Powerhose Documentation

Release 0.4

Mozilla Foundation

July 24, 2012

CONTENTS

Note: This is still a work in progress - no stable version has been released yet.

Powerhose turns your CPU-bound tasks into I/O-bound tasks so your Python applications are easier to scale.

Powerhose is an implementation of the Request-Reply Broker pattern in ZMQ.

The three main parts are:

1. a client that connects to a broker to send some jobs.
2. a broker that binds a socket to get some job (*“front”*) from clients, and another socket for workers (*“back”*) to connect.
3. workers that connect to the “back” socket of the broker, receive jobs and send back results.

When client sends a job, the broker simply re-routes it to one of its workers, then gets back the result and send it back to the client.

Workers also subscribe to a *“heartbeat”* socket that receives regular heart beats from the broker. When the beat is lost, workers simply die.

Powerhose uses Circus to manage the life of the broker and the workers, and provide high level APIs for your program.

Workers can be written in Python but also in any language that has a ZeroMQ library.

If you have CPU-Bound tasks that could be performed in a specialized C++ program for example, Powerhose is a library you could use to ease your life.

Note: Before using Powerhose, ask yourself if you really need it.

The overhead of the data exchange can be quite important, so unless your job is a CPU-bound task that takes more than 20 ms to perform, there are high chances using Powerhose will not speed it up.

There’s a `bench.py` module in the examples you can change to make some tests with your code. It will compare the speed with and without Powerhose in a multi-threaded environment so you can see if you get any benefit.

If you are curious about why we wrote this library see *Why Powerhose ?*.

CHAPTER

ONE

EXAMPLE

Here's a full example of usage – we want to delegate some work to a specialized worker.

Let's create a function that just echoes back what was sent to it, and save it in an `example` module:

```
def echo(job):
    return job.data
```

This function takes a `Job` instance that contains the data sent by Powerhose. It returns its content immediately.

Let's run our Powerhose cluster with the *powerhose* command, by simply pointing the `echo()` callable:

```
$ powerhose example.echo
[circus] Starting master on pid 51177
[circus] broker started
[circus] workers started
[circus] Arbiter now waiting for commands
```

That's it ! By default one broker and 5 workers are launched, but you can run as many workers as you need, and even add more while the system is running.

Now that the system is up and running, let's try it out in a Python shell:

```
>>> from powerhose.client import Client
>>> client = Client()
>>> client.execute('test')
'test'
```

Congrats ! You have a Powerhose system up and running !

To learn about all existing commands and their options, see *Command-line tools*.

CHAPTER

TWO

RUNNING POWERHOSE WITH CIRCUS

Of course, the goal is to keep the broker and its workers up and running on a system. You can use Daemontools, Supervisord or Circus.

Circus is our preferred system. A Circus config can look like this:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555

[watcher:master]
cmd = powerhose-broker
args = --frontend ipc:///tmp/front --backend ipc:///tmp/backend --heartbeat ipc:///tmp/heartbeat
warmup_delay = 0
numprocesses = 1

[watcher:workers]
cmd = powerhose-worker
args = --backend ipc:///tmp/backend --heartbeat ipc:///tmp/heartbeat echo_worker.echo
warmup_delay = 0
numprocesses = 5
```

This file can then be launched via **circusd**. See the Circus documentation for details on this.

CHAPTER

THREE

USING POWERHOSE PROGRAMMATICALY

The simplest way to run a Powerhose system is to use the command line as previously shown, but you can also do everything programmatically via the `get_cluster()` function.

Here's an example:

```
from powerhose import get_cluster
from powerhose.client import Client


cluster = get_cluster('example.echo', background=True)
cluster.start()

client = Client()

for i in range(10):
    print client.execute(str(i))

cluster.stop()
```

Here, the cluster is launched programmatically in the background and the client uses it as before.

To learn more about Powerhose APIs, see *Library*.

CHAPTER

FOUR

MORE DOCUMENTATION

4.1 Installation

Use pip:

```
$ pip install powerhose
```

Or download the archive on PyPI and install it manually with:

```
$ python setup.py install
```

If you want to try out Powerhose, see the *examples*.

4.2 Command-line tools

Powerhose comes with three commands:

- **powerhose-broker**: runs a broker.
- **powerhose-worker**: runs a *python* worker.
- **powerhose**: runs a broker *and* some workers.

To run a Powerhose cluster, you need to run a broker and some workers. So you would use **powerhose** *or* **powerhose-broker** and some **powerhose-worker**.

4.2.1 powerhose-broker

Runs a Powerhose broker:

```
$ powerhose-broker --help
usage: powerhose-broker [-h] [--frontend FRONTEND] [--backend BACKEND]
                        [--heartbeat HEARTBEAT] [--debug] [--check]
                        [--purge-ghosts] [--logfile LOGFILE]

Powerhose broker.

optional arguments:
-h, --help            show this help message and exit
--frontend FRONTEND   ZMQ socket to receive jobs.
--backend BACKEND     ZMQ socket for workers.
--heartbeat HEARTBEAT
```

```
                        ZMQ socket for the heartbeat.
--debug               Debug mode
--check               Use this option to check if there’s a running broker.
                        Returns the PID if a broker is up.
--purge-ghosts        Use this option to purge ghost brokers.
--logfile LOGFILE     File to log in to .
```

–check and **–purge-ghosts** are maintenance option that are not running a broker but just checking an existing broker:

```
$ powerhose-broker --check
[2012-05-25 11:11:28,282][powerhose] A broker is running. PID: 11668

$ bin/powerhose-broker --purge-ghosts
[2012-05-25 11:12:09,744][powerhose] The active broker runs at PID: 11668
[2012-05-25 11:12:09,744][powerhose] No ghosts where killed.
```

Those options can be used to health check and monitor the broker, which is the stable node of the Powerhose architecture.

4.2.2 powerhose-worker

Runs one worker.

```
$ powerhose-worker --help
usage: powerhose-worker [-h] [--backend BACKEND] [--debug] [--logfile LOGFILE]
                        [--heartbeat HEARTBEAT] [--params PARAMS]
                        [--timeout TIMEOUT] [--max-age MAX_AGE]
                        [--max-age-delta MAX_AGE_DELTA]
                        target

Runs a worker.

positional arguments:
target                Fully qualified name of the callable.

optional arguments:
-h, --help            show this help message and exit
--backend BACKEND     ZMQ socket to the broker.
--debug               Debug mode
--logfile LOGFILE     File to log in to.
--heartbeat HEARTBEAT
                        ZMQ socket for the heartbeat.
--params PARAMS       The parameters to be used in the worker.
--timeout TIMEOUT     The maximum time allowed before the thread stacks is
                        dump and the job result not sent back.
--max-age MAX_AGE     The maximum age for a worker in seconds. After that
                        delay, the worker will simply quit. When set to -1,
                        never quits.
--max-age-delta MAX_AGE_DELTA
                        The maximum value in seconds added to max_age
```

The **–max-age** option is useful when you want your worker to exit after some time. The typical use case is when you have a program that keeps some connectors open on some external ressources, and those ressources change over time.

4.2.3 powerhose

Runs one broker and several workers by using the two previous commands.

```
$ powerhose --help
usage: powerhose [-h] [--frontend FRONTEND] [--backend BACKEND]
                [--heartbeat HEARTBEAT] [--debug]
                [--numprocesses NUMPROCESSES] [--logfile LOGFILE]
                target

Run a Powerhose cluster.

positional arguments:
target                Fully qualified name of the callable.

optional arguments:
-h, --help            show this help message and exit
--frontend FRONTEND   ZMQ socket to receive jobs.
--backend BACKEND     ZMQ socket for workers.
--heartbeat HEARTBEAT
                        ZMQ socket for the heartbeat.
--debug               Debug mode
--numprocesses NUMPROCESSES
                        Number of processes to run.
--logfile LOGFILE     File to log in to .
```

4.3 Library

Powerhose is organized into a simple hierarchy of classes and a few functions:

- `get_cluster()` – a function that create a whole cluster.
- `Job` – A class that holds a job to be performed.
- `Broker` – A class that pass the jobs it receives to workers.
- `Worker` – A class that connects to a broker and pass jobs it receives to a Python callable.
- `Heartbeat` and `Stethoscope` – implements a heartbeat.
- `Client` – A class that connects to a broker and let you run jobs against it.

4.3.1 get_cluster

The `get_cluster()` function creates a `Broker` and several `Worker` instance. It can be run in the background for conveniency.

`powerhose.`**`get_cluster`**(*target, numprocesses=5, frontend='ipc:///tmp/powerhose-front.ipc', backend='ipc:///tmp/powerhose-back.ipc', heartbeat='ipc:///tmp/powerhose-beat.ipc', register='ipc:///tmp/powerhose-reg.ipc', working_dir='.', logfile='stdout', debug=False, background=False, worker_params=None, timeout=7.5, max_age=-1, max_age_delta=0*)

Runs a Powerhose cluster.

Options:

- **callable**: The Python callable that will be called when the broker receive a job.

- **numprocesses**: The number of workers. Defaults to 5.
- **frontend**: the ZMQ socket to receive jobs.
- **backend**: the ZMQ socket to communicate with workers.
- **register** : the ZMQ socket to register workers
- **heartbeat**: the ZMQ socket to receive heartbeat requests
- **working_dir**: The working directory. Defaults to ”.”
- **logfile**: The file to log into. Defaults to stdout.
- **debug**: If True, the logs are at the DEBUG level. Defaults to False
- **background**: If True, the cluster is run in the background. Defaults to False.
- **worker_params**: a dict of params to pass to the worker. Default is None
- **timeout** the maximum time allowed before the thread stacks is dumped and the job result not sent back.
- **max_age**: maximum age for a worker in seconds. After that delay, the worker will simply quit. When set to -1, never quits. Defaults to -1.
- **max_age_delta**: maximum value in seconds added to max age. The Worker will quit after *max_age + random(0, max_age_delta)* This is done to avoid having all workers quit at the same instant.

Example:

```
from powerhose import get_cluster
from powerhose.client import Client


cluster = get_cluster('example.echo', background=True)
cluster.start()

client = Client()

for i in range(10):
    print client.execute(str(i))

cluster.stop()
```

4.3.2 Job

class `powerhose.job.`**`Job`**(*data=’‘*, *headers=None*)

A Job is just a container that’s passed into the wire.

A job is composed of headers and raw data, and offers serialization.

Options:

- **data**: the raw string data (default: ‘’)
- **headers**: a mapping of headers (default: None)

`add_header`(*name*, *value*)

Adds a header.

Options:

- **name**: header name
- **value**: value

Both values should be strings. If the header already exists it's overwritten.

serialize()

Serializes the job.

The output can be sent over a wire. A serialized job can be read with a cursor with no specific preprocessing.

classmethod **load_from_string**(*data*)

Loads a job from a serialized string and return a Job instance.

Options:

- **data** : serialized string.

Example:

```
>>> from powerhose.job import Job
>>> job = Job('4*2')
>>> job.serialize()
'NONE:::4*2'
>>> Job.load_from_string('NONE:::4*2')
<powerhose.job.Job object at 0x107b78c50>
>>> Job.load_from_string('NONE:::4*2').data
'4*2'
```

4.3.3 Broker

class powerhose.broker.**Broker**(*frontend='ipc:///tmp/powerhose-front.ipc'*, *backend='ipc:///tmp/powerhose-back.ipc'*, *heartbeat='ipc:///tmp/powerhose-beat.ipc'*, *register='ipc:///tmp/powerhose-reg.ipc'*, *io_threads=1*)

Class that route jobs to workers.

Options:

- **frontend**: the ZMQ socket to receive jobs.
- **backend**: the ZMQ socket to communicate with workers.
- **heartbeat**: the ZMQ socket to receive heartbeat requests.
- **register** : the ZMQ socket to register workers

start()

Starts the broker.

stop()

Stops the broker.

4.3.4 Worker

class powerhose.worker.**Worker**(*target*, *backend='ipc:///tmp/powerhose-back.ipc'*, *heartbeat='ipc:///tmp/powerhose-beat.ipc'*, *register='ipc:///tmp/powerhose-reg.ipc'*, *ping_delay=1.0*, *ping_retries=3*, *params=None*, *timeout=7.5*, *max_age=-1*, *max_age_delta=0*)

Class that links a callable to a broker.

Options:

- **target**: The Python callable that will be called when the broker send a job.
- **backend**: The ZMQ socket to connect to the broker.
- **heartbeat**: The ZMQ socket to perform PINGs on the broker to make sure it's still alive.
- **register** : the ZMQ socket to register workers
- **ping_delay**: the delay in seconds betweem two pings.
- **ping_retries**: the number of attempts to ping the broker before quitting.
- **params** a dict containing the params to set for this worker.
- **timeout** the maximum time allowed before the thread stacks is dump and the job result not sent back.
- **max_age**: maximum age for a worker in seconds. After that delay, the worker will simply quit. When set to -1, never quits. Defaults to -1.
- **max_age_delta**: maximum value in seconds added to max age. The Worker will quit after *max_age + random(0, max_age_delta)* This is done to avoid having all workers quit at the same instant. Defaults to 0. The value must be an integer.

start()
Starts the worker

stop()
Stops the worker.

4.3.5 Heartbeat

The `Broker` class runs a `Heartbeat` instance that regularly sends a *BEAT* message on a PUB channel. Each worker has a `Stethoscope` instance that subscribes to the channel, to check if the `Broker` is still around.

class `powerhose.heartbeat.`**Heartbeat**(*endpoint='ipc:///tmp/powerhose-beat.ipc'*, *interval=2.0*, *io_loop=None*, *ctx=None*)
Class that implements a ZMQ heartbeat server.

This class sends in a ZMQ socket regular beats.

Options:

- **endpoint** : The ZMQ socket to call.
- **interval** : Interval between two beat.

stop()
Stops the Pong service

start()
Starts the Pong service

class `powerhose.heartbeat.`**Stethoscope**(*endpoint='ipc:///tmp/powerhose-beat.ipc'*, *warmup_delay=0.5*, *delay=3.0*, *retries=3*, *onbeat-lost=None*, *onbeat=None*, *io_loop=None*, *ctx=None*)
Class that implements a ZMQ heartbeat client.

Options:

- **endpoint** : The ZMQ socket to call.
- **warmup_delay** : The delay before starting to Ping. Defaults to 5s.
- **delay**: The delay between two pings. Defaults to 3s.

- **retries**: The number of attempts to ping. Defaults to 3.
- **onbeatlost**: a callable that will be called when a ping failed. If the callable returns **True**, the ping quits. Defaults to None.
- **onbeat**: a callable that will be called when a ping succeeds. Defaults to None.

stop()
Stops the Pinger

4.3.6 Client

class powerhose.client.**Client**(*frontend='ipc:///tmp/powerhose-front.ipc'*, *timeout=5.0*, *timeout_max_overflow=7.5*, *timeout_overflows=1*, *debug=False*, *ctx=None*)
Class to call a Powerhose cluster.

Options:

- **frontend**: ZMQ socket to call.
- **timeout**: maximum allowed time for a job to run. Defaults to 1s.
- **timeout_max_overflow**: maximum timeout overflow allowed. Defaults to 1.5s
- **timeout_overflows**: number of times in a row the timeout value can be overflowed per worker. The client keeps a counter of executions that were longer than the regular timeout but shorter than **timeout_max_overflow**. When the number goes over **timeout_overflows**, the usual TimeoutError is raised. When a worker returns on time, the counter is reset.

execute(*job*, *timeout=None*)
Runs the job

Options:

- **job**: Job to be performed. Can be a `Job` instance or a string. If it's a string a `Job` instance will be automatically created out of it.
- **timeout**: maximum allowed time for a job to run. If not provided, uses the one defined in the constructor.

If the job fails after the timeout, raises a `TimeoutError`.

This method is thread-safe and uses a lock. If you need to execute a lot of jobs simultaneously on a broker, use the `Pool` class.

ping(*timeout=1.0*)
Can be used to simply ping the broker to make sure it's responsive.

Returns the broker PID

4.3.7 Client Pool

class powerhose.client.**Pool**(*size=10*, *frontend='ipc:///tmp/powerhose-front.ipc'*, *timeout=5.0*, *timeout_max_overflow=7.5*, *timeout_overflows=1*, *debug=False*, *ctx=None*)
The pool class manage several `CLient` instances and publish the same interface,

Options:

- **size**: size of the pool. Defaults to 10.
- **frontend**: ZMQ socket to call.

- **timeout**: maximum allowed time for a job to run. Defaults to 5s.
- **timeout_max_overflow**: maximum timeout overflow allowed
- **timeout_overflows**: number of times in a row the timeout value can be overflowed per worker. The client keeps a counter of executions that were longer than the regular timeout but shorter than **timeout_max_overflow**. When the number goes over **timeout_overflows**, the usual TimeoutError is raised. When a worker returns on time, the counter is reset.

4.4 Examples

This section will be added later

4.5 Why Powerhose ?

Python is a great language but as soon as you are doing CPU-bound tasks, you might bump into the GIL issue if you try to run in parallel multiple threads.

The GIL is the *Global Interpreter Lock* used by the CPython & the PyPy implementations to protect some parts of the language implementation itself.

The effect on CPU-bound tasks that are performed by several threads is that you won't be able to use all your machine CPU cores in parallel like in other languages.

To solve this issue, the simplest thing to do is to use **multiprocessing**, a module that comes with the standard library and will let you spawn processes and interact with them using pickles.

But that limits you to using the Python language on both sides.

You could also use a C++ library binded into Python, but it turns out you're still locking the GIL here and there when you use a C++ bind through CPython. Unless you delegate *everything* to the C++ side, the contention can be smaller but is still there.

So ideally, we'd want a library where you can delegate some specific tasks to specialized workers, whatever language they are written into.

Of course this is feasible with the standard library, but requires extra work to set up a protocol between the master and the workers, and decide how to transport the data.

There's also tools like RabbitMQ that can let you set up a queue where the master put some job to be performed, workers can pick up.

But here, we're talking about running specific CPU-Bound jobs as fast as possible, synchronously, with no persistency at all.

Our driving use case is Mozilla's Token Server - https://wiki.mozilla.org/Services/Sagrada/TokenServer

On this server, we have an API you can call to trade a BrowserID assertion for a token you can use to authenticate to some of our services.

Powerhose is our attempt to solve this issue, and is based on ZeroMQ.

We chose ZeroMQ because:

- it's insanely fast.
- it greatly simplifies our code.
- it can work over TCP, IPC (Inter Process Communication) or even in the same process.

Powerhose allows us to:

- deploy dynamically as many workers as we want, even on other servers when it makes sense.
- write workers in C++
- greatly simplify the usage for our Python apps, since all it takes is a single `execute()` call to get the job done.

Read more about this here: http://ziade.org/2012/02/06/scaling-crypto-work-in-python/

CHAPTER

FIVE

CONTRIBUTIONS AND FEEDBACK

You can reach us for any feedback, bug report, or to contribute, at https://github.com/mozilla-services/powerhose

We can also be found in the **#services-dev** channel on irc.mozilla.org.