lighthouse Documentation

Release 1.0.0

William Glass

February 09, 2016

Contents

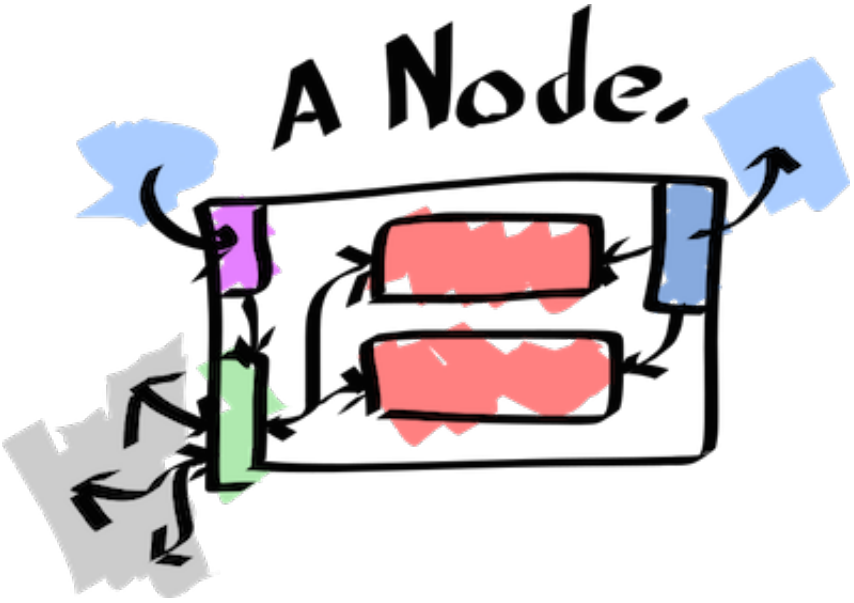
Lighthouse is a service node discovery system written in python, built with resilience, flexibility and ease-of-use in mind and inspired by Airbnb's SmartStack solution. Out of the box it supports discovery via Zookeeper with cluster load balancing handled by an automatically configured HAProxy.

To jump right in see the Getting Started page, or if you'd like to see it in action check out the Examples page.

CHAPTER 1

Overview

A lighthouse setup consists of three parts running locally on each node: a load balancer, the `lighthouse-writer` script and (usually) the `lighthouse-reporter` script.



In a Lighthouse setup, no node's application code is aware of the existence of other nodes, they talk to a local port handled by an instance of the load balancer which in turn routes traffic among the various known other nodes.

This local load balancer is automatically updated when nodes come and go via the `lighthouse-writer` script, which talks to the discovery method (e.g. Zookeeper) to keep track of which nodes on which clusters are up.

The `lighthouse-reporter` script likewise talks to the discovery method, it is responsible for running health checks on any services on the local node and reports to the discovery method that the healthy services are up and the unhealthy ones are down.

CHAPTER 2

Development

The code is hosted on GitHub

To file a bug or possible enhancement see the Issue Tracker, also found on GitHub.

CHAPTER 3

License

Lighthouse is licensed under the terms of the Apache license (2.0). See the LICENSE file for more details.

3.1 Getting Started

3.1.1 Installation

Automatic

Lighthouse is available via PyPI, installation is as easy as:

```
pip install lighthouse
```

Note that when installed this way the example files found in the source repo are not included. If you wish to use the examples, a manual install via the current source tarball is your best choice.

Manual

First download the current tarball at lighthouse-1.0.0.tar.gz, then:

```
tar -zxvf lighthouse-1.0.0.tar.gz
cd lighthouse-1.0.0
python setup.py install
```

3.1.2 Prerequisites

Python verison: Lighthouse runs on python versions 2.6 and greater, but is better vetted on 2.7 and 3.4 specifically. Versions 2.6 and PyPy are included in the test suite but are less rigorously tested manually.

Required libraries: By default the lighthouse installation depends on

- Watchdog for monitoring changes to config files
- PyYAML to parse the config files
- Kazoo to communicate with Zookeeper
- Six to maintain python 2 and 3 compatibility

HAProxy: As of right now only HAProxy version 1.4 or higher, 1.3 *might* work but is untested.

Platforms: Lighthouse is most extensively tested on Linux and Mac OSX but should run just fine on any Unix-y/POSIX platform. Native windows use is unsupported as UNIX sockets are required to control the load balancer, but a setup with cygwin is theoretically possible.

3.1.3 Optional Extras

Redis plugins

Lighthouse includes a "redis" extra package that comes with a health check for redis services. To install an extra, use square brackets when installing lighthouse:

```
pip install lighthouse[redis]
```

3.1.4 Examples

At this point you should be ready to run the examples if you've downloaded them. Simply run the `start.sh` script for the target example and then run `lighthouse-writer` and `lighthouse-reporter` passing in the path to the example directory. For more details on the included examples see Examples.

3.1.5 Configuration

The next step will of course be customizing your own Configuration.

3.2 Configuration

The lighthouse scripts are configured by passing in a root config directory which contains individual YAML config files and follows a certain layout:

```
<config dir>/
  |____logging.yaml
  |____balancers/
  |       |____haproxy.yaml
  |____discovery/
  |       |____zookeeper.yaml
  |____clusters/
  |       |____webcache.yaml
  |       |____pg-db.yaml
  |       |____users-api.yaml
  |____services/
  |       |____a_service.yaml
  |       |____other_service.yaml
```

There are five types of config file:

- **logging**:

  This file lives at the root of the config directory and its contents are passed to the standard lib `logging.config` module's `dictConfig` function.

  Configuring Logging

- **balancer**:

  Files that configure the locally-running load balancer(s). These live in the `balancers` subdirectory. The project includes a plugin for HAProxy as a balancer.

  Configuring HAProxy

- **discovery**:

  Discovery config files live in a `discovery` subdirectory, each file configures a single discovery method with a name matching the filename. The project includes a plugin for Zookeeper as a discovery method.

  Configuring Zookeeper

- **cluster**:

  Cluster config files are found under the `clusters` subdirectory and denote services used by the local machine/node that should be watched for member node updates.

  Configuring Clusters

- **service**:

  Each config file under the `services` subdirectory represents a local service to be reported as up or down via the discovery method(s). These files include configurations for a service's health checks as well. The project includes simple HTTP and Redis health checks.

  Configuring Services

Note: Service vs. Cluster terminology: Think of a "service" as used in this documentation as describing an individual service *provided* by the local node/machine, a "cluster" as a description of a service *consumed* by the local node/machine.

3.2.1 Configuring Logging

The content of the logging config file is passed along to the standard `logging` lib's dictConfig method. Lighthouse does not verify the configuration itself, but the contents should conform to the `dict config schema` since that's what the `logging` system expects.

Lighthouse *does* however provide two helper classes for logging: the *CLIHandler* and the *ContextFilter*.

As an example, this file sends logs to stdout with the `CLIHandler` attached, as well as to the local syslog system with added "program" context via the `ContextFilter`:

`logging.yaml`

```
version: 1
disable_existing_loggers: False
filters:
  context:
    "()": lighthouse.log.ContextFilter
formatters:
  syslog:
    format: 'lighthouse: [%(program)s] [%(threadName)s] %(message)s'
handlers:
  cli:
    class: 'lighthouse.log.cli.CLIHandler'
    stream: "ext://sys.stdout"
  syslog:
    class: 'logging.handlers.SysLogHandler'
```

```
    address: '/dev/log'
    facility: "local6"
    filters: ["context"]
    formatter: 'syslog'
root:
  handlers: ['syslog', 'cli']
  level: "DEBUG"
  propagate: true
```

ContextFilter

A simple `logging.Filter` subclass that adds a "program" attribute to any `LogRecord`s that pass through it. For the `lighthouse-writer` script the attribute is set to "WRITER", for `lighthouse-reporter` it is set to "REPORTER".

Useful for differentiating log lines between the two scripts.

CLIHandler

Handy `logging.StreamHandler` subclass that colors the log lines based on the thread the log originated from and the level (e.g. "info", "warning", debug", etc.)

Some example lines:

```
[2015-09-22 18:52:40 I][MainThread] Adding loggingconfig: 'logging'
[2015-09-22 18:52:40 D][MainThread] File created: /Users/william/local-config/balancers/haproxy.yml
[2015-09-22 18:52:40 I][MainThread] Adding balancer: 'haproxy'
[2015-09-22 18:52:40 I][Thread-3] Updating HAProxy config file.
[2015-09-22 18:52:40 D][MainThread] File created: /Users/william/local-config/discovery/zookeeper.yar
[2015-09-22 18:52:40 D][Thread-3] Got HAProxy version: (1, 5, 10)
[2015-09-22 18:52:41 I][MainThread] Adding discovery: 'zookeeper'
[2015-09-22 18:52:41 D][Thread-3] Got HAProxy version: (1, 5, 10)
[2015-09-22 18:52:41 I][Thread-12] Connecting to zookeeper02.oregon.internal:2181
[2015-09-22 18:52:41 I][Thread-7] Updating HAProxy config file.
[2015-09-22 18:52:41 D][MainThread] File created: /Users/william/local-config/clusters/haproxy-web.ya
[2015-09-22 18:52:41 I][Thread-3] Gracefully restarted HAProxy.
[2015-09-22 18:52:41 I][MainThread] Adding cluster: 'haproxy-web'
```

3.2.2 Configuring HAProxy

The configuration of HAProxy is one of the more complicated (read: flexible!) parts of the Lighthouse system. Just about any setup can be accommodated, it's helpful to have an HAProxy config reference on hand.

Let's start with an example:

`haproxy.yaml`

```
config_file: "/etc/haproxy.cfg"
socket_file: "/var/run/haproxy.sock"
pid_file: "/var/run/haproxy.pid"
bind_address: "127.0.0.1"
global:
  - "daemon"
  - "maxconn 40000"
  - "user haproxy"
  - "log /var/run/syslog local2"
```

```
defaults:
  - "balance roundrobin"
  - "timeout connect 10s"
  - "timeout client 20s"
  - "timeout server 20s"
stats:
  port: 9009
  uri: "/haproxy"
  timeouts:
    connect: 4000
    server: 30000
```

The only *required* configuration settings are the `config_file`, `socket_file`, and `pid_file` but such a bare-bones setup is probably not what you want. The main points of configuration will be the `global` and `defaults` settings, where you can list any HAProxy config directives that will go under those respective stanzas in the generated config file.

Note: If HAProxy is not currently running when `lighthouse-writer` tries to restart it, HAProxy will be started automatically.

Proxies

Sometimes it can be useful to list straight-up proxies in the generated HAProxy configuration. For example, if you have a 3rd-party partner API you talk to on a whitelisted IP basis you would want a dedicated proxy machine with a known IP that listens on a port and proxies to the business partner.

To facilitate such a use-case the HAProxy YAML config supports a `proxies` setting. Each entry in the mapping under `proxies` is a separate named proxy with certain *settings requirements* themselves.

For example:

`haproxy.yaml`

```
config_file: "/etc/haproxy.cfg"
socket_file: "/var/run/haproxy.sock"
pid_file: "/var/run/haproxy.pid"
bind_address: "0.0.0.0"
global:
  - "daemon"
  - "user haproxy"
proxies:
  business_partner:
      port: 1100
      upstreams:
        - host: "b2b.partner.com"
          port: 88
          max_conn: 400
      options:
        - "mode http"
```

This config sets up a "business_partner" proxy that takes traffic from the local port 1100 and forwards it to a partner server on port 88.

Peers

A new feature available in HAProxy 1.5 and newer is the concept of peers.

When a node is reported as up and available, information about the HAProxy instance that lives on the node is included along with it. This allows the config file generator to list the peers of each cluster, allowing HAProxy to coordinate cluster-wide statistics in what's known as "stick tables".

Note: This feature is automatically used and only available in HAProxy 1.5 and newer.

Stats Listener

HAProxy comes with a built-in feature for serving up a status page with all sorts of useful information. Each known backend and frontend is listed along with their statuses and usage statistics (see the live demo on haproxy.org for an example).

To enable the feature for *your* HAProxy instance, include the `stats` setting in your YAML config. A port to use for serving the page is required, check the *stats settings* section for more detailed info.

Settings

- **config_file** *(required)*:

  This is the path of the HAProxy config file that will be automatically generated by Lighthouse.

- **socket_file** *(required)*:

  The path to the UNIX socket file Lighthouse should use to communicate with HAProxy.

- **pid_file** *(required)*:

  The path to the PID file for HAProxy.

- **global**:

  Optional list of directives to put under the "global" stanza in the generated HAProxy config file.

- **defaults**:

  Optional list of directives to put under the "defaults" stanza in the generated HAProxy config file.

- **bind_address**:

  The address to bind to for the various ports HAProxy will listen on. Default is "localhost".

- **meta_cluster_ports**:

  A mapping of meta cluster name to a port. This tells HAProxy to bind to that port to handle traffic for the meta cluster.

- **proxies**:

  Optional setting section for configuring simple proxies. Each of the proxy entries have their own settings requirements, see *Proxies Settings* below.

- **stats**:

  Optional but recommended feature for having HAProxy serve a simple web page with status and metrics info (see the live demo on haproxy.org for an example). This setting has further required settings that are listed below.

Proxies Settings

- **port** *(required)*:

  The local port to bind to and listen for traffic to proxy on.

- **upstreams** *(required)*:

  List of servers to proxy traffic to. If multiple servers are listed they're balanced with a round robin algorithm.

- **bind_address**:

  Optional setting for the address to use when binding the local port. Defaults to "localhost".

- **options**:

  A list of extra directive lines to include in the generated "listen" stanza for the proxy.

Stats Settings

- **port** *(required)*:

  The local port to bind to and serve up the stats page with.

- **uri**:

  Optional uri path for the page. For example if the `port` is set to 9009 and the uri set to "/haproxy_stats", the HAProxy stats page would be available at `http://<machine address>:9009/haproxy_stats`.

- **timeouts**:

  Optional timeouts. These are a mapping from timeout name to value, the only names recognized are `connect`, `client` and `server`.

3.2.3 Configuring Zookeeper

The Zookeeper discovery method config is incredibly simple, there are two settings and they are both required.

Settings

- **hosts** *(required)*:

  A list of host strings. Each host string should include the hostname and port, separated by a colon (":").

- **path** *(required)*:

  A string setting denoting the base path to use when looking up or reporting on node availability. For example, a path of `/lighthouse/services` would mean that any services available would be found at the path `/lighthouse/services/service_name`.

Warning: Altering the "path" setting is doable, but should be avoided if at all possible. Whatever provisioning method is used to update the `zookeeper.yaml` file is almost certainly going to leave many nodes out of sync at least for a time. A situation where nodes don't agree on where to look for each other is indistinguishable from a large network outage.

Example

A simple example with a three-member zookeeper cluster and a base path:

`discovery/zookeeper.yaml`

```
hosts:
  - "zk01:2181"
  - "zk02:2181"
  - "zk03:2181"
path: "/lighthouse/services"
```

3.2.4 Configuring Clusters

Cluster configs are very simple, all that's needed is a discovery method defined by the key `discovery` and a section specific to the load balancer in use.

A simple web server example:

`clusters/webapp.yaml`

```
discovery: "zookeeper"
haproxy:
  port: 8000
  frontend:
    - "log global"
  backend:
    - "mode http"
```

In this example we're using the Zookeeper discovery method and the HAProxy load balancer. The balancer should listen locally on port 8000 and the HAProxy frontend definition should include the `log global` directive and the backend definition should include the `mode http` directive.

Meta-Clusters

In some use-cases a service might actually be composed of several clusters, with special rules for routing between them. For example, a RESTful api that routes based on URL where `/api/widgets` hits the "widgets" cluster and `/api/sprockets` hits the "sprockets" cluster.

To do this, the widget and sprocket cluster configs would use the `meta_cluster` setting and provide the "ACL" rule for how they're routed.

`clusters/widgets.yaml`

```
discovery: "zookeeper"
meta_cluster: "webapi"
haproxy:
  acl: "path_beg /api/widgets"
  backend:
    - "mode http"
```

`clusters/sprockets.yaml`

```
discovery: "zookeeper"
meta_cluster: "webapi"
haproxy:
  acl: "path_beg /api/sprockets"
  backend:
```

```
    - "mode http"
    - "maxconn 500"  # maybe the sprockets cluster is on limited hardware
```

You'll note that neither of these actually list which port for the load balancer to listen on. Rather than have each cluster config list a port and hope they match, we set the port via the `meta_clusters` setting in the load balancer config.

`haproxy.yaml`

```
config_file: "/etc/haproxy.cfg"
socket_file: "/var/run/haproxy.sock"
pid_file: "/var/run/haproxy.pid"
meta_clusters:
  webapi:
    port: 8888
    frontend:
      - "mode http"
```

This will tell HAProxy to listen on port 8888 locally and serve up the meta-service, where requests to `/api/widgets` hit the widgets cluster and requests to `/api/sprockets` get routed to an independent sprockets cluster.

Note that it also adds the "mode http" directive to the meta cluster's frontend definition, a requirement for "path_beg" ACLs. The "frontend" portion of a `meta_clusters` is a list of any frontend directives that should be added to the meta cluster's stanza.

Settings

- **discovery** *(required)*:

  The name of the discovery method to use for determining available nodes.

- **meta_cluster**:

  Name of the "meta cluster" this cluster belongs to. Care must be taken such that the meta cluster has a port set in the load balancer config file.

HAProxy Settings

The following settings are available for the `haproxy` setting of a cluster.

- **port**:

  Specifies which port the local load balancer should bind to for communicating to the cluster. Not applicable to meta-clusters.

- **acl**:

  Defines the ACL routing rule for a cluster who is a member of a meta-cluster. Not applicable to regular non-meta clusters.

- **frontend**:

  Custom HAProxy config lines for the frontend stanza generated for the cluster. Lines are validated to make sure the directive is a legal one for a frontend stanza but other than that anything goes.

- **backend**:

  Custom HAProxy config lines for the backend stanza generated for the cluster. Lines are validated to make sure the directive is a legal one for a backend stanza but other than that anything goes.

- **server_options**:

  Extra options to add to a node's `server` directive within a backend stanza. (e.g. `slowstart` if nodes in the cluster should have their traffic share ramped up gradually)

3.2.5 Configuring Services

Service configs have more required settings than most other configurable items, but are still fairly easy to define. Each service config must define the port to use for communicating with the service, as well as the discovery method used for reporting and the health checks to use to determine the service's availability.

For example, a simple redis cache service:

`services/cache.yaml`

```
port: 6379
discovery: "zookeeper"
host: "127.0.0.1"
checks:
  interval: 3
  redis:
    rise: 2
    fall: 1
```

This service runs on the default redis port of 6379, uses the Zookeeper discovery method and the redis health check. The check is performed every three seconds, the check would have to pass twice for the service to be considered "up" and fail only once to be considered "down".

Settings

- **port/ports** *(required)*:

  Port(s) that the local service is listening on. If listing multiple ports, the `ports` setting must be used. For single-port services either `port` or `ports` (with a single entry) will do.

- **discovery** *(required)*:

  Discovery method to use when reporting the local node's service(s) as up or down.

- **checks** *(required)*:

  A list of health checks to perform, which will determine if a service is up or not.

- **host**:

  Optional hostname to use when communicating with the service. Usually "localhost" or "0.0.0.0", defaults to "127.0.0.1" if not specified.

- **metadata**:

  An optional mapping of data to send along when reporting the service as up and available. Useful for providing extra context about a node for use in a balancer plugin (e.g. denoting a "master" or "slave" node).

Health Check Settings

- **interval** *(required)*:

  The time (in seconds) to wait between each health check. This setting belongs under the "checks" setting.

- **rise** *(required)*:

  The number of successful health checks that must happen in a row for the service to be considered "up". This setting belongs under individual health check configs.

- **fall** *(required)*:

  The number of failed health checks that must happen in a row for the service to be considered "down". This setting belongs under individual health check configs.

Included Health Checks

The Lighthouse project comes bundled with a handful of health checks by default, including two basic ones for HTTP-based services and lower-level TCP services.

HTTP

The HTTP health check performs a simple request to a given uri and passes if the response code is in the 2XX range. The HTTP health check has no extra dependencies but does have a required extra setting:

- **uri** *(required)*:

  The uri to hit with an HTTP request to perform the check (e.g. "/health")

TCP

The TCP health check can be used for services that don't use HTTP to communicate (e.g. redis, kafka, etc.). The health check is configured to have a "query" message sent to the service and an expected "response".

- **query** *(required)*:

  The message to send to the port via TCP (e.g. Zookeeper's "ruok")

- **response** *(required)*

  Expected response from the service. If the service responds with a different message or an error happens during the process the check will fail.

Optional Health Checks

Redis

Sends the "PING" command to the redis instance and passes if the proper "PONG" response is received. The Redis health check plugin has no extra config settings. This optional plugin requires Lighthouse to be installed with the "redis" extra:

```
pip install lighthouse[redis]
```

3.3 Examples

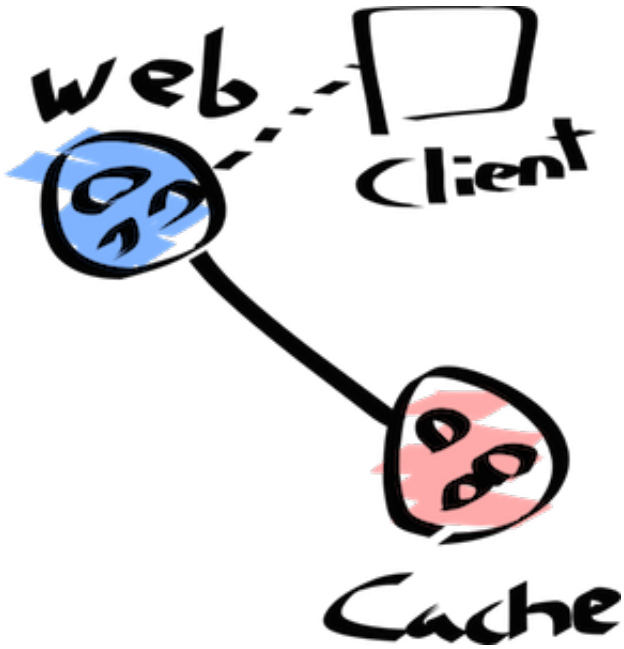
Example use-cases of lighthouse live in the `examples` directory. There are a handful of Docker images with various lighthouse setups that can be launched to create small clusters.

Each example also makes use of a "client" container that consumes the resulting services and exposes requisite ports that can be hit to show off just how the clusters handle traffic.

3.3.1 Examples List

Simple Web Cluster Example

In this example we'll construct a simple system with two clusters: a webapp cluster serving up some basic content and a cache redis cluster used in creating said content.



Creating the cache cluster

To start off with we'll launch a couple cache nodes to create the redis cluster:

```
$ ./launch.sh cache cache01

$ ./launch.sh cache cache02
```

Two should be fine for our purposes.

> **Warning:** These redis instances are independent, if a request ends up using a different cache than a previous one the results will be inconsistent! This is OK here since this is a simple example but in the real world you'll need to be mindful of how requests are routed to clusters that keep state.

Creating the web cluster

Spinning up a webapp node is a simple matter:

```
$ ./launch.sh webapp app01
```

In this part of the example we'll show off a particular feature of lighthouse: handling multiple instances of the same service on a single host. To bring up such a node:

```
$ ./launch.sh multiapp app02
```

This `multiapp` container will have two instances of the webapp process running, each on different ports but reporting as part of the same cluster.

With these two launched you should see three entries in the "webapp" section of the client container's HAProxy web interface:

webapp

	Queue			Session rate			Sessions					Bytes	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out
app01:8000	0	0	-	0	0		0	0	-	0	0	0	0
app02:8002	0	0	-	0	0		0	0	-	0	0	0	0
app02:8001	0	0	-	0	0		0	0	-	0	0	0	0
Backend	0	0		0	0		0	0	0	0	0	0	0

Two hosts, three nodes.

Sending traffic

Now that our clusters are up and discovered it's time to send traffic to them. First off we need to know which port the client image's "8000" port mapped to. This can be done with a simple `docker ps` command:

```
$ docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED
6d6db2e1842e        lighthouse.examples.client:latest      "/bin/sh -c 'supervi   13 minutes ago
82618a6a2fef        lighthouse.examples.zk:latest          "/opt/zookeeper/bin/   28 hours ago
...
```

Under the "PORTS section we find "0.0.0.0:33272->8080/tcp", so in this example the mapped port is "33274".

So a curl to `http://<docker host ip>:33274/` would yield:

```
<h1>Current count: 1</h1>
```

With each subsequent request the counter will update, and HAProxy will balance the requests among the three webapp nodes.

Going further

This example showed how a very basic set of clusters can be set up, but it doesn't have to end there! Try:

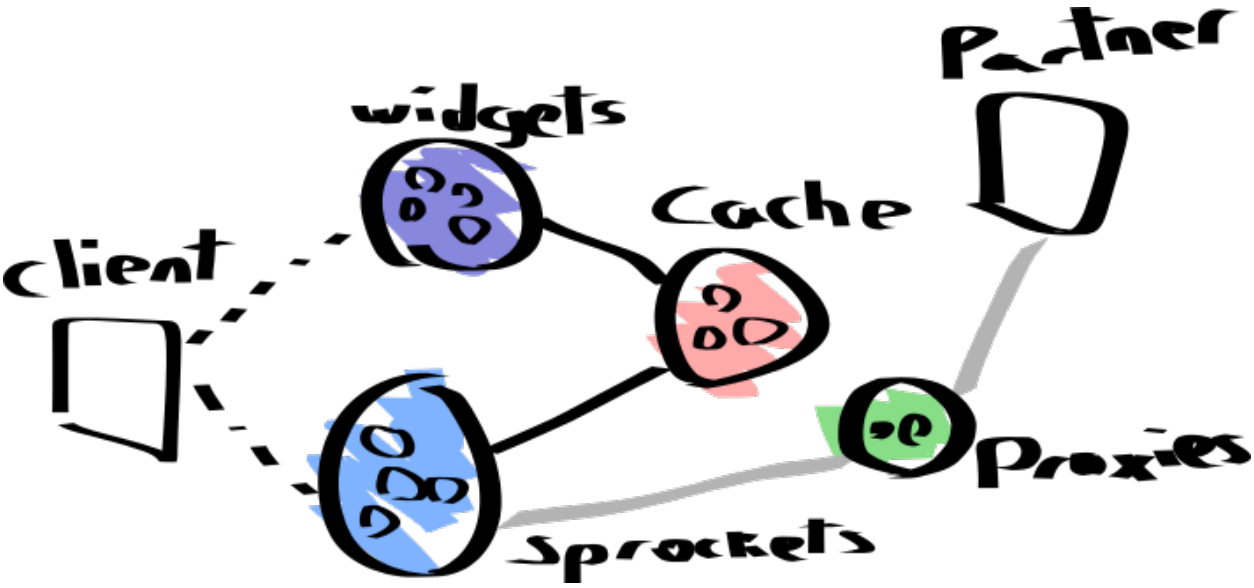
- killing and spinning up nodes to each cluster and watch the HAProxy web interface to see how it reacts
- taking a node down while blasting the cluster with traffic via tools like ApacheBench
- removing all nodes from the cache cluster while watching the reporting logs from the webapp nodes

API Meta-Cluster Example

This example will demonstrate ACL-based routing where a single API is serviced by multiple clusters, as well as the proxies feature of the HAProxy balancer plugin.

We'll have a redis-backed web API with two endpoints: one for "widgets" and one for "sprockets". Each endpoint will be served by a separate independent cluster of webapp nodes.

The "sprockets" cluster will also communicate with a "partner" machine via a cluster of proxy nodes. Regardless of how many nodes there are in the sprockets cluster and which nodes come and go, the only nodes to talk to the partner are the proxy nodes.



Creating the partner "machine"

To start off with we'll launch a single instance of a "partner" container meant to represent a 3rd-party API:

```
$ ./launch.sh partner partnerapi
```

Naming the container `partnerapi` is important, the configuration on the proxy cluster nodes will assume the "partner" is reachable via that name.

Note: This "external" container is intentionally limited. It doesn't make use lighthouse at all, and is only reachable by name from "proxy" nodes.

Creating the proxy cluster

The proxy cluster will be limited in numbers at first since in such a real-life scenario the 3rd party partner will whitelist only certain IPs:

```
$ ./launch.sh proxy proxy01
```

Naturally as this is just an example the cluster can be expanded to your heart's content.

Proxy nodes don't run any extra services themselves, rather they configure their HAProxy instances to proxy to the `partnerapi` machine. If you connect to proxy01 and look at the `/etc/haproxy.cfg` file you should see something along the lines of:

```
listen business_partner
      bind :7777
      mode http
      server partnerapi:88 partnerapi:88 maxconn 400
```

Creating the clusters

To start off with we'll create two nodes for each of the cache, widgets and sprockets clusters:

```
$ ./launch.sh cache cache01
$ ./launch.sh cache cache02
$ ./launch.sh widgets widgets01
$ ./launch.sh widgets widgets02
$ ./launch.sh sprockets sprockets01
$ ./launch.sh sprockets sprockets02
```

Once these containers are started you should see the widgets/sprockets nodes show up in the HAProxy web interface of the client node:

api_sprockets

	Queue			Session rate			Sessions					Bytes	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out
sprockets01:8000	0	0	-	0	0		0	0	-	0	0	0	
sprockets02:8000	0	0	-	0	0		0	0	-	0	0	0	
Backend	0	0		0	0		0	0	0	0	0	0	

webapp

	Queue			Session rate			Sessions					Bytes		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req
Backend	0	0		0	0		0	0	0	0	0	0	0	

api_widgets

	Queue			Session rate			Sessions					Bytes	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out
widgets02:8000	0	0	-	0	0		0	0	-	0	0	0	0
widgets01:8000	0	0	-	0	0		0	0	-	0	0	0	0
Backend	0	0		0	0		0	0	0	0	0	0	0

The widgets API

The widgets API has one endpoint, "/api/widgets" that responds to both GET and POST requests.

A GET request shows a mapping of known widgets to their count, empty at first:

```
$ curl http://<docker_ip>:<port>/api/widgets
{
  "widgets": {}
}
```

A POST to the endpoint requires a "widget" parameter set to any sort of string:

```
$ curl -XPOST -d "widget=foo" http://<docker_ip>:<port>/api/widgets
{
  "success": true
}
```

With that "foo" widget added we can see the updated count:

```
$ curl http://<docker_ip>:<port>/api/widgets
{
  "widgets": {
    "foo": 1
```

```
  }
}
```

After a few GET and POST requests, you can check the HAProxy web interface on the client and see the traffic being balanced on the "api_widgets" backend.

The sprockets API

The sprockets API is similar to widgets, it has a single endpoint that responds to both GET and POST requests but sprockets are shown as a set rather than a mapping.

However, the sprockets API also talks to the "partner" API via the proxy cluster. Each response includes a "token" grabbed from the partner machine.

GET requests will show the set:

```
$ curl http://<docker_ip>:<port>/api/sprockets
{
  "token": "8c53bb14-92ad-4722-aa07-181aeddcfb94",
  "sprockets" []
}
```

POST requests require a "sprocket" parameter and will add a new sprocket to the set:

```
$ curl -XPOST -d"sprocket=bar" http://<docker_ip>:<port>/api/sprockets
{
  "success": true,
  "token": "76a11362-d26d-496f-b981-ba864aa68877"
}
$ curl http://<docker_ip>:<port>/api/sprockets
{
  "token": "d7ee21c7-3a6f-4fc2-a1fe-0d62321bba4e",
  "sprockets" [
    "bar"
  ]
}
```

And there you have it! A series of horizontally scalable clusters that communicates with an "external" service, proxied in such a way that the external service only sees one machine talking to it.

3.3.2 Setting Up

To start with you'll need Docker set up properly. How to do that depends on your OS and is beyond the scope of this documentation but luckily the folks at Docker provide some of their own.

Once you have docker up and running, creating the example images is as simple as:

```
make
```

This shouldn't take *too* long, and once it's done you should have a handful of example docker images with names starting with "lighthouse.examples":

```
[examples] $ docker images

REPOSITORY                      TAG                 IMAGE ID            CREATED             VIRTUAL S
lighthouse.examples.client      latest              1bc85bf377a7        52 minutes ago      436.4 MB
lighthouse.examples.sprockets   latest              e8674c42d7bc        52 minutes ago      451.2 MB
```

```
lighthouse.examples.widgets     latest              f449a383522b        52 minutes ago      451.2 MB
lighthouse.examples.multiapp    latest              684fb9de9298        52 minutes ago      451.2 MB
lighthouse.examples.webapp      latest              7ab84d42ddcd        52 minutes ago      451.2 MB
lighthouse.examples.cache       latest              464a3b360d4d        52 minutes ago      449 MB
lighthouse.examples.base        latest              d990927b27e4        54 minutes ago      434.4 MB
lighthouse.examples.zk          latest              c31155053d47        2 days ago          342.7 MB
...
```

3.3.3 First Steps

There are some common components to each example that should be set up first, namely a client container and the Zookeeper discovery method.

Launching Zookeeper

The zookeeper host in the cluster is expected to be named `zk01` and use the standard ports, so it can be launched with:

```
$ docker run --name zk01 -d lighthouse.examples.zk
```

Launching a Client

Launching a client container is a simple matter of using the included `launch.sh` helper script found in the `examples` directory:

```
$ ./launch.sh client client
```

Details about the script can be found in the *launching* section.

3.3.4 Individual Nodes

Launching

Launching a new node can be done by hand via docker, but the `examples` directory includes a handy `launch.sh` script to make things easier:

```
$ ./launch.sh <type> <name>
```

Where the "<type>" matches the end of the example docker image name. For example the "lighthouse.examples.webapp" image's node type is "webapp".

The "<name>" portion is any host name you want to give to the node. Since this is an SOA and nodes are (hopefully) automatically discovered the name doesn't really matter and is mostly for convenience.

Examining

Each node container exposes two web interface ports for examining what exactly is going on: one for HAProxy and one for Supervisord, the process management tool used to run multiple processes at once in a single container. The HAProxy web interface listens on port 9009 with the URI "/haproxy", the supervisord web interface listens on port 9000.

To avoid conflicting port assignments, a container will map these ports to a random available one on the docker host. To see the resulting mapped port you'll have to run `docker ps`:

```
$ docker ps

CONTAINER ID        IMAGE                                 COMMAND                CREATED                 STA
aa037622260e        lighthouse.examples.cache:latest      "/bin/sh -c 'supervi   3 seconds ago           Up
85890f46dc8f        lighthouse.examples.zk:latest         "/opt/zookeeper/bin/   17 seconds ago          Up
```

In this example, the "cache01" container's HAProxy web interface can be accessed via `http://<docker_host_ip>:32771/haproxy` and the supervisord web interface via `http://<docker_host_ip>:32770`.

Connecting

Along with the `launch.sh` script there's also a handy `connect.sh` script:

```
$ ./connect.sh <name>
```

This will attach to the container with the "<name>" name and run an interactive bash session, helpful for examining log and configuration files by hand. Note that the TERM environment variable is not set by default so many things like `less` and `clear` might not work quite right unless it's set by hand.

For each node the lighthouse scripts run in debug mode and log quite a bit. The log files for the lighthouse script live under `/var/log/supervisor/lighthouse` in the container. The services served up by containers will generally put their logs under `/var/log/supervisor/`.

The HAProxy config is written to `/etc/haproxy.cfg`.

Stopping

Unlike launching or connecting, there is no helper script as a simple docker command does the job:

```
$ docker rm -f <name>
```

This will halt the node container and unregister it from zookeeper automatically.

3.4 Writing Plugins

Lighthouse relies on a plugin system for it's functionality. All load balancers, discovery methods and health checks are plugins, even the included ones!

All that's required for creating a new plugin is to subclass the proper base class and add that subclass to the proper entry point in your project's setup.

For example a new health check called `mycheck` might have a class called `MyCheck`, a subclass of *`lighthouse.check.Check`* and added to a `setup.py`'s setup() call:

`setup.py`

```
from setuptools import setup

setup(
    # basics...
    install_requires=[
      # dependencies for your plugin go here
    ],
    entry_points={
        "lighthouse.checks": [
```

```
            mycheck = myproject.MyCheck
        ]
    }
)
```

Each of the three plugin types have their own entrypoint. For details of each of the three plugin types see their individual documentation:

- Writing Health Check Plugins
- Writing Discovery Method Plugins
- Writing Load Balancer Plugins

3.4.1 Writing Health Check Plugins

Health check plugins are the easiest of the three plugin types to write and are liable to be the most common. Writing a new health check plugin is a simple matter of creating a *lighthouse.check.Check* subclass and exposing it via the `lighthouse.checks` entry point.

The key part of a health check plugin is the `perform()` method on the subclass, it is where the actual checking takes place. It's important that this method take no arguments and returns `True` or `False` based on the check's success.

Examples

For an example of a check that has no external dependencies but uses custom attributes and configuration, see the *lighthouse.checks.http.HTTPCheck* class included in the source distribution.

Likewise, for an example of a simple health check that involves external dependencies see the `lighthouse.checks.redis.RedisCheck` class.

Required Methods

- `validate_dependencies(cls)` *(classmethod)*:

  This classmethod should check that all required external dependencies for your health check are met.

  If the requirements are met, this method should return True. If not met it should return False.

- `validate_check_config(cls, config)` *(classmethod)*:

  The "config" argument to this classmethod is the dictionary representation of the health check's portion of the service YAML configuration, this method should validate any plugin-specific bits of that configuration. The base *lighthouse.check.Check* class automatically validates that the standard `host`, `port`, `rise` and `fall` values are present.

  If any parts of the configuration are invalid, a `ValueError` exception should be raised.

- `apply_check_config(self, config)`:

  This instance method's config argument is also the dictionary config of the health check's portion of a service's YAML config file, albeit one that has already been validated.

  This method should take the validated dictionary object and set any sort of attributes/state/etc. on the instance as necessary.

> **Warning:** It it is *incredibly important* that this method be idempotent with regards to instances of your Check subclass. Configurations can be altered at any time in any manner, sometimes with invalid values! You want your plugin's state to reflect the contents of the YAML config file at all times.

- `perform()`:

  This method is the heart of the health check. It performs the actual check and should return True if the health check passes and False if it fails.

  > **Note:** This method does not take any arguments, any sort of context required to perform the health check should be handled by applying the config and setting instance attributes.

3.4.2 Writing Discovery Method Plugins

Discovery plugins handle storing the topography data of clusters and services, the required functionality boils down to two things: "watching" and "reporting".

A plugin must be able to "watch" for changes in membership for clusters and react accordingly, updating Cluster instances and setting `should_update` events when they occur. Likewise, the plugin must *also* be able to cause such changes in membership by reporting nodes as available or down.

These plugins must be subclasses of *`lighthouse.discovery.Discovery`*, for an example implementation for using Zookeeper see the `lighthouse.zookeeper.ZookeeperDiscovery` class.

Having nodes agree with each other on the makeup of clusters they consume and/or take part in is important. The best candidates for discovery methods are strong "CP" distributed systems, that is reliable systems that give solid guarantees that nodes interacting with it see the same thing regardless of where they are.

Required Methods

- `validate_dependencies(cls)` *(classmethod)*:

  This classmethod should check that all required external dependencies for your plugin are met.

  If the requirements are met, this method should return True. If not met it should return False.

- `validate_config(cls, config)` *(classmethod)*:

  The "config" argument to this classmethod is the result of loading the YAML config file for the plugin (e.g. `checks/mycheck.yaml` for the example above).

  This method should analyze the config dictionary object and raise a `ValueError` exception for any invalid content.

- `apply_config(self, config)`:

  This instance method's config argument is also the result of a loaded YAML config file, albeit one that has already been validated.

  This method should take the validated dictionary object and set any sort of attributes/state/etc. on the instance as necessary.

> **Warning:** It it is *incredibly important* that this method be idempotent with regards to instances of your Discovery subclass. Configurations can be altered at any time in any manner, sometimes with invalid values! You want your plugin's state to reflect the contents of the YAML config file at all times.

- `connect()`:

  This method should handle any sort of connection establishment to the discovery method system. It takes no arguments.

- `disconnect()`:

  The `disconnect()` method is called when shutting down the discovery method and should take care of undoing any actions taken by the `connect()` call.

- `start_watching(self, cluster, should_update)`:

  This method registers a cluster to be "watched" by the discovery method. Whenever an update to the set of member nodes happens, this method must update the `nodes` list of the passed-in *`lighthouse.cluster.Cluster`* instance and call `set()` on the `should_update` threading event.

- `stop_watching(self, cluster)`:

  This method is the antithesis of the `start_watching` method, it is meant to undo any actions taken in calls to `start_watching` with the same cluster instance. Once this is called, any updates to the set of member nodes of the cluster shouldn't update the cluster instance or set the `should_update` event.

- `report_up(self, service)`:

  This is one of the two methods used by the `lighthouse-reporter` script. The single argument is a *`lighthouse.service.Service`* instance. This method should register the service as "up" to the discovery method such that any `lighthouse-writer` consumer processes will see the current node as up and configure their load balancers appropriately.

  This method should utilize the the *`lighthouse.node.Node`* class and its `serialize()` function to send data about the service on the local node to the discovery method's system.

- `report_down(self, service)`:

  This method is the other of the two used by `lighthouse-reporter` and is the antithesis of the `report_up` method. This method should tell the discovery method's system that the given service on the current node is no longer available.

3.4.3 Writing Load Balancer Plugins

The base class for balancer plugins (*`lighthouse.balancer.Balancer`*) is deceptively simple, most of the heavy lifting is left to plugin writers.

The only balancer-specific method a subclass must define is the `sync_file()` method, however there are implicit requirements for a load balancer plugin to work well, such as gracefully reacting to configuration changes without dropping traffic and not placing limits to the number of potential nodes handled.

Balancer plugins aren't necessary limited to *load* balancing either. Any sort of clustering system, such as the ones for RabbitMQ or PostgreSQL replication setups can benefit from having a balancer plugin that automatically determines potential member nodes. The only limit is your imagination!

Examples

The project includes an HAProxy balancer plugin via the *`lighthouse.haproxy.balancer.HAProxy`* class. HAProxy is a powerful tool with quite a few configuration options so the support code to get the plugin to work is extensive.

Required Methods

- `validate_dependencies(cls)` *(classmethod)*:

  This classmethod should check that all required external dependencies for your plugin are met.

  If the requirements are met, this method should return `True`. If not met it should return `False`.

- `validate_config(cls, config)` *(classmethod)*:

  The "config" argument to this classmethod is the result of loading the YAML config file for the plugin (e.g. `checks/mycheck.yaml` for the example above).

  This method should analyze the config dictionary object and raise a `ValueError` exception for any invalid content.

- `apply_config(self, config)`:

  This instance method's config argument is also the result of a loaded YAML config file, albeit one that has already been validated.

  This method should take the validated dictionary object and set any sort of attributes/state/etc. on the instance as necessary.

  > **Warning:** It it is *incredibly important* that this method be idempotent with regards to instances of your Balancer subclass. Configurations can be altered at any time in any manner, sometimes with invalid values! You want your plugin's state to reflect the contents of the YAML config file at all times.

- `sync_file(self, clusters)`:

  This method takes a list of *lighthouse.cluster.Cluster* instances and should write or update the load balancer's configuration files to reflect the member nodes of the clusters.

3.5 Source Docs

3.5.1 Pluggables

lighthouse.pluggable

class lighthouse.pluggable.**Pluggable**
Bases: *lighthouse.configurable.Configurable*

Base class for classes that can be defined via external plugins.

Subclasses define their *entry_point* attribute and subsequent calls to *get_installed_classes* will look up any available classes associated with that endpoint.

Entry points used by lighthouse can be found in `setup.py` in the root of the project.

entry_point = None

classmethod validate_dependencies()
Validates a plugin's external dependencies. Should return True if all dependencies are met and False if not.

Subclasses are expected to define this method.

classmethod get_installed_classes()
Iterates over installed plugins associated with the *entry_point* and returns a dictionary of viable ones keyed off of their names.

A viable installed plugin is one that is both loadable *and* a subclass of the Pluggable subclass in question.

classmethod from_config(*name*, *config*)
Behaves like the base Configurable class's *from_config()* except this makes sure that the *Pluggable* subclass with the given name is actually a properly installed plugin first.

lighthouse.balancer

class lighthouse.balancer.**Balancer**
Bases: *lighthouse.pluggable.Pluggable*

Base class for load balancer definitions.

The complexity of generating valid configuration content and updating the proper file(s) is left as details for subclasses so this base class remains incredibly simple.

All subclasses are expected to implement a *sync_file* method that is called whenever an update to the topography of nodes happens.

config_subdirectory = 'balancers'

entry_point = 'lighthouse.balancers'

sync_file(*clusters*)
This method must take a list of clusters and update any and all relevant configuration files with valid config content for balancing requests for the given clusters.

lighthouse.discovery

class lighthouse.discovery.**Discovery**
Bases: *lighthouse.pluggable.Pluggable*

Base class for discovery method plugins.

Unlike the `Balancer` base class for load balancer plugins, this discovery method plugin has several methods that subclasses are expected to define.

Subclasses are used for both the writer process *and* the reporter process so each subclass needs to be able to report on individual nodes as well as monitor and collect the status of all defined clusters.

It is important that the various instances of lighthouse running on various machines agree with each other on the status of clusters so a distributed system with strong CP characteristics is recommended.

config_subdirectory = 'discovery'

entry_point = 'lighthouse.discovery'

connect()
Subclasses should define this method to handle any sort of connection establishment needed.

disconnect()
This method is used to facilitate any shutting down operations needed by the subclass (e.g. closing connections and such).

start_watching(*cluster*, *should_update*)
Method called whenever a new cluster is defined and must be monitored for changes to nodes.

Once a cluster is being successfully watched that cluster *must* be added to the `self.watched_clusters` set!

Whenever a change is detected, the given `should_update` threading event should be set.

stop_watching(*cluster*)
This method should halt any of the monitoring started that would be started by a call to *start_watching()* with the same cluster.

Once the cluster is no longer being watched that cluster *must* be removed from the `self.watched_clusters` set!

report_up(*service*, *port*)
This method is used to denote that the given service present on the current machine should be considered up and available.

report_down(*service*, *port*)
This method is used to denote that the given service present on the current machine should be considered down and unavailable.

stop()
Simple method that sets the `shutdown` event and calls the subclass's `wind_down()` method.

lighthouse.check

class lighthouse.check.**Check**
Bases: *lighthouse.pluggable.Pluggable*

Base class for service check plugins.

Subclasses are expected to define a name for the check, plus methods for validating that any dependencies are present, the given config is valid, and of course performing the check itself.

entry_point = 'lighthouse.checks'

classmethod validate_check_config(*config*)
This method should return True if the given config is valid for the health check subclass, False otherwise.

apply_check_config(*config*)
This method takes an already-validated configuration dictionary as its only argument.

The method should set any attributes or state in the instance needed for performing the health check.

perform()
This *perform()* is at the heart of the check. Subclasses must define this method to actually perform their check. If the check passes, the method should return True, otherwise False.

Note that this method takes no arguments. Any sort of context required for performing a check should be handled by the config.

run()
Calls the *perform()* method defined by subclasses and stores the result in a `results` deque.

After the result is determined the `results` deque is analyzed to see if the `passing` flag should be updated. If the check was considered passing and the previous `self.fall` number of checks failed, the check is updated to not be passing. If the check was not passing and the previous `self.rise` number of checks passed, the check is updated to be considered passing.

last_n_results(*n*)
Helper method for returning a set number of the previous check results.

apply_config(*config*)
Sets attributes based on the given config.

Also adjusts the `results` deque to either expand (padding itself with False results) or contract (by removing the oldest results) until it matches the required length.

classmethod validate_config(*config*)
Validates that required config entries are present.

Each check requires a `host`, `port`, `rise` and `fall` to be configured.

The rise and fall variables are integers denoting how many times a check must pass before being considered passing and how many times a check must fail before being considered failing.

class `lighthouse.check.`**deque**(*iterable=()*, *maxlen=None*)
Bases: `collections.deque`

Custom collections.deque subclass for 2.6 compatibility.

The python 2.6 version of the deque class doesn't support referring to the *`maxlen`* attribute.

maxlen

3.5.2 Config Watching

`lighthouse.configurable`

class `lighthouse.configurable.`**Configurable**
Bases: `object`

Base class for targets configured by the config file watching system.

Each subclass is expected to be able to validate and apply configuration dictionaries that come from config file content.

name = None

config_subdirectory = None

classmethod validate_config(*config*)
Validates a given config, returns the validated config dictionary if valid, raises a ValueError for any invalid values.

Subclasses are expected to define this method.

apply_config(*config*)
Applies a given config to the subclass.

Setting instance attributes, for example. Subclasses are expected to define this method.

NOTE: It is *incredibly important* that this method be idempotent with regards to the instance.

classmethod from_config(*name*, *config*)
Returns a Configurable instance with the given name and config.

By default this is a simple matter of calling the constructor, but subclasses that are also `Pluggable` instances override this in order to check that the plugin is installed correctly first.

lighthouse.configs.watcher

lighthouse.configs.handler

lighthouse.configs.monitor

3.5.3 Service Topography

lighthouse.service

class lighthouse.service.**Service**
Bases: *lighthouse.configurable.Configurable*

Class representing a service provided by the current machine.

This is a straightforward Configurable subclass, it defines what a valid configuration for a service is and applies them.

config_subdirectory = 'services'

classmethod validate_config(*config*)
Runs a check on the given config to make sure that `port`/`ports` and `discovery` is defined.

classmethod validate_check_configs(*config*)
Config validation specific to the health check options.

Verifies that checks are defined along with an interval, and calls out to the `Check` class to make sure each individual check's config is valid.

apply_config(*config*)
Takes a given validated config dictionary and sets an instance attribute for each one.

For check definitions, a Check instance is is created and a `checks` attribute set to a dictionary keyed off of the checks' names. If the Check instance has some sort of error while being created an error is logged and the check skipped.

reset_status()
Sets the up/down status of the service ports to the default state.

Useful for when the configuration is updated and the checks involved in determining the status might have changed.

update_ports()
Sets the `ports` attribute to the set of valid port values set in the configuration.

update_checks(*check_configs*)
Maintains the values in the `checks` attribute's dictionary. Each key in the dictionary is a port, and each value is a nested dictionary mapping each check's name to the Check instance.

This method makes sure the attribute reflects all of the properly configured checks and ports. Removing no-longer-configured ports is left to the *run_checks* method.

run_checks()
Iterates over the configured ports and runs the checks on each one.

Returns a two-element tuple: the first is the set of ports that transitioned from down to up, the second is the set of ports that transitioned from up to down.

Also handles the case where a check for a since-removed port is run, marking the port as down regardless of the check's result and removing the check(s) for the port.

lighthouse.cluster

class lighthouse.cluster.**Cluster**
Bases: *lighthouse.configurable.Configurable*

The class representing a cluster of member nodes in a service.

A simple class that merely keeps a list of nodes and defines which discovery method is used to track said nodes.

config_subdirectory = 'clusters'

classmethod validate_config(*config*)
Validates a config dictionary parsed from a cluster config file.

Checks that a discovery method is defined and that at least one of the balancers in the config are installed and available.

apply_config(*config*)
Sets the `discovery` and `meta_cluster` attributes, as well as the configured + available balancer attributes from a given validated config.

lighthouse.node

class lighthouse.node.**Node**(*host*, *ip*, *port*, *peer=None*, *metadata=None*)
Bases: `object`

The class representing a member node of a cluster.

Consists of a `port`, a `host` and a `peer`, plus methods for serializing and deserializing themselves so that they can be transmitted back and forth via discovery methods.

name
Simple property for "naming" a node via the host and port.

classmethod current(*service*, *port*)
Returns a Node instance representing the current service node.

Collects the host and IP information for the current machine and the port information from the given service.

serialize()
Serializes the node data as a JSON map string.

classmethod deserialize(*value*)
Creates a new Node instance via a JSON map string.

Note that `port` and `ip` and are required keys for the JSON map, `peer` and `host` are optional. If `peer` is not present, the new Node instance will use the current peer. If `host` is not present, the hostname of the given `ip` is looked up.

lighthouse.peer

class lighthouse.peer.**Peer**(*name*, *ip*, *port=None*)
Bases: `object`

This class represents a host running a lighthouse reporter.

When a reporter script tells its discovery method that a node is up, it includes information about itself via this class so that writer scripts reading that information can coordinate their peers.

This is helpful for HAProxy as a way to generate "peers" config stanzas so instances of HAProxy in a given cluster can share stick-table data.

classmethod current()
Helper method for getting the current peer of whichever host we're running on.

serialize()
Serializes the Peer data as a simple JSON map string.

classmethod deserialize(*value*)
Generates a Peer instance via a JSON string of the sort generated by *Peer.deserialize*.

The `name` and `ip` keys are required to be present in the JSON map, if the `port` key is not present the default is used.

3.5.4 HAProxy

lighthouse.haproxy.balancer

class lighthouse.haproxy.balancer.**HAProxy**(**args*, ***kwargs*)
Bases: *lighthouse.balancer.Balancer*

The HAProxy balancer class.

Leverages the HAProxy control, config and stanza-related classes in order to keep the HAProxy config file in sync with the services and nodes discovered.

name = 'haproxy'

classmethod validate_dependencies()
The HAProxy Balancer doesn't use any specific python libraries so there are no extra dependencies to check for.

classmethod validate_config(*config*)
Validates that a config file path and a control socket file path and pid file path are all present in the HAProxy config.

classmethod validate_proxies_config(*proxies*)
Specific config validation method for the "proxies" portion of a config.

Checks that each proxy defines a port and a list of `upstreams`, and that each upstream entry has a host and port defined.

apply_config(*config*)
Constructs HAProxyConfig and HAProxyControl instances based on the contents of the config.

This is mostly a matter of constructing the configuration stanzas.

sync_file(*clusters*)
Generates new HAProxy config file content and writes it to the file at `haproxy_config_path`.

If a restart is not necessary the nodes configured in HAProxy will be synced on the fly. If a restart *is* necessary, one will be triggered.

restart()
Tells the HAProxy control object to restart the process.

If it's been fewer than `restart_interval` seconds since the previous restart, it will wait until the interval has passed. This staves off situations where the process is constantly restarting, as it is possible to drop packets for a short interval while doing so.

sync_nodes(*clusters*)
Syncs the enabled/disabled status of nodes existing in HAProxy based on the given clusters.

This is used to inform HAProxy of up/down nodes without necessarily doing a restart of the process.

get_current_nodes(*clusters*)
Returns two dictionaries, the current nodes and the enabled nodes.

The current_nodes dictionary is keyed off of the cluster name and values are a list of nodes known to HAProxy.

The enabled_nodes dictionary is also keyed off of the cluster name and values are list of *enabled* nodes, i.e. the same values as current_nodes but limited to servers currently taking traffic.

lighthouse.haproxy.control

class lighthouse.haproxy.control.**HAProxyControl**(*config_file_path*, *socket_file_path*, *pid_file_path*)
Bases: object

Class used to control a running HAProxy process.

Includes basic functionality for soft restarts as well as gathering info about the HAProxy process and its active nodes, plus methods for enabling or disabling nodes on the fly.

Also allows for sending commands to the HAProxy control socket itself.

restart()
Performs a soft reload of the HAProxy process.

get_version()
Returns a tuple representing the installed HAProxy version.

The value of the tuple is (<major>, <minor>, <patch>), e.g. if HAProxy version 1.5.3 is installed, this will return `(1, 5, 3)`.

get_info()
Parses the output of a "show info" HAProxy command and returns a simple dictionary of the results.

get_active_nodes()
Returns a dictionary of lists, where the key is the name of a service and the list includes all active nodes associated with that service.

enable_node(*service_name*, *node_name*)
Enables a given node name for the given service name via the "enable server" HAProxy command.

disable_node(*service_name*, *node_name*)
Disables a given node name for the given service name via the "disable server" HAProxy command.

send_command(*command*)
Sends a given command to the HAProxy control socket.

Returns the response from the socket as a string.

If a known error response (e.g. "Permission denied.") is given then the appropriate exception is raised.

process_command_response(*command*, *response*)
Takes an HAProxy socket command and its response and either raises an appropriate exception or returns the formatted response.

exception lighthouse.haproxy.control.**HAProxyControlError**
Bases: exceptions.Exception

Base exception for HAProxyControl-related actions.

exception lighthouse.haproxy.control.**UnknownCommandError**
Bases: *lighthouse.haproxy.control.HAProxyControlError*

Exception raised if an unrecognized command was sent to the HAProxy socket.

exception lighthouse.haproxy.control.**PermissionError**
Bases: *lighthouse.haproxy.control.HAProxyControlError*

Exception denoting that the HAProxy control socket does not have proper authentication level for executing a given command.

For example, if the socket is set up with a a level lower than "admin", the enable/disable server commands will fail.

exception lighthouse.haproxy.control.**UnknownServerError**
Bases: *lighthouse.haproxy.control.HAProxyControlError*

Exception raised if an enable/disable server command is executed against a backend that HAProxy doesn't know about.

lighthouse.haproxy.config

class lighthouse.haproxy.config.**HAProxyConfig**(*global_stanza*, *defaults_stanza*, *proxy_stanzas=None*, *stats_stanza=None*, *meta_clusters=None*, *bind_address=None*)
Bases: object

Class for generating HAProxy config file content.

Requires global and defaults stanzas to be passed, can optionally take a stats_stanza for enabling a stats portal.

generate(*clusters*, *version=None*)
Generates HAProxy config file content based on a given list of clusters.

get_meta_clusters(*clusters*)
Returns a dictionary keyed off of meta cluster names, where the values are lists of clusters associated with the meta cluster name.

If a meta cluster name doesn't have a port defined in the meta_cluster_ports attribute an error is given and the meta cluster is removed from the mapping.

lighthouse.haproxy.stanzas

class lighthouse.haproxy.stanzas.stanza.**Stanza**(*section_name*)
Bases: object

Subclass for config file stanzas.

In an HAProxy config file, a stanza is in the form of:

```
stanza header
    directive
    directive
    directive
```

Stanza instances have a header attribute for the header and a list of lines, one for each directive line.

add_lines(*lines*)
Simple helper method for adding multiple lines at once.

add_line(*line*)
Adds a given line string to the list of lines, validating the line first.

is_valid_line(*line*)
Validates a given line against the associated "section" (e.g. 'global' or 'frontend', etc.) of a stanza.

If a line represents a directive that shouldn't be within the stanza it is rejected. See the `directives.json` file for a condensed look at valid directives based on section.

class lighthouse.haproxy.stanzas.meta.**MetaFrontendStanza**(*name*, *port*, *lines*, *members*, *bind_address=None*)
Bases: *lighthouse.haproxy.stanzas.stanza.Stanza*

Stanza subclass representing a shared "meta" cluster frontend.

These frontends just contain ACL directives for routing requests to separate cluster backends. If a member cluster does not have an ACL rule defined in its haproxy config an error is logged and the member cluster is skipped.

class lighthouse.haproxy.stanzas.frontend.**FrontendStanza**(*cluster*, *bind_address=None*)
Bases: *lighthouse.haproxy.stanzas.stanza.Stanza*

Stanza subclass representing a "frontend" stanza.

A frontend stanza defines an address to bind to an a backend to route traffic to. A cluster can defined custom lines via a "frontend" entry in their haproxy config dictionary.

class lighthouse.haproxy.stanzas.backend.**BackendStanza**(*cluster*)
Bases: *lighthouse.haproxy.stanzas.stanza.Stanza*

Stanza subclass representing a "backend" stanza.

A backend stanza defines the nodes (or "servers") belonging to a given cluster as well as how routing/load balancing between those nodes happens.

A given cluster can define custom directives via a list of lines in their haproxy config with the key "backend".

class lighthouse.haproxy.stanzas.peers.**PeersStanza**(*cluster*)
Bases: *lighthouse.haproxy.stanzas.stanza.Stanza*

Stanza subclass representing a "peers" stanza.

This stanza lists "peer" haproxy instances in a cluster, so that each instance can coordinate and share stick-table information. Useful for tracking cluster-wide stats.

class lighthouse.haproxy.stanzas.proxy.**ProxyStanza**(*name*, *port*, *upstreams*, *options=None*, *bind_address=None*)
Bases: *lighthouse.haproxy.stanzas.stanza.Stanza*

Stanza for independent proxy directives.

These are used to add simple proxying to a system, e.g. communicating with a third party service via a dedicated internal machine with a white- listed IP.

class lighthouse.haproxy.stanzas.stats.**StatsStanza**(*port*, *uri='/'*)
Bases: *lighthouse.haproxy.stanzas.stanza.Stanza*

Stanza subclass representing a "listen" stanza specifically for the HAProxy stats feature.

Takes an optional uri parameter that defaults to the root uri.

class lighthouse.haproxy.stanzas.section.**Section**(*heading*, **stanzas*)
Bases: object

Represents a section of HAProxy config file stanzas.

This is used to organize generated config file content and provide header comments for sections describing nature of the grouped-together stanzas.

header

3.5.5 Zookeeper

lighthouse.zookeeper

3.5.6 Service Checks

lighthouse.checks.http

class lighthouse.checks.http.**HTTPCheck**(**args*, ***kwargs*)
Bases: *lighthouse.check.Check*

Simple check for HTTP services.

Pings a configured uri on the host. The check passes if the response code is in the 2xx range.

name = 'http'

classmethod validate_dependencies()
This check uses stdlib modules so dependencies are always present.

classmethod validate_check_config(*config*)
Validates the http check config. The "uri" key is required.

apply_check_config(*config*)
Takes a validated config dictionary and sets the uri, use_https and method attributes based on the config's contents.

perform()
Performs a simple HTTP request against the configured url and returns true if the response has a 2xx code.

The url can be configured to use https via the "https" boolean flag in the config, as well as a custom HTTP method via the "method" key.

The default is to not use https and the GET method.

lighthouse.checks.tcp

class lighthouse.checks.tcp.**TCPCheck**(**args*, ***kwargs*)
Bases: *lighthouse.check.Check*

Service health check using TCP request/response messages.

Sends a certain message to the configured port and passes if the response is an expected one.

name = 'tcp'

classmethod validate_dependencies()
This check uses stdlib modules so dependencies are always present.

classmethod validate_check_config(*config*)
Ensures that a query and expected response are configured.

apply_check_config(*config*)
Takes the `query` and `response` fields from a validated config dictionary and sets the proper instance attributes.

perform()
Performs a straightforward TCP request and response.

Sends the TCP `query` to the proper host and port, and loops over the socket, gathering response chunks until a full line is acquired.

If the response line matches the expected value, the check passes. If not, the check fails. The check will also fail if there's an error during any step of the send/receive process.

3.5.7 Script Classes

lighthouse.reporter

lighthouse.writer

3.5.8 Helper Modules

lighthouse.log

lighthouse.log.**setup**(*program*)
Simple function that sets the program on the ContextFilter and returns the root logger.

lighthouse.events

lighthouse.events.**wait_on_any**(**events*, ***kwargs*)
Helper method for waiting for any of the given threading events to be set.

The standard threading lib doesn't include any mechanism for waiting on more than one event at a time so we have to monkey patch the events so that their `set()` and `clear()` methods fire a callback we can use to determine how a composite event should react.

lighthouse.events.**wait_on_event**(*event*, *timeout=None*)
Waits on a single threading Event, with an optional timeout.

This is here for compatibility reasons as python 2 can't reliably wait on an event without a timeout and python 3 doesn't define a `maxint`.

3.5.9 Redis plugins

lighthouse.redis.check

class lighthouse.redis.check.**RedisCheck**(**args*, ***kwargs*)
Bases: *lighthouse.checks.tcp.TCPCheck*

Redis service checker.

Pings a redis server to make sure that it's available.

name = 'redis'

classmethod validate_check_config(*config*)
The base Check class assures that a host and port are configured so this method is a no-op.

apply_check_config(*config*)
This method doesn't actually use any configuration data, as the query and response for redis are already established.

3.5.10 Logging

lighthouse.log.config

class lighthouse.log.config.**Logging**
Bases: *lighthouse.configurable.Configurable*

Simple `Configurable` subclass that allows for runtime configuration of python's logging infrastructure.

Since python provides a handy `dictConfig` function and our system already provides the watched file contents as dicts the work here is tiny.

name = 'logging'

classmethod from_config(*name*, *config*)
Override of the base *from_config()* method that returns `None` if the name of the config file isn't "logging".

We do this in case this `Configurable` subclass winds up sharing the root of the config directory with other subclasses.

classmethod validate_config(*config*)
The validation of a logging config is a no-op at this time, the call to dictConfig() when the config is applied will do the validation for us.

apply_config(*config*)
Simple application of the given config via a call to the `logging` module's `dictConfig()` method.

lighthouse.log.context

class lighthouse.log.context.**ContextFilter**(*name=''*)
Bases: `logging.Filter`

Simple `logging.Filter` subclass that adds a *program* attribute to each `LogRecord`.

The attribute's value comes from the "program" class attribute.

Initialize a filter.

Initialize with the name of the logger which, together with its children, will have its events allowed through the filter. If no name is specified, allow every event.

program = None

filter(*record*)
Sets the *program* attribute on the record. Always returns `True` as we're not actually filtering any records, just enhancing them.

lighthouse.log.cli

lighthouse.log.cli.**color_string**(*color*, *string*)
Colorizes a given string, if coloring is available.

lighthouse.log.cli.**color_for_level**(*level*)
Returns the colorama Fore color for a given log level.

If color is not available, returns None.

lighthouse.log.cli.**create_thread_color_cycle**()
Generates a never-ending cycle of colors to choose from for individual threads.

If color is not available, a cycle that repeats None every time is returned instead.

lighthouse.log.cli.**color_for_thread**(*thread_id*)
Associates the thread ID with the next color in the `thread_colors` cycle, so that thread-specific parts of a log have a consistent separate color.

class lighthouse.log.cli.**CLIHandler**(*stream=None*)
Bases: `logging.StreamHandler`, `object`

Specialized StreamHandler that provides color output if the output is a terminal and the colorama library is available.

Initialize the handler.

If stream is not specified, sys.stderr is used.

is_tty
Returns true if the handler's stream is a terminal.

format(*record*)
Formats a given log record to include the timestamp, log level, thread ID and message. Colorized if coloring is available.

3.6 Release Notes

3.6.1 0.10.0

- Update nomenclature: "balancer" is now "coordinator", as it is a better fit with what the class actually does.
- The redis check is moved to an "extra". To install it the bracket syntax must be used (i.e. "pip install lighthouse[redis]").
- New TCPCheck health check for services that expose a TCP command for health checks, such as redis and zookeeper.
- A service that has no valid health checks no longer defaults to unavailable. Each round of checks fires a warning about having no checks but still reports as available.

3.6.2 0.11.0

- Revert update of the "balancer" nomenclature. Any sort of updates to go beyond using load balancers is a long way off and it's best not to get ahead of ourselves.
- Update TCP check's query & response to be optional. If both are omitted from a config, a simple successful connection will cause the check to pass.

3.6.3 0.11.1

- Fix bug where health check ports passed as strings caused exceptions.

3.6.4 0.11.2

- Update reporter to use a ThreadPoolExecutor rather than the older undocumented ThreadPool.
- Fix bug where if multiple services were present, only one would be checked and reported on.

3.6.5 0.12.0

- Fix bug where a blank HAProxy config would be written out and not updated
- Add a "multi-port" feature, service config files can now specify multiple ports for use cases where multiple instances of the same service run on the same machine.

3.6.6 0.13.0

- Add feature for configuring the logging system.

3.6.7 0.13.1

- Add optional ContextFilter so that the current program "WRITER" or "REPORTER" is available on log records for formatting.

3.6.8 0.14.0

- Big refactor to how concurrecy is handled, should fix situations where the writer process would spit out an incomplete HAProxy config when restarted.

3.6.9 0.15.0

- Fixed a bug where HAProxy reloads weren't supplanting the existing processes
- Fixed a bug where files with .yml as the extension were being ignored.
- Config files for haproxy now live in a "balancers" subdirectory.
- Completely revamped the logging system, logging is now configured via a "logging.yml" file at the root of the config directory.

3.6.10 0.15.1

- Fix a bug where HAProxy would be restarted several times simultaneously, winding up with several processes at once.

3.6.11 0.9.0

- Initial public release

3.6.12 0.9.1

- Small fixups to documentation and tests

3.6.13 0.9.2

- Fixes to documentation, links and spelling etc.
- Include the classifiers.txt file in the manifest so pip installs work again

3.6.14 1.0.0

- Initial stable release!
- Updates to fix stricter style tests
- Re-vamped the look and feel of the generated docs

l

A

B

C

D

E

F

R

S

T

U

V

W