zeroless Documentation

Release

Author

August 10, 2016

Contents

This documentation contains notes on some important aspects of developing Zeroless and an overview of Zeroless' API. For information on how to use ØMQ in general, see the many examples in the excellent ØMQGuide. It can give a better understanding of when to use each messaging passing pattern available (i.e. request/reply, publisher/subscriber, push/pull and pair). Also, more complex use cases, that require the composition of these patterns, are explained in further details.

Zeroless works with Python 3 (3.0), and Python 2 (2.7), with no transformations or 2to3. Finally, please don't hesitate to report zeroless-specific issues to our Tracker on GitHub.

CHAPTER 1

Zeroless x PyZMQ

Differing from PyZMQ, which tries to stay very close to the C++ implementation, this project aims to make distributed systems employing ØMQ as pythonic as possible.

Being simpler to use, Zeroless doesn't supports all of the fine aspects and features of PyZMQ. However, you can expect to find all the message passing patterns you were accustomed to (i.e. pair, request/reply, publisher/subscriber, push/pull). Despite that, the only transport available is TCP, as threads are not as efficient in Python due to the GIL and IPC is unix-only.

CHAPTER 2

Installing

Install stable releases of Zeroless with `pip`.

```
$ pip install zeroless
```

See the Install Guide for more detail.

CHAPTER 3

Documentation

Contents:

3.1 Install Guide

Install stable releases of Zeroless with `pip`.

```
$ pip install zeroless
```

3.1.1 Dependencies

Zeroless only dependency is PyZMQ.

3.1.2 Installing from Github

The canonical repository for Zeroless is on GitHub.

```
$ git clone git@github.com:zmqless/python-zeroless.git
$ cd zeroless
$ python setup.py develop
```

The best reason to install from source is to help us develop Zeroless. See the Development section for more on that.

3.2 Quickstart

In the `zeroless` module, two classes can be used to define how distributed entities are related (i.e. `Server` and `Client`). To put it bluntly, with the exception of the pair pattern, a client may be connected to multiple servers, while a server may accept incoming connections from multiple clients.

Both servers and clients are able to create a *callable* and/or *iterable*, depending on the message passing pattern. So that you can iterate over incoming messages and/or call to transmit a message.

3.3 Message Passing Patterns

Zeroless supports the following message passing patterns:

3.3.1 Push-Pull

Useful for distributing the workload among a set of workers. A common pattern in the Stream Processing field, being the cornestone of applications like Apache Storm for instance. Also, it can be seen as a generalisation of the Map-Reduce pattern.

```
import logging

from zeroless import (Server, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Binds the pull server to port 12345
# And assigns an iterable to wait for incoming messages
listen_for_push = Server(port=12345).pull()

for msg in listen_for_push:
    print(msg)
```

```
import logging

from zeroless import (Client, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Connects the client to as many servers as desired
client = Client()
client.connect_local(port=12345)

# Initiate a push client
# And assigns a callable to push messages
push = client.push()

for msg in [b"Msg1", b"Msg2", b"Msg3"]:
    push(msg)
```

3.3.2 Publisher-Subscriber

Useful for broadcasting messages to a set of peers. A common pattern for allowing real-time notifications at the client side, without having to resort to inneficient approaches like pooling. Online services like PubNub or IoT protocols like MQTT are examples of this pattern usage.

```
import logging

from zeroless import (Client, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)
```

```
# Connects the client to as many servers as desired
client = Client()
client.connect_local(port=12345)

# Initiate a subscriber client
# Assigns an iterable to wait for incoming messages with the topic 'sh'
listen_for_pub = client.sub(topics=[b'sh'])

for topic, msg in listen_for_pub:
    print(topic, ' - ', msg)
```

```
import logging

from time import sleep

from zeroless import (Server, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Binds the publisher server to port 12345
# And assigns a callable to publish messages with the topic 'sh'
pub = Server(port=12345).pub(topic=b'sh', embed_topic=True)

# Gives publisher some time to get initial subscriptions
sleep(1)

for msg in [b"Msg1", b"Msg2", b"Msg3"]:
    pub(msg)
```

Note: ZMQ's topic filtering capabilities are publisher side since ZMQ 3.0.

Last but not least, SUB sockets that bind will not get any message before they first ask for via the provided generator, so prefer to bind PUB sockets if missing some messages is not an option.

3.3.3 Request-Reply

Useful for RPC style calls. A common pattern for clients to request data and receive a response associated with the request. The HTTP protocol is well-known for adopting this pattern, being it essential for Restful services.

```
import logging

from zeroless import (Server, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Binds the reply server to port 12345
# And assigns a callable and an iterable
# To both transmit and wait for incoming messages
reply, listen_for_request = Server(port=12345).reply()

for msg in listen_for_request:
```

```
    print(msg)
    reply(msg)
```

```
import logging

from zeroless import (Client, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Connects the client to as many servers as desired
client = Client()
client.connect_local(port=12345)

# Initiate a request client
# And assigns a callable and an iterable
# To both transmit and wait for incoming messages
request, listen_for_reply = client.request()

for msg in [b"Msg1", b"Msg2", b"Msg3"]:
    request(msg)
    response = next(listen_for_reply)
    print(response)
```

3.3.4 Pair

More often than not, this pattern will be unnecessary, as the above ones or the mix of them suffices most use cases in distributed computing. Regarding its capabilities, this pattern is the most similar alternative to usual posix sockets among the aforementioned patterns. Therefore, expect one-to-one and bidirectional communication.

```
import logging

from zeroless import (Server, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Binds the pair server to port 12345
# And assigns a callable and an iterable
# To both transmit and wait for incoming messages
pair, listen_for_pair = Server(port=12345).pair()

for msg in listen_for_pair:
    print(msg)
    pair(msg)
```

```
import logging

from zeroless import (Client, log)

# Setup console logging
consoleHandler = logging.StreamHandler()
```

```
log.setLevel(logging.DEBUG)
log.addHandler(consoleHandler)

# Connects the client to a single server
client = Client()
client.connect_local(port=12345)

# Initiate a pair client
# And assigns a callable and an iterable
# To both transmit and wait for incoming messages
pair, listen_for_pair = client.pair()

for msg in [b"Msg1", b"Msg2", b"Msg3"]:
    pair(msg)
    response = next(listen_for_pair)
    print(response)
```

3.4 Additional Features

3.4.1 Logging

Python provides a wonderfull `logging` module. It can be used to track Zeroless' internal workflow in a modular way, therefore being very useful for debugging purposes.

The `zeroless` module allows logging via a global Logger object.

```
from zeroless import log
```

To enable it, just add an Handler object and set an appropriate logging level.

3.4.2 Multipart Messages

In the Zeroless API, all *callables* have a `print` like signature, therefore being able to have an infinite number of arguments. Each of these arguments are part of the whole message, that could be divided in multiple pieces. Being that useful when you have a simple message structure, with just a few fields, and don't want to rely on a data formatting standard (e.g. JSoN, XML) to maintain the message semantics. Also, given the need to parse those different parts that a single message may have, the receiver's *iterable* will return them all, at once, in transparent fashion.

For more on this, see the examples/multipart folder or check the following example:

```
from zeroless import Server

# Binds the pull server to port 12345
# And assigns an iterable to wait for incoming messages
listen_for_push = Server(port=12345).pull()

for id, msg in listen_for_push:
    print(id, ' - ', msg)
```

```
from zeroless import Client

# Connects the client to as many servers as desired
client = Client()
client.connect_local(port=12345)
```

```
# Initiate a push client
# And assigns a callable to push messages
push = client.push()

for id, msg in [(b"1", b"Msg1"), (b"2", b"Msg2"), (b"3", b"Msg3")]:
    push(id, msg)
```

3.5 Development

This page describes Zeroless development process and contains general guidelines and information on how to contribute to the project.

3.5.1 Contributing

We welcome contributions of any kind (ideas, code, tests, documentation, examples, ...).

General Contribution Guidelines

- Any non-trivial change must contain tests.
- All the functions and methods must contain Sphinx docstrings which are used to generate the API documentation.
- If you are adding a new feature, make sure to add a corresponding documentation.

Code Style Guide

- We follow PEP8 Python Style Guide.
- Use 4 spaces for a tab.
- Use 79 characters in a line.
- Make sure edited file doesn't contain any trailing whitespace.

3.5.2 Testing

Tests make use of the `py.test` framework and are located in the tests/ folder. However, we recommend the usage of `tox` as it will test our codebase against both Python 2.7 and 3.0.

To run individual tests:

```
$ py.test tests/test_desired_module.py
```

To run all the tests:

```
$ python setup.py test
```

Alternatively, you can use tox:

```
$ tox
```

All functionality (including new features and bug fixes) must include a test case to check that it works as expected, so please include tests for your patches if you want them to get accepted sooner.

CHAPTER 4

Zeroless API

Contents:

4.1 zeroless package

4.1.1 Submodules

4.1.2 zeroless.zeroless module

The Zeroless module API.

zeroless.zeroless.**log**

A global Logger object. To use it, just add an Handler object and set an appropriate logging level.

class zeroless.zeroless.**Client**
Bases: *zeroless.zeroless.Sock*

A client that can connect to a set of servers.

addresses
Returns a tuple containing all the connected addresses. Each address is a tuple with an ip address and a port.

Return type (addresses)

connect(*ip*, *port*)
Connects to a server at the specified ip and port.

Parameters

- **ip** (*str or unicode*) – an IP address
- **port** (*int*) – port number from 1024 up to 65535

Return type self

connect_local(*port*)
Connects to a server in localhost at the specified port.

Parameters **port** (*int*) – port number from 1024 up to 65535

Return type self

disconnect(*ip*, *port*)
Disconnects from a server at the specified ip and port.

Parameters

- **ip** (*str or unicode*) – an IP address
- **port** (*int*) – port number from 1024 up to 65535

Return type self

disconnect_all()
Disconnects from all connected servers. :rtype: self

disconnect_local(*port*)
Disconnects from a server in localhost at the specified port.

Parameters **port** (*int*) – port number from 1024 up to 65535

Return type self

class zeroless.zeroless.**Server**(*port*)
Bases: *zeroless.zeroless.Sock*

A server that clients can connect to.

port
Returns the port.

Return type int

class zeroless.zeroless.**Sock**
Bases: object

pair()
Returns a callable and an iterable respectively. Those can be used to both transmit a message and/or iterate over incoming messages, that were sent by a pair socket. Note that the iterable returns as many parts as sent by a pair. Also, the sender function has a `print` like signature, with an infinite number of arguments. Each one being a part of the complete message.

Return type (function, generator)

pub(*topic=b''*, *embed_topic=False*)
Returns a callable that can be used to transmit a message, with a given `topic`, in a publisher-subscriber fashion. Note that the sender function has a `print` like signature, with an infinite number of arguments. Each one being a part of the complete message.

By default, no topic will be included into published messages. Being up to developers to include the topic, at the beginning of the first part (i.e. frame) of every published message, so that subscribers are able to receive them. For a different behaviour, check the embed_topic argument.

Parameters

- **topic** (*bytes*) – the topic that will be published to (default=b'')
- **embed_topic** – set for the topic to be automatically sent as the first part (i.e. frame) of every published message (default=False)

:type embed_topic bool :rtype: function

pull()
Returns an iterable that can be used to iterate over incoming messages, that were pushed by a push socket. Note that the iterable returns as many parts as sent by pushers.

Return type generator

push()
Returns a callable that can be used to transmit a message in a push-pull fashion. Note that the sender

function has a `print` like signature, with an infinite number of arguments. Each one being a part of the complete message.

Return type function

reply()
Returns a callable and an iterable respectively. Those can be used to both transmit a message and/or iterate over incoming messages, that were requested by a request socket. Note that the iterable returns as many parts as sent by requesters. Also, the sender function has a `print` like signature, with an infinite number of arguments. Each one being a part of the complete message.

Return type (function, generator)

request()
Returns a callable and an iterable respectively. Those can be used to both transmit a message and/or iterate over incoming messages, that were replied by a reply socket. Note that the iterable returns as many parts as sent by repliers. Also, the sender function has a `print` like signature, with an infinite number of arguments. Each one being a part of the complete message.

Return type (function, generator)

sub(*topics=(b'',)*)
Returns an iterable that can be used to iterate over incoming messages, that were published with one of the topics specified in `topics`. Note that the iterable returns as many parts as sent by subscribed publishers.

Parameters **topics** (*list of bytes*) – a list of topics to subscribe to (default=b'')

Return type generator

4.1.3 Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Development

We welcome contributions of any kind (ideas, code, tests, documentation, examples, ...). See the Development section for further details.

CHAPTER 7

Links

- ØMQ Home
- The ØMQGuide
- Zeroless on GitHub
- Zeroless on PyPy
- Issue Tracker

CHAPTER 8

License

Z

A

C

D

L

P

R

S

Z