python-mcollective Documentation

Release 0.0.1

Rafael Durán Castañeda

June 23, 2015

Contents

CHAPTER 1

Contents

1.1 python-mcollective

1.1.1 Introduction

Python bindings for MCollective inspired by mcollective-python example. Making a ping discovery agent call is just 3 lines:

```
>>> config = config.Config.from_configfile('client.cfg')
>>> msg = message.Message(body='ping', agent='discovery', config=config)
>>> pprint.pprint(rpc.SimpleAction(config=config, msg=msg, agent='discovery').call())
[{':body': 'pong',
  ':msgtime': 1395419893,
  ':requestid': '003ba8142857ccb42cfc4d51262739ecafd43aca',
  ':senderagent': 'discovery',
  ':senderid': 'mco1'}]
```

1.1.2 Features

- MCollective 2.0 - 2.4
- Python 2.6 - 3.4
- All MCollective connectors (STOMP, RabbitMQ, ActiveMQ)
- SSL security provider (YAML serialization)
- Battle tested
- MCollective configuration files parsing
- MCollective filters

1.1.3 Installation

Install it just with pip:

```
$ pip install --pre python-mcollective
```

The `pre` argument is required since there is no stable releases yet.

1.1.4 Contribute

- Issue Tracker: https://github.com/rafaduran/python-mcollective/issues
- Source Code: https://github.com/rafaduran/python-mcollective
- Documentation: http://python-mcollective.readthedocs.org/en/latest/

1.1.5 Support

If you are having issues, please just open an issue on GitHub.

1.1.6 License

The project is licensed under the BSD license.

1.2 API

Contents:

1.2.1 Configuration

pymco.config

Provides MCollective configuration parsing and an entry point for getting the right plugin classes.

class pymco.config.**Config**(*configdict*, *logger=<logging.Logger object>*)
python-mcollective confiugration class.

Parameters **configdict** (*dict*) – a dictionary like object containing configuration as key values.

static **from_configfile**(*configfile*)
Reads configfile and returns a new *Config* instance

Parameters **configfile** – path to the configuration file to be parsed.

Returns *Config* instance.

static **from_configstr**(*configstr*, *section='default'*)
Parses given string an returns a new *Config* instance

Parameters

- **configstr** – configuration file content as string.
- **section** – dummy section to be used for parsing configuration as INI file.

Returns *Config* instance.

get(*name*, **args*, ***kwargs*)
Get option by key.

Parameters **key** – key to look for.

get_conn_params()
Get STOMP connection parameters for current configuration.

Returns Dictionary with stomp.py connection like key/values.

get_connector()
Get connector based on MCollective settings.

get_host_and_ports()
Get all hosts and port pairs for the current configuration.

The result must follow the `stomp.Connection` `host_and_ports` parameter.

Returns Iterable of two-tuple where the first element is the host and the second is the port.

get_security()
Get security plugin based on MCollective settings.

get_serializer(*key*)
Get serializer based on MCollective settings.

get_ssl_params()
Get SSL configuration for current connector

Returns An iterable of SSL configuration parameters to be used with `stomp.Transport.set_ssl()`.

get_user_and_password(*current_host_and_port=None*)
Get the user and password for the current host and port.

Parameters **current_host_and_port** – two-tuple iterable where the first element is the host and second is the port. This parameter is not required for *pymco.connector.stomp.StompConnector* connector.

Returns Two-tuple where the first element is the user and the second is the password for the given host and port.

Raises

- **ValueError** – if connector isn't `stomp` and `host_and_port` is not provided.
- ***pymco.exc.ConfigLookupError*** – if host and port are not found into the connector list of host and ports.

getboolean(*name*, **args*, ***kwargs*)
Get bool option by key.

Acceptable truly values are: true, y, 1 and yes, thought MCollective only officially supports 1.

Parameters **key** – key to look for.

getfloat(*name*, **args*, ***kwargs*)
Get float option by key.

Parameters **key** – key to look for.

getint(*name*, **args*, ***kwargs*)
Get int option by key.

Parameters **key** – key to look for.

pymco.config.**lookup_with_default**(*fnc*)
Wraps ConfigParser lookups, catching exceptions and providing defaults.

Parameters **fnc** – Function to be decorated.

pymco.config.**INFINITE**
Constant used for trying middleware connections indefinitely.

1.2.2 Connectors

Connectors base

python-mcollective base for MCollective connector plugins.

class pymco.connector.**BaseConnector**(*config*, *connection=None*, *logger=<logging.Logger object>*)

Base abstract class for MCollective connectors.

Parameters

- **config** (pymco.config.Config) – configuration instance.
- **connection** (*stomp.Connection*) – connection object. Optional parameter, if not given *default_connection()* result will be used.

connect(*wait=None*)

Connect to MCollective middleware.

Parameters **wait** (*bool*) – wait for connection to be established or not.

Returns self

classmethod default_connection(*config*)

Creates a stomp.Connection object with defaults

Returns stomp.Connection object.

disconnect()

Disconnet from MCollective middleware.

Returns self

get_current_host_and_port()

Get the current host and port from the tracker listener.

Returns A two-tuple, where the first element is the current host and the second the current port.

receive(*timeout*, **args*, ***kwargs*)

Subscribe to MCollective topic queue and wait for just one message.

Parameters

- **timeout** (*float*) – how long we should wait for the message in seconds.
- ***args** – extra positional arguments.
- ****kwargs** – extra keyword arguments.

Returns received message.

Raise *pymco.exc.TimeoutError* if expected messages doesn't come in given timeout seconds.

security

Security provider property.

send(*msg*, *destination*, **args*, ***kwargs*)

Send an MCollective message.

Parameters

- **msg** (pymco.message.Message) – message to be sent.
- ***args** – extra positional arguments.

- ****kwargs** – extra keyword arguments.

Returns `self`.

set_listeners()
Set default listeners.

set_ssl()
Set the SSL configuration for the current connection.

subscribe(*destination*, *id=None*, **args*, ***kwargs*)
Subscribe to MCollective queue.

Parameters

- **destination** – Target to subscribe.
- ***args** – extra positional arguments.
- ****kwargs** – extra keyword arguments.

Returns `self`.

unsubscribe(*destination*, **args*, ***kwargs*)
Unsubscribe to MCollective queue.

Parameters

- **destination** – Target to unsubscribe.
- ***args** – extra positional arguments.
- ****kwargs** – extra keyword arguments.

Returns `self`.

use_b64
Determines if the message should be base64 encoded.

pymco.connector.**get_reply_target**(*self*, *agent*, *collective*)
Get the message target for the given agent and collective.

Parameters

- **agent** – MCollective target agent name.
- **collective** – MCollective target collective.

Returns message reply target string representation for given agent and collective.

pymco.connector.**get_target**(*self*, *agent*, *collective*, *topciprefix=None*)
Get the message target for the given agent and collective.

Parameters

- **agent** – MCollective target agent name.
- **collective** – MCollective target collective.
- **topicprefix** – Required for older versions of MCollective

Returns Message target string representation for given agent and collective.

Connector plugins

ActiveMQ connector

pymco.connector.activemq Contains ActiveMQ specific connector.

class pymco.connector.activemq.**ActiveMQConnector**(*config*, *connection=None*, *logger=<logging.Logger object>*)

ActiveMQ middleware specific connector.

get_reply_target(*agent*, *collective*)
Implement pymco.connector.Connector.get_reply_target()

get_target(*agent*, *collective*)
Implement pymco.connector.Connector.get_target()

send(*msg*, *destination*, **args*, ***kwargs*)
Re-implement pymco.connector.Connector.send()

This implementation adds extra features for ActiveMQ.

RabbitMQ connector

pymco.connector.rabbitmq RabbitMQ specific connector plugin.

class pymco.connector.rabbitmq.**RabbitMQConnector**(*config*, *connection=None*, *logger=<logging.Logger object>*)

RabbitMQ middleware specific connector.

get_reply_target(*agent*, *collective*)
Implement pymco.connector.Connector.get_reply_target()

get_target(*agent*, *collective*)
Implement pymco.connector.Connector.get_target()

STOMP connector

pymco.connector.stomp STOMP specific connnector plugin.

class pymco.connector.stomp.**StompConnector**(*config*, *connection=None*, *logger=<logging.Logger object>*)

STOMP generic middleware connector.

get_reply_target(*agent*, *collective*)
Implement pymco.connector.Connector.get_reply_target()

get_target(*agent*, *collective*)
Implement pymco.connector.Connector.get_target()

1.2.3 Exceptions

pymco.exc

python-mcollective exceptions.

exception pymco.exc.**BadFilterFactOperator**
Exception raised when trying to add an unsopported fact operator to filters.

exception pymco.exc.**ConfigLookupError**
Exception raised on configuration lookups errors.

exception pymco.exc.**ImproperlyConfigured**
Exception raised on configuration errors.

exception pymco.exc.**PyMcoException**
Base class for all python-mcollective exceptions

exception pymco.exc.**TimeoutError**
Exception to be raised on timeouts

exception pymco.exc.**VerificationError**
Exception to be raised on message verification errors.

1.2.4 Listeners

pymco.listeners

stomp.py listeners for python-mcollective.

class pymco.listener.**CurrentHostPortListener**(**args*, ***kwargs*)
Listener tracking current host and port.

Some connectors, like ActiveMQ connector, may provide different user and password for each host, so we need track the current host and port in order to be able to get the right user and password when logging.

get_host()
Return current host.

Returns current host.

get_port()
Return current host.

Returns current port.

on_connecting(*host_and_port*)
Track current host and port.

Parameters **host_and_port** – A two-tuple with host as first element and port as the second.

class pymco.listener.**ResponseListener**(*config*, *connector*, *count*, *timeout=30*, *condition=None*, *logger=<logging.Logger object>*)
Listener that waits for a message response.

Parameters

- **config** – *pymco.config.Config* instance.
- **count** – number of expected messages.
- **timeout** – seconds we should wait for messages.
- **condition** – by default a `threading.Condition` object for synchronization purposes, but you can use any object implementing the `wait()` method and accepting a `timeout` argument.

on_message(*headers*, *body*)
Received messages hook.

Parameters

- **headers** – message headers.

- **body** – message body.

security
Security provider property

wait_on_message()
Wait until we get a message.

Returns self.

1.2.5 Messaging

pymco.message

python-mcollective messaging objects.

class pymco.message.**Filter**
Provides MCollective filters for python-mcollective.

This class implements collections.Mapping interface, so it can be used as non mutable mapping (read only dict), but mutable using provided add methods. So that, for adding the agent you can just use *add_agent()*:

```
filter.add_agent('package')
```

add_agent(*agent*)
Add new MCollective agent

Parameters **agent** – MCollective agent name.

Returns self so filters can be chained.

add_cfclass(*name*)
Add new class applied by your configuration management system.

Roles, cookbooks,... names may be used too.

Parameters **name** – class, role, cookbook,... name.

Returns self so filters can be chained.

add_fact(*fact*, *value*, *operator=None*)
Add a new Facter fact based filter.

Parameters

- **fact** – fact name.
- **value** – fact value.
- **operator** – Operator to be applied when comparing the fact. Valid values are: ==, <=, >=, <, >, !=. Optional parameter.

Returns self so filters can be chained.

add_identity(*identity*)
Adds new identities

Parameters **identity** – MCollective identity value.

Returns self so filters can be chained.

class pymco.message.**Message**(*body*, *agent*, *config*, *filter_=None*, ***kwargs*)
Provides MCollective messages for python-mcollective.

This class implements collections.MutableMapping interface, so it can be used as read/write mapping (dictionary).

Parameters

- **body** – the message body. It must be serializable using current serialization method.
- **agent** – message target agent.
- **config** – *pymco.config.Config* instance.
- **filter** – *Filter* instance. This parameter is optional.
- **kwargs** – Extra keyword arguments. You can set the target collective or the message ttl using them.

Raise *pymco.exc.ImproperlyConfigured* if configuration has no identity or collective is not set neither in kwargs nor in configuration.

1.2.6 RPC

pymco.rpc

MCollective RPC calls support.

class pymco.rpc.**SimpleAction**(*config*, *msg*, *agent*, *logger=<logging.Logger object>*, ***kwargs*)
Single RPC call to MCollective

Parameters

- **config** – *pymco.config.Config* instance.
- **msg** – A dictionary like object, usually a *pymco.message.Message* instance.
- ****kwargs** – extra keyword arguments. Set the collective here if you aren't targeting the main collective.

call(*timeout=5*)
Make the RPC call.

It should subscribe to the reply target, execute the RPC call and wait for the result.

Parameters **timeout** – RPC call timeout.

Returns a dictionary like object with response.

Raise *pymco.exc.TimeoutError* if expected messages don't arrive in timeout seconds.

get_reply_target()
MCollective RPC call reply target.

This should build the subscription target required for listening replies to this RPC call.

Returns middleware target for the response.

get_target()
MCollective RPC call target.

Returns middleware target for the request.

1.2.7 Security

python-mcollective security provider plugins.

Security providers base

MCollective security providers base.

class pymco.security.**SecurityProviderBase**(*config*, *logger=<logging.Logger object>*)
Abstract base class for security providers.

Parameters **config** – *pymco.config.Config* instance.

decode(*msg*, *b64=False*)
Decode given message using provided security method.

Decode will consist just on de-serialize the given message and verify it, raising a verification error if the message can't be verified.

Parameters **msg** (pymco.message.Message) – Message to be serialized.

Returns Decoded message, a `dict` like object.

deserialize(*msg*)
Deserealize message using provided serialization.

Parameters **msg** (pymco.message.Message) – message to be decoded.

Returns decoded message.

encode(*msg*, *b64=False*)
Encode given message using provided security method.

Encode will consist just on singing the message and serialize it, so we can sent it and verified for the receivers.

Parameters **msg** (pymco.message.Message) – Message to be serialized.

Returns Encoded message.

serialize(*msg*)
Serialize message using provided serialization.

Parameters **msg** (pymco.message.Message) – message to be encoded.

Returns encoded message.

pymco.security.**sign**(*self*, *msg*)
Signs the given message using provided security method.

Parameters **msg** (pymco.message.Message) – message to be signed.

Returns signed message.

pymco.security.**verify**(*self*, *msg*)
Verify the given message using provided security method.

Parameters **msg** (pymco.message.Message) – message to be verified.

Returns verified message.

Raises **pymco.exc.MessageVerificationError** If the message verification failed.

Security providers plugins

None provider

pymco.security.none Contains none specific security provider.

class pymco.security.none.**NoneProvider**(*config*, *logger=<logging.Logger object>*)
Provides message signing for MCollective::Security::None sec. provider

The none provider is a dummy provider just for developing that isn't included with MCollective but into fixtures directory.

sign(*message*)
Implement pymco.security.SecurityProvider.sign().

Add the current user as :callerid key to the message.

verify(*message*)
Implement pymco.security.SecurityProvider.verify().

It does nothing, returning always given message.

SSL provider

1.2.8 Serializers

python-mcollective serializers.

Serializers base

pymco Message [de]serialization.

pymco.serializers.**deserialize**(*self*, *msg*)
De-serialize a MCollective msg.

Parameters **msg** (pymco.message.Message) – message to be de-serialized.

Returns de-serialized message.

pymco.serializers.**serialize**(*self*, *msg*)
Serialize a MCollective msg.

Parameters **msg** – message to be serialized.

Returns serialized message.

class pymco.serializers.**SerializerBase**
Base class for all serializers.

Serializers plugins

YAML

Security providers base Provides YAML [de]serialization.

class pymco.serializers.yaml.**RubyCompatibleLoader**(*stream*)
YAML loader compatible with Ruby Symbols

class pymco.serializers.yaml.**Serializer**
YAML specific serializer.

pymco.serializers.yaml.**ruby_object_constructor**(*loader*, *suffix*, *node*)
YAML constructor for Ruby objects.

This constructor may be registered with '!ruby/object:' tag as multi constructor supporting Ruby objects. This will handle give objects as maps, so any non mapping based object may produce some issue.

pymco.serializers.yaml.**symbol_constructor**(*loader*, *node*)
YAML constructor for Ruby symbols.

This constructor may be registered with '!ruby/sym' tag in order to support Ruby symbols serialization (you can use `register_constructors()` for that), so it just need return the string scalar representation of the key (including the leading colon).

1.2.9 Test utils

python-mcollective test utils.

pymco.test.ctxt

Context information for sourcing test templates.

pymco.test.ctxt.**ROOT**
Tests root directory.

pymco.test.ctxt.**DEFAULT_CTXT**
Default context used for sourcing MCollective configuration template.

pymco.test.ctxt.**TEST_CFG**
Tests configuration file name (full path).

pymco.test.ctxt.**MSG**
MCollective sample message dictionary.

pymco.test.utils

Utils for testing purposes.

pymco.test.utils.**configfile**(*ctxt=None*)
Create a MCollective configuration file.

Parameters **ctxt** (*dict*) – the ctxt to be used for rendering MCollective configuration template.

Returns The path where the configuration file has been placed (*pymco.test.ctxt.TEST_CFG*).

pymco.test.utils.**get_template**(*name*, *package='pymco.test'*)
Load Jinja 2 template from given package.

Parameters

- **name** – template name.
- **package** – package to be used for loading the template, default is current package.

Returns `jinja2.environment.Template` object.

1.2.10 Utils

pymco.utils

python-mcollective utils that don't fit elsewhere.

pymco.utils.**import_class**(*import_path*)
Import a class based on given dotted import path string.

It just splits the import path in order to geth the module and class names, then it just calls to `__import__()` with the module name and `getattr()` with the module and the class name.

Parameters **import_path** – dotted import path string.

Returns the class once imported.

Raise `ImportError` if the class can't be imported.

pymco.utils.**import_object**(*import_path*, **args*, ***kwargs*)
Import a class and instantiate it.

Uses *import_class()* in order to import the given class by its import path and instantiate it using given positional and keyword arguments.

Parameters

- **import_path** – Same argument as *import_class()*.
- ***args** – extra pPositional arguments for object instantiation.
- ****kwargs** – extra Keyword arguments for object instantiation.

Returns an object the imported class initialized with given arguments.

pymco.utils.**load_rsa_key**(*filename*)
Read filename and try to load its contents as an RSA key.

Wrapper over `Crypto.PublicKey.RSA.importKey()`, just getting the file content first and then just loading the key from it.

Parameters **filename** – RSA key file name.

Returns loaded RSA key.

pymco.utils.**pem_to_der**(*pem*)
Convert an ascii-armored PEM certificate to a DER encoded certificate

See http://stackoverflow.com/a/12921889 for details. Python `ssl` module has it own method for this, but it shouldn't work properly and this method is required.

Parameters **pem** (*str*) – The PEM certificate as string.

1.3 python-mcollective style guide

This guide is highly inspired on OpenStack guidelines.

- Step 1: Read http://www.python.org/dev/peps/pep-0008/
- Step 2: Read http://www.python.org/dev/peps/pep-0008/ again
- Step 3: Read on

1.3.1 General

- Use only UNIX style newlines (`\n`), not Windows style (`\r\n`)
- Wrap long lines in parentheses and not a backslash for line continuation.
- Do not write `except:`, use `except Exception:` at the very least
- Include your name with TODOs as in `#TODO(yourname)`
- Do not shadow a built-in or reserved word. Example:

```
def list():
    return [1, 2, 3]

mylist = list() # BAD, shadows `list` built-in

class Foo(object):
    def list(self):
        return [1, 2, 3]

mylist = Foo().list() # OKAY, does not shadow built-in
```

- Use the `is not` operator when testing for unequal identities. Example:

```
if not X is Y:  # BAD, intended behavior is ambiguous
    pass

if X is not Y:  # OKAY, intuitive
    pass
```

- Use the `not in` operator for evaluating membership in a collection. Example:

```
if not X in Y:  # BAD, intended behavior is ambiguous
    pass

if X not in Y:  # OKAY, intuitive
    pass

if not (X in Y or X in Z):  # OKAY, still better than all those 'not's
    pass
```

1.3.2 Imports

- Do not import objects, only modules (*)
- Do not import more than one module per line
- Do not use wildcard `*` import
- Order your imports by the full module path
- Organize your imports according to the following template

(*) exceptions are:

- imports from `.` package

Example:

```
{{stdlib imports in human alphabetical order}}
\n
{{third-party lib imports in human alphabetical order}}
\n
{{project imports in human alphabetical order}}
\n
\n
{{begin your code}}
```

1.3.3 Human Alphabetical Order Examples

Example:

```
import httplib
import logging
import random
import StringIO
import time
import unittest

import eventlet
import webob.exc

import nova.api.ec2
from nova.api import openstack
from nova.auth import users
from nova.endpoint import cloud
import nova.flags
from nova import test
```

1.3.4 Docstrings

Example:

```
"""A one line docstring looks like this and ends in a period."""


"""A multi line docstring has a one-line summary, less than 80 characters.

Then a new paragraph after a newline that explains in more detail any
general information about the function, class or method. Example usages
are also great to have here if it is a complex class for function.

When writing the docstring for a class, an extra line should be placed
after the closing quotations. For more in-depth explanations for these
decisions see http://www.python.org/dev/peps/pep-0257/

If you are going to describe parameters and return values, use Sphinx, the
appropriate syntax is as follows.

:arg str foo: the foo parameter of type :py:class:`str`
:param bar: the bar parameter
:returns: return_type -- description of the return value
:returns: description of the return value
:raises: AttributeError, KeyError
"""
```

1.3.5 Dictionaries/Lists

If a dictionary (dict) or list object is longer than 80 characters, its items should be split with newlines. Embedded iterables should have their items indented. Additionally, the last item in the dictionary should have a trailing comma. This increases readability and simplifies future diffs.

Example:

```
my_dictionary = {
    "image": {
        "name": "Just a Snapshot",
        "size": 2749573,
        "properties": {
            "user_id": 12,
            "arch": "x86_64",
        },
        "things": [
            "thing_one",
            "thing_two",
        ],
        "status": "ACTIVE",
    },
}
```

Do not use `locals()` for formatting strings, it is not clear as using explicit dictionaries and can hide errors during refactoring.

1.3.6 Calling Methods

Calls to methods 80 characters or longer should format each argument with newlines. This is not a requirement, but a guideline:

```
unnecessarily_long_function_name('string one',
                                 'string two',
                                 kwarg1=constants.ACTIVE,
                                 kwarg2=['a', 'b', 'c'])
```

Rather than constructing parameters inline, it is better to break things up:

```
list_of_strings = [
    'what_a_long_string',
    'not as long',
]

dict_of_numbers = {
    'one': 1,
    'two': 2,
    'twenty four': 24,
}

object_one.call_a_method('string three',
                         'string four',
                         kwarg1=list_of_strings,
                         kwarg2=dict_of_numbers)
```

1.3.7 Creating Unit Tests

For every new feature, unit tests should be created that both test and (implicitly) document the usage of said feature. If submitting a patch for a bug that had no unit test, a new passing unit test should be added. If a submitted bug fix does have a unit test, be sure to add a new one that fails without the patch and passes with the patch.

1.3.8 Commit Messages

Using a common format for commit messages will help keep our git history readable. Follow these guidelines:

> First, provide a brief summary of 50 characters or less. Summaries of greater then 72 characters will be rejected by the gate.
>
> The first line of the commit message should provide an accurate description of the change, not just a reference to an issue. It must not end with a period and must be followed by a single blank line.
>
> Following your brief summary, provide a more detailed description of the patch, manually wrapping the text at 72 characters. This description should provide enough detail that one does not have to refer to external resources to determine its high-level functionality.

For further information on constructing high quality commit messages, and how to split up commits into a series of changes, consult the OpenStack project wiki:

> https://wiki.openstack.org/GitCommitMessages

1.3.9 Flake8

As part of Travis-CI tests setup, flake8 is being ran, so any style problem will make tests fail and no changes will integrated until the problem is fixed.

1.3.10 Further Reading

> http://google-styleguide.googlecode.com/svn/trunk/pyguide.html

1.4 Running the tests

1.4.1 Travis-CI

All pull requests will be tested with Travis-CI, so you can just trust on it. However it's recommend to test locally before sending the pull request, sections below will show you how to do that.

1.4.2 Vagrant setup

The repository is ready for running a Vagrant box with all needed in order to deploy an VM with RabbitMQ and ActiveMQ with and without SSL. You can start it just typing:

```
$ vagrant up
```

This is required for running integration tests locally, thought you can install both locally.

1.4.3 Local MCollective setup

In order to run MCollective locally you will need:

- RabbitMQ or ActiveMQ running: you can use provided Vagrant setup or just install one of them locally.
- Clone Git submodules if you didn't clone the repository recursively:

```
$ git submodule init
$ git submodule update
```

- Install dependencies, from repository root:

```
$ bundle install
```

- Then you need configuration files placed in the repository root, into `examples` directory you will find some configuration examples:

```
$ cp examples/server.23x.activemq.cfg server.cfg
$ cp examples/server.23x.activemq.cfg client.cfg
```

  Edit configuration files to fix paths for your working directory.

- Then you should be able to run both, the daemon and the client:

```
$ scripts/mcollectived
```

  From another terminal:

```
$ scripts/mco ping
```

  Now everything should be working and you should see `mco ping` output, otherwise you will need review steps before.

This is also required for running integration tests, since they spawn MCollective daemons so we can make RPC calls to them.

1.4.4 Running py.test

pytest is the test framework for python-mcollective, in order to run the tests with it:

- You probably want to create a virtualenv, with virtualenvwrapper:

```
$ mkvirtualenv pymco
```

- Install dependencies:

```
$ pip install -r requirements/tests.txt
```

- Then just type:

```
$ py.test
```

You can skip integration tests just typing:

```
$ py.test tests/unit
```

or run only integration with:

```
$ py.test tests/integration
```

1.4.5 Running tox

Additionally you can run tox for Python compatibility testing, style checks and documentation building. This setup is pretty close to the Travis-CI, so if tests pass it should pass Travis-CI.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

p

A

B

C

D

E

F

G

I