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 (3.3), and Python 2.7, with no transformations or 2to3, as well as PyPy (at least 2.0 beta), 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.

Changes in PyZMQ
CHAPTER ONE

SUPPORTED LIBZMQ

PyZMQ aims to support all stable (2.1.4, 3.2.2, 4.0.1) and active development (4.2.0) 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, and will never be supported. There never was a stable release of either.

Binary distributions (wheels on PyPI or GitHub) of PyZMQ ship with the stable version of libzmq at the time of release, built with default configuration, and include CURVE support provided by tweetnacl. For pyzmq-22.3.0, this is 4.3.4.
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

**Release** 22.3.0  
**Date** Dec 20, 2021

## 2.1.1 zmq

Python bindings for 0MQ.

### Basic Classes

#### Context

```python
class zmq.Context(io_threads: int = 1, **kwargs)
```

Create a zmq Context

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

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

- **destroy**(linger: Optional[float] = None)
  
  Close all sockets associated with this context and then terminate the context.

  **Warning:** destroy involves calling `zmq_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)
  
  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)

get default socket options for new sockets created by this Context

New in version 13.0.

**classmethod** **instance**(io_threads=1) → zmq.sugar.context.T

Returns a global Context instance.

Most single-threaded 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:

```python
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, 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)

set default socket options for new sockets created by this Context

New in version 13.0.

**classmethod** **shadow**(address)

Shadow an existing libzmq context

address is the integer address of the libzmq context or an FFI pointer to it.

```python
classmethod shadow_pyczmq(ctx: Any) → zmq.sugar.context.T
Shadow an existing pyczmq context
ctx is the FFI zctx_t * pointer

socket(socket_type: int, **kwargs)
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.
- `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.

underlying
The address of the underlying libzmq context

Socket

class zmq.Socket(*a, **kw)
The ZMQ socket object

To create a Socket, first create a Context:

c = zmq.Context.instance()
then call `c.socket(socket_type)`:

s = c.socket(zmq.ROUTER)

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.

**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, min_port=49152, max_port=65536, max_tries=100)

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)

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.

**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, upd, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.
disable_monitor()
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, upd, pgm, inproc and ipc. If the address is unicode, it is encoded to utf-8 first.

fileno()
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)
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()
Get the High Water Mark.
On libzmq 3, this gets SNDHWM if available, otherwise RCVHWM

get_monitor_socket(events=None, addr=None)
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 (string [default: None]) – The optional endpoint for the monitoring sockets.

    Returns socket – The socket is already connected and ready to receive messages.

    Return type (PAIR)

get_string(option, encoding='utf-8')
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)*

`s.get(option)`

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, encoding='utf-8')*

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`

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, flags)*

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=None, flags=PollEvent.POLLIN)**

Poll the socket for events. See **Poller** to wait for multiple sockets at once.

**Parameters**

- **timeout (int [default: None])** – 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.

**Returns** int

**recv (flags=0, copy=True, track=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=0, **kwargs)**

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=0, copy=True, track=False)**

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=0)*

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=0, encoding='utf-8')*

Receive a unicode string, as sent by `send_string`.

**Parameters**
- `flags` *(int)* – Any valid flags for `Socket.recv()`.
- `encoding` *(str [default: 'utf-8'])* – 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 `recv()` might fail

**send** *(data, flags=0, copy=True, track=False, routing_id=None, group=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 pending property will be True 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, flags=0, **kwargs)`

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*, optional) – Any valid flags for `Socket.send()`

`send_multipart(msg_parts, flags=0, copy=True, track=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 pending property will be True until the last send is completed.
send_pyobj(obj, flags=0, protocol=3, **kwargs)
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, flags=0, copy=True, encoding='utf-8', **kwargs)
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* [*default: ‘utf-8’]*) – The encoding to be used

set(option, 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)`
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, optval, encoding='utf-8')`
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, optval)`
s.set(option, 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, optval, encoding='utf-8')`
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)**

Shadow an existing libzmq socket

- address is the integer address of the libzmq socket or an FFI pointer to it.

**subscribe(topic)**

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, upd, 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)**

Unsubscribe from a topic

- Only for SUB sockets.
- New in version 15.3.

**Frame**

**class zmq.Frame(data=None, track=False, copy=None, copy_threshold=zmq.COPY_THRESHOLD)**

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:

```python
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 [default: False]`) – 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)

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, Message instances.)** – 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=-1)**

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

**Parameters timeout (float [default: -1, 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.None)**

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

**poll(timeout: Optional[int] = None)**

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 of tuples

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

Returns (rlist, wlist, xlist) – Lists correspond to sockets available for read/write/error events respectively.

Return type  tuple of lists of sockets (length 3)

Exceptions

ZMQError

class zmq.ZMQError(errno: Optional[int] = None, msg: Optional[str] = None)
Wrap an errno style error.

Parameters

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

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.
    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()
    Return the version of ZeroMQ itself as a 3-tuple of ints.
```python
zmq.pyzmq_version_info() → Union[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)
Check for zmq capability by name (e.g. ‘ipc’, ‘curve’)
New in version libzmq-4.1.

zmq.device(device_type, frontend, backend)
Start a zeromq device.
Depreciated since version libzmq-3.2: Use zmq.proxy

Parameters

• device_type (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, backend, capture)
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, backend, capture, control)
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)
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 bytestring
```
```python
zmq.curve_keypair()

generate a Z85 keypair 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, secret) – The public and private keypair as 40 byte z85-encoded bytestrings.

Return type two bytestrings
```

```python
zmq.get_includes()

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

```python
zmq.get_library_dirs()

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

### 2.1.2 devices

#### Functions

```python
zmq.device(device_type, frontend, backend)

Start a zeromq device.

Deprecated since version libzmq-3.2: Use zmq.proxy

Parameters

• device_type ((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.
```

```python
zmq.proxy(frontend, backend, capture)

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.
```

```python
zmq.proxy_steerable(frontend, backend, capture, control)

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**

```python
class zmq.devices.Device(device_type=DeviceType.QUEUE, in_type=None, out_type=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:

```python
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)**

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

**connect_{in_out}(iface)**

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

**setsockopt_{in_out}(opt, value)**

pass through 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** `int`

**context_factory**

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 (class attribute)`

**bind_in(addr)**

Enqueue ZMQ address for binding on in_socket.

See zmq.Socket.bind for details.
**bind_in_to_random_port** *(addr, *args, **kwargs)*
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)*
Enqueue ZMQ address for binding on out_socket.
See zmq.Socket.bind for details.

**bind_out_to_random_port** *(addr, *args, **kwargs)*
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)*
Enqueue ZMQ address for connecting on in_socket.
See zmq.Socket.connect for details.

**connect_out** *(addr)*
Enqueue ZMQ address for connecting on out_socket.
See zmq.Socket.connect for details.

**join** *(timeout=None)*
wait for me to finish, like Thread.join.
Reimplemented appropriately by subclasses.

**setsockopt_in** *(opt, value)*
Enqueue setsockopt(opt, value) for in_socket
See zmq.Socket.setsockopt for details.

**setsockopt_out** *(opt, value)*
Enqueue setsockopt(opt, value) for out_socket
See zmq.Socket.setsockopt for details.

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

### ThreadDevice

**class** cmbizmq.devices.ThreadDevice**(device_type=DeviceType.QUEUE, in_type=None, out_type=None)**
A Device that will be run in a background Thread.
See Device for details.
ProcessDevice

class zmq.devices.ProcessDevice(device_type=DeviceType.QUEUE, in_type=None, out_type=None)

A Device that will be run in a background Process.

See Device for details.

context_factory

    alias of zmq.sugar.context.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.

cconnect_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

```python
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

```python
class zmq.devices.ThreadProxySteerable(in_type, out_type, mon_type=SocketType.PUB, ctrl_type=None)
    ProxySteerable in a Thread. See ProxySteerable for details.
```

ProcessProxySteerable

```python
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

```python
zmq.devices.monitored_queue(in_socket, out_socket, mon_socket, in_prefix=b'in', out_prefix=b'out')

monitored_queue(in_socket, out_socket, mon_socket, in_prefix=b'in', out_prefix=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** *(Socket)* – One of the sockets to the Queue. Its messages will be prefixed with ‘in’.
- **out_socket** *(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** *(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:

```python
from zmq.decorators import context, socket

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

Decorators

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

Decorator for adding a Context to a function.

Usage:

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

New in version 15.3.

Parameters

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

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

Decorator for adding a socket to a function.

Usage:

```python
@socket(zmq.PUSH)
def foo(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: 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: eventloop.ioloop
tornado IOLoop API with zmq compatibility

This module is deprecated in pyzmq 17. To use zmq with tornado, eventloop integration is no longer required and tornado itself should be used.

Classes

ZMQIOLoop

class zmq.eventloop.ioloop.ZMQIOLoop(*args: Any, **kwargs: Any)
   DEPRECATED: No longer needed as of pyzmq-17
   PyZMQ tornado integration now works with the default tornado.ioloop.IOLoop.

Function

zmq.eventloop.ioloop.install()
   DEPRECATED
   pyzmq 17 no longer needs any special integration for tornado.

2.1.6 eventloop.future

Module: 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 zmq.eventloop.future.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

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

c = Context()

@gen.coroutine
def recv_and_process():
    sock = c.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.

```python
class zmq.eventloop.future.Context(*args, **kwargs)
```

#### Socket

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

See also:

zmq.Socket for the inherited API.

```python
class zmq.eventloop.future.Socket(context=None, socket_type=-1, io_loop=None, **kwargs)

    recv(flags=0, copy=True, track=False)
        Receive a single zmq frame.
        Returns a Future, whose result will be the received frame.
        Recommend using recv_multipart instead.

    recv_multipart(flags=0, copy=True, track=False)
        Receive a complete multipart zmq message.
        Returns a Future whose result will be a multipart message.

    send(msg, flags=0, copy=True, track=False, **kwargs)
        Send a single zmq frame.
        Returns a Future that resolves when sending is complete.
        Recommend using send_multipart instead.
```
send_multipart(msg, flags=0, copy=True, track=False, **kwargs)

Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

poll(timeout=None, flags=PollEvent.POLLIN)

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)

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 zmq.asyncio.Socket.recv_multipart(), zmq.asyncio.Socket.poll(), and zmq.asyncio.Poller.poll() return Future s.

```python
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**

**ZMQEventLoop**

class zmq.asyncio.ZMQEventLoop(selector=None)

DEPRECATED: AsyncIO eventloop using zmq_poll.

pyzmq sockets should work with any asyncio event loop as of pyzmq 17.

**Context**

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

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

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, **kwargs)

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

recv(flags=0, copy=True, track=False)

Receive a single zmq frame.

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

Recommend using recv_multipart instead.

recv_multipart(flags=0, copy=True, track=False)

Receive a complete multipart zmq message.

Returns a Future whose result will be a multipart message.

send(msg, flags=0, copy=True, track=False, **kwargs)

Send a single zmq frame.

Returns a Future that resolves when sending is complete.

Recommend using send_multipart instead.

send_multipart(msg, flags=0, copy=True, track=False, **kwargs)

Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

poll(timeout=None, flags=PollEvent.POLLIN)

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)
            Return a Future for a poll event

2.1.8 eventloop.zmqstream

Module: eventloop.zmqstream

A utility class to send to and recv from a non-blocking socket, using tornado.

See also:

• zmq.asyncio
• zmq.eventloop.future

ZMQStream

class zmq.eventloop.zmqstream.ZMQStream(socket, io_loop=None)
    A utility class to register callbacks when a zmq socket sends and receives
    For use with zmq.eventloop.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 re-
    ceive
• on_send(callback): register a callback to be run every time you call send
• send(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
close(linger=None)
    Close this stream.

closed()

flush(flag=PollEvent.None, limit=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 int
    Return type count of events handled (both send and recv)

io_loop = None

on_err(callback)
    DEPRECATED, does nothing

on_recv(callback, copy=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, copy=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)
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)
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 = None

receiving()
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, flags=0, callback=None, **kwargs)
Send jsonserialized version of an object. See zmq.socket.send_json for details.

send_multipart(msg, flags=0, copy=True, track=False, callback=None, **kwargs)
Send a multipart message, optionally also register a new callback for sends. See zmq.socket.send_multipart for details.

send_pyobj(obj, flags=0, protocol=-1, callback=None, **kwargs)
Send a Python object as a message using pickle to serialize.

See zmq.socket.send_json for details.

send_string(u, flags=0, encoding='utf-8', callback=None, **kwargs)
Send a unicode message with an encoding. See zmq.socket.send_unicode for details.
send_unicode(u, flags=0, encoding='utf-8', callback=None, **kwargs)
Send a unicode message with an encoding. See zmq.socket.send_unicode for details.

sending()
Returns True if we are currently sending to the stream.

set_close_callback(callback)
Call the given callback when the stream is closed.

socket = None

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: auth

Utilities for ZAP authentication.

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

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


Authenticator
class zmq.auth.Authenticator(context=None, encoding='utf-8', log=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:

```python
auth = zmq.Authenticator()
auth.allow("127.0.0.1")
auth.start()
while True:
    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)`
Allow (whitelist) IP address(es).

Connections from addresses not in the whitelist 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.

whitelist is mutually exclusive with blacklist.

`configure_curve(domain='*', location=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='*', credentials_provider=None)`
Configure 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.:

```python
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 “*”.

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

`configure_gssapi(domain='*', location=None)`
Configure GSSAPI authentication

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

`configure_plain(domain='*', passwords=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.

`curve_user_id(client_public_key)`
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)*
Deny (blacklist) IP address(es).
Addresses not in the blacklist will be allowed to continue with authentication.
Blacklist is mutually exclusive with whitelist.

**handle_zap_message** *(msg)*
Perform ZAP authentication

**start** ()
Create and bind the ZAP socket

**stop** ()
Close the ZAP socket

### Functions

- **zmq.auth.create_certificates** *(key_dir, name, metadata=None)*
  Create zmq certificates.
  Returns the file paths to the public and secret certificate files.

- **zmq.auth.load_certificate** *(filename)*
  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=‘.’)*
  Load public keys from all certificates in a directory

### 2.1.10 auth.asyncio

**Module:** auth.asyncio

ZAP Authenticator integrated with the asyncio IO loop.
New in version 15.2.
Classes

AsyncioAuthenticator

class zmq.auth.asyncio.AsyncioAuthenticator(context=None, loop=None)

ZAP authentication for use in the asyncio IO loop

allow(*addresses)

Allow (whitelist) IP address(es).

Connections from addresses not in the whitelist 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.

whitelist is mutually exclusive with blacklist.

configure_curve(domain="*", location=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="*", credentials_provider=None)

Configure 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.:

```python
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 “*”.

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

configure_gssapi(domain="*", location=None)

Configure GSSAPI authentication

Currently this is a no-op because there is nothing to configure with GSSAPI.
configure_plain(domain="*", passwords=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.

curve_user_id(client_public_key)
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)
Deny (blacklist) IP address(es).

Addresses not in the blacklist will be allowed to continue with authentication.

Blacklist is mutually exclusive with whitelist.

handle_zap_message(msg)
Perform ZAP authentication

start()
Start ZAP authentication

stop()
Stop ZAP authentication

2.1.11 auth.thread

Module: auth.thread

ZAP Authenticator in a Python Thread.

Classes

ThreadAuthenticator

class zmq.auth.thread.ThreadAuthenticator(context=None, encoding='utf-8', log=None)
Run ZAP authentication in a background thread

allow(*addresses)
Allow (whitelist) IP address(es).

Connections from addresses not in the whitelist 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.

whitelist is mutually exclusive with blacklist.
auth = None

configure_curve(domain='*', location=''
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='*', credentials_provider=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 "*".
To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

configure_plain(domain='*', passwords=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 = None
deny(*addresses)
Deny (blacklist) IP address(es).
Addresses not in the blacklist will be allowed to continue with authentication.
Blacklist is mutually exclusive with whitelist.

encoding = None

is_alive()
Is the ZAP thread currently running?

log = None

pipe = None

pipe_endpoint = ''
**start()**
Start the authentication thread

**stop()**
Stop the authentication thread

```python
thread = None
```

## 2.1.12 auth.ioloop

**Module:** auth.ioloop

ZAP Authenticator integrated with the tornado IOLoop.


**IOLoopAuthenticator**

```python
class zmq.auth.ioloop.IOLoopAuthenticator(context=None, encoding='utf-8', log=None, io_loop=None):
```

ZAP authentication for use in the tornado IOLoop

**allow(addresses)**
Allow (whitelist) IP address(es).

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

whitelist is mutually exclusive with blacklist.

**configure_curve(domain='*', location=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='*', credentials_provider=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.:

```python
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)
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 "*".

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

configure_gssapi(domain='*', location=None)
    Configure GSSAPI authentication
    Currently this is a no-op because there is nothing to configure with GSSAPI.

configure_plain(domain='*', passwords=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.

curve_user_id(client_public_key)
    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)
    Deny (blacklist) IP address(es).
    Addresses not in the blacklist will be allowed to continue with authentication.
    Blacklist is mutually exclusive with whitelist.

handle_zap_message(msg)
    Perform ZAP authentication

start()
    Start ZAP authentication

stop()
    Stop ZAP authentication
2.1.13 log.handlers

Module: 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:

```python
>>> 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)
```

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

Code adapted from StarCluster:

http://github.com/jtriley/StarCluster/blob/master/starcluster/logger.py

Classes

PUBHandler

```
class zmq.log.handlers.PUBHandler(interface_or_socket, context=None, root_topic='')
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.

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

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)
    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.

```python
set_name(name)
socket = None
```

**TopicLogger**

```python
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:

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

Would become:

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

```python
addFilter(filter)

Add the specified filter to this handler.
```

```python
addHandler(hdlr)

Add the specified handler to this logger.
```

```python
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.
```

```python
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.

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

```python
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.

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

```python
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.

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

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

Convenience method for logging an ERROR with exception information.
```
**fatal** *(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)

**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)*
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 (str)
• level (str|int)
• fn (str)
• lno (int)
• msg (str)
• args (float)
• func (str)
• extra (dict)
• sinfo (str)

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

manager = <logging.Manager object>

removeFilter(filter)

• filter (Filter)

Remove the specified filter from this handler.

removeHandler(hdlr)

• hdlr (Handler)

Remove the specified handler from this logger.

root = <RootLogger root (INFO)>

setLevel(level)

• level (int|str)

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: 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.

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

Return type (str, object)

zmq.ssh.tunnel.openssh_tunnel(lport, rport, server, remoteip=’127.0.0.1’, keyfile=None, password=None, timeout=60)

Create an ssh tunnel using command-line ssh that connects port lport on this machine to localhost:rport on server. The tunnel will automatically close when not in use, remaining open for a minimum of timeout seconds for an initial connection.

This creates a tunnel redirecting localhost:lport to remoteip:rport, as seen from server.

keyfile and password may be specified, but ssh config is checked for defaults.

Parameters

• lport (int) – local port for connecting to the tunnel from this machine.
• rport (int) – port on the remote machine to connect to.
server (str) – The ssh server to connect to. The full ssh server string will be parsed. user@server:port

remoteip (str [Default: 127.0.0.1]) – The remote ip, specifying the destination of the tunnel. Default is localhost, which means that the tunnel would redirect localhost:lport on this machine to localhost:rport on the server.

keyfile (str; path to public key file) – This specifies a key to be used in ssh login, default None. Regular default ssh keys will be used without specifying this argument.

password (str;) – Your ssh password to the ssh server. Note that if this is left None, you will be prompted for it if passwordless key based login is unavailable.

timeout (int [default: 60]) – The time (in seconds) after which no activity will result in the tunnel closing. This prevents orphaned tunnels from running forever.

zmq.ssh.tunnel.paramiko_tunnel (lport, rport, server, remoteip='127.0.0.1', keyfile=None, password=None, timeout=60)

launch a tunner with paramiko in a subprocess. This should only be used when shell ssh is unavailable (e.g. Windows).

This creates a tunnel redirecting localhost:lport to remoteip:rport, as seen from server.

If you are familiar with ssh tunnels, this creates the tunnel:

```
ssh server -L localhost:lport:remoteip:rport
```

keyfile and password may be specified, but ssh config is checked for defaults.

Parameters

- lport (int) – local port for connecting to the tunnel from this machine.
- rport (int) – port on the remote machine to connect to.
- server (str) – The ssh server to connect to. The full ssh server string will be parsed. user@server:port
- remoteip (str [Default: 127.0.0.1]) – The remote ip, specifying the destination of the tunnel. Default is localhost, which means that the tunnel would redirect localhost:lport on this machine to localhost:rport on the server.
- keyfile (str; path to public key file) – This specifies a key to be used in ssh login, default None. Regular default ssh keys will be used without specifying this argument.
- password (str;) – Your ssh password to the ssh server. Note that if this is left None, you will be prompted for it if passwordless key based login is unavailable.
- timeout (int [default: 60]) – The time (in seconds) after which no activity will result in the tunnel closing. This prevents orphaned tunnels from running forever.

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:** utils.jsonapi

JSON serialize to/from utf8 bytes

**versionchanged:: 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: Union[bytes, str], **kwargs) → Union[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:** utils.monitor

Module holding utility and convenience functions for zmq event monitoring.

**Functions**

`zmq.utils.monitor.parse_monitor_message(msg)`

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, flags=0)`

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 : string, the affected endpoint

Parameters

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

  zmq.utils.win32.allow_interrupt
  alias of zmq.utils.win32._allow_interrupt

### 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 22.3.0

Fixes:

- Fix `str1cpy` 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)

2.2.2 22.2.1

Fix bundling of wepoll on Windows.

2.2.3 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.
2.2.4 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 `concr140.dll`.
• adopt pre-commit for formatting, linting

2.2.5 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.

2.2.6 22.0.2

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

2.2.7 22.0.1

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

2.2.8 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 `libodium` 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.
2.2.9 21.0.2

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

2.2.10 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.

2.2.11 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 instalation 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

2.2.12 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 SPKG_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.

2.2.13 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

2.2.14 19.0.1

• Fix TypeError during garbage collection
• Fix compilation with some C++ compilers
• Fixes in tests and examples
2.2.15 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 (see Asynchronous Logging via PyZMQ)
- Log: add python -m zmq.log entrypoint
- Sources generated with Cython 0.29.15

2.2.16 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

2.2.17 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

2.2.18 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.

2.2.19 18.0.1

Fixes installation from source on non-unicode locales with Python 3. There are no code changes in this release.
2.2.20 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

2.2.21 17.1.3

- Fix compatibility with tornado 6 (removal of stack_context)

2.2.22 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

2.2.23 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

2.2.24 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:
PyZMQ Documentation, Release 22.3.0

- `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*

### 2.2.25 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.

### 2.2.26 16.0.3

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

### 2.2.27 16.0.2

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

### 2.2.28 16.0.1

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

### 2.2.29 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`
2.2.30 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.

2.2.31 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

2.2.32 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
2.2.33 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

2.2.34 15.0

PyZMQ 15 adds Future-returning sockets and pollers for both asyncio and tornado.
• add asyncio support via zmq.asyncio
• add tornado 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/.

2.2.35 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.

2.2.36 14.6.0

Changes:
• improvements in zmq.Socket.bind_to_random_port():
  – use system to allocate ports by default
  – catch EACCES on Windows
• includelibsodium 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
2.2.37 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

2.2.38 14.4.1

Bugfixes for 14.4

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

2.2.39 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 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

2.2.40 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

2.2.41 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.

2.2.42 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.

2.2.43 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.

#### 2.2.44 14.0.1

Bugfix release

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

#### 2.2.45 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*

#### 2.2.46 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.
2.2.47 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.

2.2.48 13.0.1

defunct bugfix. We do not speak of this…

2.2.49 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`

• 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.

### 2.2.50 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: `IOLoop.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:

```python
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:

*PyZMQ and gevent*

**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)).

**2.2.51  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 more boolean attribute has been added to the **Frame** (née Message) class, so that frames can be identified as terminal without extra queries of recvmore.

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.

### 2.2.52 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 **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 POLLLIN/OUT/ERR. So applications that used the low-level IOLoop.add_handler() code with POLLLIN/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.
2.2.53 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:
    ```python
    send_multipart([b'msg', b'parts'], prefix=[b'label', b'prefix'])
    ```
  - 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 `ctx.destroy()`, so that `ctx.term()` remains threadsafe and 1:1 with libzmq.
- `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

2.2.54 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 `KeyboardInterrupts` from being catchable.
- Added attribute-access for set/getsockopt. Setting/Getting attributes of `Sockets` with the names of socket options is mapped to calls of set/getsockopt.

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

- Terminating a `Context` closes the sockets it created, matching the behavior in czmq.
- `ThreadDevices` use `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.
2.2.55 2.1.7.1

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

2.2.56 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.

2.2.57 2.1.4

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

2.3 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-17 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=4.3.4
PREFIX=/usr/local
wget https://github.com/zeromq/libzmq/releases/download/v$ZMQ_VERSION/zeromq-$ZMQ_VERSION.tar.gz
tar -xzf libzmq.tar.gz
cd zeromq-$ZMQ_VERSION
./configure --prefix=$PREFIX --enable-drafts
make -j && make install
```

And to install pyzmq with draft support:

```
export ZMQ_PREFIX=$PREFIX
export ZMQ_DRAFT_API=1
pip install -v --no-binary pyzmq --pre pyzmq
```

By specifying `--no-binary` pyzmq, pip knows to not install 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_ENABLE_DRAFTS` option is required.

There are examples of the CLIENT-SERVER and RADIO-DISH patterns in the examples/draft directory of the pyzmq repository.
2.4 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.

2.4.1 The Core as Bindings

PyZMQ is currently broken up into four subpackages. First, is the Core. `zmq.core` contains the actual bindings for ZeroMQ, and no extended functionality beyond the very basic. The core modules are split, such that each basic ZeroMQ object (or function, if no object is associated) is a separate module, e.g. `zmq.core.context` contains the `Context` object, `zmq.core.poll` contains a `Poller` object, as well as the `select()` function, etc. ZMQ constants are, for convenience, all kept together in `zmq.core.constants`.

There are two reasons for breaking the core into submodules: *recompilation* and *derivative projects*. The monolithic PyZMQ became quite tedious to have to recompile everything for a small change to a single object. With separate files, that’s no longer necessary. The second reason has to do with Cython. PyZMQ is written in Cython, a tool for efficiently writing C-extensions for Python. By separating out our objects into individual `.pyx` files, each with their declarations in a `.pxd` header, other projects can write extensions in Cython and call directly to ZeroMQ at the C-level without the penalty of going through our Python objects.

2.4.2 Thread Safety

In ØMQ, Contexts are thread-safe 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

2.4.3 Socket Options as Attributes

New in version 2.1.9.

In ØMQ, 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:

```python
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.

```python
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.

2.4.4 Core Extensions

We have extended the core functionality in two ways that appear inside the core bindings, 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.
- Our Unicode discussion for more information on the trials and tribulations of working with Unicode in a C extension while supporting Python 2 and 3.

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 thread-safe 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.

2.4.5 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.

2.5 Serializing messages with PyZMQ

When sending messages over a network, you often need to marshall your data into bytes.

2.5.1 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 do want to emphasize performance should use their own serialized send/recv methods.

2.5.2 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 ([msgpack], [protobuf]), or adding compression via [zlib] in the standard library, or the super fast [blosc] 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:

```python
import zlib, cPickle as pickle

def send_zipped_pickle(socket, obj, flags=0, protocol=-1):
    """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)
```

(continues on next page)
def recv_zipped_pickle(socket, flags=0, protocol=-1):
    """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/recvvs 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)
    buf = memoryview(msg)
    A = numpy.frombuffer(buf, dtype=md['dtype'])
    return A.reshape(md['shape'])

2.6 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.

2.6.1 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

### 2.6.2 MonitoredQueue

One of ØMQ's built-in 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.

### 2.7 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.
2.7.1 AsynclO

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.

```python
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()
```

**Note:** In PyZMQ < 17, an additional step is needed to register the zmq poller prior to starting any async code:

```python
import zmq.asyncio
zmq.asyncio.install()

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

This step is no longer needed in pyzmq 17.

2.7.2 Tornado IOLoop

Facebook’s Tornado includes an eventloop for handing poll events on filedescriptors and native sockets. We have included a small part of Tornado (specifically its `ioloop`), and adapted its `IOStream` class into `ZMQStream` for handling poll 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()`. Callbacks can also be registered for send events with `on_send()`.

**Futures and coroutines**

PyZMQ 15 adds `zmq.eventloop.future`, containing a Socket subclass that returns `Future` objects for use in tornado coroutines. To use this API, import `zmq.eventloop.future.Context`. Sockets created by this Context will return Futures from any would-be blocking method.

```python
from tornado import gen, ioloop
import zmq
from zmq.eventloop.future import Context

ctx = Context.instance()

@gen.coroutine
(continues on next page)```
**def recv():**

```python
s = ctx.socket(zmq.SUB)
s.connect('tcp://127.0.0.1:5555')
s.subscribe(b'')

while True:
    msg = yield s.recv_multipart()
    print('received', msg)
s.close()
```

**ZMQStream**

ZMQStream objects let you register callbacks to handle messages as they arrive, for use with the tornado eventloop.

**send()**

ZMQStream objects do have `send()` and `send_multipart()` methods, which behaves the same way as `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:

```python
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 `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.

```python
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 `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.

install()

![Note: If you are using pyzmq < 17, there is an additional step to tell tornado to use the zmq poller instead of its default. `ioloop.install()` is no longer needed for pyzmq 17.]

With PyZMQ’s ioloop, you can use zmq sockets in any tornado application. You can tell tornado to use zmq’s poller by calling the `ioloop.install()` function:

```python
from zmq.eventloop import ioloop
ioloop.install()
```

You can also do the same thing by requesting the global instance from pyzmq:

```python
from zmq.eventloop.ioloop import IOLoop
loop = IOLoop.current()
```

This configures tornado’s `tornado.ioloop.IOLoop` to use zmq’s poller, and registers the current instance.

Either `install()` or retrieving the zmq instance must be done before the global * instance is registered, else there will be a conflict.

It is possible to use PyZMQ sockets with tornado without registering as the global instance, but it is less convenient. First, you must instruct the tornado IOLoop to use the zmq poller:

2.7. Eventloops and PyZMQ
from zmq.eventloop.ioloop import ZMQIOLoop

loop = ZMQIOLoop()

Then, when you instantiate tornado and ZMQStream objects, you must pass the `io_loop` argument to ensure that they use this loop, instead of the global instance.

This is especially useful for writing tests, such as this:

```python
from tornado.testing import AsyncTestCase
from zmq.eventloop.ioloop import ZMQIOLoop
from zmq.eventloop.zmqstream import ZMQStream

class TestZMQBridge(AsyncTestCase):
    # Use a ZMQ-compatible I/O loop so that we can use `ZMQStream`.
    def get_new_ioloop(self):
        return ZMQIOLoop()
```

You can also manually install this IOLoop as the global tornado instance, with:

```python
from zmq.eventloop.ioloop import ZMQIOLoop
loop = ZMQIOLoop()
loop.install()
```

### 2.7.3 PyZMQ and gevent

PyZMQ 2.2.0.1 ships with a gevent compatible API as `zmq.green`. To use it, simply:

```python
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* is simply `gevent_zeromq`, merged into the pyzmq project.
2.8 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.

2.8.1 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:

```python
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:

```python
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:

```python
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  hello.py

If you have this in example.py:

```python
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:

```python
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:
2.8.2 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` socketopt, e.g.:

```python
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.

2.8.3 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:

```python
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`:

```python
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.
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:

```python
>>> 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']
```

2.9 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` 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 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:

```python
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: http://www.revsys.com/writings/quicktips/ssh-tunnel.html
3.1 PyZMQ, Python2.5, and Python3

PyZMQ is a fairly light, low-level library, so supporting as many versions as is reasonable is our goal. Currently, we support at least Python 2.5-3.1. Making the changes to the codebase required a few tricks, which are documented here for future reference, either by us or by other developers looking to support several versions of Python.

Note: It is far simpler to support 2.6-3.x than to include 2.5. Many of the significant syntax changes have been backported to 2.6, so just writing new-style code would work in many cases. I will try to note these points as they come up.

3.1.1 pyversion_compat.h

Many functions we use, primarily involved in converting between C-buffers and Python objects, are not available on all supported versions of Python. In order to resolve missing symbols, we added a header utils/pyversion_compat.h that defines missing symbols with macros. Some of these macros alias new names to old functions (e.g. PyBytes_AsString), so that we can call new-style functions on older versions, and some simply define the function as an empty exception raiser. The important thing is that the symbols are defined to prevent compiler warnings and linking errors. Everywhere we use C-API functions that may not be available in a supported version, at the top of the file is the code:

```c
cdef extern from "pyversion_compat.h":
    pass
```

This ensures that the symbols are defined in the Cython generated C-code. Higher level switching logic exists in the code itself, to prevent actually calling unavailable functions, but the symbols must still be defined.

3.1.2 Bytes and Strings

Note: If you are using Python >= 2.6, to prepare your PyZMQ code for Python3 you should use the b'message' syntax to ensure all your string literal messages will still be bytes after you make the upgrade.

The most cumbersome part of PyZMQ compatibility from a user’s perspective is the fact that, since ØMQ uses C-strings, and would like to do so without copying, we must use the Py3k bytes object, which is backported to 2.6. In order to do this in a Python-version independent way, we added a small utility that unambiguously defines the string types: bytes, unicode, basestring. This is important, because str means different things on 2.x and 3.x, and
bytes is undefined on 2.5, and both unicode and basestring are undefined on 3.x. All typechecking in PyZMQ is done against these types:

<table>
<thead>
<tr>
<th>Explicit Type</th>
<th>2.x</th>
<th>3.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>str</td>
<td>bytes</td>
</tr>
<tr>
<td>unicode</td>
<td>unicode</td>
<td>str</td>
</tr>
<tr>
<td>basestring</td>
<td>basestring</td>
<td>(str, bytes)</td>
</tr>
</tbody>
</table>

**Note:** 2.5 specific

Where we really noticed the issue of bytes vs strings coming up for users was in updating the tests to run on every version. Since the b'bytes literal' syntax was not backported to 2.5, we must call "message".encode() for every string in the test suite.

**See also:**

*Unicode discussion* for more information on strings/bytes.

**PyBytes_**

The standard C-API function for turning a C-string into a Python string was a set of functions with the prefix PyString_. However, with the Unicode changes made in Python3, this was broken into PyBytes_ for bytes objects and PyUnicode_ for unicode objects. We changed all our PyString_ code to PyBytes_, which was backported to 2.6.

**Note:** 2.5 Specific:

Since Python 2.5 doesn't support the PyBytes_* functions, we had to alias them to the PyString_* methods in utils/pyversion_compat.h.

```c
#define PyBytes_FromStringAndSize PyString_FromStringAndSize
#define PyBytes_FromString PyString_FromString
#define PyBytes_AsString PyString_AsString
#define PyBytes_Size PyString_Size
```

### 3.1.3 Buffers

The layer that is most complicated for developers, but shouldn’t trouble users, is the Python C-Buffer APIs. These are the methods for converting between Python objects and C buffers. The reason it is complicated is that it keeps changing.

There are two buffer interfaces for converting an object to a C-buffer, known as new-style and old-style. Old-style buffers were introduced long ago, but the new-style is only backported to 2.6. The old-style buffer interface is not available in 3.x. There is also an old- and new-style interface for creating Python objects that view C-memory. The old-style object is called a buffer, and the new-style object is memoryview. Unlike the new-style buffer interface for objects, memoryview has only been backported to 2.7. This means that the available buffer-related functions are not the same in any two versions of Python 2.5, 2.6, 2.7, or 3.1.

We have a utils/buffers.pxd file that defines our asbuffer() and frombuffer() functions. utils/buffers.pxd was adapted from mpi4py's asbuffer.pxi. The frombuffer() functionality was added. These functions internally switch based on Python version to call the appropriate C-API functions.
See also:

Python Buffer API

3.1.4 __str__

As discussed, str is not a platform independent type. The two places where we are required to return native str objects are error.strerror(), and Message.__str__(). In both of these cases, the natural return is actually a bytes object. In the methods, the native str type is checked, and if the native str is actually unicode, then we decode the bytes into unicode:

```python
# ...
b = natural_result()
if str is unicode:
    return b.decode()
else:
    return b
```

3.1.5 Exceptions

Note: This section is only relevant for supporting Python 2.5 and 3.x, not for 2.6-3.x.

The syntax for handling exceptions has changed in Python 3. The old syntax:

```python
try:
    s.send(msg)
except zmq.ZMQError, e:
    handle(e)
```

is no longer valid in Python 3. Instead, the new syntax for this is:

```python
try:
    s.send(msg)
except zmq.ZMQError as e:
    handle(e)
```

This new syntax is backported to Python 2.6, but is invalid on 2.5. For 2.6-3.x compatible code, we could just use the new syntax. However, the only method we found to catch an exception for handling on both 2.5 and 3.1 is to get the exception object inside the exception block:

```python
try:
    s.send(msg)
except zmq.ZMQError:
    e = sys.exc_info()[1]
    handle(e)
```

This is certainly not as elegant as either the old or new syntax, but it’s the only way we have found to work everywhere.

See also:

PEP-3110
3.2 PyZMQ and Unicode

PyZMQ is built with an eye towards an easy transition to Python 3, and part of that is dealing with unicode strings. This is an overview of some of what we found, and what it means for PyZMQ.

3.2.1 First, Unicode in Python 2 and 3

In Python < 3, a `str` object is really a C string with some sugar - a specific series of bytes with some fun methods like `endswith()` and `split()`. In 2.0, the `unicode` object was added, which handles different methods of encoding. In Python 3, however, the meaning of `str` changes. A `str` in Python 3 is a full unicode object, with encoding and everything. If you want a C string with some sugar, there is a new object called `bytes`, that behaves much like the 2.x `str`. The idea is that for a user, a string is a series of characters, not a series of bytes. For simple ascii, the two are interchangeable, but if you consider accents and non-Latin characters, then the character meaning of byte sequences can be ambiguous, since it depends on the encoding scheme. They decided to avoid the ambiguity by forcing users who want the actual bytes to specify the encoding every time they want to convert a string to bytes. That way, users are aware of the difference between a series of bytes and a collection of characters, and don’t confuse the two, as happens in Python 2.x.

The problems (on both sides) come from the fact that regardless of the language design, users are mostly going to use `str` objects to represent collections of characters, and the behavior of that object is dramatically different in certain aspects between the 2.x `bytes` approach and the 3.x `unicode` approach. The `unicode` approach has the advantage of removing byte ambiguity - it’s a list of characters, not bytes. However, if you really do want the bytes, it’s very inefficient to get them. The `bytes` approach has the advantage of efficiency. A `bytes` object really is just a char* pointer with some methods to be used on it, so when interacting with, so interacting with C code, etc is highly efficient and straightforward. However, understanding a bytes object as a string with extended characters introduces ambiguity and possibly confusion.

To avoid ambiguity, hereafter we will refer to encoded C arrays as ‘bytes’ and abstract unicode objects as ‘strings’.

Unicode Buffers

Since unicode objects have a wide range of representations, they are not stored as the bytes according to their encoding, but rather in a format called UCS (an older fixed-width Unicode format). On some platforms (macOS, Windows), the storage is UCS-2, which is 2 bytes per character. On most *ix systems, it is UCS-4, or 4 bytes per character. The contents of the buffer of a `unicode` object are not encoding dependent (always UCS-2 or UCS-4), but they are platform dependent. As a result of this, and the further insistence on not interpreting `unicode` objects as bytes without specifying encoding, `str` objects in Python 3 don’t even provide the buffer interface. You simply cannot get the raw bytes of a `unicode` object without specifying the encoding for the bytes. In Python 2.x, you can get to the raw buffer, but the platform dependence and the fact that the encoding of the buffer is not the encoding of the object makes it very confusing, so this is probably a good move.

The efficiency problem here comes from the fact that simple ascii strings are 4x as big in memory as they need to be (on most Linux, 2x on other platforms). Also, to translate to/from C code that works with char*, you always have to copy data and encode/decode the bytes. This really is horribly inefficient from a memory standpoint. Essentially, Where memory efficiency matters to you, you should never ever use strings; use bytes. The problem is that users will almost always use `str`, and in 2.x they are efficient, but in 3.x they are not. We want to make sure that we don’t help the user make this mistake, so we ensure that zmq methods don’t try to hide what strings really are.
3.2.2 What This Means for PyZMQ

PyZMQ is a wrapper for a C library, so it really should use bytes, since a string is not a simple wrapper for char * like it used to be, but an abstract sequence of characters. The representations of bytes in Python are either the bytes object itself, or any object that provides the buffer interface (aka memoryview). In Python 2.x, unicode objects do provide the buffer interface, but as they do not in Python 3, where pyzmq requires bytes, we specifically reject unicode objects.

The relevant methods here are socket.send/recv, socket.get/setsockopt, socket.bind/connect. The important consideration for send/recv and set/getsockopt is that when you put in something, you really should get the same object back with its partner method. We can easily coerce unicode objects to bytes with send/setsockopt, but the problem is that the pair method of recv/getsockopt will always be bytes, and there should be symmetry. We certainly shouldn’t try to always decode on the retrieval side, because if users just want bytes, then we are potentially using up enormous amounts of excess memory unnecessarily, due to copying and larger memory footprint of unicode strings.

Still, we recognize the fact that users will quite frequently have unicode strings that they want to send, so we have added socket.<method>_string() wrappers. These methods simply wrap their bytes counterpart by encoding to/decoding from bytes around them, and they all take an encoding keyword argument that defaults to utf-8. Since encoding and decoding are necessary to translate between unicode and bytes, it is impossible to perform non-copying actions with these wrappers.

socket.bind/connect methods are different from these, in that they are strictly setters and there is not corresponding getter method. As a result, we feel that we can safely coerce unicode objects to bytes (always to utf-8) in these methods.

**Note:** For cross-language symmetry (including Python 3), the _unicode methods are now _string. Many languages have a notion of native strings, and the use of _unicode was wedded too closely to the name of such objects in Python 2. For the time being, anywhere you see _string, _unicode also works, and is the only option in pyzmq 2.1.11.

### The Methods

Overview of the relevant methods:

socket.bind(self, addr)
- addr is bytes or unicode. If unicode, encoded to utf-8 bytes

socket.connect(self, addr)
- addr is bytes or unicode. If unicode, encoded to utf-8 bytes

socket.send(self, object obj, flags=0, copy=True)
- obj is bytes or provides buffer interface.
  - if obj is unicode, raise TypeError

socket.recv(self, flags=0, copy=True)
- returns bytes if copy=True
- returns zmq.Message if copy=False:
  - message.buffer is a buffer view of the bytes
  - str(message) provides the bytes
  - unicode(message) decodes message.buffer with utf-8

socket.send_string(self, unicode s, flags=0, encoding='utf-8')
- takes a unicode string s, and sends the bytes after encoding without an extra copy, via:
  - socket.send(s.encode(encoding), flags, copy=False)
socket.recv_string(self, flags=0, encoding='utf-8')
    always returns unicode string
    there will be a UnicodeError if it cannot decode the buffer
    performs non-copying recv, and decodes the buffer with encoding
socket.setsockopt(self, opt, optval)
    only accepts bytes for optval (or int, depending on opt)
    TypeError if unicode or anything else
socket.getsockopt(self, opt)
    returns bytes (or int), never unicode
socket.setsockopt_string(self, opt, unicode optval, encoding='utf-8')
    accepts unicode string for optval
    encodes optval with encoding before passing the bytes to setsockopt
socket.getsockopt_string(self, opt, encoding='utf-8')
    always returns unicode string, after decoding with encoding
    note that zmq.IDENTITY is the onlysockopt with a string value that can be queried with getsockopt
CHAPTER
FOUR

INDICES AND TABLES

- genindex
- modindex
- search
CHAPTER
FIVE

LINKS

• ØMQ Home
• The ØMQ Guide
• PyZMQ Installation notes on the ZeroMQ website
• PyZMQ on GitHub
• Issue Tracker
[zlib] Python stdlib module for zip compression: zlib
[blosc] Blosc: A blocking, shuffling and loss-less (and crazy-fast) compression library http://www.blosc.org
Z
zmq, 5
zmq.asyncio, 31
zmq.auth, 36
zmq.auth.asyncio, 38
zmq.auth.thread, 40
zmq.decorators, 28
zmq.devices, 23
zmq.eventloop.future, 29
zmq.eventloop.ioloop, 29
zmq.eventloop.zmqstream, 33
zmq.green, 29
zmq.log.handlers, 44
zmq.ssh.tunnel, 48
zmq.utils.jsonapi, 50
zmq.utils.monitor, 50
zmq.utils.win32, 51
zmq.utils.z85, 51
INDEX

A
acquire() (zmq.log.handlers.PUBHandler method), 44
addFilter() (zmq.log.handlers.PUBHandler method), 44
addFilter() (zmq.log.handlers.TopicLogger method), 46
addHandler() (zmq.log.handlers.TopicLogger method), 46
Again (class in zmq), 20
allow() (zmq.auth.asyncio.AsyncioAuthenticator method), 39
allow() (zmq.auth.Authenticator method), 37
allow() (zmq.auth.ioloop.IOLoopAuthenticator method), 42
allow() (zmq.auth.thread.ThreadAuthenticator method), 40
allow_interrupt (in module zmq.utils.win32), 51
Authenticator (class in zmq.auth.asyncio), 39
auth (zmq.auth.thread.ThreadAuthenticator attribute), 40
Authenticator (class in zmq.auth), 36

B
bind() (zmq.Socket method), 8
bind_ctrl() (zmq.devices.ProxySteerable method), 26
bind_in() (zmq.devices.Device method), 23
bind_in_to_random_port() (zmq.devices.Device method), 23
bind_mon() (zmq.devices.Proxy method), 25
bind_out() (zmq.devices.Device method), 24
bind_out_to_random_port() (zmq.devices.Device method), 24
bind_to_random_port() (zmq.Socket method), 8
buffer (zmq.Frame attribute), 17
built-in function
socket.bind(), 89
socket.connect(), 89
socket.getsockopt(), 90
socket.getsockopt_string(), 90
socket.recv(), 89
socket.recv_string(), 89
socket.send(), 89
socket.send_string(), 89
socket.setsockopt(), 90
socket.setsockopt_string(), 90
bytes (zmq.Frame attribute), 17

C
callHandlers() (zmq.log.handlers.TopicLogger method), 46
close() (zmq.eventloop.zmqstream.ZMQStream method), 33
close() (zmq.log.handlers.PUBHandler method), 44
close() (zmq.Socket method), 8
close (zmq.Context attribute), 5
closed (zmq.Socket attribute), 7
closed() (zmq.eventloop.zmqstream.ZMQStream method), 34
configure_curve() (zmq.auth.asyncio.AsyncioAuthenticator method), 39
configure_curve() (zmq.auth.Authenticator method), 37
configure_curve() (zmq.auth.ioloop.IOLoopAuthenticator method), 43
configure_curve() (zmq.auth.thread.ThreadAuthenticator method), 41
configure_gssapi() (zmq.auth.asyncio.AsyncioAuthenticator method), 39
configure_gssapi() (zmq.auth.Authenticator method), 37
configure_gssapi() (zmq.auth.ioloop.IOLoopAuthenticator method), 43
configure_gssapi() (zmq.auth.thread.ThreadAuthenticator method), 41
configure plain() (zmq.auth.asyncio.AsyncioAuthenticator method), 39
configure plain() (zmq.auth.Authenticator method), 37
configure plain() (zmq.auth.ioloop.IOLoopAuthenticator method), 43
configure plain() (zmq.auth.thread.ThreadAuthenticator method), 41
has() (in module zmq), 21
hasHandlers() (zmq.log.handlers.TopicLogger method), 47
hwm (zmq.Socket property), 10

I
info() (zmq.log.handlers.TopicLogger method), 47
install() (in module zmq.eventloop.ioloop), 29
instance() (zmq.Context class method), 6
io_loop (zmq.eventloop.zmqstream.ZMQStream attribute), 34
IOLoopAuthenticator (class in zmq.auth.ioloop), 42
is_alive() (zmq.auth.thread.ThreadAuthenticator method), 41
isEnabledFor() (zmq.log.handlers.TopicLogger method), 47

J
join() (zmq.devices.Device method), 24
join() (zmq.Socket method), 10

L
leave() (zmq.Socket method), 10
load_certificate() (in module zmq.auth), 38
load_certificates() (in module zmq.auth), 38
loads() (in module zmq.utils.jsonapi), 50
log (zmq.auth.thread.ThreadAuthenticator attribute), 41
log() (zmq.log.handlers.TopicLogger method), 47

M
makeRecord() (zmq.log.handlers.TopicLogger method), 48
manager (zmq.log.handlers.TopicLogger attribute), 48
MessageTracker (class in zmq), 18
modify() (zmq.Poller method), 18
module
  zmq, 5
  zmq.asyncio, 31
  zmq.auth, 36
  zmq.auth.asyncio, 38
  zmq.auth.thread, 40
  zmq.decorators, 28
  zmq.devices, 23
  zmq.eventloop.future, 29
  zmq.eventloop.ioloop, 29
  zmq.eventloop.zmqstream, 33
  zmq.green, 29
  zmq.log.handlers, 44
  zmq.ssh.tunnel, 48
  zmq.utils.jsonapi, 50
  zmq.utils.monitor, 50
  zmq.utils.win32, 51
  zmq.utils.z85, 51
monitor() (zmq.Socket method), 10
monitored_queue() (in module zmq.devices), 26
MonitoredQueue (class in zmq.devices), 27

N
name (zmq.log.handlers.PUBHandler property), 45
NotDone (class in zmq), 20

O
on_err() (zmq.eventloop.zmqstream.ZMQStream method), 34
on_recv() (zmq.eventloop.zmqstream.ZMQStream method), 34
on_recv_stream() (zmq.eventloop.zmqstream.ZMQStream method), 34
on_send() (zmq.eventloop.zmqstream.ZMQStream method), 35
on_send_stream() (zmq.eventloop.zmqstream.ZMQStream method), 35
open_tunnel() (in module zmq.ssh.tunnel), 48
openssh_tunnel() (in module zmq.ssh.tunnel), 48

P
paramiko_tunnel() (in module zmq.ssh.tunnel), 49
parse_monitor_message() (in module zmq.utils.monitor), 50
pipe (zmq.auth.thread.ThreadAuthenticator attribute), 41
pipe_endpoint (zmq.auth.thread.ThreadAuthenticator attribute), 41
poll() (zmq.Poller method), 18
poll() (zmq.Socket method), 11
Poller (class in zmq), 18
poller (zmq.eventloop.zmqstream.ZMQStream attribute), 35
ProcessDevice (class in zmq.devices), 25
ProcessMonitoredQueue (class in zmq.devices), 27
ProcessProxy (class in zmq.devices), 25
ProcessProxySteerable (class in zmq.devices), 26
Proxy (class in zmq.devices), 25
proxy() (in module zmq), 21
proxy_steerable() (in module zmq), 21
ProxySteerable (class in zmq.devices), 26
PUBHandler (class in zmq.log.handlers), 44
pyzmq_version() (in module zmq), 20
pyzmq_version_info() (in module zmq), 20

R
receiving() (zmq.eventloop.zmqstream.ZMQStream method), 35
recv() (zmq.Socket method), 11
recv_json() (zmq.Socket method), 11
recv_monitor_message() (in module zmq.utils.monitor), 50
recv_multipart() (zmq.Socket method), 11
recv_pyobj() (zmq.Socket method), 12
recv_serialized() (zmq.Socket method), 12
recv_string() (zmq.Socket method), 12
register() (zmq.Poller method), 19
release() (zmq.log.handlers.PUBHandler method), 45
removeFilter() (zmq.log.handlers.PUBHandler method), 45
removeFilter() (zmq.log.handlers.TopicLogger method), 48
removeHandler() (zmq.log.handlers.TopicLogger method), 48
root (zmq.log.handlers.TopicLogger attribute), 48
root_topic (zmq.log.handlers.PUBHandler property), 45
routing_id (zmq.Frame property), 17
select() (in module zmq), 19
select_random_ports() (in module zmq.ssh.tunnel), 49
send() (zmq.eventloop.zmqstream.ZMQStream method), 35
send() (zmq.Socket method), 12
send_json() (zmq.eventloop.zmqstream.ZMQStream method), 35
send_json() (zmq.Socket method), 13
send_multipart() (zmq.eventloop.zmqstream.ZMQStream method), 35
send_pyobj() (zmq.Socket method), 13
send_pyobj() (zmq.eventloop.zmqstream.ZMQStream method), 35
send_serialized() (zmq.Socket method), 14
send_string() (zmq.eventloop.zmqstream.ZMQStream method), 35
send_string() (zmq.Socket method), 14
send_unicode() (zmq.eventloop.zmqstream.ZMQStream method), 35
sending() (zmq.eventloop.zmqstream.ZMQStream method), 36
set() (zmq.Context method), 6
set() (zmq.Frame method), 17
set() (zmq.Socket method), 14
set_close_callback() (zmq.eventloop.zmqstream.ZMQStream method), 36
set_hwm() (zmq.Socket method), 15
set_name() (zmq.log.handlers.PUBHandler method), 46
set_string() (zmq.Socket method), 15
setFormatter() (zmq.log.handlers.PUBHandler method), 45
setLevel() (zmq.log.handlers.PUBHandler method), 45
setLevel() (zmq.log.handlers.TopicLogger method), 48
setRootTopic() (zmq.log.handlers.PUBHandler method), 45
setsockopt() (zmq.Context method), 6
setsockopt() (zmq.Socket method), 15
setsockopt_ctrl() (zmq.devices.ProxySteerable method), 26
setsockopt_in() (zmq.devices.Device method), 24
setsockopt_mon() (zmq.devices.Proxy method), 25
setsockopt_out() (zmq.devices.Device method), 24
setsockopt_string() (zmq.Socket method), 15
shadow() (zmq.Context class method), 6
shadow() (zmq.Socket class method), 16
shadow_pyczmq() (zmq.Context class method), 6
Socket (class in zmq), 7
socket (zmq.eventloop.zmqstream.ZMQStream attribute), 36
socket (zmq.log.handlers.PUBHandler attribute), 46
socket() (in module zmq.decorators), 28
socket() (zmq.Context method), 7
socket.bind() built-in function, 89
socket.connect() built-in function, 89
socket.getsockopt() built-in function, 90
socket.getsockopt_string() built-in function, 90
socket.recv() built-in function, 89
socket.recv_string() built-in function, 89
socket.send() built-in function, 89
socket.send_string() built-in function, 89
socket.setsockopt() built-in function, 90
socket.setsockopt_string() built-in function, 90
start() (zmq.auth.asyncio.AsycnioAuthenticator method), 40
start() (zmq.auth.Authenticator method), 38
start() (zmq.auth.ioloop.IOLoopAuthenticator method), 43
start() (zmq.auth.thread.ThreadAuthenticator method), 42
start() (zmq.devices.Device method), 24
stop() (zmq.auth.asyncio.AsycnioAuthenticator method), 40
stop() (zmq.auth.Authenticator method), 38
Index