CarbonTube Documentation

Release 0.1.1

Gabriel Falcao

Oct 09, 2018

Contents

CHAPTER 1

Table of Contents:

1.1 Introduction

Danger: This project is **ENTIRELY EXPERIMENTAL** at the moment. Use at your own will or if you want to contribute to it

CarbonTube is an easy python DSL for describing phases of execution of a pipeline.

Even more, the phases can be scaled up independently and spread in a network.

1.1.1 This is NOT a DATA pipeline framework

This framework allows you to describe workers using a simple DSL. Each worker produces a given `job_type`.

A pipeline is a sequence of job types that will be coordinate with any idle workers that announce their availability.

This gives you the advantage of scaling your infrastructure horizontally and vertically with very little effort.

1.1.2 Features

- Describe phase workers using python classes and get them running within minutes
- Describe pipelines that can juggle with any available phases
- Easily scale phases individually in a pipeline
- Easily scale pipelines onto clusters
- Empower sys-admins to take quick action and increase the number of phases on demand, be it with new machines, new docker instances or even one-off spawned processes.
- On-demand live web interface with live pipeline cluster information

- Redis queue and metrics persistence

1.2 Basic Usage

1.2.1 Instalation

```
pip install carbontube
```

1.2.2 Defining Phases

```
import os
import uuid
import hashlib

from carbontube import Phase, Pipeline
from carbontube.storage import RedisStorageBackend


class GenerateFile(Phase):
    job_type = 'generate-file'

    def execute(self, instructions):
        size = instructions.get('size')
        if not size:
            return

        path = '/tmp/example-{0}.disposable'.format(uuid.uuid4())
        data = '\n'.join([str(uuid.uuid4()) for _ in range(size)])
        open(path, 'wb').write(data)
        return {'file_path': path}


class HashFile(Phase):
    job_type = 'calculate-hash'

    def execute(self, instructions):
        if 'file_path' not in instructions:
            return

        file_path = instructions['file_path']
        if not os.path.exists(file_path):
            msg = "Failed to hash file {0}: does not exist".format(file_path)
            self.logger.warning(msg)
            raise RuntimeError(msg)

        data = open(file_path, 'rb').read()
        return {'hash': hashlib.sha1(data).hexdigest(), 'file_path': instructions[
↪'file_path']}


class RemoveFile(Phase):
    job_type = 'delete-file'
```

(continues on next page)

(continued from previous page)

```
    def execute(self, instructions):
        path = instructions.get('file_path')
        if path and os.path.exists(path):
            os.unlink(path)
            return {'deleted_path': path}

        raise RuntimeError('file already deleted: {0}'.format(path))


class Example1(Pipeline):
    name = 'example-one'

    phases = [
        GenerateFile,
        RemoveFile
    ]

    def initialize(self):
        self.backend = RedisStorageBackend(self.name, redis_uri='redis://127.0.0.
↪1:6379')
```

1.2.3 Running the servers

```
# run the pipeline
carbontube pipeline examples/simple.py example-one \
    --sub-bind=tcp://127.0.0.1:6000 \
    --job-pull=tcp://127.0.0.1:5050

# then execute the phases separately, they will bind to random
# local tcp ports and announce their address to the pipeline
# subscriber
carbontube phase examples/simple.py generate-file \
    --sub-connect=tcp://127.0.0.1:6000
carbontube phase examples/simple.py calculate-hash \
    --sub-connect=tcp://127.0.0.1:6000
carbontube phase examples/simple.py delete-file \
    --sub-connect=tcp://127.0.0.1:6000
```

1.2.4 Feeding the pipeline with jobs

in the console

```
carbontube enqueue tcp://127.0.0.1:5050 example1 "{\"size\": 10}"
```

in python

```
from carbontube.clients import PipelineClient
client = PipelineClient("tcp://127.0.0.1:5050")
client.connect()
```

(continues on next page)

(continued from previous page)

```
job = {
    'name': 'example1'
    'instructions': {}
}
ok, payload = client.enqueue_job(job)
if ok:
    print "JOB ENQUEUED!"
else:
    print "PIPELINE'S BUFFER IS BUSY, TRY AGAIN LATER"
```

1.3 Internals Reference

1.3.1 Servers

class `carbontube.servers.`**`Pipeline`**(*name*, *concurrency=10*, *backend_class=<class 'carbontube.storage.inmemory.EphemeralStorageBackend'>*)

Pipeline server class

A pipeline must be defined only after you already at least one `Phase`.

`handle_finished_job`(*job*)

called when a job just finished processing.

When overriding this method make sure to call `super()` first

`initialize`()

Initializes the backend.

Subclasses can overload this in order to define their own backends.

`on_finished`(*event*)

called when a job just finished processing. You can override this at will

`on_started`(*event*)

called when a job just started processing.

This method is ok to be overriden by subclasses in order to take action appropriate action.

1.3.2 Clients

class `carbontube.clients.`**`PipelineClient`**(*address*, *hwm=10*)

Pipeline client

Has the ability to push jobs to a pipeline server

`connect`()

connects to the server

`enqueue_job`(*data*)

pushes a job to the pipeline.

Note that the data must be a dictionary with the following keys:

- `name` - the pipeline name
- `instructions` - a dictionary with instructions for the first phase to execute

Parameters `data` – the dictionary with the formatted payload.

Returns the payload sent to the server, which contains the job id

EXAMPLE:

```
>>> from carbontube.clients import PipelineClient

>>> properly_formatted = {
...     "name": "example1",
...     "instructions": {
...         "size": 100",
...     },
... }
>>> client = PipelineClient('tcp://127.0.0.1:5050')
>>> client.connect()
>>> ok, payload_sent = client.enqueue_job(properly_formatted)
```

1.3.3 Storage Backends

class `carbontube.storage.`**`BaseStorageBackend`**(*name*, **args*, ***kw*)

base class for storage backends

`connect`()

this method is called by the pipeline once it started to listen on zmq sockets, so this is also an appropriate time to implement your own connection to a database in a backend subclass pass

`consume_job_of_type`(*job_type*)

dequeues a job for the given type. must return None when no job is ready.

Make sure to requeue this job in case it could not be fed into an immediate worker.

`enqueue_job`(*job*)

adds the job to its appropriate queue name

`get_next_available_worker_for_type`(*job_type*)

randomly picks a workers that is currently available

`initialize`()

backend-specific constructor. This method must be overriden by subclasses in order to setup database connections and such

`register_worker`(*worker*)

register the worker as available. must return a boolean. True if the worker was successfully registered, False otherwise

`unregister_worker`(*worker*)

unregisters the worker completely, removing it from the roster

class `carbontube.storage.`**`EphemeralStorageBackend`**(*name*, **args*, ***kw*)

in-memory storage backend. It dies with the process and has no option for persistence whatsoever. Used only for testing purposes.

`connect`()

this method is called by the pipeline once it started to listen on zmq sockets, so this is also an appropriate time to implement your own connection to a database in a backend subclass pass

`consume_job_of_type`(*job_type*)

dequeues a job for the given type. must return None when no job is ready.

Make sure to requeue this job in case it could not be fed into an immediate worker.

enqueue_job(*job*)
adds the job to its appropriate queue name

get_next_available_worker_for_type(*job_type*)
randomly picks a workers that is currently available

initialize()
backend-specific constructor. This method must be overriden by subclasses in order to setup database connections and such

register_worker(*worker*)
register the worker as available. must return a boolean. True if the worker was successfully registered, False otherwise

unregister_worker(*worker*)
unregisters the worker completely, removing it from the roster

class carbontube.storage.**RedisStorageBackend**(*name*, **args*, ***kw*)
Redis Storage Backend

connect()
this method is called by the pipeline once it started to listen on zmq sockets, so this is also an appropriate time to implement your own connection to a database in a backend subclass pass

enqueue_job(*job*, *state*)
adds the job to its appropriate queue name

get_next_available_worker_for_type(*job_type*)
randomly picks a workers that is currently available

initialize(*redis_uri='redis://'*, *worker_availability_timeout=300*)
backend-specific constructor. This method must be overriden by subclasses in order to setup database connections and such

register_worker(*worker*)
register the worker as available. must return a boolean. True if the worker was successfully registered, False otherwise

unregister_worker(*worker*)
unregisters the worker completely, removing it from the roster

class carbontube.storage.**RedisJobStorage**(*name*, **args*, ***kw*)

connect()
this method is called by the pipeline once it started to listen on zmq sockets, so this is also an appropriate time to implement your own connection to a database in a backend subclass pass

initialize(*redis_uri='redis://'*, *worker_availability_timeout=300*)
backend-specific constructor. This method must be overriden by subclasses in order to setup database connections and such

1.3.4 Utilities

class carbontube.util.**CompressedPickle**(**args*, ***kw*)
Serializes to and from zlib compressed pickle

pack(*item*)
Must receive a python object and return a safe primitive (dict, list, int, string, etc).

`unpack`(*item*)
must receive a *string* and return a python object

`carbontube.util.`**`parse_port`**(*address*)
parses the port from a zmq tcp address

Parameters **`address`** – the string of address

Returns an `int` or `None`

`carbontube.util.`**`read_internal_file`**(*path*)
reads an internal file, mostly used for loading lua scripts

`carbontube.util.`**`sanitize_name`**(*name*)
ensures that a job type or pipeline name are safe for storage and handling.

Parameters **`name`** – the string

Returns a safe string

1.4 The command-line client

Python Module Index

Index