Tunir Documentation

Release 0.17

Kushal Das

Jul 24, 2017

Contents

Tunir is a simple testing tool. The goal is to have a system which is simple to setup, and easy to maintain.

Note: Please use the gotun project if you want to spin up your instances on AWS or OpenStack.

Contents:

CHAPTER 1

Why another testing tool?

I have used Jenkins before. I was maintaining one instance in one of my VPS instance. The amount of RAM required by Jenkins was too much for my small VM. I can admit that I am not a great sys-admin anyway.

As part of my daily job, I have to test the latest cloud images we build under Fedora project. While doing so, I figured out that most of it can be automated if we have a system to create/maintain/terminate cloud instances. Of course I do not want any actual cloud, it will be a different monster to maintain.

This is the point where I came up with Tunir. Tunir is a simple testing tool that will help me run automated tests for the cloud images. I kept the system generic enough to execute any kind of tests people want.

The configuration is very minimal with Tunir. There is also a golang verion called gotun which has better option to run the tests inside OpenStack or AWS.

CHAPTER 2

Installation

Tunir is written in Python. Currently it works with Python 3.5+

Clone the repository

```
$ git clone https://github.com/kushaldas/tunir.git
```

Install the dependencies

We are currently depended on the following projects or libraries.

- libvirt
- libguestfs
- libguestfs-tools
- ansible
- paramiko
- vagrant-libvirt
- pycrypto
- net-tools
- typing
- python-systemd (python2-systemd package in Fedora)
- Ansible (optional)
- libcloud

You can install them in Fedora by the following command:

```
$ sudo dnf install libguestfs-tools python3-paramiko docker-io vagrant-libvirt␣
↪ansible net-tools python3-crypto  python3-systemd python3-libcloud
```

Note: Remember to install python3-systemd package using dnf only

CHAPTER 3

Usage

Tunir is a mini continuous integration (CI) system which can run a set of commands/tests in a new cloud VM, or bare metal, or in Vagrant boxes based on the job configurations.

The current version can be used along with cron to run at predefined times. Tunir prints the output in the terminal, it also saves each command it ran, and the output in a text file located at '/var/run/tunir/tunir_results.txt'.

Configuring a new job

There are two different kinds of job configuration files, the newer one is Multi-VM config which can take any qcow2 image and use them to boot up one or more VMs. The other option is to use a JSON file based configuration which can be used for vm(s), vagrant images, or bare metal remote system based testing.

For a Multi-VM configuration for a job called **default** create **default.cfg** file as explained below. We will also require another **default.txt** file which will contain the steps for testing.

jobname.cfg

New in version 0.14.

The following example contains a job where we are creating two VMs from the given image files. The images can be either standard cloud image, or Atomic image. We generate ssh keys for each run, and use that to login to the box.

```
[general]
cpu = 1
ram = 1024

[vm1]
user = fedora
image = /home/Fedora-Cloud-Base-20141203-21.x86_64.qcow2

[vm2]
```

```
user = fedora
image = /home/Fedora-Cloud-Base-20141203-21.x86_64.qcow2
```

The above configuration file is self-explanatory. Each of the vm(s) created from the above configuration will get all the other vms' IP details in the */etc/hosts* along with vm name. Means *vm1* can ping *vm2* and vice versa. For each run, Tunir creates a new RSA key pair and pushes the public key to each vm, and uses the private key to do ssh based authentication.

How to execute a multivm job?

```
$ sudo tunir --multi jobname
```

The above commands expects a **jobname.cfg**, and a **jobname.txt** containing the commands, in the current directory. You can see below for an example of **jobname.txt**.

Debugging test vm(s)

New in version 0.14.

This can also be used a quick way to get a few vm(s) up. While using Multi-VM configuration, one can pass **–debug** command line argument, and this will make sure that the vm(s) do not get destroyed at the end of the tests. It will create a *destroy.sh* file, and print the path at the end of the run. All the vm(s) will be in running condition. You can ssh into them by using *private.key* file found in the same directory of the *destroy.sh*.

When your debugging is done, you can execute the shell script to clean up all the running instances and any temporary file created by the previous run.

```
# sh /tmp/tmpXYZ/destroy.sh
```

> **Warning:** The private key remains on the disk while running Tunir in the debug mode. Please remember to execute the destroy.sh script to clean up afterwards.

jobname.json

This file is the main configuration for the job when we just need only one vm, or using Vagrant, or testing on a remote vm/bare metal box. Below is the example of one such job.

```
{
  "name": "jobname",
  "type": "vm",
  "image": "/home/vms/Fedora-Cloud-Base-20141203-21.x86_64.qcow2",
  "ram": 2048,
  "user": "fedora",
}
```

The possible keys are mentioned below.

name The name of the job, which must match the filename.

type The type of system in which the tests will run. Possible values are vm, docker, bare.

image Path to the cloud image in case of a VM. You can provide docker image there for Docker-based tests, or the IP/hostname of the bare metal box.

ram The amount of RAM for the VM. Optional for bare or Docker types.

user The username to connect to.

password The password of the given user. Right now for cloud VM(s) connect using ssh key.

key The path to the ssh key, the password value should be an empty string for this.

port The port number as string to connect. (Required for bare type system.)

jobname.txt

This text file contains the bash commands to run in the system, one command per line. In case you are rebooting the system, you may want to use **SLEEP NUMBER_OF_SECONDS** command there.

If a command starts with @@ sign, it means the command is supposed to fail. Generally, we check the return codes of the commands to find if it failed, or not. For Docker container-based systems, we track the stderr output.

We can also have non-gating tests, means these tests can pass or fail, but the whole job status will depend on other gating tests. Any command in jobname.txt starting with ## sign will mark the test as non-gating.

Example:

```
## curl -O https://kushal.fedorapeople.org/tunirtests.tar.gz
ls /
## foobar
## ls /root
##  sudo ls /root
date
@@ sudo reboot
SLEEP 40
ls /etc
```

POLL directive

New in version 0.17.

We also have a *POLL* directive, which can be used to keep polling the vm for a successful ssh connection. It polls after every 10 seconds, and timeout is currently set for 300 seconds. One should this one instead of *SLEEP* directive after a reboot.

HOSTCOMMAND directive

New in version 0.18.

Now we have *HOSTCOMMAND* directive, which can be used to run any command in the host system itself. One major usecase for this directive wil be for generating ansible inventory file using a simple script (local). The tests will continue even if this command fails to execute properly.

HOSTTEST directive

New in version 0.18.

Now we also have the *HOSTTEST* directive, which will allow us to execute a command in the host, and count that as a part of the tests. Ansible usage is the best example for this directive.

For Multi-VM configurations

New in version 0.14.

In case where we are dealing with multiple VMs using .cfg file in our configuration, we prefix each line with the vm name (like vm1, vm2, vm3). This marks which command to run on which vm. The tool first checks the available vm names to these marks in the *jobname.txt* file, and it will complain about any extra vm marked in there. If one does not provide vm name, then it is assumed that the command will execute only on vm1 (which is the available vm).

```
vm1 sudo su -c"echo Hello > /abcd.txt"
vm2 ls /
vm1 ls /
```

In the above example the line 1, and 3 will be executed on the vm1, and line 2 will be executed on vm2.

Using Ansible

Note: If you want to run Ansible playbooks in your test, please have a look at the gotun project, it has better support for running Ansible, or any other tool in the host as the part of the test.

New in version 0.14.

Along with Multi-VM configuration, we got a new feature of using Ansible to configure the vm(s) we create. To do so, first, create the required roles, and playbook in a given path. You can write down the group of hosts with either naming like *vm1*, *vm2*, *vm3* or give them proper names like *kube-master.example.com*. For the second case, we also have to pass these hostnames in each vm definition in the configuration file. We also provide the path to the directory containing all ansible details with *ansible_dir* value.

Example configuration

```
[general]
cpu = 1
ram = 1024
ansible_dir = /home/user/contrib/ansible

[vm1]
user = fedora
image = /home/user/Fedora-Cloud-Atomic-23-20160308.x86_64.qcow2
hostname = kube-master.example.com

[vm2]
user = fedora
image = /home/user/Fedora-Cloud-Atomic-23-20160308.x86_64.qcow2
hostname = kube-node-01.example.com

[vm3]
```

```
user = fedora
image = /home/user/Fedora-Cloud-Atomic-23-20160308.x86_64.qcow2
hostname = kube-node-02.example.com
```

In the above example, we are creating 3 vm(s) with given hostnames.

Note: If the number of CPU is not mentioned in the general section, Tunir will get 1 virtual CPU for the vm.

How to execute the playbook(s)?

In the *jobname.txt* you should have a **PLAYBOOK** command as given below

```
PLAYBOOK atom.yml
vm1 sudo atomic run projectatomic/guestbookgo-atomicapp
```

In this example, we are running a playbook called *atom.yml*, and then in the vm1 we are using atomicapp to start a nulecule app :)

Execute tests on multiple pre-defined VM(s) or remote machines

```
[general]
cpu = 1
ram = 1024
ansible_dir = /home/user/contrib/ansible
pkey = /home/user/.ssh/id_rsa

[vm1]
user = fedora
ip = 192.168.122.100

[vm2]
user = fedora
ip = 192.168.122.101

[vm3]
user = fedora
ip = 192.168.122.102
```

Example of configuration file to run the tests on a remote machine

The configuration:

```
{
  "name": "remotejob",
  "type": "bare",
  "image": "192.168.1.100",
  "ram": 2048,
  "user": "fedora",
  "key": "/home/password/id_rsa"
```

```
  "port": "22"
}
```

Start a new job

```
$ sudo ./tunir --job jobname
```

Job configuration directory

You can actually provide a path to tunir so that it can pick up job configuration and commands from the given directory.:

```
$ sudo ./tunir --job jobname --config-dir /etc/tunirjobs/
```

Timeout issue

In case if one of the commands fails to return within 10 minutes (600 seconds), tunir will fail the job with a timeout error. It will be marked at the end of the results. You can change the default value in the config file with a timeout key. In the below example I am having 300 seconds as timeout for each command.:

```
 {
  "name": "jobname",
  "type": "vm",
  "image": "file:///home/vms/Fedora-Cloud-Base-20141203-21.x86_64.qcow2",
  "ram": 2048,
  "user": "fedora",
  "password": "passw0rd",
  "timeout": 300

}
```

CHAPTER 4

Using Vagrant jobs

Vagrant is a very well known system among developers for creating lightweight development systems. Now from tunir 0.7 we can use Vagrant boxes to test. In Fedora, we can have two different kind of vagrant provider, libvirt, and virtualbox.

Warning: The same host can not have both libvirt and virtualbox.

Note: Please create /var/run/tunir directory before running vagrant jobs.

How to install vagrant-libvirt?

Just do

```
# dnf install vagrant-libvirt
```

The above command will pull in all the required dependencies.

How to install Virtualbox and vagrant?

Configure required virtualbox repo

```
# curl http://download.virtualbox.org/virtualbox/rpm/fedora/virtualbox.repo > /etc/
↪yum.repos.d/virtualbox.repo
# dnf install VirtualBox-4.3  vagrant -y
# dnf install kernel-devel gcc -y
# /etc/init.d/vboxdrv setup
```

Now try using *–provider* option with vagrant command like

```
# vagrant up --provider virtualbox
```

Example of a libvirt based job file

```
{
  "name": "fedora",
  "type": "vagrant",
  "image": "/var/run/tunir/Fedora-Cloud-Atomic-Vagrant-22-20150521.x86_64.vagrant-
↪libvirt.box",
  "ram": 2048,
  "user": "vagrant",
  "port": "22"
}
```

Example of a Virtualbox based job file

```
{
  "name": "fedora",
  "type": "vagrant",
  "image": "/var/run/tunir/Fedora-Cloud-Atomic-Vagrant-22-20150521.x86_64.vagrant-
↪virtualbox.box",
  "ram": 2048,
  "user": "vagrant",
  "port": "22",
  "provider": "virtualbox"
}
```

Note: We have a special key provider in the config for Virtualbox based jobs.

CHAPTER 5

AWS support

Note: Please use the gotun project if you want to spin up your instances on AWS or OpenStack.

Now we have support to use AWS for testing using Tunir. We can have both HVM, and paravirtual types of instances to run the test. You will require Python libcloud for the same.

Note: It boots up the instances in us-west-1 zone.

Example of HVM

The following is a JSON file containing the config of a HVM instance.

```
{
  "name": "awsjob",
  "type": "aws",
  "image": "ami-a6fc90c6",
  "ram": 2048,
  "user": "fedora",
  "key": "PATH_TO_PEM",
  "size_id": "m3.2xlarge",
  "access_key": "YOUR_ACCESS_KEY",
  "secret_key": "YOUR_SECRET_KEY",
  "keyname": "YOUR_KEY_NAME",
  "security_group": "THE_GROUP_WITH_SSH",
  "virt_type": "hvm",
  "timeout": 30
}
```

Warning: Remember that m3 instances are capable of running HVM.

Example of paravirtual

Another example with paravirtual type of instance.

```
{
  "name": "awsjob",
  "type": "aws",
  "image": "ami-efff938f",
  "ram": 2048,
  "user": "fedora",
  "key": "PATH_TO_PEM",
  "size_id": "m1.xlarge",
  "access_key": "YOUR_ACCESS_KEY",
  "secret_key": "YOUR_SECRET_KEY",
  "keyname": "YOUR_KEY_NAME",
  "security_group": "THE_GROUP_WITH_SSH",
  "virt_type": "paravirtual",
  "aki": "aki-880531cd",
  "timeout": 30
}
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search