aiocoap

Release 0.3

Mar 30, 2017

Contents

The aiocoap package is a Python implementation of CoAP, the Constrained Application Protocol (RFC 7252, more info at http://coap.technology/).

It uses the Python 3's asynchronous I/O to facilitate concurrent operations while maintaining a simple to use interface and not depending on anything outside the standard library.

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. All the details are in the `aiocoap module` 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), and DTLS.
- RFC7641 (Observe): Reordering, re-registration, and active cancellation are missing.
- RFC7959 (Blockwise): Multicast exceptions missing.
- draft-ietf-core-etch-04: Only registry entries added, but that should be all that's neede on the library side.
- draft-ietf-core-resource-directory-10: A standalone resource directory server is provided along with a library function to register at one. They lack support for groups, PATCHes to endpoint locations and security considerations, and are generally rather simplistic.
- draft-ietf-core-object-security-02 (OSCOAP): Infrastructure for supporting it is in place (lacking observe and inner-blockwise support), but no simple way exists yet for launching protected servers or requests 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 3.4 or greater.

The *Usage Examples* require Python 3.5 as they use newer syntax.

Some components (eg. servers that should auto-generate `.well-known/core` resources, OSCOAP) require additional packages to be present (eg. the link_header module or Python 3.6's backported secrets module); those are reflected in "extras" dependencies, see `setup.py` for details. Python modules that require all features should declare a dependency on `aiocoap[all]`.

CHAPTER 4

Development

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

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

Bugs (ranging from "design goal" and "wishlist" to typos) are currently tracked in the github issue tracker.

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]`.

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
```

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
```

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.

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.

First, some tools

Before we get into programming, let's establish tools with which we can probe a server, and a server itself.

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

```
$ ./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/others/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.

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: None GET (ID None, token b'') remote None, 2␣
↪option(s)>
```

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

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

You might have noticed that the Uri-Path option 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 (ID 51187, token b
↪'\x00\x00\x81\x99') 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.

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 (ID 51187, token b
↪'\x00\x00\x81\x99') remote <UDP6EndpointAddress [::ffff:127.0.0.1]:5683 with local␣
↪address>, 186 byte(s) payload>
```

That's better!

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

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 (ID 51187, token b
↪'\x00\x00\x81\x99') 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 {}: {}".format(response.get_request_uri(),
↪ response.payload))
>>> run(main())
Response from coap://localhost/time: b'2016-12-07 18:16'
Response from coap://vs0.inf.ethz.ch/obs: b'18:16:11'
Response from coap://coap.me/test: b'welcome to the ETSI plugtest! last␣
↪change: 2016-12-06 16:02:33 UTC'
```

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

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

aiocoap module

aiocoap

The aiocoap package is a library that implements CoAP, the Constrained Application Protocol (RFC 7252, more info at http://coap.technology/).

Usage

In all but the most exotic applications, you will want to create a single *Context* instance that binds to the network. The *Context.create_client_context()* and *Context.create_server_context()* coroutines give you a readily connected context.

On the client side, you can request resources by assembling a *Message* and passing it to your context's *Context.request()* method, which returns a *protocol.Request* object with a protocol.Request.response future (which is a *Message* again).

On the server side, a resource tree gets built from *aiocoap.resource.Resource* objects into a *aiocoap.resource.Site*, which is assigned to the context at creation time.

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'*)
Bases: asyncio.protocols.DatagramProtocol, *aiocoap.interfaces.RequestProvider*

An object that passes messages between an application and the network

A *Context* gets bound to a network interface as an asyncio protocol. It manages the basic CoAP network mechanisms like message deduplication and retransmissions, and delegates management of blockwise transfer as well as the details of matching requests with responses to the *Request* and *Responder* classes.

In that respect, a Context (as currently implemented) is also an endpoint. It is anticipated, though, that issues arise due to which the implementation won't get away with creating a single socket, and that it will be required to deal with multiple endpoints. (E.g. the V6ONLY=0 option is not portable to some OS, and implementations might need to bind to different ports on different interfaces in multicast contexts). When those distinctions will be implemented, message dispatch will stay with the context, which will then deal with the individual endpoints.

In a way, a *Context* is the single object all CoAP messages that get treated by a single application pass by.

Context creation and destruction

Instead of passing a protocol factory to the asyncio loop's create_datagram_endpoint method, the following convenience functions are recommended for creating a context:

classmethod create_client_context(*, *dump_to=None*, *loggername='coap'*, *loop=None*)
Create a context bound to all addresses on a random listening port.

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

classmethod create_server_context(*site*, *bind=('::'*, *5683)*, *, *dump_to=None*, *loggername='coap-server'*, *loop=None*)
Create an 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.

If you choose to create the context manually, make sure to wait for its ready future to complete, as only then can messages be sent.

shutdown()
Take down the listening socket 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

A context's public API consists of the *send_message()* function, the *outgoing_requests*, *incoming_requests* and `outgoing_obvservations` dictionaries, and the `serversite` object, but those are not stabilized yet, and for most applications the following convenience functions are more suitable:

request(*request*, ***kwargs*)
TODO: create a proper interface to implement and deprecate direct instanciation again

multicast_request(*request*)

If more control is needed, eg. with observations, 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).

outgoing_requests = None
Unfinished outgoing requests (identified by token and remote)

incoming_requests = None
Unfinished incoming requests. `(path-tuple, remote): Request`

outgoing_observations = None
Observations where this context acts as client. `(token, remote) -> weak(ClientObservation)`

incoming_observations = None
Observation where this context acts as server. `(token, remote) -> ServerObservation`. This is managed by :cls:ServerObservation and *Responder.handle_observe_request()*.

shutdown()
Take down the listening socket 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).

fill_remote(*message*)

send_message(*message*, *exchange_monitor=None*)
Encode and send message. This takes care of retransmissions (if CON), message IDs and rate limiting, but does not hook any events to responses. (Use the *Request* class or responding resources instead; those are the typical callers of this function.)

If notification about the progress of the exchange is required, an ExchangeMonitor can be passed in, which will receive the appropriate callbacks.

next_token()
Reserve and return a new Token for request.

request(*request*, ***kwargs*)
TODO: create a proper interface to implement and deprecate direct instanciation again

multicast_request(*request*)

classmethod create_client_context(***, *dump_to=None*, *loggername='coap'*, *loop=None*)
Create a context bound to all addresses on a random listening port.

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

classmethod create_server_context(*site*, *bind=('::'*, *5683)*, ***, *dump_to=None*, *loggername='coap-server'*, *loop=None*)
Create an 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.

kill_transactions(*remote*, *exception=<class 'aiocoap.error.CommunicationKilled'>*)
Abort all pending exchanges and observations to a given remote.

The exact semantics of this are not yet completely frozen – currently, pending exchanges are treated as if they timeouted, server sides of observations are droppedn and client sides of observations receive an errback.

Requests that are not part of an exchange, eg. NON requests or requests that are waiting for their responses after an empty ACK are currently not handled.

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

Common mechanisms of *Request* and *MulticastRequest*

class aiocoap.protocol.**Request**(*protocol*, *app_request*, *exchange_monitor_factory=<function Request.<lambda>>*, *handle_blockwise=True*)
Bases: *aiocoap.protocol.BaseRequest*, *aiocoap.interfaces.Request*

Class used to handle single outgoing request.

Class includes methods that handle sending outgoing blockwise requests and receiving incoming blockwise responses.

cancel()

send_request(*request*)
Send a request or single request block.

This method is used in 3 situations: - sending non-blockwise request - sending blockwise (Block1) request block - asking server to send blockwise (Block2) response block

handle_response(*response*)

process_block1_in_response(*response*)
Process incoming response with regard to Block1 option.

process_block2_in_response(*response*)
Process incoming response with regard to Block2 option.

handle_final_response(*response*)

register_observation(*response*)

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.

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

Experimental Interface.

class aiocoap.protocol.**MulticastRequest**(*protocol*, *request*)
Bases: *aiocoap.protocol.BaseRequest*

handle_response(*response*)

class aiocoap.protocol.**Responder**(*protocol*, *request*, *exchange_monitor_factory=<function Responder.<lambda>>*)
Bases: object

Handler for an incoming request or (in blockwise) a group thereof

Class includes methods that handle receiving incoming blockwise requests (only atomic operation on complete requests), searching for target resources, preparing responses and sending outgoing blockwise responses.

To keep an eye on exchanges going on, a factory for ExchangeMonitor can be passed in that generates a monitor for every single message exchange created during the response.

handle_next_request(*request*)

process_block1_in_request(*request*)
Process an incoming request while in block1 phase.

This method is responsible for finishing the app_request future and thus indicating that it should not be called any more, or scheduling itself again.

dispatch_request(*initial_block*)
Dispatch incoming request - search context resource tree for resource in Uri Path and call proper CoAP Method on it.

respond_with_error(*request*, *code*, *payload*)
Helper method to send error response to client.

respond(*app_response*, *request*)
Take application-supplied response and prepare it for sending.

process_block2_in_request(*request*)
Process incoming request with regard to Block2 option

Method is recursive - calls itself until all response blocks are sent to client.

send_non_final_response(*response*, *request*)
Helper method to send a response to client, and setup a timeout for client. This also registers the responder with the protocol again to receive the next message.

send_final_response(*response*, *request*)

send_response(*response*, *request*)
Send a response or single response block.

This method is used in 4 situations: - sending success non-blockwise response - asking client to send blockwise (Block1) request block - sending blockwise (Block2) response block - sending any error response

send_empty_ack(*request*)
Send separate empty ACK when response preparation takes too long.

Currently, this can happen only once per Responder, that is, when the last block1 has been transferred and the first block2 is not ready yet.

handle_observe_request(*request*)

handle_observe_response(*request*, *response*)
Modify the response according to the Responder's understanding of the involved observation (eg. drop the observe flag it's not involved in an observation or the observation was cancelled), and update the Responder/context if the response modifies the observation state (eg. by being unsuccessful).

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

Callback collection interface to keep track of what happens to an exchange.

Callbacks will be called in sequence: `enqueued{0,1} sent retransmitted{0, MAX_RETRANSMIT} (timeout | rst | cancelled | response)`; everything after `sent` only gets called if the messae that initiated the exchange was a CON.

enqueued()

sent()

retransmitted()

timeout()

rst()

cancelled()

response(*message*)

class aiocoap.protocol.**ServerObservation**(*original_protocol*, *original_request*, *requester_log*)
Bases: object

An active CoAP observation inside a server is described as a ServerObservation object.

It keeps a complete copy of the original request for simplicity (while it actually would only need parts of that request, like the accept option).

A ServerObservation has two boolean states: accepted and cancelled. It is originally neither, gets accepted when a *ObservableResource.add_observation()* method does *accept()* it, and gets cancelled by incoming packages of the same identifier, RST/timeout on notifications or the observed resource. Beware that an accept can happen after cancellation if the client changes his mind quickly, but the resource takes time to decide whether it can be observed.

accept(*cancellation_callback*)

deregister(*reason*)

identifier

static request_key(*request*)

trigger(*response=None*)

class ObservationExchangeMonitor(*observation*)
Bases: *aiocoap.protocol.ExchangeMonitor*

These objects feed information about the success or failure of a response back to the observation.

Note that no information flows to the exchange monitor from the observation, so they may outlive the observation and need to check if it's not already cancelled before cancelling it.

enqueued()

sent()

rst()

timeout()

class aiocoap.protocol.**ClientObservation**(*original_request*)

Bases: 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.

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. (If you choose to set this explicitly set this, make sure not to set incomplete IPv6 address tuples, as they can be sent but don't compare equally with the responses). Non-roundtrippable.
- 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 .opt.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.)

- `prepath`, `postpath`: Not sure, will probably go away when resources are overhauled. Non-roundtrippable.

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

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).

```
>>> 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
>>> ignore = [OptionNumber.ETAG]
>>> m1.get_cache_key(ignore) == m2.get_cache_key(ignore)
True
```

get_request_uri ()
The absolute URI this message belongs to.

For requests, this is composed from the options (falling back to the remote). For responses, this is stored by the Request object not only to preserve the request information (which could have been kept by the requesting application), but also because the Request can know about multicast responses (which would update the host component) and redirects (FIXME do they exist?).

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.

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

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.

aiocoap.interfaces module

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

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

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

fill_remote(*message*)
Populate a message's remote property based on its .opt.uri_host or .unresolved_remote. This interface is likely to change.

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 with the response message.'

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.

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.

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).

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.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. (If, for example, a non-posix platform is added, it might be easier to rewrite the *udp6* for that platform instead of "ifdef hell").

aiocoap.transports.udp6 module

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

As this makes use of RFC 3542 options (IPV6_PKTINFO), this is likely to only work with IPv6 interfaces. Hybrid stacks are supported, though, so V4MAPPED addresses (a la *::ffff:127.0.0.1*) will be used when name resolution shows that a name is only available on V4.

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

hostinfo

uri

port

is_multicast

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

classmethod load(*data*)

class aiocoap.transports.udp6.**TransportEndpointUDP6**(*new_message_callback*, *new_error_callback*, *log*, *loop*)

Bases: *aiocoap.util.asyncio.RecvmsgDatagramProtocol*, *aiocoap.interfaces.TransportEndpoint*

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

classmethod create_client_transport_endpoint(*new_message_callback*, *new_error_callback*, *log*, *loop*, *dump_to*)

classmethod create_server_transport_endpoint(*new_message_callback*, *new_error_callback*, *log*, *loop*, *dump_to*, *bind*)

shutdown()

send(*message*)

fill_remote(*request*)

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*)

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

connection_lost(*exc*)

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*)

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

Bases: *aiocoap.interfaces.Request*

class aiocoap.proxy.client.**ProxyClientObservation**(*original_request*)
Bases: *aiocoap.protocol.ClientObservation*

real_observation = None

cancel()

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*)

render(*request*)

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*)

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*)

aiocoap.proxy.server.**splitport**(*hostport*)
Like urllib.parse.splitport, but return port as int, and as None if it equals the CoAP default port. Also, it allows giving IPv6 addresses like a netloc:

```
>>> splitport('foo')
('foo', None)
>>> splitport('foo:5683')
('foo', None)
>>> splitport('[::1]:56830')
('::1', 56830)
```

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 Content Format”>

UNSUPPORTED_MEDIA_TYPE = <Response Code 143 “4.15 Unsupported Content Format”>

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”>

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_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.

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.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">

OBJECT_SECURITY = <OptionNumber 65025 "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.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()

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

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

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

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

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

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

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

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

aiocoap.error.**UnsupportedMediaType**
alias of *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">

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

length
Indicate the length of the encoded value

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()

decode(*rawdata*)

length

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()

decode(*rawdata*)

length

class aiocoap.optiontypes.**UintOption**(*number*, *value=0*)
Bases: *aiocoap.optiontypes.OptionType*

Uint CoAP option - used to represent integer options.

encode()

decode(*rawdata*)

length

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.

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.

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)
```

BlockOption.**value**

BlockOption.**encode**()

BlockOption.**decode**(*rawdata*)

BlockOption.**length**

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. 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
<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, and responds appropriately to unsupported methods.

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*)

render(*request*)

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

add_observation(*request*, *serverobservation*)

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()

class aiocoap.resource.**WKCResource**(*listgenerator*)
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.

ct = 40

render_get(*request*)

class aiocoap.resource.**PathCapable**
Bases: object

Class that indicates that a resource promises to parse the uri_path option, and can thus be given requests for *render()*ing that contain a uri_path

class aiocoap.resource.**Site**
Bases: *aiocoap.interfaces.ObservableResource*, *aiocoap.resource.PathCapable*

Typical root element that gets passed to a `Context` and contains all the resources that can be found when the endpoint gets accessed as a server.

This provides easy registration of statical resources. Add resources at absolute locations using the *add_resource()* method.

For example, the site at

```
>>> site = Site()
>>> site.add_resource(["hello"], Resource())
```

will have requests to </hello> rendered by the new resource.

You can add another Site (or another instance of *PathCapable*) as well, those will be nested and integrally reported in a WKCResource. The path of a site should not end with an empty string (ie. a slash in the URI) – the child site's own root resource will then have the trailing slash address. Subsites can not have link-header attributes on their own (eg. *rt*) and will never respond to a request that does not at least contain a single slash after the the given path part.

For example,

```
>>> batch = Site()
>>> batch.add_resource(["light1"], Resource())
>>> batch.add_resource(["light2"], Resource())
>>> batch.add_resource([], Resource())
>>> s = Site()
>>> s.add_resource("batch", batch)
```

will have the three created resources rendered at </batch/light1>, </batch/light2> and </batch/>.

If it is necessary to respond to requests to </batch> or report its attributes in .well-known/core in addition to the above, a non-PathCapable resource can be added with the same path. This is usually considered an odd design, not fully supported, and for example doesn't support removal of resources from the site.

needs_blockwise_assembly(*request*)

render(*request*)

add_observation(*request*, *serverobservation*)

add_resource(*path*, *resource*)

remove_resource(*path*)

get_resources_as_linkheader()

aiocoap.dump module

class aiocoap.dump.**TextDumper**(*outfile*, *protocol=None*)

Bases: *aiocoap.util.asyncio.RecvmsgDatagramProtocol*

Plain text network data dumper

A TextDumper can be used to log network traffic into a file that can be converted to a PCAP-NG file as described in its header.

Currently, this discards information like addresses; it is unknown how that information can be transferred into a dump reader easily while simultaneously staying at application level and staying ignorant of particular underlying protocols' data structures.

It could previously be used stand-alone (outside of the asyncio transport/protocol mechanisms) when instanciated only with an output file (the `datagram_received()` and `sendto()` were used), but with the *`datagram_msg_received()`* substitute method, this is probably impractical now.

To use it between an asyncio transport and protocol, use the :meth:endpointfactory method.

classmethod endpointfactory(*outfile*, *actual_protocol*)
This method returns a function suitable for passing to an asyncio loop's .create_datagram_endpoint method. It will place the TextDumper between the object and the transport, transparently dumping network traffic and passing it on together with other methods defined in the protocol/transport interface.

If you need the actual protocol after generating the endpoint (which when using this method returns a TextDumper instead of an actual_protocol), you can access it using the protocol property.

protocol

datagram_msg_received(*data*, *ancdata*, *flags*, *address*)

sendmsg(*data*, *ancdata*, *flags*, *address*)

connection_made(*transport*)

close()

connection_lost(*exc*)

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 Python3.4'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.asyncio module

Extensions to asyncio and workarounds around its shortcomings

aiocoap.util.asyncio.**cancel_thoroughly**(*handle*)
Use this on a (Timer)Handle when you would .cancel() it, just also drop the callback and arguments for them to be freed soon.

class aiocoap.util.asyncio.**RecvmsgDatagramProtocol**
Bases: asyncio.protocols.DatagramProtocol

Inheriting from this indicates that the instance expects to be called back datagram_msg_received instead of datagram_received

class aiocoap.util.asyncio.**RecvmsgSelectorDatagramTransport**(**args*, ***kwargs*)
Bases: asyncio.selector_events._SelectorDatagramTransport

sendmsg(*data*, *ancdata*, *flags*, *address*)

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.**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.

classmethod sync_main(**args*, ***kwargs*)
Run the application in an AsyncIO main loop, shutting down cleanly on keyboard interrupt.

aiocoap.util.queuewithend module

This is a relic from before the `__aiter__` protocol was established; it will be phased out before aiocoap 1.0 is released.

class aiocoap.util.queuewithend.**AsyncIterable**
Bases: object

can_peek()
Return True when a result is ready to be fetched with .get_nowait(), and False when no more items can be fetched.

get_nowait()
Fetch the next item. This must only be called once after can_peek has returned True.

class aiocoap.util.queuewithend.**QueueWithEnd**(*maxsize=0*)
Bases: *aiocoap.util.queuewithend.AsyncIterable*

A QueueWithEnd shares a Queue's behavior in that it gets fed with put and consumed with get_nowait. Contrary to a Queue, this is designed to be consumed only by one entity, which uses the coroutine can_peek to make sure the get_nowait will succeed.

Another difference between a Queue and a QueueWithEnd is that the latter can also terminate (which is indicated by can_peek returning False and set by the finish coroutine) and raise exceptions (which raise from the get_nowait function and are set by the put_exception coroutine).

Type
alias of *QueueWithEnd.Type*

can_peek()

get_nowait()

put(*value*)

put_exception(*value*)

finish()

classmethod cogenerator(*maxsize=0*)
Coroutine decorator that passes a callable *asyncyield* into the function as the first argument and returns a QueueWithEnd. It is implicitly finished when the coroutine returns.

```
>>> @QueueWithEnd.cogenerator()
>>> def count_slowly(asyncyield, count_to=count_to):
...     for i in range(count_to):
...         yield from asyncio.sleep(1)
...         yield from asyncyield(i + 1)
>>> counter = count_slowly(10)
>>> while (yield from counter.can_peek()):
...     i = counter.get_nowait()
...     print("Current count is %d"%i)
```

classmethod merge(*queues*)
Asyncio's *as_completed* does not work with QueueWithEnd objects for the same reason it can't replace it (missing end-of-loop indication); the *merge* classmethod can be used instead to fetch results indiscriminately from queues as they are completed:

```
>>> @QueueWithEnd.cogenerator()
>>> def count(asyncyield):
...     for i in range(3):
...         yield from asyncyield(i + 1)
...         yield from time.sleep(0.1 * i)
>>> firstcount = count()
>>> secondcount = count()
>>> merged = QueueWithEnd.merged([firstcount, secondcount])
>>> while (yield from merged.can_peek()):
...     print(merged.get_nowait())
1
2
1
2
3
3
```

more()

value

consume()

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.crypto module

This module contains cryptographic helpers for OSCOAP

The module should be abandoned as soon as the functions can be replaced with ones from established cryptographic library bindings.

`aiocoap.util.crypto.`**`encrypt_ccm`**(*plaintext*, *aad*, *key*, *iv*, *tag_length*)

exception `aiocoap.util.crypto.`**`InvalidAEAD`**
Bases: `Exception`

`aiocoap.util.crypto.`**`decrypt_ccm`**(*ciphertext*, *aad*, *tag*, *key*, *iv*)

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.secrets.`**`token_bytes`**(*nbytes*)

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.oscoap module

This module contains the tools to send OSCOAP secured messages.

(Work in progress.)

exception `aiocoap.oscoap.`**`NotAProtectedMessage`**
Bases: `ValueError`

Raised when verification is attempted on a non-OSCOAP message

exception `aiocoap.oscoap.`**`ProtectionInvalid`**
Bases: `ValueError`

Raised when verification of an OSCOAP message fails

class `aiocoap.oscoap.`**`Algorithm`**
Bases: `object`

encrypt(*plaintext*, *aad*, *key*, *iv*)
Return (ciphertext, tag) for given input data

decrypt(*ciphertext*, *tag*, *aad*, *key*, *iv*)
Reverse encryption. Must raise ProtectionInvalid on any error stemming from untrusted data.

class aiocoap.oscoap.**AES_CCM**
Bases: *aiocoap.oscoap.Algorithm*

classmethod encrypt(*plaintext*, *aad*, *key*, *iv*)

classmethod decrypt(*ciphertext*, *tag*, *aad*, *key*, *iv*)

max_seqno

class aiocoap.oscoap.**AES_CCM_64_64_128**
Bases: *aiocoap.oscoap.AES_CCM*

value = 12

key_bytes = 16

iv_bytes = 7

tag_bytes = 8

class aiocoap.oscoap.**SecurityContext**
Bases: object

protect(*message*, *request_partiv=None*)

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

new_sequence_number()

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

class aiocoap.oscoap.**SimpleReplayWindow**(*seen=None*)
Bases: *aiocoap.oscoap.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.oscoap.**FilesystemSecurityContext** (*basedir*, *role*)
Bases: *aiocoap.oscoap.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.

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.oscoap.**verify_start** (*message*)
Extract a CID 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 (and things would even fail completely in compressed messages).

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.

Unlike the library and its tools, these examples use the modern (Python 3.5 and later) `async` idiom instead of the original asyncio `yield from`. This is to align them better with what novice users are expected to learn when introduced to asynchronous programming in Python.

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():
    """
    Example class which performs single PUT request to localhost
    port 5683 (official IANA assigned CoAP port), URI "/other/block".
    Request is sent 2 seconds after initialization.

    Payload is bigger than 1kB, and thus is 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)
    request.opt.uri_host = '127.0.0.1'
    request.opt.uri_path = ("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())
```

Server

```
import datetime
import logging

import asyncio

import aiocoap.resource as resource
import aiocoap


class BlockResource(resource.Resource):
    """
    Example resource which supports GET and PUT methods. It sends large
    responses, which trigger blockwise transfer.
    """

    def __init__(self):
        super(BlockResource, self).__init__()
        self.content = ("This is the resource's default content. It is padded "\
                "with numbers to be large enough to trigger blockwise "\
                "transfer.\n" + "0123456789\n" * 100).encode("ascii")

    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.content = request.payload
        payload = ("I've accepted the new payload. You may inspect it here in "\
                "Python's repr format:\n\n%r"%self.content).encode('utf8')
        return aiocoap.Message(payload=payload)


class SeparateLargeResource(resource.Resource):
    """
    Example resource which supports GET method. It uses asyncio.sleep to
    simulate a long-running operation, and thus forces the protocol to send
    empty ACK first.
    """

    def __init__(self):
        super(SeparateLargeResource, self).__init__()
#         self.add_param(resource.LinkParam("title", "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.
```

```
    """
    def __init__(self):
        super(TimeResource, self).__init__()

        self.notify()

    def notify(self):
        self.updated_state()
        asyncio.get_event_loop().call_later(6, self.notify)

    def update_observation_count(self, count):
        if count:
            # not that it's actually implemented like that here -- unconditional␣
↪updating works just as well
            print("Keeping the clock nearby to trigger observations")
        else:
            print("Stowing away the clock until someone asks again")

    async def render_get(self, request):
        payload = datetime.datetime.now().strftime("%Y-%m-%d %H:%M").encode('ascii')
        return aiocoap.Message(payload=payload)

#class CoreResource(resource.Resource):
#    """
#    Example Resource that provides list of links hosted by a server.
#    Normally it should be hosted at /.well-known/core
#
#    Notice that self.visible is not set - that means that resource won't
#    be listed in the link format it hosts.
#    """
#
#    def __init__(self, root):
#        resource.Resource.__init__(self)
#        self.root = root
#
#    async def render_get(self, request):
#        data = []
#        self.root.generate_resource_list(data, "")
#        payload = ",".join(data).encode('utf-8')
#        return aiocoap.Message(payload=payload, content_format=40)

# logging setup

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

def main():
    # Resource tree creation
    root = resource.Site()

    root.add_resource(('.well-known', 'core'), resource.WKCResource(root.get_
↪resources_as_linkheader))

    root.add_resource(('time',), TimeResource())

    root.add_resource(('other', 'block'), BlockResource())

    root.add_resource(('other', 'separate'), SeparateLargeResource())
```

```
    asyncio.Task(aiocoap.Context.create_server_context(root))

    asyncio.get_event_loop().run_forever()

if __name__ == "__main__":
    main()
```

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.

aiocoap-client

aiocoap-client is a simple command-line tool for interacting with CoAP servers

```
usage: aiocoap-client [-h] [-m METHOD] [--observe] [--observe-exec CMD]
                      [--accept MIME] [--proxy HOST[:PORT]] [--payload X]
                      [--content-format MIME] [-v] [-q] [--dump FILE]
                      [--interactive]
                      url
```

Required Arguments

url	CoAP address to fetch

Optional Arguments

-m="GET", --method="GET"	Name or number of request method to use (default: "GET")
--observe=False	Register an observation on the resource
--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 payload in POST or PUT requests. If X starts with an '@', its remainder is treated as a file name and read from.
--content-format	Content format sent via POST or PUT
-v, --verbose	Increase the debug output
-q, --quiet	Decrease the debug output
--dump	Log network traffic to FILE
--interactive=False	Enter interactive mode

aiocoap-proxy

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

```
usage: aiocoap-proxy [-h] [--forward] [--reverse] [--server-address HOST]
                     [--server-port PORT] [--proxy HOST[:PORT]]
                     [--namebased NAME:DEST] [--pathbased PATH:DEST]
                     [--unconditional DEST]
```

mode

--forward Run as forward proxy

--reverse Run as reverse proxy

details

--server-address="::" Address to bind the server context to

--server-port=5683 Port to bind the server context to

--proxy Relay outgoing requests through yet another proxy

Rules

--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

aiocoap-rd

a plain CoAP resource directory according to draft-ietf-core-resource-directory-09

```
usage: aiocoap-rd [-h] [--server-address HOST] [--server-port PORT]
```

Optional Arguments

--server-address="::" Address to bind the server context to

--server-port=5683 Port to bind the server context to

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.

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.

There is currently onely one tool in there:

- `aiocoap-fileserver`: Serves the current directory's contents as CoAP resources, implementing directory listing and observation. No write support yet.

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.

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

LICENSE

Python Module Index

a

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W