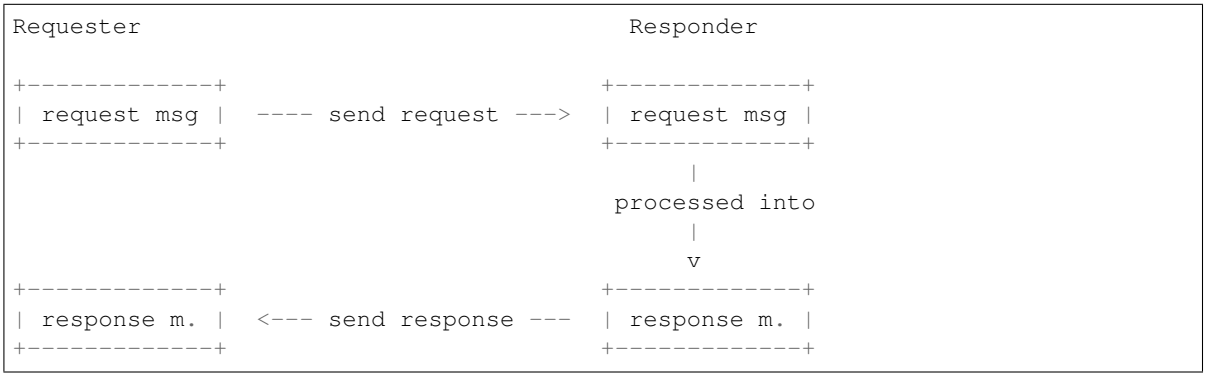
  While a message has not been transmitted, the property is managed by the *`Message`* itself using the *`set_request_uri()`* or the constructor *uri* argument.
- `request`: The request to which an incoming response message belongs; only available at the client. Managed by the *`interfaces.RequestProvider`*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

aiocoap.interfaces module

This module provides interface base classes to various aiocoap services, especially with respect to request and response handling.

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 = 1024
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. Must be divisible by 1024.

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

This interface is likely to change soon to something that is not limited to errno-style errors, and might allow transporting additional data.

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

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

send_message(*message*) → 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.

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: PlumbingRequest*)

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.

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.

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.

aiocoap.error module

Exception definitions for txThings CoAP 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 Media Type">

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

code = <Response Code 129 "4.01 Unauthorized">

aiocoap.error.**UnsupportedMediaType**
alias of *aiocoap.error.UnsupportedContentFormat*

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

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

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

Raised when request is timed out.

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

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.

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

The communication process has been aborted by request of the application.

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

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.

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

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*)
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*)
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.**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.**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**(*ctx: 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".

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.

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

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.

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.)

is_multicast = False

maximum_payload_size = 1024

maximum_block_size_exp = 6

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.

fill_or_recognize_remote(*message*)

shutdown()

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).

class aiocoap.transports.simple6.**MessageInterfaceSimple6**(*ctx: 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.
- This transport is experimental and likely to change.

class aiocoap.transports.simplesocketserver.**MessageInterfaceSimpleServer**(*ctx: 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*, *, *hostinfo=None*)

Bases: asyncio.protocols.Protocol, *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.)

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

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.

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.

is_multicast = False

is_multicast_locally = False

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.

maximum_block_size_exp
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

maximum_payload_size
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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*)

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.

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.

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.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 a plethorea of 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_RECVERR` to receive ICMP errors even on sockets that are not connected, `IPV6_JOIN_GROUP` for multicast membership management, and `recvmsg` and `MSG_ERRQUEUE` to obtain the data configured with the above options.

There are, if at all, only little attempts 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.**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.

```
>>> interface = type("FakeMessageInterface", (), {})
>>> 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
False
>>> local.hostinfo
'127.0.0.1'
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> all_coap4.is_multicast
True
```

scheme = 'coap'

interface

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.

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*)

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.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.

class aiocoap.proxy.client.**ProxyRequest**(*proxy*, *app_request*, *exchange_monitor_factory=<function ProxyRequest.<lambda>>*)
Bases: *aiocoap.interfaces.Request*

class aiocoap.proxy.client.**ProxyClientObservation**
Bases: *aiocoap.protocol.ClientObservation*

real_observation = None

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

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**(*code*, *explanation*)
Bases: Exception

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

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*)

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

apply_redirection(*request*)

class aiocoap.proxy.server.**ReverseProxyWithPooledObservations**(*outgoing_context*, *logger=None*)
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*)
Bases: *aiocoap.proxy.server.Redirector*

apply_redirection(*request*)

class aiocoap.proxy.server.**UnconditionalRedirector**(*target*)
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.

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 Media Type">
UNSUPPORTED_MEDIA_TYPE = <Response Code 143 "4.15 Unsupported Media Type">
UNPROCESSABLE_ENTITY = <Response Code 150 "4.22 Unprocessable Entity">
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">
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.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">

URI_PATH = <OptionNumber 11 "URI_PATH">

CONTENT_FORMAT = <OptionNumber 12 "CONTENT_FORMAT">

MAX_AGE = <OptionNumber 14 "MAX_AGE">

URI_QUERY = <OptionNumber 15 "URI_QUERY">

ACCEPT = <OptionNumber 17 "ACCEPT">

LOCATION_QUERY = <OptionNumber 20 "LOCATION_QUERY">

BLOCK2 = <OptionNumber 23 "BLOCK2">

BLOCK1 = <OptionNumber 27 "BLOCK1">

SIZE2 = <OptionNumber 28 "SIZE2">

PROXY_URI = <OptionNumber 35 "PROXY_URI">

PROXY_SCHEME = <OptionNumber 39 "PROXY_SCHEME">

SIZE1 = <OptionNumber 60 "SIZE1">

NO_RESPONSE = <OptionNumber 258 "NO_RESPONSE">

OBJECT_SECURITY = <OptionNumber 9 "OBJECT_SECURITY">

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.**BlockOption**(*number*, *value=None*)
Bases: *aiocoap.optiontypes.OptionType*

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)
```

value

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.

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.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.

aiocoap.resource.**link_format_to_message**(*request*, *linkformat*, *default_ct=40*)

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-0.4b1'*)

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 64 504'

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.

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

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

`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.

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.util.asyncio module

Extensions to asyncio and workarounds around its shortcomings

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.

Typical application for this is running `MyClass.sync_main()` in the program's `if __name__ == "__main__":` section.

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.socknumbers module

This module contains numeric constants that would be expected in the socket module, but are not exposed there.

For some platforms (eg. python up to 3.5 on Linux), there is an IN module that exposes them; and they are gathered from there.

As a fallback, the numbers are hardcoded. Any hints on where to get them from 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.

aiocoap.util.secrets module

This is a subset of what the Python 3.6 secrets module gives, for compatibility with earlier Python versions and for as long as there is no published & widespread backported version of it

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.

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 = '0.4b1'**
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-0.4b1'**
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: ValueError

Raised when verification is attempted on a non-OSCORE message

exception aiocoap.oscore.**ProtectionInvalid**
Bases: 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

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.**SecurityContext**
Bases: object

is_unicast = True

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.

unprotect(*protected_message*, *request_id=None*)

new_sequence_number()

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.

class aiocoap.oscore.**ReplayWindow**
Bases: object

class aiocoap.oscore.**SimpleReplayWindow**(*seen=None*)
Bases: *aiocoap.oscore.ReplayWindow*

A ReplayWindow that keeps its seen sequence numbers in a sorted list; all entries of the list and all numbers smaller than the first entry are considered seen.

This is not very efficient, but easy to understand and to serialize.

```
>>> w = SimpleReplayWindow()
>>> w.strike_out(5)
>>> w.is_valid(3)
True
>>> w.is_valid(5)
False
>>> w.strike_out(0)
>>> print(w.seen)
[0, 5]
>>> w.strike_out(1)
>>> w.strike_out(2)
>>> print(w.seen)
[2, 5]
>>> w.is_valid(1)
False
```

window_count = 64

is_valid(*number*)

strike_out(*number*)

class aiocoap.oscore.**FilesystemSecurityContext**(*basedir*, *role*)
Bases: *aiocoap.oscore.SecurityContext*

Security context stored in a directory as distinct files containing containing

- Master secret, master salt, the sender IDs of the participants, and 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.

The static files are phrased in a way that allows using the same files for server and client; only by passing "client" or "server" as role parameter at load time, the IDs are are assigned to the context as sender or recipient ID. (The sequence number file is set up in a similar way in preparation for multicast operation; but is not yet usable from a directory shared between server and client; when multicast is actually explored, the sequence file might be renamed to contain the sender ID for shared use of a directory).

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.

exception LoadError
Bases: ValueError

Exception raised with a descriptive message when trying to load a faulty security context

classmethod generate(*basedir*)
Create a security context directory from default parameters and a random key; it is an error if that directory already exists.

No SecurityContext object is returned immediately, as it is expected that the generated context can't be used immediately but first needs to be copied to another party and then can be opened in either the sender or the recipient role.

aiocoap.oscore.**verify_start**(*message*)
Extract a sender ID and ID context (if present, otherwise None) from a message for the verifier to then pick a security context to actually verify the message.

Call this only requests; for responses, you'll have to know the security context anyway, and there is usually no information to be gained.

6.4 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.

6.4.1 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.get_event_loop().run_until_complete(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.get_event_loop().run_until_complete(main())
```

6.4.2 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)

```
56
57      def __init__(self):
58          super().__init__()
59
60          self.handle = None
61
62      def notify(self):
63          self.updated_state()
64          self.reschedule()
65
66      def reschedule(self):
67          self.handle = asyncio.get_event_loop().call_later(5, self.notify)
68
69      def update_observation_count(self, count):
70          if count and self.handle is None:
71              print("Starting the clock")
72              self.reschedule()
73          if count == 0 and self.handle:
74              print("Stopping the clock")
75              self.handle.cancel()
76              self.handle = None
77
78      async def render_get(self, request):
79          payload = datetime.datetime.now().\
80                  strftime("%Y-%m-%d %H:%M").encode('ascii')
81          return aiocoap.Message(payload=payload)
82
83  # logging setup
84
85  logging.basicConfig(level=logging.INFO)
86  logging.getLogger("coap-server").setLevel(logging.DEBUG)
87
88  def main():
89      # Resource tree creation
90      root = resource.Site()
91
92      root.add_resource(['.well-known', 'core'],
93              resource.WKCResource(root.get_resources_as_linkheader))
94      root.add_resource(['time'], TimeResource())
95      root.add_resource(['other', 'block'], BlockResource())
96      root.add_resource(['other', 'separate'], SeparateLargeResource())
97
98      asyncio.Task(aiocoap.Context.create_server_context(root))
99
100     asyncio.get_event_loop().run_forever()
101
102 if __name__ == "__main__":
103     main()
```

6.5 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.

6.5.1 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 HOST[:PORT]] [--payload X]
                      [--content-format MIME] [-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.

--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 to CBOR).

-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)

6.5.2 aiocoap-proxy

a plain CoAP proxy that can work both as forward and as reverse proxy

```
usage: aiocoap-proxy [-h] [--forward] [--reverse] [--bind BIND]
                     [--tls-server-certificate CRT] [--tls-server-key KEY]
                     [--proxy HOST[:PORT]] [--namebased NAME:DEST]
                     [--pathbased PATH:DEST] [--unconditional DEST]
```

mode

Required argument for setting the operation mode

--forward Run as forward proxy

--reverse Run as reverse proxy

details

Options that govern how requests go in and out

--bind Host and/or port to bind to (see –help-bind for details)

--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

--proxy Relay outgoing requests through yet another proxy

Rules

Sequence of forwarding rules that, if matched by a request, specify a forwarding destination

--namebased If Uri-Host matches 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

6.5.3 aiocoap-rd

A plain CoAP resource directory according to draft-ietf-core-resource-directory-14

Known Caveats:

- Nothing group related is implemented.
- Multiply given registration parameters are not handled.
- It is very permissive. Not only is no security implemented.

```
usage: aiocoap-rd [-h] [--bind BIND] [--tls-server-certificate CRT]
                  [--tls-server-key KEY]
```

Named Arguments

--bind Host and/or port to bind to (see –help-bind for details)

--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

6.5.4 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.

```
usage: aiocoap-fileserver [-h] [-v] [--register [RD-URI]] [--bind BIND]
                          [--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

--bind Host and/or port to bind to (see –help-bind for details)

--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.

6.5.5 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.

6.6 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.

6.6.1 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

  https://github.com/chrysn/aiocoap/issues/148 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

6.6.2 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

6.6.3 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

6.7 LICENSE

Python Module Index

a

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W