Pizco Documentation

Release 0.1

Hernan E. Grecco

Nov 02, 2017

Contents

Pizco is Python module/package that allows python objects to communicate. Objects can be exposed to other process in the same computer or over the network, allowing clear separation of concerns, resources and permissions.

Pizco supports calling methods from remote objects and also accessing their attributes, dictionary attributes and properties. Most importantly, using a Qt-like (and Qt compatible!) signal and slot mechanism you can easily register notifications.

As ZMQ is used as the transport layer, communication is fast and efficient, and different protocols are supported. It has a complete test coverage. It runs in Python 3.2+ and requires PyZMQ. It is licensed under BSD.

CHAPTER 1

Design principles

- Reusable Agent class as communicating object for both Proxy and Server.
- ZMQ REP/REQ to handle sync access to objects.
- ZMQ PUB/SUB for notifications and async operations.
- PyQt-like signal and slots callbacks, compatible with PyQt.
- Transparent handling of methods that return concurrent.Futures.
- *Soon*: Asynchronous and batched operation on remote objects.
- Small codebase: small and easy to maintain codebase with a flat hierarchy. It is a single stand-alone module that can be installed as a package or added side by side to your project.
- *Soon*: Python 2 and 3: A single codebase that runs unchanged in Python 2.6+ and Python 3.0+.

CHAPTER 2

Pizco in action

example.py:

```
from PyQt4 import pyqtSignal as Signal

class MultiplyBy(object):

    factor_changed = Signal()

    def __init__(self, factor):
        self._factor = factor

    def calculate(self, x):
        return x * self.factor

    @property
    def factor(self):
        return self._factor

    @factor.setter
    def factor(self, value):
        if self._factor == value:
            continue
        self.factor_changed.emit(value, self._factor)
        self._factor = value
```

server.py:

```
from example import MultiplyBy

from pizco import Server

server = Server(MultiplyBy(2), 'tcp://127.0.0.1:8000')
server.serve_forever()
```

client.py:

```
import time

from pizco import Proxy

proxy = Proxy('tcp://127.0.0.1:8000')

print('{} * {} = {}'.format(proxy.factor, 8, proxy.calculate(8)))

def on_factor_changed(new_value, old_value):
    print('The factor was changed from {} to {}'.format(old_value, new_value))
    print('{} * {} = {}'.format(proxy.factor, 8, proxy.calculate(8)))

proxy.factor_changed.connect(on_factor_changed)

for n in (3, 4, 5):
    proxy.factor = n
    time.sleep(.5)
```

Start the server in a terminal and run the client in another one:

```
$ python client.py
2 * 8 = 16
The factor was changed from 2 to 3
3 * 8 = 24
The factor was changed from 3 to 4
4 * 8 = 32
The factor was changed from 4 to 5
5 * 8 = 40
```

CHAPTER 3

Contents

3.1 Getting Started

3.1.1 Installing

To install *Pizco* you need to to install PyZMQ following the instructions here. If you are having trouble at this step, a few Python distributions like EPD and Anaconda have PyZMQ preinstalled.

Then install, *Pizco* using pip or easy_install:

```
$ easy_install pizco
```

or:

```
$ pip install pizco
```

3.1.2 Basic usage

Consider the:

```
from myproject import Robot

robot = Robot()

print(robot.name)
robot.move_arm()
robot.age = 28
```

Pizco provides two classes t:

- Server: wraps an object and exposes its attributes via a ZMQ socket.
- Proxy: connects to a server, redirects attribute request to it, and collect the response.

Creating a Server is quite simple, just instantiate a Server using the object as the first parameter:

```
# This is your stuff
from myproject import Robot

from pizco import Server

server = Server(Robot())
```

If no endpoint is given, the server will bind to a random tcp port.

You can specify the endpoint with the second argument:

```
server = Server(Robot(), 'tcp://127.0.0.1:8000')
```

Any valid ZMQ endpoints is valid:

- **inproc**: local in-process (inter-thread) communication transport

  example: inproc://robbie-the-robot

- **ipc**: local inter-process communication transport

  example: ipc://robbie-the-robot

- **tcp**: unicast transport using TCP

  example: tcp://127.0.0.1:8000

In the client side, you need to create a proxy:

```
from pizco import Proxy

robot = Proxy('tcp://127.0.0.1:8000')
```

and now you can use the proxy as if it was the actual object:

```
print(robot.name)
robot.move_arm()
robot.age = 28
```

Notice that the only needed change was to the initialization code.

3.1.3 Remote exceptions

Exception in the served object are caught remotely and re-raised by the proxy and therefore the following code:

```
try:
    robot.age = input()
except ValueError as ex:
    print('That is not a valid age for the robot')
```

will work the same way if the robot is the actual object or just a Proxy to it.

Note: The remote traceback is not propagated the proxy in the current version of *Pizco*.

3.2 Futures

Starting from Python 3.2, `concurrent.futures` provides a high-level interface for asynchronously executing callables (for older versions there is a Futures Backport)

The `concurrent.futures.Future` class encapsulates the asynchronous execution of a callable and *Pizco* has in-built support for it. Instead of returning a *Future*, the Server will store it and notify the Proxy. The proxy then returns a client-side *Future* connected to the original server-side *Future*. The usage is just like normal *Futures* (there are just that after all!):

```
from myproject import Robot

from pizco import Proxy

robot = Proxy('tcp://127.0.0.1:8000')

# Move is a method from robot that returns a Future
fut = robot.think('What do you get if you multiply six by nine?')

# Do something here

# fut is an instance of `Future`, therefore you can use the
# methods and properties described in the Python docs
answer = fut.result()
```

Note: Future.cancel() is not currently implemented.

3.3 Signals and Slots (registering callbacks)

If a served object exposes a Qt signal (or Qt-like signals), you can connect a slot to it in the proxy. If you are not familiar with the Qt jargon, this is equivalent to bind (*connect*) a callback (*slot*) to an event (*signal*).

The syntax in the client side is exactly as it is done in PyQt/PySide New Style Signals and Slots:

```
def print_notification(new_position):
    print('The arm has been moved to {}'.format(new_position))

proxy = Proxy('tcp://127.0.0.1:8000')

proxy.arm_moved.connect(print_notification)
```

Under the hood, the proxy is subscribing to an event in the server using a ZMQ SUB socket. When the *arm_moved* signal is emitted server-side, the server will **asynchronously** notify the proxy using a PUB socket. The proxy will call *print_notification* when it receives message. So you have client side notification of server side events.

It is important to note that if you are using PyQt/PySide signals in your code, no modification is needed to use *Pizco*. But not only PyQt/PySide signals work. An attribute is considered *Signal* if it exposes at least three methods: *connect*, *disconnect* and *emit*.

Just like PyQt, multiple slots can be connected to the same signal:

```
proxy.arm_moved.connect(print_notification)
proxy.arm_moved.connect(mail_notification)
```

and they will be called in the order that they were added.

To disconnect from a signal:

```
proxy.arm_moved.disconnect(mail_notification)
```

Additionally, you can connect in another proxy:

```
proxy2 = Proxy('tcp://127.0.0.1:8000')

proxy2.arm_moved.connect(mail_notification)
```

Proxy connections to signals are independent and therefore disconnecting in one proxy will not have an effect on another.

3.4 Serve in process

You can directly start the server from the client application:

```
from pizco import Server

from myproject import Robot

# The class (not the object!) is given as the first argument
# and it returns a proxy to the served object.
robot_proxy = Server.serve_in_process(Robot)

robot_proxy.move_arm()

print(robot_proxy.age)
```

The server is started in a new instance of the same python interpreter. *Pizco* will provide this new process with the path of the *Robot* class but you need to be sure that any other dependency is available.

If the *Robot* constructor takes some arguments you can give them like this:

```
robot_proxy = Server.serve_in_process(Robot,
                                      args=('Robbie', ),
                                      kwargs={'age': 3})
```

Finally, you can ask *Pizco* to show the running server with a pop-up window:

```
robot_proxy = Server.serve_in_process(Robot,
                                      args=('Robbie', ),
                                      kwargs={'age': 3},
                                      gui=True)
```

3.5 Command line tool

You can start a server from the command line calling *pizco.py*. For example:

```
$ python pizco.py tcp://127.0.0.1:8000
Server started at tcp://127.0.0.1:8000
Press CTRL+c to stop ...
```

will start a server bound to localhost, port 8000. If you want to bind to a particular pub endpoint, you can specify it with an extra parameter.

When the server is created in this way, no object is served. To instantiate and serve an object, create a proxy, connect to it and call the instantiate method:

```
from myproject import Robot

proxy = Proxy('tcp://127.0.0.1:8000')
proxy._proxy_agent.instantiate(Robot,
                               args=('Robbie', ),
                               kwargs={'age': 3})
```

Additional arguments can be used to configure the server:

-g: open a small window to display the server status. If the window is closed, the server is stopped.

-v: print debug information to the console.

-p *path*: add *path* to sys.path

This is script is called under the hood by *serve_in_process* to initiated a server in detached processes.

3.6 Internals

3.6.1 Agent

The base of both `Proxy` and `Server` is the `Agent`. Each Agent has ZMQ sockets to communicate:

- a *REP* socket to receive requests. This is the main endpoint of the Agent.
- a *PUB* to emit notifications to other Agents.
- one *SUB* to subscribe to notifications from other Agents.
- one *REQ* per each Agent that it has to talk to (stored in self.connections)

The *REP* and *PUB* endpoint can be specified when the Agent is instantiated. If no endpoint is given, the sockets will bind to a random tcp port.

3.6.2 Protocol

Messages are multipart ZMQ messages. The conten

FRAME 0: Header (utf-8 encoded str)

Used for identification and filtering. It contains 3 string concatenated with a + (plus sign).

1. The protocol version (currently PZC00).
2. A unique identifier for the sender.
3. A string specifying the topic of the message.

example: PZC00+urn:uuid:ad2d9eb0-c5f8-4bfb-a37d-6b7903b041f3+value_changed

FRAME 1: Serialization (utf-8 encoded str)

Indicates the serialization protocol used in FRAME 2. Current valid values are:

- 'pickle': use the highest version available of the pickle format (default).

- 'pickleN': use the N version of the pickle format.
- 'json': use json format.

FRAME 2: Content (binary blob)

The actual content of the message.

FRAME 3: Message ID (utf-8 encoded str)

A unique identifier for the message.

example: urn:uuid:b711f2b8-277d-40df-a283-6269331db251

FRAME 4: Signature (bytes)

HMAC sha1 signature of FRAME 0:4 concatenated with Agent.hmac_key

By default, Agents use an empty signature key (no signature) and the *pickle* serializer. The simplest way to change these defaults is by using the environmental variables *PZC_KEY* and *PZC_SER*. You might want to change the serializer when running different agents on different versions of Python as not all pickle versions are supported in all python versions.

You can also change the protocol settings for an specific Agent by passing a Protocol object when the Agent is created.

3.6.3 Proxy-Server Protocol

Between the Proxy and the Server, the content of the message (Frame 2) is a tuple of three elements, being the first utf-8 str defining the subprotocol ('PSMessage'). The second and third elements specify the action and the options for that action.

3.7 API

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P