Circus Documentation

Release 0.11.1

Mozilla Foundation

May 26, 2014

Contents

Circus is a Python program which can be used to monitor and control processes and sockets.

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

To install it and try its features check out the *Step-by-step tutorial*, or read the rest of this page for a quick introduction.

CHAPTER 1

Running a Circus Daemon

Circus provides a command-line script call **circusd** that can be used to manage *processes* organized in one or more *watchers*.

Circus' command-line tool is configurable using an ini-style configuration file.

Here's a very minimal example:

```
[watcher:program]
cmd = python myprogram.py
numprocesses = 5

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

The file is then passed to *circusd*:

```
$ circusd example.ini
```

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 *Working with sockets*.

To understand why it's a killer feature, read *How does Circus stack compare to a classical stack?*.

CHAPTER 2

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 *CLI tools*

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

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

2.1 What now ?

If you are a developer and want to leverage Circus in your own project, write plugins or hooks, go to *Circus for developers*.

If you are an ops and want to manage your processes using Circus, go to *Circus for Ops*.

2.2 Contributions and Feedback

More on contributing: *Contributing to Circus*.

Useful Links:

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

2.3 Documentation index

2.3.1 Installing Circus

Circus is a Python package which is published on PyPI - the Python Package Index.

The simplest way to install it is to use pip, a tool for installing and managing Python packages:

```
$ 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 *Step-by-step tutorial*.

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

zc.buildout

We provide a zc.buildout configuration, you can use it by simply running the bootstrap script, then calling buildout:

```
$ python bootstrap.py
$ bin/buildout
```

More on Requirements

Circus works with:

- Python 2.6, 2.7, 3.2 or 3.3
- **zeromq >= 2.1.10**
  - The version of zeromq supported is ultimately determined by what version of pyzmq is installed by pip during circus installation.
  - Their current release supports 2.x (limited), 3.x, and 4.x ZeroMQ versions.
  - **Note**: If you are using PyPy instead of CPython, make sure to read their installation docs as ZeroMQ version support is not the same on PyPy.

When you install circus, the latest versions of the Python dependencies will be pulled out for you.

You can also install them manually using the pip-requirements.txt file we provide:

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

If you want to run the Web console you will need to install **circus-web**:

```
$ pip install circus-web
```

2.3.2 Tutorial

Step-by-step tutorial

The examples directory in the Circus repository contains many examples to get you started, but here's a full tutorial that gives you an overview of the features.

We're going to supervise a WSGI application.

Installation

Circus is tested on Mac OS X and Linux with the latest Python 2.6, 2.7, 3.2 and 3.3. To run a full Circus, you will also need **libzmq**, **libevent** & **virtualenv**.

On Debian-based systems:

```
$ sudo apt-get install libzmq-dev libevent-dev python-dev python-virtualenv
```

Create a virtualenv and install *circus*, *circus-web* and *chaussette* in it

```
$ virtualenv /tmp/circus
$ cd /tmp/circus
$ bin/pip install circus
$ bin/pip install circus-web
$ bin/pip install chaussette
```

Once this is done, you'll find a plethora of commands in the local bin dir.

Usage

Chaussette comes with a default Hello world app, try to run it:

```
$ bin/chaussette
```

You should be able to visit http://localhost:8080 and see *hello world*.

Stop Chaussette and add a circus.ini file in the directory containing:

```
[circus]
statsd = 1
httpd = 1

[watcher:webapp]
cmd = bin/chaussette --fd $(circus.sockets.web)
numprocesses = 3
use_sockets = True

[socket:web]
host = 127.0.0.1
port = 9999
```

This config file tells Circus to bind a socket on port *9999* and run 3 chaussettes workers against it. It also activates the Circus web dashboard and the statistics module.

Save it & run it using **circusd**:

```
$ bin/circusd --daemon circus.ini
```

Now visit http://127.0.0.1:9999, you should see the hello world app. The difference now is that the socket is managed by Circus and there are several web workers that are accepting connections against it.

Note: The load balancing is operated by the operating system so you're getting the same speed as any other pre-fork web server like Apache or NGinx. Circus does not interfer with the data that goes through.

You can also visit http://localhost:8080/ and enjoy the Circus web dashboard.

Interaction

Let's use the circusctl shell while the system is running:

```
$ bin/circusctl
circusctl 0.7.1
circusd-stats: active
circushttpd: active
webapp: active
(circusctl)
```

You get into an interactive shell. Type **help** to get all commands:

```
(circusctl) help

Documented commands (type help <topic>):
========================================
add     get            list         numprocesses  quit     rm      start   stop
decr    globaloptions  listen       numwatchers   reload   set     stats
dstats  incr           listsockets  options       restart  signal  status

Undocumented commands:
======================
EOF  help
```

Let's try basic things. Let's list the web workers processes and add a new one:

```
(circusctl) list webapp
13712,13713,13714
(circusctl) incr webapp
4
(circusctl) list webapp
13712,13713,13714,13973
```

Congrats, you've interacted with your Circus! Get off the shell with Ctrl+D and now run circus-top:

```
$ bin/circus-top
```

This is a top-like command to watch all your processes' memory and CPU usage in real time.

Hit Ctrl+C and now let's quit Circus completely via circus-ctl:

```
$ bin/circusctl quit
ok
```

Next steps

You can plug your own WSGI application instead of Chaussette's hello world simply by pointing the application callable.

Chaussette also comes with many backends like Gevent or Meinheld.

Read https://chaussette.readthedocs.org/ for all options.

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 *How does Circus stack compare to a 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 *CLI tools* command provided by Circus uses this channel.

   See *Step-by-step tutorial*.

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.

Coming from Supervisor

Supervisor is a very popular solution in the Python world and we're often asked how Circus compares with it.

If you are coming from Supervisor, this page tries to give an overview of how the tools differ.

Differences overview Supervisor & Circus have the same goals - they both manage processes and provide a command-line script — respectively **supervisord** and **circusd** — that reads a configuration file, forks new processes and maintain them alive.

Circus has an extra feature: the ability to bind sockets and let the processes it manages use them. This "pre-fork" model is used by many web servers out there, like Apache or Unicorn. Having this option in Circus can simplify a web app stack: all processes and sockets are managed by a single tool.

Both projects provide a way to control a running daemon via another script. respectively **supervisorctl** and **circusctl**. They also both have events and a way to subscribe to them. The main difference is the underlying technology: Supervisor uses XML-RPC for interacting with the daemon, while Circus uses ZeroMQ.

Circus & Supervisor both have a web interface to display what's going on. Circus' one is more advanced because you can follow in real time what's going on and interact with the daemon. It uses web sockets and is developed in a separate project (circus-web.)

There are many other subtle differences in the core design, we might list here one day... In the meantime, you can learn more about circus internals in *Overall architecture*.

Configuration Both systems use an ini-like file as a configuration.

- Supervisor documentation
- Circus documentation

Here's a small example of running an application with Supervisor. In this case, the application will be started and restarted in case it crashes

```
[program:example]
command=npm start
directory=/home/www/my-server/
user=www-data
autostart=true
autorestart=true
redirect_stderr=True
```

In Circus, the same configuration is done by:

```
[watcher:example]
cmd=npm start
working_dir=/home/www/my-server/
user=www-data
stderr_stream.class=StdoutStream
```

Notice that the stderr redirection is slightly different in Circus. The tool does not have a **tail** feature like in Supervisor, but will let you hook any piece of code to deal with the incoming stream. You can create your own stream hook (as a Class) and do whatever you want with the incoming stream. Circus provides some built-in stream classes like **StdoutStream**, **FileStream**, **WatchedFileStream**, or **TimedRotatingFileStream**.

2.3.3 Circus for Ops

> **Warning:** By default, Circus doesn't secure its messages when sending information through ZeroMQ. Before running Circus in a production environment, make sure to read the *Security* page.

The first step to manage a Circus daemon is to write its configuration file. See *Configuration*. If you are deploying a web stack, have a look at *Working with sockets*.

Circus can be deployed using Python 2.6, 2.7, 3.2 or 3.3 - most deployments out there are done in 2.7. To learn how to deploy Circus, check out *Deployment*.

To manage a Circus daemon, you should get familiar with the list of *Commands* you can use in **circusctl**. Notice that you can have the same help online when you run **circusctl** as a shell.

We also provide **circus-top**, see *CLI tools* and a nice web dashboard. see *The Web Console*.

Last, to get the most out of Circus, make sure to check out how to use plugins and hooks. See *Using built-in plugins* and *Hooks*.

Ops documentation index

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 = \*.more.config.ini
umask = 002

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

# 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

# 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

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

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

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

circus - single section

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

endpoint_owner If set to a system username and the endpoint is an ipc socket like *ipc://var/run/circusd.sock*, then ownership of the socket file will be changed to that user at startup. For more details, see *Security*. (default: None)

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

statsd If set to True, Circus runs the circusd-stats daemon. (default: False)

stats_endpoint The ZMQ PUB/SUB socket receiving publications of stats. (default: *tcp://127.0.0.1:5557*)

statsd_close_outputs If True sends the circusd-stats stdout/stderr to /dev/null. (default: False)

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

include List of config files to include. You can use wildcards (*) to include particular schemes for your files. The paths are absolute or relative to the config file. (default: None)

include_dir List of config directories. All files matching **.ini* under each directory will be included. The paths are absolute or relative to the config file. (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)

httpd_close_outputs If True, sends the circushttpd stdout/stderr to /dev/null. (default: False)

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

debug_gc If set to True, circusd outputs additional log info from the garbage collector. This can be useful in tracking down memory leaks. (default: False)

pidfile The file that must be used to keep the daemon pid.

umask Value for umask. If not set, circusd will not attempt to modify umask.

loglevel The loglevel that we want to see (default: INFO)

logoutput The logoutput file where we want to log (default: – to log on stdout). You can log to a remote syslog by using the following syntax: `syslog://host:port?facility` where host is your syslog server, port is optional and facility is the syslog facility to use. If you wish to log to a local syslog you can use `syslog:///path/to/syslog/socket?facility` instead.

loggerconfig A path to an INI, JSON or YAML file to configure standard Python logging for the Arbiter. The special value "default" uses the builtin logging configuration based on the optional loglevel and logoutput options.

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 *Formatting the commands and arguments with dynamic variables* for more information on this.

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

shell_args Command-line arguments to pass to the shell command when **shell** is True. Works only for *nix system (default: None)

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.

autostart If set to false, the watcher will not be started automatically when the arbiter starts. The watcher can be started explicitly (example: *circusctrl start myprogram*). (Default: True)

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 some stream classes you can use without prefix:

- `FileStream`: writes in a file and can do automatic log rotation
- `WatchedFileStream`: writes in a file and relies on external log rotation
- `TimedRotatingFileStream`: writes in a file and can do rotate at certain timed intervals.
- `QueueStream`: write in a memory Queue
- `StdoutStream`: writes in the stdout
- `FancyStdoutStream`: writes colored output with time prefixes 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 some stream classes you can use without prefix:

- `FileStream`: writes in a file and can do automatic log rotation
- `WatchedFileStream`: writes in a file and relies on external log rotation
- `TimedRotatingFileStream`: writes in a file and can do rotate at certain timed intervals.
- `QueueStream`: write in a memory Queue
- `StdoutStream`: writes in the stdout
- `FancyStdoutStream`: writes colored output with time prefixes 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**.

close_child_stdout If set to True, the sdout stream of each process will be sent to /dev/null after the fork. Defaults to False.

close_child_stderr If set to True, the sderr stream of each process will be sent to /dev/null after the fork. Defaults to False.

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

stop_signal The signal to send when stopping the process. Can be specified as a number or a signal name. Signal names are case-insensitive and can include 'SIG' or not. So valid examples include *quit*, *INT*, *SIGTERM* and *3*. Defaults to SIGTERM.

stop_children When sending the *stop_signal*, send it to the children as well. 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. Set to -1 to disable max_retry and retry indefinitely.

graceful_timeout The number of seconds to wait for a process to terminate gracefully before killing it.

When stopping a process, we first send it a *stop_signal*. A worker may catch this signal to perform clean up operations before exiting. If the worker is still active after graceful_timeout seconds, we send it a SIGKILL signal. It is not possible to catch SIGKILL signals so the worker will stop.

Defaults to 30s.

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.

on_demand If set to True, the processes will be started only after the first connection to one of the configured sockets (see below). If a restart is needed, it will be only triggered at the next socket event.

hooks.* Available hooks: **before_start**, **after_start**, **before_spawn**, **after_spawn**, **before_stop**, **after_stop**, **before_signal**, **after_signal**, **extended_stats**

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**, **before_spawn**, **after_start** or **after_spawn**, the startup will be aborted.

If the hook is **before_signal** and returns **False**, then the corresponding signal will not be sent (except SIGKILL which is always sent)

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*.

virtualenv When provided, points to the root of a Virtualenv directory. The watcher will scan the local **site-packages** and loads its content into the execution environment. Must be used with **copy_env** set to True. Defaults to None.

respawn If set to False, the processes handled by a watcher will not be respawned automatically. The processes can be manually respawned with the *start* command. (default: True)

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'.

interface When provided a network interface name like 'eth0', binds the socket to that particular device so that only packets received from that particular interface are processed by the socket. This can be used for example to limit which device to bind when binding on IN_ADDR_ANY (0.0.0.0) or IN_ADDR_BROADCAST (255.255.255.255). Note that this only works for some socket types, particularly AF_INET sockets.

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.

umask When provided, sets the umask that will be used to create an AF_UNIX socket. For example, *umask=000* will produce a socket with permission *777*.

replace When creating Unix sockets ('AF_UNIX'), an existing file may indicate a problem so the default is to fail. Specify *True* to simply remove the old file if you are sure that the socket is managed only by Circus.

so_reuseport If set to True and SO_REUSEPORT is available on target platform, circus will create and bind new SO_REUSEPORT socket(s) for every worker it starts which is a user of this socket(s).

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
```

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.

You can use all the watcher options, since a plugin is started like a watcher.

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

env or 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'

Section responsible for delivering environment variable to run processes.

Example:

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

[watcher:worker2]
cmd = ping 127.0.0.1

[env]
CAKE = lie
```

The variable *CAKE* will propagated to all watchers defined in config file.

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

It's possible to use wildcards as well.

Example:

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

[watcher:worker2]
cmd = ping 127.0.0.1

[env:worker*]
PATH = /bin
```

Both *worker1* and *worker2* will be run with PATH = /bin

Using environment variables When writing your configuration file, you can use environment variables defined in the *env* section or in *os.environ* itself.

You just have to use the *circus.env.* prefix.

Example:

```
[watcher:worker1]
cmd = $(circus.env.shell)

[watcher:worker2]
baz = $(circus.env.user)
bar = $(circus.env.yeah)
sup = $(circus.env.oh)

[socket:socket1]
port = $(circus.env.port)

[plugin:plugin1]
use = some.path
parameter1 = $(circus.env.plugin_param)

[env]
yeah = boo

[env:worker2]
oh = ok
```

If a variable is defined in several places, the most specialized value has precedence: a variable defined in *env:XXX* will override a variable defined in *env*, which will override a variable defined in *os.environ*.

environment substitutions can be used in any section of the configuration in any section variable.

Formatting 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.

Stream configuration Simple stream class like *QueueStream* and *StdoutStream* don't have specific attributes but some other stream class may have some:

FileStream

filename The file path where log will be written.

time_format The strftime format that will be used to prefix each time with a timestamp. By default they will be not prefixed.

i.e: %Y-%m-%d %H:%M:%S

max_bytes The max size of the log file before a new file is started. If not provided, the file is not rolled over.

backup_count The number of log files that will be kept By default backup_count is null.

Note: Rollover occurs whenever the current log file is nearly max_bytes in length. If backup_count is >= 1, the system will successively create new files with the same pathname as the base file, but with extensions ".1", ".2" etc. appended to it. For example, with a backup_count of 5 and a base file name of "app.log", you would get "app.log", "app.log.1", "app.log.2", ... through to "app.log.5". The file being written to is always "app.log" - when it gets filled up, it is closed and renamed to "app.log.1", and if files "app.log.1", "app.log.2" etc. exist, then they are renamed to "app.log.2", "app.log.3" etc. respectively.

Example:

```
[watcher:myprogram]
cmd = python -m myapp.server

stdout_stream.class = FileStream
stdout_stream.filename = test.log
stdout_stream.time_format = %Y-%m-%d %H:%M:%S
stdout_stream.max_bytes = 1073741824
stdout_stream.backup_count = 5
```

WatchedFileStream

filename The file path where log will be written.

time_format The strftime format that will be used to prefix each time with a timestamp. By default they will be not prefixed.

i.e: %Y-%m-%d %H:%M:%S

Note: WatchedFileStream relies on an external log rotation tool to ensure that log files don't become too big. The output file will be monitored and if it is ever deleted or moved by the external log rotation tool, then the output file handle will be automatically reloaded.

Example:

```
[watcher:myprogram]
cmd = python -m myapp.server

stdout_stream.class = WatchedFileStream
stdout_stream.filename = test.log
stdout_stream.time_format = %Y-%m-%d %H:%M:%S
```

TimedRotatingFileStream

filename The file path where log will be written.

backup_count The number of log files that will be kept By default backup_count is null.

time_format The strftime format that will be used to prefix each time with a timestamp. By default they will be not prefixed.

i.e: %Y-%m-%d %H:%M:%S

rotate_when The type of interval. The list of possible values is below. Note that they are not case sensitive.

Value	Type of interval
'S'	Seconds
'M'	Minutes
'H'	Hours
'D'	Days
'W0'-'W6'	Weekday (0=Monday)
'midnight'	Roll over at midnight

rotate_interval The rollover interval.

Note: TimedRotatingFileStream rotates logfiles at certain timed intervals. Rollover interval is determined by a combination of rotate_when and rotate_interval.

Example:

```
[watcher:myprogram]
cmd = python -m myapp.server

stdout_stream.class = TimedRotatingFileStream
stdout_stream.filename = test.log
stdout_stream.time_format = %Y-%m-%d %H:%M:%S
stdout_stream.utc = True
stdout_stream.rotate_when = H
stdout_stream.rotate_interval = 1
```

FancyStdoutStream

color

The name of an ascii color:

- red
- green
- yellow
- blue
- magenta
- cyan
- white

time_format The strftime format that each line will be prefixed with.

Default to: %Y-%m-%d %H:%M:%S

Example:

```
[watcher:myprogram]
cmd = python -m myapp.server
stdout_stream.class = FancyStdoutStream
stdout_stream.color = green
stdout_stream.time_format = %Y/%m/%d | %H:%M:%S
```

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.

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 specific watcher options*
- **globaloptions**: *Get the arbiter options*
- **incr**: *Increment the number of processes in a watcher*
- **ipython**: *Create shell into circusd process*
- **list**: *Get list of watchers or processes in a watcher*
- **listen**: *Subscribe 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 all options for a watcher*
- **quit**: *Quit the arbiter immediately*
- **reload**: *Reload the arbiter or a watcher*
- **reloadconfig**: *Reload the configuration file*
- **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 watchers*

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>
        "waiting": False
    }
}
```

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

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

Command line

```
$ circusctl decr <name> [<nb>] [--waiting]
```

Options

- <name>: name of the watcher
- <nb>: 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 specific watcher options This command can be used to query the current value of one or more watcher options.

ZMQ Message

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

A request message contains two properties:

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

The response object has a property `options` which is a dictionary of option names and values.

eg:

```
{
    "status": "ok",
    "options": {
        "graceful_timeout": 300,
        "send_hup": True,
    },
    time': 1332202594.754644
}
```

Command line

```
$ circusctl get <name> <key1> <key2>
```

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
- multicast_endpoint: the multicast endpoint for circusd cluster auto-discovery

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>,
        "waiting": False
    }
}
```

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

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

Command line

```
$ circusctl incr <name> [<nb>] [--waiting]
```

Options

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

Create shell into circusd process This command is only useful if you have the ipython package installed.

Command Line

```
$ circusctl ipython
```

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>]
```

Subscribe to a watcher event

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

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 struct.

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

```
circusctl listen [<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 all options for a watcher This command returns all option values for a given watcher.

ZMQ Message

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

A message contains 1 property:

- name: name of watcher

The response object has a property `options` which is a dictionary of option names and values.

eg:

```
{
    "status": "ok",
    "options": {
        "graceful_timeout": 300,
        "send_hup": True,
        ...
    },
    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
- 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",
    "waiting": False
}
```

The response return the status "ok".

If `waiting` is False (default), the call will return immediately after calling `stop_signal` on each process.

If `waiting` is True, the call will return only when the stop process is completely ended. Because of the *graceful_timeout option*, it can take some time.

Command line

```
$ circusctl quit [--waiting]
```

Reload the arbiter or a watcher This command reloads all the process in a watcher or all watchers. This will happen in one of 3 ways:

- If graceful is false, a simple restart occurs.
- If *send_hup* is true for the watcher, a HUP signal is sent to each process.
- **Otherwise:**
  - If sequential is false, the arbiter will attempt to spawn *numprocesses* new processes. If the new processes are spawned successfully, the result is that all of the old processes are stopped, since by default the oldest processes are stopped when the actual number of processes for a watcher is greater than *numprocesses*.
  - If sequential is true, the arbiter will restart each process in a sequential way (with a *warmup_delay* pause between each step)

ZMQ Message

```
{
    "command": "reload",
    "properties": {
        "name": '<name>",
        "graceful": true,
        "sequential": false,
        "waiting": False
    }
}
```

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] [--waiting]
                            [--sequential]
```

Options

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

Reload the configuration file This command reloads the configuration file, so changes in the configuration file will be reflected in the configuration of circus.

ZMQ Message

```
{
    "command": "reloadconfig",
    "waiting": False
}
```

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

Command line

```
$ circusctl reloadconfig [--waiting]
```

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",
    "properties": {
        "name": "<name>",
        "waiting": False
    }
}
```

The response return the status "ok".

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

If `waiting` is False (default), the call will return immediately after calling *stop_signal* on each process.

If `waiting` is True, the call will return only when the restart process is completely ended. Because of the *graceful_timeout option*, it can take some time.

Command line

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

Options

- <name>: name of the watcher

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

ZMQ Message

```
{
    "command": "rm",
    "properties": {
        "name": "<nameofwatcher>",
        "waiting": False
```

```
    }
}
```

The response return a status "ok".

If `waiting` is False (default), the call will return immediatly after starting to remove and stop the corresponding watcher.

If `waiting` is True, the call will return only when the remove and stop process is completly ended. Because of the *graceful_timeout option*, it can take some time.

Command line

```
$ circusctl rm <name> [--waiting]
```

Options

- <name>: name of the watcher to remove

Set a watcher option

ZMQ Message

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

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> --waiting
```

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 children of the watcher:

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

Lastly, 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> [<pid>] [--children]
        [--recursive] <signum>
```

Options:

- <name>: the name of the watcher

- <pid>: integer, the process id.
- <signum>: the signal number (or name) 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

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>",
        "waiting": False
    }
}
```

The response return the status "ok".

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

If `waiting` is False (default), the call will return immediately after calling *start* on each process.

If `waiting` is True, the call will return only when the start process is completely ended. Because of the *graceful_timeout option*, it can take some time.

Command line

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

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>
    }
}
```

Stats can be extended with the extended_stats hook but extended stats need to be requested:

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

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 [--extended] [<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 watchers This command stops a given watcher or all watchers.

ZMQ Message

```
{
    "command": "stop",
    "properties": {
        "name": "<name>",
        "waiting": False
    }
}
```

The response returns the status "ok".

If the `name` property is present, then the stop will be applied to the watcher corresponding to that name. Otherwise, all watchers will get stopped.

If `waiting` is False (default), the call will return immediatly after calling *stop_signal* on each process.

If `waiting` is True, the call will return only when the stop process is completly ended. Because of the *graceful_timeout option*, it can take some time.

Command line

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

Options

- <name>: name of the watcher

CLI tools

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*.

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
```

Besides supporting a handful of options you can also specify the endpoint *circusctl* should use using the `CIRCUSCTL_ENDPOINT` environment variable.

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]
statsd = True
```

By default, this option is not activated.

The web console is its own package, you need to install:

```
$ pip install circus-web
```

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

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

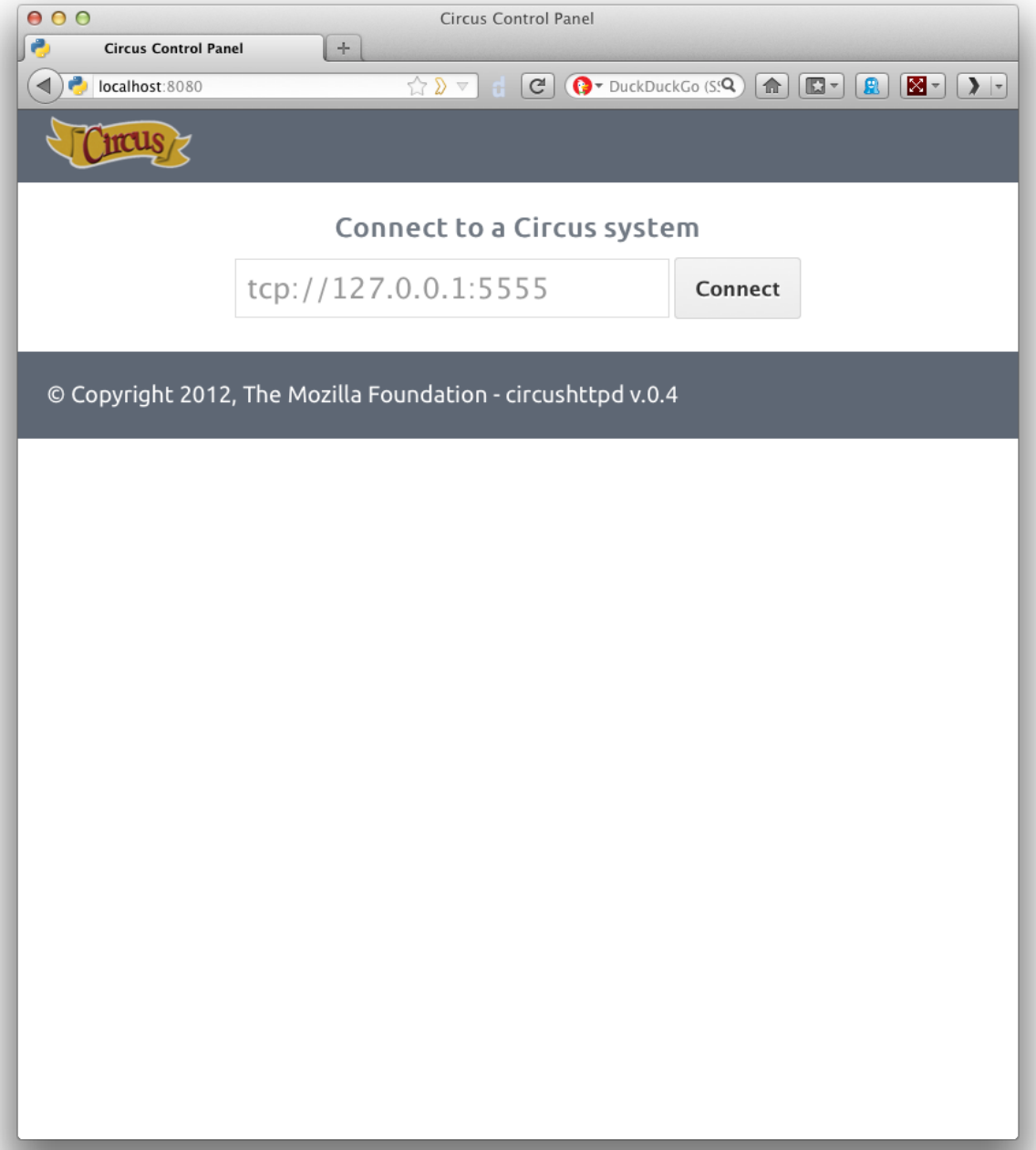
httpd_host and *httpd_port* are optional, and default to *localhost* and *8080*.

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.

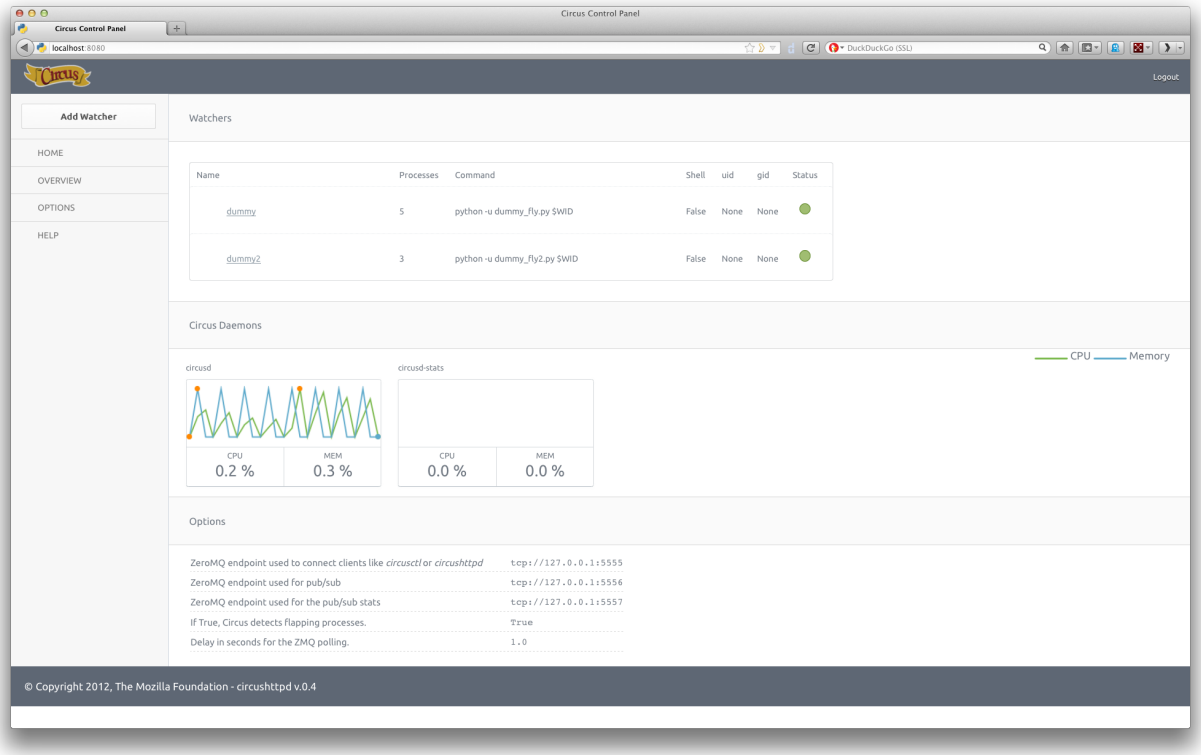
Using the console Once the script is running, you can open a browser and visit *http://localhost:8080*. You should get this screen:



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*.

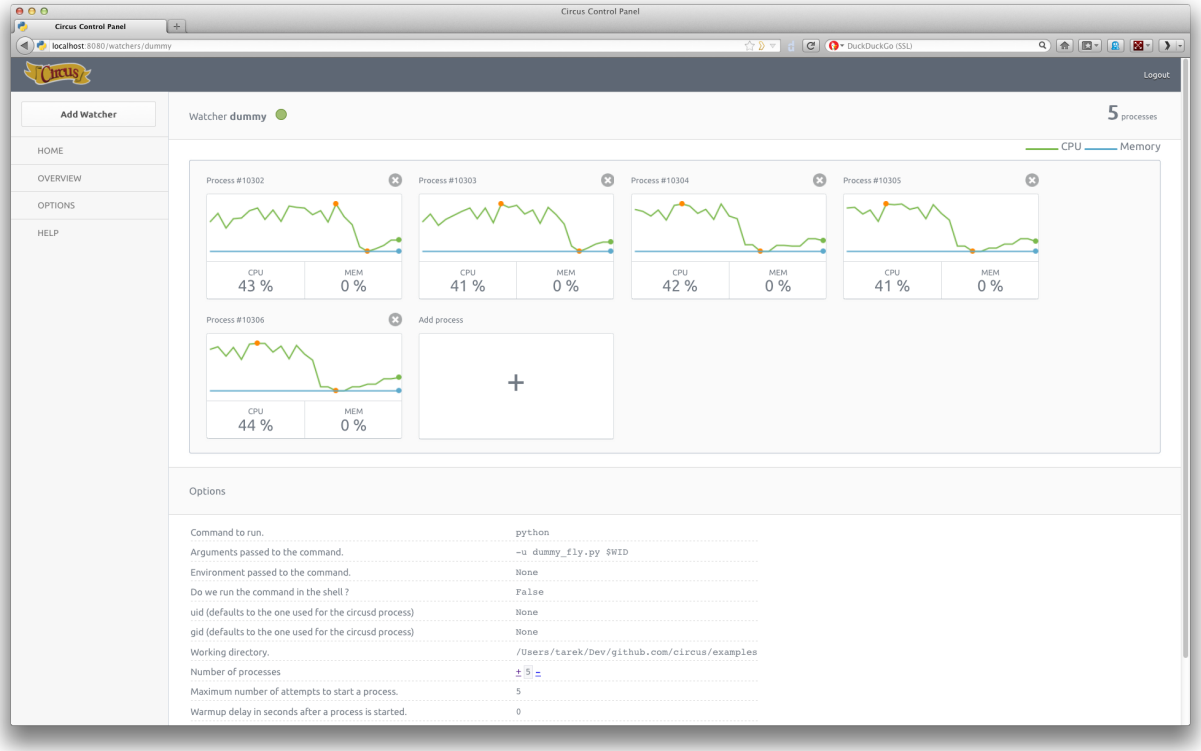
Once you hit *Connect*, the web application will connect to the Circus system.

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).



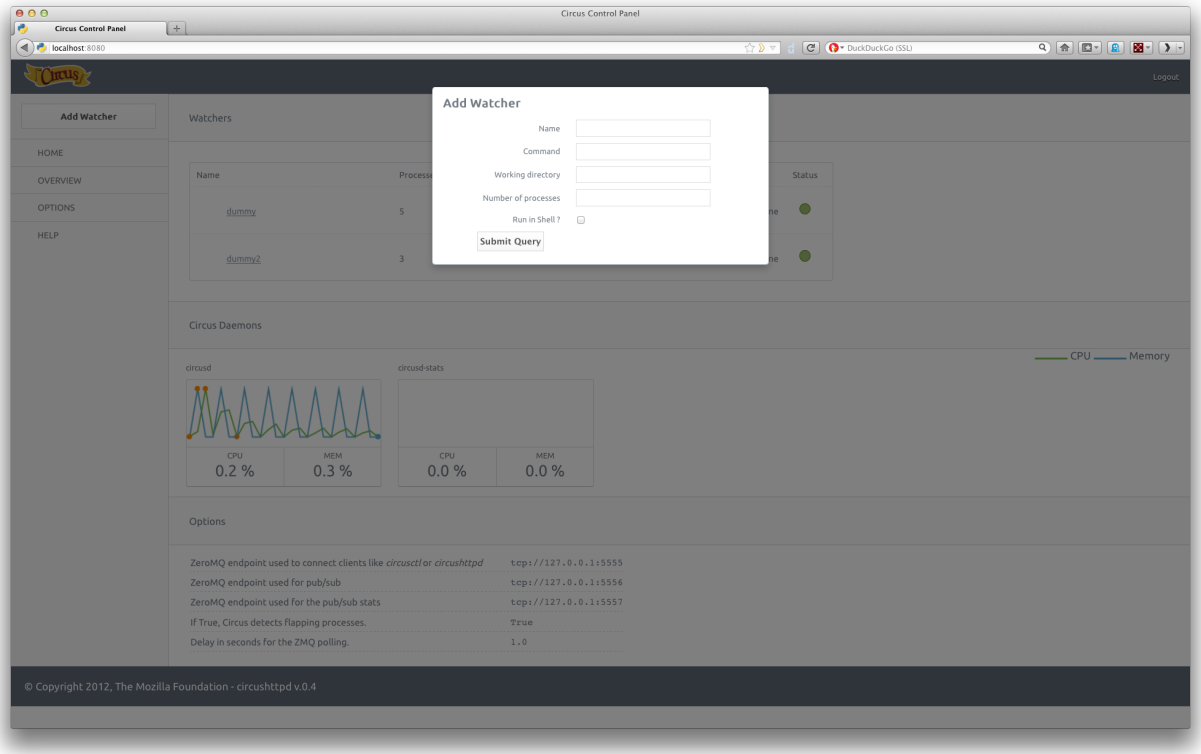
You can click on the status of each watcher to toggle it from **Active** (green) to **Inactive** (red). This change is effective immediatly and let you start & stop watchers.

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:



Running behind Nginx and Varnish Nginx can act as a proxy in front of Circus. It can 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.

Running behind Nginx >= 1.3.13 As of Nginx>=1.3.13 websockets are supported by the web server. With Nginx>=1.3.13 there is no longer a need to reroute websocket traffic via Varnish or HAProxy.

On Ubuntu you can install Nginx>=1.3.13 from Chris Lea's development branch PPA, as so:

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:chris-lea/nginx-devel
sudo apt-get update
sudo apt-get install nginx
nginx -v
```

An example Nginx config with websocket support:

```
# /etc/nginx/sites-enabled/default

upstream circusweb_server {
  server localhost:8080;
}

server {
 listen   80;
 server_name  _;

 location / {
   proxy_pass http://circusweb_server;
   proxy_http_version 1.1;
   proxy_set_header Upgrade $http_upgrade;
   proxy_set_header Connection "upgrade";
   proxy_set_header Host $http_host;
   proxy_set_header X-Forwarded-Host $http_host;
   proxy_set_header X-Real-IP $remote_addr;
   proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
   proxy_set_header X-Forwarded-Proto $scheme;
   proxy_redirect off;
  }

 location ~/media/\*(.png|.jpg|.css|.js|.ico)$ {
   alias /path_to_site-packages/circusweb/media/;
  }
}
```

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_set_header X-Forwarded-Host: $http_host;
    proxy_set_header X-Forwarded-Proto: $scheme;
    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

However, there's no native support for the combined use of HTTP Authentication and WebSockets (the server will throw HTTP 401 error codes). A workaround is to disable such authentication for the socket.io server.

Example (needs to be added before the previous rule):

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

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

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.

Working with 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.

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.

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*.

Note: Starting at Circus 0.8 there's an alternate syntax to avoid some conflicts with some config parsers. You can write:

```
*((circus.sockets.foo))*
```

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.

Using built-in plugins

Circus comes with a few built-in plugins. This section presents these plugins and their configuration options.

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

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.

RedisObserver

This services observers a redis process for you, publishes the information to statsd and offers to restart the watcher 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.

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 watcher 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.

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.

watcher the watcher this resource watcher should be looking after. (previously called `service` but `service` is now deprecated)

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

min_cpu The minimum cpu one process should consume (in %). Default: None (no minimum) You can set the min_cpu to 0 (zero), in this case if one process consume exactly 0% cpu, it will trigger an exceeded limit.

max_mem The amount of memory one process of this watcher is allowed to consume. Default: 90. If no unit is specified, the value is in %. Example: 50 If a unit is specified, the value is in bytes. Supported units are B, K, M, G, T, P, E, Z, Y. Example: 250M

min_mem The minimum memory one process of this watcher should consume. Default: None (no minimum). If no unit is specified, the value is in %. Example: 50 If a unit is specified, the value is in bytes. Supported units are B, K, M, G, T, P, E, Z, Y. Example: 250M

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

Example:

```
[circus]
; ...

[watcher:program]
cmd = sleep 120

[plugin:myplugin]
use = circus.plugins.resource_watcher.ResourceWatcher
watcher = program
min_cpu = 10
max_cpu = 70
min_mem = 0
max_mem = 20
```

Watchdog

Plugin that binds an udp socket and wait for watchdog messages. For "watchdoged" processes, the watchdog will kill them if they don't send a heartbeat in a certain period of time materialized by loop_rate * max_count. (circus will automatically restart the missing processes in the watcher)

Each monitored process should send udp message at least at the loop_rate. The udp message format is a line of text, decoded using **msg_regex** parameter. The heartbeat message MUST at least contain the pid of the process sending the message.

The list of monitored watchers are determined by the parameter **watchers_regex** in the configuration.

Configuration parameters:

use set to `circus.plugins.watchdog.WatchDog`

loop_rate watchdog loop rate in seconds. At each loop, WatchDog will looks for "dead" processes.

watchers_regex regex for matching watcher names that should be monitored by the watchdog (default: `.*` all watchers are monitored)

msg_regex regex for decoding the received heartbeat message in udp (default: `^(?P<pid>.*);(?P<timestamp>.*)$`) the default format is a simple text message: `pid;timestamp`

max_count max number of passed loop without receiving any heartbeat before restarting process (default: 3)

ip ip the watchdog will bind on (default: 127.0.0.1)

port port the watchdog will bind on (default: 1664)

Flapping

When a worker restarts too often, we say that it is *flapping*. This plugin keeps track of worker restarts and stops the corresponding watcher in case it is flapping. This plugin may be used to automatically stop workers that get constantly restarted because they're not working properly.

use set to `circus.plugins.flapping.Flapping`

attempts the number of times a process can restart, within **window** seconds, before we consider it flapping (default: 2)

window the time window in seconds to test for flapping. If the process restarts more than **attempts** times within this time window, we consider it a flapping process. (default: 1)

retry_in time in seconds to wait until we try to start again a process that has been flapping. (default: 7)

max_retry the number of times we attempt to start a process that has been flapping, before we abandon and stop the whole watcher. (default: 5) Set to -1 to disable max_retry and retry indefinitely.

active define if the plugin is active or not (default: True). If the global flag is set to False, the plugin is not started.

Options can be overriden in the watcher section using a `flapping.` prefix. For instance, here is how you would configure a specific `max_retry` value for nginx:

```
[watcher:nginx]
cmd = /path/to/nginx
flapping.max_retry = 2

[watcher:myscript]
cmd = ./my_script.py

; ... other watchers

[plugin:flapping]
use = circus.plugins.flapping.Flapping
max_retry = 5
```

CommandReloader

This plugin will restart watchers when their command file is modified. It works by checking the modification time and the path of the file pointed by the **cmd** option every **loop_rate** seconds. This may be useful while developing worker processes or even for hot code upgrade in production.

use set to `circus.plugins.command_reloader.CommandReloader`

loop_rate the frequency the plugin should check for modification in seconds. Default: 1.

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 runlevel [016]

respawn
exec /usr/local/bin/circusd /etc/circus/circusd.ini
```

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

```
# service circus start/stop/restart
```

If your system supports systemd, you can create this systemd unit file under /etc/systemd/system/circus.service.

```
[Unit]
Description=Circus process manager
After=syslog.target network.target nss-lookup.target

[Service]
Type=simple
ExecReload=/usr/bin/circusctl reload
ExecStart=/usr/bin/circusd /etc/circus/circus.ini
Restart=always
RestartSec=5

[Install]
WantedBy=default.target
```

A reboot isn't required if you run the daemon-reload command below:

```
# systemctl --system daemon-reload
```

Then circus can be managed via:

```
# systemctl start/stop/status/reload circus
```

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

2.3.4 Circus for developers

Using Circus as a library

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

myprogram = {"cmd": "python myprogram.py", "numprocesses": 4}

arbiter = get_arbiter([myprogram])
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*

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 *Using built-in 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*.

Last but not least, you can also add new commands. See *Adding new commands*.

Developers Documentation Index

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.

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`**`()`

Example:

```
from circus import get_arbiter

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

Classes 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*, *pipe_stdout=True*, *pipe_stderr=True*, *close_child_stdout=False*, *close_child_stderr=False*)

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.
- **pipe_stdout**: if True, will open a PIPE on stdout. default: True.
- **pipe_stderr**: if True, will open a PIPE on stderr. default: True.
- **close_child_stdout**: If True, redirects the child process' stdout to /dev/null after the fork. default: False.
- **close_child_stderr**: If True, redirects the child process' stdout to /dev/null after the fork. default: False.

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

children()
Return a list of children pids.

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.

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

pid
Return the *pid*

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

stderr
Return the *stdout* stream

stdout
Return the *stdout* stream

stop(**args*, ***kw*)
Stop the process and close stdout/stderr

If the corresponding process is still here (normally it's already killed by the watcher), a SIGTERM is sent, then a SIGKILL after 1 second.

The shutdown process (SIGTERM then SIGKILL) is normally taken by the watcher. So if the process is still there here, it's a kind of bad behavior because the graceful timeout won't be respected here.

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*, *shell_args=None*, *uid=None*, *max_retry=5*, *gid=None*, *send_hup=False*, *stop_signal=15*, *stop_children=False*, *env=None*, *graceful_timeout=30.0*, *prereload_fn=None*, *rlimits=None*, *executable=None*, *stdout_stream=None*, *stderr_stream=None*, *priority=0*, *loop=None*, *singleton=False*, *use_sockets=False*, *copy_env=False*, *copy_path=False*, *max_age=0*, *max_age_variance=30*, *hooks=None*, *respawn=True*, *autostart=True*, *on_demand=False*, *virtualenv=None*, *close_child_stdout=False*, *close_child_stderr=False*, ***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.
- **stop_signal**: the signal to send when stopping the process. Defaults to SIGTERM.
- **stop_children**: send the **stop_signal** to the children too. 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

  – **max_bytes**: maximum file size, after which a new output file is opened. defaults to 0 which means no maximum size (only applicable with FileStream).

  – **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 (only applicable with FileStream)

  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 - **max_bytes**: maximum file size, after which a new output file is

opened. defaults to 0 which means no maximum size (only applicable with FileStream)

–**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 (only applicable with FileStream).

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

•**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)

•**on_demand** – If True, the processes will be started only at the first connection to the socket (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_spawn*, *after_spawn*, *before_stop*, *after_stop*., *before_signal*, *after_signal* or *extended_stats*.

•**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)

•**virtualenv** – The root directory of a virtualenv. If provided, the watcher will load the environment for its execution. (default: None)

•**close_child_stdout**: If True, closes the stdout after the fork. default: False.

•**close_child_stderr**: If True, closes the stderr after the fork. default: False.

kill_process(**args*, ***kwargs*)
Kill process (stop_signal, graceful_timeout then SIGKILL)

kill_processes(**args*, ***kwargs*)
Kill all processes (stop_signal, graceful_timeout then SIGKILL)

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

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

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

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

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

spawn_process()
Spawn process.

Return True if ok, False if the watcher must be stopped

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

Writing plugins

Circus comes with a plugin system which lets 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 adds or removes processes depending on the load
- etc.

Circus itself comes with a few *built-in plugins*.

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

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.

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 may look like this:

```
from circus.plugins import CircusPlugin


class Logger(CircusPlugin):

    name = 'logger'

    def __init__(self, *args, **config):
        super(Logger, self).__init__(*args, **config)
        self.filename = config.get('filename')
        self.file = None

    def handle_init(self):
        self.file = open(self.filename, 'a+', buffering=1)

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

    def handle_recv(self, data):
        watcher_name, action, msg = self.split_data(data)
        msg_dict = self.load_message(msg)
        self.file.write('%s %s:%r\n' % (action, watcher_name, msg_dict))
```

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

For example, `Logger` may be found in a *plugins* module within 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 is 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 executed as expected.

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 --config filename:circu
[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*.

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.

Hooks

Circus provides hooks that can be used to trigger actions upon watcher events. Available hooks are:

- **before_start**: called before the watcher is started. If the hook returns **False** the startup is aborted.
- **after_start**: called after the watcher is started. If the hook returns **False** the watcher is immediately stopped and the startup is aborted.
- **before_spawn**: called before the watcher spawns a new process. If the hook returns **False** the watcher is immediately stopped and the startup is aborted.
- **after_spawn**: called after the watcher spawns a new process. If the hook returns **False** the watcher is immediately stopped and the startup is aborted.
- **before_stop**: called before the watcher is stopped. The hook result is ignored.
- **after_stop**: called after the watcher is stopped. The hook result is ignored.
- **before_signal**: called before a signal is sent to a watcher's process. If the hook returns **False** the signal is not sent (except SIGKILL which is always sent)
- **after_signal**: called after a signal is sent to a watcher's process.
- **extended_stats**: called when stats are requested with extended=True. Used for adding process-specific stats to the regular stats output.

Example 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* first and then it will start 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 an `before_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.

Hook signature A hook must follow this signature:

```
def hook(watcher, arbiter, hook_name, **kwargs):
    ...
    # If you don't return True, the hook can change
    # the behavior of circus (depending on the hook)
    return True
```

Where **watcher** is the **Watcher** class instance, **arbiter** the **Arbiter** one, **hook_name** the hook name and **kwargs** some additional optional parameters (depending on the hook type).

The **after_spawn** hook adds the pid parameters:

```
def after_spawn(watcher, arbiter, hook_name, pid, **kwargs):
    ...
    # If you don't return True, circus will kill the process
    return True
```

Where **pid** is the PID of the corresponding process.

Likewise, **before_signal** and **after_signal** hooks add pid and signum:

```
def before_signal_hook(watcher, arbiter, hook_name, pid, signum, **kwargs):
    ...
    # If you don't return True, circus won't send the signum signal
    # (SIGKILL is always sent)
    return True
```

Where **pid** is the PID of the corresponding process and **signum** is the corresponding signal.

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

Note that hooks are called with named arguments. So use the hook signature without changing argument names.

The **extended_stats** hook has its own additional parameters in **kwargs**:

```
def extended_stats_hook(watcher, arbiter, hook_name, pid, stats, **kwargs):
    ...
```

Where **pid** is the PID of the corresponding process and **stats** the regular stats to be returned. Add your own stats into **stats**. An example is in examples/uwsgi_lossless_reload.py.

As a last example, here is a super hook which can deal with all kind of signals:

```
def super_hook(watcher, arbiter, hook_name, **kwargs):
    pid = None
    signum = None
    if hook_name in ('before_signal', 'after_signal'):
        pid = kwargs['pid']
        signum = kwargs['signum']
    ...
    return True
```

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.

Adding new commands

We tried to make adding new commands as simple as possible.

You need to do three 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 currently 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 automatically included in the documentation,
    so don't be affraid of being exhaustive, that's what it is made
    for.
    """
    # all the commands inherit from 'circus.commands.base.Command'

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

    # Set waiting to True or False to define your default behavior
    # - If waiting is True, the command is run synchronously, and the client may get
    #   back results.
    # - If waiting is False, the command is run asynchronously on the server and the client immediate
```

```
    #   gets back an 'ok' response
    #
    #   By default, commands are set to waiting = False
    waiting = True

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

    properties = ['foo', 'bar']
    # properties list the command arguments that are mandatory. 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.
```

2.3.5 Use cases examples

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

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 *How does Circus stack compare to a 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 *CLI tools*, and enjoy Circus management.

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:

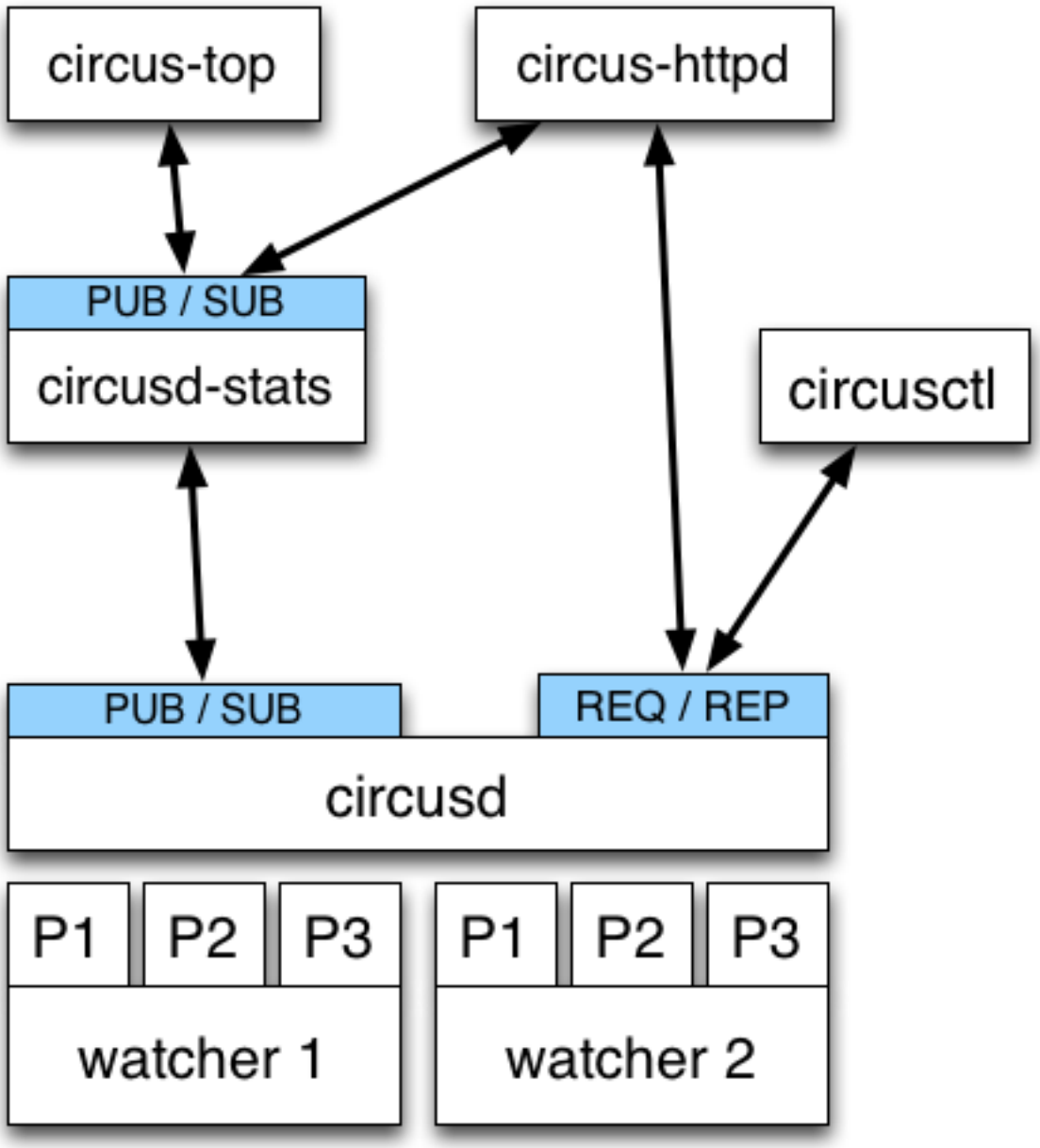
```
[watcher:dbackend]
cmd = /path/to/script.py
numprocesses=3

[env:dbackend]
PYTHONPATH = /path/to/parent-of-dproject
DJANGO_SETTINGS_MODULE=dproject.settings
```

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

2.3.6 Design decisions

Overall architecture



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: Despite 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.

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

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 statistics 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. The commands must also be run as a user that has write access to the ipc socket paths. You can modify the owner of the **endpoint** using the **endpoint_owner** config option. This allows you to run circusd as the root user, but allow non-root processes to send commands to **circusd**. Note that when using **endpoint_owner**, in

order to prevent non-root processes from being able to start arbitrary processes that run with greater privileges, the add command will enforce that new Watchers must run as the **endpoint_owner** user. Watcher definitions in the local config files will not be restricted this way.

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

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, that can take care of the security.

User and Group Permissions

By default, all processes started with Circus will be running with the same user and group as **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 be 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.

SSH tunneling

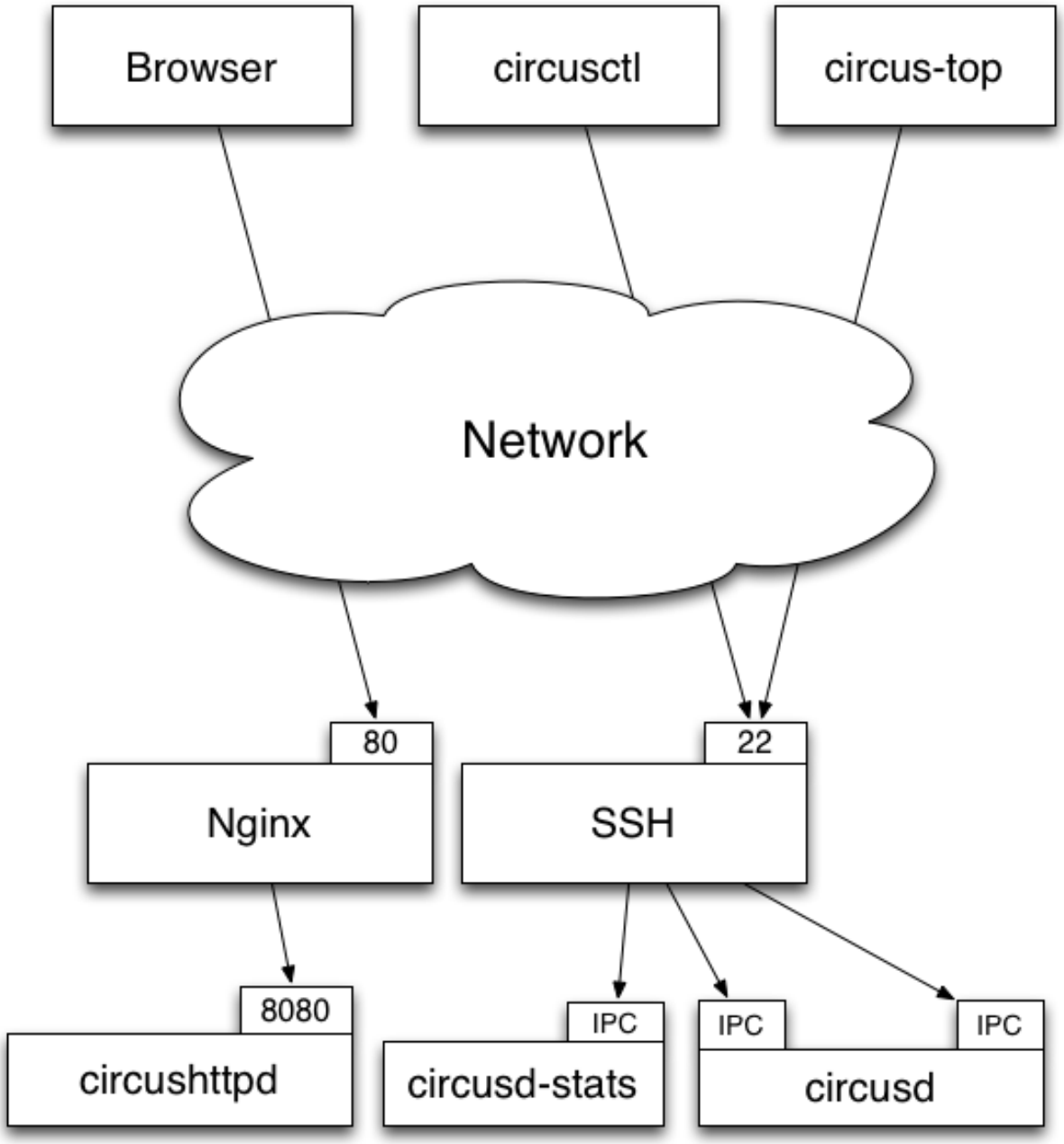
Clients can connect to a **circusd** instance by creating an SSH tunnel. To do so, pass the command line option **–ssh** followed by **user@address**, where **user** is the user on the remote server and **address** is the server's address as seen by the client. The SSH protocol will require credentials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

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.



2.3.7 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

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.

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.

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.

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
```

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)

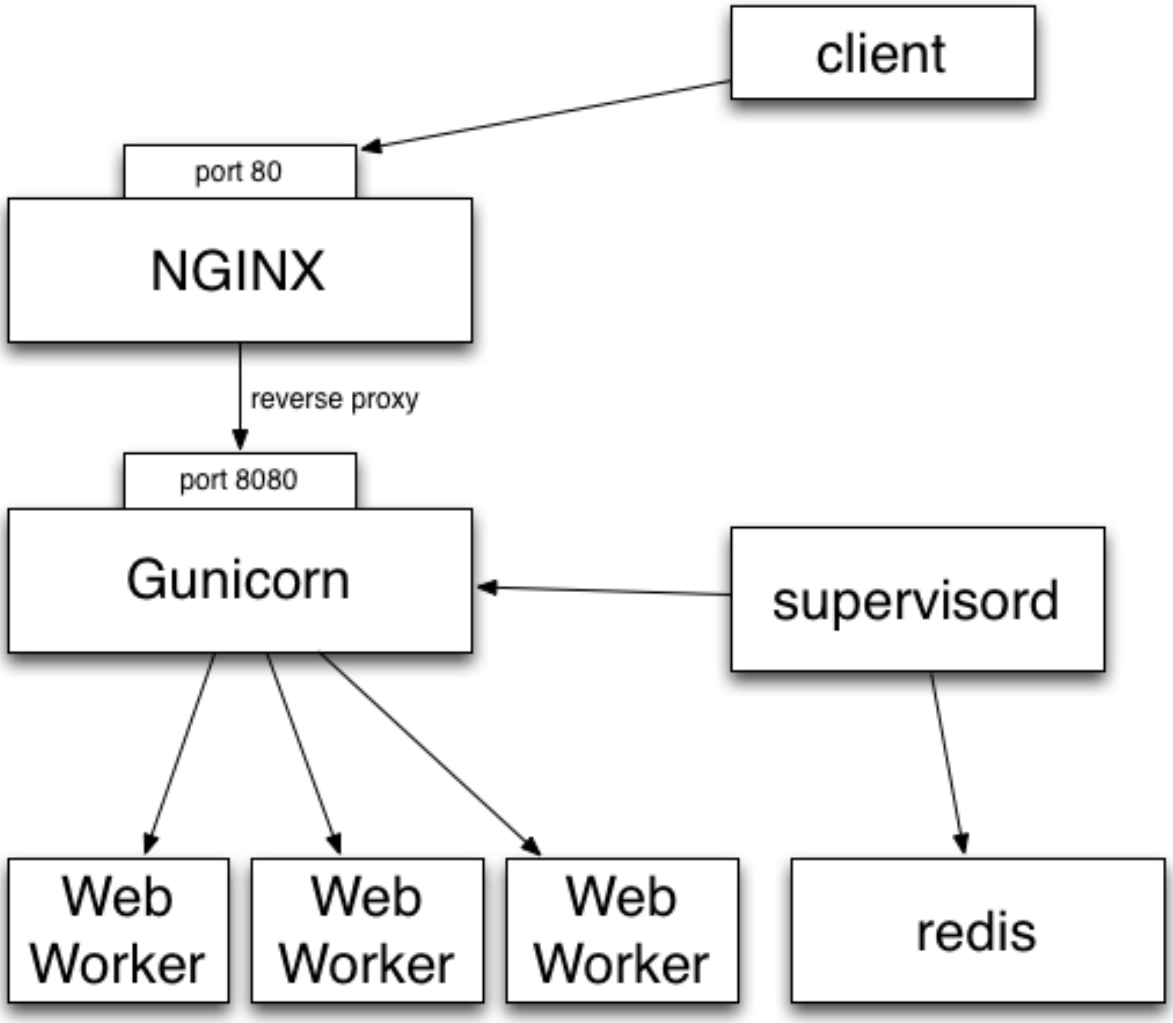
You can also start a thread in our mailing list - http://tech.groups.yahoo.com/group/circus-dev

2.3.8 Frequently Asked Questions

Here is a list of frequently asked questions about Circus:

How does Circus stack compare to a 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").

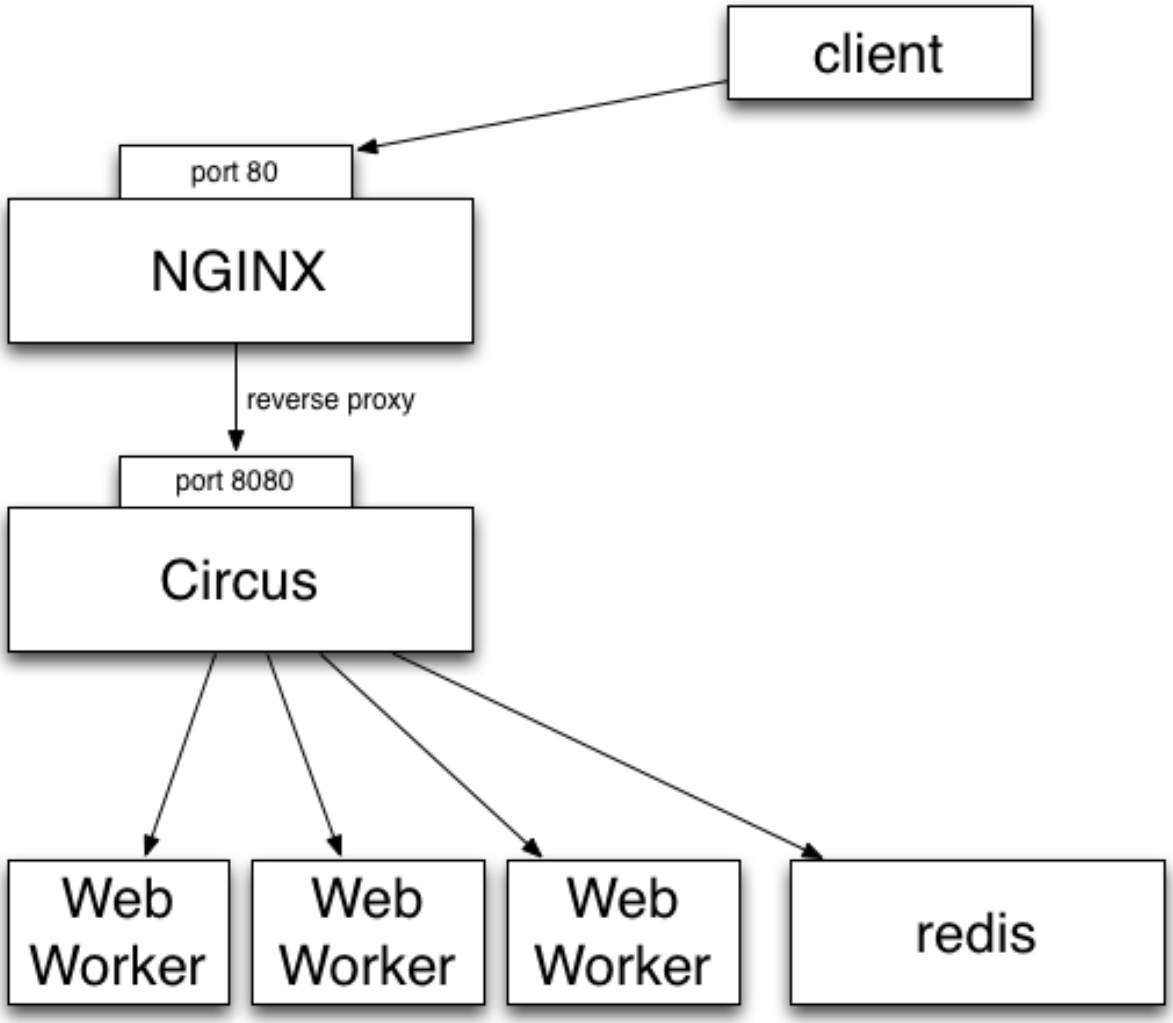
In other words you are using two levels of process managment. One that you manage and control (supervisord), and a second one that you have to manage in a different UI, with a different philosophy and less control over what's going on (the wsgi server's one)

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

How to troubleshoot Circus?

By default, *circusd* keeps its logging to *stdout* rather sparse. This lack of output can make things hard to troubleshoot when processes seem to be having trouble starting.

To increase the logging *circusd* provides, try increasing the log level. To see the available log levels just use the *–help* flag.

```
$ circus --log-level debug test.ini
```

One word of warning. If a process is flapping and the debug log level is turned on, you will see messages for each start attempt. It might be helpful to configure the app that is flapping to use a *warmup_delay* to slow down the messages to a manageable pace.

```
[watcher:webapp]
cmd = python -m myapp.wsgi
warmup_delay = 5
```

By default, *stdout* and *stderr* are captured by the *circusd* process. If you are testing your config and want to see the output in line with the circusd output, you can configure your watcher to use the *StdoutStream* class.

```
[watcher:webapp]
cmd = python -m myapp.wsgi
stdout_stream.class = StdoutStream
stderr_stream.class = StdoutStream
```

If your application is producing a traceback or error when it is trying to start up you should be able to see it in the output.

2.3.9 Changelog history

0.12 - 2014-05-22

- Fixed a regression that broke Circus on 2.6 - #782

0.11 - 2014-05-21

This release is not introducing a lot of features, and focused on making Circus more robust & stable.

Major changes/fixes:

- Make sure we cannot execute two conflictings commands on the arbiter simultanously.
- we have 2 new streams class: TimedRotatingFileStream, WatchedFileStream
- we have one new hook: after_spawn hook

- CircusPlugin is easier to use
- fix autostart=False watchers during start (regression)

More changes:

- circus messages can be routed to syslog now - #748
- endpoint_owner option added so we can define which user owns ipc socket files created by circus.
- Started Windows support (just circusctl for now)
- fixed a lot of leaks in the tests
- Allow case sensitive environment variables
- The resource plugin now accepts absolute memory values - #609
- Add support to the add command for the 'singleton' option - #767
- Allow sending arbitrary signals to child procs via resource watcher - #756
- Allow INI/JSON/YAML configuration for logging
- Make sure we're compatible with psutil 2.x *and* 3.x
- Added more metrics to the statsd provider - #698
- Fixed multicast discovery - #731
- Make start, restart and reload more uniform - #673
- Correctly initialize all use groups - #635
- improved tests stability
- many, many more things....

0.10 - 2013-11-04

Major changes:

- Now Python 3.2 & 3.3 compatible - #586
- Moved the core to a fully async model - #569
- Improved documentation - #622

More changes:

- Added stop_signal & stop_children - #594
- Make sure the watchdog plugin closes the sockets - #588
- Switched to ZMQ JSON parser
- IN not supported on all platforms - #573
- Allow global environment substitutions in any config section - #560
- Allow dashes in sections names - #546
- Now variables are expanded everywhere in the config - #554
- Added the CommandReloader plugin
- Added before_signal & after_signal hooks
- Allow flapping plugin to retry indefinitely

- Don't respawn procs when the watcher is stopping - #529 - #536
- Added a unique id for each client message - #517
- worker ids are now "slots" -
- Fixed the graceful shutdown behavior - #515
- Make sure we can add watchers even if the arbiter is not started - #503
- Make sure make sure we pop expired process - #510
- Make sure the set command can set several hooks
- Correctly support ipv6 sockets - #507
- Allow custom options for stdout_stream and stderr_stream - #495
- Added time_format for FileStream - #493
- Added new socket config option to bind to a specific interface by name

0.9.3 - 2013-09-04

- Make sure we can add watchers even if the arbiter is not started
- Make sure we pop expired process
- Make sure the set command can set one or several hooks
- Correctly support ipv6 sockets and improvments of CircusSockets
- Give path default value to prevent UnboundLocalError
- Added a test for multicast_endpoint existence in Controller initialization
- Not converting every string of digits to ints anymore
- Add tests
- No need for special cases when converting stdout_stream options
- also accept umask as an argument for consistency
- Allow custom options for stdout_stream and stderr_stream.
- Add new socket config option to bind to a specific interface by name
- Add time_format for FileStream + tests
- Update circus.upstart

0.9.2 - 2013-07-17

- When a PYTHONPATH is defined in a config file, it's loaded in sys.path so hooks can be located there - #477, #481
- Use a single argument for add_callback so it works with PyZMQ < 13.1.x - see #478

0.9 - 2013-07-16

- added [env] sections wildcards
- added global [env] secrtion
- fixed hidden exception when circus-web is not installed - #424
- make sure incr/decr commands really us the nb option - #421
- Fix watcher virtualenv site-packages not in PYTHONPATH
- make sure we dont try to remove more processes than 0 - #429
- updated bootstrap.py - #436
- fixed multiplatform separator in pythonpath virtualenv watcher
- refactored socket close function
- Ensure env sections are applied to all watchers - #437
- added the reloadconfig command
- added circus.green and removed gevent from the core - #441, #452
- silenced spurious stdout & warnings in the tests - #438
- $(circus.env.*) can be used for all options in the config now
- added a before_spawn hook
- correct the path of circusd in systemd service file - #450
- make sure we can change hooks and set streams via CLI - #455
- improved doc
- added a spawn_count stat in watcher
- added min_cpu and min_mem parameters in ResourceWatcher plugin
- added the FQDN information to the arbiter.

0.8.1 - 2013-05-28

- circusd-stats was choking on unix sockets - #415
- circusd-stats & circushttpd child processes stdout/stderr are now left open by default. Python <= 2.7.5 would choke in the logging module in case the 2/3 fds were closed - #415
- Now redirecting to /dev/null in the child process instead of closing. #417

0.8 - 2013-05-24

- Integrated log handlers into zmq io loop.
- Make redirector restartable and subsequently more robust.
- Uses zmq.green.eventloop when gevent is detected
- Added support for CIRCUSCTL_ENDPOINT environment variable to circusctl - #396
- util: fix bug in to_uid function - #397
- Remove handler on ioloop error - #398.

- Improved test coverage
- Deprecated the 'service' option for the ResourceWatcher plugin - #404
- removed psutil.error usage
- Added UDP discovery in circusd - #407
- Now allowing globs at arbitrary directory levels - #388
- Added the 'statd' configuration option - #408
- Add pidfile, logoutput and loglevel option to circus configuration file - #379
- Added a tutorial in the docs.
- make sure we're merging all sections when using include - #414
- added pipe_stdout, pipe_stderr, close_child_stderr & close_child_stdout options to the Process class
- added close_child_stderr & close_child_stdout options to the watcher

0.7.1 - 2013-05-02

- Fixed the respawn option - #382
- Make sure we use an int for the timeout - #380
- display the unix sockets as well - #381
- Make sure it works with the latest pyzmq
- introduced a second syntax for the fd notation

0.7 - 2013-04-08

- Fix get_arbiter example to use a dict for the watchers argument. #304
- Add some troubleshooting documentation #323
- Add python buildout support
- Removed the gevent and the thread redirectors. now using the ioloop - fixes #346. Relates #340
- circus.web is now its own project
- removed the pyzmq patching
- Allow the watcher to be configured but not started #283
- Add an option to load a virtualenv site dir
- added on_demand watchers
- added doc about nginx+websockets #371
- now properly parsing the options list of each command #369
- Fixed circusd-stats events handling #372
- fixed the overflow issue in circus-top #378
- many more things...

0.6 - 2012-12-18

- Patching protocols name for sockets - #248
- Don't autoscale graphs. #240
- circusctl: add per command help, from docstrings #217
- Added workers hooks
- Added Debian package - #227
- Added Redis, HTTP Observer, Full stats & Resource plugins
- Now processes can have titles
- Added autocompletion
- Added process/watcher age in the webui
- Added SSH tunnel support
- Now using pyzmq.green
- Added upstart script & Varnish doc
- Added environment variables & sections
- Added unix sockets support
- Added the *respawn* option to have single-run watchers
- Now using tox in the tests
- Allow socket substitution in args
- New doc theme
- New rotation options for streams: max_bytes/backup_count

0.5.2 - 2012-07-26

- now patching the thread module from the stdlib to avoid some Python bugs - #203
- better looking circusctl help screen
- uses pustil get_nice() when available (nice was deprecated) - #208
- added max_age support - #221
- only call listen() on SOCK_STREAM or SOCK_SEQPACKET sockets
- make sure the controller empties the plugins list in update_watchers() - #220
- added –log-level and –log-output to circushttpd
- fix the process killing via the web UI - #219
- now circus is zc.buildout compatible for scripts.
- cleanup the websocket when the client disconnect - #225
- fixed the default value for the endpoint - #199
- splitted circushttpd in logical modules

0.5.1 - 2012-07-11

- Fixed a bunch of typos in the documentation
- Added the debug option
- Package web-requirements.txt properly
- Added a errno error code in the messages - fixes #111

0.5 - 2012-07-06

- added socket support
- added a listsocket command
- sockets have stats too !
- fixed a lot of small bugs
- removed the wid - now using pid everywhere
- faster tests
- changed the variables syntax
- use pyzmq's ioloop in more places
- now using iowait for all select() calls
- incr/decr commands now have an nbprocess parameter
- Add a reproduce_env option to watchers
- Add a new UNEXISTING status to the processes
- Added the global *httpd* option to run circushttpd as a watcher

0.4 - 2012-06-12

- Added a plugin system
- Added a "singleton" option for watchers
- Fixed circus-top screen flickering
- Removed threads from circus.stats in favor of zmq periodic callbacks
- Enhanced the documentation
- Circus client now have a send_message api
- The flapping feature is now a plugin
- Every command line tool have a –version option
- Added a statsd plugin (sends the events from circus to statsd)
- The web UI now uses websockets (via socketio) to get the stats
- The web UI now uses sessions for "flash messages" in the web ui

0.3.4 - 2012-05-30

- Fixed a race condition that prevented the controller to cleanly reap finished processes.
- Now check_flapping can be controlled in the configuration. And activated/deactivated per watcher.

0.3.3 - 2012-05-29

- Fixed the regression on the uid handling

0.3.2 - 2012-05-24

- allows optional args property to add_watcher command.
- added circushttpd, circus-top and circusd-stats
- allowing Arbiter.add_watcher() to set all Watcher option
- make sure the redirectors are re-created on restarts

0.3.1 - 2012-04-18

- fix: make sure watcher' defaults aren't overrided
- added a StdoutStream class.

0.3 - 2012-04-18

- added the streaming feature
- now displaying coverage in the Sphinx doc
- fixed the way the processes are killed (no more SIGQUIT)
- the configuration has been factored out
- setproctitle support

0.2 - 2012-04-04

- Removed the *show* name. replaced by *watcher*.
- Added support for setting process **rlimit**.
- Added support for include dirs in the config file.
- Fixed a couple of leaking file descriptors.
- Fixed a core dump in the flapping
- Doc improvments
- Make sure circusd errors properly when another circusd is running on the same socket.
- get_arbiter now accepts several watchers.
- Fixed the cmd vs args vs executable in the process init.
- Fixed –start on circusctl add

0.1 - 2012-03-20

- initial release

2.3.10 Glossary: Circus-specific terms

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.

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.

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

worker, workers, process, 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.

2.3.11 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.

Licence

Contributors

See the full list at https://github.com/mozilla-services/circus/blob/master/CONTRIBUTORS.txt