Developper manual of RasPy

Release 0.0.1

bibi21000

January 20, 2015

Contents

CHAPTER 1

Installation

The following lines "clone" a GitHub repository. If you want to submit "pull requests", you need to "fork" RasPy using this guide.

1.1 Raspbian

Install official raspbian from here : http://www.raspbian.org/.

Newbies can install it from : http://www.raspberrypi.org/downloads/.

Developpers or others can also install it on standard distributions (Ubuntu, Debian, RedHat, ...).

1.2 Update packages

You can now update packages

```
sudo apt-get -y update
sudo apt-get -y dist-upgrade
```

We need to install some packages to download and build RasPy:

```
sudo apt-get -y install build-essential python-dev python-minimal python python2.7-dev python2.7-m
sudo apt-get -y install git python-setuptools python-docutils python-pylint
```

And some development librairies for rrdtool (for the logger) :

```
sudo apt-get -y install libcairo2-dev libpango1.0-dev libglib2.0-dev libxml2-dev librrd-dev
```

Some packages need to be removed as new versions are available from eggs :

```
sudo apt-get remove python-zmq libzmq1 libzmq-dev python-nose pylint
```

1.3 Download it

You should now download RasPy using git. You should not download and install RasPy with root user. Idealy, you should create a special user for running RasPy or the pi user. Keep in mind root is baaaddddd (Not support for installation as root !!!).

```
git clone https://github.com/bibi21000/RasPy.git
```

1.4 Configure your system

- access rights
- sudo nopasswd
- ...

1.5 Installation

If you want to develop for RasPy, you need to install it in develop mode :

```
sudo make develop
```

Otherwise install it normaly ... but not now ;) :

```
sudo make install
```

And be patient ... installation need to compile zmq ... It takes a while ...

If something goes wrong during install or if you want to remove RasPy from you computer, you type :

```
sudo make uninstall
```

If you want to remove dependencies, look at setup.py to get the list and use the following command for every package:

```
sudo pip uninstall package
```

1.6 Run the tests

Check that the SLEEP constant in tests/common.py ist set to 1.0 or 1.5

```
vim tests/common.py
```

You can now check that everything is fine running the tests :

```
make tests
```

If it fails ... run it again :) At last, copy / paste the full screen output and send it to the core team.

1.7 Start it

In the next monthes, you should be abble to start it :

```
make start
```

1.8 Read the doc

- docs/pdf
- docs/html

CHAPTER 2

Before starting

If you want to develop you surely need vim :

```
sudo apt-get -y install vim-nox vim-addon-manager
```

2.1 Phylosophy

Tests, tests, ... and tests :

- A bug -> a test -> a patch
- A new feature -> many test

And documentation

- A new feature -> documentation

2.2 Documentation

If you want to generate the documentation, you need to install some packages :

```
sudo apt-get -y install python-sphinx graphviz
```

And some eggs :

```
sudo pip install seqdiag sphinxcontrib-seqdiag
sudo pip install blockdiag sphinxcontrib-blockdiag
sudo pip install nwdiag sphinxcontrib-nwdiag
sudo pip install actdiag sphinxcontrib-actdiag
```

You can now generate the full documentation using :

```
make docs
```

You can also generate a part of it, for example :

```
cd docs
make html
```

2.3 Tests

Nosetests and pylint are used to test quality of code. There reports are here :

- Nosetests report

- Coverage report
- Pylint report

Coverage is not the goal but it's one : a module must have a coverage of 90% to be accpeted by core team. Otherwise it will block the packaging process. Of course, a FAILED test will also.

Keep in mind that all tests must succeed before submitting pull request. But :

- if a test is a work in progress, you can skip it using self.wipTest()
- if a test can only be run on Raspberry (ie onewire), it must call self.skipTest(message) at its start.

There is 2 ways to launch the tests. The first one to use on a Raspberry :

```
make tests
```

You can also run the developpers tests (without skipped one) on a standard computer running :

```
make devtests
```

If you're on a raspberry, you can run the full tests like this :

```
make tests
```

Running only one test module :

```
/usr/local/bin/nosetests --verbosity=2 --cover-package=raspy --with-coverage --cover-inclusive --
```

You can follow automatic tests on travis-ci.

2.4 GitHub

You can test the code, build the doc and commit it using the following command :

```
make git
```

You may use ssh_keys to do it automatically without typing password.

CHAPTER 3

Develop

3.1 A new device

3.2 A new server

CHAPTER 4

raspy package

4.1 Subpackages

4.1.1 raspy.common package

Subpackages

raspy.common.devices package

Submodules

raspy.common.devices.device module Devices.

class raspy.common.devices.device.**BaseDevice**(*json=None*)

Bases: object

The base device object

What is a device :

- a temperature sensor
- a wind sensor
- a camera
- the clock RTC
- a dimmer
- a TV
- ...

What can we do with a device : a command (same "spirit" as zwave's command classes

- configure it
- get value of a sensor
- dim a dimmer
- take a photo with camera
- ...

We shoud do auto-mapping :

- python object <-> json
- python object <-> html

We whould manage complex devices, ie a TV : it groups a channel selector (+, -, and direct access to a channel), a volume selector, ... In an ideal world we should not be obliged to create each sub-devices.

Naming convention of devices on the network : (MDP.routing_key(hostname, service)).{device_name}[.subdevice] Naming convention for devices / subdevices : categorie.device-subdevice (ie media.tv-volume, ...)

check (*json=None*)
Check that the JSON is a valid device

cmd_commands (*value=None*)
Command fof retrieving all commands supported by this device

Parameters value – the value. If value is None this command send the current config. Otherwise it will set it to calue

cmd_config (*value=None*)
Command for configuring device Must be overloaded and called by the subclass

Parameters value – the value. If value is None this command send the current config. Otherwise it will set it to calue

cmd_log (*value=None*)
Command for configuring logging of the device

cmd_poll (*value=None*)
Command for configuring polling of the device

cmd_reset (*value=None*)
Command for resetting device Must be overloaded (and called) by the subclass

commands = {}
The commands available on the device

config = {'name': None}
The device's configuration

do_poll ()
Grab the value and return it

exec_cmd (*oid*, *command*, *value=None*)
Execute a command

Parameters

- **device** – the oid device
- **command** – the cid device
- **value** – the value

Returns a value if the command succeed. None if it fails

fullname (*prefix*)
The fullname of the device

json
Check that the JSON is a valid device

log = False
Should we log this value (in the RasPy Logger)

name
The name of the device Must be unique for the instance server.

new (*json=None*)
Create a new device and return it

oid = 'base'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

poll = -1
The poll value of this device : -1 not pollable, 0 : not poll and other = poll delay in seconds

subdevices = None
The subdevices of this device

template
The template of the device

templates = {}
The templates dictionnary Every device must add an entry for its config Will be used to check the device

class raspy.common.devices.device.**Command**(***kwargs*)
Bases: object

A command for a device. Use the same "spirit" as ZWave command classes

callback = None
The callback method associated with this command

from_json(*json=None*)
Create the command from JSON

info = 'info'
Some information about the command Will be used to represent the value ie in a form, in a graph, ...

readonly = False
Is this command readonly ie a sensor

to_json()
Copy command to JSON

type = 'List'
The type of the value Will be used to represent the value ie in a form, in a graph, ...

value = None
The value

writeonly = False
Is this command writeonly ie a change dimmer command

class raspy.common.devices.device.**DeviceRegister**
Bases: object

The device register

All devices must register to this register (in main module)

check(*json=None*)
Check that the JSON is a valid device

new(***kwargs*)
Create a new device and return it

register(*device_type*)
Register a device_type under key

raspy.common.devices.media module Media devices

class raspy.common.devices.media.**MediaCamera**(***kwargs*)
Bases: raspy.common.devices.media.MediaDevice

The camera device object

new(***kwargs*)
Create a new device and return it

oid = 'media.camera'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

class raspy.common.devices.media.**MediaDevice**(***kwargs*)
Bases: raspy.common.devices.device.BaseDevice

The sensor device object

check(*json=None*)
Check that the JSON is a valid device

oid = 'media'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

class raspy.common.devices.media.**MediaTV**(***kwargs*)
Bases: raspy.common.devices.media.MediaDevice

The temperature sensor device object

new(***kwargs*)
Create a new device and return it

oid = 'media.tv'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

raspy.common.devices.sensor module Sensors devices

class raspy.common.devices.sensor.**SensorDevice**(***kwargs*)
Bases: raspy.common.devices.device.BaseDevice

The sensor device object

check(*json=None*)
Check that the JSON is a valid device

oid = 'sensor'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

class raspy.common.devices.sensor.**SensorTemperature**(***kwargs*)
Bases: raspy.common.devices.sensor.SensorDevice

The temperature sensor device object

new(***kwargs*)
Create a new device and return it

oid = 'sensor.temperature'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

class raspy.common.devices.sensor.**SensorWind**(***kwargs*)
Bases: raspy.common.devices.sensor.SensorDevice

The wind sensor device object

new(***kwargs*)
Create a new device and return it

oid = 'sensor.wind'
The Object Identifier It should be given by the core team as it can break other devices. Need to define a naming convention : sensor, sensor.temperature, media.camera, ...

Module contents

Submodules

raspy.common.MDP module

exception raspy.common.MDP.**ClientError**(*value*)
Bases: raspy.common.MDP.GenericError

Client side exception

exception raspy.common.MDP.**GenericError**(*value*)
Bases: exceptions.Exception

Generic exception

exception raspy.common.MDP.**ServerError**(*value*)
Bases: raspy.common.MDP.GenericError

Server side exception

raspy.common.MDP.**logger = <logging.Logger object at 0x2b8918e6ffd0>**
Majordomo Protocol definitions

raspy.common.MDP.**routing_key**(*hostname*, *service*)

raspy.common.dynamic module

raspy.common.dynamic.**importCode**(*code*, *name*, *add_to_sys_modules=False*)

raspy.common.executive module

class raspy.common.executive.**Executive**(*hostname='localhost'*, *service='executive'*, *broker_ip='127.0.0.1'*, *broker_port=5514*)
Bases: object

The Executive mother class for all workers

todo :

- bug : can't stop when jobs in queues

destroy()
Wait for threads and destroy contexts.

get_instance_id()
Return the instance of the exective

... todo : must be multihost and multithread.

run()
Run the executive

shutdown()
Shutdown executive.

class raspy.common.executive.**ExecutiveProcess**(*executive*, *executive_name*)
Bases: multiprocessing.process.Process

Process executing tasks from a given tasks queue

run()

shutdown()
Method to deactivate the client connection completely.

Will delete the stream and the underlying socket.

> **Warning:** The instance MUST not be used after `shutdown()` has been called.

Return type None

raspy.common.kvcliapi module

kvsimple - simple key-value message class for example applications

Author: Min RK <benjaminrk@gmail.com>

From : http://zguide.zeromq.org/py:kvsimple

class raspy.common.kvcliapi.**KvPublisherClient**(*hostname='localhost'*, *broker_ip='127.0.0.1'*, *broker_port=5514*)
Bases: object

KeyValue Protocol Client API, Python version.

Implements the client defined at http://zguide.zeromq.org/page:all#Working-with-Subtrees

From https://raw.githubusercontent.com/imatix/zguide/master/examples/Python/clonecli4.py

destroy()
Destroy object

send(*subtree='subtree'*, *key='key'*, *body='body'*)
Send the update

class raspy.common.kvcliapi.**KvSubscriberClient**(*hostname='localhost'*, *subtree='subtree'*, *broker_ip='127.0.0.1'*, *broker_port=5514*, *speed=1.0*)
Bases: object

KeyValue Protocol Client API, Python version.

Implements the client defined at http://zguide.zeromq.org/page:all#Working-with-Subtrees From https://raw.githubusercontent.com/imatix/zguide/master/examples/Python/clonecli4.py

destroy()
Destroy object

run()
Run the poller

shutdown()
Shutdown the broker.

raspy.common.kvsimple module

kvsimple - simple key-value message class for example applications

Author: Min RK <benjaminrk@gmail.com>

From : http://zguide.zeromq.org/py:kvsimple

class raspy.common.kvsimple.**KVMsg**(*sequence*, *key=None*, *body=None*)
Bases: object

Message is formatted on wire as 3 frames:

- frame 0: key (0MQ string)
- frame 1: sequence (8 bytes, network order)

- frame 2: body (blob)

body = None

dump()
Dump me to a string”

key = None

classmethod recv(*socket*)
Reads key-value message from socket, returns new kvmsg instance.

send(*socket*)
Send key-value message to socket; any empty frames are sent as such.

sequence = 0

store(*dikt*)
Store me in a dict if I have anything to store

raspy.common.mdcliapi module

Majordomo Protocol Client API, Python version.

Implements the MDP/Worker spec at http:#rfc.zeromq.org/spec:7.

Author: Min RK <benjaminrk@gmail.com> Based on Java example by Arkadiusz Orzechowski

class raspy.common.mdcliapi.**MajorDomoClient**(*broker*)
Bases: object

Majordomo Protocol Client API, Python version.

Credits : https://github.com/imatix/zguide/blob/master/examples/Python/mdcliapi.py

broker = None

client = None

ctx = None

destroy()
Destroy object

poller = None

reconnect_to_broker()
Connect or reconnect to broker

retries = 5

send(*service*, *request*)
Send request to broker and get reply by hook or crook.

Takes ownership of request message and destroys it when sent. Returns the reply message or None if there was no reply.

timeout = 500

verbose = False

class raspy.common.mdcliapi.**TitanicClient**(*broker_ip='localhost'*, *broker_port=5514*, *poll=1500*, *ttl=900*)
Bases: object

The titanic client

Credits: https://github.com/imatix/zguide/blob/master/examples/Python/ticlient.py

request(*hostname='localhost', service='worker', data=['mmi.echo'], callback=None, args=(), kwargs={}*)
Request a job for a worker to titanic

run()
Run the client in a loop

send(*service*, *request*)
Send a Majordomo request directly to worker

shutdown()
Shutdown executive.

status(*uuid*)
Retrieve the status of a work from titanic

raspy.common.mdwrkapi module

Majordomo Protocol Worker API, Python version

Implements the MDP/Worker spec at http:#rfc.zeromq.org/spec:7.

Author: Min RK <benjaminrk@gmail.com> Based on Java example by Arkadiusz Orzechowski

class raspy.common.mdwrkapi.**MajorDomoWorker**(*broker*, *service*)
Bases: object

Majordomo Protocol Worker API, Python version

Implements the MDP/Worker spec at http:#rfc.zeromq.org/spec:7.

HEARTBEAT_LIVENESS = 5

broker = None

ctx = None

destroy()
Destroy object

expect_reply = False

heartbeat = 3500

heartbeat_at = 0

liveness = 0

reconnect = 3500

reconnect_to_broker()
Connect or reconnect to broker

recv(*reply=None*)
Send reply, if any, to broker and wait for next request.

reply_to = None

send_to_broker(*command*, *option=None*, *msg=None*)
Send message to broker.

. If no msg is provided, creates one internally

service = None

shutdown()
Shutdown executive.

status = True
The status of the worker. Should be update by callback in the future

timeout = 2500

verbose = False

worker = None

raspy.common.runner module

raspy.common.server module

class raspy.common.server.**Server**(*hostname='localhost', service='worker', broker_ip='127.0.0.1', broker_port=5514*)

Bases: raspy.common.executive.Executive, raspy.common.statistics.Statistics

The generic worker

From http://zguide.zeromq.org/py:all#header-48

worker_mmi()

Retrieve mmi informations of the worker

raspy.common.statistics module

class raspy.common.statistics.**SNMP**(*oid='module.snmp.key', doc='A statistic integer value', initial=0*)

Bases: object

Abstract statistic item

set(*value*)

Set a value to the snmp object

class raspy.common.statistics.**SNMPCounter**(*oid='module.snmp.key', doc='A statistic counter value with overflow', initial=0, overflow=4294967296*)

Bases: raspy.common.statistics.SNMP

Long (32bits) with overflow

set(*value=1*)

Add value (default=1) to current value. Also manage overflow.

class raspy.common.statistics.**SNMPFloat**(*oid='module.snmp.key', doc='A statistic float value', initial=0.0*)

Bases: raspy.common.statistics.SNMP

Float counter

class raspy.common.statistics.**SNMPString**(*oid='module.snmp.key', doc='A statistic string value', initial=''*)

Bases: raspy.common.statistics.SNMP

Float counter

class raspy.common.statistics.**Statistics**

Bases: object

The statistics manager

add_statistic()

Add a new statistic to the manager

remove_statistic(*oid*)

Remove a statistic from the manager

update_statistic(*oid=''*)

Add a new statistic to the manager

worker_statistics()
Send statistics via mmi

raspy.common.supervisor module

class raspy.common.supervisor.**Supervisor**(*runner=None*)
The worker supervisor

Start executives in separate process see futures Each executive start multiples threads of workers

todo :

- bug : can't stop when jobs in queues

get_instance_id()
Return the instance of the worker : must be multihost and multithread.

reload()
Request the workers confguration against the configurator.

Will unregister all workers, stop all timers and ignore all further messages.

> **Warning:** The instance MUST not be used after shutdown() has been called.

Return type None

run()
Start the IOLoop instance

shutdown()
Shutdown supervisor.

Will unregister all workers, stop all timers and ignore all further messages.

> **Warning:** The instance MUST not be used after shutdown() has been called.

Return type None

stop_executives()
Shutdown executives.

raspy.common.zhelpers module

Helper module for example applications. Mimics ZeroMQ Guide's zhelpers.h.

raspy.common.zhelpers.**set_id**(*zsocket*)
Set simple random printable identity on socket

raspy.common.zhelpers.**zpipe**(*ctx*)
build inproc pipe for talking to threads

mimic pipe used in czmq zthread_fork.

Returns a pair of PAIRs connected via inproc

Module contents

4.1.2 raspy.servers package

Submodules

raspy.servers.broker module

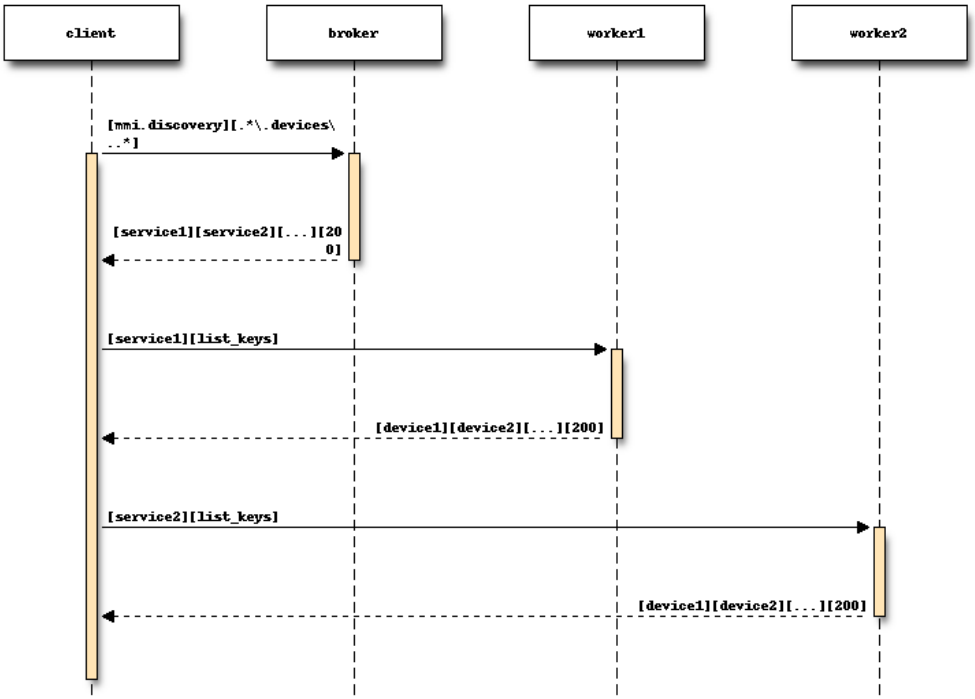
class raspy.servers.broker.**Broker**(*hostname='localhost'*, *service='broker'*, *broker_ip='127.0.0.1'*, *broker_port=15514*)

Bases: raspy.common.executive.Executive

Majordomo Protocol broker and key/value proxy.

Discovery process

Here is the way for a client to discover all devices on network



You can do the same for crons and scenarios

HEARTBEAT_EXPIRY = 17500

HEARTBEAT_INTERVAL = 3500

HEARTBEAT_LIVENESS = 5

INTERNAL_SERVICE_PREFIX = 'mmi.'

ctx = None

delete_worker(*worker*, *disconnect*)
Deletes worker from all data structures, and deletes worker.

destroy()
Disconnect all workers, destroy context.

dispatch(*service*, *msg*)
Dispatch requests to waiting workers as possible

heartbeat_at = None

poller = None

process_client(*sender*, *msg*)
Process a request coming from a client.

process_worker(*sender*, *msg*)
Process message sent to us by a worker.

purge_workers()
Look for & kill expired workers.

Workers are oldest to most recent, so we stop at the first alive worker.

require_service(*name*)
Locates the service (creates if necessary).

require_worker(*address*)
Finds the worker (creates if necessary).

run()
Main broker work happens here

send_heartbeats()
Send heartbeats to idle workers if it's time

send_to_worker(*worker*, *command*, *option*, *msg=None*)
Send message to worker.

. If message is provided, sends that message.

service_internal(*service*, *msg*)
Handle internal service according to 8/MMI specification

services = None

shutdown()
Shutdown the broker.

socket = None

waiting = None

worker_waiting(*worker*)
This worker is now waiting for work.

workers = None

class raspy.servers.broker.**Proxy**(*hostname='localhost'*, *service='broker'*, *broker_ip='*'*, *broker_port=5514*, *speed=1.0*)
Bases: threading.Thread

The publisher

Tree :

- /event/
- /scenario/
- /device/

from : http://zguide.zeromq.org/page:all#Working-with-Subtrees

run()
Run the proxy

send_single(*key*, *kvmsg*, *route*)
Send one state snapshot key-value pair to a socket

shutdown()
Shutdown the proxy.

class raspy.servers.broker.**Route**(*socket*, *identity*, *subtree*)

class raspy.servers.broker.**Service**(*name*)
Bases: object

a single Service

name = None

requests = None

waiting = None

class raspy.servers.broker.**Worker**(*identity*, *address*, *lifetime*)
Bases: object

a Worker, idle or active

address = None

expiry = None

identity = None

service = None

raspy.servers.core module

class raspy.servers.core.**Core**(*hostname='localhost'*, *service='core'*, *broker_ip='127.0.0.1'*, *broker_port=5514*)
Bases: raspy.common.server.Server

The Core server

- **Cron**
  - Generate events in the publisher
- **Scenario**
  - a scenario can run in background at startup (ie thermostat) or fired by an event (ie cron, sun is down, temperature is under 0°C)
  - a scenario can be a loop so it must be launch in a separate thread : start filling, loop until water level is ok : need to call self._stopevent.isSet() in it so that the tread can shutdown.
  - look for updates in entries list (cron, sensors, variables in the publisher) : a list mapped in friendly user's names (using store)
  - it can publish some values with publisher
  - do some work using inline code python :
  - send commands to devices, cron jobs, start other scenario, update some variables in publisher
  - we can export/import scenarios : share with friends
- **NTP / Sytem Time / RTC Sync**
  - sync from ntp to rtc
  - sync from trc to system : using sudo with no password

worker_cron()
Create a worker to handle cron requests

worker_scenario()
Create a worker to handle scenario's requests

worker_scenarios()
Create a worker to handle scenarios requests (list_keys, ...)

class raspy.servers.core.**Cron**
Bases: object

A cron job

aps_job = None

class raspy.servers.core.**CronManager**
Bases: object

The manager of cron job

jobs = {}

class raspy.servers.core.**Scenario**(*name='scenar1'*, *publisher=None*)
Bases: threading.Thread

A scenario

code = None
The code we must exec

conf = {}
The configuration of the scenario

entries = {}
The entries we must look at for an event driven scenario

fire()
Check if the scenario must be fired using entries and that is not already running. If so, call sself.run()

Returns True if the thread must be launch (self.run()), False otherwise

Return type boolean

load(*store*)
Load the scenario from titanic store

Parameters **store** – the store to get info from

Type titani_store

Returns True if the scenario was loaded from store

Return type boolean

run()
Run the scenario

running = False
Is the scenario running

shutdown()
Shutdown the scenario

store(*store*)
Store the scenario to titanic store

class raspy.servers.core.**ScenarioManager**(*publisher=None*)
Bases: object

The manager of scenarios

http://etutorials.org/Programming/Python+tutorial/Part+III+Python+Library+and+Extension+Modules/Chapter+13.+Control
http://lucumr.pocoo.org/2011/2/1/exec-in-python/ http://late.am/post/2012/04/30/the-exec-statement-and-a-python-mystery

add(*name='scenar1'*, *entries={}*, *code=None*, *conf={}*)
Add a scenario

delete(*name='scenar1'*)
Delete a scenario

list()
Return all scenarios with conf, entries, ... as json dict

list_keys()
Return all scenarios key (=name) ... as json list

load(*store*)
Load the scenarios from titanic store

store keys :

- •scenario.main.conf : a json dict for configuration of scenario
- •scenario.main.keys : a json list of the scenario's names
- •scenario.key1.conf : a json dict for configuration of scenario key1
- •scenario.key1.entries : a json dict of entries of scenario key1
- •scenario.key1.code : a json string of code of scenario key1

scenarios = {}
The scenarios

shutdown()
Shutdown the scenario manager

store(*store*)
Store the scenarios to titanic store

update(*name='scenar1'*, *entries={}*, *code=None*, *conf={}*)
Update a scenario

raspy.servers.fake module

class raspy.servers.fake.**Fake**(*hostname='localhost'*, *service='fake'*, *broker_ip='127.0.0.1'*, *broker_port=5514*)
Bases: raspy.common.server.Server

A fake server to test RasPy

- •we must developp "real" fake device : ie a temperature sensors must not send random values
- •a cyclic sensor : parameters : cycle length, min, max and unit. Will do cycle from min temp to max temp (at cycle/2) and fall back tp min temp at end of if
- •a linear sensor
- •

worker_devices()
Create a worker to handle devices requests

raspy.servers.logger module

class raspy.servers.logger.**CompressedFile**(*logfile='log1.log'*, *mode='a+'*, *compresslevel=1*)
Bases: object

A compressed log file

close()

log(*level*, *message*)

open()

readlines(*start=0*, *end=-1*, *limit=20*)
Return lines from a file

rotate()

class raspy.servers.logger.**Graph**(*logfile='log1.log'*, *mode='a+'*, *compresslevel=1*)
Bases: object

A graph

class raspy.servers.logger.**Logger**(*hostname='localhost'*, *service='logger'*, *broker_ip='127.0.0.1'*, *broker_port=5514*, *data_dir='.rapsy'*)
Bases: raspy.common.server.Server

The logger server

Will log data, events, ... in files, rrd, ... It can be called via the worker or it can log data in pthe publisher

What to log :

- numeric : data for a device. We must be abble to aggregate data from multiple devices (ie a graph for inside/outqide temperature)
- text events : door open, notification, server log, ...
- images for webcam ??? large amount of data, not a good idea to tranport it using zmq. A ftp client which sync to a server.

How to log :

- RRDtool for nuerical values:
- http://segfault.in/2010/03/python-rrdtool-tutorial/
- we must use rrcached : https://github.com/pbanaszkiewicz/python-rrdtool/blob/master/rrdtool-1.4.7/etc/rrdcached-init
- Compressed text files for log :
- stream compression : http://pymotw.com/2/bz2/index.html#module-bz2
- file rotation
- http://pymotw.com/2/gzip/
- http://www.tutorialspoint.com/python/python_files_io.htm

How to distribute graph, text logs

- via a local directory. The http server will server them to the final client => raspyweb and the logger must be launch on the same server : NO
- via sync : add a ftp server service (in python or a a package : vsftp with xinet or in standalone) : Use a lot of bandwith, How to transfer log : every minutes ??? : NO
- add a simpleHttp server here which will serve file to the proxy (apache ? so that it will cache them).
- /graph/graphkey/day, /graph/graphkey/week, /graph/graphkey/month, /graph/graphkey/year
- /log/logkey

graphes = {}

shutdown()
Shutdown executive.

worker_graph()
Create a worker to handle graph requests

worker_log()
Create a worker to handle logger requests

class raspy.servers.logger.**RrdCachedClient**(*socket_path='/var/run/rrdcached.sock'*)
demonstration class only - coded for clarity, not efficiency

shutdown()
Shutdown the client

update(*rrdfile*, *rrdtime*, *msg*)

class raspy.servers.logger.**ThreadedTCPRequestHandler**(*request*, *client_address*, *server*)
Bases: SocketServer.BaseRequestHandler

The request handler

handle()

class raspy.servers.logger.**ThreadedTCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Bases: SocketServer.ThreadingMixIn, SocketServer.TCPServer

The simple HTTP server Be careful ... no security at all

logger = None
The logger used to retrieve data_dir, log and graph dictionnaries

raspy.servers.onewire module

class raspy.servers.onewire.**OneWire**(*hostname='localhost'*, *service='onewire'*, *broker_ip='127.0.0.1'*, *broker_port=5514*, *devices_dir='/sys/bus/w1/devices'*)
Bases: raspy.common.server.Server

The OneWire server

Configuration

You need to load kernel module :

```
sudo vim /etc/modules
```

```
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.
# Parameters can be specified after the module name.
w1-therm
w1-gpio pullup=1
i2c-dev
i2c-bcm2708
spi-bcm2708
snd-bcm2835
```

And check that blacklist is correct :

```
sudo vim /etc/modprobe.d/raspi-blacklist.conf
```

```
# blacklist spi and i2c by default (many users don't need them)
blacklist spi-bcm2708
blacklist i2c-bcm2708
blacklist snd-soc-pcm512x
blacklist snd-soc-wm8804
```

At last, we must load the module in init script sothat we don't need to update this.

From https://www.modmypi.com/blog/ds18b20-one-wire-digital-temperature-sensor-and-the-raspberry-pi

worker_devices()
Create a worker to handle devices requests

raspy.servers.sync module

class raspy.servers.sync.**Sync**(*hostname='localhost'*, *service='sync'*, *broker_ip='127.0.0.1'*, *broker_port=5514*)
Bases: raspy.common.server.Server

The Sync server

Sync data from a or many folders (which we can configure via zmq) to a remote server.

Used by logger, camera, ...

Sync can be sheduled (ie every day, ...) or laucnh on demand via worker We can sync a file or a directory

run()
Sync data in a separate thread

worker_sync()
Create a worker to handle sync requests

raspy.servers.titanic module

class raspy.servers.titanic.**Titanic**(*hostname='localhost'*, *service='titanic'*, *broker_ip='127.0.0.1'*, *broker_port=5514*, *data_dir='/tmp/raspy'*)
Bases: raspy.common.executive.Executive

The Titanic helper

Also integrates a store for keys/values

From http://zguide.zeromq.org/py:all#Disconnected-Reliability-Titanic-Pattern
http://zguide.zeromq.org/py:all#Service-Oriented-Reliable-Queuing-Majordomo-Pattern
https://github.com/imatix/zguide/tree/master/examples/Python

reply_filename(*uuid*)
Returns freshly allocated reply filename for given UUID

request_filename(*uuid*)
Returns freshly allocated request filename for given UUID

run()
Run the hub

service_success(*client*, *uuid*)
Attempt to process a single request, return True if successful

store_filename(*service*)
Returns store filename for given service

titanic_close()
Create a worker to handle titanic.close

titanic.close: confirm that a reply has been stored and processed.

titanic_reply()
Create a worker to handle titanic.service

titanic.reply: fetch a reply, if available, for a given request UUID.

titanic_request(*pipe*)
Create a worker to handle titanic.request

titanic.request: store a request message, and return a UUID for the request.

titanic_store()
Create a worker to handle store services

Module contents

4.2 Module contents

CHAPTER 5

raspyweb package

5.1 Subpackages

5.1.1 raspyweb.app package

Subpackages

raspyweb.app.ajax package

Submodules

raspyweb.app.ajax.constants module The main views

raspyweb.app.ajax.models module The main views

raspyweb.app.ajax.views module The main views

raspyweb.app.ajax.views.**devices**()

raspyweb.app.ajax.views.**home**()

raspyweb.app.ajax.views.**mmi**()

Module contents

Submodules

raspyweb.app.views module

The main views

raspyweb.app.views.**home**()

raspyweb.app.views.**not_found**(*error*)

Module contents

RasPyWeb app module.

Use templates : https://pythonhosted.org/Flask-Themes/

raspyweb.app.**install_secret_key**(*app*, *filename='secret_key'*)
Configure the SECRET_KEY from a file in the instance directory.

If the file does not exist, print instructions to create it from a shell with a random key, then exit.

5.2 Submodules

5.3 raspyweb.config module

Storing all the module configurations. Here, the database is setup to use SQLite, because it's a very convenient dev env database. Most likely /config.py won't be a part of your repository and will be different on your test and production servers.

- _basedir is a trick for you to get the folder where the script runs
- DEBUG indicates that it is a dev environment, you'll get the very helpful error page from flask when an error occurs.
- SECRET_KEY will be used to sign cookies. Change it and all your users will have to login again.
- ADMINS will be used if you need to email information to the site administrators.
- SQLALCHEMY_DATABASE_URI and DATABASE_CONNECT_OPTIONS are SQLAlchemy connection options (hard to guess)
- THREAD_PAGE my understanding was 2/core... might be wrong :)
- CSRF_ENABLED and CSRF_SESSION_KEY are protecting against form post fraud
- RECAPTCHA_* WTForms comes with a RecaptchaField ready to use... just need to go to recaptcha website and get your public and private key.

Credits : https://github.com/mitsuhiko/flask/wiki/Large-app-how-to

class raspyweb.config.**Config**
Bases: object

ADMINS = frozenset(['bibi21000@gmail.com'])

BROKER_IP = '127.0.0.1'

BROKER_PORT = 5514

CSRF_ENABLED = True

CSRF_SESSION_KEY = 'somethingimpossibletoguess'

DATABASE_URI = 'sqlite://:memory:'

DEBUG = False

RECAPTCHA_OPTIONS = {'theme': 'white'}

RECAPTCHA_PRIVATE_KEY = '6LeYIbsSAAAAAJezaIq3Ft_hSTo0YtyeFG-JgRtu'

RECAPTCHA_PUBLIC_KEY = '6LeYIbsSAAAAACRPIllxA7wvXjIE411PfdB2gt2J'

RECAPTCHA_USE_SSL = False

SECRET_KEY = 'This string will be replaced with a proper key in production.'

TESTING = False

THREADS_PER_PAGE = 8

class raspyweb.config.**DevelopmentConfig**
Bases: raspyweb.config.Config

DEBUG = True

TESTING = True

class raspyweb.config.**ProductionConfig**
Bases: raspyweb.config.Config

DATABASE_URI = 'mysql://user@localhost/foo'

class raspyweb.config.**TestingConfig**
Bases: raspyweb.config.Config

TESTING = True

5.4 raspyweb.run module

Used to launch the web server.

Credits : https://github.com/mitsuhiko/flask/wiki/Large-app-how-to

raspyweb.run.**main**()

5.5 raspyweb.shell module

will allow you to get a console and enter commands within your flask environment. Maybe not as nice as debugging with pdb, but always useful (when you will initialize your database).

Credits : https://github.com/mitsuhiko/flask/wiki/Large-app-how-to

5.6 Module contents

r

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z