Chopsticks Documentation

Release 1.0

Daniel Pope

Jul 17, 2017

Contents

Chopsticks is an orchestration and remote execution library. It lets you run Python code elsewhere: on remote hosts over SSH, in a Docker sandbox, on the local host (optionally with `sudo`) - even all of these in parallel. It currently runs on Linux and Mac machines.

Nothing needs to be installed on remote hosts except Python and an SSH agent.

Chopsticks was built for extensibility. Remote hosts may import Python code from the orchestration host on demand, so remote agents can immediately use new functions you define. In effect, you have access to the same codebase on remote hosts as on the orchestration host.

As a taster, let's just get the unix time on a remote server called `www.chopsticks.io`, then disconnect:

```
import time
from chopsticks.tunnel import SSHTunnel

with SSHTunnel('www.chopsticks.io') as tun:
    print(tun.call(time.time))
```

Contents:

CHAPTER 1

Introduction

With chopsticks you can simply import functions and hand them to the remote host to be executed.

First stand up an SSH Tunnel:

```
from chopsticks.tunnel import Tunnel
tun = Tunnel('troy.example.com')
```

Then you can pass a function, to be called on the remote host:

```
import time
print('Time on %s:' % tun.host, tun.call(time.time))
```

You can use any pure-Python function in the current codebase, meaning you can create your own libraries of orchestration functions to call on remote hosts (as well as functions that call out to remote hosts using Chopsticks). Naturally those functions can import pure-Python libraries and so on. Your entire local codebase should just work remotely.

`Group` allows for executing a callable on a number of hosts in parallel:

```
from chopsticks.group import Group

group = Group([
    'web1.example.com',
    'web2.example.com',
    'web3.example.com',
])
for host, t in group.call(time.time).successful():
    print('Time on %s:' % host, t)
```

You can also run your code within Docker containers:

```
from chopsticks.tunnel import Docker
from chopsticks.facts import python_version

dkr = Docker('py36', image='python:3.6')
print(dkr.call(python_version))
```

Tunnels and Groups connect lazily (or you can connect them proactively by calling `connect()`). They are also usable as context managers:

```
# Explictly connect and disconnect
group.connect()
group.call(time.time)
group.close()

# Reconnect and disconnect as context manager
with group:
    group.call(time.time)

# Implicit reconnect
group.call(time.time)

# Disconnect when destroyed
del group
```

Naturally, any remote state (imports, globals, etc) is lost when the Tunnel/Group is closed.

Python 2/3

Chopsticks supports both Python 2 and Python 3.

Because Chopsticks takes the view that agents run out of the same codebase as the controller, agents will attempt to use a similar Python interpreter to the one for the controller process:

- `/usr/bin/python2` if the controller process is (any) Python 2.
- `/usr/bin/python3` if the controller process is (any) Python 3.

Jupyter Notebooks

For interactive exploration, Chopsticks can also be used within Jupyter Notebooks. Functions defined in Notebook cells are sent over the tunnel as fragments of Python source (rather than imported).

This generally gives good results, but is somewhat more magical than Chopsticks' standard import behaviour. Any odd behaviour should be reported via the issue tracker.

How it works

The SSH tunnel invokes the `python` binary on the remote host, and feeds it a bootstrap script via stdin.

Once bootstrapped, the remote "agent" sets up bi-directional communication over the stdin/stdout of the tunnel. This communication is used (currently) for two purposes:

- An RPC system to invoke arbitrary callables within the remote agent and pass the returned values back to the controller.
- A PEP-302 import hook system, allowing the remote agent to import pure-Python code from the controller (NB. the controller can only serve Python modules that live within the filesystem - import hooks such as zipimport/compressed eggs are not currently supported).

stdin/stdout on the agent are redirected to `/dev/null`, so calling `print()` on the remote machine will not break the tunnel.

`stderr` is echoed to the controlling console, prefixed with a hostname to identify which Tunnel it issued from. This can therefore be used to feed debugging information back to the orchestration host.

Chopsticks vs ...

It's natural to draw comparisons between Chopsticks and various existing tools, but Chopsticks is a library, not an orchestration framework in its own right, and other tools could potentially build on it.

Ansible

Ansible's YAML syntax is a lot more restrictive than Python. It is friendly for simple cases, but becomes increasingly ugly and convoluted as your scripts become more complex. By writing your orchestration scripts in Python you can take advantage of Python's rich ecosystem of syntax and tools for writing clean Python code and documenting it, which apply even for very complicated use cases.

Ansible's remote execution model involves dropping scripts, calling them, and deleting them. In Ansible 2.1, some of Ansible's support code for Python-based Ansible plugins gets shipped over SSH as part of a zipped bundle; but this doesn't extend to your own code extentions. So Chopsticks is more easily and naturally extensible: write your code how you like and let Chopsticks deal with getting it running on the remote machine.

Fabric

The big difference between Fabric and Chopsticks is that Fabric will only execute shell commands on the remote host, not Python callables. Of course you can drop Python scripts and call them, but then you're back in Ansible territory for extensibility, or you have to bootstrap the dependencies needed to execute such scripts manually.

The difference in concept goes deeper: Fabric tries to be "of SSH", exploiting all the cool SSH tunnelling features. Chopsticks doesn't care about SSH specifically; it only cares about Python and pipes. This is what allows it to work identically with Docker or subprocesses as with remote SSH hosts.

CHAPTER 2

Tunnels

Tunnels are the lowest-level API, used for invoking commands on an individual host or container. For a higher-level API that allows invoking commands in parallel across a range of hosts, see *Groups*.

An established tunnel can be used to invoke commands and receive results.

Tunnel reference

All tunnels support the following methods:

class `chopsticks.tunnel.`**`BaseTunnel`**

`call`(*callable*, **args*, ***kwargs*)
Call the given callable on the remote host.

The callable must return a value that can be serialised as JSON, but there is no such restriction on the parameters.

`close`()
Disconnect the tunnel.

Note that this will terminate the remote process and any state will be lost. This does not destroy the Tunnel object, which can be reconnected with `connect()`.

`fetch`(*remote_path*, *local_path=None*)
Fetch one file from the remote host.

If local_path is given, it is the local path to write to. Otherwise, a temporary filename will be used.

This operation supports arbitarily large files (file data is streamed, not buffered in memory).

The return value is a dict containing:

- `local_path` - the local path written to
- `remote_path` - the absolute remote path

- `size` - the number of bytes received
- `sha1sum` - a sha1 checksum of the file data

put (*local_path*, *remote_path=None*, *mode=420*)
Copy a file to the remote host.

If *remote_path* is given, it is the remote path to write to. Otherwise, a temporary filename will be used.

mode gives is the permission bits of the file to create, or 0o644 if unspecified.

This operation supports arbitarily large files (file data is streamed, not buffered in memory).

The return value is a dict containing:

- `remote_path` - the absolute remote path
- `size` - the number of bytes received
- `sha1sum` - a sha1 checksum of the file data

SSH

class `chopsticks.tunnel.`**`SSHTunnel`** (*host*, *user=None*, *sudo=False*)
A tunnel that connects to a remote host over SSH.

Parameters

- **`host`** – The hostname to connect to, as would be specified on an `ssh` command line.
- **`user`** – The username to connect as.
- **`sudo`** – If true, use `sudo` on the remote end in order to run as the `root` user. Use this when you can `sudo` to root but not `ssh` directly as the root user.

`chopsticks.tunnel.`**`Tunnel`**
alias of *SSHTunnel*

Docker

class `chopsticks.tunnel.`**`Docker`** (*name*, *image='python:2.7'*, *rm=True*)
A tunnel connected to a throwaway Docker container.

Parameters

- **`name`** – The name of the Docker instance to create.
- **`image`** – The Docker image to launch. By default, download and run an official Docker Python image corresponding to the running Python version. Official images are curated by Docker.
- **`rm`** – If true, destroy the container when the tunnel is closed.

Subprocess

class `chopsticks.tunnel.`**`Local`** (*name='localhost'*)
A tunnel to a subprocess on the same host.

Sudo

class chopsticks.tunnel.**Sudo**(*user='root'*, *name=None*)
A tunnel to a process on the same host, launched with sudo.

The *Sudo* tunnel does not deal with password dialogues etc. In order for this to work you must configure sudo not to need a password. You can do this with these lines in /etc/sudoers:

```
Cmnd_Alias PYTHON_CMDS = /usr/bin/python, /usr/bin/python2, /usr/bin/python3
%somegroup   ALL=NOPASSWD: PYTHON_CMDS
```

This would allow users in the group somegroup to be able to run the system Python interpreters using sudo, without passwords.

Warning: Naturally, as Chopsticks is a framework for executing arbitrary code, this allows executing arbitrary code as root. Only make this change if you are happy with relaxing security in this way.

Writing new tunnels

It is possible to write a new tunnel driver for any system that allows you to execute a python binary with direct relay of stdin and stdout pipes. To do this, simply subclass chopsticks.group.SubprocessTunnel. Note that all tunnel instances must have a host attibute which is used as the key for the result in the GroupResult dictionary when executing tasks in a Group.

So, strictly, these requirements apply:

- The tunnel setup machinery should not write to stdout - else you will have to identify and consume this output.
- The tunnel setup machinery should not read from stdin - else you will have to feed the required input.
- Both stdin and stdout must be binary-safe pipes.

The tunnel machinery may write to stderr; this output will be presented to the user.

Recursively tunnelling

Chopsticks can be imported and used on the remote side of a tunnel. This situation is called **recursive tunnelling**, and it has its uses. For example:

- You could create an SSHTunnel to a remote host and then Sudo to execute certain actions as root.
- You could maintain a group of SSHTunnels to physical hosts, that each construct a pool of Docker tunnels - for an instant cluster.

Recursion could be dangerous. For example, consider this function:

```
def recursive():
    with Local() as tun:
        tun.call(recursive)
```

This would effectively fork-bomb your host! To avoid this pitfall, Chopsticks has a built-in depth limit of 2. You can override this limit by setting

```
chopsticks.DEPTH_LIMIT = 3
```

Caution: Do not write

```
chopsticks.DEPTH_LIMIT += 1
```

This will undo the limiting!

CHAPTER 3

Groups

Groups can be used to perform a remote operation in parallel across a number of hosts, and collect the results.

Group API

class `chopsticks.group.`**`Group`**(*hosts*)

A group of hosts, for performing operations in parallel.

`__init__`(*hosts*)

Construct a group from a list of tunnels or hosts.

hosts may contain hostnames - in which case the connections will be made via SSH using the default settings. Alternatively, it may contain tunnel instances.

`call`(*callable*, **args*, ***kwargs*)

Call the given callable on all hosts in the group.

The given callable and parameters must be pickleable.

However, the callable's return value has a tighter restriction: it must be serialisable as JSON, in order to ensure the orchestration host cannot be compromised through pickle attacks.

The return value is a *GroupResult*.

`fetch`(*remote_path*, *local_path=None*)

Fetch files from all remote hosts.

If *local_path* is given, it is a local path template, into which the tunnel's `host` name will be substituted using `str.format()`. Hostnames generated in this way must be unique.

For example:

```
group.fetch('/etc/passwd', local_path='passwd-{host}')
```

If *local_path* is not given, a temporary file will be used for each host.

Return a *GroupResult* of dicts, each containing:

- `local_path` - the local path written to
- `remote_path` - the absolute remote path
- `size` - the number of bytes received
- `sha1sum` - a sha1 checksum of the file data

filter(*predicate*, *exclude=False*)
Return a Group of the tunnels for which *predicate* returns True.

predicate must be a no-argument callable that can be pickled.

If *exclude* is True, then return a Group that only contains tunnels for which predicate returns False.

Raise RemoteException if any hosts could not be connected or fail to evaluate the predicate.

put(*local_path*, *remote_path=None*, *mode=420*)
Copy a file to all remote hosts.

If remote_path is given, it is the remote path to write to. Otherwise, a temporary filename will be used (which will be different on each host).

mode gives the permission bits of the files to create, or 0o644 if unspecified.

This operation supports arbitarily large files (file data is streamed, not buffered in memory).

Return a *GroupResult* of dicts, each containing:

- `remote_path` - the absolute remote path
- `size` - the number of bytes received
- `sha1sum` - a sha1 checksum of the file data

Results

class chopsticks.group.**GroupResult**
The results of a *Group.call()* operation.

GroupResult behaves as a dictionary of results, keyed by hostname, although failures from individual hosts are represented as *ErrorResult* objects.

Methods are provided to easily process successes and failures separately.

failures()
Iterate over failed results as (host, err) pairs.

raise_failures()
Raise a RemoteException if there were any failures.

successful()
Iterate over successful results as (host, value) pairs.

class chopsticks.group.**ErrorResult**(*msg*, *tb=None*)
Indicates an error returned by the remote host.

Because tracebacks or error types cannot be represented across hosts this will simply consist of a message.

Error results provide the following attributes:

msg
A human-readable error message.

tb
The traceback from the remote host as a string, or `None` if unavailable.

Set operations

Groups also behave like sets over tunnels. Tunnels are compared by name for this purpose (in general, tunnels need unique names due to the way results are returned from group methods).

For example:

```
webservers = Group(['web1', 'web2'])
celery_workers = Group(['worker1', 'worker2', 'worker3'])

(webservers + celery_workers).call(install_virtualenv)
```

For this purpose, individual tunnels act as a group containing just one tunnel:

```
>>> dck1 = Docker('docker1')
>>> dck2 = Docker('docker2')
>>> dck1 + dck2
Group([Docker('docker1'), Docker('docker2')])
```

Examples

For example, this code:

```
from chopsticks.facts import ip
from chopsticks.group import Group

group = Group([
    'web1.example.com',
    'web2.example.com',
    'web3.example.com',
])
for host, addr in group.call(ip).items():
    print('%s ip:' % host, addr)
```

might output:

```
web1.example.com ip: 196.168.10.5
web3.example.com ip: 196.168.10.7
web2.example.com ip: 196.168.10.6
```

You could also construct a group from existing tunnels - or mix and match:

```
all_hosts = Group([
    'web1.example.com',
    Docker('example'),
    Local('worker')
])
```

CHAPTER 4

Queues

While `Group` lets you run one operation across many hosts, Chopsticks' *`Queue`* class lets you run a number of different operations across many hosts, so that each host is kept as busy as possible.

Conceptually, a `Queue` is actually a separate queue of operations for each host. All hosts start their first operation as soon as *`Queue.run()`* is called.

`Queue` is also Chopsticks' primary *asynchronous* API; callbacks can be registered which are called as soon as a result is available.

Queue API

class `chopsticks.queue.`**`Queue`**
A queue of tasks to be performed.

Queues build on Groups and Tunnels in order to feed tasks as quickly as possible to all connected hosts.

All methods accept a parameter *target*, which specifies which tunnels the operation should be performed with. This can be specified as a `Tunnel` or a `Group`.

Each one returns an *`AsyncResult`* which can be used to receive the result of the operation.

`call`(*target*, **args*, ***kwargs*)
Queue a *`call()`* operation to be run on the target.

`connect`(*target*, **args*, ***kwargs*)
Queue a `connect()` operation to be run on the target.

`fetch`(*target*, *remote_path*, *local_path=None*)
Queue a *`fetch()`* operation to be run on the target.

`put`(*target*, **args*, ***kwargs*)
Queue a *`put()`* operation to be run on the target.

`run`()
Run all items in the queue.

This method does not return until the queue is empty.

class chopsticks.queue.**AsyncResult**
The deferred result of a queued operation.

value
Get the value of the result.

Raise NotCompleted if the task has not yet run.

with_callback(*callback*)
Attach a callback to be called when a value is set.

Example

Let's put three separate files `a.txt`, `b.txt` and `c.txt` onto three hosts:

```
group = Group([
    'host1.example.com',
    'host2.example.com',
    'host3.example.com',
])

queue = Queue()
for f in ['a.txt', 'b.txt', 'c.txt']:
    queue.put(group, f, f)
queue.run()
```

Let's compare this to an approach using the `Group` alone:

```
group = Group([
    'host1.example.com',
    'host2.example.com',
    'host3.example.com',
])

for file in ['a.txt', 'b.txt', 'c.txt']:
    group.put(f, f)
```

The `Queue` approach will typically run faster, because we do not wait for all the tunnels to catch up after every transfer:

With a lengthy list of tasks, and inevitable variability in how long they take, the Queue is likely to finish much sooner.

CHAPTER 5

How to...

How to write a single-file Chopsticks script

Chopsticks will work very well with a neatly organised codebase of management functions, but you can also write a single file script.

Chopsticks has special logic to handle this case, which is different from the standard import machinery.

The cleanest way to write this script would be:

```
from chopsticks import Tunnel


def do_it():
    return 'done'


if __name__ == '__main__':
    with Tunnel('remote') as tun:
        tun.call(do_it)
```

Actually, only the `do_it()` function, and various globals it uses, are sent to the remote host. This code will work just fine:

```
from chopsticks import Tunnel


def do_it():
    return 'done'


tunnel = Tunnel('remote')
tunnel.call(do_it)
```

This also allows Chopsticks to be used from within Jupyter Notebooks.

How to customise interpreter paths

Chopsticks assumes that the interpreter path on a remote host will be `/usr/bin/python2` for Python 2 and `/usr/bin/python3` for Python 3. However, these paths may not always be correct.

To override the path of the interpreter you can simple subclass `Tunnel` (or the tunnel type you wish to use), and modify the `python2` and `python3` class attributes:

```
class MyTunnel(Tunnel):
    python3 = '/usr/local/bin/python3'
```

To do this for all tunnels of the same type, modify the attribute on the type:

```
Tunnel.python3 = '/usr/local/bin/python3'
```

CHAPTER 6

Examples

In this example, we install a configuration file to three servers in parallel and then restart a service:

```
import subprocess
from chopsticks.group import Group

webservers = Group(['www1', 'www2', 'www3'])

webservers.put('uwsgi.ini', '/srv/www/supervisor/uwsgi.ini')
webservers.call(
    subprocess.check_output,
    'supervisord restart uwsgi',
    shell=True
).raise_failures()
webservers.close()
```

CHAPTER 7

Version History

1.0 - 2017-07-06

API Changes

- New *Queue* API for asynchronous operations and scheduling different tasks onto different hosts.
- Chopsticks can be imported and used on remote hosts (see *Recursively tunnelling*).
- Functions defined in `__main__` modules or Jupyter notebooks can now be sent to remote hosts.
- Tunnels and Groups now connect lazily.
- Tunnels and Groups can be used as context managers to ensure they are closed.
- Tunnels and Groups can be reconnected once closed.
- Tunnels and Groups now support *set operations* (union, difference, etc). Tunnels behave as a group of one tunnel.
- New *Group.filter()* method allows filtering hosts by executing a function on each host.
- Added a *Sudo* tunnel, to run as a different user on the local machine.
- Added a `sudo` parameter to *SSHTunnel*, to run as `root` on a remote host.
- New *GroupResult.raise_failures()* allows converting `ErrorResult` to exceptions.

Internal Changes

- Parameters are now sent over the tunnels using a custom binary protocol, rather than JSON. This is more efficient for byte strings, as used in the importer machinery.
- Automatically configure the highest pickle version to use based on what is supported by the host.

0.5 - 2016-08-07

- `Group.put()` and `Group.fetch()` methods allow sending and receiving files from Tunnels in parallel.
- Raise exceptions when Tunnel methods fail.

0.4 - 2016-07-24

- Prefix lines of stderr from tunnels with hostname.
- New `Docker` tunnel, to open a tunnel into a new container.
- Added Sphinx documentation, on readthedocs.org.

0.3 - 2016-07-15

- Added support for Python 3.

0.2 - 2016-07-13

- Add `Group` for running operations on multiple hosts in parallel.

0.1 - 2016-07-12

- Initial public version

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index