Designing and Testing PyZMQ Applications

Stefan Scherfke
OFFIS – Institute for Information Technology
Oldenburg, Germany
stefan.scherfke@offis.de

PyZMQ is a powerful and easy-to-use network layer. While ZeroMQ and PyZMQ are quite well documented and good introductory tutorials exist, no best-practice guide on how to design and especially to test larger or more complex PyZMQ applications could be found. This article shows a possible way to design, document and test real-world applications. The approach presented in this article was used for the development of a distributed simulation framework and proved to work quite well in this scenario.

Keywords: PyZMQ; ZeroMQ; ØMQ; Application Design; Testing.

1 Introduction

ZeroMQ [1] (or ØMQ or ZMQ) is an intelligent messaging framework and described as "sockets on steroids" [2]. That is, they look like normal TCP sockets but actually work as you would expect sockets to work. PyZMQ [3] adds even more convenience to them, which makes it a really a good choice if you want to implement a distributed application. Another big plus for ZeroMQ is that you can integrate sub-systems written in C, Java or any other language ØMQ supports (which are a lot).

ZeroMQ provides various socket types that let you easily implement the most common communication patterns:

REQ-REP: The request-reply pattern can be used for common client/server scenarios, where the client sends a request and the server only replies to these requests.

PUB-SUB: This pattern comprises a publisher that broadcasts messages and subscribers that register at a publisher and receive its messages.

PUSH-PULL: This pattern can be used to implement a pipeline where a producer pushes a message and one of multiple available consumers receives the message and processes it.

ROUTER, DEALER: Router and dealer sockets are the most flexible socket types an can be used to implement various routing algorithms to pass messages between multiple processes.

If you've never heard of ØMQ before, I recommend to read *ZeroMQ an Introduction* by Nicholas Piël [4], before you go on with this article.

The *ØMQ Guide* [2] and *PyZMQ's documentation* [5] are really good, so you can easily get started. However, when we began to implement a larger application with it (a distributed simulation framework), several questions arose which were not covered by the documentation:

- What's the best way do design our application?
- How can we keep it readable, flexible and maintainable?
- How do we test it?

I didn't find something like a best practice article that answered my questions. So in this series of articles, I'm going to talk about what I've learned during the last months. I'm not a PyZMQ expert, but what I've done so far works quite well and I never had more tests in a project than I do have now.

You'll find the source for the examples at *bitbucket* [6]. They are written in Python 3.2 and tested under Mac OS X Lion, Ubuntu 12.04 and Windows 7, 64 bit in each case.

In this article, I'm going to talk a bit about how you could generally design your application to be flexible, maintainable and testable. I also cover various types of testing, namely *unit*, *process* and *system testing*.

2 Application Design

In this section, I am going to briefly outline the pros and cons of different approaches for PyZMQ powered applications. Before you start to actually implement your application, you should sit down and thinks about which kind of processes and communication sequences between them you need. I will suggest a template to systematically write down your thoughts. Finally I will provide an example for a possible application architecture.

2.1 Comparison of Different Approaches

In this section, I will compare three different approaches to implement a PyZMQ application: One, that's easy, but limited in practical use, one that's more flexible, but not really pythonic and one, that needs a bit more setup, but is flexible and pythonic.

All three examples feature a simple ping process and a pong process with varying complexity. I use multiprocessing to run the pong process, because that's what you should usually do in real PyZMQ applications.

All three examples will have the following output:

```
(zmq)$ python blocking_recv.py
Pong got request: ping 0
Ping got reply: pong 0
...
Pong got request: ping 4
Ping got reply: pong 4
```

Let's start with the easy one first. You just use on of the socket's recv methods in a loop:

```python
# blocking_recv.py
import multiprocessing
import zmq

addr = 'tcp://127.0.0.1:5678'

def ping():
    """Sends ping requests and waits for replies."""
    context = zmq.Context()
    sock = context.socket(zmq.REQ)
    sock.bind(addr)

    for i in range(5):
        sock.send_unicode('ping %s' % i)
        rep = sock.recv_unicode()  # This blocks until we get something
        print('Ping got reply:', rep)

def pong():
    """Waits for ping requests and replies with a pong."""
    context = zmq.Context()
    sock = context.socket(zmq.REP)
    sock.connect(addr)
    for i in range(5):
        req = sock.recv_unicode()  # This also blocks
        print('Pong got request:', req)
        sock.send_unicode('pong %s' % i)


if __name__ == '__main__':
    pong_proc = multiprocessing.Process(target=pong)
    pong_proc.start()

    ping()

    pong_proc.join()
```

So this is very easy and no that much code. The problem with this is, that it only works well if your process only uses one socket. Unfortunately, in larger applications that is rather rarely the case.

A way to handle multiple sockets per process is polling. In addition to your context and socket(s), you need a *poller*. You also have to tell it which events on which socket you are going to poll:

```
# polling.py
def pong():
    """Waits for ping requests and replies with a pong."""
    context = zmq.Context()
    sock = context.socket(zmq.REP)
    sock.bind(addr)

    # Create a poller and register the events we want to poll
    poller = zmq.Poller()
    poller.register(sock, zmq.POLLIN|zmq.POLLOUT)

    for i in range(10):
        # Get all sockets that can do something
        socks = dict(poller.poll())

        # Check if we can receive something
        if sock in socks and socks[sock] == zmq.POLLIN:
            req = sock.recv_unicode()
            print('Pong got request:', req)

        # Check if we cann send something
        if sock in socks and socks[sock] == zmq.POLLOUT:
            sock.send_unicode('pong %s' % (i // 2))

    poller.unregister(sock)
```

You see, that our *pong* function got pretty ugly. You need 10 iterations to do five ping-pongs, because in each iteration you can either send or reply. And each socket you add to your process adds two more if-statements. You could improve that design if you created a base class wrapping the polling loop and just register sockets and callbacks in an inheriting class.

That brings us to our final example. PyZMQ includes an adapted Tornado[1] eventloop[2] that handles the polling and works with ZMQStreams[3], that wrap sockets and add some functionality:

```
# eventloop.py
from zmq.eventloop import ioloop, zmqstream
```

[1]Tornado (`http://www.tornadoweb.org/`) is an open source non-blocking web server and one of Facebook's open source technologies.

[2]`http://zeromq.github.com/pyzmq/eventloop.html`

[3]`http://zeromq.github.com/pyzmq/api/generated/zmq.eventloop.zmqstream.html#zmq.eventloop.zmqstream.ZMQStream`

```python
class Pong(multiprocessing.Process):
    """Waits for ping requests and replies with a pong."""
    def __init__(self):
        super().__init__()
        self.loop = None
        self.stream = None
        self.i = 0

    def run(self):
        """Initializes the event loop, creates the sockets/streams and
        starts the (blocking) loop.
        """
        context = zmq.Context()
        self.loop = ioloop.IOLoop.instance()  # This is the event loop

        sock = context.socket(zmq.REP)
        sock.bind(addr)
        # We need to create a stream from our socket and
        # register a callback for recv events.
        self.stream = zmqstream.ZMQStream(sock, self.loop)
        self.stream.on_recv(self.handle_ping)

        # Start the loop. It runs until we stop it.
        self.loop.start()

    def handle_ping(self, msg):
        """Handles ping requests and sends back a pong."""
        # req is a list of byte objects
        req = msg[0].decode()
        print('Pong got request:', req)
        self.stream.send_unicode('pong %s' % self.i)

        # We'll stop the loop after 5 pings
        self.i += 1
        if self.i == 5:
            self.stream.flush()
            self.loop.stop()
```

This even adds more boilerplate code, but it will pay of if you use more sockets and most of that stuff in *run()* can be put into a base class. Another drawback is, that the IOLoop only uses recv_multipart()[4]. So you always get a lists of byte strings which you have to decode or deserialize on your own. However, you can use all the *send* methods socket offers (like *send_unicode()* or *send_json()*). You can also stop the loop from within a message handler.

Though the last example seems to be the most complex one, it's the best approach for larger applications. The first example would only work for very simple scenarios with only one socket per process. The *polling* example can handle multiple sockets per process but its code will quickly become cluttered and unreadable if you add more sockets.

In the next sections, I'll discuss how you could implement a PyZMQ process that uses the event loop.

2.2 Communication Design

Before you start to implement anything, you should think about what kind of processes you need in your application and which messages they exchange. You should also decide what kind of message format and serialization you want to use. PyZMQ has built-in support for Unicode[5], JSON and Pickle.

[4]`http://zeromq.github.com/pyzmq/api/generated/zmq.core.socket.html#zmq.core.socket.Socket.recv_multipart`

[5]*send* sends plain (ASCII encoded) C strings which map to Python 3 *byte* objects. A string in Python 3 is always Unicode, so there's a separate `send_unicode` method to send Unicode strings. See `http://zeromq.github.com/pyzmq/unicode.html` for more details.

JSON is nice, because it's fast and lets you integrate processes written in other languages into you application. It's also a bit safer than pickle, because you cannot receive arbitrary objects[6]. The most straightforward syntax for JSON messages is to let them be triples *[msg_type, args, kwargs]*, where *msg_type* maps to a method name and *args* and *kwargs* get passed as positional and keyword arguments.

I strongly recommend you to document each chain of messages your application sends to perform a certain task. I do this with fancy PowerPoint graphics and with even fancier ASCII art in Sphinx[7]. Here is how I would document our ping-pong:

```
Sending pings
-------------

* If the ping process sends a *ping*, the pong processes responds with a
  *pong*.
* The number of pings (and pongs) is counted. The current ping count is
  sent with each message.

::

    PingProc      PongProc
     [REQ] ---1--> [REP]
           <--2---


    1 IN : ['ping, count']
    1 OUT: ['ping, count']

    2 IN : ['pong, count']
    2 OUT: ['pong, count']
```

First, I write some bullet points that explain how the processes behave and why they behave this way. This is followed by some kind of sequence diagram that shows when which process sends which message using which socket type. Finally, I write down how the messages are looking. `# IN` is what you would pass to *send_multipart* and `# OUT` is, what is received on the other side by *recv_multipart*. If one of the participating sockets is a *ROUTER* or *DEALER*, *IN* and *OUT* will differ (though that's not the case in this example). Everything in single quotation marks (`'`) represents a JSON serialized list.

If our pong process used a *ROUTER* socket instead of the *REP* socket, it would look like this:

```
    1 IN : ['ping, count']
    1 OUT: [ping_uuid, '', 'ping, count']

    2 IN : [ping_uuid, '', 'pong, count']
    2 OUT: ['pong, count']
```

This seems like a lot of tedious work, but it really helps a lot when you need to change or look up something a few weeks later.

2.3 Application Architecture

In the examples above, the *Pong* process was responsible for setting everything up, for receiving/sending messages and for the actual application logic (counting incoming pings and creating a pong).

Such a monolithic architecture is not a very good design—at least if your application gets more complex than our little ping-pong example. What we can do about this is to extract most of the setup-related code into a base class which all your processes can inherit from, separate message handling and (de)serialization from it and finally put all the actual application logic into a separate (PyZMQ-independent) class. This will result in a three-level architecture:

[6] `http://docs.python.org/py3k/library/pickle#restricting-globals`
[7] `http://sphinx.pocoo.org`

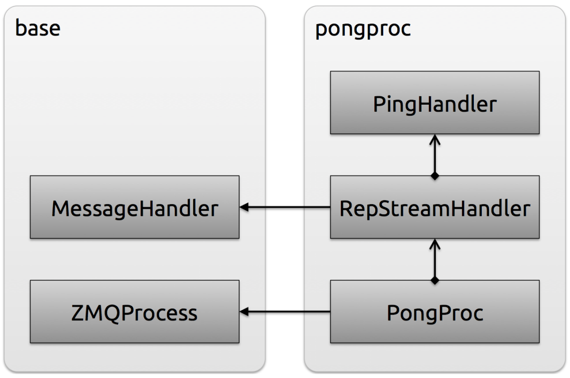

Figure 1: The refactored PongProc class now consists of three layers. The main class *PongProc* inherits *ZMQProcess*. Every stream gets a *MessageHandler*. In our example it is called *RepStreamHandler*. Finally, you can have one ore more classes containing the (PyZMQ-agnostic) application logic. In our example, it's called *PingHandler*, because it handles incoming pings.

1. The lowest level will contain the entry point of the process. It will set-up everything and start the event loop. A common base class provides utilities for creating sockets/streams and setting everything up.
2. The second level is message handling and (de) serialization. A base class performs the (de)serialization and error handling. A message handler inherits this class and implements a method for each message type that should be handled.
3. The third level will be the application logic and completely PyZMQ-agnostic.

Base classes should be defined for the first two levels to reduce redundant code in multiple processes or message handlers. Figure 1 shows the five classes our process is going to consist of.

2.3.1 ZmqProcess – The Base Class for all Processes

The base class basically implements two things:

- a *setup* method that creates a context and a loop
- a *stream* factory method for streams with a *on_recv* callback. It creates a socket and can connect/bind it to a given address or bind it to a random port (that's why it returns the port number in addition to the stream itself).

It also inherits *multiprocessing.Process*[8] so that it is easier to spawn it as a sub-process. Of course, you can also just call its *run()* method from you *main()* function.

```python
# example_app/base.py
import multiprocessing
from zmq.eventloop import ioloop, zmqstream
import zmq


class ZmqProcess(multiprocessing.Process):
    """
    This is the base for all processes and offers utility functions
    for setup and creating new streams.
    """
```

[8]http://docs.python.org/py3k/library/multiprocessing#process-and-exceptions

```python
    def __init__(self):
        super().__init__()

        self.context = None
        """The ZMQ "zmq.Context" instance."""

        self.loop = None
        """PyZMQ’s event loop ("zmq.eventloop.ioloop.IOLoop")."""

    def setup(self):
        """Creates a "context" and an event "loop" for the process."""
        self.context = zmq.Context()
        self.loop = ioloop.IOLoop.instance()

    def stream(self, sock_type, addr, bind, callback=None, subscribe=b’’):
        """
        Creates a "zmq.eventloop.zmqstream.ZMQStream".

        :param sock_type: The ZMQ socket type (e.g. ‘‘zmq.REQ‘‘)
        :param addr: Address to bind or connect to formatted as *host:port*,
                *(host, port)* or *host* (bind to random port).
        :param bind: Binds to *addr* if ‘‘True‘‘ or tries to connect to it
                otherwise.
        :param callback: A callback for
                "zmq.eventloop.zmqstream.ZMQStream.on_recv", optional
        :param subscribe: Subscription pattern for *SUB* sockets, optional,
                defaults to ‘‘b’’‘‘.
        :returns: A tuple containg the stream and the port number.
        """
        sock = self.context.socket(sock_type)

        # addr may be ’host:port’ or (’host’, port)
        if isinstance(addr, str):
            addr = addr.split(’:’)
        host, port = addr if len(addr) == 2 else (addr[0], None)

        # Bind/connect the socket
        if bind:
            if port:
                sock.bind(’tcp://%s:%s’ % (host, port))
            else:
                port = sock.bind_to_random_port(’tcp://%s’ % host)
        else:
            sock.connect(’tcp://%s:%s’ % (host, port))

        # Add a default subscription for SUB sockets
        if sock_type == zmq.SUB:
            sock.setsockopt(zmq.SUBSCRIBE, subscribe)

        # Create the stream and add the callback
        stream = zmqstream.ZMQStream(sock, self.loop)
        if callback:
            stream.on_recv(callback)

        return stream, int(port)
```

2.3.2 PongProc – The Actual Process

The *PongProc* inherits *ZmqProcess* and is the main class for our process. It creates the streams, starts the event loop and dispatches all messages to the appropriate handlers:

```python
# example_app/pongproc.py
import zmq
import base
```

```python
host = '127.0.0.1'
port = 5678

class PongProc(base.ZmqProcess):
    """
    Main processes for the Ponger. It handles ping requests and sends back
    a pong.
    """
    def __init__(self, bind_addr):
        super().__init__()
        self.bind_addr = bind_addr
        self.rep_stream = None
        self.ping_handler = PingHandler()

    def setup(self):
        """Sets up PyZMQ and creates all streams."""
        super().setup()

        # Create the stream and add the message handler
        self.rep_stream, _ = self.stream(zmq.REP, self.bind_addr, bind=True)
        self.rep_stream.on_recv(RepStreamHandler(self.rep_stream, self.stop,
                                                 self.ping_handler))

    def run(self):
        """Sets up everything and starts the event loop."""
        self.setup()
        self.loop.start()

    def stop(self):
        """Stops the event loop."""
        self.loop.stop()
```

If you are going to start this process as a sub-process via *start*, make sure everything you instantiate in *__init__* is pickle-able or it won't work on Windows (Linux and Mac OS X use *fork* to create a sub-process and *fork* just makes a copy of the main process and gives it a new process ID. On Windows, there is no *fork* and the context of your main process is pickled and sent to the sub-process [7]).

In *setup*, call `super().setup()` before you create a stream or you won't have a *loop* instance for them. We call *setup* from *run*, because the context must be created within the new system process, which wouldn't be the case if we called *setup* from *__init__*.

The *stop* method is not really necessary in this example, but it can be used to send stop messages to sub-processes when the main process terminates and to do other kinds of clean-up. You can also execute it if you except a `KeyboardInterrupt` after calling *run*:

```python
def main():
    try:
        proc = PongProc('127.0.0.1:5555')
        proc.run()
    except KeyboardInterrupt:
        proc.stop()
```

2.3.3 MessageHandler – The Base Class for Message Handlers

A PyZMQ message handler can be any callable that accepts one argument—the list of message parts as byte objects. Hence, our *MessageHandler* class needs to implement *__call__*:

```python
# exmaple_app/base.py
from zmq.utils import jsonapi as json

class MessageHandler(object):
    """
    Base class for message handlers for a "ZMQProcess".
    Inheriting classes only need to implement a handler function for each
    message type.
    """
    def __init__(self, json_load=-1):
        self._json_load = json_load

    def __call__(self, msg):
        """
        Gets called when a messages is received by the stream this handlers is
        registered at. *msg* is a list as return by
        "zmq.core.socket.Socket.recv_multipart".
        """
        # Try to JSON-decode the index "self._json_load" of the message
        i = self._json_load
        msg_type, data = json.loads(msg[i])
        msg[i] = data

        # Get the actual message handler and call it ("private" methods should
        # not be used as method handlers).
        if msg_type.startswith('_'):
            raise AttributeError('%s starts with an "_"' % msg_type)

        getattr(self, msg_type)(*msg)
```

As you can see, it's quite simple. It just tries to JSON-load the index defined by `self._json_load`. We earlier defined, that the first element of the JSON-encoded message defines the message type (e.g., *ping*). If an attribute of the same name exists in the inheriting class, it is called with the remainder of the message.

You can also add logging or additional security measures here, but that is not necessary here.

2.3.4 RepStreamHandler – The Concrete Message Handler

This class inherits the *MessageHandler*, that I just showed you, and is used in *PongProc.setup*. It defines a handler method for *ping* messages and the *plzdiekthxbye* stop message. In its *__init__* it receives references to the *rep_stream*, PongProcs *stop* method and to the *ping_handler*, our actual application logic:

```
# example_app/pongproc.py

class RepStreamHandler(base.MessageHandler):
    """Handels messages arrvinge at the PongProc's REP stream."""
    def __init__(self, rep_stream, stop, ping_handler):
        super().__init__()
        self._rep_stream = rep_stream
        self._stop = stop
        self._ping_handler = ping_handler

    def ping(self, data):
        """Send back a pong."""
        rep = self._ping_handler.make_pong(data)
        self._rep_stream.send_json(rep)

    def plzdiekthxbye(self, data):
        """Just calls "PongProc.stop"."""
        self._stop()
```

2.3.5 PingHandler – The Application Logic

The *PingHandler* contains the actual application logic (which is not much, in this example). The *make_pong* method just gets the number of pings sent with the *ping* message and creates a new *pong* message. The serialization is done by *PongProc*, so our Handler does not depend on PyZMQ:

```python
# example_app/pongproc.py

class PingHandler(object):

    def make_pong(self, num_pings):
        """Creates and returns a pong message."""
        print('Pong got request number %s' % num_pings)

        return ['pong', num_pings]
```

2.3.6 Summary

In this section I presented a class structure that allows you to implement applications with multiple processes and multiple streams per process with relatively little overhead.

Each new process must inherit *ZMQProcess*. This class encapsulates all setup-related code and helpers for actual process implementations (which is called *PongProc* in our case). Another base class, *MessageHandler*, provides a similar functionality for message handlers that you attach to a stream. In our simple example, we only need one message handler, namely *RepStreamHandler*.

The actual application logic should be completely PyZMQ-agnostic. This improves the testability of your application and also eases the migration to another socket library if your requirements should change some day.

3 Testing

Now that we have an appropriate design for our application, it's time for testing—remember, *if it's not tested, it's broken.*

My favorite testing tools are *pytest* by Holger Krekel [8] and *Mock* by Michael Ford [9].[9] Pytest is particularly awesome because of its re-evaluation of `assert` statements. If your test contains an `assert spam == 'eggs'` and the assert fails, pytest re-evaluates it and prints the value of `spam`. Really helpful and you don't need any boilerplate code for that. Mock is really nice for mocking external dependencies and asserting that your code called them in the correct way.

If you cloned the repository for this article, just run `py.test` from its root directory:

```
$ pip install pytest mock
...
Successfully installed pytest mock
Cleaning up...
$ py.test
================== test session starts ====================
platform darwin -- Python 3.2.2 -- pytest-2.2.3
collected 11 items

example_app/test/test_base.py ....
example_app/test/test_pongproc.py .......

================ 11 passed in 0.12 seconds =================
```

In the following sections, I will discuss how you can test your application on various levels, ranging from the method level (Unit tests) to tests of complete processes to system tests of your complete application.

[9]Python's built-in *unittest* module and *nose* are viable alternatives to *pytest*. There are also a few other mocking libraries available. The code coverage of your tests can be measured with *Coverage.py*, which can be used from *pytest* via the *pytest-cov* plug-in. You can find more information in the Python Testing Tools Wiki (`http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy`) and the Testing in Python mailing list (`http://lists.idyll.org/listinfo/testing-in-python`).

3.1 Unit Testing

The probability that PyZMQ works correctly is very high. The probability that your code will call a PyZMQ function in such a way that it blocks forever and halts your test runner is also very high. Therefore, it's a good idea to mock everything PyZMQ-related for your unit tests. And since your application logic might also not be implemented when you start testing your process, you should mock that, too.

What you'll actually end up testing is the following:

- Does your message handler call your application logic in the right way given a certain input message?
- Does your message handler create and send the correct reply based on the return value of your application logic?

3.1.1 ZmqProcess

Let's start with `ZmqProcess` again. After all, everything else depends on it. We need to test if its *stream* method can handle various address formats, if it creates or binds correctly and if it performs a default subscription for *SUB* sockets.

Pytest 2.2 introduced a *parametrize* decorator, that helps calling a test multiple times with varying inputs. You just define one or more arguments for your test function and a list of values for these arguments. For *test_ stream*, I only need a *kwargs* parameter containing the parameters for the *stream* call:

```python
# example_app/test/test_zmqproc.py
from zmq.eventloop import ioloop
import mock
import pytest
import zmq
import zmqproc

class TestZmqProcess(object):
    """Tests for "base.ZmqProcess"."""

    @pytest.mark.parametrize('kwargs', [
        dict(sock_type=23, addr='127.0.0.1:1234', bind=True,
             callback=mock.Mock()),
        dict(sock_type=23, addr='127.0.0.1', bind=True,
             callback=mock.Mock()),
        dict(sock_type=zmq.SUB, addr=('localhost', 1234), bind=False,
             callback=mock.Mock(), subscribe=b'ohai'),
    ])
    def test_stream(self, kwargs):
```

The next step is to create an instance of *ZmqProcess* and patch some of its attributes. We also need to set a defined return value for the socket's *bind_ to_ random_ port* method:

```python
# example_app/test/test_zmqproc.py

        zp = base.ZmqProcess()

        # Patch the ZmqProcess instance
        zp.context = mock.Mock(spec_set=zmq.Context)
        zp.loop = mock.Mock(spec_set=ioloop.IOLoop)
        sock_mock = zp.context.socket.return_value
        sock_mock.bind_to_random_port.return_value = 42
```

For the actual test, we also need to patch *ZMQStream*. Although *mock.patch* could work as a function decorator, we need to use it as context processor if we also use pytest funcargs (e.g., via the *parametrize* decorator).

```
# example_app/test/test_zmqproc.py

        # Patch ZMQStream and start testing
        with mock.patch('zmq.eventloop.zmqstream.ZMQStream') as zmqstream_mock:
            stream, port = zp.stream(**kwargs)
```

Finally, we can check the return values of our *stream* method and it made the correct calls to create the stream:

```
# example_app/test/test_zmqproc.py

            # Assert that the return values are correct
            assert stream is zmqstream_mock.return_value
            if isinstance(kwargs['addr'], tuple):
                assert port == kwargs['addr'][1]
            elif ':' in kwargs['addr']:
                assert port == int(kwargs['addr'][-4:])
            else:
                assert port == sock_mock.bind_to_random_port.return_value

            # Check that the socket was crated correctly
            assert zp.context.socket.call_args == ((kwargs['sock_type'],), {})
            if kwargs['bind'] and ':' in kwargs['addr']:
                sock_mock.bind.assert_called_with('tcp://%s' % kwargs['addr'])
            elif kwargs['bind']:
                sock_mock.bind_to_random_port.assert_called_with(
                        'tcp://%s' % kwargs['addr'])
            else:
                sock_mock.connect.assert_called_width(
                        'tcp://%s:%s' % kwargs['addr'])

            # Check creation of the stream
            zmqstream_mock.assert_called_with(sock_mock, zp.loop)
            zmqstream_mock.return_value.on_recv.assert_called_with(
                    kwargs['callback'])

            # Check default subscribtion
            if 'subscribe' in kwargs:
                sock_mock.setsockopt.assert_called_with(zmq.SUBSCRIBE,
                                                        kwargs['subscribe'])
```

3.1.2 MessageHandler

The *MessageHandler* base class has only one method, *__call__*, but I split the test for it into two methods—one that tests the JSON-loading functionality and one that checks if the correct handler method is called:

```
# example_app/test/test_base.py

class TestMessageHandler(object):
    """Tests for "base.TestMessageHandler"."""

    @pytest.mark.parametrize(('idx', 'msg'), [
        (-1, [23, b'["test", null]']),
        (1, [23, b'["test", "spam"]', 42]),
        (TypeError, [23, 42]),
        (ValueError, [23, b'["test"]23spam']),
    ])
    def test_call_json_load(self, idx, msg):
        handler = mock.Mock()
        mh = base.MessageHandler(idx if isinstance(idx, int) else -1)
        mh.test = handler

        if isinstance(idx, int):
            mh(msg)
            assert handler.call_count == 1
        else:
            pytest.raises(idx, mh, msg)

    @pytest.mark.parametrize(('ok', 'msg'), [
        (True, [23, b'["test", "spam"]', 42]),
        (AttributeError, [23, b'["_test", "spam"]', 42]),
        (TypeError, [23, b'["spam", "spam"]', 42]),
        (AttributeError, [23, b'["eggs", "spam"]', 42]),
    ])
    def test_call_get_handler(self, ok, msg):
        handler = mock.Mock()
        mh = base.MessageHandler(1)
        mh.test = handler
        mh.spam = 'spam'

        if ok is True:
            mh(msg)
            handler.assert_called_with(msg[0], 'spam', msg[2])
        else:
            pytest.raises(ok, mh, msg)
```

3.1.3 PongProc

Testing the *PongProc* is not much different from testing its base class. *pytest_funcarg__pp* will instantiate a *PongProc* instance for each test that has a `pp` argument[10]. The tests for *setup*, *run* and *stop* are easy to do. We create a few mocks and then ask them if the tested function called them correctly:

```
# example_app/test/test_pongproc.py
from zmq.utils import jsonapi as json
import mock, pytest, zmq
import pongproc

host, port = '127.0.0.1', 5678

def pytest_funcarg__pp(request):
    """Creates a PongProc instance."""
    return pongproc.PongProc((host, port))


class TestPongProc(object):
    """Tests "pongproc.PongProc"."""

    def test_setup(self, pp):
        def make_stream(*args, **kwargs):
            stream = mock.Mock()
            stream.type = args[0]
            return stream, mock.Mock()
        pp.stream = mock.Mock(side_effect=make_stream)

        with mock.patch('base.ZmqProcess.setup') as setup_mock:
            pp.setup()
            assert setup_mock.call_count == 1

        assert pp.stream.call_args_list == [
        pp.stream.assert_called_once_with(zmq.REP, (host, port), bind=True)
        assert pp.rep_stream.type == zmq.REP

        # Test if the message handler was configured correctly
```

[10]Pytest *funcargs* allow you to inject objects into tests. The results of a method *pytest_funcarg__NAME* will be passed to every test case that has a parameter *NAME*.

```python
        rsh = pp.rep_stream.on_recv.call_args[0][0]  # Get the msg handler
        assert rsh._rep_stream == pp.rep_stream
        assert rsh._stop == pp.stop

    def test_run(self, pp):
        pp.setup = mock.Mock()
        pp.loop = mock.Mock()

        pp.run()

        assert pp.setup.call_count == 1
        assert pp.loop.start.call_count == 1

    def test_stop(self, pp):
        pp.loop = mock.Mock()
        pp.stop()
        assert pp.loop.stop.call_count == 1
```

3.1.4 RepStreamHandler

Testing the actual message handler requires some mocks, but is, apart from that, straight forward. A *funcarg* method creates an instance of the message handler for each test case which we feed with a message. We then check if the application logic was called correctly and/or if a correct reply is sent:

```python
# example_app/test/test_pongproc.py

def pytest_funcarg__rsh(request):
    """Creates a RepStreamHandler instance."""
    return pongproc.RepStreamHandler(
            rep_stream=mock.Mock(),
            stop=mock.Mock(),
            ping_handler=mock.Mock(spec_set=pongproc.PingHandler()))


class TestRepStreamHandler(object):
    def test_ping(self, rsh):
        msg = ['ping', 1]
        retval = 'spam'
        rsh._ping_handler = mock.Mock(spec_set=pongproc.PingHandler)
        rsh._ping_handler.make_pong.return_value = retval

        rsh([json.dumps(msg)])

        rsh._ping_handler.make_pong.assert_called_with(msg[1])
        rsh._rep_stream.send_json.assert_called_with(retval)

    def test_plzdiekthybye(self, rsh):
        rsh([b'["plzdiekthxbye", null]'])
        assert rsh._stop.call_count == 1
```

3.1.5 PingHandler

When we are done with all that network stuff, we can finally test the application logic. Easy-peasy in our case:

```python
# example_app/test/test_pongproc.py

def pytest_funcarg__ph(request):
    """Creates a PingHandler instance."""
    return pongproc.PingHandler()

class TestPingHandler(object):
    def test_make_pong(self, ph):
        ping_num = 23
        ret = ph.make_pong(ping_num)
        assert ret == ['pong', ping_num]
```

3.1.6 Summary

Thanks to the Mock library, unit testing PyZMQ apps is really not that hard and not much different from normal unit testing. However, what we know now is only, that our process should work *in theory*. We haven't yet started it and sent real messages to it.

The next section will show you how you can automate testing complete processes.

3.2 Process Testing

Once you've made sure that your message dispatching and application logic works fine, you can actually start sending real messages to your process and checking real replies. This can be done for single processes—I call this *process testing*—and for your complete application (*system testing*).

When you test a single process, you create sockets that mimic all processes the tested process communicates with. When you do a system test, you only mimic a client or just invoke your program from the command line and check its output (e.g., what it prints to *stdtout* and *stderr* or results written to a database).

I will start with process testing, which is a bit more generalizable than system testing.

The biggest problem I ran into when I started testing processes was that I often made blocking calls to *recv* methods and these halted my tests and gave me no output about what actually went wrong. Though you can make them non-blocking by passing `zmq.NOBLOCK` as an extra argument, this doesn't solve your problems. You will now need a very precise timing and many `time.sleep(x)` calls, because *recv* will instantly raise an error if there is nothing to be received.

My solution for this was to wrap PyZMQ sockets and add a timeout to its *send* and *recv* methods. The following wrapper will try to receive something for one second and raise an exception if that failed. There's also a simple wrapper[11] for methods like *connect* or *bind*, but it is not very interesting, so I will omit it here.

```
# test/support.py

def get_wrapped_fwd(func):
    """
    Returns a wrapper, that tries to call *func* multiple time in non-blocking
    mode before rasing an "zmq.ZMQError".
    """
    def forwarder(*args, **kwargs):
        # 100 tries * 0.01 second == 1 second
        for i in range(100):
            try:
                rep = func(*args, flags=zmq.NOBLOCK, **kwargs)
                return rep

            except zmq.ZMQError:
                time.sleep(0.01)

        # We should not get here, so raise an error.
        msg = 'Could not %s message.' % func.__name__[:4]
        raise zmq.ZMQError(msg)

    return forwarder
```

This wrapper is now used to create a *TestSocket* class with the desired behavior:

[11]`https://bitbucket.org/ssc/pyzmq-article/src/tip/example_app/test/support.py#cl-2`

```python
# test/support.py

class TestSocket(object):
    """
    Wraps ZMQ "zmq.core.socket.Socket". All *recv* and *send* methods
    will be called multiple times in non-blocking mode before a
    "zmq.ZMQError" is raised.
    """
    def __init__(self, context, sock_type):
        self._context = context

        sock = context.socket(sock_type)
        self._sock = sock

        forwards = [  # These methods can simply be forwarded
            sock.bind, sock.bind_to_random_port,
            sock.connect, sock.close, sock.setsockopt,
        ]
        wrapped_fwd = [  # These methods are wrapped with a for loop
            sock.recv, sock.recv_json, sock.recv_multipart, sock.recv_unicode,
            sock.send, sock.send_json, sock.send_multipart, sock.send_unicode,
        ]

        for func in forwards:
            setattr(self, func.__name__, get_forwarder(func))

        for func in wrapped_fwd:
            setattr(self, func.__name__, get_wrapped_fwd(func))
```

In order to reuse the same ports for all test methods, you need to orderly close all sockets after each test. To handle method level setup/teardown in pytest, you need to implement a *setup_ method* and a *teardown_ method*. In the setup method, you create one or more *TestSocket* instances that mimic other processes and you also start the process to be tested:

```python
# test/process/test_pongproc.py
import pytest
import zmq
from test.support import ProcessTest, make_sock
import pongproc

host = '127.0.0.1'
port = 5678

class TestProngProc(ProcessTest):
    """Communication test for the Platform Manager process."""

    def setup_method(self, method):
        """
        Creates and starts a PongProc process and sets up sockets to
        communicate with it.
        """
        self.context = zmq.Context()

        # make_sock creates and connects a TestSocket that we will use to
        # mimic the Ping process
        self.req_sock = make_sock(self.context, zmq.REQ,
                                  connect=(host, port))

        self.pp = pongproc.PongProc((host, port))
        self.pp.start()

    def teardown_method(self, method):
        """
        Sends a kill message to the pp and waits for the process to terminate.
        """
        # Send a stop message to the prong process and wait until it joins
        self.req_sock.send_multipart([b'["plzdiekthxbye", null]'])
        self.pp.join()

        self.req_sock.close()
```

You may have noticed that our test class inherits *ProcessTest*. This class and some helpers in a conftest.py[12] allow us to use some magic that improves the readability of the actual test:

```
# test/process/test_pongproc.py

    def test_ping(self):
        """Tests a ping-pong sequence."""
        yield ('send', self.req_sock, [], ['ping', 1])

        reply = yield ('recv', self.req_sock)
        assert reply == [['pong', 1]]
```

You can just yield *send* or *recv* events from your test case! When you yield a *send*, the test machinery tries to send a message via the specified socket. When you yield a receive, *ProcessTest* tries to receive something from the socket and sends its result back to your test function, so that you can easily compare the reply with the expected result.

The example above is roughly equivalent to the following code:

```
self.req_sock.send_multipart([] + [json.dumps(['ping', 1])])

reply = self.req_sock.recv_multipart()
reply[-1] = json.loads[reply[-1]]
assert reply == [['pong', 1]]
```

So how does this work? By default, if pytest finds a test function that is a generator, it assumes that it generates further test functions. Hence, our first step is to override this behavior. We can do this in a `conftest.py` file in the `test/process/` directory by implementing a *pytest_pycollect_makeitem*[13] function. In this case, we collect generator functions like normal functions. We also need to tell pytest how to run a test on the collected generator functions. This can be done by implementing *pytest_runtest_call*[14]. If the object we are going to test is a generator function, we call the *run* method of the object's instance (this method is inherited from emphProcessTest) and pass the generator function to it.

The *run* method[15] simply iterates over all events our test function generates and calls a method with the name of the event (e.g., *send* or *recv*). Their return value is sent back into the generator. If an error occurs, the exception's traceback is modified to point to the line of code that yielded the according event and not to the line that actually raised the exception (e.g., *TestSocket.recv*), which would not be very useful in this case.

The *conftest* plug-in and the *run* method involve a lot of magic, so I omitted their code. You can find it at bitbucket by following the links in the respective footnotes.

The *ProcessTest*'s methods *send* and *recv* roughly do the same as the snippet I showed you above:

```
# test/support.py

    def send(self, socket, header, body, extra_data=[]):
        """
        JSON-encodes *body*, concatenates it with *header*, appends
        *extra_data* and sends it as multipart message over *socket*.

        *header* and *extra_data* should be lists containg byte objects or
        objects implementing the buffer interface (like NumPy arrays).
```

[12]http://pytest.org/latest/plugins.html
[13]https://bitbucket.org/ssc/pyzmq-article/src/tip/example_app/test/process/conftest.py#cl-4
[14]https://bitbucket.org/ssc/pyzmq-article/src/tip/example_app/test/process/conftest.py#cl-15
[15]https://bitbucket.org/ssc/pyzmq-article/src/tip/example_app/test/support.py#cl-126

```
        """
        socket.send_multipart(header + [json.dumps(body)] + extra_data)

    def recv(self, socket, json_load_index=-1):
        """
        Receives and returns a multipart message from *socket* and tries to
        JSON-decode the item at position *json_load_index* (defaults to ``-1``;
        the last element in the list). The original byte string will be
        replaced by the loaded object. Set *json_load_index* to ``None`` to get
        the original, unchanged message.
        """
        msg = socket.recv_multipart()
        if json_load_index is not None:
            msg[json_load_index] = json.loads(msg[json_load_index])
        return msg
```

You can even add your own event handler to your test class. I used this, for example, to add a *log* event that checks if a PyZMQ log handler sent the expected log messages:

```
def log(self, substr=''):
    """
    Receives a message and asserts, that it is a log message and that
    *substr* is in that message.

    Usage:
        yield ('log', 'Ai iz in ur log mesage')
    """
    msg = self.log_sock.recv_json()
    assert msg[0] == 'log_message'
    assert substr in msg[1]
```

3.2.1 What if your process starts further subprocesses?

In some cases, the process you are about to test starts additional subprocesses that you don't want to test. Even worse, these processes might communicate via sockets bound to random ports. They might also depend on excepting a *KeyboardInterrupt* to send stop messages to child processes or to clean something up.

The last problem is quite easy to solve: You just a send a *SIGINT* (an interrupt signal) to your process from the test:

```
import os, signal

def teardown_method(self, method):
    os.kill(self.my_proc.pid, signal.SIGINT)
    self.my_proc.join()

    # Now you can close the test sockets
```

If you don't want to start a certain subprocess, you can just mock it. Imagine, you have two processes `a.A` and `b.B`, where A starts B, then you just mock B before starting A:

```
with mock.patch('b.B'):
    self.a = A()
    self.a.start()
```

Imagine now, that A binds a socket to a random port and uses that socket to communicate with B. If you want to mock B in your tests, you need that port number in order to connect to it and send messages to A.

But how can you get that number? When A creates B, it already runs in its own process, so a simple attribute access won't work. Setting a random seed would only work if you did that directly in A when it's already running. But doing that just for the tests is not such a good idea. It also may not work reliably on all systems and Python versions.

However, A must pass the socket number to B, so that B can connect to A. Thus, we can create a mock for B that will send us its port number via an inter-process queue from the *multiprocessing* module[16]:

```python
class ProcMock(mock.Mock):
    """
    This mock returns itself when called, so it acts like both, the
    process' class and instance object.
    """
    def __init__(self):
        super().__init__()
        self.queue = multiprocessing.Queue()

    def __call__(self, port):
        """Will be called when A instantiates B and passes its port number."""
        self.queue.put(port)
        return self

    def start(self):
        return  # Just make sure the methods exists and returns nothing

    def join(self):
        return  # Just make sure the methods exists and returns nothing


class TestA(ProcessTest):

    def setup_method(self):

        b_mock = ProcMock()
        with mock.patch('b.B', new=b_mock):
            self.a = A()
            self.a.start()

        # Get the port A is listening on
        port = b_mock.queue.get()

        # ...
```

As you've seen, process testing is really not as simple as unit testing. But I always found bugs with it that my unit tests coudn't detect. If you cover all communication sequences for a process in a process test, you can be pretty sure, that it will also work flawlessly in the final application.

3.3 System Testing

If your application consists of more than one process, you still need to test whether all processes work nicely together or not. This is something you cannot simulate reliably with a process tests, as much as unit tests can't replace the process test.

Writing a good system test is very application-specific and can be, depending on the complexity of your application, very hard or very easy. Fortunately, the latter is the case for our ping-pong app. We just start it and copy its output to a file. If the output is not what we expected, we modify the file accordingly. In our test, we can now simply invoke our programm again, capture its output and compare it to the contents of the file we created before:

```python
# test/system/test_pongproc.py
import os.path
import subprocess
import pytest

def test_pongproc():
    filename = os.path.join('test', 'data', 'pongproc.out')
```

[16]http://docs.python.org/py3k/library/multiprocessing#exchanging-objects-between-processes>

```
    expected = open(filename).read()

    output = subprocess.check_output(['python', 'pongproc.py'],
                                     universal_newlines=True)

    assert output == expected
```

If your application was a server, another way of doing the system test would be to emulate a client that speaks with it. Your system test would then be very similar to your process tests, except that you only mimic the client and not all processes your main process communicates with.

Other applications (like, for instance, simulations) might create a database containing collected data. Here, you might check if these results match your expectations.

Of course you can also combine these possibilities or do something completely different . . .

3.4 “My Test Are now Running sooo Slow!”

System and process tests often run much slower than simple unit tests, so you may want to skip them most of the time. Pytest allows you to mark a test with a given name[17]. You can then (de)select tests based on their mark when you invoke pytest.

To mark a module e.g. as `process` test, just put a line `pytestmark = pytest.mark.process` somewhere in it. Likewise, you can add a `pytestmark = pytest.mark.system` to mark a module as system test.

You can now deselect process and system tests:

```
$ py.test -m "not (process or system)"
```

You can put this into a *pytest.ini* as a default setting. To override this again, use `1` or `True` as selection expression:

```
$ py.test -m 1
```

4 Summary

I showed you three ways to use PyZMQ. If you have a very simple process with only one socket, you can easily use its blocking *recv* methods. If you need more than one socket, I recommend using the event loop. And polling . . . you don’t want to use that. In my opinion, it is not very pythonic and the code will get unreadable a lot faster than with an event loop.

If you decide to use PyZMQ’s event loop, you should separate the application logic from all the PyZMQ stuff (like creating streams, sending/receiving messages and dispatching them). If your application consists of more then one process (which is usually the case), you should also create a base class with shared functionality for them.

Testing is always very important and the more complex your system gets, the more crucial are tests—both, qualitatively and quantitatively. Unit tests (a lot of them) should be the backbone of your test suite. But since they can’t detect all kinds of errors, you also need to test larger units of your application as a whole. These Process and System Tests require a bit more effort, but I think they are definitely worth the extra work. They give you a lot more confidence that your program actually works, because they are much more realistic than unit tests can be.

I hope this article provided a good overview about how to design and test applications with PyZMQ and I hope that you now run into much less problems than I did when I first started working with PyZMQ.

[17] `http://pytest.org/latest/example/markers.html#mark-examples`

References

[1] iMatix Corporation, "ØMQ – the intelligent transport layer," 2012. [Online] http://www.zeromq.org

[2] ——, "ØMQ – the guide," 2011. [Online] http://zguide.zeromq.org/page:all#Fixing-the-World

[3] ——, "ØMQ python binding," 2012. [Online] http://www.zeromq.org/bindings:python

[4] N. Piël, "ZeroMQ an introduction," 2010. [Online] http://nichol.as/zeromq-an-introduction

[5] B. E. Granger and M. Ragan-Kelley, "Pyzmq documentation," 2011. [Online] http://zeromq.github.com/pyzmq

[6] S. Scherfke, "Examples for 'designing and testing pyzmq applications'," 2012. [Online] https://bitbucket.org/ssc/pyzmq-article

[7] Python Software Foundation, "multiprocessing — process-based parallelism," 2012. [Online] http://docs.python.org/py3k/library/multiprocessing#windows

[8] H. Krekel, "pytest," 2012. [Online] http://pytest.org/latest/

[9] M. Ford, "Mock – mocking and testing library," 2012. [Online] http://www.voidspace.org.uk/python/mock/