PyZMQ Documentation

Release 15.0.0

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

November 04, 2015

Contents

Release 15.0.0

Date November 04, 2015

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 (3.2), and Python 2 (2.6), with no transformations or 2to3, as well as PyPy (at least 2.0 beta), thanks to a new CFFI backend.

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

Summary of Changes in PyZMQ

CHAPTER 1

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 libsodium for security. For pyzmq-15.0.0, this is 4.1.2.

CHAPTER 2

Using PyZMQ

2.1 The PyZMQ API

Release 15.0.0

Date November 04, 2015

2.1.1 zmq

Python bindings for 0MQ.

Basic Classes

Context

class zmq.**Context**(*io_threads=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.

__copy__(*memo=None*)
Copying a Context creates a shadow copy

__deepcopy__(*memo=None*)
Copying a Context creates a shadow copy

__del__()
deleting a Context should terminate it, without trying non-threadsafe destroy

__delattr__(*key*)
delete default sockopts as attributes

__getattr__(*key*)
get zmq options by attribute

__setattr__(*key*, *value*)
set zmq options by attribute

destroy(*linger=None*)
ctx.destroy(linger=None)

Close all sockets associated with this context, and then terminate the context. If linger is specified, the LINGER sockopt of the sockets will be set prior to closing.

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

get(*option*)
ctx.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*)
get default socket options for new sockets created by this Context

New in version 13.0.

classmethod instance(*io_threads=1*)
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:

class MyClass(object):

def __init__(self, context=None): self.context = context or Context.instance()

set(*option*, *optval*)
ctx.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*, *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.

New in version 14.1.

classmethod shadow_pyczmq(*ctx*)
Shadow an existing pyczmq context

ctx is the FFI *zctx_t* * pointer

New in version 14.1.

socket(*socket_type*)
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.

term()
ctx.term()

Close or terminate the context.

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:

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

then call `ctx.socket(socket_type)`:

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

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

__copy__(*memo=None*)
Copying a Socket creates a shadow copy

__deepcopy__(*memo=None*)
Copying a Socket creates a shadow copy

__enter__()
Sockets are context managers

New in version 14.4.

__getattr__(*key*)
get zmq options by attribute

__setattr__(*key*, *value*)
set zmq options by attribute

bind(*addr*)
s.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.

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*)
s.close(linger=None)

Close the socket.

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

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*)
s.connect(addr)

Connect to a remote 0MQ socket.

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*)
s.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.

get(*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

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** (*bitfield (int) [default: ZMQ_EVENTS_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 unicode string (unicode on py2, str on py3)

getsockopt()
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 unicode string (unicode on py2, str on py3)

hwm
get the High Water Mark

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

monitor(*addr*, *flags*)
s.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=1*)
poll the socket for events

The default is to poll forever for incoming events. Timeout is in milliseconds, if specified.

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** (*bitfield (int) [default: POLLIN]*) – The event flags to poll for (any combination of POLLIN|POLLOUT). The default is to check for incoming events (POLLIN).

Returns **events** – The events that are ready and waiting. Will be 0 if no events were ready by the time timeout was reached.

Return type bitfield (int)

recv(*flags=0*, *copy=True*, *track=False*)
s.recv(flags=0, copy=True, track=False)

Receive a message.

Parameters

- **flags** (*int*) – Any supported flag: NOBLOCK. If NOBLOCK is set, this method will raise a ZMQError with EAGAIN if a message is not ready. If NOBLOCK is not set, then this method will block until a message arrives.
- **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, Frame

Raises *ZMQError* – for any of the reasons zmq_msg_recv might fail.

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 recv flag.

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

Return type Python object

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 supported flag: NOBLOCK. If NOBLOCK is set, this method will raise a ZMQError with EAGAIN if a message is not ready. If NOBLOCK is not set, then this method will block until a message arrives.
- **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

recv_pyobj(*flags=0*)
receive a Python object as a message using pickle to serialize

Parameters **flags** (*int*) – Any valid recv flag.

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

Return type Python object

recv_string(*flags=0*, *encoding='utf-8'*)
receive a unicode string, as sent by send_string

Parameters

- **flags** (*int*) – Any valid recv flag.
- **encoding** (*str [default: 'utf-8']*) – The encoding to be used

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

Return type unicode string (unicode on py2, str on py3)

send(*data*, *flags=0*, *copy=True*, *track=False*)
s.send(data, flags=0, copy=True, track=False)

Send a message on this socket.

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

Parameters

- **data** (*object, str, Frame*) – The content of the message.
- **flags** (*int*) – Any supported flag: 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)

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.

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*) – Any valid send flag

send_multipart(*msg_parts*, *flags=0*, *copy=True*, *track=False*)
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*) – SNDMORE is handled automatically for frames before the last.
- **copy** (*bool, optional*) – Should the frame(s) be sent in a copying or non-copying manner.
- **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*)
send a Python object as a message using pickle to serialize

Parameters

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

send_string(*u*, *flags=0*, *copy=True*, *encoding='utf-8'*)
send a Python unicode string as a message with an encoding

0MQ communicates with raw bytes, so you must encode/decode text (unicode on py2, str on py3) around 0MQ.

Parameters

- **u** (*Python unicode string (unicode on py2, str on py3)*) – The unicode string to send.
- **flags** (*int, optional*) – Any valid send flag.
- **encoding** (*str [default: 'utf-8']*) – The encoding to be used

set(*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.

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** (*unicode string (unicode on py2, str on py3)*) – The value of the option to set.
- **encoding** (*str*) – The encoding to be used, default is utf8

setsockopt()
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** (*unicode string (unicode on py2, str on py3)*) – 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.

New in version 14.1.

socket_type

unbind(*addr*)
s.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

Frame

class zmq.**Frame**

__getattr__(*key*)
get zmq options by attribute

__setattr__(*key*, *value*)
set zmq options by attribute

buffer
A read-only buffer view 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)

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.

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** – This list of objects to track. This class can track the low-level Events used by the Message class, other MessageTrackers or actual Messages.

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

wait(*timeout=-1*)
mt.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=3*)
Modify the flags for an already registered 0MQ socket or native fd.

poll(*timeout=None*)
Poll the registered 0MQ or native fds for I/O.

Parameters **timeout** (*float, 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)`, 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`.

Return type list of tuples

register(*socket*, *flags=POLLIN|POLLOUT*)
p.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*)
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) -> (rlist*, *wlist*, *xlist*)
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=None*, *msg=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*, *msg='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=None*, *msg=None*)
Wrapper for zmq.EAGAIN

New in version 13.0.

ContextTerminated

class zmq.**ContextTerminated**(*errno=None*, *msg=None*)
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**()
return the version of libzmq as a string

zmq.**pyzmq_version**()
return the version of pyzmq as a string

zmq.**zmq_version_info**()
Return the version of ZeroMQ itself as a 3-tuple of ints.

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

New in version libzmq-4.1.

New in version 14.1.

zmq.**device**(*device_type*, *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.

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.**curve_keypair**()
generate a Z85 keypair for use with zmq.CURVE security

Requires libzmq (4.0) to have been linked with libsodium.

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

zmq.**get_includes**()
Return a list of directories to include for linking against pyzmq with cython.

2.1.2 devices

Functions

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.

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.

Module: zmq.devices

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

Base Devices

Device

class zmq.devices.**Device**(*device_type=3*, *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:

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

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

Parameters

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

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

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

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

daemon
int

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

context_factory
callable (class attribute)

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

bind_in(*addr*)
Enqueue ZMQ address for binding on in_socket.

See zmq.Socket.bind for details.

bind_out(*addr*)
Enqueue ZMQ address for binding on out_socket.

See zmq.Socket.bind for details.

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 zmq.devices.**ThreadDevice**(*device_type=3*, *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=3*, *in_type=None*, *out_type=None*)
A Device that will be run in a background Process.

See Device for details.

context_factory(*io_threads=1*, ***kwargs*)
Callable that returns a context. Typically either Context.instance or Context, depending on whether the device should share the global instance or not.

alias of Context

Proxy Devices

Proxy

class zmq.devices.**Proxy**(*in_type*, *out_type*, *mon_type=1*)
Threadsafe Proxy object.

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

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

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

bind_mon(*addr*)
Enqueue ZMQ address for binding on mon_socket.

See zmq.Socket.bind for details.

connect_mon(*addr*)
Enqueue ZMQ address for connecting on mon_socket.

See zmq.Socket.bind 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=1*)
Proxy in a Thread. See Proxy for more.

ProcessProxy

class zmq.devices.**ProcessProxy**(*in_type*, *out_type*, *mon_type=1*)
Proxy in a Process. See Proxy for more.

MonitoredQueue Devices

zmq.devices.**monitored_queue**()

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=1*, *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=1*, *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=1*, *in_prefix=b'in'*, *out_prefix=b'out'*)

Run zmq.monitored_queue in a background thread.

See MonitoredQueue and Proxy for details.

2.1.3 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.

2.1.4 eventloop.future

Module: eventloop.future

Future-returning APIs for coroutines.

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:tornado.gen`

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

ctx = Context()

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

Classes

Context

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

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

Socket

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

See also:

zmq.Socket for the inherited API.

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

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*)
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*)
Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

poll(*timeout=None*, *flags=1*)
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

2.1.5 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 :class:‘~.asyncio.Future‘s.

It also provides a *zmq.asyncio.ZMQEventLoop*.

```
import asyncio
import zmq
import zmq.asyncio

ctx = zmq.asyncio.Context()
loop = zmq.asyncio.ZMQEventLoop()
asyncio.set_event_loop(loop)

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

loop.run_until_complete(recv_and_process())
```

Classes

ZMQEventLoop

An asyncio event loop using zmq_poll for zmq socket support.

class zmq.asyncio.**ZMQEventLoop**(*selector=None*)
AsyncIO eventloop using zmq_poll

Context

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

class zmq.asyncio.**Context**(*io_threads=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*, *socket_type*, *io_loop=None*)
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*)
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*)
Send a complete multipart zmq message.

Returns a Future that resolves when sending is complete.

poll(*timeout=None*, *flags=1*)
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

2.1.6 auth

Module: auth

Utilities for ZAP authentication.

To run authentication in a background thread, see *zmq.auth.thread*. For integration with the tornado eventloop, see *zmq.auth.ioloop*.

New in version 14.1.

Authenticator

class zmq.auth.**Authenticator**(*context=None*, *encoding='utf-8'*, *log=None*)
Implementation of ZAP authentication for zmq connections.

Note: - libzmq provides four levels of security: default NULL (which the Authenticator does

not see), and authenticated NULL, PLAIN, and CURVE, 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.

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 PLAIN and CURVE, 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.

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.

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

2.1.7 auth.thread

Module: auth.thread

ZAP Authenticator in a Python Thread.

New in version 14.1.

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 PLAIN and CURVE, they will be allowed to continue with authentication.

whitelist is mutually exclusive with blacklist.

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.

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.

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.

is_alive()
Is the ZAP thread currently running?

start()
Start the authentication thread

stop()
Stop the authentication thread

2.1.8 auth.ioloop

Module: auth.ioloop

ZAP Authenticator integrated with the tornado IOLoop.

New in version 14.1.

IOLoopAuthenticator

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).

Connections from addresses not in the whitelist will be rejected.

- For NULL, all clients from this address will be accepted.
- For PLAIN and CURVE, 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.

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.

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

2.1.9 utils.win32

Module: zmq.utils.win32

Win32 compatibility utilities.

allow_interrupt

class zmq.utils.win32.**allow_interrupt**(*action=None*)
Utility for fixing CTRL-C events on Windows.

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

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

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

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

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

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

Example usage:

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

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

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

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

2.2 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.

2.2.1 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.

2.2.2 Thread Safety

In ØMQ, Contexts are threadsafe objects, but Sockets are **not**. It is safe to use a single Context (e.g. via *`zmq.Context.instance()`*) in your entire multithreaded application, but you should create sockets on a per-thread basis. If you share sockets across threads, you are likely to encounter uncatchable c-level crashes of your application unless you use judicious application of `threading.Lock`, but this approach is not recommended.

See also:

ZeroMQ API note on threadsafety on 2.2 or 3.2

2.2.3 Socket Options as Attributes

New in version 2.1.9.

In 0MQ, socket options are set/retrieved with the `set/getsockopt()` methods. With the class-based approach in pyzmq, it would be logical to perform these operations with simple attribute access, and this has been added in pyzmq 2.1.9. Simply assign to or request a Socket attribute with the (case-insensitive) name of a sockopt, and it should behave just as you would expect:

```
s = ctx.socket(zmq.DEALER)
s.identity = b'dealer'
s.hwm = 10
s.events
# 0
s.fd
# 16
```

Default Options on the Context

New in version 2.1.11.

Just like setting socket options as attributes on Sockets, you can do the same on Contexts. This affects the default options of any *new* sockets created after the assignment.

```
ctx = zmq.Context()
ctx.linger = 0
rep = ctx.socket(zmq.REP)
req = ctx.socket(zmq.REQ)
```

Socket options that do not apply to a socket (e.g. SUBSCRIBE on non-SUB sockets) will simply be ignored.

2.2.4 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 threadsafe communication (specifically a builtin `Queue` object), instantiating a MessageTracker takes a modest amount of time (10s of µs), so in situations instantiating many small messages, this can actually dominate performance. As a result, tracking is optional, via the `track` flag, which is optionally passed, always defaulting to `False`, in each of the three places where a Frame object (the pyzmq object for wrapping a segment of a message) is instantiated: The *`Frame`* constructor, and non-copying sends and receives.

A MessageTracker is very simple, and has just one method and one attribute. The property `MessageTracker.done` will be `True` when the Frame(s) being tracked are no longer in use by ØMQ, and *`MessageTracker.wait()`* will block, waiting for the Frame(s) to be released.

Note: A Frame cannot be tracked after it has been instantiated without tracking. If a Frame is to even have the *option* of tracking, it must be constructed with `track=True`.

2.2.5 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.

2.3 Serializing messages with PyZMQ

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

2.3.1 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.

2.3.2 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:

```
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)

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/recvs of numpy arrays including the dtype/shape data necessary for reconstructing the array.

```
import numpy

def send_array(socket, A, flags=0, copy=True, track=False):
    """send a numpy array with metadata"""
    md = dict(
        dtype = str(A.dtype),
        shape = A.shape,
    )
    socket.send_json(md, flags|zmq.SNDMORE)
    return socket.send(A, flags, copy=copy, track=track)

def recv_array(socket, flags=0, copy=True, track=False):
    """recv a numpy array"""
    md = socket.recv_json(flags=flags)
    msg = socket.recv(flags=flags, copy=copy, track=track)
    buf = buffer(msg)
    A = numpy.frombuffer(buf, dtype=md['dtype'])
    return A.reshape(md['shape'])
```

2.4 Devices in PyZMQ

See also:

ØMQ Guide Device coverage.

ØMQ has a notion of Devices - simple programs that manage a send-recv pattern for connecting two or more sockets.

Being full programs, devices include a `while(True)` loop and thus block execution permanently once invoked. We have provided in the `devices` subpackage some facilities for running these devices in the background, as well as a custom three-socket *MonitoredQueue* device.

2.4.1 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
```

2.4.2 MonitoredQueue

One of ØMQ's builtin devices is the `QUEUE`. This is a symmetric two-socket device that fully supports passing messages in either direction via any pattern. We saw a logical extension of the `QUEUE` as one that behaves in the same way with respect to the in/out sockets, but also sends every message in either direction *also* on a third *monitor* socket. For performance reasons, this *`monitored_queue()`* function is written in Cython, so the loop does not involve Python, and should have the same performance as the basic `QUEUE` device.

One shortcoming of the `QUEUE` device is that it does not support having `ROUTER` sockets as both input and output. This is because `ROUTER` sockets, when they receive a message, prepend the `IDENTITY` of the socket that sent the message (for use in routing the reply). The result is that the output socket will always try to route the incoming message back to the original sender, which is presumably not the intended pattern. In order for the queue to support a ROUTER-ROUTER connection, it must swap the first two parts of the message in order to get the right message out the other side.

To invoke a monitored queue is similar to invoking a regular ØMQ device:

```
from zmq.devices import monitored_queue
ins = ctx.socket(zmq.ROUTER)
outs = ctx.socket(zmq.DEALER)
mons = ctx.socket(zmq.PUB)
configure_sockets(ins,outs,mons)
monitored_queue(ins, outs, mons, in_prefix='in', out_prefix='out')
```

The *in_prefix* and *out_prefix* default to 'in' and 'out' respectively, and a PUB socket is most logical for the monitor socket, since it will never receive messages, and the in/out prefix is well suited to the PUB/SUB topic subscription model. All messages sent on *mons* will be multipart, the first part being the prefix corresponding to the socket that received the message.

Or for launching an MQ in the background, there are *ThreadMonitoredQueue* and *ProcessMonitoredQueue*, which function just like the base BackgroundDevice objects, but add `foo_mon()` methods for configuring the monitor socket.

2.5 Eventloops and PyZMQ

2.5.1 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()`.

install()

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:

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

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

```
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:

```
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:

```
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:

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

Futures and coroutines

PyZMQ 15 adds *zmq.eventloop.future*, containing a Socket subclass that returns `Future` objects for use in `tornado` coroutines.

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:

```
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.

```
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.

2.5.2 AsyncIO

PyZMQ 15 adds support for `asyncio` via *zmq.asyncio*.

2.5.3 PyZMQ and gevent

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

```
import zmq.green as zmq
```

Then write your code as normal.

Socket.send/recv and zmq.Poller are gevent-aware.

In PyZMQ 2.2.0.2, green.device and green.eventloop should be gevent-friendly as well.

Note: The green device does *not* release the GIL, unlike the true device in zmq.core.

---

zmq.green.eventloop includes minimally patched IOLoop/ZMQStream in order to use the gevent-enabled Poller, so you should be able to use the ZMQStream interface in gevent apps as well, though using two eventloops simultaneously (tornado + gevent) is not recommended.

Warning: There is a known issue in gevent 1.0 or libevent, which can cause zeromq socket events to be missed. PyZMQ works around this by adding a timeout so it will not wait forever for gevent to notice events. The only known solution for this is to use gevent 1.0, which is currently at 1.0b3, and does not exhibit this behavior.

See also:

zmq.green examples on GitHub.

zmq.green is simply gevent_zeromq, merged into the pyzmq project.

2.6 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.

2.6.1 PUB/SUB and Topics

The ØMQ PUB/SUB pattern consists of a PUB socket broadcasting messages, and a collection of SUB sockets that receive those messages. Each PUB message is a multipart-message, where the first part is interpreted as a topic. SUB sockets can subscribe to topics by setting their `SUBSCRIBE` sockopt, e.g.:

```
sub = ctx.socket(zmq.SUB)
sub.setsockopt(zmq.SUBSCRIBE, 'topic1')
sub.setsockopt(zmq.SUBSCRIBE, 'topic2')
```

When subscribed, the SUB socket will only receive messages where the first part *starts with* one of the topics set via `SUBSCRIBE`. The default behavior is to exclude all messages, and subscribing to the empty string ‘’ will receive all messages.

2.6.2 PUBHandler

The `PUBHandler` object is created for allowing the python logging to be emitted on a PUB socket. The main difference between a PUBHandler and a regular logging Handler is the inclusion of topics. For the most basic logging, you can simply create a PUBHandler with an interface or a configured PUB socket, and just let it go:

```
pub = context.socket(zmq.PUB)
pub.bind('tcp://*:12345')
handler = PUBHandler(pub)
logger = logging.getLogger()
logger.addHandler(handler)
```

At this point, all messages logged with the default logger will be broadcast on the pub socket.

the PUBHandler does work with topics, and the handler has an attribute `root_topic`:

```
handler.root_topic = 'myprogram'
```

Python loggers also have loglevels. The base topic of messages emitted by the PUBHandler will be of the form: `<handler.root_topic>.<loglevel>`, e.g. ‘myprogram.INFO’ or ‘whatever.ERROR’. This way, subscribers can easily subscribe to subsets of the logging messages. Log messages are always two-part, where the first part is the topic tree, and the second part is the actual log message.

```
>>> logger.info('hello there')
>>> print sub.recv_multipart()
['myprogram.INFO', 'hello there']
```

Subtopics

You can also add to the topic tree below the loglevel on an individual message basis. Assuming your logger is connected to a PUBHandler, you can add as many additional topics on the front of the message, which will be added always after the loglevel. A special delimiter defined at `zmq.log.handlers.TOPIC_DELIM` is scanned by the PUBHandler, so if you pass your own subtopics prior to that symbol, they will be stripped from the message and added to the topic tree:

```
>>> log_msg  = "hello there"
>>> subtopic = "sub.topic"
>>> msg = zmq.log.handlers.TOPIC_DELIM.join([subtopic, log_msg])
>>> logger.warn(msg)
>>> print sub.recv_multipart()
['myprogram.WARN.sub.topic', 'hello there']
```

2.7 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 not generally the same machine as server).
- remote port : the port on the remote machine that you want to connect to.
- local ip : the interface on your local machine you want to use (default: 127.0.0.1)
- local port : the local port you want to forward to the remote port (default: high random)

So once you have established the tunnel, connections to `localip:localport` will actually be connections to `remoteip:remoteport`.

In most cases, you have a zeromq url for a remote machine, but you need to tunnel the connection through an ssh server. This is

So if you would use this command from the same LAN as the remote machine:

```
sock.connect("tcp://10.0.1.2:5555")
```

to make the same connection from another machine that is outside the network, but you have ssh access to a machine `server` on the same LAN, you would simply do:

```
from zmq import ssh
ssh.tunnel_connection(sock, "tcp://10.0.1.2:5555", "server")
```

Note that `"server"` can actually be a fully specified `"user@server:port"` ssh url. Since this really just launches a shell command, all your ssh configuration of usernames, aliases, keys, etc. will be respected. If necessary, `tunnel_connection()` does take arguments for specific passwords, private keys (the ssh `-i` option), and non-default choice of whether to use paramiko.

If you are on the same network as the machine, but it is only listening on localhost, you can still connect by making the machine itself the server, and using loopback as the remote ip:

```
from zmq import ssh
ssh.tunnel_connection(sock, "tcp://127.0.0.1:5555", "10.0.1.2")
```

The `tunnel_connection()` function is a simple utility that forwards a random localhost port to the real destination, and connects a socket to the new local url, rather than the remote one that wouldn't actually work.

See also:

A short discussion of ssh tunnels: http://www.revsys.com/writings/quicktips/ssh-tunnel.html

CHAPTER 3

Notes from developing PyZMQ

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:

```
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:

Explicit Type	2.x	3.x
`bytes`	`str`	`bytes`
`unicode`	`unicode`	`str`
`basestring`	`basestring`	`(str, bytes)`

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.

```
#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:

```
# ...
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:

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

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

```
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:

```
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 (OS X, 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 only *sockopt* with a string value that can be queried with *getsockopt*

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

CHAPTER 5

Links

- ØMQ Home
- The ØMQ Guide
- PyZMQ Installation notes on the ZeroMQ website
- PyZMQ on GitHub
- Issue Tracker

Bibliography

[msgpack] Message Pack serialization library http://msgpack.org

[protobuf] Google Protocol Buffers http://code.google.com/p/protobuf

[zlib] Python stdlib module for zip compression: `zlib`

[blosc] Blosc: A blocking, shuffling and loss-less (and crazy-fast) compression library http://blosc.pytables.org/trac

Z

Symbols

A

B

C

D

F

G

H

I

J

L

M

N

P

R

S

T

U

W

Z