Circus Documentation

Release 0.6.0

Mozilla Foundation

December 18, 2012

CONTENTS

Circus is a process & socket manager. It can be used to monitor and control processes and sockets.

Circus can be driven via a command-line interface or programmatically through its python API.

It shares some of the goals of Supervisord, BluePill and Daemontools. If you are curious about what Circus brings compared to other projects, read *Why should I use Circus instead of X ?*.

Circus is designed using ZeroMQ. See *Design* for more details.

Note: By default Circus doesn't secure its messages when sending information through ZeroMQ. Before running Circus, make sure you read the *Security* page.

To install it, check out *Installing Circus*

CHAPTER

ONE

RUNNING CIRCUS

Circus provides a command-line script call **circusd** that can be used to manage one or more *watchers*. Each watcher can have one or more running *processes*.

Circus' command-line tool is configurable using an ini-style configuration file. Here is a minimal example:

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

[watcher:myprogram]
cmd = python
args = -u myprogram.py $WID
warmup_delay = 0
numprocesses = 5

[watcher:anotherprogram]
cmd = another_program
numprocesses = 2
```

The file is then run using *circusd*:

```
$ circusd example.ini
```

Note: If you use Circus with zc.buildout, and some of your commands are scripts generated by zc.buildout, you will need to use the **copy_env** and **copy_path** options in your watchers.

Besides processes, Circus can also bind sockets. Since every process managed by Circus is a child of the main Circus daemon, that means any program that's controlled by Circus can use those sockets.

Running a socket is as simple as adding a *socket* section in the config file:

```
[socket:mysocket]
host = localhost
port = 8080
```

To learn more about sockets, see *Circus Sockets*.

To understand why it's a killer feature, read *Circus stack v.s. Classical stack*.

CHAPTER

TWO

CONTROLLING CIRCUS

Circus provides two command-line tools to manage your running daemon:

- *circusctl*, a management console you can use it to perform actions such as adding or removing *workers*
- *circus-top*, a top-like console you can use to display the memory and cpu usage of your running Circus.

To learn more about these, see *Command-line tools*

Circus also offers a small web application that can connect to a running Circus daemon and let you monitor and interact with it.

Running the web application is as simple as adding an **httpd** option in the ini file:

```
[circus]
httpd = True
```

Or if you want, you can run it as a standalone process with:

```
$ circushttpd
```

By default, **circushttpd** runs on the *8080* port.

To learn more about this feature, see *The Web Console*

CHAPTER

THREE

DEVELOPING WITH CIRCUS

Circus provides high-level classes and functions that will let you manage processes in your own applications.

For example, if you want to run four processes forever, you could write:

```
from circus import get_arbiter

arbiter = get_arbiter("myprogram", 4)
try:
    arbiter.start()
finally:
    arbiter.stop()
```

This snippet will run four instances of *myprogram* and watch them for you, restarting them if they die unexpectedly.

To learn more about this, see *Circus Library*

CHAPTER

FOUR

EXTENDING CIRCUS

It's easy to extend Circus to create a more complex system, by listening to all the **circusd** events via its pub/sub channel, and driving it via commands.

That's how the flapping feature works for instance: it listens to all the processes dying, measures how often it happens, and stops the incriminated watchers after too many restarts attempts.

Circus comes with a plugin system to help you write such extensions, and a few built-in plugins you can reuse. See *Plugins*.

You can also have a more subtile startup and shutdown behavior by using the **hooks** system that will let you run arbitrary code before and after some processes are started or stopped. See *Hooks*.

CHAPTER

FIVE

MORE DOCUMENTATION

5.1 Installing Circus

Use pip:

```
$ pip install circus
```

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

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

If you want to try out Circus, see the *Examples*.

If you are using debian or any debian based distributoin, you also can use the ppa to install circus, it's at https://launchpad.net/~roman-imankulov/+archive/circus

5.1.1 More on Requirements

Circus uses:

- Python 2.6, 2.7 (3.x needs to be tested)
- zeromq >= 2.10

And on Python side:

- pyzmq 2.2.0
- iowait 0.1
- psutil 0.4.1

You can install all the py dependencies with the pip-requirements.txt file we provide manually, or just install Circus and have the latest versions of those libraries pulled for you:

```
$ pip install -r pip-requirements.txt
```

If you want to run the Web console you will need more things:

- Mako 0.7.0
- MarkupSafe 0.15
- bottle 0.10.9
- anyjson 0.3.1

- gevent 0.13.7
- gevent-socketio 0.3.5-beta
- gevent-websocket 0.3.6
- greenlet 0.3.4
- beaker 1.6.3
- http://github.com/tarekziade/gevent-zeromq

Those can be installed with:

```
$ pip install -r web-requirements.txt
```

5.2 Configuration

Circus can be configured using an ini-style configuration file.

Example:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
include = /path/to/configs/*.enabled.ini

[watcher:myprogram]
cmd = python
args = -u myprogram.py $(WID) $(ENV.VAR)
warmup_delay = 0
numprocesses = 5

    [env:myprogram]
    PATH = $PATH:/bin
    CAKE = lie

# hook
hooks.before_start = my.hooks.control_redis

# will push in test.log the stream every 300 ms
stdout_stream.class = FileStream
stdout_stream.filename = test.log
stdout_stream.refresh_time = 0.3

# optionally rotate the log file when it reaches 1 gb
# and save 5 copied of rotated files
stdout_stream.max_bytes = 1073741824
stdout_stream.backup_count = 5

[plugin:statsd]
use = circus.plugins.statsd.StatsdEmitter
host = localhost
port = 8125
sample_rate = 1.0
application_name = example

[socket:web]
```

```
host = localhost
port = 8080
```

5.2.1 circus - single section

endpoint The ZMQ socket used to manage Circus via **circusctl**. (default: *tcp://127.0.0.1:5555*)

pubsub_endpoint The ZMQ PUB/SUB socket receiving publications of events. (default: *tcp://127.0.0.1:5556*)

stats_endpoint The ZMQ PUB/SUB socket receiving publications of stats. If not configured, this feature is deactivated. (default: *tcp://127.0.0.1:5557*)

check_delay The polling interval in seconds for the ZMQ socket. (default: 5)

include List of config files to include. (default: None). You can use wildcards (*) to include particular schemes for your files.

include_dir List of config directories. All files matching **.ini* under each directory will be included. (default: None)

stream_backend Defines the type of backend to use for the streaming. Possible values are **thread** or **gevent**. (default: thread)

warmup_delay The interval in seconds between two watchers start. Must be an int. (default: 0)

httpd If set to True, Circus runs the circushttpd daemon. (default: False)

httpd_host The host ran by the circushttpd daemon. (default: localhost)

httpd_port The port ran by the circushttpd daemon. (default: 8080)

debug If set to True, all Circus stout/stderr daemons are redirected to circusd stdout/stderr (default: False)

respawn If set to False, the processes handled by a watcher will not be respawned automatically. (default: True)

Note: If you use the gevent backend for **stream_backend**, you need to install the forked version of gevent_zmq that's located at https://github.com/tarekziade/gevent-zeromq because it contains a fix that has not made it upstream yet.

5.2.2 watcher:NAME - as many sections as you want

NAME The name of the watcher. This name is used in **circusctl**

cmd The executable program to run.

args Command-line arguments to pass to the program. You can use the python format syntax here to build the parameters. Environment variables are available, as well as the worker id and the environment variables that you passed, if any, with the "env" parameter. See *Formating the commands and arguments with dynamic variables* for more information on this.

shell If True, the processes are run in the shell (default: False)

working_dir The working dir for the processes (default: None)

uid The user id or name the command should run with. (The current uid is the default).

gid The group id or name the command should run with. (The current gid is the default).

copy_env If set to true, the local environment variables will be copied and passed to the workers when spawning them. (Default: False)

copy_path If set to true, **sys.path** is passed in the subprocess environ using *PYTHONPATH*. **copy_env** has to be true. (Default: False)

warmup_delay The delay (in seconds) between running processes.

numprocesses The number of processes to run for this watcher.

rlimit_LIMIT Set resource limit LIMIT for the watched processes. The config name should match the RLIMIT_* constants (not case sensitive) listed in the Python resource module reference. For example, the config line 'rlimit_nofile = 500' sets the maximum number of open files to 500.

stderr_stream.class A fully qualified Python class name that will be instanciated, and will receive the **stderr** stream of all processes in its `__call__()` method.

Circus provides three classes you can use without prefix:

- `FileStream`: writes in a file
- `QueueStream`: write in a memory Queue
- `StdoutStream`: writes in the stdout

stderr_stream.* All options starting with *stderr_stream.* other than *class* will be passed the constructor when creating an instance of the class defined in **stderr_stream.class**.

stdout_stream.class A fully qualified Python class name that will be instanciated, and will receive the **stdout** stream of all processes in its `__call__()` method. Circus provides three classes you can use without prefix:

- `FileStream`: writes in a file
- `QueueStream`: write in a memory Queue
- `StdoutStream`: writes in the stdout

stdout_stream.* All options starting with *stdout_stream.* other than *class* will be passed the constructor when creating an instance of the class defined in **stdout_stream.class**.

send_hup if True, a process reload will be done by sending the SIGHUP signal. Defaults to False.

max_retry The number of times we attempt to start a process, before we abandon and stop the whole watcher. Defaults to 5.

priority Integer that defines a priority for the watcher. When the Arbiter do some operations on all watchers, it will sort them with this field, from the bigger number to the smallest. Defaults to 0.

singleton If set to True, this watcher will have at the most one process. Defaults to False.

use_sockets If set to True, this watcher will be able to access defined sockets via their file descriptors. If False, all parent fds are closed when the child process is forked. Defaults to False.

max_age If set then the process will be restarted sometime after max_age seconds. This is useful when processes deal with pool of connectors: restarting processes improves the load balancing. Defaults to being disabled.

max_age_variance If max_age is set then the process will live between max_age and max_age + random(0, max_age_variance) seconds. This avoids restarting all processes for a watcher at once. Defaults to 30 seconds.

hooks.* Available hooks: **before_start**, **after_start**, **before_stop**, **after_stop**

Define callback functions that hook into the watcher startup/shutdown process.

If the hook returns **False** and if the hook is one of **before_start** or **after_start**, the startup will be aborted.

Notice that a hook that fails during the stopping process will not abort it.

The callback definition can be followed by a boolean flag separated by a comma. When the flag is set to **true**, any error occuring in the hook will be ignored. If set to **false** (the default), the hook will return **False**.

More on *Hooks*.

5.2.3 socket:NAME - as many sections as you want

host The host of the socket. Defaults to 'localhost'

port The port. Defaults to 8080.

family The socket family. Can be 'AF_UNIX', 'AF_INET' or 'AF_INET6'. Defaults to 'AF_INET'.

type The socket type. Can be 'SOCK_STREAM', 'SOCK_DGRAM', 'SOCK_RAW', 'SOCK_RDM' or 'SOCK_SEQPACKET'. Defaults to 'SOCK_STREAM'.

path When provided a path to a file that will be used as a unix socket file. If a path is provided, **family** is forced to AF_UNIX and **host** and **port** are ignored.

Once a socket is created, the *${circus.sockets.NAME}* string can be used in the command (*cmd* or *args*) of a watcher. Circus will replace it by the FD value. The watcher must also have *use_sockets* set to *True* otherwise the socket will have been closed and you will get errors when the watcher tries to use it.

Example:

```
[watcher:webworker]
cmd = chaussette --fd $(circus.sockets.webapp) chaussette.util.bench_app
use_sockets = True

[socket:webapp]
host = 127.0.0.1
port = 8888
```

5.2.4 plugin:NAME - as many sections as you want

use The fully qualified name that points to the plugin class.

anything else Every other key found in the section is passed to the plugin constructor in the **config** mapping.

Circus comes with a few pre-shipped *plugins* but you can also extend them easily by *developing your own*.

5.2.5 env:WATCHERS - as many sections as you want

anything The name of an environment variable to assign value to. bash style environment substitutions are supported. for example, append /bin to *PATH* 'PATH = $PATH:/bin'

WATCHERS can be a comma separated list of watcher sections to apply this environment to. if multiple env sections match a watcher, they will be combine in the order they appear in the configuration file. later entries will take precedence.

Example:

```
[watcher:worker1]
cmd = ping 127.0.0.1

[watcher:worker2]
cmd = ping 127.0.0.1

[env:worker1,worker2]
PATH = /bin

[env:worker1]
PATH = $PATH

[env:worker2]
CAKE = lie
```

worker1 will be run with PATH = $PATH (expanded from the environment circusd was run in) *worker2* will be run with PATH = /bin and CAKE = lie

5.2.6 Formating the commands and arguments with dynamic variables

As you may have seen, it is possible to pass some information that are computed dynamically when running the processes. Among other things, you can get the worker id (WID) and all the options that are passed to the `Process`. Additionally, it is possible to access the options passed to the `Watcher` which instanciated the process.

Note: The worker id is different from the process id. It's a unique value, starting at 1, which is only unique for the watcher.

For instance, if you want to access some variables that are contained in the environment, you would need to do it with a setting like this:

```
cmd = "make-me-a-coffee --sugar $(CIRCUS.ENV.SUGAR_AMOUNT)"
```

This works with both *cmd* and *args*.

Important:

- All variables are prefixed with *circus.*
- The replacement is case insensitive.

5.3 Plugins

Circus comes with a few pre-shipped plugins you can use easily. The configuration of them is as follows:

5.3.1 Statsd

use set to 'circus.plugins.statsd.StatsdEmitter'

application_name the name used to identify the bucket prefix to emit the stats to (it will be prefixed with "circus." and suffixed with ".watcher")

host the host to post the statds data to

port the port the statsd daemon listens on

sample_rate if you prefer a different sample rate than 1, you can set it here

5.3.2 FullStats

An extension on the Statsd plugin that is also publishing the process stats. As such it has the same configuration options as Statsd and the following.

use set to 'circus.plugins.statsd.FullStats'

loop_rate the frequency the plugin should ask for the stats in seconds. Default: 60.

5.3.3 RedisObserver

This services observers a redis process for you, publishes the information to statsd and offers to restart the service when it doesn't react in a given timeout. This plugin requires redis-py to run.

It has the same configuration as statsd and adds the following:

use set to 'circus.plugins.redis_observer.RedisObserver'

loop_rate the frequency the plugin should ask for the stats in seconds. Default: 60.

redis_url the database to check for as a redis url. Default: "redis://localhost:6379/0"

timeout the timeout in seconds the request can take before it is considered down. Defaults to 5.

restart_on_timeout the name of the process to restart when the request timed out. No restart triggered when not given. Default: None.

5.3.4 HttpObserver

This services observers a http process for you by pinging a certain website regularly. Similar to the redis observer it offers to restart the service on an error. It requires tornado to run.

It has the same configuration as statsd and adds the following:

use set to 'circus.plugins.http_observer.HttpObserver'

loop_rate the frequency the plugin should ask for the stats in seconds. Default: 60.

check_url the url to check for. Default: "http://localhost/"

timeout the timeout in seconds the request can take before it is considered down. Defaults to 10.

restart_on_error the name of the process to restart when the request timed out or returned any other kind of error. No restart triggered when not given. Default: None.

5.3.5 ResourceWatcher

This services watches the resources of the given process and triggers a restart when they exceed certain limitations too often in a row.

It has the same configuration as statsd and adds the following:

use set to 'circus.plugins.resource_watcher.ResourceWatcher'

loop_rate the frequency the plugin should ask for the stats in seconds. Default: 60.

service the service (read: watcher) this resource watcher should be looking after

max_cpu The maximum cpu one process is allowed to consume (in %). Default: 90

max_mem The amount of memory one process of this watcher is allowed to consume (in %). Default: 90

health_threshold The health is the average of cpu and memory (in %) the watchers processes are allowed to consume (in %). Default: 75

max_count How often these limits (each one is counted separately) are allowed to be exceeded before a restart will be triggered. Default: 3

5.4 Hooks

Circus provides four hooks that can be used to trigger actions when a watcher is starting or stopping.

A typical use case is to control that all the conditions are met for a process to start.

Let's say you have a watcher that runs *Redis* and a watcher that runs a Python script that works with *Redis*.

With Circus you can order the startup by using the **priority** option:

```
[watcher:queue-worker]
cmd = python -u worker.py
priority = 2

[watcher:redis]
cmd = redis-server
priority = 1
```

With this setup, Circus will start **Redis** then the queue worker.

But Circus does not really control that *Redis* is up and running. It just starts the process it was asked to start.

What we miss here is a way to control that *Redis* is started, and fully functional. A function that controls this could be:

```
import redis
import time

def check_redis(*args, **kw):
    time.sleep(.5)  # give it a chance to start
    r = redis.StrictRedis(host='localhost', port=6379, db=0)
    r.set('foo', 'bar')
    return r.get('foo') == 'bar'
```

This function can be plugged into Circus as a *after_start* hook:

```
[watcher:queue-worker]
cmd = python -u worker.py
hooks.before_start = mycoolapp.myplugins.check_redis
priority = 2

[watcher:redis]
cmd = redis-server
priority = 1
```

Once Circus has started the **redis** watcher, it will start the **queue-worker** watcher, since it follows the **priority** ordering.

Just before starting the second watcher, it will run the **check_redis** function, and in case it returns **False** will abort the watcher starting process.

Available hooks are:

- **before_start**: called before the watcher is started. If the hook returns **False** the startup is aborted.
- **after_start**: called before the watcher is started. If the hook returns **False** the watcher is immediatly stopped and the startup is aborted.
- **before_stop**: called before the watcher is stopped. The hook result is ignored.
- **after_stop**: called before the watcher is stopped. The hook result is ignored.

5.4.1 Hook signature

A hook must follow this signature:

```
def hook(watcher, arbiter, hook_name):
    ...
```

Where **watcher** is the **Watcher** class instance, **arbiter** the **Arbiter** one, and **hook_name** the hook name.

You can ignore those but being able to use the watcher and/or arbiter data and methods can be useful in some hooks.

5.4.2 Hook events

Everytime a hook is run, its result is notified as an event in Circus.

There are two events related to hooks:

- **hook_success**:: a hook was successfully called. The event keys are **name** the name if the event, and **time**: the date of the events.
- **hook_failure**:: a hook has failed. The event keys are **name** the name if the event, **time**: the date of the events and **error**: the exception that occurred in the event, if any.

5.5 Command-line tools

5.5.1 circus-top

circus-top is a top-like console you can run to watch live your running Circus system. It will display the CPU, Memory usage and socket hits if you have some.

Example of output:

```
------------------------------------------------------------------------
circusd-stats
 PID                  CPU (%)              MEMORY (%)
14252                  0.8                  0.4
                       0.8 (avg)            0.4 (sum)

dummy
 PID                  CPU (%)              MEMORY (%)
14257                  78.6                 0.1
14256                  76.6                 0.1
14258                  74.3                 0.1
14260                  71.4                 0.1
14259                  70.7                 0.1
                       74.32 (avg)          0.5 (sum)
```

```
------------------------------------------------------------------------
```

circus-top is a read-only console. If you want to interact with the system, use *circusctl*.

5.5.2 circusctl

circusctl can be used to run any command listed in *Commands* . For example, you can get a list of all the watchers, you can do

```
$ circusctl list
```

5.6 Commands

At the epicenter of circus lives the command systems. *circusctl* is just a zeromq client, and if needed you can drive programmaticaly the Circus system by writing your own zmq client.

All messages are Json mappings.

For each command below, we provide a usage example with circusctl but also the input / output zmq messages.

5.6.1 circus-ctl commands

- **add**: *Add a watcher*
- **decr**: *Decrement the number of processes in a watcher*
- **dstats**: *Get circusd stats*
- **get**: *Get the value of a watcher option*
- **globaloptions**: *Get the arbiter options*
- **incr**: *Increment the number of processes in a watcher*
- **list**: *Get list of watchers or processes in a watcher*
- **listen**: *Suscribe to a watcher event*
- **listsockets**: *Get the list of sockets*
- **numprocesses**: *Get the number of processes*
- **numwatchers**: *Get the number of watchers*
- **options**: *Get the value of a watcher option*
- **quit**: *Quit the arbiter immediately*
- **reload**: *Reload the arbiter or a watcher*
- **restart**: *Restart the arbiter or a watcher*
- **rm**: *Remove a watcher*
- **set**: *Set a watcher option*
- **signal**: *Send a signal*
- **start**: *Start the arbiter or a watcher*
- **stats**: *Get process infos*

- **status**: *Get the status of a watcher or all watchers*
- **stop**: *Stop the arbiter or a watcher*

Add a watcher

This command add a watcher dynamically to a arbiter.

ZMQ Message

```
{
    "command": "add",
    "properties": {
        "cmd": "/path/to/commandline --option"
        "name": "nameofwatcher"
        "args": [],
        "options": {},
        "start": false
    }
}
```

A message contains 2 properties:

- cmd: Full command line to execute in a process
- args: array, arguments passed to the command (optional)
- name: name of watcher
- options: options of a watcher
- start: start the watcher after the creation

The response return a status "ok".

Command line

```
$ circusctl add [--start] <name> <cmd>
```

Options

- <name>: name of the watcher to create
- <cmd>: full command line to execute in a process
- –start: start the watcher immediately

Decrement the number of processes in a watcher

This comment decrement the number of processes in a watcher by -1.

ZMQ Message

```
{
    "command": "decr",
    "propeties": {
        "name": "<watchername>"
        "nb": <nbprocess>
    }
}
```

The response return the number of processes in the ‘numprocesses‘ property:

```
{ "status": "ok", "numprocesses": <n>, "time", "timestamp" }
```

Command line

```
$ circusctl descr <name> [<nbprocess>]
```

Options

- <name>: name of the watcher
- <nbprocess>: the number of processes to remove.

Get circusd stats

You can get at any time some statistics about circusd with the dstat command.

ZMQ Message

To get the circusd stats, simply run:

```
{
    "command": "dstats"
}
```

The response returns a mapping the property “infos” containing some process informations:

```
{
  "info": {
    "children": [],
    "cmdline": "python",
    "cpu": 0.1,
    "ctime": "0:00.41",
    "mem": 0.1,
    "mem_info1": "3M",
    "mem_info2": "2G",
    "nice": 0,
    "pid": 47864,
    "username": "root"
  },
  "status": "ok",
  "time": 1332265655.897085
}
```

Command Line

```
$ circusctl dstats
```

Get the value of a watcher option

This command return the watchers options values asked.

ZMQ Message

```
{
    "command": "get",
    "properties": {
        "keys": ["key1, "key2"]
        "name": "nameofwatcher"
    }
}
```

A response contains 2 properties:

- keys: list, The option keys for which you want to get the values
- name: name of watcher

The response return an object with a property "options" containing the list of key/value returned by circus.

eg:

```
{
    "status": "ok",
    "options": {
        "flapping_window": 1,
        "times": 2
    },
    time': 1332202594.754644
}
```

See Optios for for a description of options enabled?

Command line

```
$ circusctl get <name> <key> <value> <key1> <value1>
```

Get the arbiter options

This command return the arbiter options

ZMQ Message

```
{
    "command": "globaloptions",
    "properties": {
```

```
        "key1": "val1",
        ..
    }
}
```

A message contains 2 properties:

- keys: list, The option keys for which you want to get the values

The response return an object with a property "options" containing the list of key/value returned by circus.

eg:

```
{
    "status": "ok",
    "options": {
        "check_delay": 1,
        ...
    },
    time': 1332202594.754644
}
```

Command line

```
$ circusctl globaloptions
```

Options

Options Keys are:

- endpoint: the controller ZMQ endpoint
- pubsub_endpoint: the pubsub endpoint
- check_delay: the delay between two controller points

Increment the number of processes in a watcher

This comment increment the number of processes in a watcher by +1.

ZMQ Message

```
{
    "command": "incr",
    "properties": {
        "name": "<watchername>",
        "nb": <nbprocess>
    }
}
```

The response return the number of processes in the 'numprocesses' property:

```
{ "status": "ok", "numprocesses": <n>, "time", "timestamp" }
```

Command line

```
$ circusctl incr <name> [<nbprocess>]
```

Options

- <name>: name of the watcher.
- <nbprocess>: the number of processes to add.

Get list of watchers or processes in a watcher

ZMQ Message

To get the list of all the watchers:

```
{
    "command": "list",
}
```

To get the list of active processes in a watcher:

```
{
    "command": "list",
    "properties": {
        "name": "nameofwatcher",
    }
}
```

The response return the list asked. the mapping returned can either be 'watchers' or 'pids' depending the request.

Command line

```
$ circusctl list [<name>]
```

Suscribe to a watcher event

ZMQ

At any moment you can suscribe to circus event. Circus provide a PUB/SUB feed on which any clients can suscribe. The suscriber endpoint URI is set in the circus.ini configuration file.

Events are pubsub topics:

Events are pubsub topics:

- *watcher.*<*watchername*>*.reap*: when a process is reaped
- *watcher.*<*watchername*>*.spawn*: when a process is spawned
- *watcher.*<*watchername*>*.kill*: when a process is killed
- *watcher.*<*watchername*>*.updated*: when watcher configuration is updated
- *watcher.*<*watchername*>*.stop*: when a watcher is stopped

- *watcher.<watchername>.start*: when a watcher is started

All events messages are in a json.

Command line

The client has been updated to provide a simple way to listen on the events:

```
circusctl list [<topic>, ...]
```

Example of result:

```
$ circusctl listen tcp://127.0.0.1:5556
watcher.refuge.spawn: {u'process_id': 6, u'process_pid': 72976,
                       u'time': 1331681080.985104}
watcher.refuge.spawn: {u'process_id': 7, u'process_pid': 72995,
                       u'time': 1331681086.208542}
watcher.refuge.spawn: {u'process_id': 8, u'process_pid': 73014,
                       u'time': 1331681091.427005}
```

Get the list of sockets

ZMQ Message

To get the list of sockets:

```
{
    "command": "listsockets",
}
```

The response return a list of json mappings with keys for fd, name, host and port.

Command line

```
$ circusctl listsockets
```

Get the number of processes

Get the number of processes in a watcher or in a arbiter

ZMQ Message

```
{
    "command": "numprocesses",
    "propeties": {
        "name": "<watchername>"
    }

}
```

The response return the number of processes in the ‘numprocesses‘ property:

```
{ "status": "ok", "numprocesses": <n>, "time", "timestamp" }
```

If the property name isn't specified, the sum of all processes managed is returned.

Command line

```
$ circusctl numprocesses [<name>]
```

Options

- <name>: name of the watcher

Get the number of watchers

Get the number of watchers in a arbiter

ZMQ Message

```
{
    "command": "numwatchers",
}
```

The response return the number of watchers in the 'numwatchers' property:

```
{ "status": "ok", "numwatchers": <n>, "time", "timestamp" }
```

Command line

```
$ circusctl numwatchers
```

Get the value of a watcher option

This command return the watchers options values asked.

ZMQ Message

```
{
    "command": "options",
    "properties": {
        "name": "nameofwatcher",
    }
}
```

A message contains 1 property:

- name: name of watcher

The response return an object with a property "options" containing a dictionary of key/value returned by circus.

eg:

```
{
    "status": "ok",
    "options": {
        "flapping_window": 1,
        "flapping_attempts": 2,
        ...
    },
    time': 1332202594.754644
}
```

Command line

```
$ circusctl options <name>
```

Options

- <name>: name of the watcher

Options Keys are:

- numprocesses: integer, number of processes
- warmup_delay: integer or number, delay to wait between process spawning in seconds
- working_dir: string, directory where the process will be executed
- uid: string or integer, user ID used to launch the process
- gid: string or integer, group ID used to launch the process
- send_hup: boolean, if TRU the signal HUP will be used on reload
- shell: boolean, will run the command in the shell environment if true
- cmd: string, The command line used to launch the process
- env: object, define the environnement in which the process will be launch
- flapping_attempts: integer, number of times we try to relaunch a process in the flapping_window time before we stop the watcher during the retry_in time.
- flapping_window: integer or number, times in seconds in which we test the number of process restart.
- retry_in: integer or number, time in seconds we wait before we retry to launch the process if the maximum number of attempts has been reach.
- max_retry: integer, The maximum of retries loops
- graceful_timeout: integer or number, time we wait before we definitely kill a process.
- priority: used to sort watchers in the arbiter
- singleton: if True, a singleton watcher.
- max_age: time a process can live before being restarted
- max_age_variance: variable additional time to live, avoids stampeding herd.

Quit the arbiter immediately

When the arbiter receive this command, the arbiter exit.

ZMQ Message

```
{
    "command": "quit"
}
```

The response return the status “ok”.

Command line

```
$ circusctl quit
```

Reload the arbiter or a watcher

This command reload all the process in a watcher or all watchers. If a the option send_hup is set to true in a watcher then the HUP signal will be sent to the process.A graceful reload follow the following process:

1. Send a SIGQUIT signal to a process
2. Wait until graceful timeout
3. Send a SIGKILL signal to the process to make sure it is finally killed.

ZMQ Message

```
{
    "command": "reload",
    "propeties": {
        "name": '<name>",
        "graceful": true
    }
}
```

The response return the status “ok”. If the property graceful is set to true the processes will be exited gracefully.

If the property name is present, then the reload will be applied to the watcher.

Command line

```
$ circusctl reload [<name>] [--terminate]
```

Options

- <name>: name of the watcher
- –terminate; quit the node immediately

Restart the arbiter or a watcher

This command restart all the process in a watcher or all watchers. This funtion simply stop a watcher then restart it.

ZMQ Message

```
{
    "command": "restart",
    "propeties": {
        "name": '<name>"
    }
}
```

The response return the status "ok".

If the property name is present, then the reload will be applied to the watcher.

Command line

```
$ circusctl restart [<name>] [--terminate]
```

Options

- <name>: name of the watcher
- –terminate; quit the node immediately

Remove a watcher

This command remove a watcher dynamically from the arbiter. The watchers are gracefully stopped.

ZMQ Message

```
{
    "command": "rm",
    "properties": {
        "name": "nameofwatcher",
    }
}
```

A message contains 1 property:

- name: name of watcher

The response return a status "ok".

Command line

```
$ circusctl rm <name>
```

Options

- <name>: name of the watcher to create

Set a watcher option

ZMQ Message

```
{
    "command": "set",
    "properties": {
        "name": "nameofwatcher",
        "options": {
            "key1": "val1",
        }
        ..
    }
}
```

The response return the status "ok". See the command Options for a list of key to set.

Command line

```
$ circusctl set <name> <key1> <value1> <key2> <value2>
```

Send a signal

This command allows you to send a signal to all processes in a watcher, a specific process in a watcher or its children.

ZMQ Message

To send a signal to all the processes for a watcher:

```
{
    "command": "signal",
    "property": {
        "name": <name>,
        "signum": <signum>
}
```

To send a signal to a process:

```
{
    "command": "signal",
    "property": {
        "name": <name>,
        "pid": <processid>,
        "signum": <signum>
}
```

An optional property "children" can be used to send the signal to all the children rather than the process itself:

```
{
    "command": "signal",
    "property": {
        "name": <name>,
        "pid": <processid>,
        "signum": <signum>,
```

```
        "children": True
}
```

To send a signal to a process child:

```
{
    "command": "signal",
    "property": {
        "name": <name>,
        "pid": <processid>,
        "signum": <signum>,
        "child_pid": <childpid>,
}
```

It is also possible to send a signal to all the processes of the watcher and its childs:

```
{
    "command": "signal",
    "property": {
        "name": <name>,
        "signum": <signum>,
        "children": True
}
```

Last, you can send a signal to the process *and* its children, with the *recursive* option:

```
{
    "command": "signal",
    "property": {
        "name": <name>,
        "signum": <signum>,
        "recursive": True
}
```

Command line

```
$ circusctl signal <name> [<process>] [<pid>] [--children]
        [recursive] <signum>
```

Options:

- <name>: the name of the watcher
- <pid>: integer, the process id.
- <signum>: the signal number to send.
- <childpid>: the pid of a child, if any
- <children>: boolean, send the signal to all the children
- <recursive>: boolean, send the signal to the process and its children

Allowed signals are:

- 3: QUIT
- 15: TERM
- 9: KILL

- 1: HUP
- 21: TTIN
- 22: TTOU
- 30: USR1
- 31: USR2

Start the arbiter or a watcher

This command starts all the processes in a watcher or all watchers.

ZMQ Message

```
{
    "command": "start",
    "properties": {
        "name": '<name>",
    }
}
```

The response return the status "ok".

If the property name is present, the watcher will be started.

Command line

```
$ circusctl start [<name>]
```

Options

- <name>: name of the watcher

Get process infos

You can get at any time some statistics about your processes with the stat command.

ZMQ Message

To get stats for all watchers:

```
{
    "command": "stats"
}
```

To get stats for a watcher:

```
{
    "command": "stats",
    "properties": {
        "name": <name>
```

```
    }
}
```

To get stats for a process:

```
{
    "command": "stats",
    "properties": {
        "name": <name>,
        "process": <processid>
    }
}
```

The response retun an object per process with the property "info" containing some process informations:

```
{
  "info": {
    "children": [],
    "cmdline": "python",
    "cpu": 0.1,
    "ctime": "0:00.41",
    "mem": 0.1,
    "mem_info1": "3M",
    "mem_info2": "2G",
    "nice": 0,
    "pid": 47864,
    "username": "root"
  },
  "process": 5,
  "status": "ok",
  "time": 1332265655.897085
}
```

Command Line

```
$ circusctl stats [<watchername>] [<processid>]
```

Get the status of a watcher or all watchers

This command start get the status of a watcher or all watchers.

ZMQ Message

```
{
    "command": "status",
    "properties": {
        "name": '<name>",
    }
}
```

The response return the status "active" or "stopped" or the status / watchers.

Command line

```
$ circusctl status [<name>]
```

Options

- <name>: name of the watcher

Example

```
$ circusctl status dummy
active
$ circusctl status
dummy: active
dummy2: active
refuge: active
```

Stop the arbiter or a watcher

This command stop all the process in a watcher or all watchers.

ZMQ Message

```
{
    "command": "stop",
    "propeties": {
        "name": '<name>",
    }
}
```

The response return the status "ok".

If the property name is present, then the reload will be applied to the watcher.

Command line

```
$ circusctl stop [<name>]
```

Options

- <name>: name of the watcher

5.7 The Web Console

Circus comes with a Web Console that can be used to manage the system.

The Web Console lets you:

- Connect to any running Circus system
- Watch the processes CPU and Memory usage in real-time

- Add or kill processes
- Add new watchers

Note: The real-time CPU & Memory usage feature uses the stats socket. If you want to activate it, make sure the Circus system you'll connect to has the stats enpoint enabled in its configuration:

```
[circus]
...
stats_endpoint = tcp://127.0.0.1:5557
...
```

By default, this option is not activated.

The web console needs a few dependencies that you can install them using the web-requirements.txt file. Additionally, you will need to have gevent (and thus libevent) installed on your system to make this working:

```
$ bin/pip install -r web-requirements.txt
```

To enable the console, add a few options in the Circus ini file:

```
[circus]
httpd = True
httpd_host = localhost
httpd_port = 8080
```

httpd_host and *httpd_port* are optional, and default to *localhost* and *8080*.

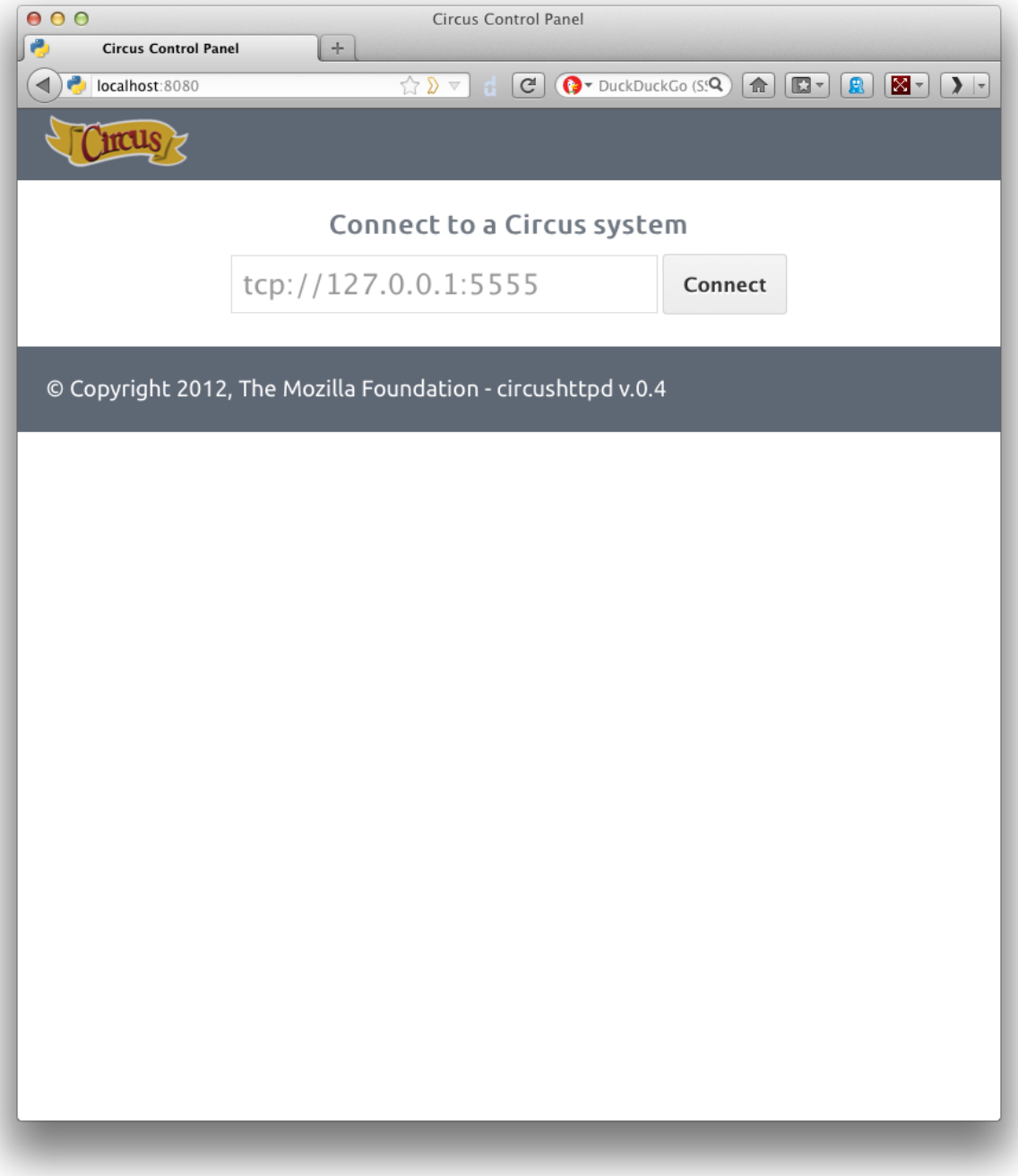
If you want to run the web app on its own, just run the **circushttpd** script:

```
$ circushttpd
Bottle server starting up...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

By default the script will run the Web Console on port 8080, but the –port option can be used to change it.

5.7.1 Using the console

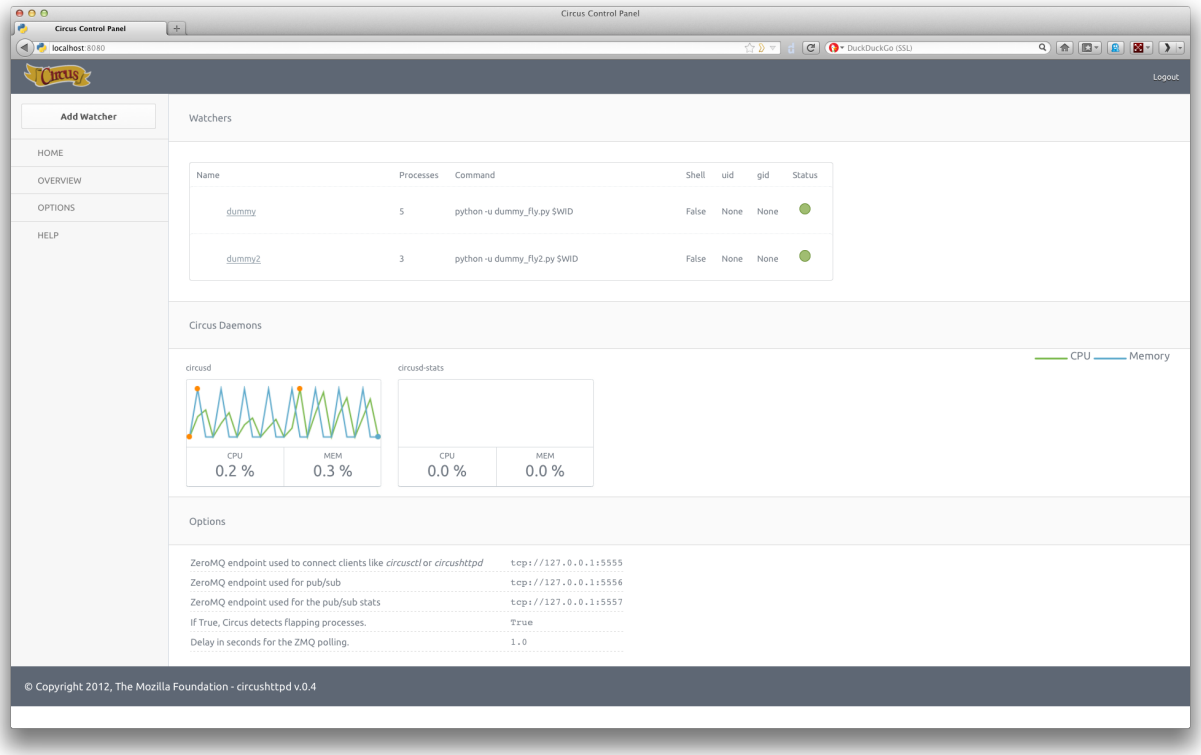
Once the script is running, you can open a browser and visit *http://localhost:8080*. You should get this screen:



The Web Console is ready to be connected to a Circus system, given its **endpoint**. By default the endpoint is *tcp://127.0.0.1:5555*.

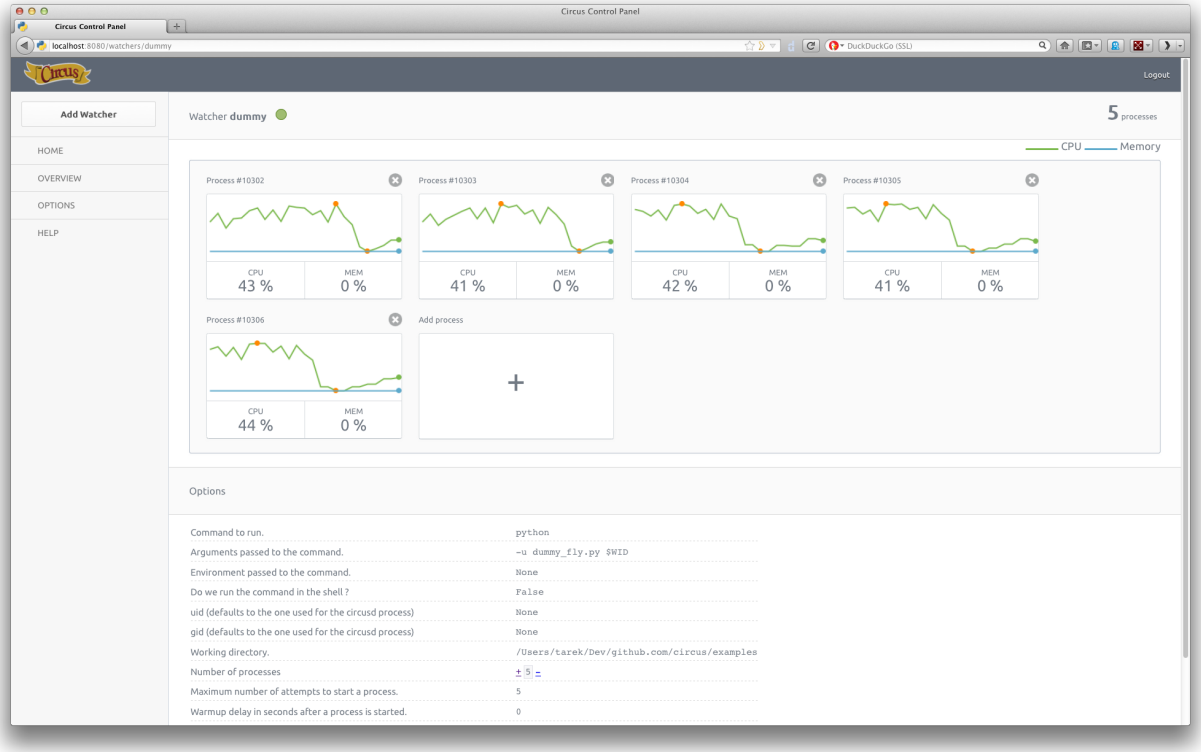
Once you hit *Connect*, the web application will connect to the Circus system.

With the Web Console logged in, you should get a list of watchers, and a real-time status of the two Circus processes (circusd and circusd-stats).



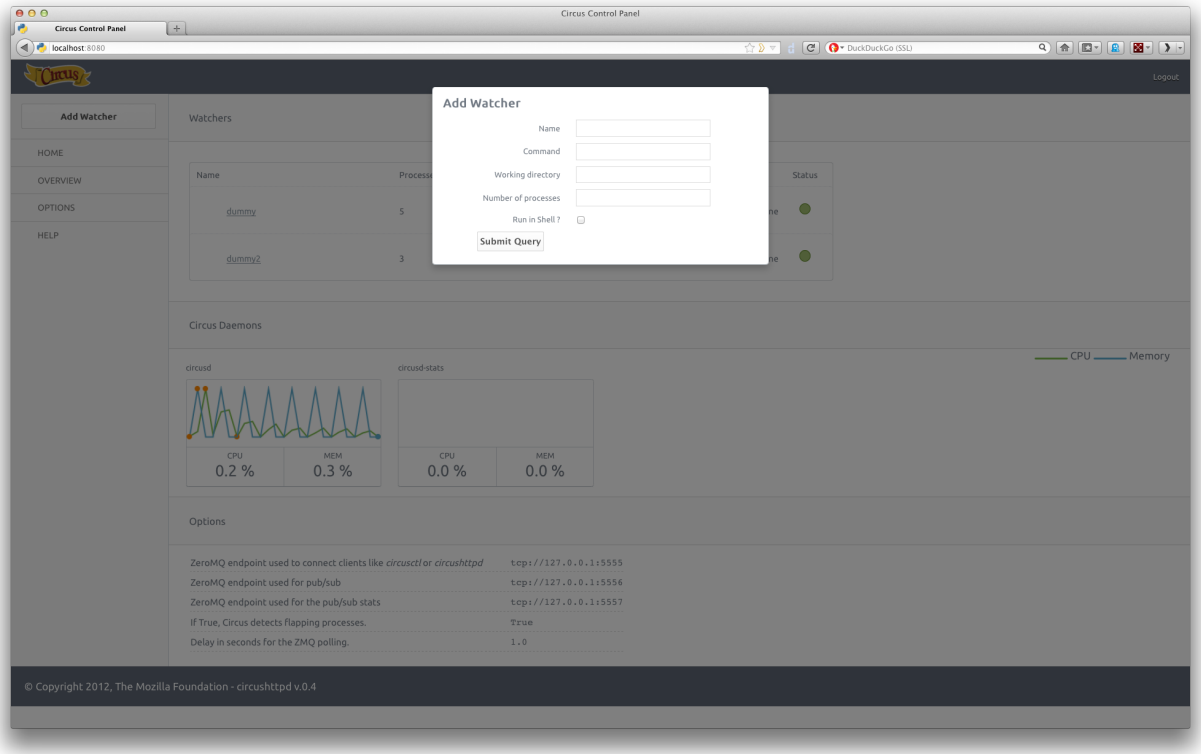
You can click on the status of each watcher to toggle it from **Active** (green) to **Inactive** (red). This change is effective immediatly and let you start & stop watchers.

If you click on the watcher name, you will get a web page for that particular watcher, with its processes:



On this screen, you can add or remove processes, and kill existing ones.

Last but not least, you can add a brand new watcher by clicking on the *Add Watcher* link in the left menu:



5.7.2 Embedding circushttpd into Circus

circushttpd is a WSGI application so you can run it with any web server that's compatible with that protocol. By default it uses the standard library **wsgiref** server, but that server does not really support any load.

You can use Chaussette to bind a WSGI server and have *circushttpd* managed by Circus itself.

To do so, make sure Chaussette is installed:

```
$ pip install chaussette
```

Then add a new *watcher* and a *socket* sections in your ini file:

```
[watcher:webconsole]
cmd = chaussette --fd $(circus.sockets.webconsole) circus.web.circushttpd.app
singleton = 1
use_sockets = 1

[socket:webconsole]
host = 127.0.0.0
port = 8080
```

That's it !

5.7.3 Running behind Nginx and Varnish

Nginx can act as a proxy in front of Circus. It an also deal with security.

To hook Nginx, you define a *location* directive that proxies the calls to Circus.

Example:

```
location ~/media/*(.jpg|.css|.js)$ {
    alias /path/to/circus/web/;
}

location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://127.0.0.1:8080;
}
```

If you want more configuration options, see http://wiki.nginx.org/HttpProxyModule.

Websockets in Nginx (v1.2.5) is currently unsupported, although it will be implemented in 1.3. To receive real-time statuses and graphs in the web console, you need to use a websocket-compatible proxy like Varnish or HAProxy. In Varnish, two backends can be defined: one for serving the web console and one for the handling the socket connections.

Example:

```
backend default {
    .host = "127.0.0.1";
    .port = "8001";
}

backend socket {
    .host = "127.0.0.1";
    .port = "8080";
    .connect_timeout = 1s;
    .first_byte_timeout = 2s;
    .between_bytes_timeout = 60s;
}

sub vcl_pipe {
     if (req.http.upgrade) {
         set bereq.http.upgrade = req.http.upgrade;
     }
}

sub vcl_recv {
    if (req.http.Upgrade ~ "(?i)websocket") {
        set req.backend = socket;
      return (pipe);
    }
}
```

Here, web console requests are bound to port 8001, and Nginx should be configured to listen on that port. Websocket connections are upgraded and piped directly to the circushttpd process listening on port 8080.

5.7.4 Password-protect circushttpd

As explained in the *Security* page, running *circushttpd* is pretty unsafe. We don't provide any security in Circus itself, but you can protect your console at the NGinx level, by using http://wiki.nginx.org/HttpAuthBasicModule

Example:

```
location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://127.0.0.1:8080;
    auth_basic            "Restricted";
    auth_basic_user_file  /path/to/htpasswd;
}
```

The **htpasswd** file contains users and their passwords, and a password prompt will pop when you access the console.

You can use Apache's htpasswd script to edit it, or the Python script they provide at: http://trac.edgewall.org/browser/trunk/contrib/htpasswd.py

Of course that's just one way to protect your web console, you could use many other techniques.

5.7.5 Extending the web console

We picked *bottle* to build the webconsole, mainly because it's a really tiny framework that doesn't do much. By having a look at the code of the web console, you'll eventually find out that it's really simple to understand.

Here is how it's split:

- The *circushttpd.py* file contains the "views" definitions and some code to handle the socket connection (via socketio).
- the *controller.py* contains a single class which is in charge of doing the communication with the circus controller. It allows to have a nicer high level API when defining the web server.

If you want to add a feature in the web console you can reuse the code that's existing. A few tools are at your disposal to ease the process:

- There is a *render_template* function, which takes the named arguments you pass to it and pass them to the template renderer and return the resulting HTML. It also passes some additional variables, such as the session, the circus version and the client if defined.
- If you want to run commands and doa redirection depending the result of it, you can use the *run_command* function, which takes a callable as a first argument, a message in case of success and a redirection url.

The `StatsNamespace` class is responsible for managing the websocket communication on the server side. Its documentation should help you to understand what it does.

5.8 Circus Sockets

Circus can bind network sockets and manage them as it does for processes.

The main idea is that a child process that's created by Circus to run one of the watcher's command can inherit from all the opened file descriptors.

That's how Apache or Unicorn works, and many other tools out there.

5.8.1 Goal

The goal of having sockets managed by Circus is to be able to manage network applications in Circus exactly like other applications.

For example, if you use Circus with Chaussette – a WGSI server, you can get a very fast web server running and manage *"Web Workers"* in Circus as you would do for any other process.

Splitting the socket managment from the network application itself offers a lot of opportunities to scale and manage your stack.

5.8.2 Design

The gist of the feature is done by binding the socket and start listening to it in **circusd**:

```
import socket

sock = socket.socket(FAMILY, TYPE)
sock.bind((HOST, PORT))
sock.listen(BACKLOG)
fd = sock.fileno()
```

Circus then keeps track of all the opened fds, and let the processes it runs as children have access to them if they want.

If you create a small Python network script that you intend to run in Circus, it could look like this:

```
import socket
import sys

fd = int(sys.argv[1])    # getting the FD from circus
sock = socket.fromfd(fd, FAMILY, TYPE)

# dealing with one request at a time
while True:
    conn, addr = sock.accept()
    request = conn.recv(1024)
    .. do something ..
    conn.sendall(response)
    conn.close()
```

Then Circus could run like this:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:dummy]
cmd = mycoolscript $(circus.sockets.foo)
use_sockets = True
warmup_delay = 0
numprocesses = 5

[socket:foo]
host = 127.0.0.1
port = 8888
```

$(circus.sockets.foo) will be replaced by the FD value once the socket is created and bound on the 8888 *port*.

5.8.3 Real-world example

Chaussette is the perfect Circus companion if you want to run your WSGI application.

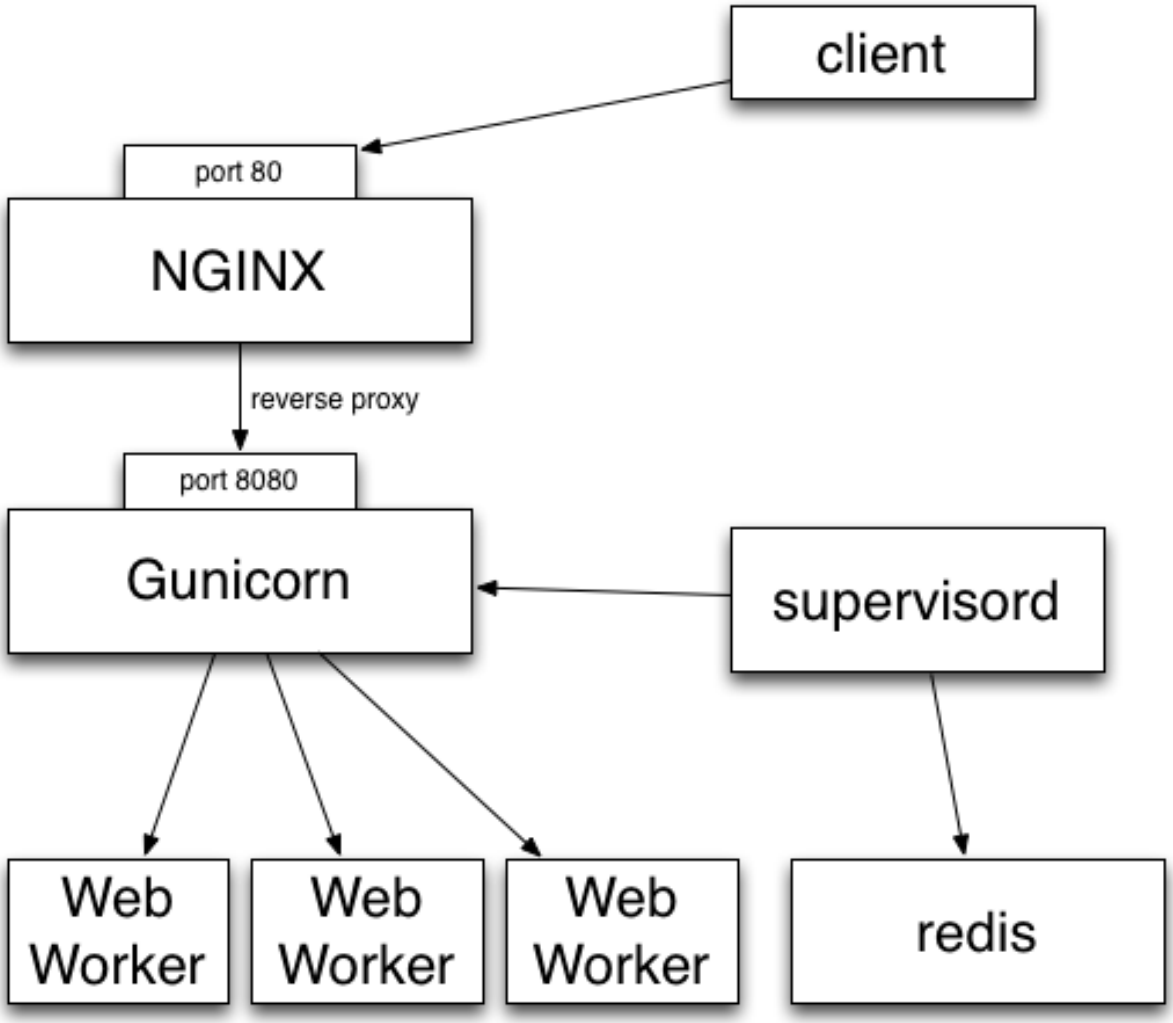
Once it's installed, running 5 **meinheld** workers can be done by creating a socket and calling the **chaussette** command in a worker, like this:

```
[circus]
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:web]
cmd = chaussette --fd $(circus.sockets.web) --backend meinheld mycool.app
use_sockets = True
numprocesses = 5

[socket:web]
host = 0.0.0.0
port = 8000
```

We did not publish benchmarks yet, but a Web cluster managed by Circus with a Gevent or Meinheld backend is as fast as any pre-fork WSGI server out there.

5.8.4 Circus stack v.s. Classical stack

In a classical WSGI stack, you have a server like Gunicorn that serves on a port or an unix socket and is usually deployed behind a web server like Nginx:



Clients call Nginx, which reverse proxies all the calls to Gunicorn.

If you want to make sure the Gunicorn process stays up and running, you have to use a program like Supervisord or upstart.

Gunicorn in turn watches for its processes ("workers").

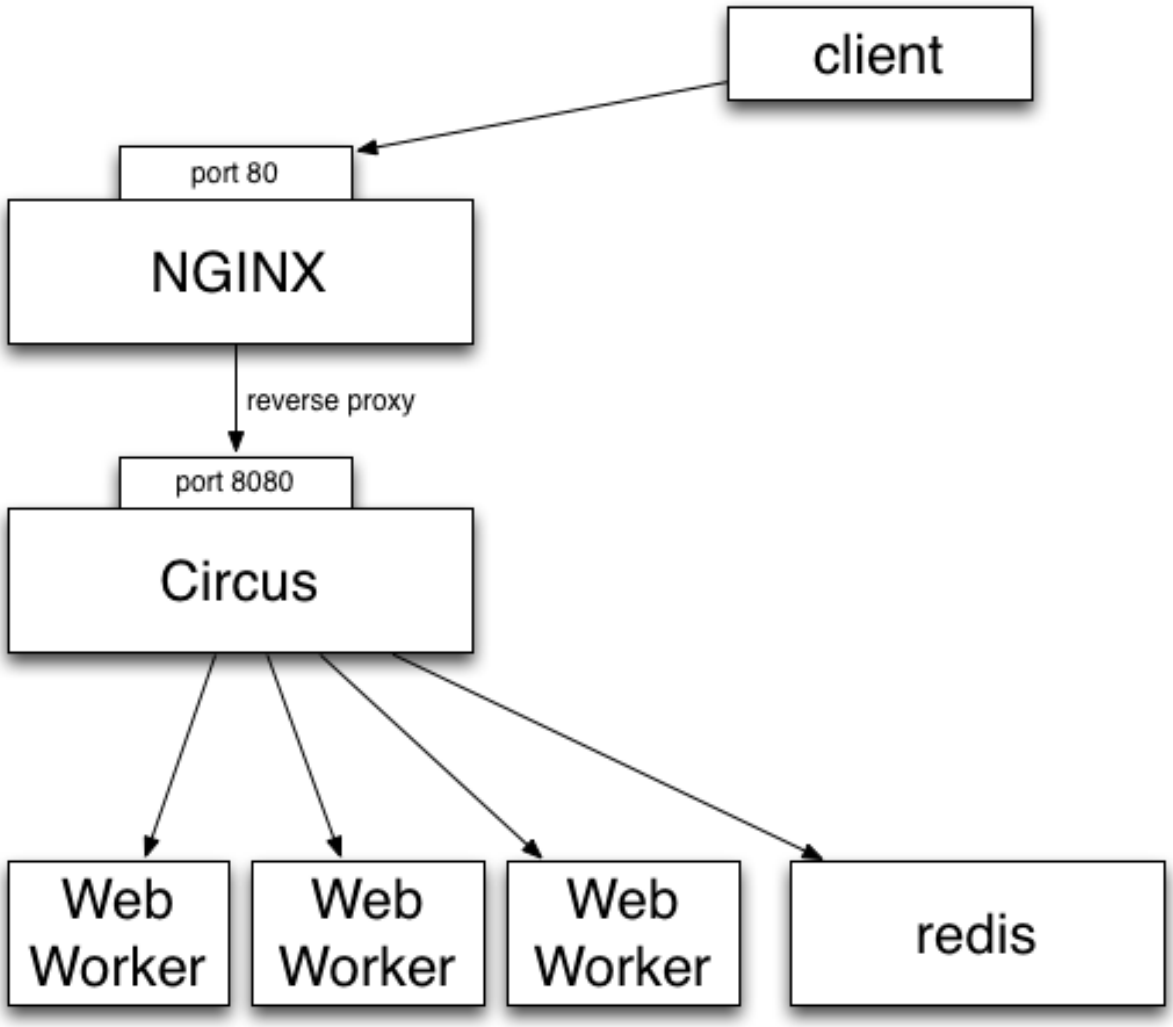
In other words you are using two levels of process managment. One that you manage and control (supervisord), and a second one that you have to manage in a different UI, with a different philosophy and less control over what's going on (the wsgi server's one)

This is true for Gunicorn and most multi-processes WSGI servers out there I know about. uWsgi is a bit different as it offers plethoras of options.

But if you want to add a Redis server in your stack, you *will* end up with managing your stack processes in two different places.

Circus' approach on this is to manage processes *and* sockets.

A Circus stack can look like this:



So, like Gunicorn, Circus is able to bind a socket that will be proxied by Nginx. Circus don't deal with the requests but simply binds the socket. It's then up to a web worker process to accept connections on the socket and do the work.

It provides equivalent features than Supervisord but will also let you manage all processes at the same level, wether they are web workers or Redis or whatever. Adding a new web worker is done exactly like adding a new Redis process.

Benches

We did a few benches to compare Circus & Chaussette with Gunicorn. To summarize, Circus is not adding any overhead and you can pick up many different backends for your web workers.

See:

- http://blog.ziade.org/2012/06/28/wgsi-web-servers-bench
- http://blog.ziade.org/2012/07/03/wsgi-web-servers-bench-part-2

5.9 Circus Library

The Circus package is composed of a high-level `get_arbiter()` function and many classes. In most cases, using the high-level function should be enough, as it creates everything that is needed for Circus to run.

You can subclass Circus' classes if you need more granularity than what is offered by the configuration.

5.9.1 The get_arbiter function

`get_arbiter()` is just a convenience on top of the various circus classes. It creates a *arbiter* (class `Arbiter`) instance with the provided options, which in turn runs a single `Watcher` with a single `Process`.

`circus.`**`get_arbiter`**(*watchers*, *controller=None*, *pubsub_endpoint=None*, *stats_endpoint=None*, *env=None*, *name=None*, *context=None*, *background=False*, *stream_backend='thread'*, *plugins=None*, *debug=False*, *proc_name='circusd'*)

Creates a Arbiter and a single watcher in it.

Options:

- **watchers** – a list of watchers. A watcher in that case is a dict containing:
  - **–name** – the name of the watcher (default: None)
  - **–cmd** – the command line used to run the Watcher.
  - **–args** – the args for the command (list or string).
  - **–executable** – When executable is given, the first item in the args sequence obtained from **cmd** is still treated by most programs as the command name, which can then be different from the actual executable name. It becomes the display name for the executing program in utilities such as **ps**.
  - **–numprocesses** – the number of processes to spawn (default: 1).
  - **–warmup_delay** – the delay in seconds between two spawns (default: 0)
  - **–shell** – if True, the processes are run in the shell (default: False)
  - **–working_dir** - the working dir for the processes (default: None)
  - **–uid** – the user id used to run the processes (default: None)
  - **–gid** – the group id used to run the processes (default: None)
  - **–env** – the environment passed to the processes (default: None)
  - **–send_hup**: if True, a process reload will be done by sending the SIGHUP signal. (default: False)
  - **–stdout_stream**: a mapping containing the options for configuring the stdout stream. Default to None. When provided, may contain:
    - ∗**class**: the fully qualified name of the class to use for streaming. Defaults to circus.stream.FileStream
    - ∗**refresh_time**: the delay between two stream checks. Defaults to 0.3 seconds.
    - ∗any other key will be passed the class constructor.
  - **–stderr_stream**: a mapping containing the options for configuring the stderr stream. Default to None. When provided, may contain:
    - ∗**class**: the fully qualified name of the class to use for streaming. Defaults to circus.stream.FileStream
    - ∗**refresh_time**: the delay between two stream checks. Defaults to 0.3 seconds.

∗any other key will be passed the class constructor.

–**max_retry**: the number of times we attempt to start a process, before we abandon and stop the whole watcher. (default: 5)

–**hooks**: callback functions for hooking into the watcher startup and shutdown process. **hooks** is a dict where each key is the hook name and each value is a 2-tuple with the name of the callable or the callabled itself and a boolean flag indicating if an exception occuring in the hook should not be ignored. Possible values for the hook name: *before_start*, *after_start*, *before_stop*, *after_stop*.

•**controller** – the zmq entry point (default: 'tcp://127.0.0.1:5555')

•**pubsub_endpoint** – the zmq entry point for the pubsub (default: 'tcp://127.0.0.1:5556')

•**stats_endpoint** – the stats endpoint. If not provided, the *circusd-stats* process will not be launched. (default: None)

•**context** – the zmq context (default: None)

•**background** – If True, the arbiter is launched in a thread in the background (default: False)

•**stream_backend** – the backend that will be used for the streaming process. Can be *thread* or *gevent*. When set to *gevent* you need to have *gevent* and *gevent_zmq* installed. (default: thread)

•**plugins** – a list of plugins. Each item is a mapping with:

–**use** – Fully qualified name that points to the plugin class

–every other value is passed to the plugin in the **config** option

•**debug** – If True the arbiter is launched in debug mode (default: False)

•**proc_name** – the arbiter process name (default: circusd)

Example:

```
from circus import get_arbiter

arbiter = get_arbiter({"cmd": "myprogram", "numprocesses": 3})
try:
    arbiter.start()
finally:
    arbiter.stop()
```

5.9.2 The classes collection

Circus provides a series of classes you can use to implement your own process manager:

- `Process`: wraps a running process and provides a few helpers on top of it.
- `Watcher`: run several instances of `Process` against the same command. Manage the death and life of processes.
- `Arbiter`: manages several `Watcher`.

class `circus.process.`**`Process`**(*wid*, *cmd*, *args=None*, *working_dir=None*, *shell=False*, *uid=None*, *gid=None*, *env=None*, *rlimits=None*, *executable=None*, *use_fds=False*, *watcher=None*, *spawn=True*)

Wraps a process.

Options:

•**wid**: the process unique identifier. This value will be used to replace the *$WID* string in the command line if present.

- **cmd**: the command to run. May contain any of the variables available that are being passed to this class. They will be replaced using the python format syntax.
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted using `shlex.split()`. Defaults to None.
- **executable**: When executable is given, the first item in the args sequence obtained from **cmd** is still treated by most programs as the command name, which can then be different from the actual executable name. It becomes the display name for the executing program in utilities such as **ps**.
- **working_dir**: the working directory to run the command in. If not provided, will default to the current working directory.
- **shell**: if *True*, will run the command in the shell environment. *False* by default. **warning: this is a security hazard**.
- **uid**: if given, is the user id or name the command should run with. The current uid is the default.
- **gid**: if given, is the group id or name the command should run with. The current gid is the default.
- **env**: a mapping containing the environment variables the command will run with. Optional.
- **rlimits**: a mapping containing rlimit names and values that will be set before the command runs.
- **use_fds**: if True, will not close the fds in the subprocess. default: False.

pid
Return the *pid*

stdout
Return the *stdout* stream

stderr
Return the *stdout* stream

send_signal(**args*, ***kw*)
Sends a signal **sig** to the process.

stop(**args*, ***kw*)
Terminate the process.

age()
Return the age of the process in seconds.

info()
Return process info.

The info returned is a mapping with these keys:

- **mem_info1**: Resident Set Size Memory in bytes (RSS)
- **mem_info2**: Virtual Memory Size in bytes (VMS).
- **cpu**: % of cpu usage.
- **mem**: % of memory usage.
- **ctime**: process CPU (user + system) time in seconds.
- **pid**: process id.
- **username**: user name that owns the process.
- **nice**: process niceness (between -20 and 20)
- **cmdline**: the command line the process was run with.

children()
Return a list of children pids.

is_child(*pid*)
Return True is the given *pid* is a child of that process.

send_signal_child(**args*, ***kw*)
Send signal *signum* to child *pid*.

send_signal_children(**args*, ***kw*)
Send signal *signum* to all children.

status
Return the process status as a constant

- RUNNING
- DEAD_OR_ZOMBIE
- UNEXISTING
- OTHER

Example:

```
>>> from circus.process import Process
>>> process = Process('Top', 'top', shell=True)
>>> process.age()
3.0107998847961426
>>> process.info()
'Top: 6812  N/A tarek Zombie N/A N/A N/A N/A N/A'
>>> process.status
1
>>> process.stop()
>>> process.status
2
>>> process.info()
'No such process (stopped?)'
```

class circus.watcher.**Watcher**(*name*, *cmd*, *args=None*, *numprocesses=1*, *warmup_delay=0.0*, *working_dir=None*, *shell=False*, *uid=None*, *max_retry=5*, *gid=None*, *send_hup=False*, *env=None*, *stopped=True*, *graceful_timeout=30.0*, *prereload_fn=None*, *rlimits=None*, *executable=None*, *stdout_stream=None*, *stderr_stream=None*, *stream_backend='thread'*, *priority=0*, *singleton=False*, *use_sockets=False*, *copy_env=False*, *copy_path=False*, *max_age=0*, *max_age_variance=30*, *hooks=None*, *respawn=True*, ***options*)
Class managing a list of processes for a given command.

Options:

- **name**: name given to the watcher. Used to uniquely identify it.
- **cmd**: the command to run. May contain *$WID*, which will be replaced by **wid**.
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted using `shlex.split()`. Defaults to None.
- **numprocesses**: Number of processes to run.
- **working_dir**: the working directory to run the command in. If not provided, will default to the current working directory.

•**shell**: if *True*, will run the command in the shell environment. *False* by default. **warning: this is a security hazard**.

•**uid**: if given, is the user id or name the command should run with. The current uid is the default.

•**gid**: if given, is the group id or name the command should run with. The current gid is the default.

•**send_hup**: if True, a process reload will be done by sending the SIGHUP signal. Defaults to False.

•**env**: a mapping containing the environment variables the command will run with. Optional.

•**rlimits**: a mapping containing rlimit names and values that will be set before the command runs.

•**stdout_stream**: a mapping that defines the stream for the process stdout. Defaults to None.

Optional. When provided, *stdout_stream* is a mapping containing up to three keys:

–**class**: the stream class. Defaults to *circus.stream.FileStream*

–**filename**: the filename, if using a FileStream

–**refresh_time**: the delay between two stream checks. Defaults to 0.3 seconds.

–**max_bytes**: maximum file size, after which a new output file is opened. defaults to 0 which means no maximum size.

–**backup_count**: how many backups to retain when rotating files according to the max_bytes parameter. defaults to 0 which means no backups are made.

This mapping will be used to create a stream callable of the specified class. Each entry received by the callable is a mapping containing:

–**pid** - the process pid

–**name** - the stream name (*stderr* or *stdout*)

–**data** - the data

•**stderr_stream**: a mapping that defines the stream for the process stderr. Defaults to None.

Optional. When provided, *stderr_stream* is a mapping containing up to three keys: - **class**: the stream class. Defaults to *circus.stream.FileStream* - **filename**: the filename, if using a FileStream - **refresh_time**: the delay between two stream checks. Defaults

to 0.3 seconds.

–**max_bytes**: maximum file size, after which a new output file is opened. defaults to 0 which means no maximum size.

–**backup_count**: how many backups to retain when rotating files according to the max_bytes parameter. defaults to 0 which means no backups are made.

This mapping will be used to create a stream callable of the specified class.

Each entry received by the callable is a mapping containing:

–**pid** - the process pid

–**name** - the stream name (*stderr* or *stdout*)

–**data** - the data

•**stream_backend** – the backend that will be used for the streaming process. Can be *thread* or *gevent*. When set to *gevent* you need to have *gevent* and *gevent_zmq* installed. (default: thread)

•**priority** – integer that defines a priority for the watcher. When the Arbiter do some operations on all watchers, it will sort them with this field, from the bigger number to the smallest. (default: 0)

- **singleton** – If True, this watcher has a single process. (default:False)
- **use_sockets** – If True, the processes will inherit the file descriptors, thus can reuse the sockets opened by circusd. (default: False)
- **copy_env** – If True, the environment in which circus is running run will be reproduced for the workers. (default: False)
- **copy_path** – If True, circusd *sys.path* is sent to the process through *PYTHONPATH*. You must activate **copy_env** for **copy_path** to work. (default: False)
- **max_age**: If set after around max_age seconds, the process is replaced with a new one. (default: 0, Disabled)
- **max_age_variance**: The maximum number of seconds that can be added to max_age. This extra value is to avoid restarting all processes at the same time. A process will live between max_age and max_age + max_age_variance seconds.
- **hooks**: callback functions for hooking into the watcher startup and shutdown process. **hooks** is a dict where each key is the hook name and each value is a 2-tuple with the name of the callable or the callabled itself and a boolean flag indicating if an exception occuring in the hook should not be ignored. Possible values for the hook name: *before_start*, *after_start*, *before_stop*, *after_stop*.
- **options** – extra options for the worker. All options found in the configuration file for instance, are passed in this mapping – this can be used by plugins for watcher-specific options.
- **respawn** – If set to False, the processes handled by a watcher will not be respawned automatically. (default: True)

notify_event(*topic*, *msg*)
Publish a message on the event publisher channel

reap_processes(**args*, ***kw*)
Reap all the processes for this watcher.

manage_processes(**args*, ***kw*)
Manage processes.

reap_and_manage_processes(**args*, ***kw*)
Reap & manage processes.

spawn_processes(**args*, ***kw*)
Spawn processes.

spawn_process()
Spawn process.

kill_process(*process*, *sig=15*)
Kill process.

kill_processes(**args*, ***kw*)
Kill all the processes of this watcher.

send_signal_child(**args*, ***kw*)
Send signal to a child.

stop(**args*, ***kw*)
Stop.

start(**args*, ***kw*)
Start.

restart(**args*, ***kw*)
Restart.

reload(**args*, ***kw*)

class circus.arbiter.**Arbiter**(*watchers*, *endpoint*, *pubsub_endpoint*, *check_delay=1.0*, *prereload_fn=None*, *context=None*, *loop=None*, *stats_endpoint=None*, *plugins=None*, *sockets=None*, *warmup_delay=0*, *httpd=False*, *httpd_host='localhost'*, *httpd_port=8080*, *debug=False*, *stream_backend='thread'*, *ssh_server=None*, *proc_name='circusd'*)

Class used to control a list of watchers.

Options:

- **watchers** – a list of Watcher objects
- **endpoint** – the controller ZMQ endpoint
- **pubsub_endpoint** – the pubsub endpoint
- **stats_endpoint** – the stats endpoint. If not provided, the *circusd-stats* process will not be launched.
- **check_delay** – the delay between two controller points (default: 1 s)
- **prereload_fn** – callable that will be executed on each reload (default: None)
- **context** – if provided, the zmq context to reuse. (default: None)
- **loop: if provided, a zmq.eventloop.ioloop.IOLoop instance** to reuse. (default: None)
- **plugins** – a list of plugins. Each item is a mapping with:
  - **use** – Fully qualified name that points to the plugin class
  - every other value is passed to the plugin in the **config** option
- **sockets** – a mapping of sockets. Each key is the socket name, and each value a `CircusSocket` class. (default: None)
- **warmup_delay** – a delay in seconds between two watchers startup. (default: 0)
- **httpd** – If True, a circushttpd process is run (default: False)
- **httpd_host** – the circushttpd host (default: localhost)
- **httpd_port** – the circushttpd port (default: 8080)
- **debug** – if True, adds a lot of debug info in the stdout (default: False)
- **stream_backend** – the backend that will be used for the streaming process. Can be *thread* or *gevent*. When set to *gevent* you need to have *gevent* and *gevent_zmq* installed. All watchers will use this setup unless stated otherwise in the watcher configuration. (default: thread)
- **proc_name** – the arbiter process name

start(**args*, ***kw*)

Starts all the watchers.

The start command is an infinite loop that waits for any command from a client and that watches all the processes and restarts them if needed.

reload(**args*, ***kw*)

Reloads everything.

Run the `prereload_fn()` callable if any, then gracefuly reload all watchers.

numprocesses()

Return the number of processes running across all watchers.

numwatchers()

Return the number of watchers.

get_watcher(*name*)
Return the watcher *name*.

add_watcher(*name*, *cmd*, ***kw*)
Adds a watcher.

Options:

- **name**: name of the watcher to add
- **cmd**: command to run.
- all other options defined in the Watcher constructor.

5.10 Deployment

Although the Circus daemon can be managed with the circusd command, it's easier to have it start on boot. If your system supports Upstart, you can create this Upstart script in /etc/init/circus.conf:

start on filesystem and net-device-up IFACE=lo

stop on shutdown

respawn exec /usr/local/bin/circusd –log-output /var/log/circus.log

–pidfile /var/run/circusd.pid /etc/circus.ini

This assumes that circus.ini is located at /etc/circus.ini. After rebooting, you can control circusd with the service command:

$ service circus start/stop/restart

5.10.1 Recipes

This section will contain recipes to deploy Circus. Until then you can look at Pete's Puppet recipe or at Remy's Chef recipe

5.11 The Plugin System

Circus comes with a plugin system which let you interact with **circusd**.

Note: We might add circusd-stats support to plugins later on

A Plugin is composed of two parts:

- a ZMQ subscriber to all events published by **circusd**
- a ZMQ client to send commands to **circusd**

Each plugin is run as a separate process under a custom watcher.

A few examples of some plugins you could create with this system:

- a notification system that sends e-mail alerts when a watcher is flapping
- a logger
- a tool that add or remove processes depending on the load

- etc.

Circus itself provides a few plugins:

- a statsd plugin, that sends to statsd all events emited by circusd
- the flapping feature which avoid to re-launch processes infinitely when they die too quickly.
- many more to come !

5.11.1 The CircusPlugin class

Circus provides a base class to help you implement plugins: `circus.plugins.CircusPlugin`

class `circus.plugins.`**`CircusPlugin`**(*endpoint*, *pubsub_endpoint*, *check_delay*, *ssh_server=None*, ***config*)

Base class to write plugins.

Options:

- **context** – the ZMQ context to use
- **endpoint** – the circusd ZMQ endpoint
- **pubsub_endpoint** – the circusd ZMQ pub/sub endpoint
- **check_delay** – the configured check delay
- **config** – free config mapping

`call`(*command*, ***props*)

Sends to **circusd** the command.

Options:

- **command** – the command to call
- **props** – keywords argument to add to the call

Returns the JSON mapping sent back by **circusd**

`cast`(*command*, ***props*)

Fire-and-forget a command to **circusd**

Options:

- **command** – the command to call
- **props** – keywords argument to add to the call

`handle_recv`(*data*)

Receives every event published by **circusd**

Options:

- **data** – a tuple containing the topic and the message.

`handle_stop`()

Called right before the plugin is stopped by Circus.

`handle_init`()

Called right befor a plugin is started - in the thread context.

When initialized by Circus, this class creates its own event loop that receives all **circusd** events and pass them to `handle_recv()`. The data received is a tuple containing the topic and the data itself.

`handle_recv()` **must** be implemented by the plugin.

The `call()` and `cast()` methods can be used to interact with **circusd** if you are building a Plugin that actively interacts with the daemon.

`handle_init()` and `handle_stop()` are just convenience methods you can use to initialize and clean up your code. `handle_init()` is called within the thread that just started. `handle_stop()` is called in the main thread just before the thread is stopped and joined.

5.11.2 Writing a plugin

Let's write a plugin that logs in a file every event happening in **circusd**. It takes one argument which is the filename.

The plugin could look like this:

```
from circus.plugins import CircusPlugin


class Logger(CircusPlugin):

    name = 'logger'

    def __init__(self, filename, **kwargs):
        super(Logger, self).__init__(**kwargs)
        self.filename = filename
        self.file = None

    def handle_init(self):
        self.file = open(self.filename, 'a+')

    def handle_stop(self):
        self.file.close()

    def handle_recv(self, data):
        topic, msg = data
        self.file.write('%s:%s' % (topic, msg))
```

That's it ! This class can be saved in any package/module, as long as it can be seen by Python.

For example, `Logger` could be found in a *plugins* module in a *myproject* package.

Async requests

In case you want to make any asynchronous operations (like a Tornado call or using periodicCall) make sure you are using the right loop. The loop you always want to be using it self.loop as it gets set up by the base class. The default loop often isn't the same and therefore code might not get excuted as expected.

5.11.3 Trying a plugin

You can run a plugin through the command line with the **circus-plugin** command, by specifying the plugin fully qualified name:

```
$ circus-plugin --endpoint tcp://127.0.0.1:5555 --pubsub tcp://127.0.0.1:5556 myproject.plugins.Logge
[INFO] Loading the plugin...
[INFO] Endpoint: 'tcp://127.0.0.1:5555'
[INFO] Pub/sub: 'tcp://127.0.0.1:5556'
[INFO] Starting
```

Another way to run a plugin is to let Circus handle its initialization. This is done by adding a **[plugin:NAME]** section in the configuration file, where *NAME* is a unique name for your plugin:

```
[plugin:logger]
use = myproject.plugins.Logger
filename = /var/myproject/circus.log
```

use is mandatory and points to the fully qualified name of the plugin.

When Circus starts, it creates a watcher with one process that runs the pointed class, and pass any other variable contained in the section to the plugin constructor via the **config** mapping.

You can also programmatically add plugins when you create a `circus.arbiter.Arbiter` class or use `circus.get_arbiter()`, see *Circus Library*.

5.11.4 Performances

Since every plugin is loaded in its own process, it should not impact the overall performances of the system as long as the work done by the plugin is not doing too many calls to the **circusd** process.

5.12 Security

Circus is built on the top of the ZeroMQ library and comes with no security at all in its protocols. However, you can run a Circus system on a server and set up an SSH tunnel to access it from another machine.

This section explains what Circus does on your system when you run it, and ends up describing how to use an SSH tunnel.

You can also read http://www.zeromq.org/area:faq#toc5

5.12.1 TCP ports

By default, Circus opens the following TCP ports on the local host:

- **5555** – the port used to control circus via **circusctl**
- **5556** – the port used for the Publisher/Subscriber channel.
- **5557** – the port used for the statitics channel – if activated.
- **8080** – the port used by the Web UI – if activated.

These ports allow client apps to interact with your Circus system, and depending on how your infrastructure is organized, you may want to protect these ports via firewalls **or** configure Circus to run using **IPC** ports.

Here's an example of running Circus using only IPC entry points:

```
[circus]
check_delay = 5
endpoint = ipc:///var/circus/endpoint
pubsub_endpoint = ipc:///var/circus/pubsub
stats_endpoint = ipc:///var/circus/stats
```

When Configured using IPC, the commands must be run from the same box, but no one can access them from outside, unlike using TCP.

Of course, if you activate the Web UI, the **8080** port will still be open.

5.12.2 circushttpd

When you run **circushttpd** manually, or when you use the **httpd** option in the ini file like this:

```
[circus]
check_delay = 5
endpoint = ipc:///var/circus/endpoint
pubsub_endpoint = ipc:///var/circus/pubsub
stats_endpoint = ipc:///var/circus/stats
httpd = 1
```

The web application will run on port *8080* and will let anyone accessing the web page manage the **circusd** daemon.

That includes creating new watchers that can run any command on your system !

Do not make it publicly available

If you want to protect the access to the web panel, you can serve it behind Nginx or Apache or any proxy-capable web server, than can take care of the security.

5.12.3 User and Group Permissions

By default, all processes started with Circus will be running with the same user and group than **circusd**. Depending on the privileges the user has on the system, you may not have access to all the features Circus provides.

For instance, some statistics features on a running processes require extended privileges. Typically, if the CPU usage numbers you get using the **stats** command are *N/A*, it means your user can't access the proc files. This will be the case by default under Mac OS X.

You may run **circusd** as root to fix this, and set the **uid** and **gid** values for each watcher to get all the features.

But beware that running **circusd** as root exposes you to potential privilege escalation bugs. While we're doing our best to avoid any bugs, running as root and facing a bug that performs unwanted actions on your system may dangerous.

The best way to prevent this is to make sure that the system running Circus is completely isolated (like a VM) **or** to run the whole system under a controlled user.

5.12.4 SSH tunneling

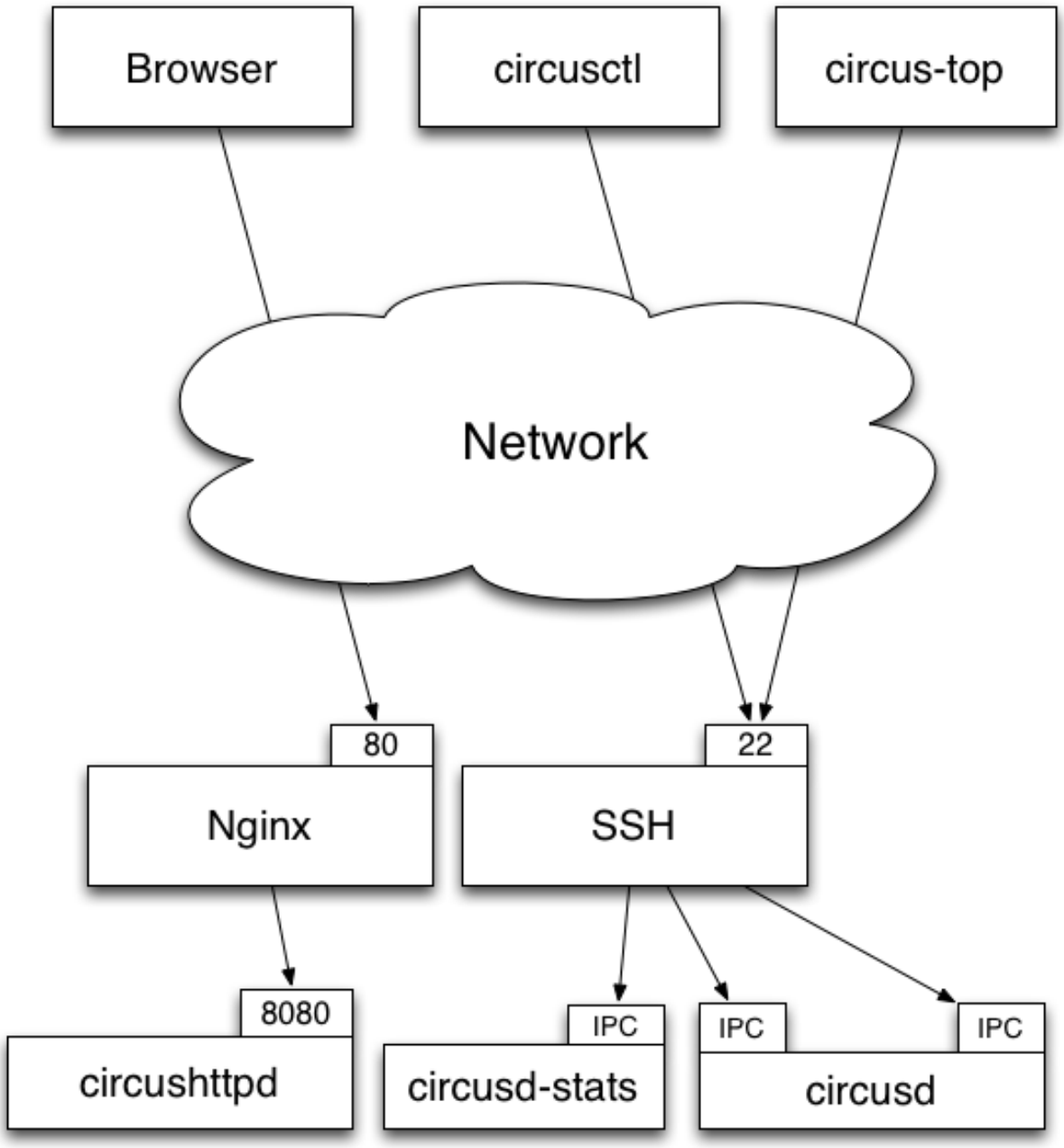
Clients can connect to a **circusd** instance by creating an SSH tunnel. To do so, pass the command line option **–ssh** followed by **user@address**, where **user** is the user on the remote server and **address** is the server's address as seen by the client. The SSH protocol will require credentials to complete the login.

If **circusd** as seen by the SSH server is not at the default endpoint address **localhost:5555** then specify the **circusd** address using the option **–endpoint**

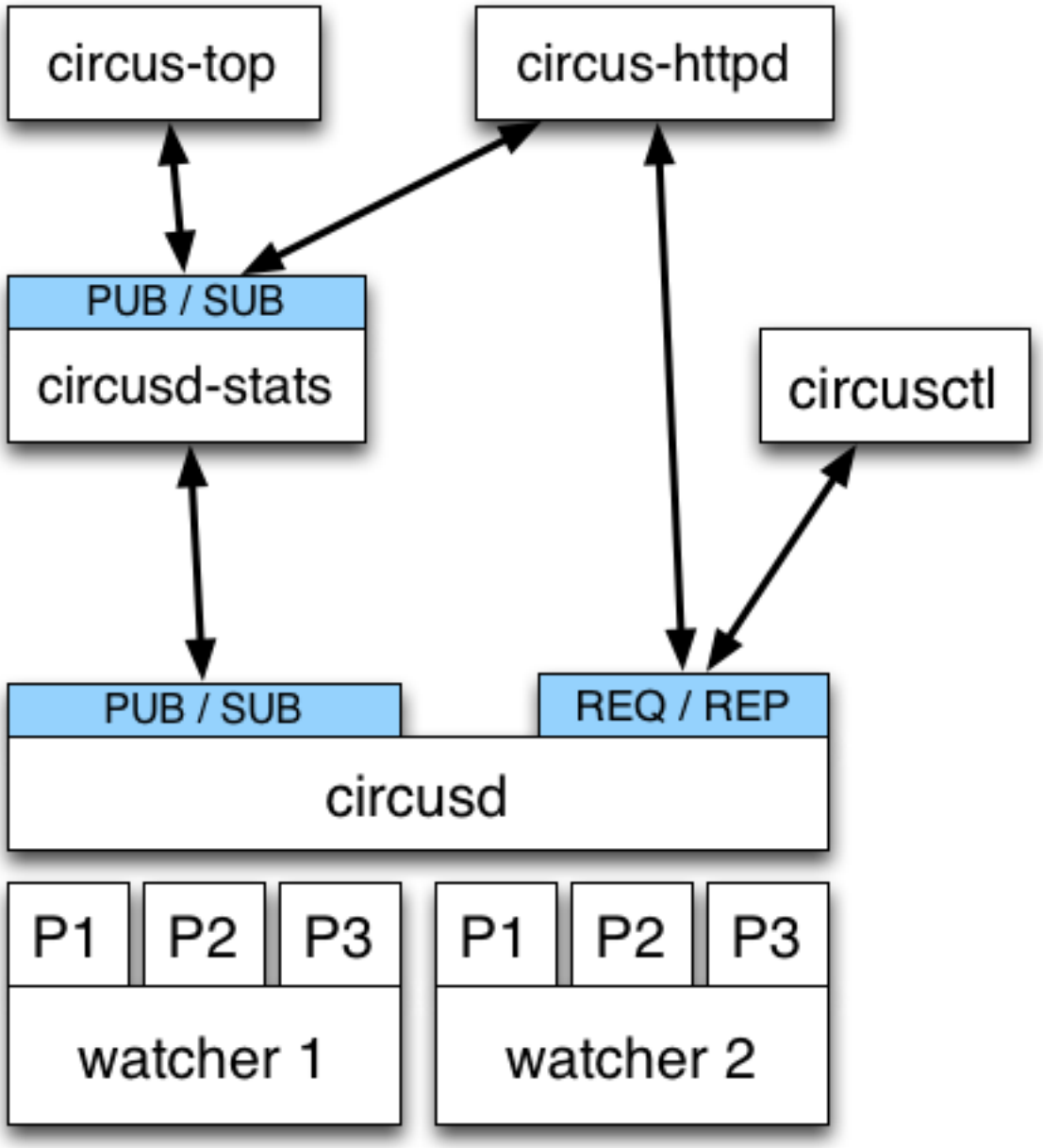
5.12.5 Secured setup example

Setting up a secured Circus server can be done by:

- Running an SSH Server
- Running Apache or Nginx on the *80* port, and doing a reverse-proxy on the *8080* port.
- Blocking the *8080* port from outside access.
- Running all ZMQ Circusd ports using IPC files instead of TCP ports, and tunneling all calls via SSH.


Browser
circusctl
circus-top
Network
80
Nginx
22
SSH
8080
circushttpd
IPC
circusd-stats
IPC
IPC
circusd

5.13 Design



Circus is composed of a main process called **circusd** which takes care of running all the processes. Each process managed by Circus is a child process of **circusd**.

Processes are organized in groups called **watchers**. A **watcher** is basically a command **circusd** runs on your system, and for each command you can configure how many processes you want to run.

The concept of *watcher* is useful when you want to manage all the processes running the same command – like restart them, etc.

circusd binds two ZeroMQ sockets:

- **REQ/REP** – a socket used to control **circusd** using json-based *commands*.

- **PUB/SUB** – a socket where **circusd** publishes events, like when a process is started or stopped.

Note: Although its name, ZeroMQ is not a queue management system. Think of it as an inter-process communication (IPC) library.

Another process called **circusd-stats** is run by **circusd** when the option is activated. **circusd-stats**'s job is to publish CPU/Memory usage statistics in a dedicated **PUB/SUB** channel.

This specialized channel is used by **circus-top** and **circus-httpd** to display a live stream of the activity.

circus-top is a console script that mimics **top** to display all the CPU and Memory usage of the processes managed by Circus.

circus-httpd is the web managment interface that will let you interact with Circus. It displays a live stream using web sockets and the **circusd-stats** channel, but also let you interact with **circusd** via its **REQ/REP** channel.

Last but not least, **circusctl** is a command-line tool that let you drive **circusd** via its **REQ/REP** channel.

You can also have plugins that subscribe to **circusd**'s **PUB/SUB** channel and let you send commands to the **REQ/REP** channel like **circusctl** would.

5.14 Why should I use Circus instead of X ?

1. **Circus simplifies your web stack process management**

   Circus knows how to manage processes *and* sockets, so you don't have to delegate web workers managment to a WGSI server.

   See *Circus stack v.s. Classical stack*

2. **Circus provides pub/sub and poll notifications via ZeroMQ**

   Circus has a *pub/sub* channel you can subscribe to. This channel receives all events happening in Circus. For example, you can be notified when a process is *flapping*, or build a client that triggers a warning when some processes are eating all the CPU or RAM.

   These events are sent via a ZeroMQ channel, which makes it different from the stdin stream Supervisord uses:

   - Circus sends events in a fire-and-forget fashion, so there's no need to manually loop through *all* listeners and maintain their states.
   - Subscribers can be located on a remote host.

   Circus also provides ways to get status updates via one-time polls on a req/rep channel. This means you can get your information without having to subscribe to a stream. The *Command-line tools* command provided by Circus uses this channel.

   See *Examples*.

3. **Circus is (Python) developer friendly**

   While Circus can be driven entirely by a config file and the *circusctl* / *circusd* commands, it is easy to reuse all or part of the system to build your own custom process watcher in Python.

   Every layer of the system is isolated, so you can reuse independently:

   - the process wrapper (`Process`)
   - the processes manager (`Watcher`)
   - the global manager that runs several processes managers (`Arbiter`)

- and so on...

4. **Circus scales**

   One of the use cases of Circus is to manage thousands of processes without adding overhead – we're dedicated to focus on this.

5.15 Examples

The **examples** directory in the Circus repository contains a few examples to get you started.

Open a shell and cd into it:

```
$ cd examples
```

Now try to run the `example1.ini` config:

```
$ circusd example1.ini
2012-03-19 13:29:48 [7843] [INFO] Starting master on pid 7843
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7844]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7845]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7846]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7847]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7848]
2012-03-19 13:29:48 [7843] [INFO] running dummy2 process [pid 7849]
2012-03-19 13:29:48 [7843] [INFO] running dummy2 process [pid 7850]
2012-03-19 13:29:48 [7843] [INFO] running dummy2 process [pid 7851]
```

Congrats, you have 8 workers running !

Now run in a separate shell the listener script:

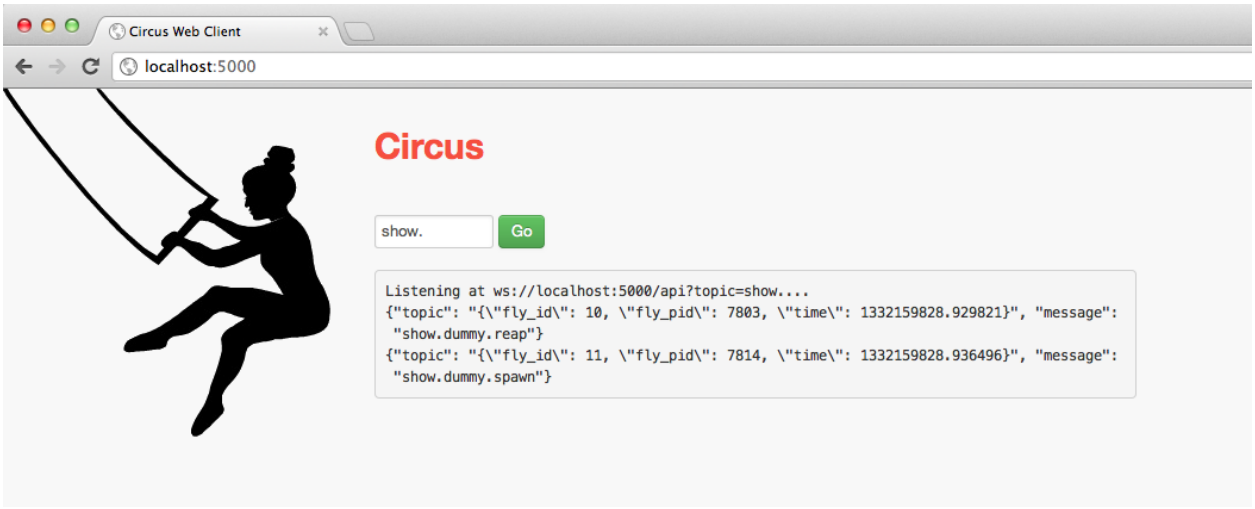
```
$ python listener.py
```

This script will print out all events happening in Circus. Try for instance to kill a worker:

```
$ kill 7849
```

You should see a few lines popping into the listener shell.

If you are brave enough, you can try the web socket demo with a web socket compatible browser. It will stream every event into the web page.



5.16 Circus Use Cases

This chapter presents a few use cases, to give you an idea on how to use Circus in your environment.

5.16.1 Running a WSGI application

Running a WSGI application with Circus is quite interesting because you can watch & manage your *web workers* using *circus-top*, *circusctl* or the Web interface.

This is made possible by using Circus sockets. See *Circus stack v.s. Classical stack*.

Let's take an example with a minimal Pyramid application:

```
from pyramid.config import Configurator
from pyramid.response import Response

def hello_world(request):
    return Response('Hello %(name)s!' % request.matchdict)

config = Configurator()
config.add_route('hello', '/hello/{name}')
config.add_view(hello_world, route_name='hello')
application = config.make_wsgi_app()
```

Save this script into an **app.py** file, then install those projects:

```
$ pip install Pyramid
$ pip install chaussette
```

Next, make sure you can run your Pyramid application using the **chaussette** console script:

```
$ chaussette app.application
Application is <pyramid.router.Router object at 0x10a4d4bd0>
Serving on localhost:8080
Using <class 'chaussette.backend._waitress.Server'> as a backend
```

And check that you can reach it by visiting **http://localhost:8080/hello/tarek**

Now that your application is up and running, let's create a Circus configuration file:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:webworker]
cmd = chaussette --fd $(circus.sockets.webapp) app.application
use_sockets = True
numprocesses = 3

[socket:webapp]
host = 127.0.0.1
port = 8080
```

This file tells Circus to bind a socket on port *8080* and run *chaussette* workers on that socket – by passing its fd.

Save it to *server.ini* and try to run it using **circusd**

```
$ circusd server.ini
[INFO] Starting master on pid 8971
[INFO] sockets started
[INFO] circusd-stats started
[INFO] webapp started
[INFO] Arbiter now waiting for commands
```

Make sure you still get the app on **http://localhost:8080/hello/tarek**.

Congrats ! you have a WSGI application running 3 workers.

You can run the *The Web Console* or the *Command-line tools*, and enjoy Circus management.

5.16.2 Running a Django application

Running a Django application is done exactly like running a WSGI application. Use the *PYTHONPATH* to import the directory the project is in, the directory that contains the directory that has settings.py in it (with Django 1.4+ this directory has manage.py in it)

> [socket:dwebapp] host = 127.0.0.1 port = 8080
>
> [watcher:dwebworker] cmd = chaussette –fd $(circus.sockets.dwebapp) dproject.wsgi.application use_sockets = True numprocesses = 2
>
> [env:dwebworker] PYTHONPATH = /path/to/parent-of-dproject

If you need to pass the *DJANGO_SETTINGS_MODULE* for a backend worker for example, you can pass that also though the *env* configation option:

> [watcher:dbackend] cmd = /path/to/script.py numprocesses=3
>
> [env:dbackend] PYTHONPATH = /path/to/parent-of-dproject DJANGO_SETTINGS_MODULE=dproject.settings

See http://chaussette.readthedocs.org for more about chaussette.

5.17 Code coverage

```
Name                              Stmts   Miss  Cover   Missing
---------------------------------------------------------------
/Users/tarek/Dev/github.com/circus/bin/bottle                      1613    875    46%   25-36, 95-96,
circus/__init__                     31     15    52%   1-14, 100-106, 112
circus/_patch                       76     57    25%   1-8, 17, 19, 23-47, 55-69, 73, 77-79, 85-94, 96-
circus/arbiter                     212     40    81%   99, 112-116, 124-132, 154-162, 178, 198-199, 228
circus/circusctl                   216    169    22%   17-18, 34-45, 53-69, 72-74, 81-90, 96, 99-115, 1
circus/client                       56      6    89%   17, 21, 54-55, 60, 76
circus/commands/addwatcher          24     14    42%   1-66, 73, 78
circus/commands/base                73     53    27%   1-13, 21, 28, 38-74, 77-82, 85, 92-100, 106-108
circus/commands/dstats              24     23     4%   1-63, 66-81
circus/commands/incrproc            21     16    24%   1-53, 61-68
circus/commands/list                28     13    54%   1-54, 66
circus/commands/listsockets         14     11    21%   1-36, 44-50
circus/commands/numprocesses        19     17    11%   1-57, 59-60, 67-70
circus/commands/numwatchers         14     13     7%   1-42, 45-48
circus/commands/options             20     18    10%   1-101, 105-111
circus/commands/quit                 7      6    14%   1-36
circus/commands/reload              17     15    12%   1-68, 70-71
circus/commands/restart             15     13    13%   1-56, 58-59
circus/commands/rmwatcher           12     10    17%   1-54
```

```
circus/commands/sendsignal        52     34    35%   1-137, 148, 150, 157-162, 167, 169, 175-182
circus/commands/set               34     22    35%   1-61, 72, 77
circus/commands/start             15     12    20%   1-53, 58
circus/commands/stats             51     46    10%   1-89, 91-102, 109-138
circus/commands/status            23     20    13%   1-65, 70-80
circus/commands/stop              12      8    33%   1-50
circus/commands/util              62     53    15%   1-43, 49, 54, 59, 62-63, 66-67, 70-75, 78-81
circus/config                    180     49    73%   45, 57-60, 87-88, 105-108, 127-150, 169, 185, 18
circus/consumer                   43     17    60%   30, 34-46, 50, 53-57
circus/controller                116     14    88%   75, 85-86, 95-97, 107, 123, 127, 148, 151, 157,
circus/plugins/__init__          146    105    28%   36-46, 50-58, 62-84, 88-96, 108-111, 121-122, 13
circus/process                   146     37    75%   3-9, 105-125, 160-165, 183, 225-226, 232, 244, 2
circus/py3compat                  47     44     6%   1-38, 43-67
circus/sighandler                 39     17    56%   29, 38-48, 51, 54, 57, 60, 63
circus/sockets                    76     10    87%   27, 62, 68, 74-76, 109-110, 120, 131
circus/stats/__init__             41     28    32%   34-85, 89
circus/stats/client              168    130    23%   30-32, 35-36, 40-45, 58-167, 172-178, 181, 184-1
circus/stats/collector           116     48    59%   8, 32, 37, 42-70, 73-97, 125-126, 132, 146-147,
circus/stats/publisher            27     20    26%   9-14, 17-30, 33-35
circus/stats/streamer            150    124    17%   21-40, 43, 46, 49-53, 57-67, 70-81, 84-114, 117-
circus/stream/__init__            42     13    69%   17, 22, 25-26, 29, 38, 41-47, 79
circus/stream/base                64     13    80%   22, 39, 58-59, 65, 69-77
circus/stream/file_stream         46     24    48%   41, 46, 52-69, 79, 81-83
circus/stream/sthread             22      2    91%   29-30
circus/util                      372    157    58%   1-48, 55-60, 63-66, 70-88, 94-96, 102, 116, 123,
circus/watcher                   408     99    76%   194, 224, 237, 246, 272, 296, 319-320, 326, 333,
circus/web/__init__                0      0   100%
circus/web/circushttpd            79     39    51%   9-12, 38, 48, 56, 66, 74, 80, 89, 94, 112, 117,
circus/web/controller            113     61    46%   9-11, 32, 48-49, 55, 58, 62-63, 67-70, 73-74, 80
circus/web/namespace              41     32    22%   11-12, 39-88, 102-104, 110
circus/web/server                 22     17    23%   7-10, 13-33
circus/web/session                32     11    66%   19, 24-34
circus/web/util                   52     22    58%   12-14, 18, 24-44, 87
base_html   NoSource: No source for code: '/Users/tarek/Dev/github.com/circus/docs/base_html': [Errno
connect_html   NoSource: No source for code: '/Users/tarek/Dev/github.com/circus/docs/connect_html':
index_html   NoSource: No source for code: '/Users/tarek/Dev/github.com/circus/docs/index_html': [Er
----------------------------------------------------------------
TOTAL                           5329   2712    49%
```

5.18 Glossary

arbiter The *arbiter* is responsible for managing all the watchers within circus, ensuring all processes run correctly.

controller A *controller* contains the set of actions that can be performed on the arbiter.

flapping The *flapping detection* subscribes to events and detects when some processes are constantly restarting.

process, worker, workers, processes A *process* is an independent OS process instance of your program. A single watcher can run one or more processes. We also call them workers.

pub/sub Circus has a *pubsub* that receives events from the watchers and dispatches them to all subscribers.

remote controller The *remote controller* allows you to communicate with the controller via ZMQ to control Circus.

watchers, watcher A *watcher* is the program you tell Circus to run. A single Circus instance can run one or more watchers.

5.19 Contributing to Circus

Circus has been started at Mozilla but its goal is not to stay only there. We're trying to build a tool that's useful for others, and easily extensible.

We really are open to any contributions, in the form of code, documentation, discussions, feature proposal etc.

You can start a topic in our mailing list : http://tech.groups.yahoo.com/group/circus-dev/

Or add an issue in our bug tracker

5.19.1 Fixing typos and enhancing the documentation

It's totally possible that your eyes are bleeding while reading this half-english half-french documentation, don't hesitate to contribute any rephrasing / enhancement on the form in the documentation. You probably don't even need to understand how Circus works under the hood to do that.

5.19.2 Adding new features

New features are of course very much appreciated. If you have the need and the time to work on new features, adding them to Circus shouldn't be that complicated. We tried very hard to have a clean and understandable API, hope it serves the purpose.

You will need to add documentation and tests alongside with the code of the new feature. Otherwise we'll not be able to accept the patch.

5.19.3 How to submit your changes

We're using git as a DVCS. The best way to propose changes is to create a branch on your side (via *git checkout -b branchname*) and commit your changes there. Once you have something ready for prime-time, issue a pull request against this branch.

We are following this model to allow to have low coupling between the features you are proposing. For instance, we can accept one pull request while still being in discussion for another one.

Before proposing your changes, double check that they are not breaking anything! You can use the *tox* command to ensure this, it will run the testsuite under the different supported python versions.

Please use : http://issue2pr.herokuapp.com/ to reference a commit to an existing circus issue, if any.

5.19.4 Avoiding merge commits

Avoiding merge commits allows to have a clean and readable history. To do so, instead of doing "git pull" and letting git handling the merges for you, using git pull –rebase will put your changes after the changes that are commited in the branch, or when working on master.

That is, for us core developers, it's not possible anymore to use the handy github green button on pull requests if developers didn't rebased their work themselves or if we wait too much time between the request and the actual merge. Instead, the flow looks like this:

```
git remote add name repo-url
git fetch name
git checkout feature-branch
git rebase master
```

```
# check that everything is working properly and then merge on master
git checkout master
git merge feature-branch
```

5.19.5 Discussing

If you find yourself in need of any help while looking at the code of Circus, you can go and find us on irc at #mozilla-circus on irc.freenode.org (or if you don't have any IRC client, use the webchat)

You can also start a thread in our mailing list - http://tech.groups.yahoo.com/group/circus-dev

5.20 How to add new commands in circus

If you want to add a new command, we tried to make this as simple as possible. You need to do three main things:

1. create a "your_command.py" file under *circus/commands/*.
2. Implement a single class in there, with predefined methods
3. Add the new command in *circus/commands/__init__.py*.

Let's say we want to add a command which returns the number of watchers actually in use, we would do something like this (extensively commented to allow you to follow more easily):

```
class NumWatchers(Command):
    """It is a good practice to describe what the class does here.

    Have a look at other commands to see how we are used to format this
    text. It will be used to automatically appear in the documentation of
    circus, so don't be affraid of being exhaustive, that's what it is made
    for.
    """
    # all the commands need to inherit from `circus.commands.base.Command`

    name = "numwatchers"
    # you need to specify a name so we find back the command somehow

    options = [('', 'optname', default_value, 'description')]
    # XXX describe options

    properties = ['foo', 'bar']
    # properties list the command argments that are mendatory. If they are
    # not provided, then an error will be thrown

    def execute(self, arbiter, props):
        # the execute method is the core of the command: put here all the
        # logic of the command and return a dict containing the values you
        # want to return, if any
        return {"numwatchers": arbiter.numwatchers()}

    def console_msg(self, msg):
        # msg is what is returned by the execute method.
        # this method is used to format the response for a console (it is
        # used for instance by circusctl to print its messages)
        return "a string that will be displayed"
```

```
    def validate(self, props):
        # this method is used to validate that the arguments passed to the
        # command are correct. An ArgumentError should be thrown in case
        # there is an error in the passed arguments (for instance if they
        # do not match together.
        # In case there is a problem wrt their content, a MessageError
        # should be thrown. This method can modify the content of the props
        # dict, it will be passed to execute afterwards.
```

5.21 Copyright

Circus was initiated by Tarek Ziade and is licenced under APLv2

Benoit Chesneau was an early contributor and did many things, like most of the circus.commands work.

5.21.1 Licence


```
Copyright 2012 - Mozilla Foundation
Copyright 2012 - Benoit Chesneau

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```


5.21.2 Contributors

See https://github.com/mozilla-services/circus/graphs/contributors

CHAPTER

SIX

CONTRIBUTIONS AND FEEDBACK

More on contribution: *Contributing to Circus*.

Useful Links:

- There's a maling list for any feedback or question: http://tech.groups.yahoo.com/group/circus-dev/
- The repository and issue tracker is at GitHub : https://github.com/mozilla-services/circus
- Join us on the IRC : Freenode, channel **#mozilla-circus**