PyZMQ Documentation

Release 26.0.0

Brian E. Granger **Min Ragan-Kelley**

Apr 15, 2024

CONTENTS

PyZMQ is the Python bindings for ØMQ. This documentation currently contains notes on some important aspects of developing PyZMQ and an overview of what the ØMQ API looks like in Python. For information on how to use ØMQ in general, see the many examples in the excellent ØMQ Guide, all of which have a version in Python.

PyZMQ works with Python 3 (3.7), as well as PyPy via CFFI.

Please don't hesitate to report pyzmq-specific issues to our tracker on GitHub. General questions about ØMQ are better sent to the ØMQ tracker or mailing list.

CHAPTER

ONE

SUPPORTED LIBZMQ

PyZMQ aims to support all stable (3.2.2, 4.0.1) versions of libzmq. Building the same pyzmq against various versions of libzmq is supported, but only the functionality of the linked libzmq will be available.

Note: libzmq 3.0-3.1 are not supported, as they never received a stable release.

Binary distributions (wheels on PyPI) of PyZMQ ship with the stable version of libzmq at the time of release, built with default configuration, and include CURVE support provided by libsodium. For pyzmq-26.0.0, this is 4.3.5.

CHAPTER

TWO

USING PYZMQ

To get started with ZeroMQ, read the ZeroMQ guide, which has every example implemented using PyZMQ.

You can also check out the examples in the pyzmq repo.

2.1 The PyZMQ API

2.1.1 zmq

Python bindings for 0MQ

Basic Classes

Note: For typing purposes, `zmq.Context` and `zmq.Socket` are Generics, which means they will accept any Context or Socket implementation.

The base `zmq.Context()` constructor returns the type `zmq.Context[zmq.Socket[bytes]]`. If you are using type annotations and want to *exclude* the async subclasses, use the resolved types instead of the base Generics:

```
ctx: zmq.Context[zmq.Socket[bytes]] = zmq.Context()
sock: zmq.Socket[bytes]
```

in pyzmq 26, these are available as the Type Aliases (not actual classes!):

```
ctx: zmq.SyncContext = zmq.Context()
sock: zmq.SyncSocket
```

Context

class zmq.**Context**(*io_threads: int = 1*)

class zmq.**Context**(*io_threads:* Context)

class zmq.**Context**(**, shadow:* Context | *int*)

Create a zmq Context

A zmq Context creates sockets via its `ctx.socket` method.

Changed in version 24: When using a Context as a context manager (`with zmq.Context()`), or deleting a context without closing it first, `ctx.destroy()` is called, closing any leftover sockets, instead of `ctx.term()` which requires sockets to be closed first.

This prevents hangs caused by `ctx.term()` if sockets are left open, but means that unclean destruction of contexts (with sockets left open) is not safe if sockets are managed in other threads.

New in version 25: Contexts can now be shadowed by passing another Context. This helps in creating an async copy of a sync context or vice versa:

```
ctx = zmq.Context(async_ctx)
```

Which previously had to be:

```
ctx = zmq.Context.shadow(async_ctx.underlying)
```

closed

boolean - whether the context has been terminated. If True, you can no longer use this Context.

destroy(*linger: int | None = None*) → None

Close all sockets associated with this context and then terminate the context.

> **Warning:** destroy involves calling *Socket.close()*, which is **NOT** threadsafe. If there are active sockets in other threads, this must not be called.

Parameters

linger (*int, optional*) – If specified, set LINGER on sockets prior to closing them.

get(*option: int*)

Get the value of a context option.

See the 0MQ API documentation for zmq_ctx_get for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

Parameters

option (*int*) – The option to get. Available values will depend on your version of libzmq. Examples include:

```
zmq.IO_THREADS, zmq.MAX_SOCKETS
```

Returns

optval – The value of the option as an integer.

Return type

int

getsockopt(*opt: int*) → str | bytes | int

get default socket options for new sockets created by this Context

New in version 13.0.

classmethod instance(*io_threads: int = 1*) → *zmq.Context*

Returns a global Context instance.

Most single-process applications have a single, global Context. Use this method instead of passing around Context instances throughout your code.

A common pattern for classes that depend on Contexts is to use a default argument to enable programs with multiple Contexts but not require the argument for simpler applications:

```
class MyClass(object):
    def __init__(self, context=None):
        self.context = context or Context.instance()
```

Changed in version 18.1: When called in a subprocess after forking, a new global instance is created instead of inheriting a Context that won't work from the parent process.

set(*option: int*, *optval*)

Set a context option.

See the 0MQ API documentation for zmq_ctx_set for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

Parameters

- **option** (*int*) – The option to set. Available values will depend on your version of libzmq. Examples include:

  ```
  zmq.IO_THREADS, zmq.MAX_SOCKETS
  ```

- **optval** (*int*) – The value of the option to set.

setsockopt(*opt: int*, *value: Any*) → None

set default socket options for new sockets created by this Context

New in version 13.0.

classmethod shadow(*address: int | Context*) → *zmq.Context*

Shadow an existing libzmq context

address is a zmq.Context or an integer (or FFI pointer) representing the address of the libzmq context.

New in version 14.1.

New in version 25: Support for shadowing `zmq.Context` objects, instead of just integer addresses.

classmethod shadow_pyczmq(*ctx: Any*) → *zmq.Context*

Shadow an existing pyczmq context

ctx is the FFI `zctx_t *` pointer

New in version 14.1.

socket(*socket_type: int*, *socket_class: Callable[[zmq.Context, int], zmq.Socket] | None = None*, ***kwargs: Any*) → *zmq.Socket*

Create a Socket associated with this Context.

Parameters

- **socket_type** (*int*) – The socket type, which can be any of the 0MQ socket types: REQ, REP, PUB, SUB, PAIR, DEALER, ROUTER, PULL, PUSH, etc.

- **socket_class** (zmq.Socket) – The socket class to instantiate, if different from the default for this Context. e.g. for creating an asyncio socket attached to a default Context or vice versa.

  New in version 25.

- **kwargs** – will be passed to the __init__ method of the socket class.

term() → None

Close or terminate the context.

Context termination is performed in the following steps:

- Any blocking operations currently in progress on sockets open within context shall raise *zmq.ContextTerminated*. With the exception of socket.close(), any further operations on sockets open within this context shall raise *zmq.ContextTerminated*.
- **After interrupting all blocking calls, term shall block until the following conditions are satisfied:**
  - All sockets open within context have been closed.
  - For each socket within context, all messages sent on the socket have either been physically transferred to a network peer, or the socket's linger period set with the zmq.LINGER socket option has expired.

For further details regarding socket linger behaviour refer to libzmq documentation for ZMQ_LINGER.

This can be called to close the context by hand. If this is not called, the context will automatically be closed when it is garbage collected, in which case you may see a ResourceWarning about the unclosed context.

underlying

The address of the underlying libzmq context

Socket

class zmq.**Socket**(*ctx_or_socket: Context*, *socket_type: int*, *, *copy_threshold: int | None = None*)

class zmq.**Socket**(*, *shadow: Socket | int*, *copy_threshold: int | None = None*)

class zmq.**Socket**(*ctx_or_socket: Socket*)

The ZMQ socket object

To create a Socket, first create a Context:

```
ctx = zmq.Context.instance()
```

then call `ctx.socket(socket_type)`:

```
s = ctx.socket(zmq.ROUTER)
```

New in version 25: Sockets can now be shadowed by passing another Socket. This helps in creating an async copy of a sync socket or vice versa:

```
s = zmq.Socket(async_socket)
```

Which previously had to be:

```
s = zmq.Socket.shadow(async_socket.underlying)
```

`closed`

boolean - whether the socket has been closed. If True, you can no longer use this Socket.

`copy_threshold`

integer - size (in bytes) below which messages should always be copied. Zero-copy support has nontrivial overhead due to the need to coordinate garbage collection with the libzmq IO thread, so sending small messages (typically < 10s of kB) with `copy=False` is often more expensive than with `copy=True`. The initial default value is 65536 (64kB), a reasonable default based on testing.

Defaults to *zmq.COPY_THRESHOLD* on socket construction. Setting *zmq.COPY_THRESHOLD* will define the default value for any subsequently created sockets.

New in version 17.

`bind`(*addr*)

Bind the socket to an address.

This causes the socket to listen on a network port. Sockets on the other side of this connection will use `Socket.connect(addr)` to connect to this socket.

Returns a context manager which will call unbind on exit.

New in version 20.0: Can be used as a context manager.

New in version 26.0: binding to port 0 can be used as a context manager for binding to a random port. The URL can be retrieved as `socket.last_endpoint`.

Parameters

addr (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported include tcp, udp, pgm, epgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

`bind_to_random_port`(*addr: str*, *min_port: int = 49152*, *max_port: int = 65536*, *max_tries: int = 100*) → int

Bind this socket to a random port in a range.

If the port range is unspecified, the system will choose the port.

Parameters

- **addr** (*str*) – The address string without the port to pass to `Socket.bind()`.
- **min_port** (*int, optional*) – The minimum port in the range of ports to try (inclusive).
- **max_port** (*int, optional*) – The maximum port in the range of ports to try (exclusive).
- **max_tries** (*int, optional*) – The maximum number of bind attempts to make.

Returns

port – The port the socket was bound to.

Return type

int

Raises

ZMQBindError – if `max_tries` reached before successful bind

`close`(*linger=None*) → None

Close the socket.

If linger is specified, LINGER sockopt will be set prior to closing.

Note: closing a zmq Socket may not close the underlying sockets if there are undelivered messages. Only after all messages are delivered or discarded by reaching the socket's LINGER timeout (default: forever) will the underlying sockets be closed.

This can be called to close the socket by hand. If this is not called, the socket will automatically be closed when it is garbage collected, in which case you may see a ResourceWarning about the unclosed socket.

connect(*addr*)

Connect to a remote 0MQ socket.

Returns a context manager which will call disconnect on exit.

New in version 20.0: Can be used as a context manager.

Parameters
addr (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported are tcp, udp, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

disable_monitor() → None

Shutdown the PAIR socket (created using get_monitor_socket) that is serving socket events.

New in version 14.4.

disconnect(*addr*)

Disconnect from a remote 0MQ socket (undoes a call to connect).

New in version libzmq-3.2.

New in version 13.0.

Parameters
addr (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported are tcp, udp, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

fileno() → int

Return edge-triggered file descriptor for this socket.

This is a read-only edge-triggered file descriptor for both read and write events on this socket. It is important that all available events be consumed when an event is detected, otherwise the read event will not trigger again.

New in version 17.0.

get(*option: int*)

Get the value of a socket option.

See the 0MQ API documentation for details on specific options.

Parameters
option (*int*) – The option to get. Available values will depend on your version of libzmq. Examples include:

```
zmq.IDENTITY, HWM, LINGER, FD, EVENTS
```

Returns
optval – The value of the option as a bytestring or int.

Return type
int or bytes

get_hwm() → int

Get the High Water Mark.

On libzmq 3, this gets SNDHWM if available, otherwise RCVHWM

get_monitor_socket(*events: int | None = None*, *addr: str | None = None*) → *zmq.Socket*

Return a connected PAIR socket ready to receive the event notifications.

New in version libzmq-4.0.

New in version 14.0.

Parameters

- **events** (*int*) – default: zmq.EVENT_ALL The bitmask defining which events are wanted.
- **addr** (*str*) – The optional endpoint for the monitoring sockets.

Returns
socket – The PAIR socket, connected and ready to receive messages.

Return type
zmq.Socket

get_string(*option: int*, *encoding='utf-8'*) → str

Get the value of a socket option.

See the 0MQ documentation for details on specific options.

Parameters
option (*int*) – The option to retrieve.

Returns
optval – The value of the option as a unicode string.

Return type
str

getsockopt(*option: int*)

Get the value of a socket option.

See the 0MQ API documentation for details on specific options.

Parameters
option (*int*) – The option to get. Available values will depend on your version of libzmq. Examples include:

```
zmq.IDENTITY, HWM, LINGER, FD, EVENTS
```

Returns
optval – The value of the option as a bytestring or int.

Return type
int or bytes

getsockopt_string(*option: int*, *encoding='utf-8'*) → str

Get the value of a socket option.

See the 0MQ documentation for details on specific options.

Parameters
option (*int*) – The option to retrieve.

Returns
optval – The value of the option as a unicode string.

Return type
str

property hwm: int

Property for High Water Mark.

Setting hwm sets both SNDHWM and RCVHWM as appropriate. It gets SNDHWM if available, otherwise RCVHWM.

join(*group*)

Join a RADIO-DISH group

Only for DISH sockets.

libzmq and pyzmq must have been built with ZMQ_BUILD_DRAFT_API

New in version 17.

leave(*group*)

Leave a RADIO-DISH group

Only for DISH sockets.

libzmq and pyzmq must have been built with ZMQ_BUILD_DRAFT_API

New in version 17.

monitor(*addr*, *events: int = 65535*)

Start publishing socket events on inproc. See libzmq docs for zmq_monitor for details.

While this function is available from libzmq 3.2, pyzmq cannot parse monitor messages from libzmq prior to 4.0.

Parameters

- **addr** (*str*) – The inproc url used for monitoring. Passing None as the addr will cause an existing socket monitor to be deregistered.
- **events** (*int*) – default: zmq.EVENT_ALL The zmq event bitmask for which events will be sent to the monitor.

poll(*timeout: int | None = None*, *flags: int = PollEvent.POLLIN*) → int

Poll the socket for events.

See *Poller* to wait for multiple sockets at once.

Parameters

- **timeout** (*int*) – The timeout (in milliseconds) to wait for an event. If unspecified (or specified None), will wait forever for an event.
- **flags** (*int*) – default: POLLIN. POLLIN, POLLOUT, or POLLIN|POLLOUT. The event flags to poll for.

Returns
event_mask – The poll event mask (POLLIN, POLLOUT), 0 if the timeout was reached without an event.

Return type
int

recv(*flags=0*, *copy: bool = True*, *track: bool = False*)

Receive a message.

With flags=NOBLOCK, this raises *ZMQError* if no messages have arrived; otherwise, this waits until a message arrives. See *Poller* for more general non-blocking I/O.

Parameters

- **flags** (*int*) – 0 or NOBLOCK.
- **copy** (*bool*) – Should the message be received in a copying or non-copying manner? If False a Frame object is returned, if True a string copy of message is returned.
- **track** (*bool*) – Should the message be tracked for notification that ZMQ has finished with it? (ignored if copy=True)

Returns

msg – The received message frame. If `copy` is False, then it will be a Frame, otherwise it will be bytes.

Return type

bytes or *Frame*

Raises

ZMQError – for any of the reasons zmq_msg_recv might fail (including if NOBLOCK is set and no new messages have arrived).

recv_json(*flags: int = 0*, ***kwargs*) → list | str | int | float | dict

Receive a Python object as a message using json to serialize.

Keyword arguments are passed on to json.loads

Parameters

flags (*int*) – Any valid flags for *Socket.recv()*.

Returns

obj – The Python object that arrives as a message.

Return type

Python object

Raises

ZMQError – for any of the reasons *recv()* might fail

recv_multipart(*flags: int = 0*, ***, *copy: Literal[True]*, *track: bool = False*) → list[bytes]

recv_multipart(*flags: int = 0*, ***, *copy: Literal[False]*, *track: bool = False*) → list[*Frame*]

recv_multipart(*flags: int = 0*, ***, *track: bool = False*) → list[bytes]

recv_multipart(*flags: int = 0*, *copy: bool = True*, *track: bool = False*) → list[*Frame*] | list[bytes]

Receive a multipart message as a list of bytes or Frame objects

Parameters

- **flags** (*int, optional*) – Any valid flags for *Socket.recv()*.
- **copy** (*bool, optional*) – Should the message frame(s) be received in a copying or non-copying manner? If False a Frame object is returned for each part, if True a copy of the bytes is made for each frame.
- **track** (*bool, optional*) – Should the message frame(s) be tracked for notification that ZMQ has finished with it? (ignored if copy=True)

Returns
msg_parts – A list of frames in the multipart message; either Frames or bytes, depending on `copy`.

Return type
list

Raises
ZMQError – for any of the reasons *recv()* might fail

recv_pyobj(*flags: int = 0*) → Any

Receive a Python object as a message using pickle to serialize.

Parameters
flags (*int*) – Any valid flags for *Socket.recv()*.

Returns
obj – The Python object that arrives as a message.

Return type
Python object

Raises
ZMQError – for any of the reasons *recv()* might fail

recv_serialized(*deserialize*, *flags=0*, *copy=True*)

Receive a message with a custom deserialization function.

New in version 17.

Parameters

- **deserialize** (*callable*) – The deserialization function to use. deserialize will be called with one argument: the list of frames returned by recv_multipart() and can return any object.
- **flags** (*int, optional*) – Any valid flags for *Socket.recv()*.
- **copy** (*bool, optional*) – Whether to recv bytes or Frame objects.

Returns
obj – The object returned by the deserialization function.

Return type
object

Raises
ZMQError – for any of the reasons *recv()* might fail

recv_string(*flags: int = 0*, *encoding: str = 'utf-8'*) → str

Receive a unicode string, as sent by send_string.

Parameters

- **flags** (*int*) – Any valid flags for *Socket.recv()*.
- **encoding** (*str*) – The encoding to be used

Returns
s – The Python unicode string that arrives as encoded bytes.

Return type
str

Raises

ZMQError – for any of the reasons *Socket.recv()* might fail

send(*data: Any*, *flags: int = 0*, *copy: bool = True*, ***, *track: Literal[True]*, *routing_id: int | None = None*, *group: str | None = None*) → *MessageTracker*

send(*data: Any*, *flags: int = 0*, *copy: bool = True*, ***, *track: Literal[False]*, *routing_id: int | None = None*, *group: str | None = None*) → None

send(*data: Any*, *flags: int = 0*, ***, *copy: bool = True*, *routing_id: int | None = None*, *group: str | None = None*) → None

send(*data: Any*, *flags: int = 0*, *copy: bool = True*, *track: bool = False*, *routing_id: int | None = None*, *group: str | None = None*) → *MessageTracker* | None

Send a single zmq message frame on this socket.

This queues the message to be sent by the IO thread at a later time.

With flags=NOBLOCK, this raises *ZMQError* if the queue is full; otherwise, this waits until space is available. See *Poller* for more general non-blocking I/O.

Parameters

- **data** (*bytes, Frame, memoryview*) – The content of the message. This can be any object that provides the Python buffer API (i.e. `memoryview(data)` can be called).
- **flags** (*int*) – 0, NOBLOCK, SNDMORE, or NOBLOCK|SNDMORE.
- **copy** (*bool*) – Should the message be sent in a copying or non-copying manner.
- **track** (*bool*) – Should the message be tracked for notification that ZMQ has finished with it? (ignored if copy=True)
- **routing_id** (*int*) – For use with SERVER sockets
- **group** (*str*) – For use with RADIO sockets

Returns

- **None** (if `copy` or not track) – None if message was sent, raises an exception otherwise.
- **MessageTracker** (*if track and not copy*) – a MessageTracker object, whose `done` property will be False until the send is completed.

Raises

- **TypeError** – If a unicode object is passed
- **ValueError** – If `track=True`, but an untracked Frame is passed.
- **ZMQError** – If the send does not succeed for any reason (including if NOBLOCK is set and the outgoing queue is full).

Changed in version 17.0: DRAFT support for routing_id and group arguments.

send_json(*obj: Any*, *flags: int = 0*, ***kwargs*) → None

Send a Python object as a message using json to serialize.

Keyword arguments are passed on to json.dumps

Parameters

- **obj** (*Python object*) – The Python object to send
- **flags** (*int*) – Any valid flags for *Socket.send()*

send_multipart(*msg_parts: Sequence*, *flags: int = 0*, *copy: bool = True*, *track: bool = False*, ***kwargs*)

Send a sequence of buffers as a multipart message.

The zmq.SNDMORE flag is added to all msg parts before the last.

Parameters

- **msg_parts** (*iterable*) – A sequence of objects to send as a multipart message. Each element can be any sendable object (Frame, bytes, buffer-providers)
- **flags** (*int, optional*) – Any valid flags for *Socket.send()*. SNDMORE is added automatically for frames before the last.
- **copy** (*bool, optional*) – Should the frame(s) be sent in a copying or non-copying manner. If copy=False, frames smaller than self.copy_threshold bytes will be copied anyway.
- **track** (*bool, optional*) – Should the frame(s) be tracked for notification that ZMQ has finished with it (ignored if copy=True).

Returns

- **None** (*if copy or not track*)
- **MessageTracker** (*if track and not copy*) – a MessageTracker object, whose done property will be False until the last send is completed.

send_pyobj(*obj: Any*, *flags: int = 0*, *protocol: int = 4*, ***kwargs*) → *Frame* | None

Send a Python object as a message using pickle to serialize.

Parameters

- **obj** (*Python object*) – The Python object to send.
- **flags** (*int*) – Any valid flags for *Socket.send()*.
- **protocol** (*int*) – The pickle protocol number to use. The default is pickle.DEFAULT_PROTOCOL where defined, and pickle.HIGHEST_PROTOCOL elsewhere.

send_serialized(*msg*, *serialize*, *flags=0*, *copy=True*, ***kwargs*)

Send a message with a custom serialization function.

New in version 17.

Parameters

- **msg** (The message to be sent. Can be any object serializable by `serialize`.) –
- **serialize** (*callable*) – The serialization function to use. serialize(msg) should return an iterable of sendable message frames (e.g. bytes objects), which will be passed to send_multipart.
- **flags** (*int, optional*) – Any valid flags for *Socket.send()*.
- **copy** (*bool, optional*) – Whether to copy the frames.

send_string(*u: str*, *flags: int = 0*, *copy: bool = True*, *encoding: str = 'utf-8'*, ***kwargs*) → *Frame* | None

Send a Python unicode string as a message with an encoding.

0MQ communicates with raw bytes, so you must encode/decode text (str) around 0MQ.

Parameters

- **u** (*str*) – The unicode string to send.
- **flags** (*int, optional*) – Any valid flags for *Socket.send()*.

- **encoding** (*str*) – The encoding to be used

set(*option: int*, *optval*)

Set socket options.

See the 0MQ API documentation for details on specific options.

Parameters

- **option** (*int*) – The option to set. Available values will depend on your version of libzmq. Examples include:

```
zmq.SUBSCRIBE, UNSUBSCRIBE, IDENTITY, HWM, LINGER, FD
```

- **optval** (*int or bytes*) – The value of the option to set.

Notes

Warning: All options other than zmq.SUBSCRIBE, zmq.UNSUBSCRIBE and zmq.LINGER only take effect for subsequent socket bind/connects.

set_hwm(*value: int*) → None

Set the High Water Mark.

On libzmq 3, this sets both SNDHWM and RCVHWM

Warning: New values only take effect for subsequent socket bind/connects.

set_string(*option: int*, *optval: str*, *encoding='utf-8'*) → None

Set socket options with a unicode object.

This is simply a wrapper for setsockopt to protect from encoding ambiguity.

See the 0MQ documentation for details on specific options.

Parameters

- **option** (*int*) – The name of the option to set. Can be any of: SUBSCRIBE, UNSUBSCRIBE, IDENTITY
- **optval** (*str*) – The value of the option to set.
- **encoding** (*str*) – The encoding to be used, default is utf8

setsockopt(*option: int*, *optval*)

Set socket options.

See the 0MQ API documentation for details on specific options.

Parameters

- **option** (*int*) – The option to set. Available values will depend on your version of libzmq. Examples include:

```
zmq.SUBSCRIBE, UNSUBSCRIBE, IDENTITY, HWM, LINGER, FD
```

- **optval** (*int or bytes*) – The value of the option to set.

Notes

> **Warning:** All options other than zmq.SUBSCRIBE, zmq.UNSUBSCRIBE and zmq.LINGER only take effect for subsequent socket bind/connects.

setsockopt_string(*option: int*, *optval: str*, *encoding='utf-8'*) → None

Set socket options with a unicode object.

This is simply a wrapper for setsockopt to protect from encoding ambiguity.

See the 0MQ documentation for details on specific options.

Parameters

- **option** (*int*) – The name of the option to set. Can be any of: SUBSCRIBE, UNSUBSCRIBE, IDENTITY
- **optval** (*str*) – The value of the option to set.
- **encoding** (*str*) – The encoding to be used, default is utf8

classmethod shadow(*address: int | Socket*) → *zmq.Socket*

Shadow an existing libzmq socket

address is a zmq.Socket or an integer (or FFI pointer) representing the address of the libzmq socket.

New in version 14.1.

New in version 25: Support for shadowing `zmq.Socket` objects, instead of just integer addresses.

subscribe(*topic: str | bytes*) → None

Subscribe to a topic

Only for SUB sockets.

New in version 15.3.

unbind(*addr*)

Unbind from an address (undoes a call to bind).

New in version libzmq-3.2.

New in version 13.0.

Parameters

addr (*str*) – The address string. This has the form 'protocol://interface:port', for example 'tcp://127.0.0.1:5555'. Protocols supported are tcp, udp, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

underlying

The address of the underlying libzmq socket

unsubscribe(*topic: str | bytes*) → None

Unsubscribe from a topic

Only for SUB sockets.

New in version 15.3.

Frame

class zmq.**Frame**

A zmq message Frame class for non-copying send/recvs and access to message properties.

A `zmq.Frame` wraps an underlying `zmq_msg_t`.

Message *properties* can be accessed by treating a Frame like a dictionary (`frame["User-Id"]`).

New in version 14.4,: libzmq 4

Frames created by `recv(copy=False)` can be used to access message properties and attributes, such as the CURVE User-Id.

For example:

```
frames = socket.recv_multipart(copy=False)
user_id = frames[0]["User-Id"]
```

This class is used if you want to do non-copying send and recvs. When you pass a chunk of bytes to this class, e.g. `Frame(buf)`, the ref-count of `buf` is increased by two: once because the Frame saves `buf` as an instance attribute and another because a ZMQ message is created that points to the buffer of `buf`. This second ref-count increase makes sure that `buf` lives until all messages that use it have been sent. Once 0MQ sends all the messages and it doesn't need the buffer of `buf`, 0MQ will call `Py_DECREF(s)`.

Parameters

- **data** (*object, optional*) – any object that provides the buffer interface will be used to construct the 0MQ message data.
- **track** (*bool*) – whether a *MessageTracker* should be created to track this object. Tracking a message has a cost at creation, because it creates a threadsafe Event object.
- **copy** (*bool*) – default: use copy_threshold Whether to create a copy of the data to pass to libzmq or share the memory with libzmq. If unspecified, copy_threshold is used.
- **copy_threshold** (*int*) – default: *zmq.COPY_THRESHOLD* If copy is unspecified, messages smaller than this many bytes will be copied and messages larger than this will be shared with libzmq.

buffer

A memoryview of the message contents.

bytes

The message content as a Python bytes object.

The first time this property is accessed, a copy of the message contents is made. From then on that same copy of the message is returned.

get(*option*)

Get a Frame option or property.

See the 0MQ API documentation for zmq_msg_get and zmq_msg_gets for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

Changed in version 14.3: add support for zmq_msg_gets (requires libzmq-4.1) All message properties are strings.

Changed in version 17.0: Added support for `routing_id` and `group`. Only available if draft API is enabled with libzmq >= 4.2.

property group

The RADIO-DISH group of the message.

Requires libzmq >= 4.2 and pyzmq built with draft APIs enabled.

New in version 17.

property routing_id

The CLIENT-SERVER routing id of the message.

Requires libzmq >= 4.2 and pyzmq built with draft APIs enabled.

New in version 17.

set(*option*, *value*)

Set a Frame option.

See the 0MQ API documentation for zmq_msg_set for details on specific options.

New in version libzmq-3.2.

New in version 13.0.

Changed in version 17.0: Added support for `routing_id` and `group`. Only available if draft API is enabled with libzmq >= 4.2.

MessageTracker

class zmq.**MessageTracker**(**towatch: tuple[*MessageTracker | Event | Frame*]*)

A class for tracking if 0MQ is done using one or more messages.

When you send a 0MQ message, it is not sent immediately. The 0MQ IO thread sends the message at some later time. Often you want to know when 0MQ has actually sent the message though. This is complicated by the fact that a single 0MQ message can be sent multiple times using different sockets. This class allows you to track all of the 0MQ usages of a message.

Parameters

towatch (Event, MessageTracker, zmq.Frame) – This objects to track. This class can track the low-level Events used by the Message class, other MessageTrackers or actual Messages.

property done

Is 0MQ completely done with the message(s) being tracked?

wait(*timeout: float | int = -1*)

Wait for 0MQ to be done with the message or until `timeout`.

Parameters

timeout (*float*) – default: -1, which means wait forever. Maximum time in (s) to wait before raising NotDone.

Returns

if done before `timeout`

Return type

None

Raises

NotDone – if `timeout` reached before I am done.

Polling

Poller

class zmq.**Poller**

A stateful poll interface that mirrors Python's built-in poll.

modify(*socket*, *flags=PollEvent.POLLIN | POLLOUT*)

Modify the flags for an already registered 0MQ socket or native fd.

poll(*timeout: int | None = None*) → list[tuple[Any, int]]

Poll the registered 0MQ or native fds for I/O.

If there are currently events ready to be processed, this function will return immediately. Otherwise, this function will return as soon the first event is available or after timeout milliseconds have elapsed.

Parameters
timeout (*int*) – The timeout in milliseconds. If None, no `timeout` (infinite). This is in milliseconds to be compatible with `select.poll()`.

Returns
events – The list of events that are ready to be processed. This is a list of tuples of the form `(socket, event_mask)`, where the 0MQ Socket or integer fd is the first element, and the poll event mask (POLLIN, POLLOUT) is the second. It is common to call `events = dict(poller.poll())`, which turns the list of tuples into a mapping of `socket : event_mask`.

Return type
list

register(*socket*, *flags=POLLIN | POLLOUT*)

Register a 0MQ socket or native fd for I/O monitoring.

register(s,0) is equivalent to unregister(s).

Parameters

- **socket** (*zmq.Socket or native socket*) – A zmq.Socket or any Python object having a `fileno()` method that returns a valid file descriptor.
- **flags** (*int*) – The events to watch for. Can be POLLIN, POLLOUT or POLLIN|POLLOUT. If `flags=0`, socket will be unregistered.

unregister(*socket: Any*)

Remove a 0MQ socket or native fd for I/O monitoring.

Parameters
socket (Socket) – The socket instance to stop polling.

zmq.**select**(*rlist*, *wlist*, *xlist*, *timeout=None*)

Return the result of poll as a lists of sockets ready for r/w/exception.

This has the same interface as Python's built-in `select.select()` function.

Parameters

- **timeout** (*float, optional*) – The timeout in seconds. If None, no timeout (infinite). This is in seconds to be compatible with `select.select()`.
- **rlist** (*list*) – sockets/FDs to be polled for read events
- **wlist** (*list*) – sockets/FDs to be polled for write events

- **xlist** (*list*) – sockets/FDs to be polled for error events

Returns

- **rlist** (*list*) – list of sockets or FDs that are readable
- **wlist** (*list*) – list of sockets or FDs that are writable
- **xlist** (*list*) – list of sockets or FDs that had error events (rare)

Constants

All libzmq constants are available as top-level attributes (`zmq.PUSH`, etc.), as well as via enums (`zmq.SocketType.PUSH`, etc.).

Changed in version 23: constants for unavailable socket types or draft features will always be defined in pyzmq, whether the features themselves are available or not.

New in version 23: Each category of zmq constant is now available as an IntEnum.

`zmq.COPY_THRESHOLD`

The global default "small message" threshold for copying when `copy=False`. Copying has a thread-coordination cost, so zero-copy only has a benefit for sufficiently large messages.

enum `zmq.SocketType`(*value*)

zmq socket types

New in version 23.

Member Type
`int`

Valid values are as follows:

PAIR = <SocketType.PAIR: 0>

PUB = <SocketType.PUB: 1>

SUB = <SocketType.SUB: 2>

REQ = <SocketType.REQ: 3>

REP = <SocketType.REP: 4>

DEALER = <SocketType.DEALER: 5>

ROUTER = <SocketType.ROUTER: 6>

PULL = <SocketType.PULL: 7>

PUSH = <SocketType.PUSH: 8>

XPUB = <SocketType.XPUB: 9>

XSUB = <SocketType.XSUB: 10>

STREAM = <SocketType.STREAM: 11>

SERVER = <SocketType.SERVER: 12>

CLIENT = <SocketType.CLIENT: 13>

RADIO = <SocketType.RADIO: 14>

DISH = <SocketType.DISH: 15>

GATHER = <SocketType.GATHER: 16>

SCATTER = <SocketType.SCATTER: 17>

DGRAM = <SocketType.DGRAM: 18>

PEER = <SocketType.PEER: 19>

CHANNEL = <SocketType.CHANNEL: 20>

enum zmq.SocketOption(*value*)

Options for Socket.get/set

New in version 23.

Member Type
int

Valid values are as follows:

HWM = <SocketOption.HWM: 1>

AFFINITY = <SocketOption.AFFINITY: 4>

ROUTING_ID = <SocketOption.ROUTING_ID: 5>

SUBSCRIBE = <SocketOption.SUBSCRIBE: 6>

UNSUBSCRIBE = <SocketOption.UNSUBSCRIBE: 7>

RATE = <SocketOption.RATE: 8>

RECOVERY_IVL = <SocketOption.RECOVERY_IVL: 9>

SNDBUF = <SocketOption.SNDBUF: 11>

RCVBUF = <SocketOption.RCVBUF: 12>

RCVMORE = <SocketOption.RCVMORE: 13>

FD = <SocketOption.FD: 14>

EVENTS = <SocketOption.EVENTS: 15>

TYPE = <SocketOption.TYPE: 16>

LINGER = <SocketOption.LINGER: 17>

RECONNECT_IVL = <SocketOption.RECONNECT_IVL: 18>

BACKLOG = <SocketOption.BACKLOG: 19>

RECONNECT_IVL_MAX = <SocketOption.RECONNECT_IVL_MAX: 21>

MAXMSGSIZE = <SocketOption.MAXMSGSIZE: 22>

SNDHWM = <SocketOption.SNDHWM: 23>

```
RCVHWM = <SocketOption.RCVHWM: 24>

MULTICAST_HOPS = <SocketOption.MULTICAST_HOPS: 25>

RCVTIMEO = <SocketOption.RCVTIMEO: 27>

SNDTIMEO = <SocketOption.SNDTIMEO: 28>

LAST_ENDPOINT = <SocketOption.LAST_ENDPOINT: 32>

ROUTER_MANDATORY = <SocketOption.ROUTER_MANDATORY: 33>

TCP_KEEPALIVE = <SocketOption.TCP_KEEPALIVE: 34>

TCP_KEEPALIVE_CNT = <SocketOption.TCP_KEEPALIVE_CNT: 35>

TCP_KEEPALIVE_IDLE = <SocketOption.TCP_KEEPALIVE_IDLE: 36>

TCP_KEEPALIVE_INTVL = <SocketOption.TCP_KEEPALIVE_INTVL: 37>

IMMEDIATE = <SocketOption.IMMEDIATE: 39>

XPUB_VERBOSE = <SocketOption.XPUB_VERBOSE: 40>

ROUTER_RAW = <SocketOption.ROUTER_RAW: 41>

IPV6 = <SocketOption.IPV6:  42>

MECHANISM = <SocketOption.MECHANISM: 43>

PLAIN_SERVER = <SocketOption.PLAIN_SERVER: 44>

PLAIN_USERNAME = <SocketOption.PLAIN_USERNAME: 45>

PLAIN_PASSWORD = <SocketOption.PLAIN_PASSWORD: 46>

CURVE_SERVER = <SocketOption.CURVE_SERVER: 47>

CURVE_PUBLICKEY = <SocketOption.CURVE_PUBLICKEY: 48>

CURVE_SECRETKEY = <SocketOption.CURVE_SECRETKEY: 49>

CURVE_SERVERKEY = <SocketOption.CURVE_SERVERKEY: 50>

PROBE_ROUTER = <SocketOption.PROBE_ROUTER: 51>

REQ_CORRELATE = <SocketOption.REQ_CORRELATE: 52>

REQ_RELAXED = <SocketOption.REQ_RELAXED: 53>

CONFLATE = <SocketOption.CONFLATE: 54>

ZAP_DOMAIN = <SocketOption.ZAP_DOMAIN: 55>

ROUTER_HANDOVER = <SocketOption.ROUTER_HANDOVER: 56>

TOS = <SocketOption.TOS: 57>

CONNECT_ROUTING_ID = <SocketOption.CONNECT_ROUTING_ID: 61>

GSSAPI_SERVER = <SocketOption.GSSAPI_SERVER: 62>
```

GSSAPI_PRINCIPAL = <SocketOption.GSSAPI_PRINCIPAL: 63>

GSSAPI_SERVICE_PRINCIPAL = <SocketOption.GSSAPI_SERVICE_PRINCIPAL: 64>

GSSAPI_PLAINTEXT = <SocketOption.GSSAPI_PLAINTEXT: 65>

HANDSHAKE_IVL = <SocketOption.HANDSHAKE_IVL: 66>

SOCKS_PROXY = <SocketOption.SOCKS_PROXY: 68>

XPUB_NODROP = <SocketOption.XPUB_NODROP: 69>

BLOCKY = <SocketOption.BLOCKY: 70>

XPUB_MANUAL = <SocketOption.XPUB_MANUAL: 71>

XPUB_WELCOME_MSG = <SocketOption.XPUB_WELCOME_MSG: 72>

STREAM_NOTIFY = <SocketOption.STREAM_NOTIFY: 73>

INVERT_MATCHING = <SocketOption.INVERT_MATCHING: 74>

HEARTBEAT_IVL = <SocketOption.HEARTBEAT_IVL: 75>

HEARTBEAT_TTL = <SocketOption.HEARTBEAT_TTL: 76>

HEARTBEAT_TIMEOUT = <SocketOption.HEARTBEAT_TIMEOUT: 77>

XPUB_VERBOSER = <SocketOption.XPUB_VERBOSER: 78>

CONNECT_TIMEOUT = <SocketOption.CONNECT_TIMEOUT: 79>

TCP_MAXRT = <SocketOption.TCP_MAXRT: 80>

THREAD_SAFE = <SocketOption.THREAD_SAFE: 81>

MULTICAST_MAXTPDU = <SocketOption.MULTICAST_MAXTPDU: 84>

VMCI_BUFFER_SIZE = <SocketOption.VMCI_BUFFER_SIZE: 85>

VMCI_BUFFER_MIN_SIZE = <SocketOption.VMCI_BUFFER_MIN_SIZE: 86>

VMCI_BUFFER_MAX_SIZE = <SocketOption.VMCI_BUFFER_MAX_SIZE: 87>

VMCI_CONNECT_TIMEOUT = <SocketOption.VMCI_CONNECT_TIMEOUT: 88>

USE_FD = <SocketOption.USE_FD: 89>

GSSAPI_PRINCIPAL_NAMETYPE = <SocketOption.GSSAPI_PRINCIPAL_NAMETYPE: 90>

GSSAPI_SERVICE_PRINCIPAL_NAMETYPE = <SocketOption.GSSAPI_SERVICE_PRINCIPAL_NAMETYPE: 91>

BINDTODEVICE = <SocketOption.BINDTODEVICE: 92>

TCP_ACCEPT_FILTER = <SocketOption.TCP_ACCEPT_FILTER: 38>

IPC_FILTER_PID = <SocketOption.IPC_FILTER_PID: 58>

IPC_FILTER_UID = <SocketOption.IPC_FILTER_UID: 59>

```
IPC_FILTER_GID = <SocketOption.IPC_FILTER_GID: 60>

IPV4ONLY = <SocketOption.IPV4ONLY: 31>

ZAP_ENFORCE_DOMAIN = <SocketOption.ZAP_ENFORCE_DOMAIN: 93>

LOOPBACK_FASTPATH = <SocketOption.LOOPBACK_FASTPATH: 94>

METADATA = <SocketOption.METADATA: 95>

MULTICAST_LOOP = <SocketOption.MULTICAST_LOOP: 96>

ROUTER_NOTIFY = <SocketOption.ROUTER_NOTIFY: 97>

XPUB_MANUAL_LAST_VALUE = <SocketOption.XPUB_MANUAL_LAST_VALUE: 98>

SOCKS_USERNAME = <SocketOption.SOCKS_USERNAME: 99>

SOCKS_PASSWORD = <SocketOption.SOCKS_PASSWORD: 100>

IN_BATCH_SIZE = <SocketOption.IN_BATCH_SIZE: 101>

OUT_BATCH_SIZE = <SocketOption.OUT_BATCH_SIZE: 102>

WSS_KEY_PEM = <SocketOption.WSS_KEY_PEM: 103>

WSS_CERT_PEM = <SocketOption.WSS_CERT_PEM: 104>

WSS_TRUST_PEM = <SocketOption.WSS_TRUST_PEM: 105>

WSS_HOSTNAME = <SocketOption.WSS_HOSTNAME: 106>

WSS_TRUST_SYSTEM = <SocketOption.WSS_TRUST_SYSTEM: 107>

ONLY_FIRST_SUBSCRIBE = <SocketOption.ONLY_FIRST_SUBSCRIBE: 108>

RECONNECT_STOP = <SocketOption.RECONNECT_STOP: 109>

HELLO_MSG = <SocketOption.HELLO_MSG: 110>

DISCONNECT_MSG = <SocketOption.DISCONNECT_MSG: 111>

PRIORITY = <SocketOption.PRIORITY: 112>

BUSY_POLL = <SocketOption.BUSY_POLL: 113>

HICCUP_MSG = <SocketOption.HICCUP_MSG: 114>

XSUB_VERBOSE_UNSUBSCRIBE = <SocketOption.XSUB_VERBOSE_UNSUBSCRIBE: 115>

TOPICS_COUNT = <SocketOption.TOPICS_COUNT: 116>

NORM_MODE = <SocketOption.NORM_MODE: 117>

NORM_UNICAST_NACK = <SocketOption.NORM_UNICAST_NACK: 118>

NORM_BUFFER_SIZE = <SocketOption.NORM_BUFFER_SIZE: 119>

NORM_SEGMENT_SIZE = <SocketOption.NORM_SEGMENT_SIZE: 120>

NORM_BLOCK_SIZE = <SocketOption.NORM_BLOCK_SIZE: 121>
```

NORM_NUM_PARITY = <SocketOption.NORM_NUM_PARITY: 122>

NORM_NUM_AUTOPARITY = <SocketOption.NORM_NUM_AUTOPARITY: 123>

NORM_PUSH = <SocketOption.NORM_PUSH: 124>

enum zmq.Flag(*value*)

Send/recv flags

New in version 23.

Member Type
int

Valid values are as follows:

DONTWAIT = <Flag.DONTWAIT: 1>

SNDMORE = <Flag.SNDMORE: 2>

enum zmq.PollEvent(*value*)

Which events to poll for in poll methods

Member Type
int

Valid values are as follows:

POLLIN = <PollEvent.POLLIN: 1>

POLLOUT = <PollEvent.POLLOUT: 2>

POLLERR = <PollEvent.POLLERR: 4>

POLLPRI = <PollEvent.POLLPRI: 8>

enum zmq.ContextOption(*value*)

Options for Context.get/set

New in version 23.

Member Type
int

Valid values are as follows:

IO_THREADS = <ContextOption.IO_THREADS: 1>

MAX_SOCKETS = <ContextOption.MAX_SOCKETS: 2>

SOCKET_LIMIT = <ContextOption.SOCKET_LIMIT: 3>

THREAD_SCHED_POLICY = <ContextOption.THREAD_SCHED_POLICY: 4>

MAX_MSGSZ = <ContextOption.MAX_MSGSZ: 5>

MSG_T_SIZE = <ContextOption.MSG_T_SIZE: 6>

THREAD_AFFINITY_CPU_ADD = <ContextOption.THREAD_AFFINITY_CPU_ADD: 7>

THREAD_AFFINITY_CPU_REMOVE = <ContextOption.THREAD_AFFINITY_CPU_REMOVE: 8>

THREAD_NAME_PREFIX = <ContextOption.THREAD_NAME_PREFIX: 9>

enum zmq.MessageOption(*value*)

Options on zmq.Frame objects

New in version 23.

Member Type
int

Valid values are as follows:

MORE = <MessageOption.MORE: 1>

SHARED = <MessageOption.SHARED: 3>

SRCFD = <MessageOption.SRCFD: 2>

enum zmq.Event(*value*)

Socket monitoring events

New in version 23.

Member Type
int

Valid values are as follows:

PROTOCOL_ERROR_ZMTP_UNSPECIFIED = <Event.PROTOCOL_ERROR_ZMTP_UNSPECIFIED: 268435456>

PROTOCOL_ERROR_ZAP_UNSPECIFIED = <Event.PROTOCOL_ERROR_ZAP_UNSPECIFIED: 536870912>

CONNECTED = <Event.CONNECTED: 1>

CONNECT_DELAYED = <Event.CONNECT_DELAYED: 2>

CONNECT_RETRIED = <Event.CONNECT_RETRIED: 4>

LISTENING = <Event.LISTENING: 8>

BIND_FAILED = <Event.BIND_FAILED: 16>

ACCEPTED = <Event.ACCEPTED: 32>

ACCEPT_FAILED = <Event.ACCEPT_FAILED: 64>

CLOSED = <Event.CLOSED: 128>

CLOSE_FAILED = <Event.CLOSE_FAILED: 256>

DISCONNECTED = <Event.DISCONNECTED: 512>

MONITOR_STOPPED = <Event.MONITOR_STOPPED: 1024>

HANDSHAKE_FAILED_NO_DETAIL = <Event.HANDSHAKE_FAILED_NO_DETAIL: 2048>

HANDSHAKE_SUCCEEDED = <Event.HANDSHAKE_SUCCEEDED: 4096>

HANDSHAKE_FAILED_PROTOCOL = <Event.HANDSHAKE_FAILED_PROTOCOL: 8192>

HANDSHAKE_FAILED_AUTH = <Event.HANDSHAKE_FAILED_AUTH: 16384>

PIPES_STATS = <Event.PIPES_STATS: 65536>

enum zmq.NormMode(*value*)

Values for zmq.NORM_MODE socket option

New in version 26.

New in version libzmq-4.3.5: (draft)

Member Type
int

Valid values are as follows:

FIXED = <NormMode.FIXED: 0>

CC = <NormMode.CC: 1>

CCL = <NormMode.CCL: 2>

CCE = <NormMode.CCE: 3>

CCE_ECNONLY = <NormMode.CCE_ECNONLY: 4>

enum zmq.RouterNotify(*value*)

Values for zmq.ROUTER_NOTIFY socket option

New in version 26.

New in version libzmq-4.3.0: (draft)

Member Type
int

Valid values are as follows:

CONNECT = <RouterNotify.CONNECT: 1>

DISCONNECT = <RouterNotify.DISCONNECT: 2>

enum zmq.ReconnectStop(*value*)

Select behavior for socket.reconnect_stop

New in version 25.

Member Type
int

Valid values are as follows:

CONN_REFUSED = <ReconnectStop.CONN_REFUSED: 1>

HANDSHAKE_FAILED = <ReconnectStop.HANDSHAKE_FAILED: 2>

AFTER_DISCONNECT = <ReconnectStop.AFTER_DISCONNECT: 4>

enum zmq.SecurityMechanism(*value*)

Security mechanisms (as returned by `socket.get(zmq.MECHANISM)`)

New in version 23.

Member Type
int

Valid values are as follows:

NULL = <SecurityMechanism.NULL: 0>

PLAIN = <SecurityMechanism.PLAIN: 1>

CURVE = <SecurityMechanism.CURVE: 2>

GSSAPI = <SecurityMechanism.GSSAPI: 3>

enum zmq.DeviceType(*value*)

Device type constants for zmq.device

Member Type
int

Valid values are as follows:

STREAMER = <DeviceType.STREAMER: 1>

FORWARDER = <DeviceType.FORWARDER: 2>

QUEUE = <DeviceType.QUEUE: 3>

enum zmq.Errno(*value*)

libzmq error codes

New in version 23.

Member Type
int

Valid values are as follows:

EAGAIN = <Errno.EAGAIN: 11>

EFAULT = <Errno.EFAULT: 14>

EINVAL = <Errno.EINVAL: 22>

ENOTSUP = <Errno.ENOTSUP: 95>

EPROTONOSUPPORT = <Errno.EPROTONOSUPPORT: 93>

ENOBUFS = <Errno.ENOBUFS: 105>

ENETDOWN = <Errno.ENETDOWN: 100>

EADDRINUSE = <Errno.EADDRINUSE: 98>

EADDRNOTAVAIL = <Errno.EADDRNOTAVAIL: 99>

ECONNREFUSED = <Errno.ECONNREFUSED: 111>

EINPROGRESS = <Errno.EINPROGRESS: 115>

ENOTSOCK = <Errno.ENOTSOCK: 88>

EMSGSIZE = <Errno.EMSGSIZE: 90>

EAFNOSUPPORT = <Errno.EAFNOSUPPORT: 97>

ENETUNREACH = <Errno.ENETUNREACH: 101>

ECONNABORTED = <Errno.ECONNABORTED: 103>

ECONNRESET = <Errno.ECONNRESET: 104>

ENOTCONN = <Errno.ENOTCONN: 107>

ETIMEDOUT = <Errno.ETIMEDOUT: 110>

EHOSTUNREACH = <Errno.EHOSTUNREACH: 113>

ENETRESET = <Errno.ENETRESET: 102>

EFSM = <Errno.EFSM: 156384763>

ENOCOMPATPROTO = <Errno.ENOCOMPATPROTO: 156384764>

ETERM = <Errno.ETERM: 156384765>

EMTHREAD = <Errno.EMTHREAD: 156384766>

Exceptions

ZMQError

class zmq.**ZMQError**(*errno: int | None = None*, *msg: str | None = None*)

Wrap an errno style error.

Parameters

- **errno** (*int*) – The ZMQ errno or None. If None, then `zmq_errno()` is called and used.
- **msg** (*str*) – Description of the error or None.

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

ZMQVersionError

class zmq.**ZMQVersionError**(*min_version: str*, *msg: str = 'Feature'*)

Raised when a feature is not provided by the linked version of libzmq.

New in version 14.2.

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

Again

class zmq.**Again**(*errno='ignored'*, *msg='ignored'*)

Wrapper for zmq.EAGAIN

New in version 13.0.

ContextTerminated

class zmq.**ContextTerminated**(*errno='ignored'*, *msg='ignored'*)

Wrapper for zmq.ETERM

New in version 13.0.

NotDone

class zmq.**NotDone**

Raised when timeout is reached while waiting for 0MQ to finish with a Message

See also:

MessageTracker.wait
object for tracking when ZeroMQ is done

ZMQBindError

class zmq.**ZMQBindError**

An error for `Socket.bind_to_random_port()`.

See also:

Socket.bind_to_random_port

Functions

zmq.**zmq_version**() → str

return the version of libzmq as a string

zmq.**pyzmq_version**() → str

return the version of pyzmq as a string

zmq.**zmq_version_info**() → tuple[int, int, int]

Return the version of ZeroMQ itself as a 3-tuple of ints.

zmq.**pyzmq_version_info**() → tuple[int, int, int] | tuple[int, int, int, float]

return the pyzmq version as a tuple of at least three numbers

If pyzmq is a development version, `inf` will be appended after the third integer.

zmq.**has**(*capability*) → bool

Check for zmq capability by name (e.g. 'ipc', 'curve')

New in version libzmq-4.1.

New in version 14.1.

zmq.**device**(*device_type: int*, *frontend:* zmq.Socket, *backend:* zmq.Socket = *None*)

Start a zeromq device.

Deprecated since version libzmq-3.2: Use zmq.proxy

Parameters

- **device_type** (*int*) – one of: QUEUE, FORWARDER, STREAMER The type of device to start.
- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.

zmq.**proxy**(*frontend:* zmq.Socket, *backend:* zmq.Socket, *capture:* zmq.Socket = *None*)

Start a zeromq proxy (replacement for device).

New in version libzmq-3.2.

New in version 13.0.

Parameters

- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.
- **capture** (Socket *(optional)*) – The Socket instance for capturing traffic.

zmq.**proxy_steerable**(*frontend:* zmq.Socket, *backend:* zmq.Socket, *capture:* zmq.Socket = *None*, *control:* zmq.Socket = *None*)

Start a zeromq proxy with control flow.

New in version libzmq-4.1.

New in version 18.0.

Parameters

- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.
- **capture** (Socket *(optional)*) – The Socket instance for capturing traffic.
- **control** (Socket *(optional)*) – The Socket instance for control flow.

zmq.**curve_public**(*secret_key*) → bytes

Compute the public key corresponding to a secret key for use with zmq.CURVE security

Requires libzmq (4.2) to have been built with CURVE support.

Parameters
private – The private key as a 40 byte z85-encoded bytestring

Returns
The public key as a 40 byte z85-encoded bytestring

Return type
bytes

zmq.**curve_keypair**() → tuple[bytes, bytes]

generate a Z85 key pair for use with zmq.CURVE security

Requires libzmq (4.0) to have been built with CURVE support.

New in version libzmq-4.0.

New in version 14.0.

Returns

- **public** (*bytes*) – The public key as 40 byte z85-encoded bytestring.
- **private** (*bytes*) – The private key as 40 byte z85-encoded bytestring.

zmq.**get_includes**()

Return a list of directories to include for linking against pyzmq with cython.

zmq.**get_library_dirs**()

Return a list of directories used to link against pyzmq's bundled libzmq.

zmq.**strerror**(*errno: int*) → str

Return the error string given the error number.

2.1.2 devices

Functions

zmq.**device**(*device_type: int*, *frontend:* zmq.Socket, *backend:* zmq.Socket = *None*)

Start a zeromq device.

Deprecated since version libzmq-3.2: Use zmq.proxy

Parameters

- **device_type** (*int*) – one of: QUEUE, FORWARDER, STREAMER The type of device to start.
- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.

zmq.**proxy**(*frontend:* zmq.Socket, *backend:* zmq.Socket, *capture:* zmq.Socket = *None*)

Start a zeromq proxy (replacement for device).

New in version libzmq-3.2.

New in version 13.0.

Parameters

- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.
- **capture** (Socket *(optional)*) – The Socket instance for capturing traffic.

zmq.**proxy_steerable**(*frontend:* zmq.Socket, *backend:* zmq.Socket, *capture:* zmq.Socket = *None*, *control:* zmq.Socket = *None*)

Start a zeromq proxy with control flow.

New in version libzmq-4.1.

New in version 18.0.

Parameters

- **frontend** (Socket) – The Socket instance for the incoming traffic.
- **backend** (Socket) – The Socket instance for the outbound traffic.
- **capture** (Socket *(optional)*) – The Socket instance for capturing traffic.
- **control** (Socket *(optional)*) – The Socket instance for control flow.

Module: zmq.devices

0MQ Device classes for running in background threads or processes.

Base Devices

Device

class zmq.devices.**Device**(*device_type: int = DeviceType.QUEUE*, *in_type: int | None = None*, *out_type: int | None = None*)

A 0MQ Device to be run in the background.

You do not pass Socket instances to this, but rather Socket types:

```
Device(device_type, in_socket_type, out_socket_type)
```

For instance:

```
dev = Device(zmq.QUEUE, zmq.DEALER, zmq.ROUTER)
```

Similar to zmq.device, but socket types instead of sockets themselves are passed, and the sockets are created in the work thread, to avoid issues with thread safety. As a result, additional bind_{in|out} and connect_{in|out} methods and setsockopt_{in|out} allow users to specify connections for the sockets.

Parameters

- **device_type** (*int*) – The 0MQ Device type
- **{in|out}_type** (*int*) – zmq socket types, to be passed later to context.socket(). e.g. zmq.PUB, zmq.SUB, zmq.REQ. If out_type is < 0, then in_socket is used for both in_socket and out_socket.

bind_{in_out}(iface)

passthrough for `{in|out}_socket.bind(iface)`, to be called in the thread

connect_{in_out}(iface)

passthrough for `{in|out}_socket.connect(iface)`, to be called in the thread

setsockopt_{in_out}(opt,value)

passthrough for `{in|out}_socket.setsockopt(opt, value)`, to be called in the thread

daemon

sets whether the thread should be run as a daemon Default is true, because if it is false, the thread will not exit unless it is killed

Type
bool

context_factory

This is a class attribute. Function for creating the Context. This will be Context.instance in ThreadDevices, and Context in ProcessDevices. The only reason it is not instance() in ProcessDevices is that there may be a stale Context instance already initialized, and the forked environment should *never* try to use it.

Type
callable

bind_in(*addr: str*) → None

Enqueue ZMQ address for binding on in_socket.

See zmq.Socket.bind for details.

bind_in_to_random_port(*addr: str*, **args*, ***kwargs*) → int

Enqueue a random port on the given interface for binding on in_socket.

See zmq.Socket.bind_to_random_port for details.

New in version 18.0.

bind_out(*addr: str*) → None

Enqueue ZMQ address for binding on out_socket.

See zmq.Socket.bind for details.

bind_out_to_random_port(*addr: str*, **args*, ***kwargs*) → int

Enqueue a random port on the given interface for binding on out_socket.

See zmq.Socket.bind_to_random_port for details.

New in version 18.0.

connect_in(*addr: str*) → None

Enqueue ZMQ address for connecting on in_socket.

See zmq.Socket.connect for details.

connect_out(*addr: str*)

Enqueue ZMQ address for connecting on out_socket.

See zmq.Socket.connect for details.

join(*timeout: float | None = None*) → None

wait for me to finish, like Thread.join.

Reimplemented appropriately by subclasses.

setsockopt_in(*opt: int*, *value: Any*) → None

Enqueue setsockopt(opt, value) for in_socket

See zmq.Socket.setsockopt for details.

setsockopt_out(*opt: int*, *value: Any*)

Enqueue setsockopt(opt, value) for out_socket

See zmq.Socket.setsockopt for details.

start() → None

Start the device. Override me in subclass for other launchers.

ThreadDevice

class zmq.devices.**ThreadDevice**(*device_type: int = DeviceType.QUEUE*, *in_type: int | None = None*, *out_type: int | None = None*)

A Device that will be run in a background Thread.

See Device for details.

ProcessDevice

class zmq.devices.**ProcessDevice**(*device_type: int = DeviceType.QUEUE*, *in_type: int | None = None*, *out_type: int | None = None*)

A Device that will be run in a background Process.

See Device for details.

context_factory

alias of *Context*

Proxy Devices

Proxy

class zmq.devices.**Proxy**(*in_type*, *out_type*, *mon_type=SocketType.PUB*)

Threadsafe Proxy object.

See zmq.devices.Device for most of the spec. This subclass adds a <method>_mon version of each <method>_{in|out} method, for configuring the monitor socket.

A Proxy is a 3-socket ZMQ Device that functions just like a QUEUE, except each message is also sent out on the monitor socket.

A PUB socket is the most logical choice for the mon_socket, but it is not required.

bind_mon(*addr*)

Enqueue ZMQ address for binding on mon_socket.

See zmq.Socket.bind for details.

connect_mon(*addr*)

Enqueue ZMQ address for connecting on mon_socket.

See zmq.Socket.connect for details.

setsockopt_mon(*opt*, *value*)

Enqueue setsockopt(opt, value) for mon_socket

See zmq.Socket.setsockopt for details.

ThreadProxy

class zmq.devices.**ThreadProxy**(*in_type*, *out_type*, *mon_type=SocketType.PUB*)

Proxy in a Thread. See Proxy for more.

ProcessProxy

class zmq.devices.**ProcessProxy**(*in_type*, *out_type*, *mon_type=SocketType.PUB*)

Proxy in a Process. See Proxy for more.

ProxySteerable

class zmq.devices.**ProxySteerable**(*in_type*, *out_type*, *mon_type=SocketType.PUB*, *ctrl_type=None*)

Class for running a steerable proxy in the background.

See zmq.devices.Proxy for most of the spec. If the control socket is not NULL, the proxy supports control flow, provided by the socket.

If PAUSE is received on this socket, the proxy suspends its activities. If RESUME is received, it goes on. If TERMINATE is received, it terminates smoothly. If the control socket is NULL, the proxy behave exactly as if zmq.devices.Proxy had been used.

This subclass adds a <method>_ctrl version of each <method>_{in|out} method, for configuring the control socket.

New in version libzmq-4.1.

New in version 18.0.

bind_ctrl(*addr*)

Enqueue ZMQ address for binding on ctrl_socket.

See zmq.Socket.bind for details.

connect_ctrl(*addr*)

Enqueue ZMQ address for connecting on ctrl_socket.

See zmq.Socket.connect for details.

setsockopt_ctrl(*opt*, *value*)

Enqueue setsockopt(opt, value) for ctrl_socket

See zmq.Socket.setsockopt for details.

ThreadProxySteerable

class zmq.devices.**ThreadProxySteerable**(*in_type*, *out_type*, *mon_type=SocketType.PUB*, *ctrl_type=None*)

ProxySteerable in a Thread. See ProxySteerable for details.

ProcessProxySteerable

class zmq.devices.**ProcessProxySteerable**(*in_type*, *out_type*, *mon_type=SocketType.PUB*, *ctrl_type=None*)

ProxySteerable in a Process. See ProxySteerable for details.

MonitoredQueue Devices

zmq.devices.**monitored_queue**(*in_socket:* zmq.Socket, *out_socket:* zmq.Socket, *mon_socket:* zmq.Socket, *in_prefix:* *bytes* = *b'in'*, *out_prefix:* *bytes* = *b'out'*)

Start a monitored queue device.

A monitored queue is very similar to the zmq.proxy device (monitored queue came first).

Differences from zmq.proxy:

- monitored_queue supports both in and out being ROUTER sockets (via swapping IDENTITY prefixes).
- monitor messages are prefixed, making in and out messages distinguishable.

Parameters

- **in_socket** (zmq.Socket) – One of the sockets to the Queue. Its messages will be prefixed with 'in'.
- **out_socket** (zmq.Socket) – One of the sockets to the Queue. Its messages will be prefixed with 'out'. The only difference between in/out socket is this prefix.
- **mon_socket** (zmq.Socket) – This socket sends out every message received by each of the others with an in/out prefix specifying which one it was.
- **in_prefix** (*str*) – Prefix added to broadcast messages from in_socket.
- **out_prefix** (*str*) – Prefix added to broadcast messages from out_socket.

MonitoredQueue

class zmq.devices.**MonitoredQueue**(*in_type*, *out_type*, *mon_type=SocketType.PUB*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Class for running monitored_queue in the background.

See zmq.devices.Device for most of the spec. MonitoredQueue differs from Proxy, only in that it adds a `prefix` to messages sent on the monitor socket, with a different prefix for each direction.

MQ also supports ROUTER on both sides, which zmq.proxy does not.

If a message arrives on `in_sock`, it will be prefixed with `in_prefix` on the monitor socket. If it arrives on out_sock, it will be prefixed with `out_prefix`.

A PUB socket is the most logical choice for the mon_socket, but it is not required.

ThreadMonitoredQueue

class zmq.devices.**ThreadMonitoredQueue**(*in_type*, *out_type*, *mon_type=SocketType.PUB*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Run zmq.monitored_queue in a background thread.

See MonitoredQueue and Proxy for details.

ProcessMonitoredQueue

class zmq.devices.**ProcessMonitoredQueue**(*in_type*, *out_type*, *mon_type=SocketType.PUB*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Run zmq.monitored_queue in a separate process.

See MonitoredQueue and Proxy for details.

2.1.3 decorators

Module: zmq.decorators

Decorators for running functions with context/sockets.

New in version 15.3.

Like using Contexts and Sockets as context managers, but with decorator syntax. Context and sockets are closed at the end of the function.

For example:

```
from zmq.decorators import context, socket

@context()
@socket(zmq.PUSH)
def work(ctx, push):
    ...
```

Decorators

zmq.decorators.**context**(**args*, ***kwargs*)

Decorator for adding a Context to a function.

Usage:

```
@context()
def foo(ctx):
    ...
```

New in version 15.3.

Parameters

name (*str*) – the keyword argument passed to decorated function

zmq.decorators.**socket**(**args*, ***kwargs*)

Decorator for adding a socket to a function.

Usage:

```
@socket(zmq.PUSH)
def foo(push):
    ...
```

New in version 15.3.

Parameters

- **name** (*str*) – the keyword argument passed to decorated function
- **context_name** (*str*) – the keyword only argument to identify context object

2.1.4 green

Module: zmq.green

zmq.green - gevent compatibility with zeromq.

Usage

Instead of importing zmq directly, do so in the following manner:

import zmq.green as zmq

Any calls that would have blocked the current thread will now only block the current green thread.

This compatibility is accomplished by ensuring the nonblocking flag is set before any blocking operation and the ØMQ file descriptor is polled internally to trigger needed events.

2.1.5 eventloop.ioloop

Module: zmq.eventloop.ioloop

This module is deprecated in pyzmq 17. Use tornado.ioloop.

2.1.6 eventloop.future

Module: zmq.eventloop.future

Future-returning APIs for tornado coroutines.

See also:

zmq.asyncio

New in version 15.0.

As of pyzmq 15, there is a new Socket subclass that returns Futures for recv methods, which can be found at *Socket*. You can create these sockets by instantiating a *Context* from the same module. These sockets let you easily use zmq with tornado's coroutines.

See also:

tornado.gen

```
from tornado import gen
from zmq.eventloop.future import Context

ctx = Context()


@gen.coroutine
def recv_and_process():
    sock = ctx.socket(zmq.PULL)
    sock.bind(url)
    msg = yield sock.recv_multipart()  # waits for msg to be ready
    reply = yield async_process(msg)
    yield sock.send_multipart(reply)
```

Classes

Context

Context class that creates Future-returning sockets. See *zmq.Context* for more info.

class zmq.eventloop.future.**Context**(**args: Any*, ***kwargs: Any*)

Socket

Socket subclass that returns Future s from blocking methods, for use in coroutines and async applications.

See also:

zmq.Socket for the inherited API.

class zmq.eventloop.future.**Socket**(*context=None*, *socket_type=-1*, *io_loop=None*, *_from_socket: Socket | None = None*, ***kwargs*)

recv(*flags: int = 0*, *copy: bool = True*, *track: bool = False*) → Awaitable[bytes | *Frame*]

Receive a single zmq frame.

Returns a Future, whose result will be the received frame.

Recommend using recv_multipart instead.

recv_multipart(*flags: int = 0*, *copy: bool = True*, *track: bool = False*) → Awaitable[list[bytes] | list[*Frame*]]

Receive a complete multipart zmq message.

Returns a Future whose result will be a multipart message.

send(*data: Any*, *flags: int = 0*, *copy: bool = True*, *track: bool = False*, ***kwargs: Any*) → Awaitable[*MessageTracker* | None]

Send a single zmq frame.

Returns a Future that resolves when sending is complete.

Recommend using send_multipart instead.

send_multipart(*msg_parts: Any*, *flags: int = 0*, *copy: bool = True*, *track=False*, ***kwargs*) → Awaitable[*MessageTracker* | None]

Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

poll(*timeout=None*, *flags=PollEvent.POLLIN*) → Awaitable[int]

poll the socket for events

returns a Future for the poll results.

Poller

Poller subclass that returns `Future` s from poll, for use in coroutines and async applications.

See also:

zmq.Poller for the inherited API.

class zmq.eventloop.future.**Poller**

poll(*timeout=-1*) → Awaitable[list[tuple[Any, int]]]

Return a Future for a poll event

2.1.7 asyncio

Module: zmq.asyncio

AsyncIO support for zmq

Requires asyncio and Python 3.

New in version 15.0.

As of 15.0, pyzmq now supports `asyncio`, via *zmq.asyncio*. When imported from this module, blocking methods such as *Socket.recv_multipart()*, *Socket.poll()*, and *Poller.poll()* return `Future` s.

```
import asyncio
import zmq
import zmq.asyncio

ctx = zmq.asyncio.Context()


async def recv_and_process():
    sock = ctx.socket(zmq.PULL)
    sock.bind(url)
    msg = await sock.recv_multipart()  # waits for msg to be ready
    reply = await async_process(msg)
    await sock.send_multipart(reply)


asyncio.run(recv_and_process())
```

Classes

Context

Context class that creates Future-returning sockets. See *zmq.Context* for more info.

class zmq.asyncio.**Context**(*io_threads: int = 1*)

class zmq.asyncio.**Context**(*io_threads: Context*)

class zmq.asyncio.**Context**(**, shadow: Context | int*)

Context for creating asyncio-compatible Sockets

Socket

Socket subclass that returns `asyncio.Future` s from blocking methods, for use in coroutines and async applications.

See also:

zmq.Socket for the inherited API.

class zmq.asyncio.**Socket**(*context=None*, *socket_type=-1*, *io_loop=None*, *_from_socket: Socket | None = None*, ***kwargs*)

Socket returning asyncio Futures for send/recv/poll methods.

recv(*flags: int = 0*, *copy: bool = True*, *track: bool = False*) → Awaitable[bytes | *Frame*]

Receive a single zmq frame.

Returns a Future, whose result will be the received frame.

Recommend using recv_multipart instead.

recv_multipart(*flags: int = 0*, *copy: bool = True*, *track: bool = False*) → Awaitable[list[bytes] | list[*Frame*]]

Receive a complete multipart zmq message.

Returns a Future whose result will be a multipart message.

send(*data: Any*, *flags: int = 0*, *copy: bool = True*, *track: bool = False*, ***kwargs: Any*) → Awaitable[*MessageTracker* | None]

Send a single zmq frame.

Returns a Future that resolves when sending is complete.

Recommend using send_multipart instead.

send_multipart(*msg_parts: Any*, *flags: int = 0*, *copy: bool = True*, *track=False*, ***kwargs*) → Awaitable[*MessageTracker* | None]

Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

poll(*timeout=None*, *flags=PollEvent.POLLIN*) → Awaitable[int]

poll the socket for events

returns a Future for the poll results.

Poller

Poller subclass that returns asyncio.Future s from poll, for use in coroutines and async applications.

See also:

zmq.Poller for the inherited API.

class zmq.asyncio.**Poller**

Poller returning asyncio.Future for poll results.

poll(*timeout=-1*) → Awaitable[list[tuple[Any, int]]]

Return a Future for a poll event

2.1.8 eventloop.zmqstream

Module: zmq.eventloop.zmqstream

A utility class for event-based messaging on a zmq socket using tornado.

See also:

- *zmq.asyncio*
- *zmq.eventloop.future*

ZMQStream

class zmq.eventloop.zmqstream.**ZMQStream**(*socket:* Socket, *io_loop: IOLoop* | *None* = *None*)

A utility class to register callbacks when a zmq socket sends and receives

For use with tornado IOLoop.

There are three main methods

Methods:

- **on_recv(callback, copy=True):**
  register a callback to be run every time the socket has something to receive
- **on_send(callback):**
  register a callback to be run every time you call send
- **send_multipart(self, msg, flags=0, copy=False, callback=None):**
  perform a send that will trigger the callback if callback is passed, on_send is also called.

  There are also send_multipart(), send_json(), send_pyobj()

Three other methods for deactivating the callbacks:

- **stop_on_recv():**
  turn off the recv callback
- **stop_on_send():**
  turn off the send callback

which simply call on_<evt>(None).

The entire socket interface, excluding direct recv methods, is also provided, primarily through direct-linking the methods. e.g.

```
>>> stream.bind is stream.socket.bind
True
```

New in version 25: send/recv callbacks can be coroutines.

Changed in version 25: ZMQStreams only support base zmq.Socket classes (this has always been true, but not enforced). If ZMQStreams are created with e.g. async Socket subclasses, a RuntimeWarning will be shown, and the socket cast back to the default zmq.Socket before connecting events.

Previously, using async sockets (or any zmq.Socket subclass) would result in undefined behavior for the arguments passed to callback functions. Now, the callback functions reliably get the return value of the base `zmq.Socket` send/recv_multipart methods (the list of message frames).

close(*linger: int | None = None*) → None

Close this stream.

closed() → bool

flush(*flag: int = PollEvent.POLLIN | POLLOUT*, *limit: int | None = None*)

Flush pending messages.

This method safely handles all pending incoming and/or outgoing messages, bypassing the inner loop, passing them to the registered callbacks.

A limit can be specified, to prevent blocking under high load.

flush will return the first time ANY of these conditions are met:

- No more events matching the flag are pending.
- the total number of events handled reaches the limit.

Note that if `flag|POLLIN != 0`, recv events will be flushed even if no callback is registered, unlike normal IOLoop operation. This allows flush to be used to remove *and ignore* incoming messages.

Parameters

- **flag** (*int*) – default=POLLIN|POLLOUT 0MQ poll flags. If flag|POLLIN, recv events will be flushed. If flag|POLLOUT, send events will be flushed. Both flags can be set at once, which is the default.
- **limit** (*None or int, optional*) – The maximum number of messages to send or receive. Both send and recv count against this limit.

Returns

count of events handled (both send and recv)

Return type

int

io_loop: IOLoop

on_err(*callback: Callable*)

DEPRECATED, does nothing

on_recv(*callback: Callable[[list[bytes]], Any]*) → None

on_recv(*callback: Callable[[list[bytes]], Any]*, *copy: Literal[True]*) → None

on_recv(*callback: Callable[[list[*Frame*]], Any]*, *copy: Literal[False]*) → None

on_recv(*callback: Callable[[list[Frame]], Any] | Callable[[list[bytes]], Any]*, *copy: bool = True*)

Register a callback for when a message is ready to recv.

There can be only one callback registered at a time, so each call to `on_recv` replaces previously registered callbacks.

on_recv(None) disables recv event polling.

Use on_recv_stream(callback) instead, to register a callback that will receive both this ZMQStream and the message, instead of just the message.

Parameters

- **callback** (*callable*) – callback must take exactly one argument, which will be a list, as returned by socket.recv_multipart() if callback is None, recv callbacks are disabled.
- **copy** (*bool*) – copy is passed directly to recv, so if copy is False, callback will receive Message objects. If copy is True, then callback will receive bytes/str objects.
- **Returns** (*None*) –

on_recv_stream(*callback: Callable[[ZMQStream, list[bytes]], Any]*) → None

on_recv_stream(*callback: Callable[[ZMQStream, list[bytes]], Any]*, *copy: Literal[True]*) → None

on_recv_stream(*callback: Callable[[ZMQStream, list[Frame]], Any]*, *copy: Literal[False]*) → None

on_recv_stream(*callback: Callable[[ZMQStream, list[Frame]], Any] | Callable[[ZMQStream, list[bytes]], Any]*, *copy: bool = True*)

Same as on_recv, but callback will get this stream as first argument

callback must take exactly two arguments, as it will be called as:

```
callback(stream, msg)
```

Useful when a single callback should be used with multiple streams.

on_send(*callback: Callable[[Sequence[Any], MessageTracker | None], Any]*)

Register a callback to be called on each send

There will be two arguments:

```
callback(msg, status)
```

- `msg` will be the list of sendable objects that was just sent
- `status` will be the return result of socket.send_multipart(msg) - MessageTracker or None.

Non-copying sends return a MessageTracker object whose `done` attribute will be True when the send is complete. This allows users to track when an object is safe to write to again.

The second argument will always be None if copy=True on the send.

Use on_send_stream(callback) to register a callback that will be passed this ZMQStream as the first argument, in addition to the other two.

on_send(None) disables recv event polling.

Parameters

callback (*callable*) – callback must take exactly two arguments, which will be the message being sent (always a list), and the return result of socket.send_multipart(msg) - MessageTracker or None.

if callback is None, send callbacks are disabled.

on_send_stream(*callback: Callable[[ZMQStream, Sequence[Any], MessageTracker | None], Any]*)

Same as on_send, but callback will get this stream as first argument

Callback will be passed three arguments:

```
callback(stream, msg, status)
```

Useful when a single callback should be used with multiple streams.

poller: *Poller*

receiving() → bool

Returns True if we are currently receiving from the stream.

send(*msg*, *flags=0*, *copy=True*, *track=False*, *callback=None*, ***kwargs*)

Send a message, optionally also register a new callback for sends. See zmq.socket.send for details.

send_json(*obj: Any*, *flags: int = 0*, *callback: Callable | None = None*, ***kwargs: Any*)

Send json-serialized version of an object. See zmq.socket.send_json for details.

send_multipart(*msg: Sequence[Any]*, *flags: int = 0*, *copy: bool = True*, *track: bool = False*, *callback: Callable | None = None*, ***kwargs: Any*) → None

Send a multipart message, optionally also register a new callback for sends. See zmq.socket.send_multipart for details.

send_pyobj(*obj: Any*, *flags: int = 0*, *protocol: int = -1*, *callback: Callable | None = None*, ***kwargs: Any*)

Send a Python object as a message using pickle to serialize.

See zmq.socket.send_json for details.

send_string(*u: str*, *flags: int = 0*, *encoding: str = 'utf-8'*, *callback: Callable | None = None*, ***kwargs: Any*)

Send a unicode message with an encoding. See zmq.socket.send_unicode for details.

send_unicode(*u: str*, *flags: int = 0*, *encoding: str = 'utf-8'*, *callback: Callable | None = None*, ***kwargs: Any*)

Send a unicode message with an encoding. See zmq.socket.send_unicode for details.

sending() → bool

Returns True if we are currently sending to the stream.

set_close_callback(*callback: Callable | None*)

Call the given callback when the stream is closed.

socket: *Socket*

stop_on_err()

DEPRECATED, does nothing

stop_on_recv()

Disable callback and automatic receiving.

stop_on_send()

Disable callback on sending.

2.1.9 auth

Module: zmq.auth

Utilities for ZAP authentication.

To run authentication in a background thread, see *zmq.auth.thread*. For integration with the asyncio event loop, see *zmq.auth.asyncio*.

Authentication examples are provided in the pyzmq codebase, under /examples/security/.

New in version 14.1.

Authenticator

class zmq.auth.**Authenticator**(*context:* Context | *None* = *None*, *encoding: str* = *'utf-8'*, *log: Any* = *None*)

Implementation of ZAP authentication for zmq connections.

This authenticator class does not register with an event loop. As a result, you will need to manually call handle_zap_message:

```
auth = zmq.Authenticator()
auth.allow("127.0.0.1")
auth.start()
while True:
    await auth.handle_zap_msg(auth.zap_socket.recv_multipart())
```

Alternatively, you can register auth.zap_socket with a poller.

Since many users will want to run ZAP in a way that does not block the main thread, other authentication classes (such as *zmq.auth.thread*) are provided.

Note:

- libzmq provides four levels of security: default NULL (which the Authenticator does not see), and authenticated NULL, PLAIN, CURVE, and GSSAPI, which the Authenticator can see.
- until you add policies, all incoming NULL connections are allowed. (classic ZeroMQ behavior), and all PLAIN and CURVE connections are denied.
- GSSAPI requires no configuration.

allow(**addresses: str*) → None

Allow IP address(es).

Connections from addresses not explicitly allowed will be rejected.

- For NULL, all clients from this address will be accepted.
- For real auth setups, they will be allowed to continue with authentication.

allow is mutually exclusive with deny.

allow_any: bool

certs: Dict[str, Dict[bytes, Any]]

configure_curve(*domain: str = '*', location: str | PathLike = '.'*) → None

Configure CURVE authentication for a given domain.

CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys.

To cover all domains, use "*".

You can add and remove certificates in that directory at any time. configure_curve must be called every time certificates are added or removed, in order to update the Authenticator's state

To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

configure_curve_callback(*domain: str = '*', credentials_provider: Any = None*) → None

Configure CURVE authentication for a given domain.

CURVE authentication using a callback function validating the client public key according to a custom mechanism, e.g. checking the key against records in a db. credentials_provider is an object of a class which implements a callback method accepting two parameters (domain and key), e.g.:

```
class CredentialsProvider(object):

    def __init__(self):
        ...e.g. db connection

    def callback(self, domain, key):
        valid = ...lookup key and/or domain in db
        if valid:
            logging.info('Authorizing: {0}, {1}'.format(domain, key))
            return True
        else:
            logging.warning('NOT Authorizing: {0}, {1}'.format(domain, key))
            return False
```

To cover all domains, use "*".

configure_gssapi(*domain: str = '*', location: str | None = None*) → None

Configure GSSAPI authentication

Currently this is a no-op because there is nothing to configure with GSSAPI.

configure_plain(*domain: str = '*', passwords: Dict[str, str] | None = None*) → None

Configure PLAIN authentication for a given domain.

PLAIN authentication uses a plain-text password file. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

context: *Context*

credentials_providers: Dict[str, Any]

curve_user_id(*client_public_key: bytes*) → str

Return the User-Id corresponding to a CURVE client's public key

Default implementation uses the z85-encoding of the public key.

Override to define a custom mapping of public key : user-id

This is only called on successful authentication.

Parameters

client_public_key (*bytes*) – The client public key used for the given message

Returns
user_id – The user ID as text

Return type
unicode

deny(**addresses: str*) → None
Deny IP address(es).

Addresses not explicitly denied will be allowed to continue with authentication.

deny is mutually exclusive with allow.

encoding: `str`

async handle_zap_message(*msg: List[bytes]*)
Perform ZAP authentication

log: `Any`

passwords: `Dict[str, Dict[str, str]]`

start() → None
Create and bind the ZAP socket

stop() → None
Close the ZAP socket

zap_socket: *Socket*

Functions

zmq.auth.**create_certificates**(*key_dir: str | PathLike*, *name: str*, *metadata: Dict[str, str] | None = None*) → Tuple[str, str]
Create zmq certificates.

Returns the file paths to the public and secret certificate files.

zmq.auth.**load_certificate**(*filename: str | PathLike*) → Tuple[bytes, bytes | None]
Load public and secret key from a zmq certificate.

Returns (public_key, secret_key)

If the certificate file only contains the public key, secret_key will be None.

If there is no public key found in the file, ValueError will be raised.

zmq.auth.**load_certificates**(*directory: str | PathLike = '.'*) → Dict[bytes, bool]
Load public keys from all certificates in a directory

2.1.10 auth.asyncio

Module: zmq.auth.asyncio

ZAP Authenticator integrated with the asyncio IO loop.

New in version 15.2.

Classes

AsyncioAuthenticator

class zmq.auth.asyncio.**AsyncioAuthenticator**(*context:* Context | *None* = *None*, *loop:* *Any* = *None*, *encoding:* *str* = *'utf-8'*, *log:* *Any* = *None*)

ZAP authentication for use in the asyncio IO loop

allow(**addresses:* *str*) → None

Allow IP address(es).

Connections from addresses not explicitly allowed will be rejected.

- For NULL, all clients from this address will be accepted.
- For real auth setups, they will be allowed to continue with authentication.

allow is mutually exclusive with deny.

allow_any: **bool**

certs: **Dict[str, Dict[bytes, Any]]**

configure_curve(*domain:* *str* = *'*'*, *location:* *str* | *PathLike* = *'.'*) → None

Configure CURVE authentication for a given domain.

CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys.

To cover all domains, use "*".

You can add and remove certificates in that directory at any time. configure_curve must be called every time certificates are added or removed, in order to update the Authenticator's state

To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

configure_curve_callback(*domain:* *str* = *'*'*, *credentials_provider:* *Any* = *None*) → None

Configure CURVE authentication for a given domain.

CURVE authentication using a callback function validating the client public key according to a custom mechanism, e.g. checking the key against records in a db. credentials_provider is an object of a class which implements a callback method accepting two parameters (domain and key), e.g.:

```
class CredentialsProvider(object):

    def __init__(self):
        ...e.g. db connection

    def callback(self, domain, key):
        valid = ...lookup key and/or domain in db
        if valid:
```

(continues on next page)

(continued from previous page)

```
            logging.info('Authorizing: {0}, {1}'.format(domain, key))
            return True
        else:
            logging.warning('NOT Authorizing: {0}, {1}'.format(domain, key))
            return False
```

To cover all domains, use "*".

configure_gssapi(*domain: str = '*', location: str | None = None*) → None

Configure GSSAPI authentication

Currently this is a no-op because there is nothing to configure with GSSAPI.

configure_plain(*domain: str = '*', passwords: Dict[str, str] | None = None*) → None

Configure PLAIN authentication for a given domain.

PLAIN authentication uses a plain-text password file. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

context: *zmq.Context*

credentials_providers: Dict[str, Any]

curve_user_id(*client_public_key: bytes*) → str

Return the User-Id corresponding to a CURVE client's public key

Default implementation uses the z85-encoding of the public key.

Override to define a custom mapping of public key : user-id

This is only called on successful authentication.

Parameters
client_public_key (*bytes*) – The client public key used for the given message

Returns
user_id – The user ID as text

Return type
unicode

deny(**addresses: str*) → None

Deny IP address(es).

Addresses not explicitly denied will be allowed to continue with authentication.

deny is mutually exclusive with allow.

encoding: str

async handle_zap_message(*msg: List[bytes]*)

Perform ZAP authentication

log: Any

passwords: Dict[str, Dict[str, str]]

start() → None

Start ZAP authentication

stop() → None

Stop ZAP authentication

zap_socket: *zmq.Socket*

2.1.11 auth.thread

Module: zmq.auth.thread

ZAP Authenticator in a Python Thread.

New in version 14.1.

Classes

ThreadAuthenticator

class zmq.auth.thread.**ThreadAuthenticator**(*context:* Context | *None* = *None*, *encoding:* *str* = *'utf-8'*, *log:* *Any* = *None*)

Run ZAP authentication in a background thread

allow(**addresses:* *str*) → None

Allow IP address(es).

Connections from addresses not explicitly allowed will be rejected.

- For NULL, all clients from this address will be accepted.
- For real auth setups, they will be allowed to continue with authentication.

allow is mutually exclusive with deny.

allow_any: bool

certs: Dict[str, Dict[bytes, Any]]

configure_curve(*domain:* *str* = *'*'*, *location:* *str* | *PathLike* = *'.'*) → None

Configure CURVE authentication for a given domain.

CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys.

To cover all domains, use "*".

You can add and remove certificates in that directory at any time. configure_curve must be called every time certificates are added or removed, in order to update the Authenticator's state

To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

configure_curve_callback(*domain:* *str* = *'*'*, *credentials_provider:* *Any* = *None*) → None

Configure CURVE authentication for a given domain.

CURVE authentication using a callback function validating the client public key according to a custom mechanism, e.g. checking the key against records in a db. credentials_provider is an object of a class which implements a callback method accepting two parameters (domain and key), e.g.:

```
class CredentialsProvider(object):

    def __init__(self):
        ...e.g. db connection

    def callback(self, domain, key):
        valid = ...lookup key and/or domain in db
        if valid:
            logging.info('Authorizing: {0}, {1}'.format(domain, key))
            return True
        else:
            logging.warning('NOT Authorizing: {0}, {1}'.format(domain, key))
            return False
```

To cover all domains, use "*".

configure_gssapi(*domain: str = '*'*, *location: str | None = None*) → None

Configure GSSAPI authentication

Currently this is a no-op because there is nothing to configure with GSSAPI.

configure_plain(*domain: str = '*'*, *passwords: Dict[str, str] | None = None*) → None

Configure PLAIN authentication for a given domain.

PLAIN authentication uses a plain-text password file. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

context: *zmq.Context*

credentials_providers: Dict[str, Any]

curve_user_id(*client_public_key: bytes*) → str

Return the User-Id corresponding to a CURVE client's public key

Default implementation uses the z85-encoding of the public key.

Override to define a custom mapping of public key : user-id

This is only called on successful authentication.

Parameters
client_public_key (*bytes*) – The client public key used for the given message

Returns
user_id – The user ID as text

Return type
unicode

deny(**addresses: str*) → None

Deny IP address(es).

Addresses not explicitly denied will be allowed to continue with authentication.

deny is mutually exclusive with allow.

encoding: str

async handle_zap_message(*msg: List[bytes]*)

Perform ZAP authentication

is_alive() → bool

Is the ZAP thread currently running?

log: Any

passwords: Dict[str, Dict[str, str]]

pipe: *zmq.Socket*

pipe_endpoint: str = ''

start() → None

Start the authentication thread

stop() → None

Stop the authentication thread

thread: *AuthenticationThread*

zap_socket: *zmq.Socket*

class zmq.auth.thread.**AuthenticationThread**(*authenticator:* Authenticator, *pipe:* Socket)

A Thread for running a zmq Authenticator

This is run in the background by ThreadAuthenticator

2.1.12 auth.ioloop

Module: :mod}`zmq.auth.ioloop`

This module is deprecated in pyzmq 25. Use *zmq.auth.asyncio*.

2.1.13 log.handlers

Module: zmq.log.handlers

pyzmq logging handlers.

This mainly defines the PUBHandler object for publishing logging messages over a zmq.PUB socket.

The PUBHandler can be used with the regular logging module, as in:

```
>>> import logging
>>> handler = PUBHandler('tcp://127.0.0.1:12345')
>>> handler.root_topic = 'foo'
>>> logger = logging.getLogger('foobar')
>>> logger.setLevel(logging.DEBUG)
>>> logger.addHandler(handler)
```

Or using `dictConfig`, as in:

```
>>> from logging.config import dictConfig
>>> socket = Context.instance().socket(PUB)
>>> socket.connect('tcp://127.0.0.1:12345')
>>> dictConfig({
```

(continues on next page)

(continued from previous page)

```
>>>     'version': 1,
>>>     'handlers': {
>>>         'zmq': {
>>>             'class': 'zmq.log.handlers.PUBHandler',
>>>             'level': logging.DEBUG,
>>>             'root_topic': 'foo',
>>>             'interface_or_socket': socket
>>>         }
>>>     },
>>>     'root': {
>>>         'level': 'DEBUG',
>>>         'handlers': ['zmq'],
>>>     }
>>> })
```

After this point, all messages logged by `logger` will be published on the PUB socket.

Code adapted from StarCluster:

https://github.com/jtriley/StarCluster/blob/StarCluster-0.91/starcluster/logger.py

Classes

PUBHandler

class zmq.log.handlers.**PUBHandler**(*interface_or_socket: str | Socket*, *context: Context | None = None*, *root_topic: str = ''*)

A basic logging handler that emits log messages through a PUB socket.

Takes a PUB socket already bound to interfaces or an interface to bind to.

Example:

```
sock = context.socket(zmq.PUB)
sock.bind('inproc://log')
handler = PUBHandler(sock)
```

Or:

```
handler = PUBHandler('inproc://loc')
```

These are equivalent.

Log messages handled by this handler are broadcast with ZMQ topics `this.root_topic` comes first, followed by the log level (DEBUG,INFO,etc.), followed by any additional subtopics specified in the message by: log.debug("subtopic.subsub::the real message")

acquire()

Acquire the I/O thread lock.

addFilter(*filter*)

Add the specified filter to this handler.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

ctx: *Context*

emit(*record*)

Emit a log message on my socket.

filter(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()

Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(*record*)

Format a record.

get_name()

handle(*record*)

Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)

Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

property name

release()

Release the I/O thread lock.

removeFilter(*filter*)

Remove the specified filter from this handler.

property root_topic: str

setFormatter(*fmt*, *level=0*)

Set the Formatter for this handler.

If no level is provided, the same format is used for all levels. This will overwrite all selective formatters set in the object constructor.

setLevel(*level*)

Set the logging level of this handler. level must be an int or a str.

setRootTopic(*root_topic: str*)

Set the root topic for this handler.

This value is prepended to all messages published by this handler, and it defaults to the empty string ''. When you subscribe to this socket, you must set your subscription to an empty string, or to at least the first letter of the binary representation of this string to ensure you receive any messages from this handler.

If you use the default empty string root topic, messages will begin with the binary representation of the log level string (INFO, WARN, etc.). Note that ZMQ SUB sockets can have multiple subscriptions.

set_name(*name*)

socket: *Socket*

TopicLogger

class zmq.log.handlers.**TopicLogger**(*name*, *level=0*)

A simple wrapper that takes an additional argument to log methods.

All the regular methods exist, but instead of one msg argument, two arguments: topic, msg are passed.

That is:

```
logger.debug('msg')
```

Would become:

```
logger.debug('topic.sub', 'msg')
```

addFilter(*filter*)

Add the specified filter to this handler.

addHandler(*hdlr*)

Add the specified handler to this logger.

callHandlers(*record*)

Pass a record to all relevant handlers.

Loop through all handlers for this logger and its parents in the logger hierarchy. If no handler was found, output a one-off error message to sys.stderr. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger whose handlers are called.

critical(*level*, *topic*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

debug(*level*, *topic*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)

error(*level*, *topic*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=1)

exception(*msg*, **args*, *exc_info=True*, ***kwargs*)

Convenience method for logging an ERROR with exception information.

fatal(*level*, *topic*, *msg*, **args*, ***kwargs*)

Don't use this method, use critical() instead.

filter(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

findCaller(*stack_info=False*, *stacklevel=1*)

Find the stack frame of the caller so that we can note the source file name, line number and function name.

getChild(*suffix*)

Get a logger which is a descendant to this one.

This is a convenience method, such that

logging.getLogger('abc').getChild('def.ghi')

is the same as

logging.getLogger('abc.def.ghi')

It's useful, for example, when the parent logger is named using __name__ rather than a literal string.

getEffectiveLevel()

Get the effective level for this logger.

Loop through this logger and its parents in the logger hierarchy, looking for a non-zero logging level. Return the first one found.

handle(*record*)

Call the handlers for the specified record.

This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied.

hasHandlers()

See if this logger has any handlers configured.

Loop through all handlers for this logger and its parents in the logger hierarchy. Return True if a handler was found, else False. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger which is checked for the existence of handlers.

info(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "interesting problem", exc_info=1)

isEnabledFor(*level*)

Is this logger enabled for level 'level'?

log(*level*, *topic*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with level and topic.

To pass exception information, use the keyword argument exc_info with a True value:

```
logger.log(level, "zmq.fun", "We have a %s",
        "mysterious problem", exc_info=1)
```

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

A factory method which can be overridden in subclasses to create specialized LogRecords.

manager = <logging.Manager object>

removeFilter(*filter*)

Remove the specified filter from this handler.

removeHandler(*hdlr*)

Remove the specified handler from this logger.

root = <RootLogger root (WARNING)>

setLevel(*level*)

Set the logging level of this logger. level must be an int or a str.

warn(*level*, *topic*, *msg*, **args*, ***kwargs*)

warning(*level*, *topic*, *msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)

2.1.14 ssh.tunnel

Module: `zmq.ssh.tunnel`

Basic ssh tunnel utilities, and convenience functions for tunneling zeromq connections.

Functions

zmq.ssh.tunnel.**open_tunnel**(*addr*, *server*, *keyfile=None*, *password=None*, *paramiko=None*, *timeout=60*)

Open a tunneled connection from a 0MQ url.

For use inside tunnel_connection.

Returns
(url, tunnel) – The 0MQ url that has been forwarded, and the tunnel object

Return type
(str, object)

zmq.ssh.tunnel.**select_random_ports**(*n*)

Select and return n random ports that are available.

zmq.ssh.tunnel.**try_passwordless_ssh**(*server*, *keyfile*, *paramiko=None*)

Attempt to make an ssh connection without a password. This is mainly used for requiring password input only once when many tunnels may be connected to the same server.

If paramiko is None, the default for the platform is chosen.

zmq.ssh.tunnel.**tunnel_connection**(*socket*, *addr*, *server*, *keyfile=None*, *password=None*, *paramiko=None*, *timeout=60*)

Connect a socket to an address via an ssh tunnel.

This is a wrapper for socket.connect(addr), when addr is not accessible from the local machine. It simply creates an ssh tunnel using the remaining args, and calls socket.connect('tcp://localhost:lport') where lport is the randomly selected local port of the tunnel.

2.1.15 utils.jsonapi

Module: zmq.utils.jsonapi

JSON serialize to/from utf8 bytes

Changed in version 22.2: Remove optional imports of different JSON implementations. Now that we require recent Python, unconditionally use the standard library. Custom JSON libraries can be used via custom serialization functions.

Functions

zmq.utils.jsonapi.**dumps**(*o: Any*, ***kwargs*) → bytes

Serialize object to JSON bytes (utf-8).

Keyword arguments are passed along to `json.dumps()`.

zmq.utils.jsonapi.**loads**(*s: bytes | str*, ***kwargs*) → dict | list | str | int | float

Load object from JSON bytes (utf-8).

Keyword arguments are passed along to `json.loads()`.

2.1.16 utils.monitor

Module: zmq.utils.monitor

Module holding utility and convenience functions for zmq event monitoring.

Functions

zmq.utils.monitor.**parse_monitor_message**(*msg: list[bytes]*) → dict

decode zmq_monitor event messages.

Parameters

msg (*list(bytes)*) – zmq multipart message that has arrived on a monitor PAIR socket.

First frame is:

```
16 bit event id
32 bit event value
no padding
```

Second frame is the endpoint as a bytestring

Returns

event – event description as dict with the keys `event`, `value`, and `endpoint`.

Return type

dict

zmq.utils.monitor.**recv_monitor_message**(*socket: Socket*, *flags: int = 0*) → Awaitable[dict]

zmq.utils.monitor.**recv_monitor_message**(*socket: Socket[bytes]*, *flags: int = 0*) → dict

Receive and decode the given raw message from the monitoring socket and return a dict.

Requires libzmq 4.0

The returned dict will have the following entries:

event

[int] the event id as described in `libzmq.zmq_socket_monitor`

value

[int] the event value associated with the event, see `libzmq.zmq_socket_monitor`

endpoint

[str] the affected endpoint

Changed in version 23.1: Support for async sockets added. When called with a async socket, returns an awaitable for the monitor message.

Parameters

- **socket** (*zmq.Socket*) – The PAIR socket (created by other.get_monitor_socket()) on which to recv the message
- **flags** (*int*) – standard zmq recv flags

Returns

event – event description as dict with the keys `event`, `value`, and `endpoint`.

Return type

dict

2.1.17 utils.z85

Module: zmq.utils.z85

Python implementation of Z85 85-bit encoding

Z85 encoding is a plaintext encoding for a bytestring interpreted as 32bit integers. Since the chunks are 32bit, a bytestring must be a multiple of 4 bytes. See ZMQ RFC 32 for details.

Functions

zmq.utils.z85.**decode**(*z85bytes*)

decode Z85 bytes to raw bytes, accepts ASCII string

zmq.utils.z85.**encode**(*rawbytes*)

encode raw bytes into Z85

2.1.18 utils.win32

Module: zmq.utils.win32

Win32 compatibility utilities.

allow_interrupt

class zmq.utils.win32.**allow_interrupt**(*action: Callable[[], Any] | None = None*)

Utility for fixing CTRL-C events on Windows.

On Windows, the Python interpreter intercepts CTRL-C events in order to translate them into `KeyboardInterrupt` exceptions. It (presumably) does this by setting a flag in its "console control handler" and checking it later at a convenient location in the interpreter.

However, when the Python interpreter is blocked waiting for the ZMQ poll operation to complete, it must wait for ZMQ's `select()` operation to complete before translating the CTRL-C event into the `KeyboardInterrupt` exception.

The only way to fix this seems to be to add our own "console control handler" and perform some application-defined operation that will unblock the ZMQ polling operation in order to force ZMQ to pass control back to the Python interpreter.

This context manager performs all that Windows-y stuff, providing you with a hook that is called when a CTRL-C event is intercepted. This hook allows you to unblock your ZMQ poll operation immediately, which will then result in the expected `KeyboardInterrupt` exception.

Without this context manager, your ZMQ-based application will not respond normally to CTRL-C events on Windows. If a CTRL-C event occurs while blocked on ZMQ socket polling, the translation to a `KeyboardInterrupt` exception will be delayed until the I/O completes and control returns to the Python interpreter (this may never happen if you use an infinite timeout).

A no-op implementation is provided on non-Win32 systems to avoid the application from having to conditionally use it.

Example usage:

```
def stop_my_application():
    # ...

with allow_interrupt(stop_my_application):
    # main polling loop.
```

In a typical ZMQ application, you would use the "self pipe trick" to send message to a `PAIR` socket in order to interrupt your blocking socket polling operation.

In a Tornado event loop, you can use the `IOLoop.stop` method to unblock your I/O loop.

2.2 Changes in PyZMQ

This is a coarse summary of changes in pyzmq versions. For a full changelog, consult the git log.

2.2.1 26

pyzmq 26 is a small release, but with some big changes *hopefully* nobody will notice, except for some users (especially on Windows) where pyzmq releases did not work.

The highlights are:

- The Cython backend has been rewritten using Cython 3's pure Python mode.
- The build system has been rewritten to use CMake via scikit-build-core instead of setuptools (setup.py is gone!).
- Bundled libzmq is updated to 4.3.5, which changes its license from LGPL to MPL.

This means:

1. Cython >=3.0 is now a build requirement (if omitted, source distributions *should* still build from Cython-generated .c files without any Cython present)
2. pyzmq's Cython backend is a single extension module, which should improve install size, import time, compile time, etc.
3. pyzmq's Cython backend is now BSD-licensed, matching the rest of pyzmq.
4. The license of the libzmq library (included in pyzmq wheels) starting with 4.3.5 is now Mozilla Public License 2.0 (MPL-2.0).
5. when building pyzmq from source and it falls back on bundled libzmq, libzmq and libsodium are built as static libraries using their own build systems (CMake for libzmq, autotools for libsodium except on Windows where it uses msbuild) rather than bundling libzmq with tweetnacl as a Python Extension.

Since the new build system uses libzmq and libsodium's own build systems, evaluated at install time, building pyzmq with bundled libzmq from source should be much more likely to succeed on a variety of platforms than the previous method, where their build system was skipped and approximated as a Python extension. But I would also be *very* surprised if I didn't break anything in the process of replacing 14 years of setup.py from scratch, especially cases like cross-compiling. Please report any issues you encounter building pyzmq.

See *build docs* for more info.

New:

- Experimental support for wheels on windows-arm64
- `Socket.bind('tcp://ip:0')` can be used as a context manager to bind to a random port. The resulting URL can be retrieved as `socket.last_endpoint`.

- Add `SyncSocket` and `SyncContext` type aliases for the default Socket/Context implementations, since the base classes are Generics, type-wise. These are type aliases only to be used in type checking, not actual classes.

Enhancements:

- `repr(Frame)` now produces a nice repr, summarizing Frame contents (without getting too large), e.g. `<zmq.Frame(b'abcdefghijkl'...52B)>`

Breaking changes:

- `str(Frame)` no longer returns the whole frame contents interpreted as utf8-bytes. Instead, it returns the new summarized repr, which produces more logical results with `print`, etc. `bytes(Frame)` remains unchanged, and utf-8 text strings can still be produced with: `bytes(Frame).decode("utf8")`, which works in all versions of pyzmq and does the same thing.
- Stop building Python 3.7 wheels for manylinux1, which reached EOL in January, 2022. The new build system doesn't seem to be able to find cmake in that environment.

2.2.2 25

25.1.2

- Fix builds with some recent compilers and bundled libzmq
- Fix builds with upcoming Cython 3.1

25.1.1

25.1.1 is the first stable release with Python 3.12 wheels.

Changes:

- Allow Cython 0.29.35 to build Python 3.12 wheels (no longer require Cython 3)

Bugs fixed:

- Fix builds on Solaris by including generated platform.hpp
- Cleanup futures in `Socket.poll()` that are cancelled and never return
- Fix builds with `-j` when numpy is present in the build env

25.1.0

pyzmq 25.1 mostly changes some packaging details of pyzmq, including support for installation from source on Python 3.12 beta 1.

Enhancements:

- Include address in error message when bind/connect fail.

Packaging changes:

- Fix inclusion of some test files in source distributions.
- Add Cython as a build-time dependency in `build-system.requires` metadata, following current recommendations of the Cython maintainers. We still ship generated Cython sources in source distributions, so it is not a *strict* dependency for packagers using `--no-build-isolation`, but pip will install Cython as part of building pyzmq from source. This makes it more likely that past pyzmq releases will install on future Python releases, which often require an update to Cython but not pyzmq itself. For Python 3.12, Cython >=3.0.0b3 is required.

25.0.2

- Fix handling of shadow sockets in ZMQStream when the original sockets have been closed. A regression in 25.0.0, seen with jupyter-client 7.

25.0.1

Tiny bugfix release that should only affect users of *PUBHandler* or pyzmq repackagers.

- Fix handling of custom Message types in *PUBHandler*
- Small lint fixes to satisfy changes in mypy
- License files have been renamed to more standard LICENSE.BSD, LICENSE.LESSER to appease some license auto-detect tools.

25.0.0

New:

- Added `socket_class` argument to *zmq.Context.socket()*
- Support shadowing sockets with socket objects, not just via address, e.g. `zmq.asyncio.Socket(other_socket)`. Shadowing an object preserves a reference to the original, unlike shadowing via address.
- in *zmq.auth*, CredentialsProvider callbacks may now be async.
- *ZMQStream* callbacks may now be async.
- Add *zmq.ReconnectStop* draft constants.
- Add manylinux_2_28 wheels for x86_64 CPython 3.10, 3.11, and PyPy 3.9 (these are *in addition to* not *instead of* the manylinux_2014 wheels).

Fixed:

- When *ZMQStream* is given an async socket, it now warns and hooks up events correctly with the underlying socket, so the callback gets the received message, instead of sending the callback the incorrect arguments.
- Fixed toml parse error in `pyproject.toml`, when installing from source with very old pip.
- Removed expressed dependency on `py` when running with pypy, which hasn't been used in some time.

Deprecated:

- `zmq.auth.ioloop.IOLoopAuthenticator` is deprecated in favor of *zmq.auth.asyncio.AsyncioAuthenticator*
- As part of migrating toward modern pytest, `zmq.tests.BaseZMQTestCase` is deprecated and should not be used outside pyzmq.
- `python setup.py test` is deprecated as a way to launch the tests. Just use `pytest`.

Removed:

- Bundled subset of tornado's IOLoop (deprecated since pyzmq 17) is removed, so ZMQStream cannot be used without an actual install of tornado.
- Remove support for tornado 4, meaning tornado is always assumed to run on asyncio.

2.2.3 24

24.0.1

- Fix several possible resource warnings and deprecation warnings when cleaning up contexts and sockets, especially in pyzmq's own tests and when implicit teardown of objects is happening during process teardown.

24.0.0

pyzmq 24 has two breaking changes (one only on Windows), though they are not likely to affect most users.

Breaking changes:

- Due to a libzmq bug causing unavoidable crashes for some users, Windows wheels no longer bundle libzmq with AF_UNIX support. In order to enable AF_UNIX on Windows, pyzmq must be built from source, linking an appropriate build of libzmq (e.g. `libzmq-v142`). AF_UNIX support will be re-enabled in pyzmq wheels when libzmq published fixed releases.
- Using a *zmq.Context* as a context manager or deleting a context without closing it now calls *zmq.Context.destroy()* at exit instead of *zmq.Context.term()*. This will have little effect on most users, but changes what happens when user bugs result in a context being *implicitly* destroyed while sockets are left open. In almost all cases, this will turn what used to be a hang into a warning. However, there may be some cases where sockets are actively used in threads, which could result in a crash. To use sockets across threads, it is critical to properly and explicitly close your contexts and sockets, which will always avoid this issue.

2.2.4 23.2.1

Improvements:

- First release with wheels for Python 3.11 (thanks cibuildwheel!).
- linux aarch64 wheels now bundle the same libzmq (4.3.4) as all other builds, thanks to switching to native arm builds on CircleCI.

Fixes:

- Some type annotation fixes in devices.

2.2.5 23.2.0

Improvements:

- Use `zmq.Event` enums in `parse_monitor_message` for nicer reprs

Fixes:

- Fix building bundled libzmq with `ZMQ_DRAFT_API=1`
- Fix subclassing `zmq.Context` with additional arguments in the constructor. Subclasses may now have full control over the signature, rather than purely adding keyword-only arguments
- Typos and other small fixes

2.2.6 23.1.0

Fixing some regressions in 23.0:

- Fix global name of `zmq.EVENT_HANDSHAKE_*` constants
- Fix constants missing when using `import zmq.green as zmq`

Compatibility fixes:

- *zmq.utils.monitor.recv_monitor_message()* now supports async Sockets.
- Fix build with mingw

2.2.7 23.0.0

Changes:

- all zmq constants are now available as Python enums (e.g. `zmq.SocketType.PULL`, `zmq.SocketOption.IDENTITY`), generated statically from zmq.h instead of at compile-time. This means that checks for the *presence* of a constant (`hasattr(zmq, 'RADIO')`) is not a valid check for the presence of a feature. This practice has never been robust, but it may have worked sometimes. Use direct checks via e.g. *zmq.has()* or *zmq.zmq_version_info()*.
- A bit more type coverage of Context.term and Context.socket

Compatibility fixes:

- Remove all use of deprecated stdlib distutils
- Update to Cython 0.29.30 (required for Python 3.11 compatibility)
- Compatibility with Python 3.11.0b1

Maintenance changes:

- Switch to myst for docs
- Deprecate `zmq.utils.strtypes`, now unused
- Updates to autoformatting, linting
- New wheels for PyPy 3.9
- Manylinux wheels for CPython 3.10 are based on manylinux2014

2.2.8 22.3.0

Fixes:

- Fix `strlcpy` compilation issues on alpine, freebsd. Adds new build-time dependency on `packaging`.
- In event-loop integration: warn instead of raise when triggering callback on a socket whose context has been closed.
- Bundled libzmq in wheels backport a patch to avoid crashes due to inappropriate closing of libsodium's random generator when using CurveZMQ.

Changes:

- New ResourceWarnings when contexts and sockets are closed by garbage collection, which can be a source of hangs and leaks (matches open files)

2.2.9 22.2.1

Fix bundling of wepoll on Windows.

2.2.10 22.2.0

New features:

- IPC support on Windows: where available (64bit Windows wheels and bundled libzmq when compiling from source, via wepoll), IPC should work on appropriate Windows versions.
- Nicer reprs of contexts and sockets
- Memory allocated by `recv(copy=False)` is no longer read-only
- asyncio: Always reference current loop instead of attaching to the current loop at instantiation time. This fixes e.g. contexts and/or sockets instantiated prior to a call to `asyncio.run`.
- ssh: `$PYZMQ_PARAMIKO_HOST_KEY_POLICY` can be used to set the missing host key policy, e.g. `AutoAdd`.

Fixes:

- Fix memory corruption in gevent integration
- Fix `memoryview(zmq.Frame)` with cffi backend
- Fix threadsafety issue when closing sockets

Changes:

- pypy Windows wheels are 64b-only, following an update in cibuildwheel 2.0
- deprecate `zmq.utils.jsonapi` and remove support for non-stdlib json implementations in `send/recv_json`. Custom serialization methods should be used instead.

2.2.11 22.1.0

New features:

- asyncio: experimental support for Proactor eventloop if tornado 6.1 is available by running a selector in a background thread.

Fixes:

- Windows: fix type of `socket.FD` option in win-amd64
- asyncio: Cancel timers when using HWM with async Sockets

Other changes:

- Windows: update bundled libzmq dll URLs for Windows. Windows wheels no longer include concrt140.dll.
- adopt pre-commit for formatting, linting

2.2.12 22.0.3

- Fix fork-safety bug in garbage collection thread (regression in 20.0) when using subprocesses.
- Start uploading universal wheels for ARM Macs.

2.2.13 22.0.2

- Add workaround for bug in DLL loading for Windows wheels with conda Python >= 3.8

2.2.14 22.0.1

- Fix type of `Frame.bytes` for non-copying recvs with CFFI backend (regression in 21.0)
- Add manylinux wheels for pypy

2.2.15 22.0.0

This is a major release due to changes in wheels and building on Windows. Code changes from 21.0 are minimal.

- Some typing fixes
- Bump bundled libzmq to 4.3.4
- Strip unused symbols in manylinux wheels, resulting in dramatically smaller binaries. This matches behavior in v20 and earlier.
- Windows CPython wheels bundle public libzmq binary builds, instead of building libzmq as a Python Extension. This means they include libsodium for the first time.
- Our own implementation of bundling libzmq into pyzmq on Windows is removed, instead relying on delvewheel (or installations putting dlls on %PATH%) to bundle dependency dlls.
- The (new in 21.0) Windows wheels for PyPy likely require the Windows vcredist package. This may have always been the case, but the delvewheel approach doesn't seem to work.
- Windows + PyPy is now the only remaining case where a wheel has libzmq built as an Extension. All other builds ship libzmq built using its own tooling, which should result in better, more stable builds.

2.2.16 21.0.2

- Fix wheels on macOS older than 10.15 (sets MACOSX_DEPLOYMENT_TARGET to 10.9, matching wheel ABI tag).

2.2.17 21.0.1

pyzmq-21.0.1 only changes CI configuration for Windows wheels (built with VS2017 instead of VS2019), fixing compatibility with some older Windows on all Pythons and removing requirement of VC++ redistributable package on latest Windows and Python < 3.8.

There still appears to be a compatibility issue with Windows 7 that will be fixed ASAP. Until then, you can pin `pip install pyzmq<21`.

There are no changes from 21.0.0 for other platforms.

2.2.18 21.0

pyzmq 21 is a major version bump because of dropped support for old Pythons and some changes in packaging. CPython users should not face major compatibility issues if installation works at all :) PyPy users may see issues with the new implementation of send/recv. If you do, please report them!

The big changes are:

- drop support for Python 3.5. Python >= 3.6 is required
- mypy type stubs, which should improve static analysis of pyzmq, especially for dynamically defined attributes such as zmq constants. These are new! Let us know if you find any issues.
- support for zero-copy and sending bufferables with cffi backend. This is experimental! Please report issues.
- More wheels!
  - linux-aarch64 on Python 3.7-3.9
  - wheels for pypy36, 37 on Linux and Windows (previously just mac)

We've totally redone the wheel-building setup, so let us know if you start seeing installation issues!

Packaging updates:

- Require Python >= 3.6, required for good type annotation support
- Wheels for macOS no longer build libzmq as a Python Extension, instead 'real' libzmq is built and linked to libsodium, bundled with delocate. This matches the longstanding behavior of Linux wheels, and should result in better performance.
- Add manylinux wheels for linux-aarch64. These bundle an older version of libzmq than the rest.
- Build wheels for python3.8, 3.9 with manylinux2010 instead of manylinux1. Wheels for older Pythons will still be built on manylinux1.
- rework cffi backend in setup.py
- All wheels are built on GitHub Actions (most with cibuildwheel) instead of Min's laptop (finally!).

New features:

- zero-copy support in CFFI backend (`send(copy=False)` now does something).
- Support sending any buffer-interface-providing objects in CFFI backend.

Bugs fixed:

- Errors during teardown of asyncio Sockets
- Missing MSVCP140.dll in Python 3.9 wheels on Windows, causing vcruntime-redist package to be required to use the Python 3.9 wheels for pyzmq 20.0

2.2.19 20.0

20.0 is a major version bump because of dropped support for old Pythons and some changes in packaging, but there are only small changes for users with relatively recent versions of Python.

Packaging updates:

- Update bundled libzmq to 4.3.3
- Drop support for Python < 3.5 (all versions of Python < 3.6 are EOL at time of release)
- Require setuptools to build from source

- Require Cython 0.29 to build from version control (sdists still ship .c files, so will never need Cython)
- Respect $PKG_CONFIG env for finding libzmq when building from source

New features:

- *Socket.bind()* and *Socket.connect()* can now be used as context managers.

Fixes:

- Better error when libzmq is bundled and fails to be loaded.
- Hold GIL while calling `zmq_curve_` functions, which may fix apparent threadsafety issues.

2.2.20 19.0.2

- Regenerate Cython sources with 0.29.21 in sdists for compatibility with Python 3.9
- Handle underlying socket being closed in ZMQStream with warning instead of error
- Improvements to socket cleanup during process teardown
- Fix debug-builds on Windows
- Avoid importing ctypes during startup on Windows
- Documentation improvements
- Raise `AttributeError` instead of `ZMQError(EINVAL)` on attempts to read write-only attributes, for compatibility with mocking

2.2.21 19.0.1

- Fix TypeError during garbage collection
- Fix compilation with some C++ compilers
- Fixes in tests and examples

2.2.22 19.0

- Cython backend: Build Cython extensions with language level "3str" (requires Cython 0.29)
- Cython backend: You can now `cimport zmq`
- Asyncio: Fix memory leak in Poller
- Log: Much improved logging in *zmq.log.handlers* (see *Asynchronous Logging via PyZMQ*)
- Log: add `python -m zmq.log` entrypoint
- Sources generated with Cython 0.29.15

2.2.23 18.1.1

- Fix race condition when shutting down ZAP thread while events are still processing (only affects tests)
- Publish wheels for Python 3.8 on all platforms
- Stop publishing wheels for Python 3.4 on Windows
- Sources generated with Cython 0.29.14

2.2.24 18.1.0

- Compatibility with Python 3.8 release candidate by regenerating Cython courses with Cython 0.29.13
- bump bundled libzmq to 4.3.2
- handle cancelled futures in asyncio
- make *zmq.Context.instance()* fork-safe
- fix errors in *zmq.Context.destroy()* when opening and closing many sockets

2.2.25 18.0.2

- Compatibility with Python 3.8 prerelease by regenerating Cython sources with Cython 0.29.10.
- Fix language_level=2 in Cython sources, for compatibility with Cython 0.30
- Show missing path for ENOENT errors on ipc connections.

2.2.26 18.0.1

Fixes installation from source on non-unicode locales with Python 3. There are no code changes in this release.

2.2.27 18.0.0

- Update bundled libzmq to 4.3.1 (fixes CVE-2019-6250)
- Added *proxy_steerable()* and *zmq.devices.ProxySteerable*
- Added `bind_{in|out|mon}_to_random_port` variants for proxy device methods
- Performance improvements for sends with asyncio
- Fix sending memoryviews/bytearrays with cffi backend

2.2.28 17.1.3

- Fix compatibility with tornado 6 (removal of stack_context)

2.2.29 17.1.2

- Fix possible hang when working with asyncio
- Remove some outdated workarounds for old Cython versions
- Fix some compilation with custom compilers
- Remove unneeded link of libstdc++ on PyPy

2.2.30 17.1.0

- Bump bundled libzmq to 4.2.5
- Improve tornado 5.0 compatibility (use `current()` instead of `instance()` to get default loops in *ZMQStream* and `.IOLoopAuthenticator`)
- Add support for *curve_public()*
- Remove delayed import of json in `send/recv_json`
- Add *Authenticator.configure_curve_callback()*
- Various build fixes
- sdist sources generated with Cython 0.28.3
- Stop building wheels for Python 3.4, start building wheels for Python 3.7

2.2.31 17.0.0

- Add *zmq.Socket.send_serialized()* and *zmq.Socket.recv_serialized()* for sending/receiving messages with custom serialization.
- Add *zmq.Socket.copy_threshold* and *zmq.COPY_THRESHOLD*. Messages smaller than this are always copied, regardless of `copy=False`, to avoid overhead of zero-copy bookkeeping on small messages.
- Added visible deprecation warnings to bundled tornado IOLoop. Tornado eventloop integration shouldn't be used without a proper tornado install since pyzmq 14.
- Allow pyzmq asyncio/tornado integration to run without installing `zmq_poll` implementation. The following methods and classes are deprecated and no longer required:
  - `zmq.eventloop.ioloop.install`
  - `zmq.eventloop.ioloop.IOLoop`
  - `zmq.asyncio.install`
  - `zmq.asyncio.ZMQEventLoop`
- Set RPATH correctly when building on macOS.
- Compatibility fixes with tornado 5.0.dev (may not be quite enough for 5.0 final, which is not yet released as of pyzmq 17).
- Draft support for CLIENT-SERVER `routing_id` and `group`.

  See also:

  Working with libzmq DRAFT sockets

2.2.32 16.0.4

- Regenerate Cython sources in sdists with Cython 0.27.3, fixing builds on CPython 3.7.
- Add warning when using bundled tornado, which was deprecated too quietly in 14.x.

2.2.33 16.0.3

- Regenerate Cython sources in sdists with Cython 0.27.2, fixing builds on CPython 3.7.

2.2.34 16.0.2

- Workaround bug in libzmq-4.2.0 causing EINVAL on poll.

2.2.35 16.0.1

- Fix erroneous EAGAIN that could happen on async sockets
- Bundle libzmq 4.1.6

2.2.36 16.0

- Support for Python 2.6 and Python 3.2 is dropped. For old Pythons, use **`pip install "pyzmq<16"`** to get the last version of pyzmq that supports these versions.
- Include zmq.h
- Deprecate `zmq.Stopwatch`. Native Python timing tools can be used instead.
- Better support for using pyzmq as a Cython library
  - bundle zmq.h when pyzmq bundles libzmq as an extension
  - add *zmq.get_library_dirs()* to find bundled libzmq
- Updates to setup.py for Cython 0.25 compatibility
- Various asyncio/future fixes:
  - support raw sockets in pollers
  - allow cancelling async sends
- Fix `IOLoop.current()` in *zmq.green*

2.2.37 15.4

- Load bundled libzmq extension with import rather than CDLL, which should fix some manifest issues in certain cases on Windows.
- Avoid installing asyncio sources on Python 2, which confuses some tools that run `python -m compileall`, which reports errors on the Python 3-only files.
- Bundle msvcp.dll in Windows wheels on CPython 3.5, which should fix wheel compatibility systems without Visual C++ 2015 redistributable.
- *zmq.Context.instance()* is now threadsafe.

- FIX: sync some behavior in zmq_poll and setting LINGER on close/destroy with the CFFI backend.
- PERF: resolve send/recv immediately if events are available in async Sockets
- Async Sockets (asyncio, tornado) now support `send_json`, `send_pyobj`, etc.
- add preliminary support for `zmq.DRAFT_API` reflecting ZMQ_BUILD_DRAFT_API, which indicates whether new APIs in prereleases are available.

2.2.38 15.3

- Bump bundled libzmq to 4.1.5, using tweetnacl for bundled curve support instead of libsodium
- FIX: include .pxi includes in installation for consumers of Cython API
- FIX: various fixes in new async sockets
- Introduce *zmq.decorators* API for decorating functions to create sockets or contexts
- Add *zmq.Socket.subscribe()* and *zmq.Socket.unsubscribe()* methods to sockets, so that assignment is no longer needed for subscribing. Verbs should be methods! Assignment is still supported for backward-compatibility.
- Accept text (unicode) input to z85 encoding, not just bytes
- *zmq.Context.socket()* forwards keyword arguments to the *Socket* constructor

2.2.39 15.2

- FIX: handle multiple events in a single register call in *zmq.asyncio*
- FIX: unicode/bytes bug in password prompt in `zmq.ssh` on Python 3
- FIX: workaround gevent monkeypatches in garbage collection thread
- update bundled minitornado from tornado-4.3.
- improved inspection by setting `binding=True` in cython compile options
- add asyncio Authenticator implementation in *zmq.auth.asyncio*
- workaround overflow bug in libzmq preventing receiving messages larger than `MAX_INT`

2.2.40 15.1

- FIX: Remove inadvertent tornado dependency when using *zmq.asyncio*
- FIX: 15.0 Python 3.5 wheels didn't work on Windows
- Add GSSAPI support to Authenticators
- Support new constants defined in upcoming libzmq-4.2.dev

2.2.41 15.0

PyZMQ 15 adds Future-returning sockets and pollers for both `asyncio` and `tornado.concurrent`.

- add `asyncio` support via *zmq.asyncio*
- add `tornado.concurrent` future support via *zmq.eventloop.future*
- trigger bundled libzmq if system libzmq is found to be < 3. System libzmq 2 can be forced by explicitly requesting `--zmq=/prefix/`.

2.2.42 14.7.0

Changes:

- Update bundled libzmq to 4.1.2.
- Following the lead of Python 3.5, interrupted system calls will be retried.

Fixes:

- Fixes for CFFI backend on Python 3 + support for PyPy 3.
- Verify types of all frames in *send_multipart()* before sending, to avoid partial messages.
- Fix build on Windows when both debug and release versions of libzmq are found.
- Windows build fixes for Python 3.5.

2.2.43 14.6.0

Changes:

- improvements in *zmq.Socket.bind_to_random_port()*: : - use system to allocate ports by default
  - catch EACCES on Windows
- include libsodium when building bundled libzmq on Windows (includes wheels on PyPI)
- pyzmq no longer bundles external libzmq when making a bdist. You can use delocate to do this.

Bugfixes:

- add missing `ndim` on memoryviews of Frames
- allow `copy.copy()` and `copy.deepcopy()` on Sockets, Contexts

2.2.44 14.5.0

Changes:

- use pickle.DEFAULT_PROTOCOL by default in send_pickle
- with the release of pip-6, OS X wheels are only marked as 10.6-intel, indicating that they should be installable on any newer or single-arch Python.
- raise SSHException on failed check of host key

Bugfixes:

- fix method name in utils.wi32.allow_interrupt
- fork-related fixes in garbage collection thread

- add missing import in `zmq.__init__`, causing failure to import in some circumstances

2.2.45 14.4.1

Bugfixes for 14.4

- SyntaxError on Python 2.6 in zmq.ssh
- Handle possible bug in garbage collection after fork

2.2.46 14.4.0

New features:

- Experimental support for libzmq-4.1.0 rc (new constants, plus *zmq.has()*).
- Update bundled libzmq to 4.0.5
- Update bundled libsodium to 1.0.0
- Fixes for SSH dialogs when using *zmq.ssh.tunnel* to create tunnels
- More build/link/load fixes on OS X and Solaris
- Get Frame metadata via dict access (libzmq 4)
- Contexts and Sockets are context managers (term/close on `__exit__`)
- Add *zmq.utils.win32.allow_interrupt* context manager for catching SIGINT on Windows

Bugs fixed:

- Bundled libzmq should not trigger recompilation after install on PyPy

2.2.47 14.3.1

Note: pyzmq-14.3.1 is the last version to include bdists for Python 3.3

Minor bugfixes to pyzmq 14.3:

- Fixes to building bundled libzmq on OS X < 10.9
- Fixes to import-failure warnings on Python 3.4
- Fixes to tests
- Pull upstream fixes to zmq.ssh for ssh multiplexing

2.2.48 14.3.0

- PyZMQ no longer calls *Socket.close()* or *Context.term()* during process cleanup. Changes to garbage collection in Python 3.4 make this impossible to do sensibly.
- *ZMQStream.close()* closes its socket immediately, rather than scheduling a timeout.
- Raise the original ImportError when importing zmq fails. Should be more informative than `no module cffi. ...`

Warning: Users of Python 3.4 should not use pyzmq < 14.3, due to changes in garbage collection.

2.2.49 14.2.0

New Stuff

- Raise new ZMQVersionError when a requested method is not supported by the linked libzmq. For backward compatibility, this subclasses NotImplementedError.

Bugs Fixed

- Memory leak introduced in pyzmq-14.0 in zero copy.
- OverflowError on 32 bit systems in zero copy.

2.2.50 14.1.0

Security

The headline features for 14.1 are adding better support for libzmq's security features.

- When libzmq is bundled as a Python extension (e.g. wheels, eggs), libsodium is also bundled (excluding Windows), ensuring that libzmq security is available to users who install from wheels
- New *zmq.auth*, implementing zeromq's ZAP authentication, modeled on czmq zauth. For more information, see the examples.

Other New Stuff

- Add PYZMQ_BACKEND for enabling use of backends outside the pyzmq codebase.
- Add *underlying* property and *shadow()* method to Context and Socket, for handing off sockets and contexts. between pyzmq and other bindings (mainly pyczmq).
- Add TOS, ROUTER_HANDOVER, and IPC_FILTER constants from libzmq-4.1-dev.
- Add Context option support in the CFFI backend.
- Various small unicode and build fixes, as always.
- *send_json()* and *recv_json()* pass any extra kwargs to `json.dumps/loads`.

Deprecations

- `Socket.socket_type` is deprecated, in favor of `Socket.type`, which has been available since 2.1.

2.2.51 14.0.1

Bugfix release

- Update bundled libzmq to current (4.0.3).
- Fix bug in *Context.destroy()* with no open sockets.
- Threadsafety fixes in the garbage collector.
- Python 3 fixes in *zmq.ssh.tunnel*.

2.2.52 14.0.0

- Update bundled libzmq to current (4.0.1).
- Backends are now implemented in `zmq.backend` instead of `zmq.core`. This has no effect on public APIs.
- Various build improvements for Cython and CFFI backends (PyPy compiles at build time).
- Various GIL-related performance improvements - the GIL is no longer touched from a zmq IO thread.
- Adding a constant should now be a bit easier - only zmq/sugar/constant_names should need updating, all other constant-related files should be automatically updated by `setup.py constants`.
- add support for latest libzmq-4.0.1 (includes ZMQ_CURVE security and socket event monitoring).

New stuff

- *Socket.monitor()*
- *Socket.get_monitor_socket()*
- *zmq.curve_keypair()*
- *zmq.utils.monitor*
- *zmq.utils.z85*

2.2.53 13.1.0

The main new feature is improved tornado 3 compatibility. PyZMQ ships a 'minitornado' submodule, which contains a small subset of tornado 3.0.1, in order to get the IOLoop base class. zmq.eventloop.ioloop.IOLoop is now a simple subclass, and if the system tornado is 3.0, then the zmq IOLoop is a proper registered subclass of the tornado one itself, and minitornado is entirely unused.

2.2.54 13.0.2

Bugfix release!

A few things were broken in 13.0.0, so this is a quick bugfix release.

- **FIXED** EAGAIN was unconditionally turned into KeyboardInterrupt
- **FIXED** we used totally deprecated ctypes_configure to generate constants in CFFI backend
- **FIXED** memory leak in CFFI backend for PyPy
- **FIXED** typo prevented IPC_PATH_MAX_LEN from ever being defined
- **FIXED** various build fixes - linking with librt, Cython compatibility, etc.

2.2.55 13.0.1

defunct bugfix. We do not speak of this…

2.2.56 13.0.0

PyZMQ now officially targets libzmq-3 (3.2.2), 0MQ 2.1.4 is still supported for the indefinite future, but 3.x is recommended. PyZMQ has detached from libzmq versioning, and will just follow its own regular versioning scheme from now on. PyZMQ bdists will include whatever is the latest stable libzmq release (3.2.2 for pyzmq-13.0).

Note: set/get methods are exposed via get/setattr on all Context, Socket, and Frame classes. This means that subclasses of these classes that require extra attributes **must declare these attributes at the class level**.

Experiments Removed

- The Threadsafe ZMQStream experiment in 2.2.0.1 was deemed inappropriate and not useful, and has been removed.
- The `zmq.web` experiment has been removed, to be developed as a standalone project.

New Stuff

- Support for PyPy via CFFI backend (requires py, ctypes-configure, and cffi).
- Add support for new APIs in libzmq-3
  - *Socket.disconnect()*
  - *Socket.unbind()*
  - *Context.set()*
  - *Context.get()*
  - *Frame.set()*
  - *Frame.get()*
  - *zmq.proxy()*
  - *zmq.devices.Proxy*

  - Exceptions for common zmq errnos: *zmq.Again*, *zmq.ContextTerminated* (subclass *ZMQError*, so fully backward-compatible).

- Setting and getting *Socket.hwm* sets or gets *both* SNDHWM/RCVHWM for libzmq-3.
- Implementation splits core Cython bindings from pure-Python subclasses with sugar methods (send/recv_multipart). This should facilitate non-Cython backends and PyPy support [spoiler: it did!].

Bugs Fixed

- Unicode fixes in log and monitored queue
- MinGW, ppc, cross-compilation, and HP-UX build fixes
- *zmq.green* should be complete - devices and tornado eventloop both work in gevent contexts.

2.2.57 2.2.0.1

This is a tech-preview release, to try out some new features. It is expected to be short-lived, as there are likely to be issues to iron out, particularly with the new pip-install support.

Experimental New Stuff

These features are marked 'experimental', which means that their APIs are not set in stone, and may be removed or changed in incompatible ways in later releases.

Threadsafe ZMQStream

With the IOLoop inherited from tornado, there is exactly one method that is threadsafe: `add_callback()`. With this release, we are trying an experimental option to pass all IOLoop calls via this method, so that ZMQStreams can be used from one thread while the IOLoop runs in another. To try out a threadsafe stream:

```
stream = ZMQStream(socket, threadsafe=True)
```

pip install pyzmq

PyZMQ should now be pip installable, even on systems without libzmq. In these cases, when pyzmq fails to find an appropriate libzmq to link against, it will try to build libzmq as a Python extension. This work is derived from pyzmq_static.

To this end, PyZMQ source distributions include the sources for libzmq (2.2.0) and libuuid (2.21), both used under the LGPL.

zmq.green

The excellent gevent_zeromq socket subclass which provides gevent compatibility has been merged as *zmq.green*.

See also:

zmq.green

Bugs Fixed

- TIMEO sockopts are properly included for libzmq-2.2.0
- avoid garbage collection of sockets after fork (would cause `assert (mailbox.cpp:79)`).

2.2.58 2.2.0

Some effort has gone into refining the pyzmq API in this release to make it a model for other language bindings. This is principally made in a few renames of objects and methods, all of which leave the old name for backwards compatibility.

Note: As of this release, all code outside `zmq.core` is BSD licensed (where possible), to allow more permissive use of less-critical code and utilities.

Name Changes

- The `Message` class has been renamed to *Frame*, to better match other zmq bindings. The old Message name remains for backwards-compatibility. Wherever pyzmq docs say "Message", they should refer to a complete zmq atom of communication (one or more Frames, connected by ZMQ_SNDMORE). Please report any remaining instances of Message==MessagePart with an Issue (or better yet a Pull Request).
- All `foo_unicode` methods are now called `foo_string` (`_unicode` remains for backwards compatibility). This is not only for cross-language consistency, but it makes more sense in Python 3, where native strings are unicode, and the `_unicode` suffix was wedded too much to Python 2.

Other Changes and Removals

- `prefix` removed as an unused keyword argument from *send_multipart()*.
- ZMQStream *send()* default has been changed to `copy=True`, so it matches Socket *send()*.
- ZMQStream *on_err()* is deprecated, because it never did anything.
- Python 2.5 compatibility has been dropped, and some code has been cleaned up to reflect no-longer-needed hacks.
- Some Cython files in `zmq.core` have been split, to reduce the amount of Cython-compiled code. Much of the body of these files were pure Python, and thus did not benefit from the increased compile time. This change also aims to ease maintaining feature parity in other projects, such as pyzmq-ctypes.

New Stuff

- *Context* objects can now set default options when they create a socket. These are set and accessed as attributes to the context. Socket options that do not apply to a socket (e.g. SUBSCRIBE on non-SUB sockets) will simply be ignored.
- *on_recv_stream()* has been added, which adds the stream itself as a second argument to the callback, making it easier to use a single callback on multiple streams.
- A `Frame.more` boolean attribute has been added to the *Frame* (née Message) class, so that frames can be identified as terminal without extra queries of `Socket.rcvmore`.

Experimental New Stuff

These features are marked 'experimental', which means that their APIs are not set in stone, and may be removed or changed in incompatible ways in later releases.

- `zmq.web` added for load-balancing requests in a tornado webapp with zeromq.

2.2.59 2.1.11

- remove support for LABEL prefixes. A major feature of libzmq-3.0, the LABEL prefix, has been removed from libzmq, prior to the first stable libzmq 3.x release.
  - The prefix argument to *send_multipart()* remains, but it continue to behave in exactly the same way as it always has on 2.1.x, simply prepending message parts.
  - *recv_multipart()* will always return a list, because prefixes are once again indistinguishable from regular message parts.
- add *zmq.Socket.poll()* method, for simple polling of events on a single socket.
- no longer require monkeypatching tornado IOLoop. The `ioloop.ZMQPoller` class is a poller implementation that matches tornado's expectations, and pyzmq sockets can be used with any tornado application just by specifying the use of this poller. The pyzmq IOLoop implementation now only trivially differs from tornado's.

  It is still recommended to use `ioloop.install()`, which sets *both* the zmq and tornado global IOLoop instances to the same object, but it is no longer necessary.

  Warning: The most important part of this change is that the `IOLoop.READ/WRITE/ERROR` constants now match tornado's, rather than being mapped directly to the zmq `POLLIN/OUT/ERR`. So applications that used the low-level `IOLoop.add_handler` code with `POLLIN/OUT/ERR` directly (used to work, but was incorrect), rather than using the IOLoop class constants will no longer work. Fixing these to use the IOLoop constants should be insensitive to the actual value of the constants.

2.2.60 2.1.10

- Add support for libzmq-3.0 LABEL prefixes:

  Warning: This feature has been removed from libzmq, and thus removed from future pyzmq as well.

  - send a message with label-prefix with:

    ```
    send_multipart([b"msg", b"parts"], prefix=[b"label", b"prefix"])
    ```

  - *zmq.Socket.recv_multipart()* returns a tuple of `(prefix,msg)` if a label prefix is detected
  - ZMQStreams and devices also respect the LABEL prefix
- add czmq-style close&term as *zmq.Context.destroy()*, so that *zmq.Context.term()* remains threadsafe and 1:1 with libzmq.
- *zmq.Socket.close()* takes optional linger option, for setting linger prior to closing.
- add *zmq_version_info()* and *pyzmq_version_info()* for getting libzmq and pyzmq versions as tuples of numbers. This helps with the fact that version string comparison breaks down once versions get into double-digits.
- ioloop changes merged from upstream Tornado 2.1

2.2.61 2.1.9

- added zmq.ssh tools for tunneling socket connections, copied from IPython
- Expanded sockopt support to cover changes in libzmq-4.0 dev.
- Fixed an issue that prevented `KeyboardInterrupt` from being catchable.
- Added attribute-access for set/getsockopt. Setting/Getting attributes of *Socket*s with the names of socket options is mapped to calls of set/getsockopt.

```
s.hwm = 10
s.identity = b"whoda"
s.linger
# -1
```

- Terminating a *Context* closes the sockets it created, matching the behavior in czmq.
- *ThreadDevice*s use *zmq.Context.instance()* to create sockets, so they can use inproc connections to sockets in other threads.
- fixed units error on *zmq.select()*, where the poll timeout was 1000 times longer than expected.
- Add missing `DEALER/ROUTER` socket type names (currently aliases, to be replacements for `XREP/XREQ`).
- base libzmq dependency raised to 2.1.4 (first stable release) from 2.1.0.

2.2.62 2.1.7.1

- bdist for 64b Windows only. This fixed a type mismatch on the `ZMQ_FD` sockopt that only affected that platform.

2.2.63 2.1.7

- Added experimental support for libzmq-3.0 API
- Add `zmq.eventloop.ioloop.install` for using pyzmq's IOLoop in a tornado application.

2.2.64 2.1.4

- First version with binary distribution support
- Added *zmq.Context.instance()* method for using a single Context throughout an application without passing references around.

2.3 Using PyZMQ

2.3.1 Building pyzmq

pyzmq publishes around a hundred wheels for each release, so hopefully very few folks need to build pyzmq from source.

pyzmq 26 has a whole new build system using CMake via scikit-build-core.

~all options can be specified via environment variables with the same name, in order to play nicely with pip.

Installing from source

When compiling pyzmq, it is generally recommended that zeromq be installed separately, via homebrew, apt, yum, etc:

```
# Debian-based
sudo apt-get install libzmq3-dev

# Fedora-based
sudo yum install libzmq3-devel

# homebrew
brew install zeromq
```

You can install pyzmq from source with `pip` by telling it `--no-binary pyzmq`:

```
python3 -m pip install pyzmq --no-binary pyzmq
```

or an editable install from a local checkout:

```
python3 -m pip install -e .
```

Building from source uses CMake via scikit-build-core. CMake >= 3.28 is required. scikit-build-core will attempt to download cmake if a satisfactory version is not found.

Examples

First, some quick examples of influencing pyzmq's build.

Build a wheel against already-installed libzmq:

```
export ZMQ_PREFIX=/usr/local
python3 -m pip install pyzmq --no-binary pyzmq
```

Force building bundled libzmq with the draft API:

```
export ZMQ_PREFIX=bundled
export ZMQ_BUILD_DRAFT=1
python3 -m pip install pyzmq --no-binary pyzmq
```

Finding libzmq

First, pyzmq tries to find libzmq to link against it.

pyzmq will first try to search using standard CMake methods, followed by pkg-config.

You can pass through arguments to the build system via the CMAKE_ARGS environment variable. e.g.

```
CMAKE_ARGS="-DCMAKE_PREFIX_PATH=/path/to/something"
```

or

```
PKG_CONFIG_PATH="$PREFIX/lib/pkgconfig"
```

If pyzmq doesn't find your libzmq via the default search, or you want to skip the search and tell pyzmq exactly where to look, set ZMQ_PREFIX (this skips cmake/pkgconfig entirely):

```
ZMQ_PREFIX=/path/to/zmq  # should contain 'include', 'lib', etc.
```

Disabling bundled build fallback

You may want to keep the default search, which will import targets from CMake, pkg-config, etc., but make *sure* libzmq is found.

To do this, set PYZMQ_NO_BUNDLE. If you set only this, pyzmq will still search via standard means, but *fail* if libzmq is not found, rather than falling back on the bundled static library.

```
-DPYZMQ_NO_BUNDLE=ON
```

Building bundled libzmq

If pyzmq doesn't find a libzmq to link to, it will fall back on building libzmq itself. You can tell pyzmq to skip searching for libzmq and always build the bundled version with `ZMQ_PREFIX=bundled`.

When building a bundled libzmq, pyzmq downloads and builds libzmq and libsodium as static libraries. These static libraries are then linked to by the pyzmq extension and discarded.

Bundled libzmq is supported on a best-effort basis, and isn't expected to work everywhere with zero configuration. If you have trouble building bundled libzmq, please do report it. But the best solution is usually to install libzmq yourself via the appropriate mechanism *before* building pyzmq.

Building bundled libsodium

libsodium is built first, with `configure` most places:

```
./configure --enable-static --disable-shared --with-pic
make
make install
```

or `msbuild` on Windows:

```
msbuild /m /v:n /p:Configuration=StaticRelease /pPlatform=x64 builds/msvc/vs2022/
↪libsodium.sln
```

You can *add* arguments to configure with a semicolon-separated list, by specifying `PYZMQ_LIBSODIUM_CONFIGURE_ARGS` variable:

```
PYZMQ_LIBSODIUM_CONFIGURE_ARGS="--without-pthread --enable-minimal"
# or
CMAKE_ARGS="-DPYZMQ_LIBSODIUM_CONFIGURE_ARGS=--without-pthread;--enable-minimal"
```

and `PYZMQ_LIBSODIUM_MSBUILD_ARGS` on Windows:

```
PYZMQ_LIBSODIUM_MSBUILD_ARGS="/something /else"
# or
CMAKE_ARGS="-DPYZMQ_LIBSODIUM_MSBUILD_ARGS=/something;/else"
```

Note: command-line arguments from environment variables are expected to be space-separated (`-a -b`), while CMake variables are expected to be CMake lists (semicolon-separated) (`-a;-b`).

Building bundled libzmq

The `libzmq-static` static library target is imported via FetchContent, which means the libzmq CMake build is used on all platforms. This means that configuring the build of libzmq itself is done directly via CMAKE_ARGS, and all of libzmq's cmake flags should be available. See libzmq's install docs for more.

For example, to enable OpenPGM:

```
CMAKE_ARGS="-DWITH_OPENPGM=ON"
```

Specifying bundled versions

You can specify which version of libsodium/libzmq to bundle with:

```
-DPYZMQ_LIBZMQ_VERSION=4.3.5
-DPYZMQ_LIBSODIUM_VERSION=1.0.19
```

or the specify the full URL to download (e.g. to test bundling an unreleased version):

```
-DPYZMQ_LIBZMQ_URL="https://github.com/zeromq/libzmq/releases/download/v4.3.5/zeromq-4.3.
↪5.tar.gz"
-DPYZMQ_LIBSODIUM_URL="https://download.libsodium.org/libsodium/releases/libsodium-1.0.
↪19.tar.gz"
```

Warning: Only the default versions are supported and there is no guarantee that bundling versions will work, but you are welcome to try!

Windows notes

I'm not at all confident in building things on Windows, but so far things work in CI. I've done my best to expose options to allow users to override things if they don't work, but it's not really meant to be customizable; it's meant to allow you to workaround my mistakes without waiting for a release.

libsodium ships several solutions for msbuild, identified by `/builds/msvc/vs{year}/libsodium.sln`. pyzmq tries to guess which solution to use based on the MSVC_VERSION CMake variable, but you can skip the guess by specifying `-D PYZMQ_LIBSODIUM_VS_VERSION=2022` to explicitly use the vs2022 solution.

Passing arguments

pyzmq has a few CMake options to influence the build. All options are settable as environment variables, as well. Other than `ZMQ_PREFIX` and `ZMQ_DRAFT_API` which have been around forever, environment variables for building pyzmq have the prefix `PYZMQ_`.

The `_ARGS` variables that are meant to pass-through command-line strings accept standard command-line format from environment, or semicolon-separated lists when specified directly to cmake.

So

```
export ZMQ_PREFIX=bundled
export PYZMQ_LIBZMQ_VERSION=4.3.4
export PYZMQ_LIBSODIUM_CONFIGURE_ARGS=--disable-pie --minimal

python3 -m build .
```

is equivalent to

```
export CMAKE_ARGS="-DZMQ_PREFIX=bundled -DPYZMQ_LIBZMQ_VERSION=4.3.4 -DPYZMQ_LIBSODIUM_
↪CONFIGURE_ARGS=--disable-pie;--minimal"
python3 -m build .
```

Most cmake options can be seen below:

`cmake -LH` output for pyzmq, which can be passed via `CMAKE_ARGS`. Most of these can also be specified via environment variables.

```
# Path to a program.
CYTHON:FILEPATH=$PREFIX/bin/cython

# semicolon-separated list of arguments to pass to ./configure for bundled libsodium
PYZMQ_LIBSODIUM_CONFIGURE_ARGS:STRING=

# semicolon-separated list of arguments to pass to msbuild for bundled libsodium
PYZMQ_LIBSODIUM_MSBUILD_ARGS:STRING=

# full URL to download bundled libsodium
PYZMQ_LIBSODIUM_URL:STRING=

# libsodium version when bundling
PYZMQ_LIBSODIUM_VERSION:STRING=1.0.19

# Visual studio solution version for bundled libsodium (default: detect from MSVC_
↪VERSION)
PYZMQ_LIBSODIUM_VS_VERSION:STRING=

# full URL to download bundled libzmq
PYZMQ_LIBZMQ_URL:STRING=

# libzmq version when bundling
PYZMQ_LIBZMQ_VERSION:STRING=4.3.5

# Prohibit building bundled libzmq. Useful for repackaging, to allow default search for␣
↪libzmq and requiring it to succeed.
PYZMQ_NO_BUNDLE:BOOL=OFF

# whether to build the libzmq draft API
ZMQ_DRAFT_API:BOOL=OFF

# libzmq installation prefix or 'bundled'
ZMQ_PREFIX:STRING=auto

# The directory containing a CMake configuration file for ZeroMQ.
ZeroMQ_DIR:PATH=$PREFIX/lib/cmake/ZeroMQ
```

`cmake -LH` output for libzmq, showing additional arguments that can be passed to CMAKE_ARGS when building bundled libzmq

```
# Path to a program.
A2X_EXECUTABLE:FILEPATH=A2X_EXECUTABLE-NOTFOUND

# Choose polling system for zmq_poll(er)_*. valid values are
# poll or select [default=poll unless POLLER=select]
API_POLLER:STRING=

# Whether or not to build the shared object
BUILD_SHARED:BOOL=ON

```

(continues on next page)

(continued from previous page)

```
# Whether or not to build the static archive
BUILD_STATIC:BOOL=ON

# Whether or not to build the tests
BUILD_TESTS:BOOL=ON

# Build with static analysis(make take very long)
ENABLE_ANALYSIS:BOOL=OFF

# Build with address sanitizer
ENABLE_ASAN:BOOL=OFF

# Run tests that require sudo and capsh (for cap_net_admin)
ENABLE_CAPSH:BOOL=OFF

# Include Clang
ENABLE_CLANG:BOOL=ON

# Enables cpack rules
ENABLE_CPACK:BOOL=ON

# Enable CURVE security
ENABLE_CURVE:BOOL=OFF

# Build and install draft classes and methods
ENABLE_DRAFTS:BOOL=ON

# Enable/disable eventfd
ENABLE_EVENTFD:BOOL=OFF

# Build using compiler intrinsics for atomic ops
ENABLE_INTRINSICS:BOOL=OFF

# Automatically close libsodium randombytes. Not threadsafe without getrandom()
ENABLE_LIBSODIUM_RANDOMBYTES_CLOSE:BOOL=ON

# Build with empty ZMQ_EXPORT macro, bypassing platform-based automated detection
ENABLE_NO_EXPORT:BOOL=OFF

# Enable precompiled headers, if possible
ENABLE_PRECOMPILED:BOOL=ON

# Use radix tree implementation to manage subscriptions
ENABLE_RADIX_TREE:BOOL=ON

# Build with thread sanitizer
ENABLE_TSAN:BOOL=OFF

# Build with undefined behavior sanitizer
ENABLE_UBSAN:BOOL=OFF

# Enable WebSocket transport
```

(continues on next page)

(continued from previous page)

```
ENABLE_WS:BOOL=ON

#
LIBZMQ_PEDANTIC:BOOL=ON

#
LIBZMQ_WERROR:BOOL=OFF

# Choose polling system for I/O threads. valid values are
# kqueue, epoll, devpoll, pollset, poll or select [default=autodetect]
POLLER:STRING=

# Path to a library.
RT_LIBRARY:FILEPATH=RT_LIBRARY-NOTFOUND

# Build html docs
WITH_DOCS:BOOL=ON

# Use libbsd instead of builtin strlcpy
WITH_LIBBSD:BOOL=ON

# Use libsodium
WITH_LIBSODIUM:BOOL=OFF

# Use static libsodium library
WITH_LIBSODIUM_STATIC:BOOL=OFF

# Enable militant assertions
WITH_MILITANT:BOOL=OFF

# Build with support for NORM
WITH_NORM:BOOL=OFF

# Use NSS instead of builtin sha1
WITH_NSS:BOOL=OFF

# Build with support for OpenPGM
WITH_OPENPGM:BOOL=OFF

# Build with perf-tools
WITH_PERF_TOOL:BOOL=ON

# Use TLS for WSS support
WITH_TLS:BOOL=ON

# Build with support for VMware VMCI socket
WITH_VMCI:BOOL=OFF

# install path for ZeroMQConfig.cmake
ZEROMQ_CMAKECONFIG_INSTALL_DIR:STRING=lib/cmake/ZeroMQ

# ZeroMQ library
```

(continues on next page)

(continued from previous page)

```
ZEROMQ_LIBRARY:STRING=libzmq

# Build as OS X framework
ZMQ_BUILD_FRAMEWORK:BOOL=OFF

# Build the tests for ZeroMQ
ZMQ_BUILD_TESTS:BOOL=ON

# Choose condition_variable_t implementation. Valid values are
# stl11, win32api, pthreads, none [default=autodetect]
ZMQ_CV_IMPL:STRING=stl11

# Output zmq library base name
ZMQ_OUTPUT_BASENAME:STRING=zmq
```

Cross-compiling pyzmq

Cross-compiling Python extensions is tricky!

To cross-compile pyzmq, in general you need:

- Python built for the 'build' machine
- Python built for the 'host' machine (identical version)
- cross-compiling toolchain (e.g. `aarch64-linux-gnu-gcc`)
- Python setup to cross-compile (crossenv is the popular tool these days, and includes lots of info for cross-compiling for Python, but pyzmq makes no assumptions)

It is probably a good idea to build libzmq/libsodium separately and link them with ZMQ_PREFIX, as cross-compiling bundled libzmq is not guaranteed to work.

I don't have a lot of experience cross-compiling, but we have two example Dockerfiles that appear to work to cross-compile pyzmq. These aren't official or supported, but they appear to work and may be useful as reference to get you started.

```
FROM ubuntu:22.04
RUN apt-get -y update \
 && apt-get -y install curl unzip cmake ninja-build openssl xz-utils build-essential␣
↪libz-dev libssl-dev

ENV BUILD_PREFIX=/opt/build
ENV PATH=${BUILD_PREFIX}/bin:$PATH

ARG PYTHON_VERSION=3.11.8
WORKDIR /src
RUN curl -L -o python.tar.xz https://www.python.org/ftp/python/${PYTHON_VERSION}/Python-$
↪{PYTHON_VERSION}.tar.xz \
 && tar -xf python.tar.xz \
 && rm python.tar.xz \
 && mv Python-* cpython

# build our 'build' python
```

(continues on next page)

(continued from previous page)

```
WORKDIR /src/cpython
RUN ./configure --prefix=${BUILD_PREFIX}
RUN make -j4
RUN make install

# sanity check
RUN python3 -c 'import ssl' \
 && python3 -m ensurepip \
 && python3 -m pip install --upgrade pip

# get our cross-compile toolchain
# I'm on aarch64, so use x86_64 as host
ENV BUILD="aarch64-linux-gnu"
ENV HOST="x86_64-linux-gnu"
RUN HOST_PKG=$(echo $HOST | sed s@_@-@g) \
 && apt-get -y install binutils-$HOST_PKG gcc-$HOST_PKG g++-$HOST_PKG
ENV CC=$HOST-gcc \
    CXX=$HOST-g++

# build our 'host' python
WORKDIR /src/cpython
RUN make clean
ENV HOST_PREFIX=/opt/host
RUN ./configure \
    --prefix=${HOST_PREFIX} \
    --host=$HOST \
    --build=$BUILD \
    --with-build-python=$BUILD_PREFIX/bin/python3 \
    --without-ensurepip \
    ac_cv_buggy_getaddrinfo=no \
    ac_cv_file__dev_ptmx=yes \
    ac_cv_file__dev_ptc=no
RUN make -j4
RUN make install

WORKDIR /src

# # (optional) cross-compile libsodium, libzmq
# WORKDIR /src
# ENV LIBSODIUM_VERSION=1.0.19
# RUN curl -L -O "https://download.libsodium.org/libsodium/releases/libsodium-$
↪{LIBSODIUM_VERSION}.tar.gz" \
#  && tar -xzf libsodium-${LIBSODIUM_VERSION}.tar.gz \
#  && mv libsodium-stable libsodium \
#  && rm libsodium*.tar.gz
#
# WORKDIR /src/libsodium
# RUN ./configure --prefix="${HOST_PREFIX}" --host=$HOST
# RUN make -j4
# RUN make install
#
# # build libzmq
```

(continues on next page)

(continued from previous page)

```
# WORKDIR /src
# ENV LIBZMQ_VERSION=4.3.5
# RUN curl -L -O "https://github.com/zeromq/libzmq/releases/download/v${LIBZMQ_VERSION}/
↪zeromq-${LIBZMQ_VERSION}.tar.gz" \
#  && tar -xzf zeromq-${LIBZMQ_VERSION}.tar.gz \
#  && mv zeromq-${LIBZMQ_VERSION} zeromq
# WORKDIR /src/zeromq
# RUN ./configure --prefix="$HOST_PREFIX" --host=$HOST --disable-perf --disable-Werror --
↪without-docs --enable-curve --with-libsodium=$HOST_PREFIX --disable-drafts --disable-
↪libsodium_randombytes_close
# RUN make -j4
# RUN make install

# setup crossenv
WORKDIR /src
ENV CROSS_PREFIX=/opt/cross
RUN python3 -m pip install crossenv \
 && python3 -m crossenv ${HOST_PREFIX}/bin/python3 ${CROSS_PREFIX}
ENV PATH=${CROSS_PREFIX}/bin:$PATH

# install build dependencies in crossenv
RUN . ${CROSS_PREFIX}/bin/activate \
 && build-pip install build pyproject_metadata scikit-build-core pathspec cython

# if pyzmq is bundling libsodium, tell it to cross-compile
# not required if libzmq is already installed
ENV PYZMQ_LIBSODIUM_CONFIGURE_ARGS="--host $HOST"
ARG PYZMQ_VERSION=26.0.0b2
# build wheel of pyzmq
WORKDIR /src
RUN python3 -m pip download --no-binary pyzmq --pre pyzmq==$PYZMQ_VERSION \
 && tar -xzf pyzmq-*.tar.gz \
 && rm pyzmq-*.tar.gz \
 && . ${CROSS_PREFIX}/bin/activate \
 && cross-python -m build --no-isolation --skip-dependency-check --wheel ./pyzmq*

# there is now a pyzmq wheel in /src/pyzmq-$version/dist/pyzmq-$VERSION-cp311-cp311-
↪linux_x86_64.whl
```

```
FROM ubuntu:22.04
RUN apt-get -y update \
 && apt-get -y install curl unzip cmake ninja-build openssl xz-utils build-essential␣
↪libz-dev libssl-dev

ENV BUILD_PREFIX=/opt/build
ENV PATH=${BUILD_PREFIX}/bin:$PATH

ARG PYTHON_VERSION=3.11.8
WORKDIR /src
RUN curl -L -o python.tar.xz https://www.python.org/ftp/python/${PYTHON_VERSION}/Python-$
↪{PYTHON_VERSION}.tar.xz \
 && tar -xf python.tar.xz \
```

(continues on next page)

(continued from previous page)

```
 && rm python.tar.xz \
 && mv Python-* cpython

# build our 'build' python
WORKDIR /src/cpython
RUN ./configure --prefix=${BUILD_PREFIX}
RUN make -j4
RUN make install

# sanity check
RUN python3 -c 'import ssl' \
 && python3 -m ensurepip \
 && python3 -m pip install --upgrade pip

# get our cross-compile toolchain from NDK
WORKDIR /opt
RUN curl -L -o ndk.zip https://dl.google.com/android/repository/android-ndk-r26c-linux.
↪zip \
 && unzip ndk.zip \
 && rm ndk.zip \
 && mv android-* ndk
ENV BUILD="x86_64-linux-gnu"
ENV HOST="aarch64-linux-android34"
ENV PATH=/opt/ndk/toolchains/llvm/prebuilt/linux-x86_64/bin:$PATH
ENV CC=$HOST-clang \
    CXX=$HOST-clang++ \
    READELF=llvm-readelf

# build our 'host' python
WORKDIR /src/cpython
RUN make clean
ENV HOST_PREFIX=/opt/host
RUN ./configure \
    --prefix=${HOST_PREFIX} \
    --host=$HOST \
    --build=$BUILD \
    --with-build-python=$BUILD_PREFIX/bin/python3 \
    --without-ensurepip \
    ac_cv_buggy_getaddrinfo=no \
    ac_cv_file__dev_ptmx=yes \
    ac_cv_file__dev_ptc=no
RUN make -j4
RUN make install

# (optional) cross-compile libsodium, libzmq
WORKDIR /src
ENV LIBSODIUM_VERSION=1.0.19
RUN curl -L -O "https://download.libsodium.org/libsodium/releases/libsodium-${LIBSODIUM_
↪VERSION}.tar.gz" \
 && tar -xzf libsodium-${LIBSODIUM_VERSION}.tar.gz \
 && mv libsodium-stable libsodium \
 && rm libsodium*.tar.gz
```

(continues on next page)

(continued from previous page)

```
WORKDIR /src/libsodium
# need CFLAGS or libsodium >= 1.0.20 https://github.com/android/ndk/issues/1945
ENV CFLAGS="-march=armv8-a+crypto"
RUN ./configure --prefix="${HOST_PREFIX}" --host=$HOST
RUN make -j4
RUN make install

# build libzmq
WORKDIR /src
ENV LIBZMQ_VERSION=4.3.5
RUN curl -L -O "https://github.com/zeromq/libzmq/releases/download/v${LIBZMQ_VERSION}/
↪zeromq-${LIBZMQ_VERSION}.tar.gz" \
 && tar -xzf zeromq-${LIBZMQ_VERSION}.tar.gz \
 && mv zeromq-${LIBZMQ_VERSION} zeromq
WORKDIR /src/zeromq
RUN ./configure --prefix="$HOST_PREFIX" --host=$HOST --disable-perf --disable-Werror --
↪without-docs --enable-curve --with-libsodium=$HOST_PREFIX --disable-drafts --disable-
↪libsodium_randombytes_close
RUN make -j4
RUN make install

# setup crossenv
ENV CROSS_PREFIX=/opt/cross
RUN python3 -m pip install crossenv \
 && python3 -m crossenv ${HOST_PREFIX}/bin/python3 ${CROSS_PREFIX}
ENV PATH=${CROSS_PREFIX}/bin:$PATH

# install build dependencies in crossenv
RUN . ${CROSS_PREFIX}/bin/activate \
 && build-pip install build pyproject_metadata scikit-build-core pathspec cython

ENV ZMQ_PREFIX=${HOST_PREFIX}
# if pyzmq is bundling libsodium, tell it to cross-compile
# not required if libzmq is already installed
ENV PYZMQ_LIBSODIUM_CONFIGURE_ARGS="--host $HOST"
ARG PYZMQ_VERSION=26.0.0b2
# build wheel of pyzmq
WORKDIR /src
RUN python3 -m pip download --no-binary pyzmq --pre pyzmq==$PYZMQ_VERSION \
 && tar -xzf pyzmq-*.tar.gz \
 && rm pyzmq-*.tar.gz \
 && . ${CROSS_PREFIX}/bin/activate \
 && cross-python -m build --no-isolation --skip-dependency-check --wheel ./pyzmq*

# there is now a pyzmq wheel in /src/pyzmq-$VERSION/dist/pyzmq-$VERSION-cp311-cp311-
↪linux_aarch64.whl
```

2.3.2 More Than Just Bindings

PyZMQ is ostensibly the Python bindings for ØMQ, but the project, following Python's 'batteries included' philosophy, provides more than just Python methods and objects for calling into the ØMQ C++ library.

The Core as Bindings

PyZMQ is currently broken up into subpackages. First, is the Backend. `zmq.backend` contains the actual bindings for ZeroMQ, and no extended functionality beyond the very basics required. This is the *compiled* portion of pyzmq, either with Cython (for CPython) or CFFI (for PyPy).

Thread Safety

In ØMQ, Contexts are threadsafe objects, but Sockets are **not**. It is safe to use a single Context (e.g. via *zmq.Context.instance()*) in your entire multithreaded application, but you should create sockets on a per-thread basis. If you share sockets across threads, you are likely to encounter uncatchable c-level crashes of your application unless you use judicious application of `threading.Lock`, but this approach is not recommended.

See also:

ZeroMQ API note on threadsafety on 2.2 or 3.2

Socket Options as Attributes

New in version 2.1.9.

In 0MQ, socket options are set/retrieved with the `set/getsockopt()` methods. With the class-based approach in pyzmq, it would be logical to perform these operations with simple attribute access, and this has been added in pyzmq 2.1.9. Simply assign to or request a Socket attribute with the (case-insensitive) name of a sockopt, and it should behave just as you would expect:

```
s = ctx.socket(zmq.DEALER)
s.identity = b"dealer"
s.hwm = 10
s.events
# 0
s.fd
# 16
```

Default Options on the Context

New in version 2.1.11.

Just like setting socket options as attributes on Sockets, you can do the same on Contexts. This affects the default options of any *new* sockets created after the assignment.

```
ctx = zmq.Context()
ctx.linger = 0
rep = ctx.socket(zmq.REP)
req = ctx.socket(zmq.REQ)
```

Socket options that do not apply to a socket (e.g. SUBSCRIBE on non-SUB sockets) will simply be ignored.

libzmq constants as Enums

New in version 23.

libzmq constants are now available as Python enums, making it easier to enumerate socket options, etc.

Context managers

New in version 14: Context/Sockets as context managers

New in version 20: bind/connect context managers

For more Pythonic resource management, contexts and sockets can be used as context managers. Just like standard-library socket and file methods, entering a context:

```
import zmq

with zmq.Context() as ctx:
    with ctx.socket(zmq.PUSH) as s:
        s.connect(url)
        s.send_multipart([b"message"])
    # exiting Socket context closes socket
# exiting Context context terminates context
```

In addition, each bind/connect call may be used as a context:

```
with socket.connect(url):
    s.send_multipart([b"message"])
# exiting connect context calls socket.disconnect(url)
```

Core Extensions

We have extended the core functionality in some ways that appear inside the `zmq.sugar` layer, and are not general ØMQ features.

Builtin Serialization

First, we added common serialization with the builtin `json` and `pickle` as first-class methods to the *`Socket`* class. A socket has the methods *`send_json()`* and *`send_pyobj()`*, which correspond to sending an object over the wire after serializing with `json` and `pickle` respectively, and any object sent via those methods can be reconstructed with the *`recv_json()`* and *`recv_pyobj()`* methods. Unicode strings are other objects that are not unambiguously sendable over the wire, so we include *`send_string()`* and *`recv_string()`* that simply send bytes after encoding the message ('utf-8' is the default).

See also:

- *Further information* on serialization in pyzmq.

MessageTracker

The second extension of basic ØMQ functionality is the *MessageTracker*. The MessageTracker is an object used to track when the underlying ZeroMQ is done with a message buffer. One of the main use cases for ØMQ in Python is the ability to perform non-copying sends. Thanks to Python's buffer interface, many objects (including NumPy arrays) provide the buffer interface, and are thus directly sendable. However, as with any asynchronous non-copying messaging system like ØMQ or MPI, it can be important to know when the message has actually been sent, so it is safe again to edit the buffer without worry of corrupting the message. This is what the MessageTracker is for.

The MessageTracker is a simple object, but there is a penalty to its use. Since by its very nature, the MessageTracker must involve threadsafe communication (specifically a builtin `Queue` object), instantiating a MessageTracker takes a modest amount of time (10s of µs), so in situations instantiating many small messages, this can actually dominate performance. As a result, tracking is optional, via the `track` flag, which is optionally passed, always defaulting to `False`, in each of the three places where a Frame object (the pyzmq object for wrapping a segment of a message) is instantiated: The *Frame* constructor, and non-copying sends and receives.

A MessageTracker is very simple, and has just one method and one attribute. The property *MessageTracker.done* will be `True` when the Frame(s) being tracked are no longer in use by ØMQ, and *MessageTracker.wait()* will block, waiting for the Frame(s) to be released.

Note: A Frame cannot be tracked after it has been instantiated without tracking. If a Frame is to even have the *option* of tracking, it must be constructed with `track=True`.

Extensions

So far, PyZMQ includes four extensions to core ØMQ that we found basic enough to be included in PyZMQ itself:

- *zmq.log* : Logging handlers for hooking Python logging up to the network
- *zmq.devices* : Custom devices and objects for running devices in the background
- *zmq.eventloop* : The Tornado event loop, adapted for use with ØMQ sockets.
- *zmq.ssh* : Simple tools for tunneling zeromq connections via ssh.

2.3.3 Serializing messages with PyZMQ

When sending messages over a network, you often need to marshall your data into bytes.

Builtin serialization

PyZMQ is primarily bindings for libzmq, but we do provide three builtin serialization methods for convenience, to help Python developers learn libzmq. Python has two primary packages for serializing objects: `json` and `pickle`, so we provide simple convenience methods for sending and receiving objects serialized with these modules. A socket has the methods *send_json()* and *send_pyobj()*, which correspond to sending an object over the wire after serializing with json and pickle respectively, and any object sent via those methods can be reconstructed with the *recv_json()* and *recv_pyobj()* methods.

These methods designed for convenience, not for performance, so developers who want to emphasize performance should use their own serialized send/recv methods.

Using your own serialization

In general, you will want to provide your own serialization that is optimized for your application or library availability. This may include using your own preferred serialization ([1],[2]), or adding compression via[3] in the standard library, or the super fast[4] library.

There are two simple models for implementing your own serialization: write a function that takes the socket as an argument, or subclass Socket for use in your own apps.

For instance, pickles can often be reduced substantially in size by compressing the data. The following will send *compressed* pickles over the wire:

```
import pickle
import zlib


def send_zipped_pickle(socket, obj, flags=0, protocol=pickle.HIGHEST_PROTOCOL):
    """pickle an object, and zip the pickle before sending it"""
    p = pickle.dumps(obj, protocol)
    z = zlib.compress(p)
    return socket.send(z, flags=flags)


def recv_zipped_pickle(socket, flags=0):
    """inverse of send_zipped_pickle"""
    z = socket.recv(flags)
    p = zlib.decompress(z)
    return pickle.loads(p)
```

A common data structure in Python is the numpy array. PyZMQ supports sending numpy arrays without copying any data, since they provide the Python buffer interface. However just the buffer is not enough information to reconstruct the array on the receiving side. Here is an example of a send/recv that allow non-copying sends/recvs of numpy arrays including the dtype/shape data necessary for reconstructing the array.

```
import numpy


def send_array(socket, A, flags=0, copy=True, track=False):
    """send a numpy array with metadata"""
    md = dict(
        dtype=str(A.dtype),
        shape=A.shape,
    )
    socket.send_json(md, flags | zmq.SNDMORE)
    return socket.send(A, flags, copy=copy, track=track)


def recv_array(socket, flags=0, copy=True, track=False):
    """recv a numpy array"""
    md = socket.recv_json(flags=flags)
    msg = socket.recv(flags=flags, copy=copy, track=track)
```

(continues on next page)

[1] Message Pack serialization library https://msgpack.org
[2] Google Protocol Buffers https://github.com/protocolbuffers/protobuf
[3] Python stdlib module for zip compression: `zlib`
[4] Blosc: A blocking, shuffling and loss-less (and crazy-fast) compression library https://www.blosc.org

(continued from previous page)

```
    buf = memoryview(msg)
    A = numpy.frombuffer(buf, dtype=md["dtype"])
    return A.reshape(md["shape"])
```

2.3.4 Devices in PyZMQ

See also:

ØMQ Guide Device coverage.

ØMQ has a notion of Devices - simple programs that manage a send-recv pattern for connecting two or more sockets. Being full programs, devices include a `while(True)` loop and thus block execution permanently once invoked. We have provided in the *devices* subpackage some facilities for running these devices in the background, as well as a custom three-socket *MonitoredQueue* device.

BackgroundDevices

It seems fairly rare that in a Python program one would actually want to create a zmq device via *device()* in the main thread, since such a call would block execution forever. The most likely model for launching devices is in background threads or processes. We have provided classes for launching devices in a background thread with *ThreadDevice* and via multiprocessing with *ProcessDevice*. For threadsafety and running across processes, these methods do not take Socket objects as arguments, but rather socket types, and then the socket creation and configuration happens via the BackgroundDevice's `foo_in()` proxy methods. For each configuration method (bind/connect/setsockopt), there are proxy methods for calling those methods on the Socket objects created in the background thread or process, prefixed with 'in_' or 'out_', corresponding to the `in_socket` and `out_socket`:

```
from zmq.devices import ProcessDevice

pd = ProcessDevice(zmq.QUEUE, zmq.ROUTER, zmq.DEALER)
pd.bind_in('tcp://*:12345')
pd.connect_out('tcp://127.0.0.1:12543')
pd.setsockopt_in(zmq.IDENTITY, 'ROUTER')
pd.setsockopt_out(zmq.IDENTITY, 'DEALER')
pd.start()
# it will now be running in a background process
```

MonitoredQueue

One of ØMQ's builtin devices is the `QUEUE`. This is a symmetric two-socket device that fully supports passing messages in either direction via any pattern. We saw a logical extension of the `QUEUE` as one that behaves in the same way with respect to the in/out sockets, but also sends every message in either direction *also* on a third `monitor` socket. For performance reasons, this *monitored_queue()* function is written in Cython, so the loop does not involve Python, and should have the same performance as the basic `QUEUE` device.

One shortcoming of the `QUEUE` device is that it does not support having `ROUTER` sockets as both input and output. This is because `ROUTER` sockets, when they receive a message, prepend the `IDENTITY` of the socket that sent the message (for use in routing the reply). The result is that the output socket will always try to route the incoming message back to the original sender, which is presumably not the intended pattern. In order for the queue to support a ROUTER-ROUTER connection, it must swap the first two parts of the message in order to get the right message out the other side.

To invoke a monitored queue is similar to invoking a regular ØMQ device:

```
from zmq.devices import monitored_queue
ins = ctx.socket(zmq.ROUTER)
outs = ctx.socket(zmq.DEALER)
mons = ctx.socket(zmq.PUB)
configure_sockets(ins,outs,mons)
monitored_queue(ins, outs, mons, in_prefix='in', out_prefix='out')
```

The `in_prefix` and `out_prefix` default to 'in' and 'out' respectively, and a PUB socket is most logical for the monitor socket, since it will never receive messages, and the in/out prefix is well suited to the PUB/SUB topic subscription model. All messages sent on `mons` will be multipart, the first part being the prefix corresponding to the socket that received the message.

Or for launching an MQ in the background, there are *ThreadMonitoredQueue* and *ProcessMonitoredQueue*, which function just like the base BackgroundDevice objects, but add `foo_mon()` methods for configuring the monitor socket.

2.3.5 Eventloops and PyZMQ

As of pyzmq 17, integrating pyzmq with eventloops should work without any pre-configuration. Due to the use of an edge-triggered file descriptor, this has been known to have issues, so please report problems with eventloop integration.

AsyncIO

PyZMQ 15 adds support for `asyncio` via *zmq.asyncio*, containing a Socket subclass that returns `asyncio.Future` objects for use in `asyncio` coroutines. To use this API, import *zmq.asyncio.Context*. Sockets created by this Context will return Futures from any would-be blocking method.

```
import asyncio
import zmq
from zmq.asyncio import Context

ctx = Context.instance()


async def recv():
    s = ctx.socket(zmq.SUB)
    s.connect("tcp://127.0.0.1:5555")
    s.subscribe(b"")
    while True:
        msg = await s.recv_multipart()
        print("received", msg)
    s.close()
```

Tornado IOLoop

Tornado adds some utility on top of asyncio. You can use *zmq.asyncio* socket in a tornado application without any special handling.

We have adapted tornado's IOStream class into *ZMQStream* for handling message events on ØMQ sockets. A ZMQStream object works much like a Socket object, but instead of calling *recv()* directly, you register a callback with *on_recv_stream()*, which will be called with the result of ~.zmq.Socket.recv_multipart. Callbacks can also be registered for send events with *on_send()*.

ZMQStream

ZMQStream objects let you register callbacks to handle messages as they arrive, for use with the tornado eventloop.

ZMQStream.send()

ZMQStream objects do have *send()* and *send_multipart()* methods, which behaves the same way as *zmq.Socket.send()*, but instead of sending right away, the IOLoop will wait until socket is able to send (for instance if HWM is met, or a REQ/REP pattern prohibits sending at a certain point). Messages sent via send will also be passed to the callback registered with *on_send()* after sending.

on_recv()

ZMQStream.on_recv() is the primary method for using a ZMQStream. It registers a callback to fire with messages as they are received, which will *always* be multipart, even if its length is 1. You can easily use this to build things like an echo socket:

```
s = ctx.socket(zmq.REP)
s.bind("tcp://localhost:12345")
stream = ZMQStream(s)


def echo(msg):
    stream.send_multipart(msg)


stream.on_recv(echo)
ioloop.IOLoop.instance().start()
```

on_recv can also take a copy flag, just like *zmq.Socket.recv()*. If copy=False, then callbacks registered with on_recv will receive tracked *Frame* objects instead of bytes.

Note: A callback must be registered using either *ZMQStream.on_recv()* or *ZMQStream.on_recv_stream()* before any data will be received on the underlying socket. This allows you to temporarily pause processing on a socket by setting both callbacks to None. Processing can later be resumed by restoring either callback.

on_recv_stream()

ZMQStream.on_recv_stream() is just like on_recv above, but the callback will be passed both the message and the stream, rather than just the message. This is meant to make it easier to use a single callback with multiple streams.

```
s1 = ctx.socket(zmq.REP)
s1.bind("tcp://localhost:12345")
stream1 = ZMQStream(s1)

s2 = ctx.socket(zmq.REP)
s2.bind("tcp://localhost:54321")
stream2 = ZMQStream(s2)


def echo(stream, msg):
    stream.send_multipart(msg)


stream1.on_recv_stream(echo)
stream2.on_recv_stream(echo)

ioloop.IOLoop.instance().start()
```

flush()

Sometimes with an eventloop, there can be multiple events ready on a single iteration of the loop. The *ZMQStream.flush()* method allows developers to pull messages off of the queue to enforce some priority over the event loop ordering. flush pulls any pending events off of the queue. You can specify to flush only recv events, only send events, or any events, and you can specify a limit for how many events to flush in order to prevent starvation.

PyZMQ and gevent

PyZMQ 2.2.0.1 ships with a gevent compatible API as *zmq.green*. To use it, simply:

```
import zmq.green as zmq
```

Then write your code as normal.

Socket.send/recv and zmq.Poller are gevent-aware.

In PyZMQ 2.2.0.2, green.device and green.eventloop should be gevent-friendly as well.

Note: The green device does *not* release the GIL, unlike the true device in zmq.core.

---

zmq.green.eventloop includes minimally patched IOLoop/ZMQStream in order to use the gevent-enabled Poller, so you should be able to use the ZMQStream interface in gevent apps as well, though using two eventloops simultaneously (tornado + gevent) is not recommended.

Warning: There is a known issue in gevent 1.0 or libevent, which can cause zeromq socket events to be missed. PyZMQ works around this by adding a timeout so it will not wait forever for gevent to notice events. The only known solution for this is to use gevent 1.0, which is currently at 1.0b3, and does not exhibit this behavior.

See also:

zmq.green examples on GitHub.

zmq.green began as gevent_zeromq, merged into the pyzmq project.

2.3.6 Working with libzmq DRAFT sockets

libzmq-4.2 has introduced the concept of unstable DRAFT APIs. As of libzmq-4.2, this includes the CLIENT-SERVER and RADIO-DISH patterns.

Because these APIs are explicitly unstable, pyzmq does not support them by default, and pyzmq binaries (wheels) will not be built with DRAFT API support. However, pyzmq can be built with draft socket support, as long as you compile pyzmq yourself with a special flag.

To install libzmq with draft support:

```
ZMQ_VERSION=4.3.5
PREFIX=/usr/local
CPU_COUNT=${CPU_COUNT:-$(python3 -c "import os; print(os.cpu_count())")}


wget https://github.com/zeromq/libzmq/releases/download/v${ZMQ_VERSION}/zeromq-${ZMQ_
↪VERSION}.tar.gz -O libzmq.tar.gz
tar -xzf libzmq.tar.gz
cd zeromq-${ZMQ_VERSION}
./configure --prefix=${PREFIX} --enable-drafts
make -j${CPU_COUNT} && make install
```

And then build pyzmq with draft support:

```
export ZMQ_PREFIX=${PREFIX}
export ZMQ_DRAFT_API=1
pip install -v pyzmq --no-binary pyzmq
```

By specifying `--no-binary pyzmq`, pip knows to not install the pre-built wheels, and will compile pyzmq from source.

The `ZMQ_PREFIX=$PREFIX` part is only necessary if libzmq is installed somewhere not on the default search path. If libzmq is installed in `/usr/local` or similar, only the `ZMQ_DRAFT_API` option is required.

There are examples of the CLIENT-SERVER and RADIO-DISH patterns in the `examples/draft` directory of the pyzmq repository.

2.3.7 Asynchronous Logging via PyZMQ

See also:

- The ØMQ guide coverage of PUB/SUB messaging
- Python logging module documentation

Python provides extensible logging facilities through its `logging` module. This module allows for easily extensible logging functionality through the use of `Handler` objects. The most obvious case for hooking up pyzmq to logging would be to broadcast log messages over a PUB socket, so we have provided a *`PUBHandler`* class for doing just that.

You can use PyZMQ as a log handler with no previous knowledge of how ZMQ works, and without writing any ZMQ-specific code in your Python project.

Getting Started

Ensure you have installed the pyzmq package from pip, ideally in a virtual environment you created for your project:

```
pip install pyzmq
```

Next, configure logging in your Python module and setup the ZMQ log handler:

```
import logging
from zmq.log.handlers import PUBHandler

zmq_log_handler = PUBHandler('tcp://127.0.0.1:12345')
logger = logging.getLogger()
logger.addHandler(zmq_log_handler)
```

Usually, you will add the handler only once in the top-level module of your project, on the root logger, just as we did here.

You can choose any IP address and port number that works on your system. We used `tcp://127.0.0.1:12345` to broadcast events via TCP on the localhost interface at port 12345. Make note of what you choose here as you will need it later when you listen to the events.

Logging messages works exactly like normal. This will send an INFO-level message on the logger we configured above, and that message will be published on a ZMQ PUB/SUB socket:

```
logger.info('hello world!')
```

You can use this module's built-in command line interface to "tune in" to messages broadcast by the log handler. To start the log watcher, run this command from a shell that has access to the pyzmq package (usually a virtual environment):

```
python -m zmq.log tcp://127.0.0.1:12345
```

Then, in a separate process, run your Python module that emits log messages. You should see them appear almost immediately.

Using the Log Watcher

The included log watcher command line utility is helpful not only for viewing messages, but also a programming guide to build your own ZMQ subscriber for log messages.

To see what options are available, pass the `--help` parameter:

```
python -m zmq.log --help
```

The log watcher includes features to add a timestamp to the messages, align the messages across different error levels, and even colorize the output based on error level.

Slow Joiner Problem

The great thing about using ZMQ sockets is that you can start the publisher and subscribers in any order, and you can start & stop any of them while you leave the others running.

When using ZMQ for logging, this means you can leave the log watcher running while you start & stop your main Python module.

However, you need to be aware of what the ZMQ project calls the "slow joiner problem" . To oversimplify, it means it can take a bit of time for subscribers to re-connect to a publisher that has just started up again. If the publisher starts and immediately sends a message, subscribers will likely miss it.

The simplistic workaround when using PyZMQ for logging is to `sleep()` briefly after startup, before sending any log messages. See the complete example below for more details.

Custom Log Formats

A common Python logging recipe encourages use of the current module name as the name of the logger. This allows your log messages to reflect your code hierarchy in a larger project with minimal configuration.

You will need to set a different formatter to see these names in your ZMQ-published logs. The setFormatter() method accepts a logging.Formatter instance and optionally a log level to apply the handler to. For example:

```
zmq_log_handler = PUBHandler('tcp://127.0.0.1:12345')
zmq_log_handler.setFormatter(logging.Formatter(fmt='{name} > {message}', style='{'))
zmq_log_handler.setFormatter(logging.Formatter(fmt='{name} #{lineno:>3} > {message}',␣
↪style='{'), logging.DEBUG)
```

Root Topic

By default, the PUBHandler and log watcher use the empty string as the root topic for published messages. This works well out-of-the-box, but you can easily set a different root topic string to take advantage of ZMQ's built-in topic filtering mechanism.

First, set the root topic on the handler:

```
zmq_log_handler = PUBHandler("<tcp://127.0.0.1:12345>")
zmq_log_handler.setRootTopic("custom_topic")
```

Then specify that topic when you start the log watcher:

```
python -m zmq.log -t custom_topic <tcp://127.0.0.1:12345>
```

Complete example

Assuming this project hierarchy:

```
example.py
greetings.py
hello.py
```

If you have this in `example.py`:

```
import logging
from time import sleep
from zmq.log.handlers import PUBHandler

from greetings import hello

zmq_log_handler = PUBHandler("tcp://127.0.0.1:12345")
zmq_log_handler.setFormatter(logging.Formatter(fmt="{name} > {message}", style="{"))
zmq_log_handler.setFormatter(
    logging.Formatter(fmt="{name} #{lineno:>3} > {message}", style="{"), logging.DEBUG
)
zmq_log_handler.setRootTopic("greeter")

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
logger.addHandler(zmq_log_handler)

if __name__ == "__main__":
    sleep(0.1)
    msg_count = 5
    logger.warning("Preparing to greet the world...")
    for i in range(1, msg_count + 1):
        logger.debug("Sending message {} of {}".format(i, msg_count))
        hello.world()
        sleep(1.0)
    logger.info("Done!")
```

And this in `hello.py`:

```
import logging

logger = logging.getLogger(__name__)

def world():
    logger.info('hello world!')
```

You can start a log watcher in one process:

```
python -m zmq.log -t greeter --align tcp://127.0.0.1:12345
```

And then run `example.py` in another process:

```
python example.py
```

You should see the following output from the log watcher:

```
greeter.WARNING | root > Preparing to greet the world...
greeter.DEBUG   | root # 21 > Sending message 1 of 5
greeter.INFO    | greetings.hello > hello world!
greeter.DEBUG   | root # 21 > Sending message 2 of 5
greeter.INFO    | greetings.hello > hello world!
greeter.DEBUG   | root # 21 > Sending message 3 of 5
greeter.INFO    | greetings.hello > hello world!
greeter.DEBUG   | root # 21 > Sending message 4 of 5
greeter.INFO    | greetings.hello > hello world!
greeter.DEBUG   | root # 21 > Sending message 5 of 5
greeter.INFO    | greetings.hello > hello world!
greeter.INFO    | root > Done!
```

PUB/SUB and Topics

The ØMQ PUB/SUB pattern consists of a PUB socket broadcasting messages, and a collection of SUB sockets that receive those messages. Each PUB message is a multipart-message, where the first part is interpreted as a topic. SUB sockets can subscribe to topics by setting their `SUBSCRIBE` sockopt, e.g.:

```
sub = ctx.socket(zmq.SUB)
sub.setsockopt(zmq.SUBSCRIBE, 'topic1')
sub.setsockopt(zmq.SUBSCRIBE, 'topic2')
```

When subscribed, the SUB socket will only receive messages where the first part *starts with* one of the topics set via `SUBSCRIBE`. The default behavior is to exclude all messages, and subscribing to the empty string ‘’ will receive all messages.

PUBHandler

The *PUBHandler* object is created for allowing the python logging to be emitted on a PUB socket. The main difference between a PUBHandler and a regular logging Handler is the inclusion of topics. For the most basic logging, you can simply create a PUBHandler with an interface or a configured PUB socket, and just let it go:

```
pub = context.socket(zmq.PUB)
pub.bind('tcp://*:12345')
handler = PUBHandler(pub)
logger = logging.getLogger()
logger.addHandler(handler)
```

At this point, all messages logged with the default logger will be broadcast on the pub socket.

the PUBHandler does work with topics, and the handler has an attribute `root_topic`:

```
handler.root_topic = "myprogram"
```

Python loggers also have loglevels. The base topic of messages emitted by the PUBHandler will be of the form: <handler.root_topic>.<loglevel>, e.g. `myprogram.INFO` or ‘whatever.ERROR’. This way, subscribers can easily subscribe to subsets of the logging messages. Log messages are always two-part, where the first part is the topic tree, and the second part is the actual log message.

```
logger.info("hello there")
print(sub.recv_multipart())
```

```
[b"myprogram.INFO", b"hello there"]
```

Subtopics

You can also add to the topic tree below the loglevel on an individual message basis. Assuming your logger is connected to a PUBHandler, you can add as many additional topics on the front of the message, which will be added always after the loglevel. A special delimiter defined at `zmq.log.handlers.TOPIC_DELIM` is scanned by the PUBHandler, so if you pass your own subtopics prior to that symbol, they will be stripped from the message and added to the topic tree:

```
>>> log_msg  = "hello there"
>>> subtopic = "sub.topic"
>>> msg = zmq.log.handlers.TOPIC_DELIM.join([subtopic, log_msg])
>>> logger.warn(msg)
>>> print sub.recv_multipart()
['myprogram.WARN.sub.topic', 'hello there']
```

2.3.8 Tunneling PyZMQ Connections with SSH

New in version 2.1.9.

You may want to connect ØMQ sockets across machines, or untrusted networks. One common way to do this is to tunnel the connection via SSH. IPython introduced some tools for tunneling ØMQ connections over ssh in simple cases. These functions have been brought into pyzmq as *zmq.ssh.tunnel* under IPython's BSD license.

PyZMQ will use the shell ssh command via pexpect by default, but it also supports using paramiko for tunnels, so it should work on Windows.

An SSH tunnel has five basic components:

- server : the SSH server through which the tunnel will be created
- remote ip : the IP of the remote machine *as seen from the server* (remote ip may be, but is not not generally the same machine as server).
- remote port : the port on the remote machine that you want to connect to.
- local ip : the interface on your local machine you want to use (default: 127.0.0.1)
- local port : the local port you want to forward to the remote port (default: high random)

So once you have established the tunnel, connections to `localip:localport` will actually be connections to `remoteip:remoteport`.

In most cases, you have a zeromq url for a remote machine, but you need to tunnel the connection through an ssh server. This is

So if you would use this command from the same LAN as the remote machine:

```
sock.connect("tcp://10.0.1.2:5555")
```

to make the same connection from another machine that is outside the network, but you have ssh access to a machine `server` on the same LAN, you would simply do:

```
from zmq import ssh

ssh.tunnel_connection(sock, "tcp://10.0.1.2:5555", "server")
```

Note that "server" can actually be a fully specified "user@server:port" ssh url. Since this really just launches a shell command, all your ssh configuration of usernames, aliases, keys, etc. will be respected. If necessary, *tunnel_connection()* does take arguments for specific passwords, private keys (the ssh -i option), and non-default choice of whether to use paramiko.

If you are on the same network as the machine, but it is only listening on localhost, you can still connect by making the machine itself the server, and using loopback as the remote ip:

```
from zmq import ssh

ssh.tunnel_connection(sock, "tcp://127.0.0.1:5555", "10.0.1.2")
```

The *tunnel_connection()* function is a simple utility that forwards a random localhost port to the real destination, and connects a socket to the new local url, rather than the remote one that wouldn't actually work.

See also:

A short discussion of ssh tunnels: https://www.revsys.com/writings/quicktips/ssh-tunnel.html

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FOUR

LINKS

- ØMQ Home
- The ØMQ Guide
- PyZMQ on GitHub
- Issue Tracker

PYTHON MODULE INDEX

Z

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T