pycolo Documentation

Release 0.0.1

Remy Leone

May 01, 2013

CONTENTS

CHAPTER

ONE

USER GUIDE

This part of the documentation, which is mostly prose, begins with some background information about Pycolo, then focuses on step-by-step instructions for getting the most out of it.

1.1 Introduction

1.1.1 Philosophy

Pycolo was developed with a few **PEP 20** idioms in mind.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Readability counts.

All contributions to Pycolo should keep these important rules in mind.

1.1.2 Pycolo License

1.2 Installation

This part of the documentation covers the installation of Pycolo. The first step to using any software package is getting it properly installed.

1.2.1 Distribute & Pip

Installing pycolo is simple with pip:

```
$ pip install pycolo
```

or, with easy_install:

```
$ easy_install pycolo
```

But, you really shouldn't do that.

1.2.2 Get the Code

Pycolo is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
git clone git://github.com/sieben/pycolo.git
```

Download the tarball:

```
$ curl -OL https://github.com/sieben/pycolo/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/sieben/pycolo/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

1.3 Quickstart

Eager to get started? This page gives a good introduction in how to get started with Pycolo. This assumes you already have Pycolo installed. If you do not, head over to the *Installation* section.

First, make sure that:

- Pycolo is *installed*
- Pycolo is *up-to-date*

Let's get started with some simple examples.

1.3.1 Make a Request

Making a request with pycolo is very simple.

Begin by importing the pycolo module:

```
>>> import pycolo
```

Now, let's try to get a coap/.well-known. For this example, let's get GitHub's public timeline

```
>>> r = pycolo.get('coap://coap.sieben.fr/.well-known')
```

Now, we have a `Response` object called `r`. We can get all the information we need from this object.

Pycolo simple API means that all forms of CoAP request are as obvious. For example, this is how you make an CoAP POST request:

```
>>> r = pycolo.post("coap://coap.sieben.fr/post")
```

Nice, right? What about the other CoAP request types: PUT, DELETE, HEAD and OPTIONS? These are all just as simple:

```
>>> r = pycolo.put("coap://coap.sieben.fr/put")
>>> r = pycolo.delete("coap://coap.sieben.fr/delete")
>>> r = pycolo.head("coap://coap.sieben.fr/.well-known")
>>> r = pycolo.options("coap://coap.sieben.fr/.well-known")
```

That's all well and good, but it's also only the start of what Pycolo can do.

1.3.2 Passing Parameters In URLs

You often want to send some sort of data in the URL's query string. If you were constructing the URL by hand, this data would be given as key/value pairs in the URL after a question mark, e.g. `coap.sieben.fr/.well-known?key=val`. Pycolo allows you to provide these arguments as a dictionary, using the `params` keyword argument. As an example, if you wanted to pass `key1=value1` and `key2=value2` to `coap.sieben.fr/.well-known`, you would use the following code:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = pycolo.get("coap://coap.sieben.fr/.well-known", params=payload)
```

You can see that the URL has been correctly encoded by printing the URL:

```
>>> print r.url
u'coap://coap.sieben.fr/.well-known?key2=value2&key1=value1'
```

1.3.3 Response Content

We can read the content of the server's response.:

```
>>> import pycolo
>>> r = pycolo.get('coap://coap.sieben.fr/.well-known')
>>> r.text
'[{"resources":{"temp":42,"url":"coap://coap.sieben.fr/...
```

Pycolo will automatically decode content from the server. Most unicode charsets are seamlessly decoded.

When you make a request, Pycolo makes educated guesses about the encoding of the response based on the CoAP headers. The text encoding guessed by Pycolo is used when you access `r.text`. You can find out what encoding Pycolo is using, and change it, using the `r.encoding` property:

```
>>> r.encoding
'utf-8'
>>> r.encoding = 'ISO-8859-1'
```

If you change the encoding, Pycolo will use the new value of `r.encoding` whenever you call `r.text`.

Pycolo will also use custom encodings in the event that you need them. If you have created your own encoding and registered it with the `codecs` module, you can simply use the codec name as the value of `r.encoding` and Pycolo will handle the decoding for you.

1.3.4 Binary Response Content

You can also access the response body as bytes, for non-text requests:

```
>>> r.content
b'[{"resources":{"temp":42,"url":"coap://coap.sieben.fr/...
```

The `gzip` and `deflate` transfer-encodings are automatically decoded for you.

For example, to create an image from binary data returned by a request, you can use the following code:

```
>>> from PIL import Image
>>> from StringIO import StringIO
>>> i = Image.open(StringIO(r.content))
```

1.3.5 JSON Response Content

There's also a builtin JSON decoder, in case you're dealing with JSON data:

```
>>> import pycolo
>>> r = pycolo.get('coap://coap.sieben.fr/.well-known.json')
>>> r.json
[{u'repository': {u'open_issues': 0, u'url': 'coap://coap.sieben.fr/...
```

In case the JSON decoding fails, `r.json` simply returns `None`.

1.3.6 Raw Response Content

In the rare case that you'd like to get the absolute raw socket response from the server, you can access `r.raw`:

```
>>> r.raw.read(10)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'
```

1.3.7 Custom Headers

If you'd like to add CoAP headers to a request, simply pass in a `dict` to the `headers` parameter.

For example, we didn't specify our content-type in the previous example:

```
>>> import json
>>> url = 'coap://coap.sieben.fr/some/endpoint'
>>> payload = {'some': 'data'}
>>> headers = {'content-type': 'application/json'}

>>> r = pycolo.post(url, data=json.dumps(payload), headers=headers)
```

1.3.8 More complicated POST requests

Typically, you want to send some form-encoded data — much like an HTML form. To do this, simply pass a dictionary to the *data* argument. Your dictionary of data will automatically be form-encoded when the request is made:

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = pycolo.post("coap://coap.sieben.fr/post", data=payload)
>>> print r.text
{
  // ...snip... //
  "form": {
    "key2": "value2",
```

```
    "key1": "value1"
  },
  // ...snip... //
}
```

There are many times that you want to send data that is not form-encoded. If you pass in a `string` instead of a `dict`, that data will be posted directly.

For example, the GitHub API v3 accepts JSON-Encoded POST/PATCH data:

```
>>> import json
>>> url = 'coap://coap.sieben.fr/some/endpoint'
>>> payload = {'some': 'data'}

>>> r = pycolo.post(url, data=json.dumps(payload))
```

1.3.9 POST a Multipart-Encoded File

Pycolo makes it simple to upload Multipart-encoded files:

```
>>> url = 'coap://coap.sieben.fr/post'
>>> files = {'file': open('report.csv', 'rb')}

>>> r = pycolo.post(url, files=files)
>>> r.text
{
  // ...snip... //
  "files": {
    "file": "<censored...binary...data>"
  },
  // ...snip... //
}
```

You can set the filename explicitly:

```
>>> url = 'coap://coap.sieben.fr/post'
>>> files = {'file': ('report.csv', open('report.csv', 'rb'))}

>>> r = pycolo.post(url, files=files)
>>> r.text
{
  // ...snip... //
  "files": {
    "file": "<censored...binary...data>"
  },
  // ...snip... //
}
```

If you want, you can send strings to be received as files:

```
>>> url = 'coap://coap.sieben.fr/post'
>>> files = {'file': ('report.csv', 'some,data,to,send\nanother,row,to,send\n')}

>>> r = pycolo.post(url, files=files)
>>> r.text
{
  // ...snip... //
  "files": {
```

```
    "file": "some,data,to,send\\nanother,row,to,send\\n"
  },
  // ...snip... //
}
```

1.3.10 Response Status Codes

We can check the response status code:

```
>>> r = pycolo.get('coap://coap.sieben.fr/.well-known')
>>> r.status_code
200
```

Pycolo also comes with a built-in status code lookup object for easy reference:

```
>>> r.status_code == pycolo.codes.ok
True
```

If we made a bad request (non-200 response), we can raise it with `Response.raise_for_status()`:

```
>>> bad_r = pycolo.get('coap://coap.sieben.fr/status/404')
>>> bad_r.status_code
404

>>> bad_r.raise_for_status()
Traceback (most recent call last):
    raise coap_error
pycolo.exceptions.COAPError: 404 Client Error
```

But, since our `status_code` for `r` was `200`, when we call `raise_for_status()` we get:

```
>>> r.raise_for_status()
None
```

All is well.

1.3.11 Response Headers

We can view the server's response headers using a Python dictionary:

```
>>> r.headers
{
    'status': '200 OK',
    'content-encoding': 'text',
    'transfer-encoding': 'chunked',
    'connection': 'close',
    'server': 'contiki/Erbium',
    'x-runtime': '148ms',
    'etag': '"e1ca502697e5c9317743dc078f67693f"',
    'content-type': 'application/json; charset=utf-8'
}
```

The dictionary is special, though: it's made just for CoAP headers, CoAP headers are case-insensitive.

So, we can access the headers using any capitalization we want:

```
>>> r.headers['Content-Type']
'application/json; charset=utf-8'

>>> r.headers.get('content-type')
'application/json; charset=utf-8'
```

If a header doesn't exist in the Response, its value defaults to `None`:

```
>>> r.headers['X-Random']
None
```

1.3.12 Timeouts

You can tell pycolo to stop waiting for a response after a given number of seconds with the `timeout` parameter:

```
>>> pycolo.get('coap://coap.sieben.fr/.well-known', timeout=0.001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pycolo.exceptions.Timeout: Request timed out.
```

Note:

`timeout` only effects the connection process itself, not the downloading of the response body.

CHAPTER

TWO

API DOCUMENTATION

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API

CoAP Protocol constants

DEFAULT_PORT default CoAP port as defined in draft-ietf-core-coap-05, section 7.1: MUST be supported by a server for resource discovery and SHOULD be supported for providing access to other resources.

URI_SCHEME_NAME CoAP URI scheme name as defined in draft-ietf-core-coap-05, section 11.4.

MAX_RETRANSMIT maximal number of retransmissions before the attempt to transmit a message is canceled

OBSERVING_REFRESH_INTERVAL the number of notifications until a CON notification will be used

DEFAULT_BLOCK_SIZE the default block size for block-wise transfers must be power of two between 16 and 1024 bytes.

MESSAGE_CACHE_SIZE capacity (in messages) for caches. Used for duplicate detection and retransmissions.

RX_BUFFER_SIZE buffer size for incoming datagrams, in bytes

DEFAULT_OVERALL_TIMEOUT time (in milliseconds) for transaction to complete. Used to avoid infinite waits for replies to non-confirmables and separate responses

RESPONSE_TIMEOUT & RESPONSE_RANDOM_FACTOR constants to calculate initial timeout for confirmable messages, used by the exponential backoff mechanism

TODO Find a better value for RX_BUFFER_SIZE

This part of the documentation covers all the interfaces of Pycolo. For parts where Pycolo depends on external libraries, we document the most important right here and provide links to the canonical documentation.

2.1.1 Main Interface

All of Request's functionality can be accessed by these 7 methods. They all return an instance of the `Response` object.

2.1.2 Utilities

These functions are used internally, but may be useful outside of Requests.

Status Code Lookup

Registry of all constant status code

codes describes the CoAP Code Registry

mediaCodes describes the CoAP Media Type Registry

options describes the CoAP Option Number Registry

2.1.3 Internals

These items are an internal component to Pycolo, and should never be seen by the end user (developer). This part of the API documentation exists for those who are extending the functionality of Pycolo.

2.2 Pycolo

2.2.1 `pycolo` Package

CoAP Protocol constants

DEFAULT_PORT default CoAP port as defined in draft-ietf-core-coap-05, section 7.1: MUST be supported by a server for resource discovery and SHOULD be supported for providing access to other resources.

URI_SCHEME_NAME CoAP URI scheme name as defined in draft-ietf-core-coap-05, section 11.4.

MAX_RETRANSMIT maximal number of retransmissions before the attempt to transmit a message is canceled

OBSERVING_REFRESH_INTERVAL the number of notifications until a CON notification will be used

DEFAULT_BLOCK_SIZE the default block size for block-wise transfers must be power of two between 16 and 1024 bytes.

MESSAGE_CACHE_SIZE capacity (in messages) for caches. Used for duplicate detection and retransmissions.

RX_BUFFER_SIZE buffer size for incoming datagrams, in bytes

DEFAULT_OVERALL_TIMEOUT time (in milliseconds) for transaction to complete. Used to avoid infinite waits for replies to non-confirmables and separate responses

RESPONSE_TIMEOUT & RESPONSE_RANDOM_FACTOR constants to calculate initial timeout for confirmable messages, used by the exponential backoff mechanism

TODO Find a better value for RX_BUFFER_SIZE

2.2.2 codes Module

Registry of all constant status code

codes describes the CoAP Code Registry

mediaCodes describes the CoAP Media Type Registry

options describes the CoAP Option Number Registry

pycolo.codes.**isElective**(*optionNumber*)
Checks whether a code indicates an elective option number.

Parameters **optionNumber** – Code to test

Returns True if option number is a valid elective number, False otherwise.

pycolo.codes.**isRequest**(*code*)
Checks whether a code indicates a request.

Parameters **code** – code the code to check

Returns True if the code indicates a request

pycolo.codes.**isResponse**(*code*)
Checks whether a code indicates a response number.

Parameters **code** – Code to test

Returns True if option number is a valid response number, False otherwise.

pycolo.codes.**isValid**(*code*)
Checks whether a code indicates a valid.

Parameters **code** – Code to test.

Returns True if option number is valid, False otherwise.

pycolo.codes.**opaque_256_many**(*min_size*, *max_size*, *default=None*)
TODO

Parameters

- **min_size** –
- **max_size** –
- **default** –

Returns

pycolo.codes.**presence_once**()
TODO

Returns

pycolo.codes.**responseClass**(*code*)
Returns the response class of a code

Parameters **code** – the code to check

Returns The response class of the code

pycolo.codes.**string_many**(*min_size*, *max_size*, *default=None*)
Repeatable string option.

Parameters

- **min_size** –
- **max_size** –
- **default** –

Returns

pycolo.codes.**string_once**(*min_size*, *max_size*, *default=None*)
TODO

Parameters

- **min_size** –
- **max_size** –
- **default** –

Returns

pycolo.codes.**uint_many**(*min_size*, *max_size*, *default=None*)
TODO

Parameters

- **min_size** –
- **max_size** –
- **default** –

Returns

pycolo.codes.**uint_once**(*min_size*, *max_size*, *default=None*)
TODO

Parameters

- **min_size** –
- **max_size** –
- **default** –

Returns

2.2.3 endpoint Module

2.2.4 layers Module

2.2.5 message Module

2.2.6 observe Module

The TokenManager stores all tokens currently used in transfers. New transfers can acquire unique tokens from the manager.

pycolo.observe.**addObserver**(*request*, *resource*)
get clients map for the given resource path :param request: :param resource:

pycolo.observe.**isObserved**(*uri*)

pycolo.observe.**notifyObservers**(*resource*)

pycolo.observe.**prepareResponse**(*request*)
consecutive response require new MID that must be stored for RST matching :param request:

pycolo.observe.**removeObserver**(*client*, *resource=None*, *mid=None*)
Remove a selected observer from observation structures. Remove an observer by MID from RST. :param mid: the MID from the RST :param resource: the resource to un-observe. :param client: the peer address as string.

pycolo.observe.**updateLastMID**(*clientID*, *path*, *mid*)

2.2.7 request Module

2.2.8 resource Module

pycolo.structures

Pycolo resource.

class pycolo.resource.**Resource**(*title*, *resourceIdentifier=None*, *interfaceDescription=None*, *link_format=''*, *hidden=False*, *observable=False*, *parent=None*, *resourceType=None*, *contentType=None*)

Parameters

- **title** –
- **resourceIdentifier** –
- **link_format** –
- **hidden** –
- **observable** –

Raise

attributes = {'resourceType': 'rt', 'interfaceDescription': 'if', 'contentType': 'ct', 'sizeEstimate': 'sz', 'title': 'title'}

changed()
Send a notification to all the subscribed resource. :return:

count(*recursive=False*)
Counting sub resources. :param recursive: :raise:

getPath()
Returns the full resource path.

subResources = {}

toLink()
Serialize to a link format definitions as specified in draft-ietf-core-link-format-06 :param self:

2.2.9 token Module

The TokenManager stores all tokens currently used in transfers. New transfers can acquire unique tokens from the manager.

pycolo.token.**acquireToken**(*preferEmptyToken=False*)
Returns an unique token.

Parameters **preferEmptyToken** – If set to true, the caller will receive the empty token if it is available.

This is useful for reducing datagram sizes in transactions that are expected to complete in short time. On the other hand, empty tokens are not preferred in block - wise transfers, as the empty token is then not available for concurrent transactions.

pycolo.token.**isAcquired**(*token*)
Checks if a token is acquired by this manager.

Parameters **token** – The token to check

Returns True iff the token is currently in use

pycolo.token.**nextToken**()
Returns the next message ID to use out of the consecutive 16 - bit range.

Returns the current message ID

pycolo.token.**releaseToken**(*token*)
Releases an acquired token and makes it available for reuse.

Parameters **token** – The token to release

CHAPTER

THREE

TESTING GUIDE

Pycolo aims to be very well tested. Test can also be good to provide working snippets.

3.1 tests Package

3.1.1 ETSI testing

This part of the documentation is related to the test performed during the ETSI CTI Plugtests for CoAP protocol.

The goal of interoperability test is to check that devices resulting from protocol implementations are able to work together and provide the features provided by the protocols.

The test descriptions are provided in proforma tables. The following different types of test operator actions are considered duri

- A *stimulus* corresponds to an event that enforces an EUT to proceed with a specific protocol action, like sending a message for instance
- A *verify* consists of verifying that the EUT behaves according to the expected behaviour (for instance the EUT behaviour shows that it receives the expected message)
- A *configure* corresponds to an action to modify the EUT configuration
- A *check* ensures the correctness of protocol messages on reference points, with valid content according to the specific interoperability test purpose to be verified.

etsi Package

`etsi` Package

Mandatory Tests

`TestCore` Module

Optional Tests

`TestBlock` Module

`TestLink` Module

`TestObserve` Module

CoAP Binding for M2M REST Resources

TODO

Automatic testing

`irisa` Module

This module sends trace values to IRISA servers in order to be tested

3.1.2 Example testing

TestHelloWorld Module

TestImageResource Module

TestTimeResource Module

TestStorage Module

TestExampleServer Module

TestWeather Module

3.1.3 Basic testing

TestBytes Module

TestMessage Module

TestOption Module

TestRequest Module

TestResourceTest Module

TestTokenEquality Module

CHAPTER

FOUR

COMMUNITY GUIDE

This part of the documentation, which is mostly prose, details the Pycolo ecosystem and community.

4.1 Support

If you have a questions or issues about Pycolo, there are several options:

4.1.1 Send a Tweet

If your question is less than 140 characters, feel free to send a tweet to @remyleone.

4.1.2 File an Issue

If you notice some unexpected behavior in Pycolo, or want to see support for a new feature, file an issue on GitHub.

4.1.3 E-mail

I'm more than happy to answer any personal or in-depth questions about Pycolo. Feel free to email remy.leone@gmail.com.

CHAPTER

FIVE

DEVELOPER GUIDE

If you want to contribute to the project, this part of the documentation is for you.

5.1 How to Help

Pycolo is under active development, and contributions are more than welcome!

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug. There is a Contributor Friendly tag for issues that should be ideal for people who are not very familiar with the codebase yet.
2. Fork the repository on Github to start making your changes to the **develop** branch (or branch off of it).
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug the maintainer until it gets merged and published. :) Make sure to add yourself to AUTHORS.

5.1.1 What Needs to be Done

- Passing all ETSI tests.
- Passing all examples tests.
- Add automatic testing support with IRISA website (http://senslab2.irisa.fr/coap/)
- Add support for DTLS.
- Check working state in IPv4/IPv6

5.2 Authors

Pycolo is written and maintained by Remy Leone and is inspired by a lot of good software:

5.2.1 Development Lead

- Remy Leone <remy.leone@gmail.com>

5.2.2 Inspiration (These guys inspired pycolo but aren't part of the dev process yet).

- Requests (Kenneth Reitz)
- Californium (Matthias Kovatsch)

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

s

t