Illinois Data Science Initiative

Introduction to Fabric for Cluster Management

Version: 0.0.1

Author(s): Quinn Jarrell, Professor Brunner, Benjamin Congdon

April 11, 2017

Introduction to Fabric for Cluster Management

Quinn Jarrell[1,2,3], **Professor Brunner**[1,2,3], **and Benjamin Congdon**[1,2,3]

[1] *National Center For Supercomputing Applications (NCSA)*
[2] *Laboratory for Computation, Data, and Machine Learning*
[3] *Univeristy of Illinois*

Compiled April 11, 2017

Fabric provides a lightweight framework for running administrative tasks on cluster nodes that can easily be extended and customized. Fabric allows cluster operators to programatically SSH into nodes and make changes by defining a set of common tasks and providing a command-line interface for executing those tasks.

https://github.com/lcdm-uiuc

INTRODUCTION

Fabric is an incredibly useful and versatile tool for cluster management. Unlike other orchestration providers like Puppet, Fabric doesn't require that it 'own' all the infrastructure it manages. Fabric's approach is the opposite: it simply allows cluster operators to automate aspects of the jobs they already do by constructing scripts from a bottom-up approach. Thus, Fabric can be used to bootstrap existing clusters, as well as aid in setting up and configuring new ones.

Additionally, Fabric has a *"it's just Python"* approach to scripting, allowing virtually limitless options for making advanced orchestration scripts.

This report will outline some useful techniques and applications of Fabric as a cluster orchestration framework.

ASSUMPTIONS

This technical report will make the following assumptions:

- All nodes in the implementer's cluster are accessible by SSH in some fashion, whether it be by simple login or the use of an SSH key.
- Implementer is familiar with the Python language.
- The implementer has Python 2.5-2.7 installed on the host they will be using to run the orchestration commands.

INSTALLING FABRIC

As Fabric heavily leverages SSH, it has some extra dependencies that may not be installed by default. Ensure that the following dependencies are met, and install using the package manager for your OS as needed:

- gcc
- libffi-devel
- python-devel
- openssl-devel

Once these dependencies have been installed, we can install the Fabric package with:

```
$ pip install fabric
```

Then, open a Python interpreter and run `import fabric` to ensure it has installed correctly.

```
$ python
>>> import fabric
>>>
```

FABRIC IDEOLOGY

Fabric is, at its core, an automation layer on top of SSH. Its analogous to a Pythonic, scriptable version of `clusterssh`. When you run the Fabric command, `fab`, you select a task included in your Fabric configuration file (the *"Fabfile"*). Fabric then opens a SSH shell to all the hosts specified and executes the given task on all the hosts. This Fabric task usually consists of several common SSH actions (creating files, running commands, starting scripts, etc.) chained together. The output of these actions is, by default, piped back to the Fabric user so they can observe the execution of the commands.

Thus, Fabric - and Python, for that matter - need only be installed on the nodes that you initiate these orchestration events from. Additionally, the *fabfile* only needs to be present on the orchestration node. The hosts that you run Fabric commands on only 'see' Fabric as an SSH client.

Fabric is compatible with most methods of authentication to hosts (user/password or SSH keys), which allows proper security procedures to be observed while executing tasks on the cluster.

WRITING A 'FABFILE'

Fabric commands are written in a Python module called a *Fabfile*. When the `fab` command is run (to execute a Fabric command), Fabric looks for a `fabfile` module in the current directory. To begin, this can be as simple has a single file, `fabfile.py`. However, as a project scales, this can be expanded into a full Python module, allowing scalable project organization.

An example of a simple `fabfile.py` is as follows:

[language=Python, showstringspaces=false] from fabric import *

def update(): sudo("yum update") sudo("yum upgrade -y")

This command would be executed as

```
$ fab -H localhost update
```

and would update `yum` packages on `localhost`. You can expand the `-H` option to include a comma-separated list of hosts to run the command on.

Operations

Fabric has a set of operations that can be run on remote hosts. The most heavily used ones are:

- `run()` - Runs a command as the logged-in user on the remote host.
- `sudo()` - Runs a command as *sudo* on the remote host (assuming correct permissions).
- `local()` - Runs a command as logged-in user on the local host.
- `open_shell()` - Opens a shell on the remote host, allowing the operator to execute commands.
- `put()` - Pushes a file from the local host to the remote host.
- `put()` - Downloads a file from the remote host to the local host.
- `reboot()` - Restarts the remote host and attempts to reconnect once the host comes back online.

Each of these operations has a return value that gives an indication of the return status of the execution. Operations like `run()` and `sudo()` return the `stdout` of the command, and have a `.stderr` attribute for the standard error stream. This can be used when chaining commands together to infer information about the state of the remote host, and adjust execution as necessary.

Additionally, it's worth noting the *fail-fast* approach that Fabric takes. By default, if any command returns with a non-zero exit code, Fabric halts execution of the task. This can be disabled by configuring `fabric.env.warn_only`, if necessary.

Decorators

Fabric tasks can be defined simply as functions in the *fabfile*. Fabric also includes a few useful decorators to alter the default execution behavior of the task functions:

- `@serial` - Run the task serially across hosts by default. (Finishes the execution on one host before starting on the next host.) This is useful for sensitive or network intensive tasks, like downloading large packages, so at most 1 node is being acted upon at a time.
- `@parallel` - Run the task in parallel across hosts by default. (Starts the execution on many hosts simultaneously, and continue execution in parallel.) This is useful for computationally expensive or long running tasks,like package manager updates, which can safely be run by many nodes simultaneously.
- `@task` - Explicitly declare the function as a task. (Useful in larger *fabfile* modules)

Fabfile Maintainability

Once in production, *fabfile*s should stay mostly static. Changing production *fabfile*s can cause inconsistencies across the cluster. For example, if our *fabfile* created a worker node from a base OS image, changes in this *fabfile* after deployment would lead to different node configuration after Fabric execution.

We found it helpful to version control all of our *fabfile*s in a centralized git repository.

Additionally, proper commenting and documentation style should be used to promote maintainability. Fabric tasks are usually procedural, so readability is generally simple to maintain.

EXECUTING FABRIC TASKS

The simplest way to run a Fabric command, as discussed previously, is:

```
$ fab <Task>
```

However, Fabric execution can be extended and customized to increase its usefulness.

For example, running multiple fabric tasks chains the tasks together, allowing the composition of more advanced workflows:

```
$ fab <Task1> <Task2>
```

The following sections will discuss a variety of techniques useful in the execution of Fabric tasks.

Specifying Hosts

The simplest means for providing Fabric with a list of hosts is by passing them in as a comma-separated list following the `-H` flag. However, Fabric also allows you to progamatically set the list of hosts by altering the list of hosts in `fabric.env.hosts`.

For version-controlled maintainability, we tend to keep out list of cluster nodes in a text file and load in this list at runtime in a manner as follows: [language=Python, showstringspaces=false] from fabric import * env.hosts = open('hosts').readlines()

One can also imagine that it would be useful to programatically pull down a list of hosts from a cluster management tool like Apache Ambari, and set Fabric's host list dynamically:

[language=Python, showstringspaces=false, breaklines=true] import requests from fabric import *

CLUSTER$_NAME = "test_cluster"AMBARI_HOST = "http : //localhost : 8080"USER, PASS = "admin", "admin"api_url = AMBARI_HOST, CLUSTER_NAMEapi_url+ = "/api/v1/clusters/"api_url+ = CLUSTER_NAMEapi_url+ = "/hosts"headers = "X - Requested - By" : "ambari"$

Get spark host list from Ambari r = requests.get(AMBARI$_HOST, headers = headersauth = (USER, PASS))$

Assign host list to Fabric hosts hosts = r.json()['items'] env.hosts = [h['Hosts']['host$_name']]forhinhosts]$

def update(): sudo("yum update")

Fabric also has the concept of host "roles". Roles allow hosts to be treated differently by Fabric based on their type (i.e. a database host may receive a different set of commands than a web server host for a given command, or a task may only be applicable to one set of hosts because of their role). However, for our purposes we assume relative homogeneity within cluster nodes, so this feature has limited use.

Specifying a Fabfile

By default, Fabric looks for either a `fabfile.py` file or a `fabfile` module when it is executed. However, you can create a *fabfile* with an arbitrary name, and specify the path to that file with the `-F` flag. We have found it useful to make the name of the *fabfile* reflect it's utility (i.e. `volume_management.py`).

```
$ fab -F /path/to/my_custom_fabfile.py <Task>
```

Authentication

SSH

An SSH key may be specified with `-i <KEY_FILE>`. Note that you may have to change your hostnames to be in the form `USERNAME@hostname` in order to select the right user to login as.

```
$ fab <Task> -i ~/.ssh/id_rsa -H centos@cluter.local
```

User Password

Passwords may either be specified by `-I` for a password prompt at execution (recommended), or with `--password` to pass the password as part of the command. There are more involved options for specifying per-host passwords that are documented well in the Fabric documentation.

Parallel Execution

By default, Fabric runs tasks in serial across hosts. So, if we run the example `fab update` command, it will start the update the first host, wait until that update has completed, and repeat that process on successive hosts until completion.

For many tasks, it saves a lot of time to run in parallel. This can be accomplished by using the `-P` flag when executing the command, or by adding the `@parallel` decorator, as alluded to previously.

USAGE IDEAS FOR FABRIC

While Fabric can be used to automate many common cluster management tasks, we have found it to be especially useful in automating the following types of tasks:

- Installing and upgrading Python site-packages on remote hosts.
- Transforming a base CentOS image into a worker and adding the host to the cluster.
- Attaching volumes to remote hosts.
- Executing a command on all the hosts in the cluster.

The LCDM Github organization has a repository containing the Fabfiles used to manage our cluster, which you may find useful in constructing your own Fabfiles.

CONCLUSION

Fabric is a powerful infrastructure orchestration tool that allows for work-flows of varying complexities. It has relatively few dependencies or infrastructure constraints and allows for heavily customizable patterns of task execution. Fabric's light footprint and scalable configuration allows it to be a good candidate for initial cluster management, and bootstrapping automation of existing clusters.