Get Familiar with Remote Execution and Job Orchestration

It is possible to perform remote execution and job orchestration on a number of virtual machines with software like Fabric and Ansible, as this article explains.

Modern computing environments are highly distributed in nature and comprise many physical and/or virtual machines. In fact, with the advent of cloud computing in the last few years, most of the infrastructure has gone virtual with the running of a large number of VMs (virtual machines) to cater to the needs of applications. The need to execute commands on a number of machines is a very common day-to-day requirement for any modern infrastructure. If the operating system running on the machines is GNU/Linux, which is most often the case these days, then SSH sessions are used to execute the desired remote commands. The SSH command provided by various GNU/Linux distributions, MacOS X and Windows (you need to install the very popular Putty SSH client here) is the most universal way to SSH into remote machines (requires sshd running and accepting SSH connections from outside, which is the default in almost all GNU/Linux distributions) in an interactive or batch mode. Using the interactive SSH sessions is fine when working with a few remote machines, but is not suitable when the need is to remote execute on a number of machines without any human intervention.

The good news is that modern dynamic languages like Python, Ruby and even Perl have some good libraries and frameworks around SSH to create feature-rich remote execution utilities to manage hundreds and thousands of machines with less effort. Fabric is a Python library and command line tool around SSH, which enables remote serial or parallel execution of commands on machine. It provides features like retrying multiple times with some delay in between, skipping unresponsive machines as well. You could also use the Fabric API to create many standalone remote execution tools in Python, taking advantage of the capabilities of both the Python standard library, extra modules and Fabric.

Ansible is the big daddy of Fabric, and it is a very popular choice currently, in the IT world. It is a full-fledged but no-fuss cloud provisioning, configuration management and job orchestration framework created in Python. Ansible further

abstracts the remote execution activity in higher level code chunks to control the remote machines intelligently, at scale, using an easy-to-learn modelling language.

The examples provided in this article are based on Ubuntu 14.04 LTS and MacOS X.

The traditional *NIX way through SSH

The basic structure of the *bash* command to remotely execute commands is shown below. The option *-tt* is required in case of the *sudo* command.

```
ssh -tt -oStrictHostKeyChecking=no user@<remote hostname that
is resolvable or IPv4 address> 'command(s) to run, separated
by ; or bunched together by && or || bash operators>'
```

This command presents a prompt to enter a password in case the remote login authentication requires it. If you use key-pair authentication, as is prevalent in the world of cloud based virtual machines, then you need to append *-i <private keypair file>* to the command as well. Typing the password for the remote authentication becomes cumbersome if you need to remote execute on a number of machines. You can use a utility known as *sshpass* to handle the password entry part through the command line, a file having that password or an environment variable. You can install *sshpass* through *sudo apt-get install -y sshpass* or compile that by downloading the tarball, extracting it and executing the command *sudo apt-get install -y build-essential && ./configure && sudo make install* in the source directory. The usage of *sshpass* with *ssh* is shown below, and you can see its *help* menu through *sshpass -h* to know more about the other options it provides:

```
sshpass -p '<password>' ssh <all the remaining options and
arguments>
```

The SSH commands need to be further wrapped in bash scripts to create home grown remote execution utilities that parse users, hostnames, IPs, commands, etc, from some configuration files and to run the remote execution in parallel. The advantage of this approach is that you don't need any extra dependency to start with, as bash is the de-facto option on almost every GNU/Linux distribution. So the development of the bash wrappers around the SSH command is a very quick way to start with. But the disadvantage of the bash approach is that the bash wrapper becomes complicated and hard to manage when you need to have a more intelligent tool. Features like retrying multiple times with some delay in between, skipping unresponsive machines, discovering machines at runtime, etc, complicate the tool development efforts a lot when using the raw SSH and bash approach.

The 'Python way' through Fabric

Job orchestration is the term very commonly used for so many functions in cloud environments. But I am using the term here to denote activities that are performed remotely in any cloud set-up. I'm denoting the activities to be performed on the remote VMs as jobs, and a job could consist of a very simple command or a complicated workflow of a number of simple/complicated commands. So when using the term 'job orchestration' in this article, I mean 'trigger and manage various related or unrelated remote and automated activities' to achieve some end result in a cloud set-up. For example, running configuration management tools jobs to provision VMs out-of-the-box, patching a group of VMs through remote jobs, monitoring/verifying VMs in an agentless manner through remote dump jobs, etc.

As mentioned in the earlier section, there is a need to use more general and user-friendly tools and frameworks rather than a raw bash based approach if we need to automate remote job execution in all kinds of cloud set-ups. Fabric is that kind of programmable framework that encapsulates all complications around using SSH to create jobs for automatic remote execution in a serial or parallel manner. Fabric is Python based and that means it can do everything possible through the data structures, constructs, standard libraries and external modules available in the language. Also, Python is one of the core components available, by default, in almost every GNU/Linux distribution. This means you only need to set up the Fabric components, and everything is set.

Fabric is a non-centralised framework. So you need to install it on one or more virtual/physical machines from where you need to trigger the jobs for your remote machines. The recommended way to install Fabric is to use the *pip* command on GNU/Linux and MacOS X. There is a version of Pip for Windows as well. You also need to download and install Python and Pycrypto (a version that matches the Python version) on Windows, from the links provided in the Reference section at the end of the article. The command to install Fabric using Pip is (remove the *sudo* part of the command on Windows):

```
sudo pip install -U fabric
```

Note: Based on my experience on Ubuntu 14.04 LTS, I would recommend installing Pip through *easy_install* using the command *sudo easy_install pip* and not through the Python-Pip package (run the command *sudo apt-get install -y python-setuptools* to get *easy_install*). Then run the command *sudo apt-get install -y python-dev build-essential && sudo pip install -U fabric* to build and install Fabric. Pip wasn't installed on my MacOS X laptop so I had to install it using *sudo easy_install pip* (*easy_install* was already available there). Also, installing Fabric through Pip failed on MacOS X initially due to the non-matching version of the Paramiko library. It required me to install the right version of Paramiko using the following command:

```
sudo pip uninstall fabric paramiko && sudo pip install
paramiko==<version shown in the error> && sudo pip install
fabric
```

Now, it's time to dirty our hands with Fabric, since it has been successfully installed, and verify it by running *fab -h* on the command line console, which produces some usage output. Fabric requires a file named *fabfile.py*, by default, to read the job definitions and their settings to trigger them for remote execution. Fabric jobs are Python sub-routines, so they could be as simple as running a simple command or could be complicated provisioning workflows. Shown below is a code snippet, which is a basic template of *fabfile*. It consists of a routine to dump *disk_usage*, information on processing cores, total runtime memory available and the version of Python installed:

```
from fabric.api import env, run, sudo, get, put

env.warn_only = True
env.skip_bad_hosts = True
env.colorize_errors = True
env.connection_attempts = 3

def dump():
  try:
    run('df -kh')
    run('grep -i processor /proc/cpuinfo')
    run('grep -i memtotal /proc/meminfo')
    run('python --version')

  except Exception, e:
    print(" Error: %s" %e)
```

You could view the job(s) to be remotely executed in *fabfile* by issuing the *fab -l* command and initiate those jobs on a number of remote servers using the command: *fab <Job> -H <server1>,<server2>,...,<serverN> -u <common user> -p <common password>*. You could also use a key pair instead of the password, using the option *-i* and run the *fabfile* routine(s) in parallel, with the option *-P* (the parallel execution is broken on Windows due to multi-processing and issues with Pickle modules there). In fact, the *fab* command provides a lot of options to run your job orchestration in a number of ways, and running the command *fab -h* gives users the help regarding all the options supported.

If it seems too much typing on the console and your options are fixed, then you could mention everything in the *fabfile*. You could also group different servers based upon the functionalities they offer, and then run similar or different jobs on different groups of servers. The *fabfile* shown below is more advanced, and all the options are set within it (note the use of *put/sudo/get* in various jobs, which is an example of lightweight non-centralised provisioning, using Fabric):

```
<start of code>

from fabric.api import env, run, sudo, get, put

env.user = "<sudo USER>"
env.key_filename = "<KEY PAIR>"
env.warn_only = True
env.skip_bad_hosts = True
env.colorize_errors = True
env.connection_attempts = 3

env.roledefs = {
                "WEB": [
                        "<webserverVM1>",
                        ...,
                        "<webserverVMn>",
                       ],
                "DATABASE": [
                             "<dbserverVM1>",
                             ...,
                             "<dbserverVMm>",
                            ],
                ...,
               }

def provWebServer():
  try:
    put("<webserver provisioning script>", "/tmp")
    sudo("cd /tmp && bash <webserver provisioning script>")
    get("/tmp/<webserver provisioning log>")
  except Exception, e:
    print(" Error: %s" %e)

def provDBServer():
  try:
    put("<dbserver provisioning script>", "/tmp")
    sudo("cd /tmp && bash <dbserver provisioning script>")
    get("/tmp/<dbserver provisioning log>")
  except Exception, e:
    print(" Error: %s" %e)

<end of code>
```

Now just run the command *fab provWebServer -R "WEB"* to provision all your servers that are required to act as Web servers and the command *fab provDBServer -R "DATABASE"* to provision database servers. Keeping all your provisioning logic in separate scripts ensures the provisioning and orchestration logic are separate, so you can change your provisioning needs just by changing the provisioning script. You could also create Fabric jobs that take parameters at runtime by issuing the *fab <JOB>: Param1,Param2,..* command format, by creating Python sub-routines taking various arguments.

The *fab* command is only one way of leveraging Fabric functionalities. The Fabric framework provides its whole API to create standalone remote execution and job orchestration tools. Using the Fabric API is the most appropriate choice in case you need to create some standalone tools, where different stages of the whole workflow are created at runtime using various cloud based APIs. For example, the Fabric API could fill the job orchestration pieces for a tool that creates cloud based resources on-the-fly, with the various steps to create cloud based environments having to be remotely executed out-of-the-box. The following code demonstrates the usage of the Fabric API by putting together a workflow that cleans up the */tmp* folder for leftover *script/zip/log* files, uploads a zip archive and unzips that to */tmp;* kicks off a bash script unzipped with *sudo* and, finally, downloads the log generated by the remote script execution:

```
from fabric.api import env, run, sudo, get, put

env.user = "<sudo USER>"
env.key_filename = "<KEY PAIR>"
env.warn_only = True
env.skip_bad_hosts = True
env.connection_attempts = 5
env.command_timeout = <TIMEOUT for REMOTE SCRIPT>
env.abort_on_prompts = True
...
env.host_string = hostname

oput = sudo('rm -f /tmp/*.zip /tmp/*.log /tmp/*.sh')
if oput.failed:
  <Handle Failue and dump str(oput)>
else:
  <Dump str(oput)>

oput = put(os.path.join(location, zipname), '/tmp')
if oput.failed:
  <Handle Failure and dump str(oput)>
else:
  <Dump str(oput)>

oput = run('unzip /tmp/%s -d /tmp' %zipname)
if oput.failed:
  <Handle Failure and dump str(oput)>
else:
  <Dump str(oput)>

oput = sudo('cd /tmp && bash %s.sh' %script)
if oput.failed:
  <Handle Failure and dump str(oput)>
else:
  <Dump str(oput)>

oput = get('/tmp/%s.log' %logfile, '.')
if oput.failed:
  <Handle Failure and dump str(oput)>
else:
  <Dump str(oput)>
```

The above examples should help you get started fast with Fabric. Fabric provides a lot of functionality, and the good detailed documentation on its site is enough to unlock its rich features.

Job orchestration through Ansible

Ansible is another option for non-centralised job orchestration. In fact, it's also Python based and automatically selects native SSH or the same SSH framework *Paramiko* on top of which Fabric is built. But Ansible is much more powerful and complicated compared to Fabric—it is a complete configuration management and job orchestrator in a single package. It can perform an easy task like running a command on remote hosts in parallel, as well as complicated tasks like bringing up and configuring cloud environments in a single shot, apart from everything and in between. Ansible uses a very simple command language based on YML, which is powerful enough to perform idempotent configuration management as well as remote job orchestration. In its configuration management capabilities, Ansible is similar to other industry standard configuration management software like Puppet and Chef, but more robust since it is non-centralised and easier due to its very simple DSL.

The prerequisites to controlling anything remotely through Ansible are SSH and Python v2.4+, which are part of almost all of the GNU/Linux based distributions. It means that you could just install Ansible on single or multiple control machines and it's all set to go. Ansible requires Python v2.6+, and the command to install it on GNU/Linux or MacOS X (Windows is not currently supported as the control machine) control machines is:

```
sudo pip install ansible
```

Now typing *ansible --help* in the console should dump a help screen mentioning all the options and arguments possible with the Ansible command line.

As I mentioned earlier, Ansible doesn't require anything like *fabfile* as in the case of Fabric, in order to start your exploration with it. But it requires an inventory file with the hostnames or IPs of the remote nodes, on which you want to run commands or scripts. The inventory file for Ansible is just a place to put down the information about the virtual/physical hosts you are planning to manage, and this information could be static or dynamic. The static inventory makes sense in case your hosts are fixed most of the time, but if you

are planning to manage some cloud environment like AWS EC2, then dynamic inventory is the way to go. The Ansible project provides some out-of-the-box scripts to work with dynamic inventories for cloud projects like AWS EC2, Rackspace Cloud, OpenStack, etc. The default inventory file for Ansible is */etc/ansible/hosts*, but you could use any desired location with the option *-i* with the Ansible command line or by tuning the hostfile in *ansible.cfg*. There are a number of ways in which you could put the hosts' information in the inventory file but shown below is the basic format of the inventory, which clubs various hosts into different logical groups based upon their functionalities:

```
[webservers]
<WEBSERVER1>
<WEBSERVER2>
...

[dbservers]
<DBSERVERS1>
<DBSERVERS2>
...
```

The advantage of grouping different hosts is that you could orchestrate anything on all the hosts under a group, just by supplying the group name to Ansible, which defines the functionalities you want to execute on remote hosts as modules. You need to mention the desired module to run Ansible. The following command is the way you could test whether all your Web servers are responding:

```
ansible webservers -m ping -i /usr/local/etc/ansible/hosts
-u <LOGIN> --private-key <PRIVATE KEY>
```

The following Ansible command transfers a shell script to all your servers and executes it on all the hosts in parallel:

```
ansible all -m script -a "<YOUR SHELL SCRIPT>" -i /usr/
local/etc/ansible/hosts -u <LOGIN> --private-key <PRIVATE
KEY>
```

The command to collect the information about your hosts, known as facts, is:

```
ansible all -m setup -i /usr/local/etc/ansible/hosts -u
<LOGIN> --private-key <PRIVATE KEY>
```

You could reboot all your RedHat based *dbserver* hosts using the following command:

```
ansible dbservers -s -m shell -a "/sbin/reboot" -i /usr/
local/etc/ansible/hosts -u <SUDO LOGIN> --private-key
<PRIVATE KEY>
```

The above single-line commands are just an ad hoc way to use Ansible, and it comes prepackaged with a lot of modules covering everything from a simple ping to complicated cloud based provisioning. Ansible takes a description for the workflows containing various activities and orchestration in a YAML file format, and then executes that to bring about the desired result. These YAML descriptions containing instructions for Ansible to execute are known as playbooks, which are a standard way of accomplishing anything through Ansible in a single shot. It's highly recommended that you start any multi-step workflow by creating the playbooks. These are meant to be stored in your version control system to share and reuse, once you verify them for the correct workflows.

Ansible also reads its global settings from a config file named *ansible.cfg* (highly commented example of a config file link is provided in the Reference section at the end of the article) and that could save your typing when using the Ansible and Ansible-playbook commands. For example, if you put the following settings in *ansbile.cfg* then you don't need to mention these again and again while using the Ansible and Ansible-playbook commands:

```
[default]
...
hostfile       = <Host file path>
host_key_checking = False
log_path = <Dir to contain ansible.log>
private_key_file = <Private key file path>
...
```

With the basic settings in the *ansible.cfg*, you could run any Ansible playbook simple by typing (*-vvvv* option allows the verbose output useful for debugging):

```
ansible-playbook <name of the playbook> -vvvv
```

Now let's explore some fundamentals of how to create the Ansible playbooks. A playbook defines various tasks that are to be performed over a group of remote hosts defined in the Ansible inventory. The combination of hosts and related tasks together is known as a 'play' in Ansible terminology. A playbook could contain multiple plays in case you want to perform different tasks on different hosts. By default, the tasks in a play are performed in parallel over the hosts defined by the respective groups in the inventory. But Ansible also provides the constructs through which you could instruct to run the tasks over hosts in serial manner. The following example presents a playbook, which runs operating systems-specific scripts in parallel on different groups of hosts defined in the inventory:

```
<start of code>
```

```
- - -
- hosts: webservers
  gather_facts: yes
  remote_user: <sudo User>
  sudo: yes

  tasks:
    - name: transfer and run Ubuntu specific script
      script: provision_at_ubuntu.sh arg1 arg2
      register: provdeb
      when: ansible_os_family == "Debian"
    - debug: var=provdeb.stdout_lines

    - name: transfer and run RHEL specific script
      script: provision_at_centos.sh arg1 arg2
      register: provrhel
      when: ansible_os_family == "Redhat"
    - debug: var="provrhel.stdout_lines"

    - hosts: dbservers
      gather_facts: no
      remote_user: <sudo User>
      sudo: yes

  tasks:
   - name: transfer and run diagnostic script one host at a
time
     script: dump_diagnostics.sh arg1 arg2
     register: dumpdiag
     serial: 1
   - debug: var=dumpdiag.stdout_lines
```

This playbook demonstrates some useful constructs provided by Ansible. The facts gathered by Ansible are useful to run your task based upon some characteristics of the remote nodes like type of operating system, etc. The *register* keyword assigns the output of a task to some variable, and *debug* is a module that is useful to dump a variable. You can use a serial keyword to define a batch of hosts on which a task is executed in the serial manner. There is a looping functionality provided by the *with_items* keyword that runs a task over a list of items. Ansible also provides a module called *raw*, which enables you to run a command over SSH with the need for Python on remote hosts. This is helpful to bootstrap the necessary components like Python, etc, on 'Pythonless' hosts like routers or other devices, and then use the full module system provided by Ansible.

The modules *shell* and *command* are other ways to run remote commands in the playbooks. The main difference between these modules is that the *shell* module runs the command through */bin/sh* and understands variables like $HOME, "<", ">", "|" , "&", etc. Ansible stops a playbook run when it first encounters any error after running the command, but you can alter this behaviour by using *ignore_errors*, in case you just want to fire off some remote commands, irrespective of their return codes. The following example shows how easily you can trigger a set of commands on remote hosts:

```
- - -
    - hosts: redis
      gather_facts: no
      remote_user: <sudo User>
      sudo: yes

  tasks:
    - name: run commands
      shell: "{{ item }}"
      register: output
      with_items:
        - dmidecode > /tmp/biosdump.log 2>&1
        - cd /tmp && ifconfig > nwiface.log 2>&1
        - sync
        - df -kh

     - dump: var=output.stdout_lines
```

I have only touched on Ansible modules that execute remote commands and some basic features of the playbooks in this article. But Ansible is so feature rich and powerful that it could handle infrastructure comprising thousands of cloud or physical servers, single-handedly. The detailed exploration of Ansible involves a learning curve and could produce material. But you could get started with the knowledge gained in this article, and go through the Ansible documentation to dig deeper into the subject and create complicated workflows with it. END

References

[1] sshpass homepage: *http://sourceforge.net/projects/sshpass/?source=navbar*
[2] Fabric homepage: *http://www.fabfile.org*
[3] Python for Windows: *https://www.python.org/downloads/windows/*
[4] pip for Windows: *https://sites.google.com/site/pydatalog/python/pip-for-windows*
[5] pycrypto for Windows: *http://www.voidspace.org.uk/python/modules.shtml#pycrypto*
[6] Ansible config file: *https://raw.githubusercontent.com/ansible/ansible/devel/examples/ansible.cfg*
[7] Ansible documentation: *http://docs.ansible.com*

By: Ankur Kumar

The author is a systems and infrastructure developer/architect and researcher.