aiorpcX Documentation

Release 0.22.1

Neil Booth

Apr 28, 2022

Contents

A generic asyncio library implementation of RPC suitable for an application that is a client, server or both.

The package includes a module with full coverage of JSON RPC versions 1.0 and 2.0, JSON RPC protocol auto-detection, and arbitrary message framing. It also comes with a SOCKS proxy client.

The current version is 0.22.1.

The library API is not stable and may change radically. These docs are out of date and will be updated when the API settles.

CHAPTER 1

Source Code

The project is hosted on GitHub. and uses Travis for Continuous Integration.

Python version at least 3.6 is required.

Please submit an issue on the bug tracker if you have found a bug or have a suggestion to improve the library.

CHAPTER 2

Authors and License

Neil Booth wrote the code, which is derived from the original JSON RPC code of ElectrumX.

The code is released under the MIT Licence.

CHAPTER 3

Documentation

3.1 ChangeLog

Note: The aiorpcX API changes regularly and is still unstable. I hope to finalize it for a 1.0 release in the coming months.

3.1.1 Version 0.22.1 (25 May 2021)

- release tasks as they complete in the task group; this might appear as a memory-leak for long-standing sessions

3.1.2 Version 0.22.0 (25 Apr 2021)

- join() waits for all cancelled tasks to finish, including daemonic ones

3.1.3 Version 0.21.1 (24 Apr 2021)

- handle peername of None in network code
- strip redundant whitespace from JSON (SomberNight)

3.1.4 Version 0.21.0 (11 Mar 2021)

- There have been significant semantic and API changes for TaskGroups. Their behaviour is now consistent, reliable and they have the same semantics as curio. As such I consider their API finalized and stable. In addition to the notes below for 0.20.x:
- closed() became the attribute joined.

- cancel_remaining() does not cancel daemonic tasks. As before it waits for the cancelled tasks to complete.
- On return from join() all tasks including deamonic ones have been cancelled, but nothing is waited for. If leaving a TaskGroup context because of an exception, cancel_remaining() - which can block - is called before join().

3.1.5 Version 0.20.2 (10 Mar 2021)

- result, exception, results and exceptions are now attributes. They raise a RuntimeError if called before a TaskGroup's join() operation has returned.

3.1.6 Version 0.20.1 (06 Mar 2021)

- this release contains some significant API changes which users will need to carefully check their code for.
- the report_crash argument to spawn() is removed; instead a new one is named daemon. A daemon task's exception (if any) is ignored by a TaskGroup.
- the join() method of TaskGroup (and so also when TaskGroup is used as a context manager) does not raise the exception of failed tasks. The full semantics are precisely described in the TaskGroup() docstring. Briefly: any task being cancelled or raising an exception causes join() to finish and all remaining tasks, including daemon tasks, to be cancelled. join() does not propagate task exceptions.
- the cancel_remaining() method of TaskGroup does not propagate any task exceptions
- TaskGroup supports the additional attributes 'tasks' and 'daemons'. Also, after join() has completed, result() returns the result (or raises the exception) of the first completed task. exception() returns the exception (if any) of the first completed task. results() returns the results of all tasks and exceptions() returns the exceptions raised by all tasks. daemon tasks are ignored.
- The above changes bring the implementation in line with curio proper and the semantic changes it made over a year ago, and ensure that join() behaves consistently when called more than once.

3.1.7 Version 0.18.4 (20 Nov 2019)

- handle time.time() not making progress. fixing #26 (SomberNight)
- handle SOCKSError in _connect_one (SomberNight)
- add SOCKSRandomAuth: Jeremy Rand

3.1.8 Version 0.18.3 (19 May 2019)

- minor bugfix release, fixing #22
- make JSON IDs independent across sessions, make websockets dependency optional (SomberNight)

3.1.9 Version 0.18.2 (19 May 2019)

- minor bugfix release

3.1.10 Version 0.18.1 (09 May 2019)

- convert incoming websocket text frames to binary. Convert outgoing messages to text frames if possible.

3.1.11 Version 0.18.0 (09 May 2019)

- Add *websocket* support as client and server by using Aymeric Augustin's excellent websockets package.

  Unfortunately this required changing several APIs. The code now distinguishes the previous TCP and SSL based-connections as *raw sockets* from the new websockets. The old Connector and Server classes are gone. Use *connect_rs()* and *serve_rs()* to connect a client and start a server for raw sockets; and *connect_ws()* and *serve_ws()* to do the same for websockets.

  SessionBase no longer inherits *asyncio.Protocol* as it is now transport-independent. Sessions no longer take a framer in their constructor: websocket messages are already framed, so instead a framer is passed to *connect_rs()* and *serve_rs()* if the default *NewlineFramer* is not wanted.

  A session is only instantiated when a connection handshake is completed, so *connection_made()* is no longer a method. *connection_lost()* and *abort()* are now coroutines; if overriding either be sure to call the base class implementation.

  is_send_buffer_full() was removed.

- Updated and added new examples
- JSON RPC message handling was made more efficient by using futures instead of events internally

3.1.12 Version 0.17.0 (22 Apr 2019)

- Add some new APIs, update others
- Add Service, NetAddress, ServicePart, validate_port, validate_protocol
- SessionBase: new API proxy() and remote_address(). Remove peer_address() and peer_address_str()
- SOCKSProxy: auto_detect_address(), auto_detect_host() renamed auto_detect_at_address() and auto_detect_at_host(). auto_detect_at_address() takes a NetAddress.

3.1.13 Version 0.16.2 (21 Apr 2019)

- fix force-close bug

3.1.14 Version 0.16.1 (20 Apr 2019)

- resolve socks proxy host using getaddrinfo. In particular, IPv6 is supported.
- add two new APIs

3.1.15 Version 0.16.0 (19 Apr 2019)

- session closing is now robust; it is safe to await session.close() from anywhere
- API change: FinalRPCError removed; raise ReplyAndDisconnect instead. This responds with a normal result, or an error, and then disconnects. e.g.:

```
raise ReplyAndDisconnect(23)
raise ReplyAndDisconnect(RPCError(1, "message"))
```

- the session base class' private method _close() is removed. Use await close() instead.
- workaround uvloop bug https://github.com/MagicStack/uvloop/issues/246

3.1.16 Version 0.15.0 (16 Apr 2019)

- error handling improved to include costing

3.1.17 Version 0.14.1 (16 Apr 2019)

- fix a bad assertion

3.1.18 Version 0.14.0 (15 Apr 2019)

- timeout handling improvements
- RPCSession: add log_me, send_request_timeout
- Concurrency: respect semaphore queue ordering
- cleaner protocol auto-detection

3.1.19 Version 0.13.6 (14 Apr 2019)

- RPCSession: concurrency control of outgoing requests to target a given response time
- SessionBase: processing_timeout will time-out processing of incoming requests. This helps prevent ever-growing request backlogs.
- SessionBase: add is_send_buffer_full()

3.1.20 Version 0.13.5 (13 Apr 2019)

- robustify concurrency handling

3.1.21 Version 0.13.3 (13 Apr 2019)

- export Concurrency class. Tweak some default constants.

3.1.22 Version 0.13.2 (12 Apr 2019)

- wait for task to complete on close. Concurrency improvements.

3.1.23 Version 0.13.0 (12 Apr 2019)

- fix concurrency handling; bump version as API changed

3.1.24 Version 0.12.1 (09 Apr 2019)

- improve concurrency handling; expose new API

3.1.25 Version 0.12.0 (09 Apr 2019)

- switch from bandwidth to a generic cost metric for sessions

3.1.26 Version 0.11.0 (06 Apr 2019)

- rename 'normalize_corofunc' to 'instantiate_coroutine'
- remove spawn() member of SessionBase
- add FinalRPCError (ghost43)
- more reliable cancellation on connection closing

3.1.27 Version 0.10.5 (16 Feb 2019)

- export 'normalize_corofunc'
- batches: fix handling of session loss; add test

3.1.28 Version 0.10.4 (07 Feb 2019)

- SessionBase: add closed_event, tweak closing process
- testsuite cleanup

3.1.29 Version 0.10.3 (07 Feb 2019)

- NewlineFramer: max_size of 0 does not limit buffering (SomberNight)
- trivial code / deprecation warning cleanups

3.1.30 Version 0.10.2 (29 Dec 2018)

- TaskGroup: faster cancellation (SomberNight)
- as for curio, remove wait argument to TaskGroup.join()
- setup.py: read the file to extract the version; see #10

3.1.31 Version 0.10.1 (07 Nov 2018)

- bugfixes for transport closing and session task spawning

3.1.32 Version 0.10.0 (05 Nov 2018)

- add session.spawn() method
- make various member variables private

3.1.33 Version 0.9.1 (04 Nov 2018)

- abort sessions which wait too long to send a message

3.1.34 Version 0.9.0 (25 Oct 2018)

- support of binary messaging and framing
- support of plain messaging protocols. Messages do not have an ID and do not expect a response; any response cannot reference the message causing it as it has no ID (e.g. the Bitcoin network protocol).
- removed the client / server session distinction. As a result there is now only a single session class for JSONRPC-style messaging, namely RPCSession, and a single session class for plain messaging protocols, MessageSession. Client connections are initiated by the session-independent Connector class.

3.1.35 Version 0.8.2 (25 Sep 2018)

- bw_limit defaults to 0 for ClientSession, bandwidth limiting is mainly intended for servers
- don't close proxy sockets on an exception during the initial SOCKS handshake; see #8. This works around an asyncio bug still present in Python 3.7
- make CodeMessageError hashable. This works around a Python bug fixed somewhere between Python 3.6.4 and 3.6.6

3.1.36 Version 0.8.1 (12 Sep 2018)

- remove report_crash arguments from TaskGroup methods
- ignore bandwidth limits if set <= 0

3.1.37 Version 0.8.0 (12 Sep 2018)

- change TaskGroup semantics: the first error of a member task is raised by the TaskGroup instead of TaskGroupError (which is now removed). Code wanting to query the status / results of member tasks should loop on group.next_done().

3.1.38 Version 0.7.3 (17 Aug 2018)

- fix #5; more tests added

3.1.39 Version 0.7.2 (16 Aug 2018)

- Restore batch functionality in Session class
- Less verbose logging
- Increment and test error count on protocol errors
- fix #4

3.1.40 Version 0.7.1 (09 Aug 2018)

- TaskGroup.cancel_remaining() must wait for the tasks to complete
- Fix some tests whose success / failure depended on time races
- fix #3

3.1.41 Version 0.7.0 (08 Aug 2018)

- Fix wait=object and cancellation
- Change Session and JSONRPCConnection APIs
- Fix a test that would hang on some systems

3.1.42 Version 0.6.2 (06 Aug 2018)

- Fix a couple of issues shown up by use in ElectrumX; add testcases

3.1.43 Version 0.6.0 (04 Aug 2018)

- Rework the API; docs are not yet updated
- New JSONRPCConnection object that manages the state of a connection, replacing the RPCProcessor class. It hides the concept of request IDs from higher layers; allowing simpler and more intuitive RPC datastructures
- The API now prefers async interfaces. In particular, request handlers must be async
- The API generally throws exceptions earlier for nonsense conditions
- TimeOut and TaskSet classes removed; use the superior curio primitives that 0.5.7 introduced instead
- SOCKS protocol implementation made i/o agnostic so the code can be used whatever your I/O framework (sync, async, threads etc). The Proxy class, like the session class, remains asyncio
- Testsuite cleaned up and shrunk, now works in Python 3.7 and also tests uvloop

3.1.44 Version 0.5.9 (29 Jul 2018)

- Remove “async” from __aiter__ which apparently breaks Python 3.7

3.1.45 Version 0.5.8 (28 Jul 2018)

- Fix __str__ in TaskGroupError

3.1.46 Version 0.5.7 (27 Jul 2018)

- Implement some handy abstractions from curio on top of asyncio

3.1.47 Version 0.5.6

- Define a ConnectionError exception, and set it on uncomplete requests when a connection is lost. Previously, those requests were cancelled, which does not give an informative error message.

3.2 Framing

Message *framing* is the method by which RPC messages are wrapped in a byte stream so that message boundaries can be determined.

`aiorpcx` provides an abstract base class for framers, and a single implementation: *`NewlineFramer`*. A framer must know how to take outgoing messages and frame them, and also how to break an incoming byte stream into message frames in order to extract the RPC messages from it.

class `aiorpcx.`**`FramerBase`**

Derive from this class to implement your own message framing methodology.

`frame`(*messages*)

Frame each message and return the concatenated result.

Parameters **`message`** – an iterable; each message should be of type `bytes` or `bytearray`

Returns the concatenated bytestream

Return type bytes

`messages`(*data*)

Parameters **`data`** – incoming data of type `bytes` or `bytearray`

Raises **`MemoryError`** – if the internal data buffer overflows

Note: since this may raise an exception, the caller should process messages as they are yielded. Converting the messages to a list will lose earlier ones if an exception is raised later.

class `aiorpcx.`**`NewlineFramer`**(*max_size=1000000*)

A framer where messages are delimited by an ASCII newline character in a text stream. The internal buffer for partial messages will hold up to *max_size* bytes.

3.3 JSON RPC

The `aiorpcx` module provides classes to interpret and construct JSON RPC protocol messages. Class instances are not used; all methods are class methods. Just call methods on the classes directly.

class `aiorpcx.`**`JSONRPC`**

An abstract base class for concrete protocol classes. *`JSONRPCv1`* and *`JSONRPCv2`* are derived protocol classes implementing JSON RPC versions 1.0 and 2.0 in a strict way.

class `aiorpcx.`**`JSONRPCv1`**

A derived class of *`JSONRPC`* implementing version 1.0 of the specification.

class `aiorpcx.`**`JSONRPCv2`**

A derived class of *`JSONRPC`* implementing version 2.0 of the specification.

class aiorpcx.**JSONRPCLoose**

A derived class of *JSONRPC*. It accepts messages that conform to either version 1.0 or version 2.0. As it is loose, it will also accept messages that conform strictly to neither version.

Unfortunately it is not possible to send messages that are acceptable to strict implementations of both versions 1.0 and 2.0, so it sends version 2.0 messages.

class aiorpcx.**JSONRPCAutoDetect**

Auto-detects the JSON RPC protocol version spoken by the remote side based on the first incoming message, from *JSONRPCv1*, *JSONRPCv2* and *JSONRPCLoose*. The RPC processor will then switch to that protocol version.

3.3.1 Message interpretation

classmethod JSONRPC.**message_to_item**(*message*)

Convert a binary message into an RPC object describing the message and return it.

Parameters **message** (*bytes*) – the message to interpret

Returns the RPC object

Return type *RPCRequest*, *RPCResponse* or *RPCBatch*.

If the message is ill-formed, return an *RPCRequest* object with its `method` set to an *RPCError* instance describing the error.

3.3.2 Message construction

These functions convert an RPC item into a binary message that can be passed over the network after framing.

classmethod JSONRPC.**request_message**(*item*)

Convert a request item to a message.

Parameters **item** – an *RPCRequest* item

Returns the message

Return type bytes

classmethod JSONRPC.**response_message**(*item*)

Convert a response item to a message.

Parameters **item** – an *RPCResponse* item

Returns the message

Return type bytes

classmethod JSONRPC.**error_message**(*item*)

Convert an error item to a message.

Parameters **item** – an *RPCError* item

Returns the message

Return type bytes

classmethod JSONRPC.**batch_message**(*item*)

Convert a batch item to a message.

Parameters **item** – an *RPCBatch* item

Returns the message

Return type bytes

classmethod `JSONRPC.`**`encode_payload`**(*payload*)

Encode a Python object as a JSON string and convert it to bytes. If the object cannot be encoded as JSON, a JSON "internal error" error message is returned instead, with ID equal to the "id" member of *payload* if that is a dictionary, otherwise `None`.

Parameters **`payload`** – a Python object that can be represented as JSON. Numbers, strings, lists, dictionaries, `True`, `False` and `None` are all valid.

Returns a JSON message

Return type bytes

3.4 RPC items

The `aiorpcx` module defines some classes, instances of which will be returned by some of its APIs. You should not need to instantiate these objects directly.

An instance of one of these classes is called an *item*.

class `aiorpcx.`**`RPCRequest`**

An RPC request or notification that has been received, or an outgoing notification.

Outgoing requests are represented by *`RPCRequestOut`* objects.

`method`

The RPC method being invoked, a string.

If an incoming request is ill-formed, so that, e.g., its method could not be determined, then this will be an *`RPCError`* instance that describes the error.

`args`

The arguments passed to the RPC method. This is a list or a dictionary, a dictionary if the arguments were passed by parameter name.

`request_id`

The ID given to the request so that responses can be associated with requests. Normally an integer, or `None` if the request is a *notification*. Rarely it might be a floating point number or string.

`is_notification`()

Returns `True` if the request is a notification (its *`request_id`* is `None`), otherwise `False`.

class `aiorpcx.`**`RPCRequestOut`**

An outgoing RPC request that is not a notification. A subclass of *`RPCRequest`* and `asyncio.Future`.

When an outgoing request is created, typically via the `send_request()` method of a client or server session, you can specify a callback to be called when the request is done. The callback is passed the request object, and the result can be obtained via its `result()` method.

A request can also be await-ed. Currently the result of await-ing is the same as calling `result()` on the request but this may change in future.

class `aiorpcx.`**`RPCResponse`**

An incoming or outgoing response. Outgoing response objects are automatically created by the framework when a request handler returns its result.

`result`

The response result, a Python object. If an error occurred this will be an *`RPCError`* object describing the error.

request_id
The ID of the request this is a repsonse to. Notifications do not get responses so this will never be `None`.

If `result` in an `RPCError` their `request_id` attributes will match.

class aiorpcx.**RPCError**
Represents an error, either in an `RPCResponse` object if an error occurred processing a request, or in a `RPCRequest` if an incoming request was ill-formed.

message
The error message as a string.

code
The error code, an integer.

request_id
The ID of the request that gave an error if it could be determined, otherwise `None`.

class aiorpcx.**RPCBatch**
Represents an incoming or outgoing RPC response batch, or an incoming RPC request batch.

items
A list of the items in the batch. The list cannot be empty, and each item will be an `RPCResponse` object for a response batch, and an `RPCRequest` object for a request batch.

Notifications and requests can be mixed together.

Batches are iterable through their items, and taking their length returns the length of the items list.

requests()
A generator that yields non-notification items of a request batch, or each item for a response batch.

request_ids()
A *frozenset* of all request IDs in the batch, ignoring notifications.

is_request_batch()
Return `True` if the batch is a request batch.

class aiorpcx.**RPCBatchOut**
An outgoing RPC batch. A subclass of `RPCBatch` and `asyncio.Future`.

When an outgoing request batch is created, typically via the `new_batch()` method of a client or server session, you can specify a callback to be called when the batch is done. The callback is passed the batch object.

Each non-notification item in an `RPCBatchOut` object is itself an `RPCRequestOut` object that can be independently waited on or cancelled. Notification items are `RPCRequest` objects. Since batches are responded to as a whole, all member requests will be completed simultaneously. The order of callbacks of member requests, and of the batch itself, is unspecified.

Cancelling a batch, or calling its `set_result()` or `set_exception()` methods cancels all its requests.

add_request(*method*, *args=None*, *on_done=None*)
Add a request to the batch. A callback can be specified that will be called when the request completes. Returns the `RPCRequestOut` request that was added to the batch.

add_notification(*method*, *args=None*)
Add a notification to the batch.

3.4.1 RPC Protocol Classes

RPC protocol classes should inherit from `RPCProtocolBase`. The base class provides a few utility functions returning `RPCError` objects. The derived class should redefine some constant class attributes.

class aiorpcx.**RPCProtocolBase**

INTERNAL_ERROR
The integer error code to use for an internal error.

INVALID_ARGS
The integer error code to use when an RPC request passes invalid arguments.

INVALID_REQUEST
The integer error code to use when an RPC request is invalid.

METHOD_NOT_FOUND
The integer error code to use when an RPC request is for a non-existent method.

classmethod JSONRPC.**internal_error**(*request_id*)
Return an *RPCError* object with error code `INTERNAL_ERROR` for the given request ID. The error message will be `"internal error processing request"`.

Parameters **request_id** – the request ID, normally an integer or string

Returns the error object

Return type *RPCError*

classmethod JSONRPC.**args_error**(*message*)
Return an *RPCError* object with error code `INVALID_ARGS` with the given error message and a request ID of `None`.

Parameters **message** (*str*) – the error message

Returns the error object

Return type *RPCError*

classmethod JSONRPC.**invalid_request**(*message*, *request_id=None*)
Return an *RPCError* object with error code `INVALID_REQUEST` with the given error message and request ID.

Parameters

- **message** (*str*) – the error message
- **request_id** – the request ID, normally an integer or string

Returns the error object

Return type *RPCError*

classmethod JSONRPC.**method_not_found**(*message*)
Return an *RPCError* object with error code `METHOD_NOT_FOUND` with the given error message and a request ID `None`.

Parameters **message** (*str*) – the error message

Returns the error object

Return type *RPCError*

3.5 Exceptions

exception aiorpcx.**ConnectionError**
When a connection is lost that has pending requests, this exception is set on those requests.

3.5.1 Server

A simple wrapper around an `asyncio.Server` object (see asyncio.Server).

class `aiorpcx.`**`Server`**(*protocol_factory*, *host=None*, *port=None*, ***, *loop=None*, ***kwargs*)

Creates a server that listens for connections on *host* and *port*. The server does not actually start listening until `listen()` is await-ed.

protocol_factory is any callable returning an `asyncio.Protocol` instance. You might find returning an instance of `ServerSession`, or a class derived from it, more useful.

loop is the event loop to use, or `asyncio.get_event_loop()` if `None`.

kwargs are passed through to loop.create_server().

A server instance has the following attributes:

`loop`
The event loop being used.

`host`
The host passed to the constructor

`port`
The port passed to the constructor

`server`
The underlying `asyncio.Server` object when the server is listening, otherwise `None`.

`listen``()`
Start listening for incoming connections. Return an `asyncio.Server` instance, which can also be accessed via `server`.

This method is a coroutine.

`close``()`
Close the listening socket if the server is listening, and wait for it to close. Return immediately if the server is not listening.

This does nothing to protocols and transports handling existing connections. On return `server` is `None`.

`wait_closed``()`
Returns when the server has closed.

This method is a coroutine.

3.5.2 Sessions

Convenience classes are provided for client and server sessions.

class `aiorpcx.`**`ClientSession`**(*host*, *port*, ***, *rpc_protocol=None*, *framer=None*, *scheduler=None*, *loop=None*, *proxy=None*, ***kwargs*)

An instance of an `asyncio.Protocol` class that represents an RPC session with a remote server at *host* and *port*, as documented in loop.create_connection().'

If *proxy* is not given, `create_connection()` uses `loop.create_connection()` to attempt a connection, otherwise `SOCKSProxy.create_connection()`. You can pass additional arguments to those functions with *kwargs* (*host* and *port* and *loop* are used as given).

rpc_protocol specifies the RPC protocol the server speaks. If `None` the protocol returned by `default_rpc_protocol()` is used.

framer handles RPC message framing, and if `None` then the framer returned by `default_framer()` is used.

scheduler should be left as `None`.

Logging will be sent to *logger*, `None` will use a logger specific to the *`ClientSession`* object's class.

`create_connection``()`
Make a connection attempt to the remote server. If successful this return a `(transport, protocol)` pair.

This method is a coroutine.

`default_rpc_protocol``()`
You can override this method to provide a default RPC protocol. *`JSONRPCv2`* is returned by the default implementation.

`default_framer``()`
You can override this method to provide a default message frmaer. A new *`NewlineFramer`* instance is returned by the default implementation.

The *`ClientSession`* and `ServerSession` classes share a base class that has the following attributes and methods:

3.6 SOCKS Proxy

The `aiorpcx` package includes a SOCKS proxy client. It understands the `SOCKS4`, `SOCKS4a` and `SOCKS5` protocols.

3.6.1 Exceptions

`exception` `aiorpcx.`**`SOCKSError`**
The base class of SOCKS exceptions. Each raised exception will be an instance of a derived class.

`exception` `aiorpcx.`**`SOCKSProtocolError`**
A subclass of *`SOCKSError`*. Raised when the proxy does not follow the `SOCKS` protocol.

`exception` `aiorpcx.`**`SOCKSFailure`**
A subclass of *`SOCKSError`*. Raised when the proxy refuses or fails to make a connection.

3.6.2 Authentication

Currently the only supported authentication method is with a username and password. Usernames can be used by all SOCKS protocols, but only `SOCKS5` uses the password.

`class` `aiorpcx.`**`SOCKSUserAuth`**
A `namedtuple` for authentication with a SOCKS server. It has two members:

`username`
A string.

`password`
A string. Ignored by the *`SOCKS4`* and *`SOCKS4a`* protocols.

3.6.3 Protocols

When creating a `SocksProxy` object, a protocol must be specified and be one of the following.

class aiorpcx.**SOCKS4**
An abstract class representing the `SOCKS4` protocol.

class aiorpcx.**SOCKS4a**
An abstract class representing the `SOCKS4a` protocol.

class aiorpcx.**SOCKS5**
An abstract class representing the `SOCKS5` protocol.

3.6.4 Proxy

You can create a *SOCKSProxy* object directly, but using one of its auto-detection class methods is likely more useful.

class aiorpcx.**SOCKSProxy**(*address*, *protocol*, *auth*)
An object representing a SOCKS proxy. The address is a Python socket address typically a (host, port) pair for IPv4, and a (host, port, flowinfo, scopeid) tuple for IPv6.

The *protocol* is one of *SOCKS4*, *SOCKS4a* and *SOCKS5*.

auth is a *SOCKSUserAuth* object or `None`.

After construction, *host*, *port* and *peername* are set to `None`.

classmethod auto_detect_address(*address*, *auth*, ***, *loop=None*, *timeout=5.0*)
Try to detect a SOCKS proxy at *address*.

Protocols *SOCKS5*, *SOCKS4a* and *SOCKS4* are tried in order. If a SOCKS proxy is detected return a *SOCKSProxy* object, otherwise `None`. Returning a proxy object only means one was detected, not that it is functioning - for example, it may not have full network connectivity.

auth is a *SOCKSUserAuth* object or `None`.

If testing any protocol takes more than *timeout* seconds, it is timed out and taken as not detected.

This class method is a coroutine.

classmethod auto_detect_host(*host*, *ports*, *auth*, ***, *loop=None*, *timeout=5.0*)
Try to detect a SOCKS proxy on *host* on one of the *ports*.

Call *auto_detect_address()* for each `(host, port)` pair until a proxy is detected, and return it, otherwise `None`.

auth is a *SOCKSUserAuth* object or `None`.

If testing any protocol on any port takes more than *timeout* seconds, it is timed out and taken as not detected.

This class method is a coroutine.

create_connection(*protocol_factory*, *host*, *port*, ***, *resolve=False*, *loop=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *timeout=30.0*)
Connect to (host, port) through the proxy in the background. When successful, the coroutine returns a `(transport, protocol, address)` triple, and sets the proxy attribute *peername*.

- If *resolve* is `True`, *host* is resolved locally rather than by the proxy. *family*, *proto*, *flags* are the optional address family, protocol and flags passed to loop.getaddrinfo() to get a list of remote addresses. If given, these should all be integers from the corresponding `socket` module constants.
- *ssl* is as documented for loop.create_connection().

If successfully connected the `_address` member of the protocol is set. If *resolve* is `True` it is set to the successful address, otherwise `(host, port)`.

If connecting takes more than *timeout* seconds an `asyncio.TimeoutError` exception is raised.

This method is a coroutine.

`host`
Set on a successful *`create_connection()`* to the host passed to the proxy server. This will be the resolved address if its *resolve* argument was `True`.

`port`
Set on a successful *`create_connection()`* to the host passed to the proxy server.

`peername`
Set on a successful *`create_connection()`* to the result of `socket.getpeername()` on the socket connected to the proxy.

CHAPTER 4

Indices and tables

- genindex
- search

Index