Python Testing Infrastructure Documentation

Release 0.1

The PyTI team

April 12, 2016

Contents

The python testing infrastructure is a Continuous Integration tool able to test python distributions availabe at PyPI against a set of tests, providing a set of metrics.

Why ? The python package index doesn't currently checks the uploaded distributions against anything and thus anyone can upload what he wants, without having any kind of control.

PyTI does'nt try to enforce quality on PyPI, but aims to provide a global overview of the health of python distributions available there.

The tests range from installation (is everything installable smoothly? does it install files in weird looking places?) to the detection of harmful/malicious behaviours. It is also possible to run test suites and run quality checking tools such as PEP8 or PyLint.

Actually, the possibilities are not limited and we tried to make something easy to extend and to use.

PyTI is split into different sub-project: goatlib, the task manager, which handles task scheduling, goatlog which handle execution reporting, and pythia which handles the virtual machines management.

CHAPTER 1

Documentation

1.1 Overall design of the slave

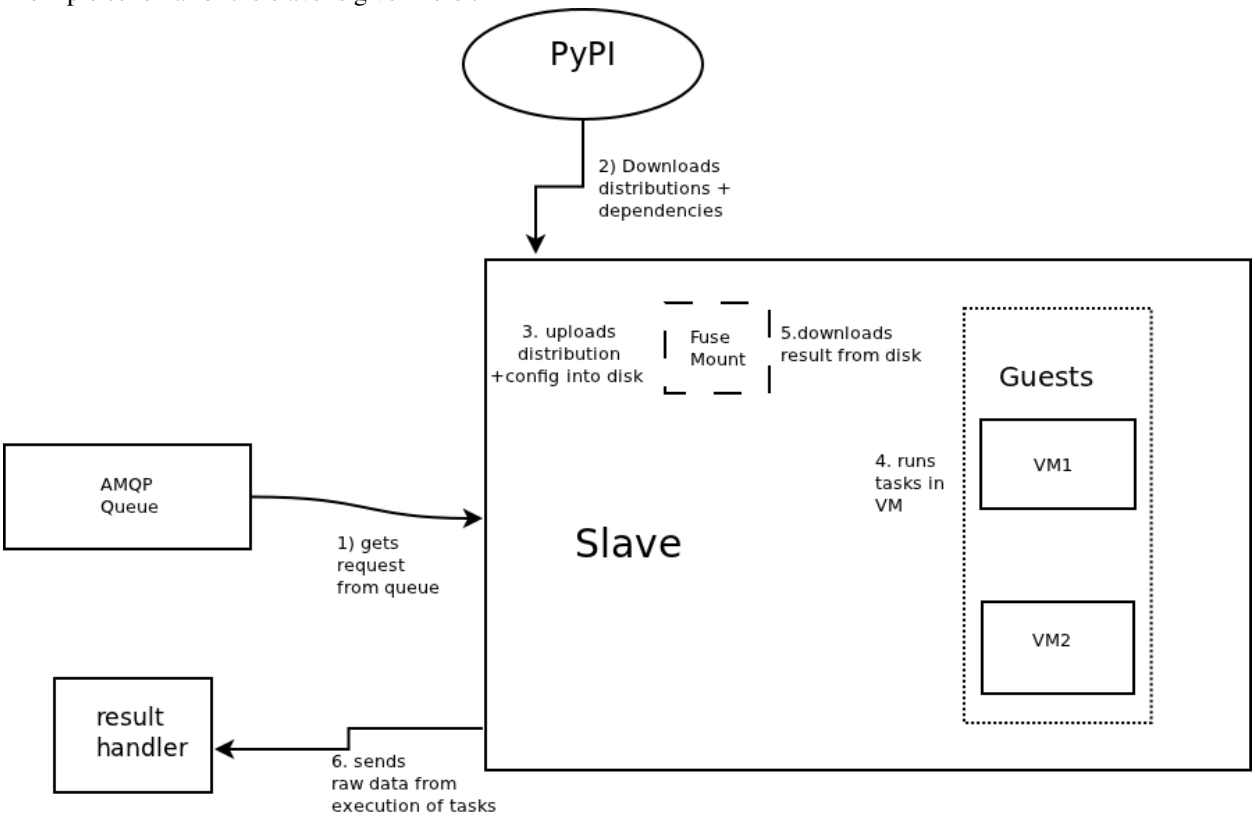
This part of the Python testing infrastructure is able to:

prepare distributions and send them to virtual machine for execution # launch a series of tasks on the virtual machine # get back information about those tasks (output of those)

In other words, it is the implementation of one slave in a master/slave architecture. It is able to manage a set of different machines, get back results and rollback the VM in a known state.

Pretty much, the purpose of the slave boils down to testing the projects (as requested by master) in an isolated environment(VM).

A simple schema for the slave is given here :



All VM operations like starting/stopping rollback and snapshot is discussed in vms and all disk operations like copying into disk and viceversa is discussed in diskhandler

Launching the slave:

```
$ pyti-slave --config pyti_config
```

This launches a deamon for the slave to receive requests from the master and run the tests as per the requests. The pyti_config file is an optional configuration file which contains basic information like:

```
[config]
pyti_slave_root = /home/yeswanth/pyti
total_vms = 3
```

The above config file refers to the environment variable PYTI_SLAVE_ROOT and the total VMs supported by the VMs

The slave then listens to orders on it's queue 'pyti_requests' as described in master

You also can run specific tasks on specific distributions by asking so:

Inorder to get the last version of the "foobar" project and run all tests on it,we need to provide a json file which contains project details. Here again the config file is optional:

```
$ pyti-vm-run --target json_file --config config_file
```

A sample json can for "foobar" project that represents which platforms it needs to run on:

```
{
 "project_name": "rarfile",
 "project_version": "1.0",
 "recipe_identifier": "recipe",
 "platform_identifier": "linux-debian",
 "platform_variants": Null
}
```

Internally, this means that the following things are done:

- load all the connectionz to the VMs, know what is the state for each of them (stopped, processing, ...) For this number of VMs available at any given time is stored . This is handled by *pyti.slave.stateofvm*
- get a spare VM
- download the wanted distribution, get its dependencies, save all of that on a IO disk. This is handled by *pyti.slave.pypidownload*
- start the VM, wait until the tasks are all run.
- read the content of the IO disk in a particular folder
- put those results in a python dict
- stop the vm + rollback it
- return the results

1.2 Installing Dependencies in slave

Please install in the following order.

1.2.1 VirtualBox

Virtualbox can either be downloaded directly at virtualbox . It is also packaged in Ubuntu/Debian in the name of *virtualbox-ose*. A simple apt-get command will be able to fetch the packages and install them.

1.2.2 Libvirt

Before installing libvirt make sure you have already installed python-dev. It is packaged under the same name in Ubuntu

From Source

Libvirt source packages are available at source . Download the package and extract it . Run the following commands after that

```
$ ./configure --with-vbox --with-python
```

This should not print any error. Also check that the bindings for vbox and python are added (It prints the report at the end, A simple "Yes" for python and vbox is sufficient to know that it is added)

```
$ sudo make
$ sudo make install
$ sudo ldconfig -v
```

The above two commands installs libvirt . After this a run of ldconfig or a similar utility is sufficient.

From Binaries

Libvirt in Ubuntu does not have Virtualbox support by default. So we will have to compile it from source to include Virtualbox driver and create the binaries as discussed in this forum. It is recommended to install libvirt from source (the above method)

1.2.3 VDFuse

VDfuse is required to mount the virtual hard disk on to the host machine. VDfuse is packaged in Ubuntu and Debian with the package name *virtualbox-ose-fuse*. A simple apt-get command will be able to fetch the package and install it. Inorder to make vdfuse work for every user

/etc/fuse.conf file has to be edited to add 'user_allow_other':

```
$ sudo echo 'user_allow_other'>>/etc/fuse.conf
```

1.2.4 Fuse-ext2

Fuse-ext2 is required to mount a partition on user space. It is packaged in Debian and Ubuntu with the name *fuseext2*.

1.3 Handling Virtual Machines (VMs)

The *pyti.vms* modules handles all the VM features, such as start, stop, rollback etc. It is the central and prefered way to control the different virtual machines on the Python Testing Infrastructure.

Its original goal is to be used on the slaves of the master/slave architecture. It is able to: mount different disks to a VM, start it, wait until it stops and return information about the changes that have been made on the filesystem.

1.3.1 Terminology

The terminology used here is mainly coming from libvirt. Here are some definitions, you can refer yourself to the libvirt documentation for more information.

1.3.2 VirtualMachine class

To deal with virtual machines, you will use the VirtualMachine class.

To initialize a VirtualMachine object include VM name, Connection object(which represents the hypervisor VirtualBox), configuration by either an XML file or by a python dictionary:

```
>>> VirtualMachine('arch linux #1', "vbox:///session", config=config)
```

- For XML file , libvirt defined XML has to be used http://libvirt.org/formatdomain.html

You can specify it using the *xmlfile* argument:

```
>>> VirtualMachine('arch linux #1', "vbox:///session",
        xmlfile="yourfile.xml")
```

For config as a dictionary, you should specify:

- a name
- the memory
- the disk location (of the iso file) (optional parameter for live boot)
- the hard disk location(of the virtual hard disk file)

For instance:

```
config = {
    'name': 'virtual machine A',
    'memory': '128',
    'disk_location': 'dsl.iso',
    'hd_location':'disk.vdi'
}
```

Example

The following example starts a Virtual Machine, creates a snapshot and rollbacks:

```
>>> from vms import *
>>> config = {'name': 'test123', 'memory':'123', \
          'disk_location': 'dsl-4.4.10.iso', \
          'hd_location': 'disk.vdi'}
>>> a = VirtualMachine('hey', "vbox:///session", config=config)
>>> a.start()
>>> a.createSnapshot('hello', 'blah')
>>> a.rollback('hello')
```

1.4 Handling Virtual Hard Disk for Host (diskhandler)

The *pyti.diskhandler* modules helps in transferring data from the host machine to the guest and viceversa. Its main objective is to upload the distributions to the guest for testing and download the raw data on the host (after the tests are conducted in the guest i.e the VM).

1.4.1 DiskOperations class

To initialize DiskOperations object we need to include the disk_path of the virtual hard disk which we want to access along with the two mount points (one for fusing the virtual hard disk onto the host to a recognizable format and another for actually mounting the virtual hard disk)

Example

The following example mounts a partition from the disk , uploads a file to the disk , also downloads a file from the disk:

```
>>> from diskhandler import *
>>> d = DiskOperations('/home/yeswanth/a.vdi', '/mnt/fuse', '/mnt/guest')
>>> d.upload("/home/yeswanth/a.txt", "/home/pyti/a.txt")
>>> d.download("/home/pyti/b.txt", "/home/yeswanth/b.txt")
>>> d.close_connection()
```

1.5 Master Slave Communication

This part of Python Testing Infrastructure talks about how master sends job requests to the slave and how slave picks up job request

1.5.1 Queue

The communication between master and slave is done using AMQP protocol implemented by RabbitMQ server. The Queue is asynchronous and persistent(i.e job requests will still remain in the queue even if Master shutdowns)

1.5.2 Master

Master processes the PyPI request into meaningful job requests so that slave can be able to read it and execute it easily. So master looks up at setup.cfg file of the PyPI project and writes a config which consists of the project name, project version, platform to be run on, platform variants (which means what does the VM require for the project to run , for eg some projects may need MySQL server) and recipe identifier(which defines what tests have to be conducted on the project)

Master then converts this config into JSON format and puts it in the Queue::

```
    >>> from pyti.dispatcher.dispatcher import Queue, Serializer
    >>> s = Serializer('demo_project', '1.0', 'debian', None, 'recipe')
    >>> data = s.serialize_data()
    >>> q = Queue('demo')
    >>> q.send(data)
```

1.5.3 Slave

The slave side is more sophisticated than the master. A slave can have multiple VMs of multiple platforms. So each time it looks up in the Queue for wanting a job, it has to take care of VMs of different platforms running in the slave. So when slave requires jobs, it looks up in the Queue if there are available jobs and once it gets the job request , it has to check if VMs for the platform to be tested on are busy or not. If they are busy, slave releases the job request back to the Queue:

```
>>> from pyti.slave.manager import Queue
>>> q = Queue('demo')
>>> q.listen()
```