aiozmq

Release 0.7.1

September 20, 2015

Contents

ZeroMQ integration with asyncio (**PEP 3156**).

CHAPTER 1

Features

- Implements *create_zmq_connection()* coroutine for making 0MQ connections.
- Provides *ZmqTransport* and *ZmqProtocol*
- Provides RPC *Request-Reply*, *Push-Pull* and *Publish-Subscribe* patterns for *remote calls*.

Note: The library works on Linux, MacOS X and Windows.

But Windows is a second-class citizen in *ZeroMQ* world, sorry.

Thus *aiozmq* has *limited* support for Windows also.

Limitations are:

- You obviously cannot use *ipc://name* schema for *endpoint*
- aiozmq's loop *aiozmq.ZmqEventLoop* is built on top of `select` system call, so it's not fast comparing to `asyncio.ProactorEventLoop` and it doesn't support subprocesses.

CHAPTER 2

Library Installation

The *core* requires only *pyzmq* and can be installed (with pyzmq as dependency) by executing:

```
pip3 install aiozmq
```

Also probably you want to use `aiozmq.rpc`. RPC module is **optional** and requires *msgpack*. You can install *msgpack-python* by executing:

```
pip3 install msgpack-python
```

Note: *aiozmq* can be executed by *Python 3* only. The most Linux distributions uses *pip3* for installing *Python 3 libraries*. But your system may be using *Python 3* by default than try just *pip* instead of *pip3*. The same may be true for *virtualenv*, *travis continuous integration system* etc.

CHAPTER 3

Source code

The project is hosted on GitHub

Please feel free to file an issue on bug tracker if you have found a bug or have some suggestion for library improvement.

The library uses Travis for Continious Integration.

CHAPTER 4

IRC channel

You can discuss the library on Freenode at **#aio-libs** channel.

CHAPTER 5

Dependencies

- Python 3.3 and *asyncio* or Python 3.4+
- *ZeroMQ* 3.2+
- *pyzmq* 13.1+ (did not test with earlier versions)
- aiozmq.rpc requires *msgpack*

CHAPTER 6

Authors and License

The `aiozmq` package is initially written by Nikolay Kim, now maintained by Andrew Svetlov. It's BSD licensed and freely available. Feel free to improve this package and send a pull request to GitHub.

CHAPTER 7

Getting Started

Low-level request-reply example:

```
import asyncio
import aiozmq
import zmq

@asyncio.coroutine
def go():
    router = yield from aiozmq.create_zmq_stream(
        zmq.ROUTER,
        bind='tcp://127.0.0.1:*')

    addr = list(router.transport.bindings())[0]
    dealer = yield from aiozmq.create_zmq_stream(
        zmq.DEALER,
        connect=addr)

    for i in range(10):
        msg = (b'data', b'ask', str(i).encode('utf-8'))
        dealer.write(msg)
        data = yield from router.read()
        router.write(data)
        answer = yield from dealer.read()
        print(answer)
    dealer.close()
    router.close()

asyncio.get_event_loop().run_until_complete(go())
```

Example of RPC usage:

```
import aiozmq.rpc

class ServerHandler(aiozmq.rpc.AttrHandler):
    @aiozmq.rpc.method
    def remote_func(self, a:int, b:int) -> int:
        return a + b

@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        ServerHandler(), bind='tcp://127.0.0.1:5555')
    client = yield from aiozmq.rpc.connect_rpc(
```

```
        connect='tcp://127.0.0.1:5555')

    ret = yield from client.rpc.remote_func(1, 2)
    assert 3 == ret

    server.close()
    client.close()

asyncio.get_event_loop().run_until_complete(go())
```

Note: To execute the last example you need to *install msgpack* first.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

8.1 Streams API

New in version 0.6.

aiozmq provides a high level stream oriented API on top of the low-level API (*ZmqTransport* and *ZmqProtocol*) which can provide a more convinient API.

Here's an example:

```
import asyncio
import aiozmq
import zmq

@asyncio.coroutine
def go():
    router = yield from aiozmq.create_zmq_stream(
        zmq.ROUTER,
        bind='tcp://127.0.0.1:*')

    addr = list(router.transport.bindings())[0]
    dealer = yield from aiozmq.create_zmq_stream(
        zmq.DEALER,
        connect=addr)

    for i in range(10):
        msg = (b'data', b'ask', str(i).encode('utf-8'))
        dealer.write(msg)
        data = yield from router.read()
        router.write(data)
        answer = yield from dealer.read()
        print(answer)
    dealer.close()
    router.close()

asyncio.get_event_loop().run_until_complete(go())
```

The code creates two streams for request and response part of *ZeroMQ* connection and sends message through the wire with waiting for response.

Socket events can also be monitored when using streams.

```
import asyncio
import aiozmq
import zmq


@asyncio.coroutine
def monitor_stream(stream):
    try:
        while True:
            event = yield from stream.read_event()
            print(event)
    except aiozmq.ZmqStreamClosed:
        pass


@asyncio.coroutine
def go():
    router = yield from aiozmq.create_zmq_stream(
        zmq.ROUTER,
        bind='tcp://127.0.0.1:*')
    addr = list(router.transport.bindings())[0]

    dealer = yield from aiozmq.create_zmq_stream(
        zmq.DEALER)

    yield from dealer.transport.enable_monitor()

    asyncio.Task(monitor_stream(dealer))

    yield from dealer.transport.connect(addr)

    for i in range(10):
        msg = (b'data', b'ask', str(i).encode('utf-8'))
        dealer.write(msg)
        data = yield from router.read()
        router.write(data)
        answer = yield from dealer.read()
        print(answer)

    router.close()
    dealer.close()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.1.1 create_zmq_stream

aiozmq.**create_zmq_stream**(*zmq_type*, *, *bind=None*, *connect=None*, *loop=None*, *zmq_sock=None*, *high_read=None*, *low_read=None*, *high_write=None*, *low_write=None*, *events_backlog=100*)

A wrapper for `create_zmq_connection()` returning a ZeroMQ stream (`ZmqStream` instance).

The arguments are all the usual arguments to `create_zmq_connection()` plus high and low watermarks for reading and writing messages.

This function is a coroutine.

Parameters

- **zmq_type** (*int*) – a type of *ZeroMQ* socket (*zmq.REQ*, *zmq.REP*, *zmq.PUB*, *zmq.SUB*, zmq.PAIR*, *zmq.DEALER*, *zmq.ROUTER*, *zmq.PULL*, *zmq.PUSH*, etc.)
- **bind** (*str or iterable of strings*) – endpoints specification.

  Every *endpoint* generates call to `ZmqTransport.bind()` for accepting connections from specified endpoint.

  Other side should use *connect* parameter to connect to this transport.
- **connect** (*str or iterable of strings*) – endpoints specification.

  Every *endpoint* generates call to `ZmqTransport.connect()` for connecting transport to specified endpoint.

  Other side should use bind parameter to wait for incoming connections.
- **zmq_sock** (*zmq.Socket*) – a preexisting zmq socket that will be passed to returned transport.
- **loop** (*asyncio.AbstractEventLoop*) – optional event loop instance, `None` for default event loop.
- **high_read** (*int*) – high-watermark for reading from *ZeroMQ* socket. `None` by default (no limits).
- **low_read** (*int*) – low-watermark for reading from *ZeroMQ* socket. `None` by default (no limits).
- **high_write** (*int*) – high-watermark for writing into *ZeroMQ* socket. `None` by default (no limits).
- **low_write** (*int*) – low-watermark for writing into *ZeroMQ* socket. `None` by default (no limits).
- **events_backlog** (*int*) – backlog size for monitoring events, `100` by default. It specifies size of event queue. If count of unread events exceeds *events_backlog* the oldest events are discarded.

  Use `None` for unlimited backlog size.

Returns ZeroMQ stream object, `ZmqStream` instance.

New in version 0.7: events_backlog parameter

8.1.2 ZmqStream

class aiozmq.**ZmqStream**

A class for sending and receiving *ZeroMQ* messages.

transport
ZmqTransport instance, used for the stream.

at_closing()
Return `True` if the buffer is empty and *feed_closing()* was called.

close()
Close the stream and underlying *ZeroMQ* socket.

drain()
Wait until the write buffer of the underlying transport is flushed.

The intended use is to write:

```
w.write(data)
yield from w.drain()
```

When the transport buffer is full (the protocol is paused), block until the buffer is (partially) drained and the protocol is resumed. When there is nothing to wait for, the yield-from continues immediately.

This method is a coroutine.

exception()
Get the stream exception.

get_extra_info(*name*, *default=None*)
Return optional transport information: see `asyncio.BaseTransport.get_extra_info()`.

read()
Read one *ZeroMQ* message from the wire and return it.

Raise *ZmqStreamClosed* if the stream was closed.

read_event()
Read one *ZeroMQ* monitoring event and return it.

Raise *ZmqStreamClosed* if the stream was closed.

Monitoring mode should be enabled by *ZmqTransport.enable_monitor()* call first:

```
yield from stream.transport.enable_monitor()
```

New in version 0.7.

write(*msg*)
Writes message *msg* into *ZeroMQ* socket.

Parameters **msg** – a sequence (`tuple` or `list`), containing multipart message daata.

Internal API

set_exception(*exc*)
Set the exception to *exc*. The exception may be retrieved by *exception()* call or raised by next *read()*, *the private method*.

set_transport(*transport*)
Set the transport to *transport*, *the private method*.

set_read_buffer_limits(*high=None*, *low=None*)
Set read buffer limits, *the private method*.

feed_closing()
Feed the socket closing signal, *the private method*.

feed_msg(*msg*)
Feed *msg* message to the stream's internal buffer. Any operations waiting for the data will be resumed.

The private method.

feed_event(*event*)
Feed a socket *event* message to the stream's internal buffer.

The private method.

8.1.3 Exceptions

exception aiozmq.**ZmqStreamClosed**
Raised by read operations on closed stream.

8.2 Remote Procedure Calls

8.2.1 Intro

While *core API* provides a core support for *ZeroMQ* transports, the *End User* may need some high-level API.

Thus we have the *aiozmq.rpc* module for Remote Procedure Calls.

The main goal of the module is to provide *easy-to-use interface* for calling some method from the remote process (which can be running on the other host).

ZeroMQ itself gives some handy sockets but says nothing about RPC.

On the other hand, this module provides *human* API, but it is not compatible with *other implementations*.

If you need to support a custom protocol over *ZeroMQ* layer, please feel free to build your own implementation on top of the *core primitives*.

The *aiozmq.rpc* supports three pairs of communications:

- *Request-Reply*
- *Push-Pull*
- *Publish-Subscribe*

Warning: *aiozmq.rpc* module is **optional** and requires *msgpack*. You can install *msgpack-python* by executing:

```
pip3 install msgpack-python\>=0.4.0
```

8.2.2 Request-Reply

This is a **Remote Procedure Call** pattern itself. Client calls a remote function on server and waits for the returned value. If the remote function raises an exception, that exception instance is also raised on the client side.

Let's assume we have *N* clients bound to *M* servers. Any client can connect to several servers and any server can listen to multiple *endpoints*.

When client sends a message, the message will be delivered to any server that is ready (doesn't processes another message).

When the server sends a reply with the result of the remote call back, the result is routed to the client that has sent the request originally.

This pair uses *DEALER/ROUTER* *ZeroMQ* sockets.

The basic usage is:

```
import asyncio
from aiozmq import rpc

class Handler(rpc.AttrHandler):

    @rpc.method
    def remote(self, arg1, arg2):
        return arg1 + arg2

@asyncio.coroutine
def go():
    server =  yield from rpc.serve_rpc(Handler(),
                                       bind='tcp://127.0.0.1:5555')

    client = yield from rpc.connect_rpc(connect='tcp://127.0.0.1:5555')

    ret = yield from client.call.remote(1, 2)
    assert ret == 3

event_loop.run_until_complete(go())
```

aiozmq.rpc.**connect_rpc**(*, *connect=None*, *bind=None*, *loop=None*, *error_table=None*, *timeout=None*, *translation_table=None*)

A coroutine that creates and connects/binds *RPC* client.

Usually for this function you need to use *connect* parameter, but *ZeroMQ* does not forbid to use *bind*.

Parameters *bind*, *connect* and *loop* work like that of *aiozmq.create_zmq_connection()*.

Parameters

- **error_table** (*dict*) – an optional table for custom exception translators.

  See also:

  Exception translation at client side

- **timeout** (*float*) – an optional timeout for RPC calls. If *timeout* is not *None* and remote call takes longer than *timeout* seconds then asyncio.TimeoutError will be raised at client side. If the server will return an answer after timeout has been raised that answer **is ignored**.

  See also:

  RPCClient.with_timeout() method.

- **translation_table** (*dict*) – an optional table for custom value translators.

  See also:

  Value translators

Returns *RPCClient* instance.

aiozmq.rpc.**serve_rpc**(*handler*, *, *bind=None*, *connect=None*, *loop=None*, *log_exceptions=False*, *exclude_log_exceptions=()*, *translation_table=None*, *timeout=None*)

A coroutine that creates and connects/binds *RPC* server instance.

Usually for this function you need to use *bind* parameter, but *ZeroMQ* does not forbid to use *connect*.

Parameters *bind*, *connect* and *loop* work like that of `aiozmq.create_zmq_connection()`.

Parameters

- **handler** (aiozmq.rpc.AbstractHander) –

  an object which processes incoming RPC calls.

  Usually you like to pass `AttrHandler` instance.

- **log_exceptions** (*bool*) – log exceptions from remote calls if `True`.

  See also:

  Logging exceptions from remote calls at server side

- **exclude_log_exceptions** (*sequence*) – sequence of exception types that should not to be logged if *log_exceptions* is `True`.

  See also:

  Logging exceptions from remote calls at server side

- **translation_table** (*dict*) – an optional table for custom value translators.

  See also:

  Value translators

- **timeout** (*float*) – timeout for performing handling of async server calls.

  If call handling takes longer than *timeout* then procedure will be cancelled with `asyncio.TimeoutError`.

  The value should be a bit longer than timeout for client side.

Returns `Service` instance.

Changed in version 0.2: Added *log_exceptions* parameter.

8.2.3 Push-Pull

This is a **Notify** aka **Pipeline** pattern. Client calls a remote function on the server and **doesn't** wait for the result. If a *remote function call* raises an exception, this exception is only **logged** at the server side. Client **cannot** get any information about *processing the remote call on server*.

Thus this is **one-way** communication: **fire and forget**.

Let's assume that we have *N* clients bound to *M* servers. Any client can connect to several servers and any server can listen to multiple *endpoints*.

When client sends a message, the message will be delivered to any server that is *ready* (doesn't processes another message).

That's all.

This pair uses *PUSH*/*PULL* *ZeroMQ* sockets.

The basic usage is:

```
import asyncio
from aiozmq import rpc

class Handler(rpc.AttrHandler):
```

```
    @rpc.method
    def remote(self):
        do_something(arg)

@asyncio.coroutine
def go():
    server =  yield from rpc.serve_pipeline(Handler(),
                                            bind='tcp://127.0.0.1:5555')

    client = yield from rpc.connect_pipeline(connect='tcp://127.0.0.1:5555')

    ret = yield from client.notify.remote(1)

event_loop.run_until_complete(go())
```

aiozmq.rpc.**connect_pipeline**(*, *connect=None*, *bind=None*, *loop=None*, *error_table=None*, *translation_table=None*)

A coroutine that creates and connects/binds *pipeline* client.

Parameters *bind*, *connect* and *loop* work like that of *aiozmq.create_zmq_connection()*.

Usually for this function you need to use *connect* parameter, but *ZeroMQ* does not forbid to use *bind*.

Parameters **translation_table** (*dict*) – an optional table for custom value translators.

See also:

Value translators

Returns *PipelineClient* instance.

aiozmq.rpc.**serve_pipeline**(*handler*, *, *connect=None*, *bind=None*, *loop=None*, *log_exceptions=False*, *exclude_log_exceptions=()*, *translation_table=None*, *timeout=None*)

A coroutine that creates and connects/binds *pipeline* server instance.

Usually for this function you need to use *bind* parameter, but *ZeroMQ* does not forbid to use *connect*.

Parameters *bind*, *connect* and *loop* work like that of *aiozmq.create_zmq_connection()*.

Parameters

- **handler** (aiozmq.rpc.AbstractHander) –

  an object which processes incoming *pipeline* calls.

  Usually you like to pass *AttrHandler* instance.

- **log_exceptions** (*bool*) – log exceptions from remote calls if `True`.

  See also:

  Logging exceptions from remote calls at server side

- **exclude_log_exceptions** (*sequence*) – sequence of exception types that should not to be logged if *log_exceptions* is `True`.

  See also:

  Logging exceptions from remote calls at server side

- **translation_table** (*dict*) – an optional table for custom value translators.

  See also:

  Value translators

- **timeout** (*float*) – timeout for performing handling of async server calls.

  If call handling takes longer than *timeout* then procedure will be cancelled with `asyncio.TimeoutError`.

  The value should be a bit longer than timeout for client side.

Returns *Service* instance.

Changed in version 0.2: Added *log_exceptions* parameter.

8.2.4 Publish-Subscribe

This is **PubSub** pattern. It's very close to *Publish-Subscribe* but has some difference:

- server *subscribes* to *topics* in order to receive messages only from that *topics*.
- client sends a message to concrete *topic*.

Let's assume we have *N* clients bound to *M* servers. Any client can connect to several servers and any server can listen to multiple *endpoints*.

When client sends a message to *topic*, the message will be delivered to servers that only has been subscribed to this *topic*.

This pair uses *PUB/SUB* *ZeroMQ* sockets.

The basic usage is:

```
import asyncio
from aiozmq import rpc

class Handler(rpc.AttrHandler):

    @rpc.method
    def remote(self):
        do_something(arg)

@asyncio.coroutine
def go():
    server =  yield from rpc.serve_pubsub(Handler(),
                                          subscribe='topic',
                                          bind='tcp://127.0.0.1:5555')

    client = yield from rpc.connect_pubsub(connect='tcp://127.0.0.1:5555')

    ret = yield from client.publish('topic').remote(1)

event_loop.run_until_complete(go())
```

aiozmq.rpc.**connect_pubsub**(*, *connect=None*, *bind=None*, *loop=None*, *error_table=None*, *translation_table=None*)

A coroutine that creates and connects/binds *pubsub* client.

Usually for this function you need to use *connect* parameter, but *ZeroMQ* does not forbid to use *bind*.

Parameters *bind*, *connect* and *loop* work like that of *aiozmq.create_zmq_connection()*.

Parameters **translation_table** (*dict*) – an optional table for custom value translators.

See also:

Value translators

Returns *PubSubClient* instance.

aiozmq.rpc.**serve_pubsub**(*handler*, ***, *connect=None*, *bind=None*, *subscribe=None*, *loop=None*, *log_exceptions=False*, *exclude_log_exceptions=()*, *translation_table=None*, *timeout=None*)

A coroutine that creates and connects/binds *pubsub* server instance.

Usually for this function you need to use *bind* parameter, but *ZeroMQ* does not forbid to use *connect*.

Parameters *bind*, *connect* and *loop* work like that of *aiozmq.create_zmq_connection()*.

param aiozmq.rpc.AbstractHander handler

an object which processes incoming *pipeline* calls.

Usually you like to pass *AttrHandler* instance.

param bool log_exceptions log exceptions from remote calls if `True`.

See also:

Logging exceptions from remote calls at server side

param sequence exclude_log_exceptions sequence of exception types that should not to be logged if *log_exceptions* is `True`.

See also:

Logging exceptions from remote calls at server side

param subscribe subscription specification.

Subscribe server to *topics*.

Allowed parameters are `str`, `bytes`, *iterable* of *str* or *bytes*.

param dict translation_table an optional table for custom value translators.

See also:

Value translators

param float timeout timeout for performing handling of async server calls.

If call handling takes longer than *timeout* then procedure will be cancelled with `asyncio.TimeoutError`.

The value should be a bit longer than timeout for client side.

return `PubSubService` instance.

raise OSError on system error.

raise TypeError if arguments have inappropriate type.

Changed in version 0.2: Added *log_exceptions* parameter.

8.2.5 Exception translation at client side

If a remote server method raises an exception, that exception is passed back to the client and raised on the client side, as follows:

```
try:
    yield from client.call.func_raises_value_error()
except ValueError as exc:
    log.exception(exc)
```

The rules for exception translation are:

- if remote method raises an exception — server answers with *full exception class name* (like `package.subpackage.MyError`) and *exception constructor arguments* (`args`).
- *translator table* is a *mapping* of `{excpetion_name: exc_class}` where keys are *full names* of exception class (str) and values are exception classes.
- if translation is found then client code gives exception `raise exc_class(args)`.
- user defined translators are searched first.
- all builtin exceptions are translated by default.
- *`NotFoundError`* and *`ParameterError`* are translated by default also.
- if there is no registered traslation then `GenericError(excpetion_name, args)` is raised.

For example if custom RPC server handler can raise `mod1.Error1` and `pack.mod2.Error2` then *error_table* should be:

```
from mod1 import Error1
from pack.mod2 import Error2

error_table = {'mod1.Error1': Error1,
               'pack.mod2.Error2': Error2}

client = loop.run_until_complete(
    rpc.connect_rpc(connect='tcp://127.0.0.1:5555',
                    error_table=error_table))
```

You have to have the way to import exception classes from server-side. Or you can build your own translators without server-side code, use only string for *full exception class name* and tuple of *args* — that's up to you.

See also:

error_table argument in *`connect_rpc()`* function.

8.2.6 Signature validation

The library supports **optional** validation of the remote call signatures.

If validation fails then *`ParameterError`* is raised on client side.

All validations are done on RPC server side, then errors are translated back to client.

Let's take a look on example of user-defined RPC handler:

```
class Handler(rpc.AttrHandler):

    @rpc.method
    def func(self, arg1: int, arg2) -> float:
        return arg1 + arg2
```

Parameter arg1 and *return value* has *annotaions*, *int* and *float* correspondingly.

At the call time, if *parameter* has an *annotaion*, then *actual value* passed and RPC method is calculated as `actual_value = annotation(value)`. If there is no annotaion for parameter, the value is passed as-is.

Changed in version 0.1.2: Function default values are not passed to an *annotaion*.

Annotaion should be any *callable* that accepts a value as single argument and returns *actual value*.

If annotation call raises exception, that exception is sent to the client wrapped in *`ParameterError`*.

Value, returned by RPC call, can be checked by optional *return annotation*.

Thus `int` can be a good annotation: it raises `TypeError` if *arg1* cannot be converted to *int*.

Usually you need more complex check, say parameter can be *int* or *None*.

You always can write a custom validator:

```
def int_or_none(val):
   if isinstance(val, int) or val is None:
       return val
   else:
       raise ValueError('bad value')

class Handler(rpc.AttrHandler):
    @rpc.method
    def func(self, arg: int_or_none):
        return arg
```

Writing a tons of custom validators is inconvinient, so we recommend to use *trafaret* library (can be installed via `pip3 install trafaret`).

This is example of trararet annotation:

```
import trafaret as t

class Handler(rpc.AttrHandler):
    @rpc.method
    def func(self, arg: t.Int|t.Null):
        return arg
```

Trafaret has advanced types like *List* and *Dict*, so you can put your complex JSON-like structure as RPC method annotation. Also you can create custom trafarets if needed. It's easy, trust me.

8.2.7 Value translators

aiozmq.rpc uses *msgpack* for transfering python objects from client to server and back.

You can think about *msgpack* as: this is a-like JSON but fast and compact.

Every object that can be passed to `json.dump()`, can be passed to `msgpack.dump()` also. The same for unpacking.

The only difference is: *aiozmq.rpc* converts all `lists` to `tuples`. The reasons is are:

- you never need to modify given list as it is your *incoming* value. If you still want to use `list` data type you can do it easy by `list(val)` call.
- tuples are a bit faster for unpacking.
- tuple can be a *key* in `dict`, so you can pack something like `{(1,2): 'a'}` and unpack it on other side without any error. Lists cannot be *keys* in dicts, they are unhashable.

  This point is the main reason for choosing tuples. Unfortunatelly msgpack gives no way to mix tuples and lists in the same pack.

But sometimes you want to call remote side with *non-plain-json* arguments. `datetime.datetime` is a good example. *`aiozmq.rpc`* supports all family of dates, times and timezones from `datetime` *from-the-box* (*predefined translators*).

If you need to transfer a custom object via RPC you should register **translator** at both server and client side. Say, you need to pass the instances of your custom class `Point` via RPC. There is an example:

```
import asyncio
import aiozmq, aiozmq.rpc
import msgpack

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if isinstance(other, Point):
            return (self.x, self.y) == (other.x, other.y)
        return NotImplemented

translation_table = {
    0: (Point,
        lambda value: msgpack.packb((value.x, value.y)),
        lambda binary: Point(*msgpack.unpackb(binary))),
}

class ServerHandler(aiozmq.rpc.AttrHandler):
    @aiozmq.rpc.method
    def remote(self, val):
        return val

@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        ServerHandler(), bind='tcp://127.0.0.1:5555',
        translation_table=translation_table)
    client = yield from aiozmq.rpc.connect_rpc(
        connect='tcp://127.0.0.1:5555',
        translation_table=translation_table)

    ret = yield from client.call.remote(Point(1, 2))
    assert ret == Point(1, 2)
```

You should create a *translation table* and pass it to both *connect_rpc()* and *serve_rpc()*. That's all, server and client now have all information about passing your `Point` via the wire.

- Translation table is the dict.
- Keys should be an integers in range [0, 127]. We recommend to use keys starting from 0 for custom translators, high numbers are reserved for library itself (it uses the same schema for passing *datetime* objects etc).
- Values are tuples of `(translated_class, packer, unpacker)`.
  - *translated_class* is a class which you want to pass to peer.
  - *packer* is a *callable* which receives your class instance and returns `bytes` of *instance data*.
  - *unpacker* is a *callable* which receives `bytes` of *instance data* and returns your *class instance*.
- When the library tries to pack your class instance it searches the *translation table* in ascending order.
- If your object is an `instance` of *translated_class* then *packer* is called and resulting `bytes` will be sent to peer.
- On unpacking *unpacker* is called with the `bytes` received by peer. The result should to be your class instance.

Warning: Please be careful with *translation table* order. Say, if you have `object` at position 0 then every lookup will stop at this. Even *datetime* objects will be redirected to *packer* and *unpacker* for registered *object* type.

Warning: While the easiest way to write *packer* and *unpacker* is to use `pickle` we **don't encourage that**. The reason is simple: *pickle* packs an object itself and all instances which are referenced by that object. So you can easy pass via network a half of your program without any warning.

Table of predefined translators:

Ordinal	Class
123	`datetime.tzinfo`
124	`datetime.timedelta`
125	`datetime.time`
126	`datetime.datetime`
127	`datetime.date`

Note: pytz timezones processed by predefined traslator for *tzinfo* (ordinal number 123) because they are inherited from `datetime.tzinfo`. So you don't need to register a custom translator for `pytz.datetime`.

That's happens because *aiozmq.rpc* uses `pickle` for translation `datetime` classes.

Pickling in this particular case is **safe** because all datetime classes are terminals and doesn't have a links to foreign class instances.

8.2.8 Logging exceptions from remote calls at server side

By default *aiozmq.rpc* does no logging if remote call raises an exception.

That behavoir can be changed by passing `log_exceptions=True` to rpc servers: *serve_rpc()*, *serve_pipeline()* and *serve_pubsub()*.

If, say, you make PubSub server as:

```
server =  yield from rpc.serve_pubsub(handler,
                                       subscribe='topic',
                                       bind='tcp://127.0.0.1:5555',
                                       log_exceptions=True)
```

then exceptions raised from *handler* remote calls will be logged by standard *aiozmq.rpc.logger*.

But sometimes you don't want to log exceptions of some types.

Say, you use your own exceptions as part of public API to report about expected failures. In this case you probably want to pass that exceptions over the log, but record all other unexpected errors.

For that case you can use *exclude_log_exceptions* parameter:

```
server =  yield from rpc.serve_rpc(handler,
                                   bind='tcp://127.0.0.1:7777',
                                   log_exceptions=True,
                                   exclude_log_exceptions=(MyError,
                                                           OtherError))
```

8.2.9 Exceptions

exception `aiozmq.rpc.`**`Error`**

Base class for *aiozmq.rpc* exceptions. Derived from `Exception`.

exception aiozmq.rpc.**GenericError**
Subclass of *Error*, raised when a remote call produces exception that cannot be translated.

exc_type
A string contains *full name* of unknown exception("package.module.MyError").

arguments
A tuple of arguments passed to *unknown exception* constructor

See also:

BaseException.args - parameters for exception constructor.

See also:

Exception translation at client side

exception aiozmq.rpc.**NotFoundError**
Subclass of both *Error* and LookupError, raised when a remote call name is not found at RPC server.

exception aiozmq.rpc.**ParameterError**
Subclass of both *Error* and ValueError, raised by remote call when parameter substitution or *remote method signature validation* is failed.

exception aiozmq.rpc.**ServiceClosedError**
Subclass of *Error*, raised *Service* has been closed.

See also:

Service.transport property.

8.2.10 Clases

@aiozmq.rpc.**method**
Marks a decorated function as RPC endpoint handler.

The func object may provide arguments and/or return annotations. If so annotations should be callable objects and they will be used to validate received arguments and/or return value.

Example:

```
@aiozmq.rpc.method
def remote(a: int, b: int) -> int:
    return a + b
```

Methods are objects that returned by *AbstractHander.__getitem__()* lookup at RPC method search stage.

class aiozmq.rpc.**AbstractHander**
The base class for all RPC handlers.

Every handler should be *AbstractHandler* by direct inheritance or indirect subclassing (method *__getitem__* should be defined).

Therefore *AttrHandler* and dict are both good citizens.

Returned value eighter should implement AbstractHandler interface itself for looking up forward or must be callable decorated by *method()*.

__getitem__(*self*, *key*)
Returns subhandler or terminal function decorated by *method()*.
Raises **KeyError** if key is not found.

See also:

`start_server()` coroutine.

class `aiozmq.rpc.`**AttrHandler**

Subclass of `AbstractHandler`. Does lookup for *subhandlers* and *rpc methods* by `getattr()`.

There is an example of trivial *handler*:

```
class ServerHandler(aiozmq.rpc.AttrHandler):
    @aiozmq.rpc.method
    def remote_func(self, a:int, b:int) -> int:
        return a + b
```

class `aiozmq.rpc.`**Service**

RPC service base class.

Instances of *Service* (or descendants) are returned by coroutines that creates clients or servers (*connect_rpc()*, *serve_rpc()* and others).

Implements `asyncio.AbstractServer`.

transport

The readonly property that returns service's *transport*.

You can use the transport to dynamically bind/unbind, connect/disconnect etc.

Raises *aiozmq.rpc.ServiceClosedError* if the service has been closed.

close()

Stop serving.

This leaves existing connections open.

wait_closed()

Coroutine to wait until service is closed.

See also:

Signature validation

class `aiozmq.rpc.`**RPCClient**

Class that returned by *connect_rpc()* call. Inherited from *Service*.

For RPC calls use `rpc` property.

call

The readonly property that returns ephemeral object used to making RPC call.

Construction like:

```
ret = yield from client.call.ns.method(1, 2, 3)
```

makes a remote call with arguments(1, 2, 3) and returns answer from this call.

You can also pass *named parameters*:

```
ret = yield from client.call.ns.method(1, b=2, c=3)
```

If the call raises exception that exception propagates to client side.

Say, if remote raises `ValueError` client catches `ValueError` instance with *args* sent by remote:

```
        try:
            yield from client.call.raise_value_error()
        except ValueError as exc:
            process_error(exc)
```

with_timeout(*timeout*)
Override default timeout for client. Can be used in two forms:

```
        yield from client.with_timeout(1.5).call.func()
```

and:

```
        with client.with_timeout(1.5) as new_client:
            yield from new_client.call.func1()
            yield from new_client.call.func2()
```

Parameters **timeout** (*float*) – a timeout for RPC calls. If *timeout* is not *None* and remote call takes longer than *timeout* seconds then `asyncio.TimeoutError` will be raised at client side. If the server will return an answer after timeout has been raised that answer **is ignored**.

See also:

connect_rpc() coroutine.

See also:

Exception translation at client side and *Signature validation*

class aiozmq.rpc.**PipelineClient**
Class that returned by *connect_pipeline()* call. Inherited from *Service*.

notify
The readonly property that returns ephemeral object used to making notification call.

Construction like:

```
        ret = yield from client.notify.ns.method(1, 2, 3)
```

makes a remote call with arguments(1, 2, 3) and returns *None*.

You cannot get any answer from the server.

class aiozmq.rpc.**PubSubClient**
Class that returned by *connect_pubsub()* call. Inherited from *Service*.

For *pubsub* calls use `publish()` method.

publish(*topic*)
The call that returns ephemeral object used to making *publisher call*.

Construction like:

```
        ret = yield from client.publish('topic').ns.method(1, b=2)
```

makes a remote call with arguments `(1, b=2)` and topic name `b'topic'` and returns *None*.

You cannot get any answer from the server.

See also:

Signature validation

8.2.11 Logger

aiozmq.rpc.**logger**
An instance of logging.Logger with *name* aiozmq.rpc.

The library sends log messages (*Logging exceptions from remote calls at server side* for example) to this logger. You can configure your own handlers to fiter, save or what-you-wish the log events from the library.

8.3 Core API

8.3.1 create_zmq_connection

aiozmq.**create_zmq_connection**(*protocol_factory*, *zmq_type*, ***, *bind=None*, *connect=None*, *zmq_sock=None*, *loop=None*)
Create a ZeroMQ connection.

This method is a coroutine.

If you don't use *bind* or *connect* params you can do it later by *ZmqTransport.bind()* and *ZmqTransport.connect()* calls.

Parameters

- **protocol_factory** (*callable*) – a factory that instantiates *ZmqProtocol* object.
- **zmq_type** (*int*) – a type of *ZeroMQ* socket (*zmq.REQ*, *zmq.REP*, *zmq.PUB*, *zmq.SUB*, zmq.PAIR*, *zmq.DEALER*, *zmq.ROUTER*, *zmq.PULL*, *zmq.PUSH*, etc.)
- **bind** (*str or iterable of strings*) – endpoints specification.

  Every *endpoint* generates call to *ZmqTransport.bind()* for accepting connections from specified endpoint.

  Other side should use *connect* parameter to connect to this transport.
- **connect** (*str or iterable of strings*) – endpoints specification.

  Every *endpoint* generates call to *ZmqTransport.connect()* for connecting transport to specified endpoint.

  Other side should use bind parameter to wait for incoming connections.
- **zmq_sock** (*zmq.Socket*) – a preexisting zmq socket that will be passed to returned transport.
- **loop** (*asyncio.AbstractEventLoop*) – optional event loop instance, None for default event loop.

Returns a pair of (transport, protocol) where transport supports *ZmqTransport* interface.

Return type tuple

New in version 0.5.

8.3.2 ZmqTransport

class aiozmq.**ZmqTransport**
Transport for *ZeroMQ* connections. Implements asyncio.BaseTransport interface.

End user should never create *ZmqTransport* objects directly, he gets it by `yield from aiozmq.create_zmq_connection()` call.

get_extra_info(*key*, *default=None*)
Return optional transport information if name is present otherwise return *default*.

ZmqTransport supports the only valid *key*: `"zmq_socket"`. The value is `zmq.Socket` instance.

Parameters

- **name** (*str*) – name of info record.
- **default** – default value

close()
Close the transport.

Buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's *connection_lost()* method will (eventually) called with *None* as its argument.

write(*data*)
Write message to the transport.

Parameters **data** – iterable to send as multipart message.

This does not block; it buffers the data and arranges for it to be sent out asynchronously.

abort()
Close the transport immediately.

Buffered data will be lost. No more data will be received. The protocol's *connection_lost()* method will (eventually) be called with *None* as it's argument.

getsockopt(*option*)
Get *ZeroMQ* socket option.

Parameters **option** (*int*) – a constant like *zmq.SUBSCRIBE*, *zmq.UNSUBSCRIBE*, *zmq.TYPE* etc.

For list of available options please see: http://api.zeromq.org/master:zmq-getsockopt

Returns option value

Raises **OSError** if call to ZeroMQ was unsuccessful.

setsockopt(*option*, *value*)

Set *ZeroMQ* socket option.

param int option a constant like *zmq.SUBSCRIBE*, *zmq.UNSUBSCRIBE*, *zmq.TYPE* etc.

param value a new option value, it's type depend of option name.

For list of available options please see: http://api.zeromq.org/master:zmq-setsockopt

get_write_buffer_limits()
Get the *high*- and *low*-water limits for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use *set_write_buffer_limits()* to set the limits.

New in version 0.6.

set_write_buffer_limits(*high=None*, *low=None*)
Set the high- and low-water limits for write flow control.

Parameters

- **high** (*int or None*) – high-water limit
- **low** (*int or None*) – low-water limit

These two values control when to call the protocol's *pause_writing()* and *resume_writing()* methods. If specified, the low-water limit must be less than or equal to the high-water limit. Neither value can be negative.

The defaults are implementation-specific. If only the high-water limit is given, the low-water limit defaults to a implementation-specific value less than or equal to the high-water limit. Setting high to zero forces low to zero as well, and causes *pause_writing()* to be called whenever the buffer becomes non-empty. Setting low to zero causes *resume_writing()* to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use *get_write_buffer_limits()* to get the limits.

get_write_buffer_size()
Return the current size of the write buffer.

pause_reading()
Pause the receiving end.

No data will be passed to the protocol's *ZmqProtocol.msg_received()* method until *ZmqTransport.resume_reading()* is called.

See also:

ZmqTransport.resume_reading() method.

resume_reading()
Resume the receiving end.

Data received will once again be passed to the protocol's *ZmqProtocol.msg_received()* method.

See also:

ZmqTransport.pause_reading() method.

bind(*endpoint*)
Bind transpot to *endpoint*. See http://api.zeromq.org/master:zmq-bind for details.

This method is a coroutine.

Parameters **endpoint** – a string in format `transport://address` as *ZeroMQ* requires.

Returns bound endpoint, unwinding wildcards if needed.

Return type `str`

Raises

- **OSError** – on error from ZeroMQ layer
- **TypeError** – if *endpoint* is not a `str`

unbind(*endpoint*)
Unbind transpot from *endpoint*.

This method is a coroutine.

Parameters **endpoint** – a string in format `transport://address` as *ZeroMQ* requires.

Returns *None*

Raises

- **OSError** – on error from ZeroMQ layer
- **TypeError** – if *endpoint* is not a `str`

bindings()
Return immutable set of *endpoints* bound to transport.

Note: Returned endpoints include only ones that has been bound via `ZmqTransport.bind()` or `create_zmq_connection()` calls and do not include bindings that have been done on *zmq_sock* before `create_zmq_connection()` call.

---

connect(*endpoint*)
Connect transpot to *endpoint*. See http://api.zeromq.org/master:zmq-connect for details.

This method is a coroutine.

Parameters **endpoint** (*str*) – a string in format `transport://address` as *ZeroMQ* requires.

For tcp connections the *endpoint* should specify *IPv4* or *IPv6* address, not *DNS* name. Use `yield from get_event_loop().getaddrinfo(host, port)` for translating *DNS* into *IP address*.

Returns endpoint

Return type `str`

Raises

- **ValueError** – if the endpoint is a tcp DNS address.
- **OSError** – on error from ZeroMQ layer
- **TypeError** – if *endpoint* is not a `str`

disconnect(*endpoint*)
Disconnect transpot from *endpoint*.

This method is a coroutine.

Parameters **endpoint** – a string in format `transport://address` as *ZeroMQ* requires.

Returns *None*

Raises

- **OSError** – on error from ZeroMQ layer
- **TypeError** – if *endpoint* is not a `str`

connections()
Return immutable set of *endpoints* connected to transport.

Note: Returned endpoints include only ones that has been connected via `ZmqTransport.connect()` or `create_zmq_connection()` calls and do not include connections that have been done to *zmq_sock* before `create_zmq_connection()` call.

---

subscribe(*value*)
Establish a new message filter on *SUB* transport.

Newly created *SUB* transports filters out all incoming messages, therefore you should call this method to establish an initial message filter.

An empty (`b''`) *value* subscribes to all incoming messages. A non-empty value subscribes to all messages beginning with the specified prefix. Multiple filters may be attached to a single *SUB* transport, in which case a message shall be accepted if it matches at least one filter.

Parameters **value** (*bytes*) – a filter value to add to *SUB* filters.

Raises

- **NotImplementedError** – the transport is not *SUB*.
- **TypeError** – when *value* is not bytes.

Warning: Unlike to *ZeroMQ* socket level the call first check for *value* in *ZmqTransport.subscriptions()* and does nothing if the transport already has been subscribed to the *value*.

unsubscribe(*value*)

Remove an existing message filter on a *SUB* transport.

The filter specified must match an existing filter previously established with the *ZmqTransport.subscribe()*.

If the transport has several instances of the same filter attached the `.unsubscribe()` removes only one instance, leaving the rest in place and functional (if you use *ZmqTransport.subscribe()* to adding new filters that never happens, see *difference between aiozmq and ZeroMQ raw sockets* for details).

Parameters **value** (*bytes*) – a filter value to add to *SUB* filters.

Raises

- **NotImplementedError** – the transport is not *SUB*.
- **TypeError** – when *value* is not bytes.

subscriptions()

Return immutable set of subscriptions (set of bytes) subscribed on transport.

Note: Returned subscriptions include only ones that has been subscribed via *ZmqTransport.subscribe()* call and do not include subscribtions that have been done to zmq_sock before *create_zmq_connection()* call.

Raises **NotImplementedError** the transport is not *SUB*.

enable_monitor(*events=None*)

Enables socket events to be reported for this socket. Socket events are passed to the protocol's *ZmqProtocol.event_received()* method.

The socket event monitor capability requires `libzmq >= 4` and `pyzmq >= 14.4`.

This method is a coroutine.

Parameters **events** – a bitmask of socket events to watch for. If no value is specified then all events will monitored (i.e. `zmq.EVENT_ALL`). For list of available events please see: http://api.zeromq.org/4-0:zmq-socket-monitor

Raises **NotImplementedError** if *libzmq* or *pyzmq* versions do not support socket monitoring.

New in version 0.7.

disable_monitor()

Stop the socket event monitor.

This method is a coroutine.

New in version 0.7.

8.3.3 ZmqProtocol

class aiozmq.**ZmqProtocol**

Protocol for *ZeroMQ* connections. Derives from asyncio.BaseProtocol.

connection_made(*transport*)

Called when a connection is made.

Parameters **transport** (ZmqTransport) – representing the pipe connection. To receive data, wait for *msg_received()* calls. When the connection is closed, *connection_lost()* is called.

connection_lost(*exc*)

Called when the connection is lost or closed.

Parameters **exc** (instance of Exception or derived class) – an exception object or *None* (the latter meaning the connection was aborted or closed).

pause_writing()

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

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

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

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

---

resume_writing()

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

See *pause_writing()* for details.

msg_received(*data*)

Called when some ZeroMQ message is received.

Parameters **data** (*list*) – the multipart list of bytes with at least one item.

event_received(*event*)

Called when a ZeroMQ socket event is received.

This method is only called when a socket monitor is enabled.

Parameters **event** (namedtuple) – a SocketEvent namedtuple containing 3 items: *event*, *value*, and *endpoint*.

New in version 0.7.

8.3.4 Exception policy

Every call to `zmq.Socket` method can raise `zmq.ZMQError` exception. But all methods of *ZmqEventLoop* and *ZmqTransport* translate ZMQError into `OSError` (or descendat) with errno and strerror borrowed from underlying ZMQError values.

The reason for translation is that Python 3.3 implements **PEP 3151 — Reworking the OS and IO Exception Hierarchy** which gets rid of exceptions zoo and uses `OSError` and descendants for all exceptions generated by system function calls.

aiozmq implements the same pattern. Internally it looks like:

```
try:
    return self._zmq_sock.getsockopt(option)
except zmq.ZMQError as exc:
    raise OSError(exc.errno, exc.strerror)
```

Also public methods of *aiozmq* will never raise `InterruptedError` (aka *EINTR*), they process interruption internally.

8.3.5 Getting aiozmq version

aiozmq.**version**
: a text version of the library:

```
'0.1.0 , Python 3.3.2+ (default, Feb 28 2014, 00:52:16) \n[GCC 4.8.1]'
```

aiozmq.**version_info**
: a named tuple with version information, useful for comparison:

```
VersionInfo(major=0, minor=1, micro=0, releaselevel='alpha', serial=0)
```

The Python itself uses the same schema (`sys.version_info`).

8.3.6 Installing ZeroMQ event loop

Deprecated since version 0.5: *aiozmq* works with any *asyncio* event loop, it doesn't require dedicated event loop policy.

To use *ZeroMQ* layer you **may** install proper event loop first.

The recommended way is to setup *global event loop policy*:

```
import asyncio
import aiozmq

asyncio.set_event_loop_policy(aiozmq.ZmqEventLoopPolicy())
```

That installs *ZmqEventLoopPolicy* globally. After installing you can get event loop instance from main thread by `asyncio.get_event_loop()` call:

```
loop = asyncio.get_event_loop()
```

If you need to execute event loop in your own (not main) thread you have to set it up first:

```
import threading

def thread_func():
```

```
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop()

    loop.run_forever()

thread = threading.Thread(target=thread_func)
thread.start()
```

8.3.7 ZmqEventLoopPolicy

Deprecated since version 0.5: *aiozmq* works with any *asyncio* event loop, it doesn't require dedicated event loop policy.

ZeroMQ policy implementation for accessing the event loop.

In this policy, each thread has its own event loop. However, we only automatically create an event loop by default for the main thread; other threads by default have no event loop.

ZmqEventLoopPolicy implements an `asyncio.AbstractEventLoopPolicy` interface.

class `aiozmq.`**ZmqEventLoopPolicy**

Create policy for ZeroMQ event loops.

Note: policy should be **installed**, see *Installing ZeroMQ event loop*.

---

get_event_loop()

Get the event loop.

If current thread is the main thread and there are no registered event loop for current thread then the call creates new event loop and registers it.

Returns Return an instance of *ZmqEventLoop*.

Raises **RuntimeError** if there is no registered event loop for current thread.

new_event_loop()

Create a new event loop.

You must call *ZmqEventLoopPolicy.set_event_loop()* to make this the current event loop.

set_event_loop(*loop*)

Set the event loop.

As a side effect, if a child watcher was set before, then calling `.set_event_loop()` from the main thread will call `asyncio.AbstractChildWatcher.attach_loop()` on the child watcher.

Parameters **loop** – an `asyncio.AbstractEventLoop` instance or *None*

Raises **TypeError** if loop is not instance of `asyncio.AbstractEventLoop`

get_child_watcher()

Get the child watcher

If not yet set, a `asyncio.SafeChildWatcher` object is automatically created.

Returns Return an instance of `asyncio.AbstractChildWatcher`.

set_child_watcher(*watcher*)

Set the child watcher.

Parameters **watcher** – an `asyncio.AbstractChildWatcher` instance or *None*

Raises TypeError if watcher is not instance of `asyncio.AbstractChildWatcher`

8.3.8 ZmqEventLoop

Deprecated since version 0.5: *aiozmq* works with any *asyncio* event loop, it doesn't require dedicated event loop object.

Event loop with *ZeroMQ* support.

Follows `asyncio.AbstractEventLoop` specification and has *create_zmq_connection()* method for *ZeroMQ* sockets layer.

class `aiozmq.`**ZmqEventLoop**(*, *zmq_context=None*)

Parameters **zmq_context** (*zmq.Context*) – explicit context to use for ZeroMQ socket creation inside *ZmqEventLoop.create_zmq_connection()* calls. *aiozmq* shares global context returned by `zmq.Context.instance()` call if *zmq_context* parameter is `None`.

create_zmq_connection(*protocol_factory*, *zmq_type*, *, *bind=None*, *connect=None*, *zmq_sock=None*)

Create a ZeroMQ connection.

If you don't use *bind* or *connect* params you can do it later by *ZmqTransport.bind()* and *ZmqTransport.connect()* calls.

Parameters

- **protocol_factory** (*callable*) – a factory that instantiates *ZmqProtocol* object.
- **zmq_type** (*int*) – a type of *ZeroMQ* socket (*zmq.REQ*, *zmq.REP*, *zmq.PUB*, *zmq.SUB*, zmq.PAIR*, *zmq.DEALER*, *zmq.ROUTER*, *zmq.PULL*, *zmq.PUSH*, etc.)
- **bind** (*str or iterable of strings*) – endpoints specification.

  Every *endpoint* generates call to *ZmqTransport.bind()* for accepting connections from specified endpoint.

  Other side should use *connect* parameter to connect to this transport.
- **connect** (*str or iterable of strings*) – endpoints specification.

  Every *endpoint* generates call to *ZmqTransport.connect()* for connecting transport to specified endpoint.

  Other side should use bind parameter to wait for incoming connections.
- **zmq_sock** (*zmq.Socket*) – a preexisting zmq socket that will be passed to returned transport.

Returns a pair of `(transport, protocol)` where transport supports *ZmqTransport* interface.

Return type `tuple`

8.4 Examples of aiozmq usage

There is a list of examples from aiozmq/examples

Every example is a correct tiny python program.

8.4.1 Simple DEALER-ROUTER pair implemented on Core level

```
import asyncio
import aiozmq
import zmq


class ZmqDealerProtocol(aiozmq.ZmqProtocol):

    transport = None

    def __init__(self, queue, on_close):
        self.queue = queue
        self.on_close = on_close

    def connection_made(self, transport):
        self.transport = transport

    def msg_received(self, msg):
        self.queue.put_nowait(msg)

    def connection_lost(self, exc):
        self.on_close.set_result(exc)


class ZmqRouterProtocol(aiozmq.ZmqProtocol):

    transport = None

    def __init__(self, on_close):
        self.on_close = on_close

    def connection_made(self, transport):
        self.transport = transport

    def msg_received(self, msg):
        self.transport.write(msg)

    def connection_lost(self, exc):
        self.on_close.set_result(exc)


@asyncio.coroutine
def go():
    router_closed = asyncio.Future()
    dealer_closed = asyncio.Future()
    router, _ = yield from aiozmq.create_zmq_connection(
        lambda: ZmqRouterProtocol(router_closed),
        zmq.ROUTER,
        bind='tcp://127.0.0.1:*')

    addr = list(router.bindings())[0]
    queue = asyncio.Queue()
    dealer, _ = yield from aiozmq.create_zmq_connection(
        lambda: ZmqDealerProtocol(queue, dealer_closed),
        zmq.DEALER,
        connect=addr)
```

```
    for i in range(10):
        msg = (b'data', b'ask', str(i).encode('utf-8'))
        dealer.write(msg)
        answer = yield from queue.get()
        print(answer)
    dealer.close()
    yield from dealer_closed
    router.close()
    yield from router_closed


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.2 DEALER-ROUTER pair implemented with streams

```
import asyncio
import aiozmq
import zmq


@asyncio.coroutine
def go():
    router = yield from aiozmq.create_zmq_stream(
        zmq.ROUTER,
        bind='tcp://127.0.0.1:*')

    addr = list(router.transport.bindings())[0]
    dealer = yield from aiozmq.create_zmq_stream(
        zmq.DEALER,
        connect=addr)

    for i in range(10):
        msg = (b'data', b'ask', str(i).encode('utf-8'))
        dealer.write(msg)
        data = yield from router.read()
        router.write(data)
        answer = yield from dealer.read()
        print(answer)
    dealer.close()
    router.close()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.3 Remote Procedure Call

```
import asyncio
import aiozmq.rpc


class ServerHandler(aiozmq.rpc.AttrHandler):

    @aiozmq.rpc.method
    def remote_func(self, a: int, b: int) -> int:
        return a + b


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        ServerHandler(), bind='tcp://*:*')
    server_addr = list(server.transport.bindings())[0]

    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr)

    ret = yield from client.call.remote_func(1, 2)
    assert 3 == ret

    server.close()
    yield from server.wait_closed()
    client.close()
    yield from client.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.4 Pipeline aka Notifier

```
import asyncio
import aiozmq.rpc
from itertools import count


class Handler(aiozmq.rpc.AttrHandler):

    def __init__(self):
        self.connected = False

    @aiozmq.rpc.method
    def remote_func(self, step, a: int, b: int):
        self.connected = True
        print("HANDLER", step, a, b)
```

```
@asyncio.coroutine
def go():
    handler = Handler()
    listener = yield from aiozmq.rpc.serve_pipeline(
        handler, bind='tcp://*:*')
    listener_addr = list(listener.transport.bindings())[0]

    notifier = yield from aiozmq.rpc.connect_pipeline(
        connect=listener_addr)

    for step in count(0):
        yield from notifier.notify.remote_func(step, 1, 2)
        if handler.connected:
            break
        else:
            yield from asyncio.sleep(0.01)

    listener.close()
    yield from listener.wait_closed()
    notifier.close()
    yield from notifier.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.5 Publish-Subscribe

```
import asyncio
import aiozmq.rpc
from itertools import count


class Handler(aiozmq.rpc.AttrHandler):

    def __init__(self):
        self.connected = False

    @aiozmq.rpc.method
    def remote_func(self, step, a: int, b: int):
        self.connected = True
        print("HANDLER", step, a, b)


@asyncio.coroutine
def go():
    handler = Handler()
    subscriber = yield from aiozmq.rpc.serve_pubsub(
        handler, subscribe='topic', bind='tcp://127.0.0.1:*',
        log_exceptions=True)
    subscriber_addr = list(subscriber.transport.bindings())[0]
```

```
    print("SERVE", subscriber_addr)

    publisher = yield from aiozmq.rpc.connect_pubsub(
        connect=subscriber_addr)

    for step in count(0):
        yield from publisher.publish('topic').remote_func(step, 1, 2)
        if handler.connected:
            break
        else:
            yield from asyncio.sleep(0.1)

    subscriber.close()
    yield from subscriber.wait_closed()
    publisher.close()
    yield from publisher.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.6 Translation RPC exceptions back to client

```
import asyncio
import aiozmq.rpc


class CustomError(Exception):

    def __init__(self, val):
        self.val = val
        super().__init__(val)


exc_name = CustomError.__module__+'.'+CustomError.__name__
error_table = {exc_name: CustomError}


class ServerHandler(aiozmq.rpc.AttrHandler):
    @aiozmq.rpc.method
    def remote(self, val):
        raise CustomError(val)


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        ServerHandler(), bind='tcp://*:*')
    server_addr = list(server.transport.bindings())[0]

    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr,
```

```
        error_table=error_table)

    try:
        yield from client.call.remote('value')
    except CustomError as exc:
        exc.val == 'value'

    server.close()
    client.close()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.7 Translation instances of custom classes via RPC

```
import asyncio
import aiozmq.rpc
import msgpack


class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if isinstance(other, Point):
            return (self.x, self.y) == (other.x, other.y)
        return NotImplemented


translation_table = {
    0: (Point,
        lambda value: msgpack.packb((value.x, value.y)),
        lambda binary: Point(*msgpack.unpackb(binary))),
}


class ServerHandler(aiozmq.rpc.AttrHandler):
    @aiozmq.rpc.method
    def remote(self, val):
        return val


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        ServerHandler(), bind='tcp://*:*',
        translation_table=translation_table)
    server_addr = list(server.transport.bindings())[0]
```

```
    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr,
        translation_table=translation_table)

    ret = yield from client.call.remote(Point(1, 2))
    assert ret == Point(1, 2)

    server.close()
    yield from server.wait_closed()
    client.close()
    yield from client.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.8 Validation of RPC methods

```
import asyncio
import aiozmq.rpc


class ServerHandler(aiozmq.rpc.AttrHandler):

    @aiozmq.rpc.method
    def remote_func(self, a: int, b: int) -> int:
        return a + b


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        ServerHandler(), bind='tcp://*:*')
    server_addr = list(server.transport.bindings())[0]

    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr)

    try:
        yield from client.call.unknown_function()
    except aiozmq.rpc.NotFoundError as exc:
        print("client.rpc.unknown_function(): {}".format(exc))

    try:
        yield from client.call.remote_func(bad_arg=1)
    except aiozmq.rpc.ParametersError as exc:
        print("client.rpc.remote_func(bad_arg=1): {}".format(exc))

    try:
        yield from client.call.remote_func(1)
    except aiozmq.rpc.ParametersError as exc:
```

```
        print("client.rpc.remote_func(1): {}".format(exc))

    try:
        yield from client.call.remote_func('a', 'b')
    except aiozmq.rpc.ParametersError as exc:
        print("client.rpc.remote_func('a', 'b'): {}".format(exc))

    server.close()
    yield from server.wait_closed()
    client.close()
    yield from client.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.9 RPC lookup in nested namespaces

```
import asyncio
import aiozmq.rpc


class Handler(aiozmq.rpc.AttrHandler):

    def __init__(self, ident):
        self.ident = ident
        self.subhandler = SubHandler(self.ident, 'subident')

    @aiozmq.rpc.method
    def a(self):
        return (self.ident, 'a')


class SubHandler(aiozmq.rpc.AttrHandler):

    def __init__(self, ident, subident):
        self.ident = ident
        self.subident = subident

    @aiozmq.rpc.method
    def b(self):
        return (self.ident, self.subident, 'b')


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        Handler('ident'), bind='tcp://*:*')
    server_addr = list(server.transport.bindings())[0]

    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr)
```

```
    ret = yield from client.call.a()
    assert ('ident', 'a') == ret

    ret = yield from client.call.subhandler.b()
    assert ('ident', 'subident', 'b') == ret

    server.close()
    yield from server.wait_closed()
    client.close()
    yield from client.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.10 Use dict as RPC lookup table

```
import asyncio
import aiozmq.rpc


@aiozmq.rpc.method
def a():
    return 'a'


@aiozmq.rpc.method
def b():
    return 'b'


handlers_dict = {'a': a,
                 'subnamespace': {'b': b}}


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        handlers_dict, bind='tcp://*:*')
    server_addr = list(server.transport.bindings())[0]

    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr)

    ret = yield from client.call.a()
    assert 'a' == ret

    ret = yield from client.call.subnamespace.b()
    assert 'b' == ret

    server.close()
```

```
    yield from server.wait_closed()
    client.close()
    yield from client.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.11 Use dynamic RPC lookup

```
import asyncio
import aiozmq.rpc


class DynamicHandler(aiozmq.rpc.AttrHandler):

    def __init__(self, namespace=()):
        self.namespace = namespace

    def __getitem__(self, key):
        try:
            return getattr(self, key)
        except AttributeError:
            return DynamicHandler(self.namespace + (key,))

    @aiozmq.rpc.method
    def func(self):
        return (self.namespace, 'val')


@asyncio.coroutine
def go():
    server = yield from aiozmq.rpc.serve_rpc(
        DynamicHandler(), bind='tcp://*:*')
    server_addr = list(server.transport.bindings())[0]

    client = yield from aiozmq.rpc.connect_rpc(
        connect=server_addr)

    ret = yield from client.call.func()
    assert ((), 'val') == ret, ret

    ret = yield from client.call.a.func()
    assert (('a',), 'val') == ret, ret

    ret = yield from client.call.a.b.func()
    assert (('a', 'b'), 'val') == ret, ret

    server.close()
    yield from server.wait_closed()
    client.close()
```

```
    yield from client.wait_closed()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.4.12 Socket event monitor

```
'''
This example demonstrates how to use the ZMQ socket monitor to receive
socket events.

The socket event monitor capability requires libzmq >= 4 and pyzmq >= 14.4.

'''

import asyncio
import aiozmq
import zmq


ZMQ_EVENTS = {
    getattr(zmq, name): name.replace('EVENT_', '').lower().replace('_', ' ')
    for name in [i for i in dir(zmq) if i.startswith('EVENT_')]}


def event_description(event):
    ''' Return a human readable description of the event '''
    return ZMQ_EVENTS.get(event, 'unknown')


class Protocol(aiozmq.ZmqProtocol):

    def __init__(self):
        self.wait_ready = asyncio.Future()
        self.wait_done = asyncio.Future()
        self.wait_closed = asyncio.Future()
        self.count = 0

    def connection_made(self, transport):
        self.transport = transport
        self.wait_ready.set_result(True)

    def connection_lost(self, exc):
        self.wait_closed.set_result(exc)

    def msg_received(self, data):
        # This protocol is used by both the Router and Dealer sockets in
        # this example. Router sockets prefix messages with the identity
        # of the sender and hence contain two frames in this simple test
        # protocol.
        if len(data) == 2:
```

```
            identity, msg = data
            assert msg == b'Hello'
            self.transport.write([identity, b'World'])
        else:
            msg = data[0]
            assert msg == b'World'
            self.count += 1
            if self.count >= 4:
                self.wait_done.set_result(True)

    def event_received(self, event):
        print(
            'event:{}, value:{}, endpoint:{}, description:{}'.format(
                event.event, event.value, event.endpoint,
                event_description(event.event)))


@asyncio.coroutine
def go():

    st, sp = yield from aiozmq.create_zmq_connection(
        Protocol, zmq.ROUTER, bind='tcp://127.0.0.1:*')
    yield from sp.wait_ready
    addr = list(st.bindings())[0]

    ct, cp = yield from aiozmq.create_zmq_connection(
        Protocol, zmq.DEALER, connect=addr)
    yield from cp.wait_ready

    # Enable the socket monitor on the client socket. Socket events
    # are passed to the 'event_received' method on the client protocol.
    yield from ct.enable_monitor()

    # Trigger some socket events while also sending a message to the
    # server. When the client protocol receives 4 response it will
    # fire the wait_done future.
    for i in range(4):
        yield from asyncio.sleep(0.1)
        yield from ct.disconnect(addr)
        yield from asyncio.sleep(0.1)
        yield from ct.connect(addr)
        yield from asyncio.sleep(0.1)
        ct.write([b'Hello'])

    yield from cp.wait_done

    # The socket monitor can be explicitly disabled if necessary.
    # yield from ct.disable_monitor()

    # If a socket monitor is left enabled on a socket being closed,
    # the socket monitor will be closed automatically.
    ct.close()
    yield from cp.wait_closed

    st.close()
    yield from sp.wait_closed
```

```
def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


if __name__ == '__main__':
    # import logging
    # logging.basicConfig(level=logging.DEBUG)

    if (zmq.zmq_version_info() < (4,) or
            zmq.pyzmq_version_info() < (14, 4,)):
        raise NotImplementedError(
            "Socket monitor requires libzmq >= 4 and pyzmq >= 14.4, "
            "have libzmq:{}, pyzmq:{}".format(
                zmq.zmq_version(), zmq.pyzmq_version()))

    main()
```

8.4.13 Stream socket event monitor

```
import asyncio
import aiozmq
import zmq


@asyncio.coroutine
def monitor_stream(stream):
    try:
        while True:
            event = yield from stream.read_event()
            print(event)
    except aiozmq.ZmqStreamClosed:
        pass


@asyncio.coroutine
def go():
    router = yield from aiozmq.create_zmq_stream(
        zmq.ROUTER,
        bind='tcp://127.0.0.1:*')
    addr = list(router.transport.bindings())[0]

    dealer = yield from aiozmq.create_zmq_stream(
        zmq.DEALER)

    yield from dealer.transport.enable_monitor()

    asyncio.Task(monitor_stream(dealer))

    yield from dealer.transport.connect(addr)

    for i in range(10):
        msg = (b'data', b'ask', str(i).encode('utf-8'))
        dealer.write(msg)
        data = yield from router.read()
        router.write(data)
        answer = yield from dealer.read()
```

```
        print(answer)

    router.close()
    dealer.close()


def main():
    asyncio.get_event_loop().run_until_complete(go())
    print("DONE")


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

8.5 Glossary

annotaion Additional value that can be bound to any function argument and return value.

See **PEP 3107**.

asyncio Reference implementation of **PEP 3156**

See https://pypi.python.org/pypi/asyncio/

callable Any object that can be called. Use `callable()` to check that.

endpoint A string consisting of two parts as follows: *transport://address*.

The transport part specifies the underlying transport protocol to use. The meaning of the address part is specific to the underlying transport protocol selected.

The following transports are defined:

inproc local in-process (inter-thread) communication transport, see http://api.zeromq.org/master:zmq-inproc.

ipc local inter-process communication transport, see http://api.zeromq.org/master:zmq-ipc

tcp unicast transport using TCP, see http://api.zeromq.org/master:zmq_tcp

pgm, epgm reliable multicast transport using PGM, see http://api.zeromq.org/master:zmq_pgm

enduser Software engeneer who wants to *just use* human-like communications via that library.

We offer that simple API for RPC, Push/Pull and Pub/Sub services.

msgpack Fast and compact binary serialization format.

See http://msgpack.org/ for standard description. https://pypi.python.org/pypi/msgpack-python/ is Python implementation.

pyzmq PyZMQ is the Python bindings for *ZeroMQ*.

See https://github.com/zeromq/pyzmq

trafaret Trafaret is a validation library with support for data structure convertors.

See https://github.com/Deepwalker/trafaret

ZeroMQ ØMQ (also spelled ZeroMQ, 0MQ or ZMQ) is a high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ØMQ system can run without a dedicated message broker. The library is designed to have a familiar socket-style API.

See http://zeromq.org/

a

Symbols

A

B

C

D

E

F

G

L

M

N