Scrapy Cluster Documentation

Release 1.2

IST Research

Sep 27, 2017

Contents

This documentation provides everything you need to know about the Scrapy based distributed crawling project, Scrapy Cluster.

CHAPTER 1

Introduction

This set of documentation serves to give an introduction to Scrapy Cluster, including goals, architecture designs, and how to get started using the cluster.

Overview

This Scrapy project uses Redis and Kafka to create a distributed on demand scraping cluster.

The goal is to distribute seed URLs among many waiting spider instances, whose requests are coordinated via Redis. Any other crawls those trigger, as a result of frontier expansion or depth traversal, will also be distributed among all workers in the cluster.

The input to the system is a set of Kafka topics and the output is a set of Kafka topics. Raw HTML and assets are crawled interactively, spidered, and output to the log. For easy local development, you can also disable the Kafka portions and work with the spider entirely via Redis, although this is not recommended due to the serialization of the crawl requests.

Dependencies

Please see the `requirements.txt` within each sub project for Pip package dependencies.

Other important components required to run the cluster

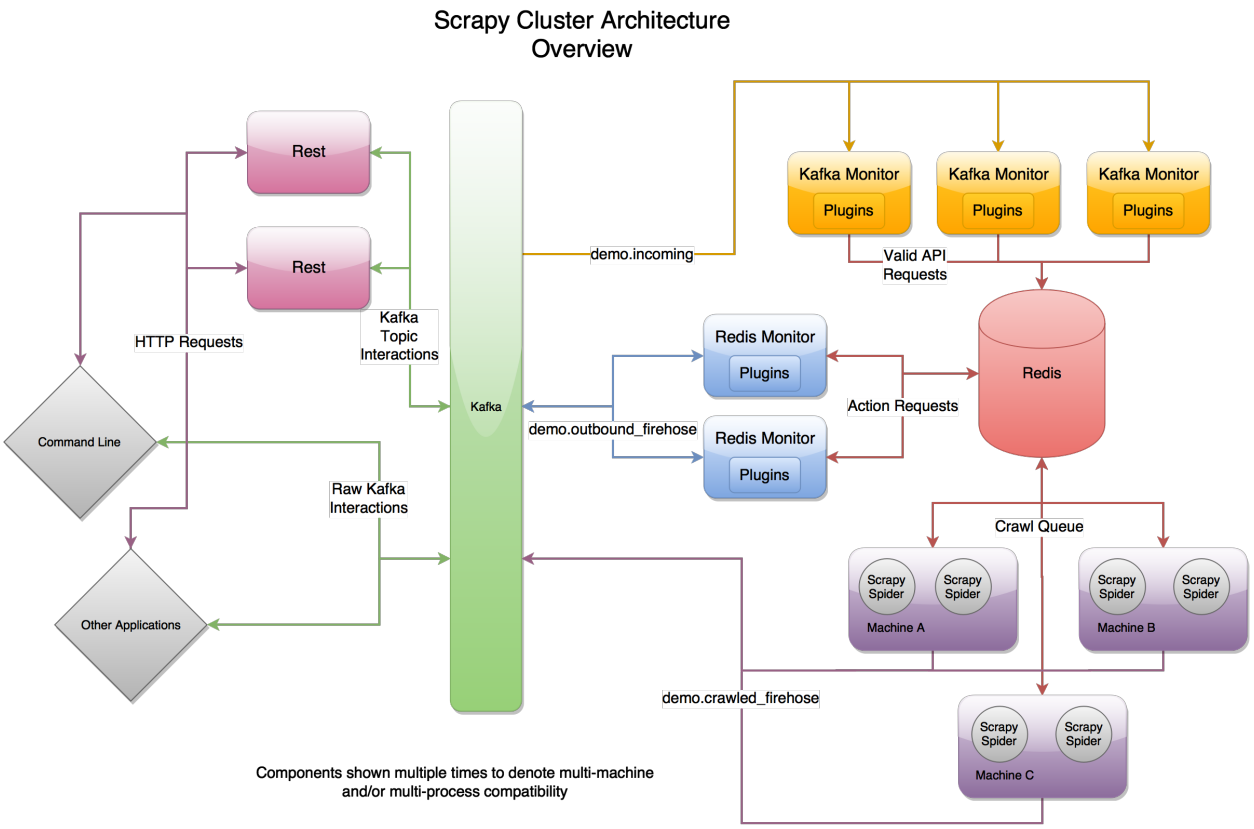
- Python 2.7: https://www.python.org/downloads/
- Redis: http://redis.io
- Zookeeper: https://zookeeper.apache.org
- Kafka: http://kafka.apache.org

Core Concepts

This project tries to bring together a bunch of new concepts to Scrapy and large scale distributed crawling in general. Some bullet points include:

- The spiders are dynamic and on demand, meaning that they allow the arbitrary collection of any web page that is submitted to the scraping cluster
- Scale Scrapy instances across a single machine or multiple machines
- Coordinate and prioritize their scraping effort for desired sites
- Persist data across scraping jobs
- Execute multiple scraping jobs concurrently
- Allows for in depth access into the information about your scraping job, what is upcoming, and how the sites are ranked
- Allows you to arbitrarily add/remove/scale your scrapers from the pool without loss of data or downtime
- Utilizes Apache Kafka as a data bus for any application to interact with the scraping cluster (submit jobs, get info, stop jobs, view results)
- Allows for coordinated throttling of crawls from independent spiders on separate machines, but behind the same IP Address

Architecture



At the highest level, Scrapy Cluster operates on a single input Kafka topic, and two separate output Kafka topics. All incoming requests to the cluster go through the `demo.incoming` kafka topic, and depending on what the request is will generate output from either the `demo.outbound_firehose` topic for action requests or `demo.crawled_firehose` topics for html crawl requests.

Each of the three core pieces are extendable in order add or enhance their functionality. Both the Kafka Monitor and Redis Monitor use 'Plugins' in order to enhance their abilities, whereas Scrapy uses 'Middlewares', 'Pipelines', and 'Spiders' to allow you to customize your crawling. Together these three components and the Rest service allow for scaled and distributed crawling across many machines.

Quick Start

This guide does not go into detail as to how everything works, but hopefully will get you scraping quickly. For more information about each process works please see the rest of the documentation.

Setup

There are a number of different options available to set up Scrapy Cluster. You can chose to provision with *Vagrant Quickstart*, use the *Docker Quickstart*, or manually configure via the *Cluster Quickstart* yourself.

Vagrant Quickstart

The Vagrant Quickstart provides you a simple Vagrant Virtual Machine in order to try out Scrapy Cluster. It is not meant to be a production crawling machine, and should be treated more like a test ground for development.

1. Ensure you have Vagrant and VirtualBox installed on your machine. For instructions on setting those up please refer to the official documentation.
2. Download the latest `master` branch release via

```
git clone https://github.com/istresearch/scrapy-cluster.git
# or
git clone git@github.com:istresearch/scrapy-cluster.git
```

You may also use the pre-packaged git releases, however the latest commit on the `master` branch will always have to most up to date code or minor tweaks that do not deserve another release.

Lets assume our project is now in `~/scrapy-cluster`

3. Stand up the Scrapy Cluster Vagrant machine. By default this will start an **Ubuntu** virtual machine. If you would like to us **CentOS**, change the following line in the `Vagrantfile` in the root of the project

```
node.vm.box = 'centos/7'
```

Bring the machine up.

```
$ cd ~/scrapy-cluster
$ vagrant up
# This will set up your machine, provision it with Ansible, and boot the required␣
↪services
```

Note: The initial setup may take some time due to the setup of Zookeeper, Kafka, and Redis. This is only a one time setup and is skipped when running it in the future.

4. SSH onto the machine

```
$ vagrant ssh
```

5. Ensure everything under supervision is running.

```
vagrant@scdev:~$ sudo supervisorctl status
kafka                            RUNNING   pid 1812, uptime 0:00:07
redis                            RUNNING   pid 1810, uptime 0:00:07
zookeeper                        RUNNING   pid 1809, uptime 0:00:07
```

Note: If you receive the message `unix:///var/run/supervisor.sock no such file`, issue the following command to start Supervisord: `sudo service supervisord start`, then check the status like above.

6. Create a Virtual Environment for Scrapy Cluster, and activate it. This is where the python packages will be installed to.

```
vagrant@scdev:~$ virtualenv sc
...
vagrant@scdev:~$ source sc/bin/activate
```

7. The Scrapy Cluster folder is mounted in the `/vagrant/` directory

```
(sc) vagrant@scdev:~$ cd /vagrant/
```

8. Install the Scrapy Cluster packages

```
(sc) vagrant@scdev:/vagrant$ pip install -r requirements.txt
```

9. Ensure the offline tests pass

```
(sc) vagrant@scdev:/vagrant$ ./run_offline_tests.sh
# There should be 5 core pieces, each of them saying all tests passed like so
----------------------------------------------------------------------
Ran 20 tests in 0.034s
...
----------------------------------------------------------------------
Ran 9 tests in 0.045s
...
```

10. Ensure the online tests pass

```
(sc) vagrant@scdev:/vagrant$ ./run_online_tests.sh
# There should be 5 major blocks here, ensuring your cluster is setup correctly.
...
----------------------------------------------------------------------
Ran 2 tests in 0.105s
...
----------------------------------------------------------------------
Ran 1 test in 26.489s
```

Warning: If this test fails, it most likely means the Virtual Machine's Kafka is in a finicky state. Issue the following command and then retry the online test to fix Kafka: `sudo supervisorctl restart kafka`

You now appear to have a working test environment, so jump down to *Your First Crawl* to finish the quickstart.

Docker Quickstart

The Docker Quickstart will help you spin up a complete standalone cluster thanks to Docker and Docker Compose. All individual components will run in standard docker containers, and be controlled through the `docker-compose` command line interface.

1. Ensure you have Docker Engine and Docker Compose installed on your machine. For more information about installation please refer to Docker's official documentation.
2. Download and unzip the latest release here.

Lets assume our project is now in `~/scrapy-cluster`

3. Run docker compose

```
$ docker-compose up -d
```

This will pull the latest stable images from Docker hub and build your scraping cluster.

At time of writing, there is no Docker container to interface and run all of the tests within your compose-based cluster. Instead, if you wish to run the unit and integration tests plese see the following steps.

4. To run the integration tests, get into the bash shell on any of the containers.

   Kafka monitor

   ```
   $ docker exec -it scrapycluster_kafka_monitor_1 bash
   ```

   Redis monitor

   ```
   $ docker exec -it scrapycluster_redis_monitor_1 bash
   ```

   Crawler

   ```
   $ docker exec -it scrapycluster_crawler_1 bash
   ```

   Rest

   ```
   $ docker exec -it scrapycluster_rest_1 bash
   ```

5. Run the unit and integration test for that component. Note that your output may be slightly different but your tests should pass consistently.

```
$ ./run_docker_tests.sh
...

----------------------------------------------------------------------
Ran 20 tests in 5.742s

OK
...

----------------------------------------------------------------------
Ran 1 test in 27.583s

OK
```

This script will run both of offline unit tests and the online integration tests for your particular container. You will want to do this on all three component containers.

You now appear to have a working docker environment, so jump down to *Your First Crawl* to finish the quickstart. Note that since this is a precanned cluster thanks to docker compose, you have everything already spun up except the dump utilities.

Cluster Quickstart

The Cluster Quickstart will help you set up your components across a number of different machines. Here, we assume everything runs on a single box with external Kafka, Zookeeper, and Redis.

1. Make sure you have Apache Zookeeper, Apache Kafka, and Redis up and running on your cluster. For more information about standing those up, please refer to the official project documentation.
2. Download and unzip the latest release here.

Lets assume our project is now in `~/scrapy-cluster`

3. Install the requirements on every machine

```
$ cd ~/scrapy-cluster
$ pip install -r requirements.txt
```

4. Run the offline unit tests to ensure everything seems to be functioning correctly.

```
$ ./run_offline_tests.sh
# There should be 5 core pieces, each of them saying all tests passed like so
----------------------------------------------------------------------
Ran 20 tests in 0.034s
...
----------------------------------------------------------------------
Ran 9 tests in 0.045s
...
```

Lets now setup and ensure our cluster can talk with Redis, Kafka, and Zookeeper

5. Add a new file called `localsettings.py` in the Kafka Monitor folder.

```
$ cd kafka-monitor/
$ vi localsettings.py
```

Add the following to your new custom local settings.

```
# Here, 'scdev' is the host with Kafka and Redis
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
```

It is recommended you use this 'local' override instead of altering the default `settings.py` file, in order to preserve the original configuration the cluster comes with in case something goes wrong, or the original settings need updated.

6. Now, lets run the online integration test to see if our Kafka Monitor is set up correctly

```
$ python tests/online.py -v
test_feed (__main__.TestKafkaMonitor) ... ok
test_run (__main__.TestKafkaMonitor) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.104s

OK
```

This integration test creates a dummy Kafka topic, writes a JSON message to it, ensures the Kafka Monitor reads the message, and puts the request into Redis.

Warning: If your integration test fails, please ensure the port(s) are open on the machine your Kafka cluster and your Redis host resides on, and that the particular machine this is set up on can access the specified hosts.

7. We now need to do the same thing for the Redis Monitor

```
$ cd ../redis-monitor
$ vi localsettings.py
```

Add the following to your new custom local settings.

```
# Here, 'scdev' is the host with Kafka and Redis
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
```

8. Run the online integration tests

```
$ python tests/online.py -v
test_process_item (__main__.TestRedisMonitor) ... ok
test_sent_to_kafka (__main__.TestRedisMonitor) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.028s

OK
```

This integration test creates a dummy entry in Redis, ensures the Redis Monitor processes it, and writes the result to a dummy Kafka Topic.

Warning: If your integration test fails, please ensure the port(s) are open on the machine your Kafka cluster and your Redis host resides on, and that the particular machine this is set up on can access the specified hosts.

9. Now let's setup our crawlers.

```
$ cd ../crawlers/crawling/
$ vi localsettings.py
```

Add the following fields to override the defaults

```
# Here, 'scdev' is the host with Kafka, Redis, and Zookeeper
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
ZOOKEEPER_HOSTS = 'scdev:2181'
```

10. Run the online integration tests to see if the crawlers work.

```
$ cd ../
$ python tests/online.py -v
...
----------------------------------------------------------------------
Ran 1 test in 23.191s

OK
```

This test spins up a spider using the internal Scrapy API, directs it to a real webpage to go crawl, then ensures it writes the result to Kafka.

Note: This test takes around 20 - 25 seconds to complete, in order to compensate for server response times or potential crawl delays.

Note: You may see 'Deprecation Warnings' while running this test! This is okay and may be caused by irregularities in Scrapy or how we are using or overriding packages.

Warning: If your integration test fails, please ensure the port(s) are open on the machine your Kafka cluster, your Redis host, and your Zookeeper hosts. Ensure that the machines the crawlers are set up on can access the desired hosts, and that your machine can successfully access the internet.

11. If you would like, you can set up the rest service as well

```
$ cd ../rest/
$ vi localsettings.py
```

Add the following fields to override the defaults

```
# Here, 'scdev' is the host with Kafka and Redis
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
```

12. Run the online integration tests to see if the rest service works.

```
$ python tests/online.py -v
test_status (__main__.TestRestService) ...  * Running on http://0.0.0.0:62976/ (Press␣
↪CTRL+C to quit)
127.0.0.1 - - [11/Nov/2016 17:09:17] "GET / HTTP/1.1" 200 -
ok

----------------------------------------------------------------------
Ran 1 test in 15.034s

  OK
```

Your First Crawl

At this point you should have a Scrapy Cluster setup that has been tested and appears to be operational. We can choose to start up either a bare bones cluster, or a fully operational cluster.

Note: You can append `&` to the end of the following commands to run them in the background, but we recommend you open different terminal windows to first get a feel of how the cluster operates.

The following commands outline what you would run in a traditional environment. If using a container based solution these commands are ran when you run the container itself.

Bare Bones:

- The Kafka Monitor:

```
python kafka_monitor.py run
```

- A crawler:

```
scrapy runspider crawling/spiders/link_spider.py
```

- The dump utility located in Kafka Monitor to see your crawl results

```
python kafkadump.py dump -t demo.crawled_firehose
```

Fully Operational:

- The Kafka Monitor (1+):

```
python kafka_monitor.py run
```

- The Redis Monitor (1+):

```
python redis_monitor.py
```

- A crawler (1+):

```
scrapy runspider crawling/spiders/link_spider.py
```

- The rest service (1+):

```
python rest_service.py
```

- The dump utility located in Kafka Monitor to see your crawl results

```
python kafkadump.py dump -t demo.crawled_firehose
```

- The dump utility located in Kafka Monitor to see your action results

```
python kafkadump.py dump -t demo.outbound_firehose
```

Which ever setup you chose, every process within should stay running for the remainder that your cluster is in an operational state.

Note: If you chose to set the Rest service up, this section may also be performed via the *Rest* endpoint. You just need to ensure the JSON identified in the following section is properly fed into the *feed* rest endpoint.

The follwing commands can be ran from the command line, whether that is on the machine itself or inside the Kafka Monitor container depends on the setup chosen above.

1. We now need to feed the cluster a crawl request. This is done via the same Kafka Monitor python script, but with different command line arguements.

```
python kafka_monitor.py feed '{"url": "http://istresearch.com", "appid":"testapp",
↪"crawlid":"abc123"}'
```

You will see the following output on the command line for that successful request:

```
2015-12-22 15:45:37,457 [kafka-monitor] INFO: Feeding JSON into demo.incoming
{
    "url": "http://istresearch.com",
    "crawlid": "abc123",
    "appid": "testapp"
}
2015-12-22 15:45:37,459 [kafka-monitor] INFO: Successfully fed item to Kafka
```

You will see an error message in the log if the script cannot connect to Kafka in time.

2. After a successful request, the following chain of events should occur in order:

1. The Kafka monitor will receive the crawl request and put it into Redis
2. The spider periodically checks for new requests, and will pull the request from the queue and process it like a normal Scrapy spider.
3. After the scraped item is yielded to the Scrapy item pipeline, the Kafka Pipeline object will push the result back to Kafka
4. The Kafka Dump utility will read from the resulting output topic, and print out the raw scrape object it received

3. The Redis Monitor utility is useful for learning about your crawl while it is being processed and sitting in redis, so we will pick a larger site so we can see how it works (this requires a full deployment).

Crawl Request:

```
python kafka_monitor.py feed '{"url": "http://dmoz.org", "appid":"testapp", "crawlid":
↪"abc1234", "maxdepth":1}'
```

Now send an `info` action request to see what is going on with the crawl:

```
python kafka_monitor.py feed '{"action":"info", "appid":"testapp", "uuid":"someuuid",
↪"crawlid":"abc1234", "spiderid":"link"}'
```

The following things will occur for this action request:

1. The Kafka monitor will receive the action request and put it into Redis
2. The Redis Monitor will act on the info request, and tally the current pending requests for the particular `spiderid`, `appid`, and `crawlid`
3. The Redis Monitor will send the result back to Kafka
4. The Kafka Dump utility monitoring the actions will receive a result similar to the following:

```
{u'server_time': 1450817666, u'crawlid': u'abc1234', u'total_pending': 25, u
↪'total_domains': 2, u'spiderid': u'link', u'appid': u'testapp', u'domains
↪': {u'twitter.com': {u'low_priority': -9, u'high_priority': -9, u'total':␣
↪1}, u'dmoz.org': {u'low_priority': -9, u'high_priority': -9, u'total': 24}}
↪, u'uuid': u'someuuid'}
```

In this case we had 25 urls pending in the queue, so yours may be slightly different.

4. If the crawl from step 1 is still running, lets stop it by issuing a `stop` action request (this requires a full deployment).

Action Request:

```
python kafka_monitor.py feed  '{"action":"stop", "appid":"testapp", "uuid":"someuuid",
↪ "crawlid":"abc1234", "spiderid":"link"}'
```

The following things will occur for this action request:

1. The Kafka monitor will receive the action request and put it into Redis
2. The Redis Monitor will act on the stop request, and purge the current pending requests for the particular `spiderid`, `appid`, and `crawlid`
3. The Redis Monitor will blacklist the `crawlid`, so no more pending requests can be generated from the spiders or application
4. The Redis Monitor will send the purge total result back to Kafka
5. The Kafka Dump utility monitoring the actions will receive a result similar to the following:

```
{u'total_purged': 90, u'server_time': 1450817758, u'crawlid': u'abc1234', u
↪'spiderid': u'link', u'appid': u'testapp', u'action': u'stop'}
```

In this case we had 90 urls removed from the queue. Those pending requests are now completely removed from the system and the spider will go back to being idle.

Hopefully you now have a working Scrapy Cluster that allows you to submit jobs to the queue, receive information about your crawl, and stop a crawl if it gets out of control. For a more in depth look, please continue reading the documentation for each component.

Overview Learn about the Scrapy Cluster Architecture.

Quick Start A Quick Start guide to those who want to jump right in.

CHAPTER 2

Kafka Monitor

The Kafka Monitor serves as the entry point into the crawler architecture. It validates API requests so that any point afterwards can ensure the data is in the correct format.

Design

The design of the Kafka Monitor stemmed from the need to define a format that allowed for the creation of crawls in the crawl architecture from any application. If the application could read and write to the kafka cluster then it could write messages to a particular kafka topic to create crawls.

Soon enough those same applications wanted the ability to retrieve information about their crawls, stop them, or get information about their cluster. We decided to make a dynamic interface that could support all of the request needs, but utilize the same base code. This base code is now known as the Kafka Monitor, which utilizes various Plugins to extend or alter the Kafka Monitor's functionality.

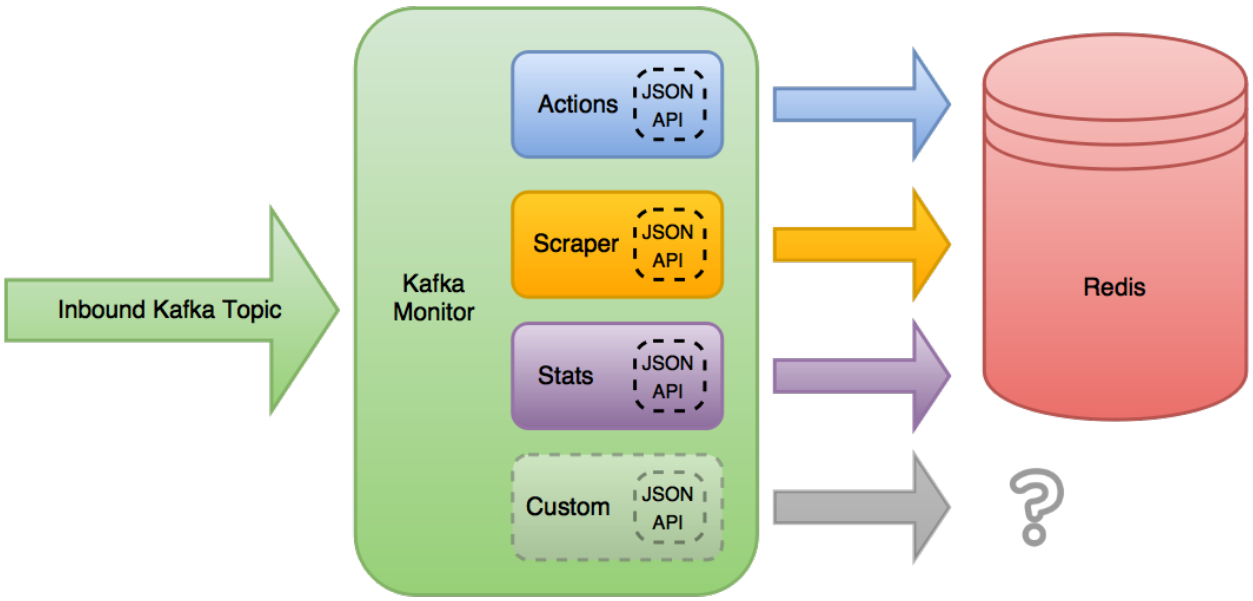
The Kafka Monitor reads from the desired inbound Kafka topic, and applies the currently loaded Plugin's JSON APIs to the received message. The first Plugin to have a valid JSON Schema for the received JSON object is then allowed to do its own processing and manipulation of the object.

In Scrapy Cluster's use case, the default Plugins write their requests into Redis keys, but the functionality does not stop there. The Kafka Monitor settings can alter which plugins are loaded, or add new plugins to extend functionality. These modules allow the Kafka Monitor core to have a small footprint but allow extension or different plugins to be ran.

The Kafka Monitor can be ran as a single process, or part of the same Kafka consumer group spread across multiple machines and processes. This allows distributed and fault tolerant throughput to ensure the crawl requests to the cluster are always read.

From our own internal debugging and ensuring other applications were working properly, a utility program called Kafka Dump was also created in order to be able to interact and monitor the kafka messages coming through. This is a small dump utility with no external dependencies, to allow users to get an insight into what is being passed through the Kafka topics within the cluster.

From our own internal debugging and ensuring other applications were working properly, a utility program called Kafka Dump was also created in order to be able to interact and monitor the kafka messages coming through. This is



a small dump utility with no external dependencies, to allow users to get an insight into what is being passed through the Kafka topics within the cluster.

Quick Start

First, create a `localsettings.py` file to track your custom settings. You can override any setting in the normal `settings.py` file, and a typical file may look like the following:

```
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
LOG_LEVEL = 'DEBUG'
```

kafka_monitor.py

The Kafka Monitor allows two modes of operation.

Run

Continuous run mode. Will accept incoming Kafka messages from a topic, validate the message as JSON and then against all possible JSON API's, and then allow the valid API Plugin to process the object.

> **Attention:** Run mode is the main process you should have running!

```
$ python kafka_monitor.py run
```

Feed

JSON Object feeder into your desired Kafka Topic. This takes a valid JSON object from the command line and inserts it into the desired Kafka Topic, to then be consumed by the *Run* command above.

```
$ python kafka_monitor.py feed '{"url": "http://istresearch.com", "appid":"testapp",
↪"crawlid":"ABC123"}'
```

The command line feed is very slow and should not be used in production. Instead, you should write your own continuously running application to feed Kafka the desired API requests that you require.

kafkadump.py

A basic kafka topic utility used to check message flows in your Kafka cluster.

Dump

Dumps the messages from a particular topic.

```
$ python kafkadump.py dump -t demo.crawled_firehose
```

This utility by default consumes from the end of the desired Kafka Topic, and can be useful if left running in helping you debug the messages currently flowing through.

List

Lists all the topics within your cluster.

```
$ python kafkadump.py list
```

Typical Usage

Open three terminals.

Terminal 1:

Monitor your kafka output

```
$ python kafkadump.py dump -t demo.crawled_firehose -p
```

Terminal 2:

Run the Kafka Monitor

```
$ python kafka_monitor.py run
```

Terminal 3:

Feed an item

```
$ python kafka_monitor.py feed '{"url": "http://istresearch.com", "appid":"testapp",
↪"crawlid":"ABC123"}'
2016-01-05 15:14:44,829 [kafka-monitor] INFO: Feeding JSON into demo.incoming
{
```

```
    "url": "http://istresearch.com",
    "crawlid": "ABC123",
    "appid": "testapp"
}
2016-01-05 15:14:44,830 [kafka-monitor] INFO: Successfully fed item to Kafka
```

You should see a log message come through Terminal 2 stating the message was received.

```
2016-01-05 15:14:44,836 [kafka-monitor] INFO: Added crawl to Redis
```

At this point, your Kafka Monitor is working.

If you have a *Crawler* running, you should see the html come through Terminal 1 in a couple of seconds like so:

```
{
    "body": "<body omitted>",
    "crawlid": "ABC123",
    "response_headers": {
        <headers omitted>
    },
    "response_url": "http://istresearch.com",
    "url": "http://istresearch.com",
    "status_code": 200,
    "status_msg": "OK",
    "appid": "testapp",
    "links": [],
    "request_headers": {
        "Accept-Language": "en",
        "Accept-Encoding": "gzip,deflate",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "User-Agent": "Scrapy/1.0.3 (+http://scrapy.org)"
    },
    "attrs": null,
    "timestamp": "2016-01-05T20:14:54.653703"
}
```

If you can see the html result, it means that both your Kafka Monitor and crawlers are up and running!

API

The Kafka Monitor consists of a combination of Plugins that allow for API validation and processing of the object received.

Kafka Topics

Incoming Kafka Topic: `demo.incoming` - The topic to feed properly formatted cluster requests.

Outbound Result Kafka Topics:

- `demo.crawled_firehouse` - A firehose topic of all resulting crawls within the system. Any single page crawled by the Scrapy Cluster is guaranteed to come out this pipe.
- `demo.crawled_<appid>` - A special topic created for unique applications that submit crawl requests. Any application can listen to their own specific crawl results by listening to the the topic created under the `appid` they used to submit the request. These topics are a subset of the crawl firehose data and only contain the results that are applicable to the application who submitted it.

Note: The crawl result `appid` topics are disabled by default, as they create duplicate data within your Kafka application. You can override the *setting* in the Crawler if you wish to turn this on.

- `demo.outbound_firehose` - A firehose topic of all the special info, stop, expire, and stat requests. This topic will have all non-crawl data requested from the cluster.
- `demo.outbound_<appid>` - A special topic for unique applications to read only their action request data. The `appid` used in the submission of the non-crawl API request will be written to both this topic and the firehose topic.

Note: The outbound `appid` topics are disabled by default, as they create duplicate data within Kafka. You can override the *setting* for the Redis Monitor if you wish to turn this on.

Crawl API

The crawl API defines the type of crawl you wish to have the cluster execute. The following properties are available to control the crawling cluster:

Required

- **appid:** The application ID that submitted the crawl request. This should be able to uniquely identify who submitted the crawl request
- **crawlid:** A unique crawl ID to track the executed crawl through the system. Crawl ID's are passed along when a maxdepth > 0 is submitted, so anyone can track all of the results from a given seed url. Crawl ID's also serve as a temporary duplication filter, so the same crawl ID will not continue to recrawl pages it has already seen.
- **url:** The initial seed url to begin the crawl from. This should be a properly formatted full path url from which the crawl will begin from

Optional:

- **spiderid:** The spider to use for the crawl. This feature allows you to chose the spider you wish to execute the crawl from
- **maxdepth:** The depth at which to continue to crawl new links found on pages
- **priority:** The priority of which to given to the url to be crawled. The Spiders will crawl the highest priorities first.
- **allowed_domains:** A list of domains that the crawl should stay within. For example, putting `[ "cnn.com" ]` will only continue to crawl links of that domain.
- **allow_regex:** A list of regular expressions to apply to the links to crawl. Any hits within from any regex will allow that link to be crawled next.
- **deny_regex:** A list of regular expressions that will deny links to be crawled. Any hits from these regular expressions will deny that particular url to be crawled next, as it has precedence over `allow_regex`.
- **deny_extensions:** A list of extensions to deny crawling, defaults to the extensions provided by Scrapy (which are pretty substantial).
- **expires:** A unix timestamp in seconds since epoch for when the crawl should expire from the system and halt. For example, `1423669147` means the crawl will expire when the crawl system machines reach 3:39pm on 02/11/2015. This setting does not account for timezones, so if the machine time is set to EST(-5) and you give a UTC time for three minutes in the future, the crawl will run for 5 hours and 3 mins!

- **useragent:** The header request user agent to fake when scraping the page. If none it defaults to the Scrapy default.
- **cookie:** A cookie string to be used when executing the desired crawl.
- **attrs:** A generic object, allowing an application to pass any type of structured information through the crawl in order to be received on the other side. Useful for applications that would like to pass other data through the crawl.

Examples

Kafka Request:

```
$ python kafka_monitor.py feed '{"url": "http://www.apple.com/", "appid":
↪"testapp", "crawlid":"myapple"}'
```

- Submits a single crawl of the homepage of apple.com

```
$ python kafka_monitor.py feed '{"url": "http://www.dmoz.org/", "appid":
↪"testapp", "crawlid":"abc123", "maxdepth":2, "priority":90}'
```

- Submits a dmoz.org crawl spidering 2 levels deep with a high priority

```
$ python kafka_monitor.py feed '{"url": "http://aol.com/", "appid":"testapp",
↪ "crawlid":"a23bbqwewqe", "maxdepth":3, "allowed_domains":["aol.com"],
↪"expires":1423591888}'
```

- Submits an aol.com crawl that runs for (at the time) 3 minutes with a large depth of 3, but limits the crawlers to only the `aol.com` domain so as to not get lost in the weeds of the internet.

Kafka Response:

```
{
    "body": "<body string omitted>",
    "crawlid": "ABC123",
    "response_url": "http://istresearch.com",
    "url": "http://istresearch.com",
    "status_code": 200,
    "status_msg": "OK",
    "appid": "testapp",
    "links": [],
    "request_headers": {
        "Accept-Language": "en",
        "Accept-Encoding": "gzip,deflate",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;
↪q=0.8",
        "User-Agent": "Scrapy/1.0.3 (+http://scrapy.org)"
    },
    "attrs": null,
    "timestamp": "2016-01-05T20:14:54.653703"
}
```

All of your crawl response objects will come in a well formatted JSON object like the above example.

Action API

The Action API allows for information to be gathered from the current scrape jobs, as well as stopping crawls while they are executing. These commands are executed by the Redis Monitor, and the following properties are available to control.

Required

- **appid:** The application ID that is requesting the action.
- **spiderid:** The spider used for the crawl (in this case, `link`)
- **action:** The action to take place on the crawl. Options are either `info` or `stop`
- **uuid:** A unique identifier to associate with the action request. This is used for tracking purposes by the applications who submit action requests.

Optional:

- **crawlid:** The unique `crawlid` to act upon. Only needed when stopping a crawl or gathering information about a specific crawl.

Examples

Information Action

The `info` action can be conducted in two different ways.

- **Application Info Request**

  Returns application level information about the crawls being conducted. It is a summarization of various `crawlid` statistics

  Kafka Request:

```
$ python kafka_monitor.py feed '{"action": "info", "appid":"testapp
↪", "crawlid":"ABC123", "uuid":"abc12333", "spiderid":"link"}'
```

  Kafka Response:

```
{
    "server_time": 1452094322,
    "uuid": "abc12333",
    "total_pending": 2092,
    "total_domains": 130,
    "total_crawlids": 2,
    "spiderid": "link",
    "appid": "testapp",
    "crawlids": {
        "ist234": {
            "domains": {
                "google.com": {
                    "low_priority": -9,
                    "high_priority": -9,
                    "total": 1
                },
                "twitter.com": {
                    "low_priority": -9,
                    "high_priority": -9,
                    "total": 6
                },
```

```
                <domains ommitted for clarity>
            },
            "distinct_domains": 4,
            "expires": "1452094703",
            "total": 84
        },
        "ABC123": {
            "domains": {
                "ctvnews.ca": {
                    "low_priority": -19,
                    "high_priority": -19,
                    "total": 2
                },
                "quora.com": {
                    "low_priority": -29,
                    "high_priority": -29,
                    "total": 1
                },
                <domains omitted for clarity>
            },
            "distinct_domains": 129,
            "total": 2008
        }
    }
}
```

Here, there were two different `crawlid`'s in the queue for the `link` spider that had the specified `appid`. The json return value is the basic structure seen above that breaks down the different `crawlid`'s into their domains, total, their high/low priority in the queue, and if they have an expiration.

- **Crawl ID Information Request**

  This is a very specific request that is asking to poll a specific `crawlid` in the `link` spider queue. Note that this is very similar to the above request but with one extra parameter.

  Kafka Request:

```
$ python kafka_monitor.py feed '{"action": "info", "appid":"testapp
↪", "crawlid":"ABC123", "uuid":"abc12333", "spiderid":"link"}'
```

  Kafka Response:

```
{
    "server_time": 1452094050,
    "crawlid": "ABC123",
    "total_pending": 582,
    "total_domains": 22,
    "spiderid": "link",
    "appid": "testapp",
    "domains": {
        "duckduckgo.com": {
            "low_priority": -19,
            "high_priority": -19,
            "total": 2
        },
        "wikipedia.org": {
            "low_priority": -19,
            "high_priority": -19,
```

```
            "total": 1
        },
        <domains omitted for clarity>
    },
    "uuid": "abc12333"
}
```

The response to the info request is a simple json object that gives statistics about the crawl in the system, and is very similar to the results for an `appid` request. Here we can see that there were 582 requests in the queue yet to be crawled of all the same priority.

Stop Action

The `stop` action is used to abruptly halt the current crawl job. A request takes the following form:

Kafka Request

```
$ python kafka_monitor.py feed '{"action": "stop", "appid":"testapp",
↪"crawlid":"ist234",  "uuid":"1ist234", "spiderid":"link"}'
```

After the request is processed, only spiders within the cluster currently in progress of downloading a page will continue. All other spiders will not crawl that same `crawlid` past a depth of 0 ever again, and all pending requests will be purged from the queue.

Kafka Response:

```
{
    "total_purged": 2008,
    "server_time": 1452095096,
    "crawlid": "ABC123",
    "uuid": "1ist234",
    "spiderid": "link",
    "appid": "testapp",
    "action": "stop"
}
```

The json response tells the application that the stop request was successfully completed, and states how many requests were purged from the particular queue. **Expire Notification**

An `expire` notification is generated by the Redis Monitor any time an on going crawl is halted because it has exceeded the time it was supposed to stop. A crawl request that includes an `expires` attribute will generate an expire notification when it is stopped by the Redis Monitor.

Kafka Response:

```
{
    "total_expired": 84,
    "server_time": 1452094847,
    "crawlid": "ist234",
    "spiderid": "link",
    "appid": "testapp",
    "action": "expired"
}
```

This notification states that the `crawlid` of "ist234" expired within the system, and that 84 pending requests were removed.

Stats API

The Stats API allows you to gather metrics, health, and general crawl statistics about your cluster. This is not the same as the *Action API*, which allows you to alter or gather information about specific jobs within the cluster.

Required:

- **uuid:** The unique identifier associated with the request. This is useful for tracking purposes by the application who submitted the request.
- **appid:** The application ID that is requesting the action.

Optional:

- **stats:** The type of stats you wish to receive. Defaults to **all**, includes:
  - **all** - Gathers **kafka-monitor**, **redis-monitor**, and **crawler** stats.
  - **kafka-monitor** - Gathers information about the Kafka Monitor
  - **redis-monitor** - Gathers information about the Redis Monitor
  - **crawler** - Gathers information about the crawlers in the cluster. Includes both the **spider**, **machine**, and **queue** information below
  - **spider** - Gathers information about the existing spiders in your cluster
  - **machine** - Gathers information about the different spider machines in your cluster
  - **queue** - Gathers information about the various spider queues within your Redis instance
  - **rest** - Gathers information about the Rest services in your cluster

Stats request results typically have numeric values for dictionary keys, like `900`, `3600`, `86400`, or in the special case `lifetime`. These numbers indicate **rolling time windows** in seconds for processing throughput. So if you see a value like `"3600":14` you can interpret this as *in the last 3600 seconds, the kafka-monitor saw 14 requests"*. In the case of lifetime, it is the total count over the cluster's operation.

Examples

Kafka Monitor

Stats Request:

The Kafka Monitor stats gives rolling time windows of incoming requests, failed requests, and individual plugin processing. An example request and response is below.

```
$ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"cdefgh1", "stats
↪":"kafka-monitor"}'
```

Response from Kafka:

```
{
    "server_time": 1452102876,
    "uuid": "cdefghi",
    "nodes": {
        "Madisons-MacBook-Pro-2.local": [
            "e2be29329410",
            "60295fece216"
        ]
    },
    "plugins": {
```

```
        "ActionHandler": {
            "604800": 11,
            "lifetime": 17,
            "43200": 11,
            "86400": 11,
            "21600": 11
        },
        "ScraperHandler": {
            "604800": 10,
            "lifetime": 18,
            "43200": 5,
            "86400": 10,
            "21600": 5
        },
        "StatsHandler": {
            "900": 2,
            "3600": 2,
            "604800": 3,
            "43200": 3,
            "lifetime": 10,
            "86400": 3,
            "21600": 3
        }
    },
    "appid": "testapp",
    "fail": {
        "604800": 4,
        "lifetime": 7,
        "43200": 4,
        "86400": 4,
        "21600": 4
    },
    "stats": "kafka-monitor",
    "total": {
        "900": 2,
        "3600": 2,
        "604800": 28,
        "43200": 23,
        "lifetime": 51,
        "86400": 28,
        "21600": 23
    }
}
```


Here we can see the various totals broken down by total, fail, and each loaded plugin.

Crawler

A Crawler stats request can consist of rolling time windows of spider stats, machine stats, queue stats, or all of them combined. The following examples serve to illustrate this.

- **Spider**

  Spider stats requests gather information about the crawl results from your cluster, and other information about the number of spiders running.

  Kafka Request:

```
$ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"hij1",
↪"stats":"spider"}'
```

Kafka Response:

```
{
    "stats": "spider",
    "appid": "testapp",
    "server_time": 1452103525,
    "uuid": "hij1",
    "spiders": {
        "unique_spider_count": 1,
        "total_spider_count": 2,
        "link": {
            "count": 2,
            "200": {
                "604800": 44,
                "lifetime": 61,
                "43200": 39,
                "86400": 44,
                "21600": 39
            },
            "504": {
                "lifetime": 4
            }
        }
    }
}
```

Here, we see that there have been many 200 responses within our collection time periods, but the 504 response rolling time window has dropped off, leaving only the lifetime stat remaining. We can also see that we are currently running 2 `link` spiders in the cluster.

If there were different types of spiders running, we could also see their stats based off of the key within the spider dictionary.

- **Machine**

  Machine stats give information about the crawl machines your spiders are running on. They aggregate the total crawl results for their host, and disregard the spider type(s) running on the machine.

  Kafka Request:

```
$ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"hij12",
↪ "stats":"machine"}'
```

  Kafka Response:

```
{
    "stats": "machine",
    "server_time": 1452104037,
    "uuid": "hij12",
    "machines": {
        "count": 1,
        "Madisons-MacBook-Pro-2.local": {
            "200": {
                "604800": 44,
                "lifetime": 61,
                "43200": 39,
```

```
                "86400": 44,
                "21600": 39
            },
            "504": {
                "lifetime": 4
            }
        }
    },
    "appid": "testapp"
}
```


You can see we only have a local machine running those 2 crawlers, but if we had more machines their aggregated stats would be displayed in the same manner.

- **Queue**

  Queue stats give you information about the current queue backlog for each spider, broken down by spider type and domain.

  Kafka Request:

  ```
  $ python kafka_monitor.py feed '{"stats":"queue", "appid":"test",
  ↪"uuid":"1234567890"}'
  ```

  Kafka Response:


```
{
    "stats": "queue",
    "queues": {
        "queue_link": {
            "domains": [
                {
                    "domain": "istresearch.com",
                    "backlog": 229
                }
            ],
            "spider_backlog": 229,
            "num_domains": 1
        },
        "total_backlog": 229
    },
    "server_time": 1461700519,
    "uuid": "1234567890",
    "appid": "test"
}
```


  Here, we have only one active spider queue with one domain. If there were more domains or more spiders that data would be displayed in the same format.

- **Crawler**

  The crawler stat is a combination of the **machine**, **spider**, and **queue** requests.

  Kafka Request:

  ```
  $ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"hij1",
  ↪"stats":"crawler"}'
  ```

  Kafka Response:

```
{
    "stats": "crawler",
    "uuid": "hij1",
    "spiders": {
        "unique_spider_count": 1,
        "total_spider_count": 2,
        "link": {
            "count": 2,
            "200": {
                "604800": 44,
                "lifetime": 61,
                "43200": 39,
                "86400": 44,
                "21600": 39
            },
            "504": {
                "lifetime": 4
            }
        }
    },
    "appid": "testapp",
    "server_time": 1452104450,
    "machines": {
        "count": 1,
        "Madisons-MacBook-Pro-2.local": {
            "200": {
                "604800": 44,
                "lifetime": 61,
                "43200": 39,
                "86400": 44,
                "21600": 39
            },
            "504": {
                "lifetime": 4
            }
        }
    },
    "queues": {
        "queue_link": {
            "domains": [
                {
                    "domain": "istresearch.com",
                    "backlog": 229
                }
            ],
            "spider_backlog": 229,
            "num_domains": 1
        },
        "total_backlog": 229
    }
}
```


There is no new data returned in the **crawler** request, just an aggregation.

Redis Monitor

The Redis Monitor stat request returns rolling time window statistics about totals, failures, and plugins like the Kafka Monitor stat requests above.

Stats Request:

```
$ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"2hij1", "stats":
↪"redis-monitor"}'
```

Response from Kafka:


```
{
    "stats": "redis-monitor",
    "uuid": "2hij1",
    "nodes": {
        "Madisons-MacBook-Pro-2.local": [
            "918145625a1e"
        ]
    },
    "plugins": {
        "ExpireMonitor": {
            "604800": 2,
            "lifetime": 2,
            "43200": 2,
            "86400": 2,
            "21600": 2
        },
        "StopMonitor": {
            "604800": 5,
            "lifetime": 6,
            "43200": 5,
            "86400": 5,
            "21600": 5
        },
        "StatsMonitor": {
            "900": 4,
            "3600": 8,
            "604800": 9,
            "43200": 9,
            "lifetime": 16,
            "86400": 9,
            "21600": 9
        },
        "InfoMonitor": {
            "604800": 6,
            "lifetime": 11,
            "43200": 6,
            "86400": 6,
            "21600": 6
        }
    },
    "appid": "testapp",
    "server_time": 1452104685,
    "total": {
        "900": 4,
        "3600": 8,
        "604800": 22,
        "43200": 22,
        "lifetime": 35,
        "86400": 22,
        "21600": 22
    }
}
```

In the above response, note that the `fail` key is omitted because there have been no failures in processing the requests to the redis monitor. All other plugins and totals are represented in the same format as usual.

Rest

The Rest stat request returns some basic information about the number of Rest services running within your cluster.

Stats Request:

```
$ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"2hij1", "stats":
↪"rest"}'
```

Response from Kafka:

```
{
  "data": {
    "appid": "testapp",
    "nodes": {
      "scdev": [
        "c4ec35bf9c1a"
      ]
    },
    "server_time": 1489343194,
    "stats": "rest",
    "uuid": "2hij1"
  },
  "error": null,
  "status": "SUCCESS"
}
```

The response above shows the number of unique nodes running on each machine. Here, we see only one Rest service is running on the host `scdev`.

All

The All stat request is an aggregation of the **kafka-monitor**, **crawler**, and **redis-monitor** stat requests. It does not contain any new information that the API does not already provide.

Kafka Request:

```
$ python kafka_monitor.py feed '{"appid":"testapp", "uuid":"hij3", "stats":
↪"all"}'
```

Kafka Response:

```
{
    "kafka-monitor": {
        "nodes": {
            "Madisons-MacBook-Pro-2.local": [
                "e2be29329410",
                "60295fece216"
            ]
        },
        "fail": {
            "604800": 4,
            "lifetime": 7,
            "43200": 4,
            "86400": 4,
            "21600": 4
```

```
        },
        "total": {
            "900": 3,
            "3600": 9,
            "604800": 35,
            "43200": 30,
            "lifetime": 58,
            "86400": 35,
            "21600": 30
        },
        "plugins": {
            "ActionHandler": {
                "604800": 11,
                "lifetime": 17,
                "43200": 11,
                "86400": 11,
                "21600": 11
            },
            "ScraperHandler": {
                "604800": 10,
                "lifetime": 18,
                "43200": 5,
                "86400": 10,
                "21600": 5
            },
            "StatsHandler": {
                "900": 3,
                "3600": 9,
                "604800": 10,
                "43200": 10,
                "lifetime": 17,
                "86400": 10,
                "21600": 10
            }
        }
    },
    "stats": "all",
    "uuid": "hij3",
    "redis-monitor": {
        "nodes": {
            "Madisons-MacBook-Pro-2.local": [
                "918145625a1e"
            ]
        },
        "total": {
            "900": 3,
            "3600": 9,
            "604800": 23,
            "43200": 23,
            "lifetime": 36,
            "86400": 23,
            "21600": 23
        },
        "plugins": {
            "ExpireMonitor": {
                "604800": 2,
                "lifetime": 2,
                "43200": 2,
```

```
                "86400": 2,
                "21600": 2
            },
            "StopMonitor": {
                "604800": 5,
                "lifetime": 6,
                "43200": 5,
                "86400": 5,
                "21600": 5
            },
            "StatsMonitor": {
                "900": 3,
                "3600": 9,
                "604800": 10,
                "43200": 10,
                "lifetime": 17,
                "86400": 10,
                "21600": 10
            },
            "InfoMonitor": {
                "604800": 6,
                "lifetime": 11,
                "43200": 6,
                "86400": 6,
                "21600": 6
            }
        }
    },
    "rest": {
        "nodes": {
            "Madisons-MacBook-Pro-2.local": [
                "c4ec35bf9c1a"
            ]
        }
    },
    "appid": "testapp",
    "server_time": 1452105176,
    "crawler": {
        "spiders": {
            "unique_spider_count": 1,
            "total_spider_count": 2,
            "link": {
                "count": 2,
                "200": {
                    "604800": 44,
                    "lifetime": 61,
                    "43200": 39,
                    "86400": 44,
                    "21600": 39
                },
                "504": {
                    "lifetime": 4
                }
            }
        },
        "machines": {
            "count": 1,
            "Madisons-MacBook-Pro-2.local": {
```

```
                "200": {
                    "604800": 44,
                    "lifetime": 61,
                    "43200": 39,
                    "86400": 44,
                    "21600": 39
                },
                "504": {
                    "lifetime": 4
                }
            }
        },
        "queues": {
            "queue_link": {
                "domains": [
                    {
                        "domain": "istresearch.com",
                        "backlog": 229
                    }
                ],
                "spider_backlog": 229,
                "num_domains": 1
            },
            "total_backlog": 229
        }
    }
}
```


Zookeeper API

The Zookeeper API allows you to update and remove cluster wide blacklists or crawl rates for domains. A **domain** based update will apply a cluster wide throttle against the domain, and a **blacklist** update will apply a cluster wide ban on crawling that domain. Any blacklist domain or crawl setting will automatically be applied to all currently running spiders.

Required:

- **uuid:** The unique identifier associated with the request. This is useful for tracking purposes by the application who submitted the request.
- **appid:** The application ID that is requesting the action.
- **action:** The type of action you wish to apply. May be any of the following:
  - **domain-update** - Update or add a domain specific throttle (requires both **hits** and **window** below)
  - **domain-remove** - Removes a cluster wide domain throttle from Zookeeper
  - **blacklist-update** - Completely ban a domain from being crawled by the cluster
  - **blacklist-remove** - Remove a crawl ban within the cluster
- **domain:** The domain to apply the **action** to. This must be the same as the domain portion of the queue key generated in Redis, like `ebay.com`.

Optional:

- **hits:** The number of hits allowed for the domain within the time window
- **window:** The number of seconds the hits is applied to

- **scale:** A scalar between 0 and 1 to apply the number of hits to the domain.

Note: For more information on controlling the Crawlers, please see *here*

Examples

Domain Update

Zookeeper Request:

Updates or adds the domain specific configuration

```
$ python kafka_monitor.py feed '{"uuid":"abc123", "appid":"madisonTest",
↪"action":"domain-update", "domain":"dmoz.org", "hits":60, "window":60,
↪"scale":0.9}'
```

Response from Kafka:

```
{
    "action": "domain-update",
    "domain": "dmoz.org",
    "server_time": 1464402128,
    "uuid": "abc123",
    "appid": "madisonTest"
}
```

The response reiterates what has been done to the cluster wide settings.

Domain Remove

Zookeeper Request:

Removes the domain specific configuration

```
$ python kafka_monitor.py feed '{"uuid":"abc123", "appid":"madisonTest",
↪"action":"domain-remove", "domain":"dmoz.org"}'
```

Response from Kafka:

```
{
    "action": "domain-remove",
    "domain": "dmoz.org",
    "server_time": 1464402146,
    "uuid": "abc123",
    "appid": "madisonTest"
}
```

The response reiterates what has been done to the cluster wide settings.

Blacklist Update

Zookeeper Request:

Updates or adds to the cluster blacklist

```
$ python kafka_monitor.py feed '{"uuid":"abc123", "appid":"madisonTest",
↪"action":"blacklist-update", "domain":"ebay.com"}'
```

Response from Kafka:

```
{
    "action": "blacklist-update",
    "domain": "ebay.com",
    "server_time": 1464402173,
    "uuid": "abc123",
    "appid": "madisonTest"
}
```

The response reiterates what has been done to the cluster wide settings.

Blacklist Remove

Zookeeper Request:

Removes the blacklist from the cluster, allowing it to revert back to normal operation on that domain

```
$ python kafka_monitor.py feed '{"uuid":"abc123", "appid":"madisonTest",
↪"action":"blacklist-remove", "domain":"ebay.com"}'
```

Response from Kafka:

```
{
    "action": "blacklist-remove",
    "domain": "ebay.com",
    "server_time": 1464402160,
    "uuid": "abc123",
    "appid": "madisonTest"
}
```

The response reiterates what has been done to the cluster wide settings.

Plugins

Plugins serve as a way to validate and process incoming JSON to the Kafka Monitor. Each plugin consists of the JSON Schema it will validate against, and the resulting code to process a valid object.

The Kafka Monitor loads the desired *Plugins* in the order specified in the settings file, and will evaluate them starting with the lowest ranked plugin.

Each plugin consists of a class inherited from the base plugin class, and one or two python methods to accommodate setup and processing.

Default Plugins

By default, plugins live in the `plugins` directory within the Kafka Monitor. The following plugins come with the Kafka Monitor.

Action Handler

The `action_handler.py` and the `action_schema.json` create an actions API that accepts information and stop requests about specific crawls within Scrapy Cluster.

For more information please see the *Action API*.

Scraper Handler

The `scraper_handler.py` and the `scraper_schema.json` files combine to create an API that allows for crawl requests to be made to the Kafka Monitor.

For more information about the API, please refer to the documentation *here*.

Stats Handler

The `stats_handler.py` and the `stats_schema.json` files create the validating and processing API for statistics gathering.

For more information please refer to the *Stats API*.

Zookeeper Handler

The `zookeeper_handler.py` and the `zookeeper_schema.json` files enable the ability to update the crawl domain blacklist.

For more information please see the *Zookeeper API*

Extension

Creating your own Plugin for the Kafka Monitor allows you to customize your own Kafka API's or processing python scripts to allow new functionality to be added to the Kafka Monitor. You will need two new files in the `plugins` directory, a **json** file for schema validation, and a **python** plugin file to define how you would like to process valid incoming objects.

The following python code template should be used when creating a new plugin:

plugins/new_plugin.py

```
from base_handler import BaseHandler


class NewPlugin(BaseHandler):

    schema = "new_schema.json"

    def setup(self, settings):
        '''
        Setup your plugin here
        '''
        pass

    def handle(self, dict):
        '''
        Processes a valid API request

        @param dict: a valid dictionary object
        '''
        pass
```

The plugin class should inherit from the BaseHandler plugin class, which will allow easy integration into the plugin framework. The `setup()` method is passed a dictionary created from the settings loaded from your local and default settings files. You should set up connections or other variables here to be used in your handle method.

The `handle()` method is passed a dictionary object when a valid object comes into the Kafka Monitor. It will **not** receive invalid objects, so you can assume that any object passed into the function is valid according to your json schema file defined in the `schema` variable at the top of the class.

The JSON Schema defines the type of objects you would like to to process. Below is an example:

plugins/new_schema.json

```
{
    "type": "object",
    "properties": {
        "uuid": {
            "type": "string",
            "minLength": 1,
            "maxLength": 40
        },
        "my_string": {
            "type": "string",
            "minLength": 1,
            "maxLength": 100
        }
    },
    "required": [
        "uuid",
        "my_string"
    ]
}
```

In the `handle()` method, you would receive objects that have both a `uuid` field and a `my_string` field. You are now free to do any additional processing, storage, or manipulation of the object within your plugin! You now should add it to your `localsettings.py` file.

localsettings.py

```
PLUGINS = {
    'plugins.new_plugin.NewPlugin': 400,
}
```

You have now told the Kafka Monitor to load not only the default plugins, but your new plugin as well with a rank of 400. If you restart your Kafka Monitor the plugin will be loaded.

Additional Info

Every Kafka Monitor plugin is provided a Scrapy Cluster logger, under the variable name `self.logger`. You can use this logger to generate debug, info, warnings, or any other log output you need to help gather information from your plugin. This is the same logger that the Kafka Monitor uses, so your desired settings will be preserved.

Settings

This page covers all of the various settings contained within the Kafka Monitor. The sections are broken down by functional component.

Core

SLEEP_TIME

Default: `0.01`

The number of seconds the main process will sleep between checking for new crawl requests to take care of.

HEARTBEAT_TIMEOUT

Default: `120`

The amount of time the statistics key the Kafka Monitor instance lives to self identify to the rest of the cluster. Used for retrieving stats about the number of Kafka Monitor instances currently running.

Redis

REDIS_HOST

Default: `'localhost'`

The Redis host.

REDIS_PORT

Default: `6379`

The port to use when connecting to the `REDIS_HOST`.

REDIS_DB

Default: `0`

The Redis database to use when connecting to the `REDIS_HOST`.

Kafka

KAFKA_HOSTS

Default: `'localhost:9092'`

The Kafka host. May have multiple hosts separated by commas within the single string like `'h1:9092,h2:9092'`.

KAFKA_INCOMING_TOPIC

Default: `'demo.incoming'`

The Kafka Topic to read API requests from.

KAFKA_GROUP

Default: `'demo-group'`

The Kafka Group to identify the consumer.

KAFKA_FEED_TIMEOUT

Default: `5`

How long to wait (in seconds) before timing out when trying to feed a JSON string into the `KAFKA_INCOMING_TOPIC`

KAFKA_CONSUMER_AUTO_OFFSET_RESET

Default: `'earliest'`

When the Kafka Consumer encounters and unexpected error, move the consumer offset to the 'latest' new message, or the 'earliest' available.

KAFKA_CONSUMER_TIMEOUT

Default: `50`

Time in ms spent to wait for a new message during a `feed` call that expects a response from the Redis Monitor

KAFKA_CONSUMER_COMMIT_INTERVAL_MS

Default: `5000`

How often to commit Kafka Consumer offsets to the Kafka Cluster

KAFKA_CONSUMER_AUTO_COMMIT_ENABLE

Default: `True`

Automatically commit Kafka Consumer offsets.

KAFKA_CONSUMER_FETCH_MESSAGE_MAX_BYTES

Default: `10 * 1024 * 1024`

The maximum size of a single message to be consumed by the Kafka Consumer. Defaults to 10 MB

KAFKA_PRODUCER_BATCH_LINGER_MS

Default: `25`

The time to wait between batching multiple requests into a single one sent to the Kafka cluster.

KAFKA_PRODUCER_BUFFER_BYTES

Default: `4 * 1024 * 1024`

The size of the TCP send buffer when transmitting data to Kafka

Plugins

PLUGIN_DIR

Default: `'plugins/'`

The folder containing all of the Kafka Monitor plugins. **PLUGINS**

Default:

```
{
    'plugins.scraper_handler.ScraperHandler': 100,
    'plugins.action_handler.ActionHandler': 200,
    'plugins.stats_handler.StatsHandler': 300,
    'plugins.zookeeper_handler.ZookeeperHandler': 400,
}
```

The default plugins loaded for the Kafka Monitor. The syntax for this dictionary of settings is `'<folder>.<file>.<class_name>': <rank>`. Where lower ranked plugin API's are validated first.

Logging

LOGGER_NAME

Default: `'kafka-monitor'`

The logger name.

LOG_DIR

Default: `'logs'`

The directory to write logs into. Only applicable when `LOG_STDOUT` is set to `False`.

LOG_FILE

Default: `'kafka_monitor.log'`

The file to write the logs into. When this file rolls it will have `.1` or `.2` appended to the file name. Only applicable when `LOG_STDOUT` is set to `False`.

LOG_MAX_BYTES

Default: `10 * 1024 * 1024`

The maximum number of bytes to keep in the file based log before it is rolled.

LOG_BACKUPS

Default: `5`

The number of rolled file logs to keep before data is discarded. A setting of `5` here means that there will be one main log and five rolled logs on the system, generating six log files total.

LOG_STDOUT

Default: `True`

Log to standard out. If set to `False`, will write logs to the file given by the `LOG_DIR/LOG_FILE`

LOG_JSON

Default: `False`

Log messages will be written in JSON instead of standard text messages.

LOG_LEVEL

Default: `'INFO'`

The log level designated to the logger. Will write all logs of a certain level and higher.

Note: More information about logging can be found in the utilities *Log Factory* documentation.

Stats

STATS_TOTAL

Default: `True`

Calculate total receive and fail stats for the Kafka Monitor.

STATS_PLUGINS

Default: `True`

Calculate total receive and fail stats for each individual plugin within the Kafka Monitor.

STATS_CYCLE

Default: `5`

How often to check for expired keys and to roll the time window when doing stats collection.

STATS_DUMP

Default: `60`

Dump stats to the logger every X seconds. If set to `0` will not dump statistics.

STATS_TIMES

Default:

```
[
    'SECONDS_15_MINUTE',
    'SECONDS_1_HOUR',
    'SECONDS_6_HOUR',
    'SECONDS_12_HOUR',
    'SECONDS_1_DAY',
    'SECONDS_1_WEEK',
]
```

Rolling time window settings for statistics collection, the above settings indicate stats will be collected for the past 15 minutes, the past hour, the past 6 hours, etc.

Note: For more information about stats collection, please see the *Stats Collector* documentation.

Design Learn about the design considerations for the Kafka Monitor

Quick Start How to use and run the Kafka Monitor

API The default Kafka API the comes with Scrapy Cluster

Plugins Gives an overview of the different plugin components within the Kafka Monitor, and how to make your own.

Settings Explains all of the settings used by the Kafka Monitor

CHAPTER 3

Crawler

These documents give you an overview of the core Scrapy Cluster Crawler and how to work within both Scrapy and Scrapy Cluster to fit your needs.

Design

The Scrapy Cluster allows for multiple concurrent spiders located on different machines to coordinate their crawling efforts against a submitted crawl job. The crawl queue is managed by Redis, and each spider utilizes a modified Scrapy Scheduler to pull from the redis queue.

After the page has been successfully crawled by the spider, it is yielded to the item pipeline for further processing. If the page was not successfully crawled, the retry middleware included with Scrapy Cluster will resubmit the page back to the queue, to potentially be tried by another spider.

A generic `link` spider has been provided as a starting point into cluster based crawling. The link spider simply crawls all links found on the page to the depth specified by the Kafka API request.

Spiders

The core design of the provided link spider is that it tries to be simple in concept and easy to extend into further applications. Scrapy itself is a very powerful and extendable crawling framework, and this crawling project utilizes a unique combination of extensions and modifications to try to meet a new cluster based crawling approach.

Every spider in the cluster is stand alone, meaning that it can function as a complete Scrapy spider in the cluster without any other spiders or processes running on that machine. As long as the spider can connect to Kafka and Redis then you will have a complete working 'cluster'. The power comes from when you stand multiple crawlers up on one machine, or spread them across different machines with different IP addresses. Now you get the processing power behind a Scrapy spider *and* the benefit of redundancy and parallel computations on your crawl job.

Each crawl job that is submitted to the cluster is given both a priority and is split into it's respective domain based queue. For every subsequent level deep in the crawl that priority decreases by 10, and further requests to crawl pages are put into their own domain queue. This allows you to manage not only the crawl job, but how hard your cluster hits a particular domain of interest.

An example diagram of the breadth first crawling is shown below:

As you can see above, the initial seed url generates 4 new links. Scrapy Cluster uses a Redis priority based queue, so the spiders continue to attempt to pop from the highest priority crawl request for each domain. New links found upon subsequent requests are decreased in priority, and then put back into their respective domain based queues. This allows for the equal priority links to be crawled first.

When a spider encounters a link it has already seen, the duplication filter based on the request's `crawlid` will filter it out. The spiders will continue to traverse the resulting graph generated until they have reached either their maximum link depth or have exhausted all possible urls.

When a crawl is submitted to the cluster, the request it placed into its own specialized queue based on that particular spider type and *cluster throttle strategy*. This allows your cluster to crawl at the desired speed you would like to go to, even though your Scrapy Spiders may be on completely different machines!

The diagram above shows that you can have multiple instances of different spiders running on the same machines, that are all coordinated through a single redis instance. Since every spider knows its own type, it uses the request scheduler to pull new requests from the different domain queues with its name. New requests that come in from Kafka, and requests generated by finding new links while spidering are all sent back into the same priority queue to be crawled.

For example, you can have an ongoing crawl that submits high depth crawl jobs beginning at a priority of 70. These crawls take a lot of resources and will continue to run, but if a user decides to submit a crawl of priority 90, they will immediately get their crawl result returned to them.

Overall, the designs considered for the core scraping component of Scrapy Cluster enable:

- Extendable crawlers thanks to Scrapy
- Distributed crawl efforts across arbitrary machines
- Multiple spider processes capable of independent logic
- Coordinated, lossless frontier expansion of the crawl job
- Distributed throttling and coordination so your scraping cluster does not overload any particular website
- Ability to pass crawl jobs to other spiders within the cluster

Components

This section explains the individual files located within the Scrapy crawler project.

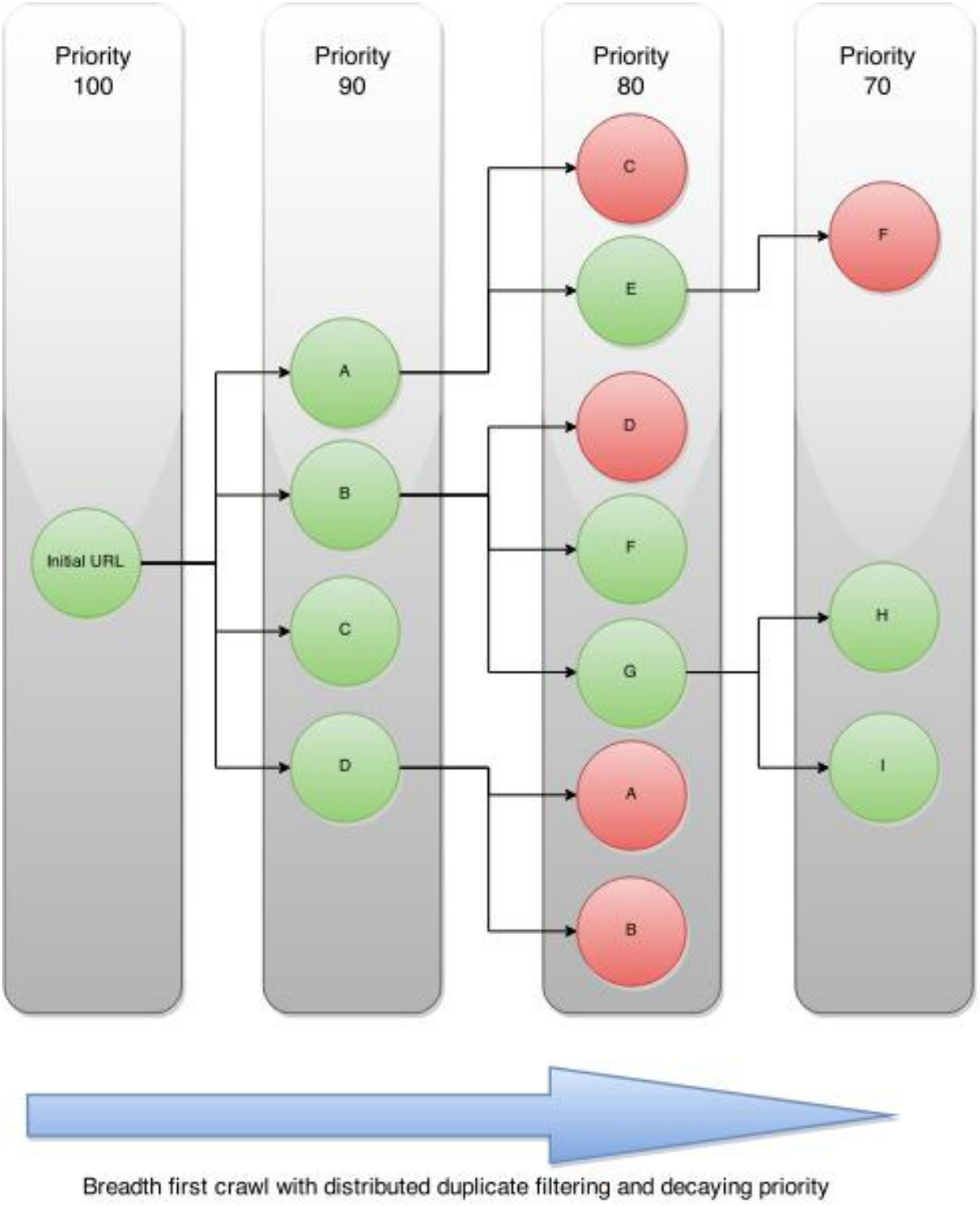
custom_cookies.py

Enables long lived spiders to not cache the cookies received in the Spider cookie jar, yet pass cookies in all Requests. This prevents the spiders from caching response cookies and making subsequent requests with those cookies for a completely different crawl job.

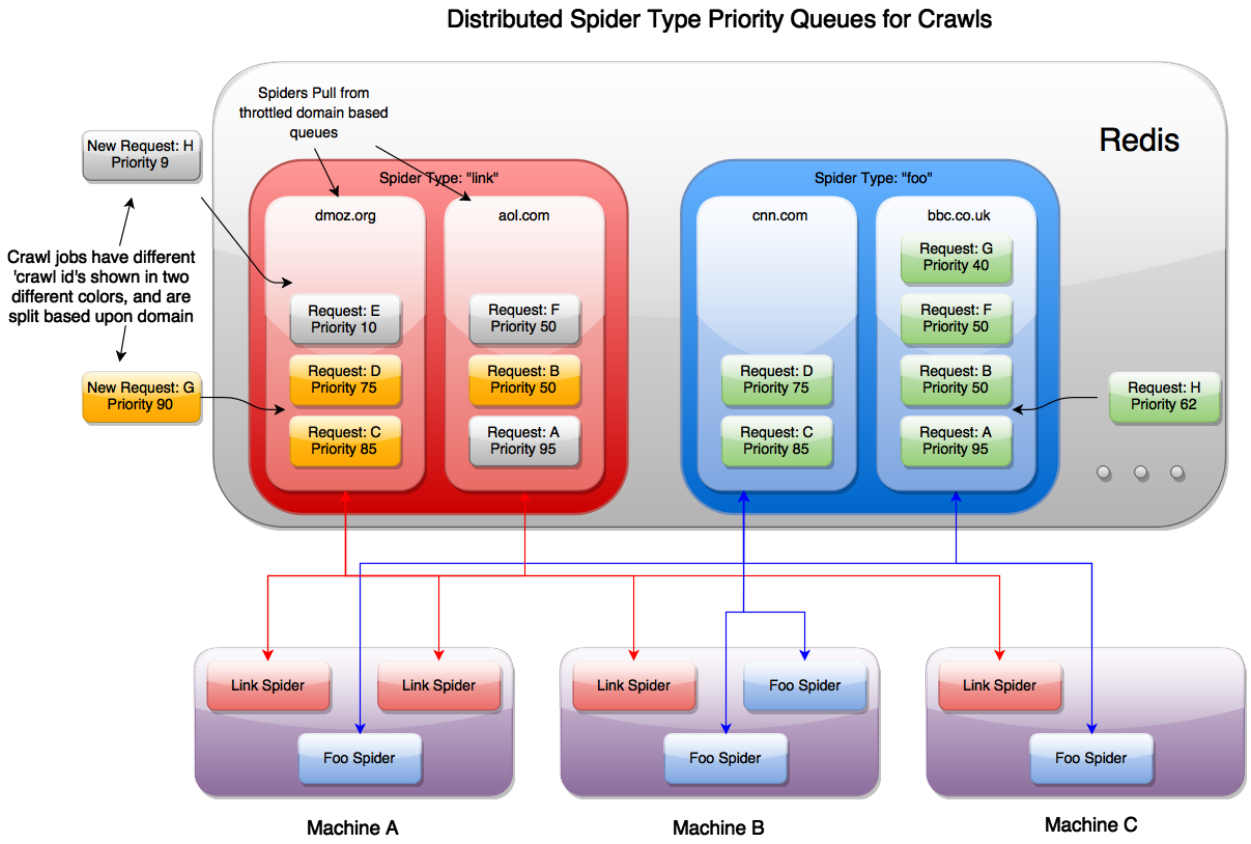
distributed_scheduler.py

A Scrapy Scheduler that tries to find new requests from the Redis queues associated with the spider's `name`. Since we are using a link spider, the domain queues it will look for will be called `link:<domain>:queue`.

The scheduler has all of the same overridden functions you would expect, except for `has_pending_requests()`. In our tests we found inconsistencies in Scrapy when returning `True` when grabbing a new item from within that function or letting the item be returned from `next_request`. For now we return `False` so that the spider goes back to idle, but it is worth investigating more in the future.



Breadth first crawl with distributed duplicate filtering and decaying priority



items.py

Holds a single basic item yielded by the crawlers to the item pipeline. The RawResponseItem returns other metadata associated with the crawl including:

- response url
- status code
- status message
- headers
- body text
- links found
- passed through attributes

log_retry_middleware.py

Logs and collects statistics about the Spider receiving 504 timeout status codes. This allows you to see in the Scrapy Cluster logs when your Spiders are having trouble connecting to the desired web pages.

meta_passthrough_middleware.py

Ensures the minimum amount of metadata information from the response is passed through to subsequent requests for the distribtued scheduler to work.

pipelines.py

The pipelines file is a basic Scrapy Item Pipeline with a three classes contained within. The pipeline classes log to ensure we received the item, it was sent (successfully or not) to Kafka, and then to log that the item's result. The pipeline also checks to make sure that the Kafka topic exists before sending the message to it.

redis_dupefilter.py

An extremely basic class that serves as a crawl link duplication filter utilizing a Redis Set. This allows two important things:

- Any unique `crawlid` will not recrawl a url it has already seen
- New crawl requests with a **different** `crawlid` can crawl those same links, without being effected by other crawl duplication filters

This allows for a crawl job over a variety of links to not waste resources by crawling the same things. If you would like to recrawl those same urls, simply submit the same url with a different crawl identifier to the API. If you would like to continue to expand your crawl frontier, submit a crawl with the same identifier.

Note: If you continue to submit the same `crawlid` and none of the urls have changed, the crawl prematurely stop because it found zero new links to spider.

---

redis_retry_middleware.py

This class is a Scrapy Downloader Middleware that catches 504 timeout exceptions thrown by the spider. These exceptions are handled differently from other status codes because the spider never even got to the url, so the downloader throws an error.

The url is thrown back into the cluster queue at a lower priority so the cluster can try all other higher priority urls before the one that failed. After a certain amount of retries, the url is given up on and discarded from the queue.

redis_stats_middleware.py

A Spider middleware that allows the spider to record Scrapy Cluster statistics about crawl response codes within Redis. This middleware grabs the response code from the Response object and increments a *StatsCollector* counter.

settings.py

Holds both Scrapy and *Scrapy Cluster* settings. To override these variables please create a `localsettings.py` file, and add your variables in there.

spiders/link_spider.py

An introduction into generic link crawling, the LinkSpider inherits from the base class RedisSpider to take advantage of a simple html content parse. The spider's main purpose is to generate two things:

1. Generate more urls to crawl, found by grabbing all the links on the page
2. Generate a `RawResponseItem` to be processed by the item pipeline.

These two things enable generic depth based crawling, and the majority of the code used within the class is to generate those two objects. For a single page this spider might yield 100 urls to crawl and 1 html item to be processed by the Kafka pipeline.

Note: We do not need to use the duplication filter here, as the scheduler handles that for us. All this spider cares about is generating the two items listed above.

---

spiders/lxmlhtml.py

This is actually a custom version of the Scrapy `LxmlParserLinkExtractor` but with one slight alteration. We do not want Scrapy to throw link extraction parsing errors when encountering a site with malformed html or bad encoding, so we changed it to ignore errors instead of complaining. This allows for the continued processing of the scraped page all the way through the pipeline even if there are utf encoding problems.

spiders/redis_spider.py

A base class that extends the default Scrapy Spider so we can crawl continuously in cluster mode. All you need to do is implement the `parse` method and everything else is taken care of behind the scenes.

Note: There is a method within this class called `reconstruct_headers()` that is very important you take advantage of! The issue we ran into was that we were dropping data in our headers fields when encoding the item into json. The Scrapy shell didn't see this issue, print statements couldn't find it, but it boiled down to the python list being treated as a single element. We think this may be a formal defect in Python 2.7 but have not made an issue yet as the bug needs much more testing.

---

spiders/wandering_spider.py

Another example spider for crawling within Scrapy Cluster. This spider randomly hops around one link at a time. You can read more about how this spider was created *here*

Quick Start

This is a complete Scrapy crawling project located in `crawler/`.

First, create a `crawling/localsettings.py` file to track your custom settings. You can override any setting in the normal `settings.py` file, and a typical file may look like the following:

```
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
ZOOKEEPER_HOSTS = 'scdev:2181'
SC_LOG_LEVEL = 'DEBUG'
```

Scrapy

Then run the Scrapy spider like normal:

```
scrapy runspider crawling/spiders/link_spider.py
```

To run multiple crawlers, simply run in the background across X number of machines. Because the crawlers coordinate their efforts through Redis, any one crawler can be brought up/down on any machine in order to add crawling capacity.

Typical Usage

Open four terminals.

Terminal 1:

Monitor your kafka output

```
$ python kafkadump.py dump -t demo.crawled_firehose -p
```

Terminal 2:

Run the Kafka Monitor

```
$ python kafka_monitor.py run
```

Note: This assumes you have your *Kafka Monitor* already working.

Terminal 3:

Run your Scrapy spider

```
scrapy runspider crawling/spiders/link_spider.py
```

Terminal 4:

Feed an item

```
$ python kafka_monitor.py feed '{"url": "http://dmoztools.net/", "appid":"testapp",
↪"crawlid":"09876abc"}'
2016-01-21 23:22:23,830 [kafka-monitor] INFO: Feeding JSON into demo.incoming
{
    "url": "http://dmoztools.net/",
    "crawlid": "09876abc",
    "appid": "testapp"
}
2016-01-21 23:22:23,832 [kafka-monitor] INFO: Successfully fed item to Kafka
```

You should see a log message come through Terminal 2 stating the message was received.

```
2016-01-21 23:22:23,859 [kafka-monitor] INFO: Added crawl to Redis
```

Next, you should see your Spider in Terminal 3 state the crawl was successful.

```
2016-01-21 23:22:35,976 [scrapy-cluster] INFO: Scraped page
2016-01-21 23:22:35,979 [scrapy-cluster] INFO: Sent page to Kafka
```

At this point, your Crawler is up and running!

If you are still listening to the Kafka Topic in Terminal 1, the following should come through.

```
{
    "body": "<body ommitted>",
    "crawlid": "09876abc",
    "response_headers": {
        <headers omitted>
    },
    "response_url": "http://dmoztools.net/",
    "url": "http://dmoztools.net/",
    "status_code": 200,
    "status_msg": "OK",
    "appid": "testapp",
    "links": [],
    "request_headers": {
        "Accept-Language": "en",
        "Accept-Encoding": "gzip,deflate",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "User-Agent": "Scrapy/1.0.4 (+http://scrapy.org)"
    },
    "attrs": null,
    "timestamp": "2016-01-22T04:22:35.976672"
}
```


This completes the crawl submission, execution, and receival of the requested data.

Controlling

Scrapy Cluster requires coordination between the different crawling machines in order to ensure maximum content throughput while enabling the cluster manager to control how fast their machines hit different websites.

Scrapy Cluster comes with two major strategies for controlling how fast your pool of spiders hit different domains. This is determined by spider type and/or IP Address, but both act upon the different Domain Queues.

Domain Queues

Registered domain names are what are commonly used to reference a general website. Some examples are `google.com`, `wikipedia.org`, `amazon.com`, `scrapy.org`... you get the idea.

Scrapy Cluster allows you to control how fast your cluster hits each of these domains, without interfering with other domains. Each domain is independently coordinated and throttled by the collective Spiders running, allowing both fine tuned and generalized control of how fast your spiders react to new domains and how fast they should crawl them.

General Domain Settings

The following settings will help you control how spiders generally behave when dealing with different domains.

QUEUE_WINDOW - A rolling time window of the number of seconds your cluster should remember domain specific hits

QUEUE_HITS - Controls how many domain hits are allowed within your `QUEUE_WINDOW`

Putting these two together, this means that you can control X number of `QUEUE_HITS` in a y `QUEUE_WINDOW`. So **5 hits every 60** seconds, or **10 hits every 30 seconds**. This setting applies cluster-wide, unless overridden via a *Domain Specific Configuration*.

This is **not** the same as the Scrapy Auto Throttle! Scrapy's Auto Throttle works great when running a single Scrapy Spider process, but falls short when coordinating concurrent processes across different machines. Scrapy Cluster's throttling mechanism allows for spiders to coordinate crawls spread across machines.

> **Warning:** Scrapy Cluster by default comes with a *very modest* 10 hits per 60 seconds, for any domain it sees. This may seem very slow, but when scaled horizontally across many IP addresses allows you to crawl at a large scale with minimal fear of any individual IP address being blocked.

SCHEDULER_QUEUE_REFRESH - Controls how often your spiders check for new domains (other than the ones they have already seen). This is time intensive and means your spider is not crawling, so a higher but responsive time is recommended here.

QUEUE_MODERATED - Determines whether you would like cluster of spiders to hit the domain at even intervals spread throughout the `QUEUE_WINDOW`, or execute crawls as fast as possible and then pause. If you crawl at 10 hits every 60 seconds, a moderated queue would allow your spiders to crawl at one request every 6 seconds (60 sec / 10 hits = 6 secs between every 1 hit). Turning off this setting would allow your spiders to reach their 6 hit cap as fast as possible within the 60 second window.

SCHEDULER_QUEUE_TIMEOUT - Gives you control over how long stagnant domain queues persist within the spider before they are expired. This prevents memory build up where a spider has every domain it has ever seen in memory. Instead, only the domains that have been active within this window will stay around. If a domain expires, it can be easily recreated when a new request generated for it.

Domain Specific Configuration

For crawls where you know a specific target site, you can override the general settings above and fine tune how fast your cluster hits a specific site.

To do this, please use the *Zookeeper API* to add, remove, and update domain specific blacklists or domain specific throttle configuration. Adding a domain blacklist means that **all spiders** will ignore crawl requests for a domain, whereas a domain specific throttle will force **all spiders** to crawl the domain at that rate (a blacklist will override this).

Through using the Zookeeper API you will build up a configuration file containing your blacklist and domain throttles.

example.yml

This is an example yaml configuration file for overriding the general settings on a site by site basis. Scrapy Cluster requires the configuration to be like the following:

```
blacklist:
    - <domain3.com>
domains:
    <domain1.com>:
        window: <QUEUE_WINDOW>
        hits: <QUEUE_HITS>
    <domain2.org>:
        window: <QUEUE_WINDOW>
        hits: <QUEUE_HITS>
        scale: 0.5
```

The yaml syntax dictates both a blacklist and a series of domains. The blacklist is a list of domains that all spiders should ignore. Within each domain, there is required at minimum to be both a `window` and `hits` value. These correspond to the `QUEUE_HITS` and `QUEUE_WINDOW` above. There is also an optional value called `scale`, where you can apply a scale value between 0 and 1 to the domain of choice. The combination of the window, hits, and scale values allows you to fine tune your cluster for targeted domains, but to apply the *general settings* to any other domain.

Note: Your crawler cluster does **not** need to be restarted to read their new cluster configuration! The crawlers use the *Zookeeper Watcher* utility class to dynamically receive and update their configuration.

file_pusher.py

Warning: The `file_pusher.py` manual script is deprecated in favor of the *Zookeeper API*, and the documentation here may be removed in the future

Once you have a desired yaml configuration, the next step is to push it into Zookeeper using the `file_pusher.py` script. This is a small script that allows you to deploy crawler configuration to the cluster.

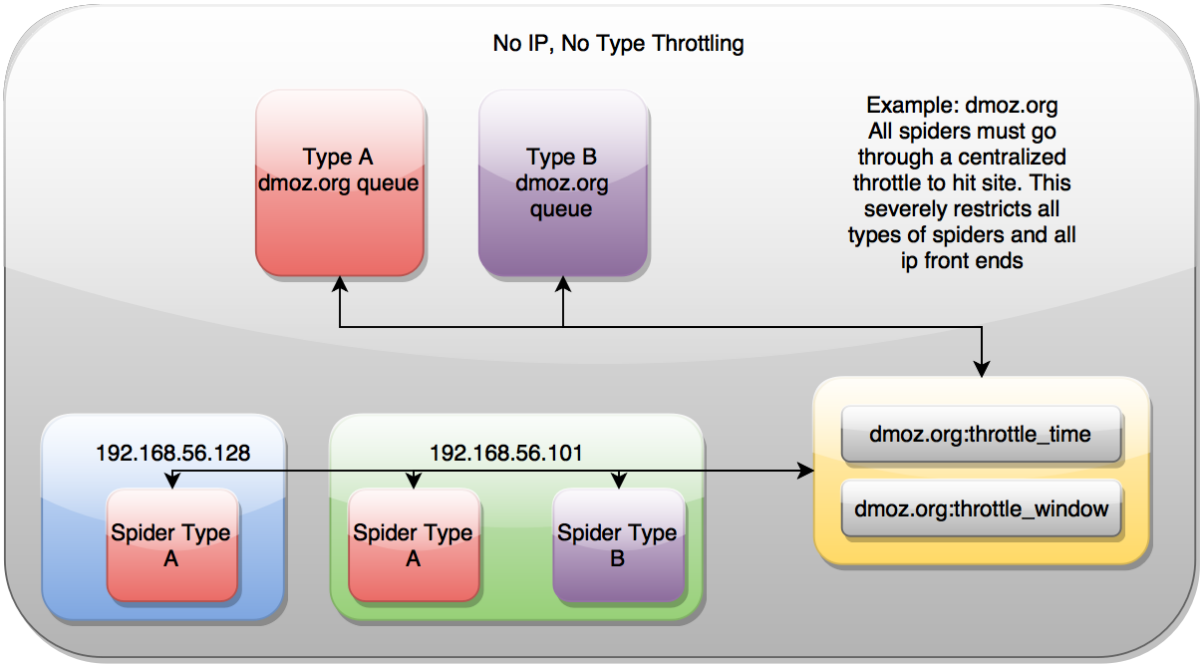
```
$ python file_pusher.py -f example.yml -z scdev
updating conf file
```

Here we pushed our example configuration file to the Zookeeper host located at `scdev`. That is all you need to do in order to update your crawler configuration! You can also use an external application to update your Zookeeper configuration file, as long as it conforms to the required yaml syntax.

Throttle Mechanism

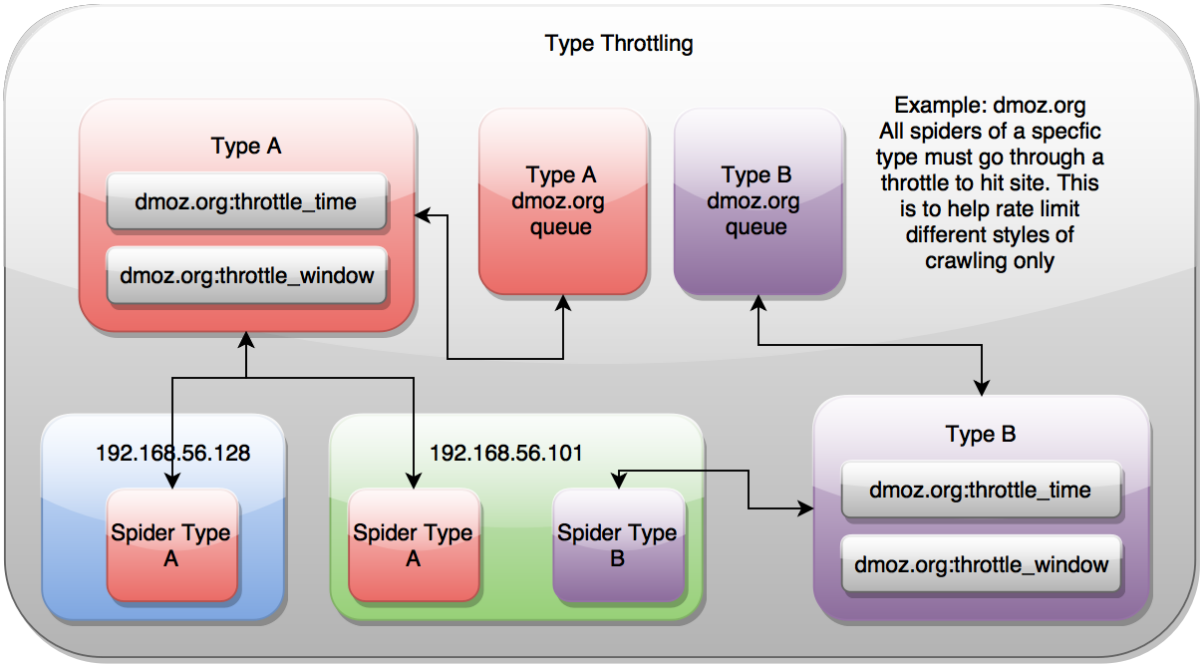
Now that we have determined how fast our cluster hits a particular domain, we need to determine how that domain throttle is applied to our spiders and our crawling machines. Each of the four different throttle types are outlined below.

No IP Address, No Type



No throttle style dictates that the domain coordination is done through a single place. It is indifferent to the spider or the IP addresses of the machines crawling, ensuring they are all rate limited by one mechanism only.

Type



Type throttling means that for each domain, spiders of a different type (ie `A`, `B`, `link`, `foo`) will orchestrate with themselves to control how fast the cluster hits the domain. This disregards the public IP address of the machines that the Scrapy requests are routed through, so spiders on different machines are throttled based on how fast all of the other spiders in the cluster have hit that particular domain.

IP Address

IP Address throttling controls the cluster based on the spider's public facing IP Address, but ignores the type of spider it is. This is most useful when you have various spiders running on your machines, but only want to hit a domain a certain rate.

IP Address and Type

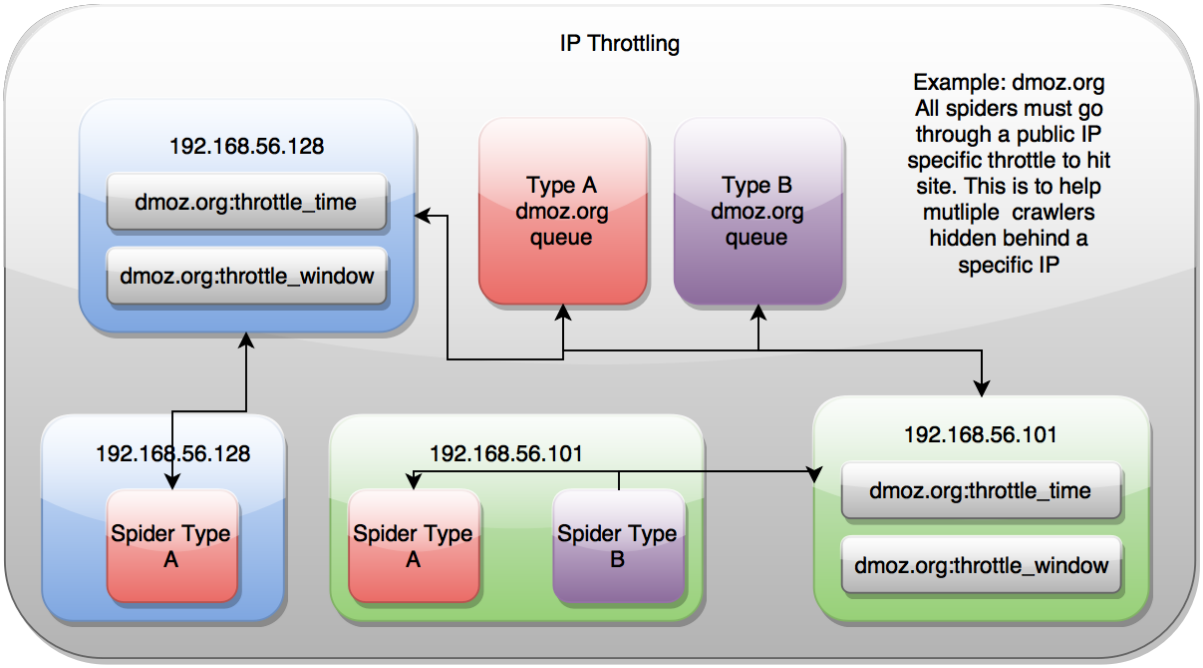
IP and Type throttling combines both of the above throttle styles, and allows your spiders to control themselves based upon both their public IP address and the Spider type. This is useful when you have multiple spiders on the same machine that are configured to hit different proxies, and would like to control how fast they hit a domain based upon their spider type *and* their public IP address.

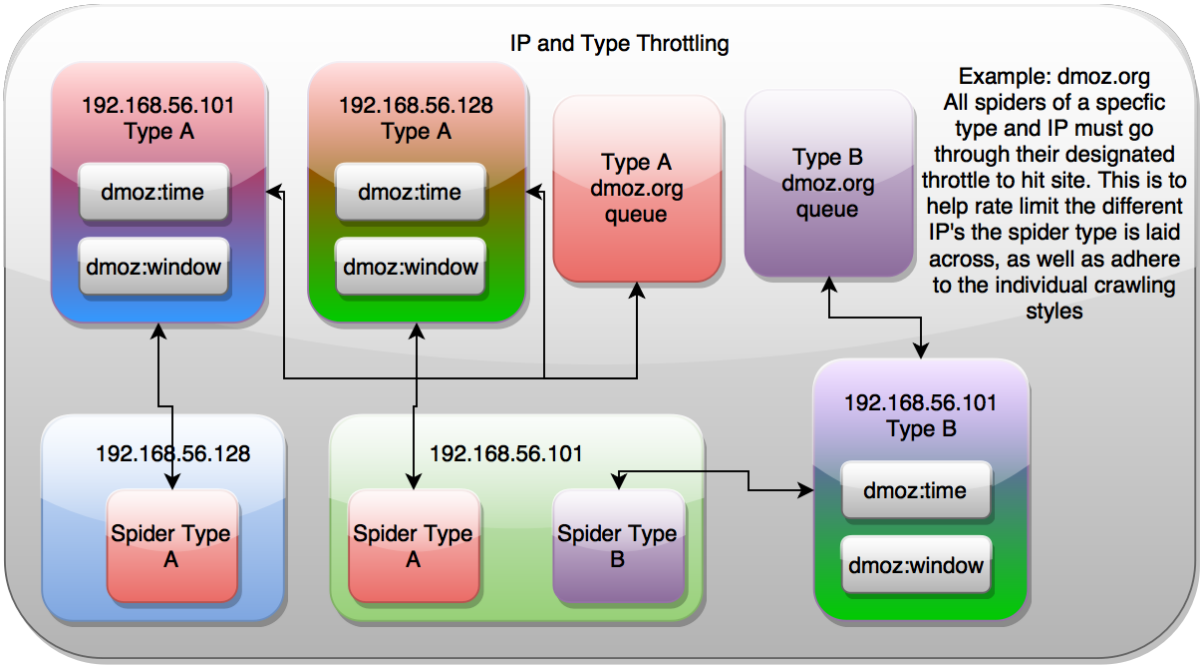
Settings

To utilize the different throttle mechanisms you can alter the following settings in your `localsettings.py` file. You then need to restart your crawling processes for the new settings to take effect.

SCHEDULER_TYPE_ENABLED - Flag to set the **Type** Style throttling

SCHEDULER_IP_ENABLED - Flag to set the **IP Address** Style throttling


IP Throttling
Example: dmoz.org
All spiders must go
through a public IP
specific throttle to hit
site. This is to help
mutliple crawlers
hidden behind a
specific IP
192.168.56.128
dmoz.org:throttle_time
dmoz.org:throttle_window
Type A
dmoz.org
queue
Type B
dmoz.org
queue
192.168.56.128
Spider Type
A
192.168.56.101
Spider Type
A
Spider Type
B
192.168.56.101
dmoz.org:throttle_time
dmoz.org:throttle_window


IP and Type Throttling
Example: dmoz.org
All spiders of a specfic
type and IP must go
through their designated
throttle to hit site. This is to
help rate limit the different
IP's the spider type is laid
across, as well as adhere
to the individual crawling
styles
192.168.56.101
Type A
dmoz:time
dmoz:window
192.168.56.128
Type A
dmoz:time
dmoz:window
Type A
dmoz.org
queue
Type B
dmoz.org
queue
192.168.56.128
Spider Type
A
192.168.56.101
Spider Type
A
Spider Type
B
192.168.56.101
Type B
dmoz:time
dmoz:window

Combining Domain Queues and Throttling

At the core of Scrapy Cluster is a Redis priority queue that holds all of the requests for a particular spider type and domain, like `link:dmoz.org:queue`. The configured throttle determines when an individual Scrapy process can receive a new request from the Redis Queues. Only when the throttle says that it is "ok" will the Spider be returned a link to process.

This results in Spiders across the cluster continually polling all available domain queues for new requests, but only receiving requests when the throttle mechanism indicates that the request limit has not gone beyond the max desired configuration. Because the throttle coordination is conducted via Redis, it is not reliant on any one Scrapy process to determine whether the cluster can or can't crawl a particular domain.

If the spider polls a domain and is denied a request, it will cycle through all other known domains until it finds one that it can process. This allows for very high throughput when crawling many domains simultaneously. Domain A may only allow 10 hits per minute, domain B allows for 30 hits per minute, and domain C allows for 60 hits per minute. **In this case, all three domains can be crawled at the same time by the cluster while still respecting the domain specific rate limits.**

By tuning your cluster configuration for your machine setup and desired crawl rate, you can easily scale your Scrapy Cluster to process as much data as your network can handle.

Inter-spider Communication

By default, Scrapy Cluster spiders yield `Request`'s to their own spider type. This means that **link** spiders will crawl other **link** spider requests, and if you have another spider running those requests will not interfere.

The distributed scheduler that spider's use is actually flexible in that **you can yield ''Requests'' to other spiders within your cluster.** This is possible thanks to the `spiderid` that is built into every crawl request that the cluster deals with.

The spider `name` at the top of your Spider class dictates the identifier you can use when yielding requests.

```
class LinkSpider(RedisSpider):
  name = "link"
```

You can alse see this same name being used in the Redis Queues

```
<spiderid>:<domain>:queue
```

Thanks to the scheduler being indifferent as to what requests it is processing, Spider A can yield requests to Spider B, with both of them using different parsing, pipelines, middlewares, and anything else that is customizable for your spider. All you need to do is set the `spiderid` meta field within your request.

```
response.meta['spiderid'] = 'othername'
```

While this use case does not come up very often, you can imagine a couple scenarios where this might be useful:

- Your cluster is doing a large crawl using the `link` spider, but you have special domains where you would like to switch to a different crawling approach. When Spider A (the one doing the large crawl) hits a target website, it yields a request to Spider B which does a more detailed or custom scrape of the site in question.
- You are following web links from your favorite social media website and submitting them to be crawled by your cluster. On occasion, you get a "login" prompt that your spider can't handle. When that login prompt is detected, you yield to a special `login` spider in order to handle the extra logic of scraping that particular site.
- You are scraping a shopping site, and already know all of the main pages you would like to grab. All of the links on your shopping site are actually for products, which have a different set of elements that require different middlewares or other logic to be applied. You main site spider then yields requests to the `product` spider.

So how is this different than using the `callback` parameter within a normal Scrapy Spider? It's different because these Spiders may be completely different Scrapy projects, with their own settings, middleware, items, item pipelines, downloader middlewares, or anything else you need to enhance your Scrapy spider. Using a callback requires you to either combine your code, add extra logic, or not conduct special processing you would otherwise get from using two different Scrapy spiders to do very different jobs.

The spiders can yield requests to each other, in a chain, or any other manner in order for your cluster to be successful.

Extension

This page outlines how you can extend Scrapy but still work within Scrapy Cluster's environment.

General

In general, Scrapy Cluster embraces the extension and flexibility of Scrapy, while providing new hooks and frameworks in order to do highly distributed crawling and orchestration. You can see some instances of this within the Scrapy project, as Scrapy Cluster by default needs to use some middlewares and item pipelines to fit things together.

The most heavily customized components of the Scrapy Cluster project involve a distributed scheduler and base spider class. These two classes work together to allow for distributed crawling, but otherwise do not interfere with normal Scrapy processes.

Additional Info

There is a Scrapy Cluster logger available throughout the project, and there should be plenty of examples on how to create a logger at any one part of your Scrapy Project. You should see examples for how to create a logger in the `pipelines.py` file and the `log_retry_middleware.py` file.

Note: Spiders that use the base `RedisSpider` class already have a Scrapy Cluster logger, located at `self._logger`.

Spiders

Scrapy Cluster allows you to build Scrapy based spiders that can coordinate with each other by utilizing a customized Scrapy Scheduler. Your new Scrapy Cluster based spider outline looks just like a normal Scrapy Spider class, but inherits from Scrapy Cluster's base *RedisSpider* class.

Below is an outline of the spider file.

```
from redis_spider import RedisSpider


class MySpider(RedisSpider):
    name = "myspider"

    def __init__(self, *args, **kwargs):
        super(MySpider, self).__init__(*args, **kwargs)

    def parse(self, response):
        # used for collecting statistics about your crawl
        self._increment_status_code_stat(response)
```

```
        # process your response here
        # yield Scrapy Requests and Items like normal
```

That is it! Your spider is now hooked into your scraping cluster, and can do any kind of processing with your responses like normal.

Additional Info

All Spiders that inherit from the base spider class `RedisSpider` will have access to a Scrapy Cluster logger found at `self._logger`. This is not the same as the Scrapy logger, but can be used in conjunction with it. The following extra functions are provided to the spider as well.

- **reconstruct_headers(response)** - A patch that fixes malformed header responses seen in some websites. This error surfaces when you go to debug or look at the headers sent to Kafka, and you find some of the headers present in the spider are non-existent in the item sent to Kafka. Returns a `dict`.

You can `yield` requests from your Spider just like a normal Scrapy Spider. Thanks to the built in in Scrapy Cluster `MetaPassthroughMiddleware`, you don't have to worry about the additional overhead required for distributed crawling. If you look at both the `WanderingSpider` and `LinkSpider` examples, you will see that the only extra information passed into the request via the `meta` fields are related to what we actually want to do with the spider.

Don't want to use the ''RedisSpider'' base class? That's okay, as long as your spider can adhere to the following guidelines:

- Connect a signal to your crawler so it does not stop when idle.

```
    ...
    self.crawler.signals.connect(self.spider_idle, signal=signals.spider_idle)

def spider_idle(self):
    raise DontCloseSpider
```

- Implement the logging and parse methods

```
def set_logger(self, logger):
    # would allow you to set the Scrapy Cluster logger
    pass

def parse(self, response):
    # your normal parse method
    pass
```

With that, in combination with the settings, middlewares, and pipelines already provided by Scrapy Cluster, you should be able to use a customer spider with little effort.

Example

Let's create a new Spider that integrates within Scrapy Cluster. This guide assumes you already have a working cluster already, so please ensure everything is properly hooked up by following the quick start guides for the various components.

We are going to create a Wandering Spider. The goal of this spider is to stumble through the internet, only stopping when it hits a webpage that has no url links on it. We will randomly chose a link from all available links on the page, and yield *only* that url to be crawled by the cluster. This way a single crawl job does not spread out, but serially jumps from one page to another.

Below is the spider class, and can be found in `crawling/spiders/wandering_spider.py`.

```
# Example Wandering Spider
import scrapy

from scrapy.http import Request
from lxmlhtml import CustomLxmlLinkExtractor as LinkExtractor
from scrapy.conf import settings

from crawling.items import RawResponseItem
from redis_spider import RedisSpider

import random


class WanderingSpider(RedisSpider):
    '''
    A spider that randomly stumbles through the internet, until it hits a
    page with no links on it.
    '''
    name = "wandering"

    def __init__(self, *args, **kwargs):
        super(WanderingSpider, self).__init__(*args, **kwargs)

    def parse(self, response):
        # debug output for receiving the url
        self._logger.debug("crawled url {}".format(response.request.url))
        # collect stats

        # step counter for how many pages we have hit
        step = 0
        if 'step' in response.meta:
            step = response.meta['step']

        # Create Item to send to kafka
        # capture raw response
        item = RawResponseItem()
        # populated from response.meta
        item['appid'] = response.meta['appid']
        item['crawlid'] = response.meta['crawlid']
        item['attrs'] = response.meta['attrs']
        # populated from raw HTTP response
        item["url"] = response.request.url
        item["response_url"] = response.url
        item["status_code"] = response.status
        item["status_msg"] = "OK"
        item["response_headers"] = self.reconstruct_headers(response)
        item["request_headers"] = response.request.headers
        item["body"] = response.body
        item["links"] = []
        # we want to know how far our spider gets
        if item['attrs'] is None:
            item['attrs'] = {}

        item['attrs']['step'] = step

        self._logger.debug("Finished creating item")
```

```
        # determine what link we want to crawl
        link_extractor = LinkExtractor(
                            allow_domains=response.meta['allowed_domains'],
                            allow=response.meta['allow_regex'],
                            deny=response.meta['deny_regex'],
                            deny_extensions=response.meta['deny_extensions'])

        links = link_extractor.extract_links(response)

        # there are links on the page
        if len(links) > 0:
            self._logger.debug("Attempting to find links")
            link = random.choice(links)
            req = Request(link.url, callback=self.parse)

            # increment our step counter for this crawl job
            req.meta['step'] = step + 1

            # pass along our user agent as well
            if 'useragent' in response.meta and \
                        response.meta['useragent'] is not None:
                    req.headers['User-Agent'] = response.meta['useragent']

            # debug output
            self._logger.debug("Trying to yield link '{}'".format(req.url))

            # yield the Request to the scheduler
            yield req
        else:
            self._logger.info("Did not find any more links")

        # raw response has been processed, yield to item pipeline
        yield item
```

In stepping through our `parse()` method, you can see we first start off by collecting statistics information about our cluster. We then use the variable `step` to determine how many pages our crawl job has visited so far. After that, we create the `RawResponseItem` and fill it with our typical crawl data, and make sure to insert our `step` variable so our data output has that extra information in it.

After that, we create a link extractor and do a `random.choice()` from our extracted links, and yield the request. At the bottom we finally yeild our response item to the item pipeline.

You can now spin a few spiders up by running the following command.

```
scrapy runspider crawling/spiders/wandering_spider.py
```

Then, feed your cluster.

```
python kafka_monitor.py feed '{"url": "http://dmoz.org", "appid":"testapp", "crawlid":
↪"test123456", "spiderid":"wandering"}'
```

If you are looking at your `demo.crawled_firehose` Kafka Topic using the `kafkadump.py` script, you will begin to see output like so...

```
{
    "body": <omitted>,
    "crawlid": "test123456",
    "response_url": "http://www.dmoz.org/",
```

```
    "url": "http://www.dmoz.org/",
    "status_code": 200,
    "status_msg": "OK",
    "appid": "testapp",
    "links": [],
    "request_headers": {
        "Accept-Language": "en",
        "Accept-Encoding": "gzip,deflate",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "User-Agent": "Scrapy/1.0.4 (+http://scrapy.org)"
    },
    "attrs": {
        "step": 0
    },
    "timestamp": "2016-01-23T22:01:33.379721"
}
{
    "body": <omitted>,
    "crawlid": "test123456",
    "response_url": "http://www.dmoz.org/Computers/Hardware/",
    "url": "http://www.dmoz.org/Computers/Hardware/",
    "status_code": 200,
    "status_msg": "OK",
    "appid": "testapp",
    "links": [],
    "request_headers": {
        "Accept-Language": "en",
        "Accept-Encoding": "gzip,deflate",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "User-Agent": "Scrapy/1.0.4 (+http://scrapy.org)"
    },
    "attrs": {
        "step": 1
    },
    "timestamp": "2016-01-23T22:01:35.566280"
}
```


Notice the `attrs` field has our step value, and we can now track all of the hops the Scrapy Cluster is making. Your cluster is now serially working on that particular crawl job until it hits a page it has already seen, or does not find any links in the response.

You can also fire up more than one crawl job at a time, and track the steps that job makes. After creating some more jobs and letting the cluster run for a while, here is a snapshot of the Redis Monitor crawl data dump.


```
2016-01-23 17:47:21,164 [redis-monitor] INFO: Crawler Stats Dump:
{
    "total_spider_count": 4,
    "unique_spider_count": 1,
    "wandering_200_21600": 108,
    "wandering_200_3600": 60,
    "wandering_200_43200": 108,
    "wandering_200_604800": 108,
    "wandering_200_86400": 108,
    "wandering_200_900": 49,
    "wandering_200_lifetime": 107,
    "wandering_404_21600": 4,
    "wandering_404_3600": 1,
    "wandering_404_43200": 4,
```

```
    "wandering_404_604800": 4,
    "wandering_404_86400": 4,
    "wandering_404_900": 1,
    "wandering_404_lifetime": 4,
    "wandering_spider_count": 4
}
```

You now have two different examples of how Scrapy Cluster extends Scrapy to give you distributed crawling capabilities.

Settings

The following settings are Scrapy Cluster specific. For all other Scrapy settings please refer to the official Scrapy documentation here.

Redis

REDIS_HOST

Default: `'localhost'`

The Redis host.

REDIS_PORT

Default: `6379`

The port to use when connecting to the `REDIS_HOST`.

REDIS_DB

Default: `0`

The Redis database to use when connecting to the `REDIS_HOST`.

Kafka

KAFKA_HOSTS

Default: `'localhost:9092'`

The Kafka host. May have multiple hosts separated by commas within the single string like `'h1:9092,h2:9092'`.

KAFKA_TOPIC_PREFIX

Default: `'demo'`

The Kafka Topic prefix to use when generating the outbound Kafka topics. **KAFKA_APPID_TOPICS**

Default: `False`

Flag to send data to both the firehose and Application ID specific Kafka topics. If set to `True`, results will be sent to both the `demo.outbound_firehose` **and** `demo.outbound_<appid>` Kafka topics, where `<appid>` is the Application ID used to submit the request. This is useful if you have many applications utilizing your cluster but only would like to listen to results for your specific application. **KAFKA_BASE_64_ENCODE**

Default: `False`

Base64 encode the raw crawl body from the crawlers. This is useful when crawling malformed utf8 encoded pages, where json encoding throws an error. If an error occurs when encoding the crawl object in the item pipeline, there will be an error thrown and the result will be dropped.

KAFKA_PRODUCER_BATCH_LINGER_MS

Default: `25`

The time to wait between batching multiple requests into a single one sent to the Kafka cluster.

KAFKA_PRODUCER_BUFFER_BYTES

Default: `4 * 1024 * 1024`

The size of the TCP send buffer when transmitting data to Kafka

Zookeeper

ZOOKEEPER_ASSIGN_PATH

Default: `/scrapy-cluster/crawler/`

The location to store Scrapy Cluster domain specific configuration within Zookeeper

ZOOKEEPER_ID

Default: `all`

The file identifier to read crawler specific configuration from. This file is located within the `ZOOKEEPER_ASSIGN_PATH` folder above.

ZOOKEEPER_HOSTS

Default: `localhost:2181`

The zookeeper host to connect to.

Scheduler

SCHEDULER_PERSIST

Default: `True`

Determines whether to clear all Redis Queues when the Scrapy Scheduler is shut down. This will wipe all domain queues for a particular spider type.

SCHEDULER_QUEUE_REFRESH

Default: `10`

How many seconds to wait before checking for new or expiring domain queues. This is also dictated by internal Scrapy processes, so setting this any lower does not guarantee a quicker refresh time.

SCHEDULER_QUEUE_TIMEOUT

Default: `3600`

The number of seconds older domain queues are allowed to persist before they expire. This acts as a cache to clean out queues from memory that have not been used recently. **SCHEDULER_BACKLOG_BLACKLIST**

Default: `True`

Allows blacklisted domains to be added back to Redis for future crawling. If set to `False`, domains matching the Zookeeper based domain blacklist will not be added back in to Redis.

Throttle

QUEUE_HITS

Default: `10`

When encountering an unknown domain, throttle the domain to X number of hits within the `QUEUE_WINDOW`

QUEUE_WINDOW

Default: `60`

The number of seconds to count and retain cluster hits for a particular domain.

QUEUE_MODERATED

Default: `True`

Moderates the outbound domain request flow to evenly spread the `QUEUE_HITS` throughout the `QUEUE_WINDOW`.
DUPEFILTER_TIMEOUT

Default: `600`

Number of seconds to keep **crawlid** specific duplication filters around after the latest crawl with that id has been conducted. Putting this setting too low may allow crawl jobs to crawl the same page due to the duplication filter being wiped out.

SCHEDULER_IP_REFRESH

Default: `60`

The number of seconds to wait between refreshing the Scrapy process's public IP address. Used when doing *IP* based throttling.

PUBLIC_IP_URL

Default: `'http://ip.42.pl/raw'`

The default URL to grab the Crawler's public IP Address from.

IP_ADDR_REGEX

Default: `(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})`

The regular expression used to find the Crawler's public IP Address from the `PUBLIC_IP_URL` response. The first element from the results of this regex will be used as the ip address.

SCHEDULER_TYPE_ENABLED

Default: `True`

If set to true, the crawling process's spider type is taken into consideration when throttling the crawling cluster.

SCHEDULER_IP_ENABLED

Default: `True`

If set to true, the crawling process's public IP Address is taken into consideration when throttling the crawling cluster.

Note: For more information about Type and IP throttling, please see the *throttle* documentation.

SCHEUDLER_ITEM_RETRIES

Default: `2`

Number of cycles through all known domain queues the Scheduler will take before the Spider is considered idle and waits for Scrapy to retry processing a request.

Logging

SC_LOGGER_NAME

Default: `'sc-crawler'`

The Scrapy Cluster logger name.

SC_LOG_DIR

Default: `'logs'`

The directory to write logs into. Only applicable when `SC_LOG_STDOUT` is set to `False`.

SC_LOG_FILE

Default: `'sc_crawler.log'`

The file to write the logs into. When this file rolls it will have `.1` or `.2` appended to the file name. Only applicable when `SC_LOG_STDOUT` is set to `False`.

SC_LOG_MAX_BYTES

Default: `10 * 1024 * 1024`

The maximum number of bytes to keep in the file based log before it is rolled.

SC_LOG_BACKUPS

Default: `5`

The number of rolled file logs to keep before data is discarded. A setting of `5` here means that there will be one main log and five rolled logs on the system, totaling six log files.

SC_LOG_STDOUT

Default: `True`

Log to standard out. If set to `False`, will write logs to the file given by the `LOG_DIR/LOG_FILE`

SC_LOG_JSON

Default: `False`

Log messages will be written in JSON instead of standard text messages.

SC_LOG_LEVEL

Default: `'INFO'`

The log level designated to the logger. Will write all logs of a certain level and higher.

Note: More information about logging can be found in the utilities *Log Factory* documentation.

Stats

STATS_STATUS_CODES

Default: `True`

Collect Response status code metrics

STATUS_RESPONSE_CODES

Default:

```
[
    200,
    404,
    403,
    504,
]
```

Determines the different Response status codes to collect metrics against if metrics collection is turned on.

STATS_CYCLE

Default: `5`

How often to check for expired keys and to roll the time window when doing stats collection.

STATS_TIMES

Default:

```
[
    'SECONDS_15_MINUTE',
    'SECONDS_1_HOUR',
    'SECONDS_6_HOUR',
    'SECONDS_12_HOUR',
    'SECONDS_1_DAY',
    'SECONDS_1_WEEK',
]
```

Rolling time window settings for statistics collection, the above settings indicate stats will be collected for the past 15 minutes, the past hour, the past 6 hours, etc.

Note: For more information about stats collection, please see the *Stats Collector* documentation.

Design Learn about the design considerations for the Scrapy Cluster Crawler

Quick Start How to use and run the distributed crawlers

Controlling Learning how to control your Scrapy Cluster will enable you to get the most out of it

Extension How to use both Scrapy and Scrapy Cluster to enhance your crawling capabilites

Settings Explains all of the settings used by the Crawler

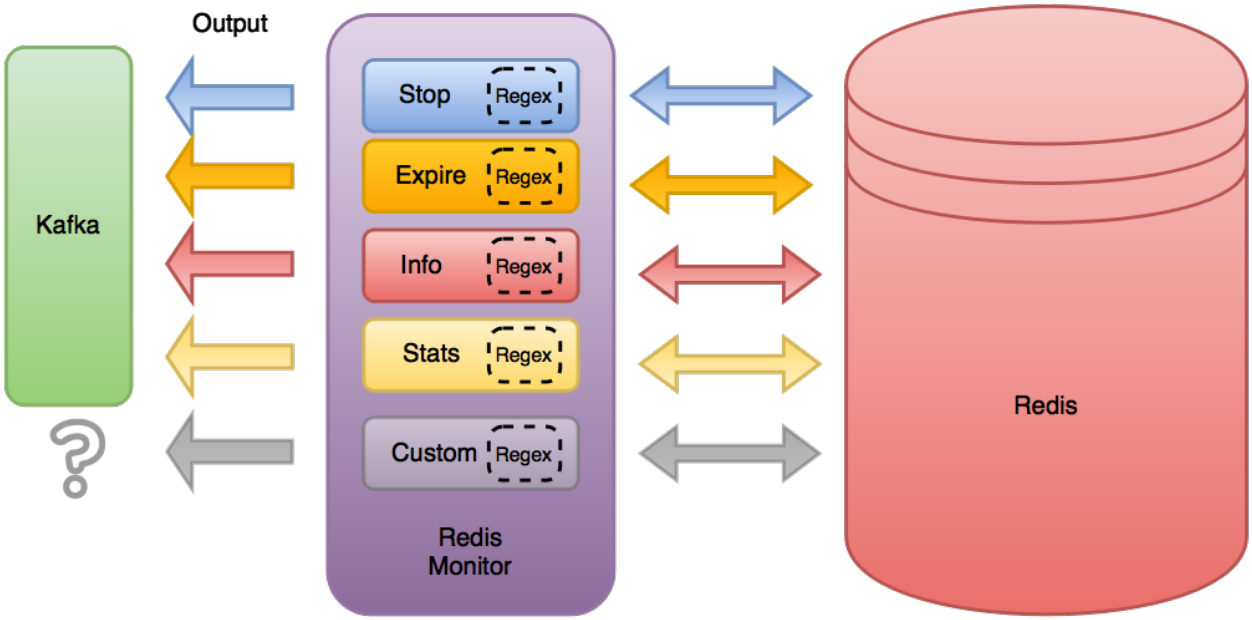
CHAPTER 4

Redis Monitor

The Redis Monitor serves to moderate the redis based crawling queue. It is used to expire old crawls, stop existing crawls, and gather information about the cluster.

Design

The Redis Monitor is designed to act as a surgical instrument which allows an application to peer into the queues and keys managed by Redis that are used by Scrapy Cluster. It runs independently of the cluster, and interacts with the items within Redis through the use of Plugins.



Every Plugin used by the Redis Monitor looks for a specific set of keys within Redis, which is determined by a regular

expression. When a key is found within that matches the expression, both the key and the value are transferred to the Plugin for further processing.

For Scrapy Cluster, the default plugins take their results and either do data manipulation further within Redis, or send a Kafka message back out to the requesting application. The Redis Monitor's settings allow for full customization of the kinds of plugins you wish to run, and in the order they are processed.

This allows the core Redis Monitor code to remain small, but allows for customization and extendability. That, in combination with the ability to run multiple Redis Monitors across a number of different machines, provides fault tolerance and the ability to interact with the cluster in many different simultaneous ways.

Quick Start

First, create a `localsettings.py` to track your overridden custom settings. You can override any setting you find within the Redis Monitor's `settings.py`, and a typical file may look like the following:

```
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
```

redis_monitor.py

The main mode of operation for the Redis Monitor is to run the python script to begin monitoring the Redis instance your cluster is interacting with.

```
$ python redis_monitor.py
```

Typical Usage

Typical usage of the Redis Monitor is done in conjunction with the Kafka Monitor, since the Kafka Monitor is the gateway into the cluster.

> **Warning:** This guide assumes you have a running Kafka Monitor already. Please see the Kafka Monitor *Quick Start* for more information.

Open three terminals.

Terminal 1:

Monitor your kafka output

```
$ python kafkadump.py dump -t demo.outbound_firehose -p
```

Terminal 2:

Run the Redis Monitor

```
$ python redis_monitor.py
```

Terminal 3:

Feed an item

```
$ python kafka_monitor.py feed '{"stats": "kafka-monitor", "appid":"testapp", "uuid":
↪"myuuidhere"}'
2016-01-18 20:58:18,130 [kafka-monitor] INFO: Feeding JSON into demo.incoming
{
    "stats": "kafka-monitor",
    "uuid": "myuuidhere",
    "appid": "testapp"
}
2016-01-18 20:58:18,131 [kafka-monitor] INFO: Successfully fed item to Kafka
```

You should see a log message come through Terminal 2 stating the message was received.

```
2016-01-18 20:58:18,228 [redis-monitor] INFO: Received kafka-monitor stats request
2016-01-18 20:58:18,233 [redis-monitor] INFO: Sent stats to kafka
```

After a short time, you should see the result come through your Kafka topic in Terminal 1.

```
{
    "server_time": 1453168698,
    "uuid": "myuuidhere",
    "plugins": {
        "StatsHandler": {
            "900": 2,
            "3600": 2,
            "604800": 2,
            "43200": 2,
            "lifetime": 19,
            "86400": 2,
            "21600": 2
        },
        "ScraperHandler": {
            "604800": 6,
            "lifetime": 24
        },
        "ActionHandler": {
            "604800": 7,
            "lifetime": 24
        }
    },
    "appid": "testapp",
    "fail": {
        "lifetime": 7
    },
    "stats": "kafka-monitor",
    "total": {
        "900": 2,
        "3600": 2,
        "604800": 15,
        "43200": 2,
        "lifetime": 73,
        "86400": 2,
        "21600": 2
    }
}
```

At this point, your Redis Monitor is functioning.

Plugins

Plugins give the Redis Monitor additional functionality for monitoring your Redis instance. They evaluate all the keys within your Redis instance, and act upon the key/value when the expression matches.

The Redis Monitor loads the desired *Plugins* in the order specified in the settings file, and will evaluate them starting with the lowest ranked plugin.

Each plugin consists of a class inherited from the base plugin class, and a couple different methods for setup and processing.

Default Plugins

By default, plugins live in the `plugins` directory within the Redis Monitor. The following plugins come standard.

Info Monitor

The `info_monitor.py` provides all of the Information Action responses from Scrapy Cluster.

For more information please see the *Information Action* API.

Expire Monitor

The `expire_monitor.py` provides Scrapy Cluster the ability to continuously monitor crawls that need to be expired at a designated time.

For more information please see the *Expire Action* API.

Stop Monitor

The `stop_monitor.py` allows API requests to stop a crawl that is in process.

For more information please see the *Stop Action* API.

Stats Monitor

The `stats_monitor.py` gives access to stats collected across all three major components of Scrapy Cluster.

For more information please see the *Stats API*

Zookeeper Monitor

The `zookeeper_monitor.py` adds the ability to update the crawler blacklist once the request is recieved.

For more information please see the *Zookeeper API*

Extension

Creating your own Plugin for the Redis Monitor allows you to add new functionality to monitor your Redis instance, and act when certain keys are seen. You will need to add a new file in the `plugins` directory that will define your Redis key regular expression and how you process your results.

If you would like only default functionality, the following python code template should be used when creating a new plugin:

plugins/my_monitor.py

```
from base_monitor import BaseMonitor


class MyMonitor(BaseMonitor):
    '''
    My custom monitor
    '''

    regex = "<my_key_regex_here>"

    def setup(self, settings):
        '''
        Setup the handler

        @param settings: The loaded settings file
        '''
        pass

    def check_precondition(self, key, value):
        '''
        Override if special conditions need to be met
        '''
        pass

    def handle(self, key, value):
        '''
        Processes a vaild action info request

        @param key: The key that matched the request
        @param value: The value associated with the key
        '''
        pass

    def close(self):
        '''
        Called when the over arching Redis Monitor is closed
        '''
        pass
```

If you would like to send messages back out to Kafka, use the following template.

```
from kafka_base_monitor import KafkaBaseMonitor


class MyMonitor(KafkaBaseMonitor):

    regex = "<my_key_regex_here>"
```

```
    def setup(self, settings):
        '''
        Setup kafka
        '''
        KafkaBaseMonitor.setup(self, settings)

    def check_precondition(self, key, value):
        '''
        Override if special conditions need to be met
        '''
        pass

    def handle(self, key, value):
        '''
        Processes a vaild action info request

        @param key: The key that matched the request
        @param value: The value associated with the key
        '''
        # custom code here builds a result dictionary `results`
        # ...
        # now, send to kafka
        if self._send_to_kafka(results):
            self.logger.info('Sent results to kafka')
        else:
            self.logger.error('Failed to send results to kafka')
```

Regardless of either template you choose, you will inherit from a base class that provides easy integration into the plugin framework. The `regex` variable at the top of each class should contain the Redis key pattern your plugin wishes to operate on.

The `setup()` method is passed a dictionary created from the settings loaded from your local and default settings files. You can set up connections, variables, or other items here to be used in your handle method.

The `check_precondition()` method is called for every potential key match, and gives the plugin the opportunity to determine whether it actually wants to process that object at the given time. For example, this is used in the `expire_monitor.py` file to check whether the expire timestamp value stored in the key is greater than the current time. If it is, return `True` and your plugin's `handle()` method will be called, otherwise, return `False`.

When the `handle()` method is called, it is passed both the key that matched your pattern, and the value stored within the key. You are free to do whatever you want with the data, but once you are done the key is removed from Redis. The key will be removed if an exception is thrown consitently when trying to process our action within any of the two plugin methods, or if the `handle()` method completes as normal. This is to prevent reprocessing of matched keys, so use the `check_precondition()` method to prevent a key from getting deleted too early.

If you need to tear down anything within your plugin, you can use the `close()` method to ensure proper clean up of the data inside the plugin. Once you are ready to add your plugin to the Redis Monitor, edit your `localsettings.py` file and add the following lines.

```
PLUGINS = {
    'plugins.my_monitor.MyMonitor': 500,
}
```

You have now told the Redis Monitor to not only load the default plugins, but to add your new plugin as well with a rank of 500. Restart the Redis Monitor for it to take effect.

Additional Info

Every Redis Monitor plugin is provided a Scrapy Cluster logger, under the variable name `self.logger`. You can use this logger to generate debug, info, warnings, or any other log output you need to help gather information from your plugin. This is the same logger that the core Redis Monitor uses, so your desired settings will be preserved.

Each Plugin is also provided a default Redis Connection variable, named `self.redis_conn`. This variable is an instance of a Redis Connection thanks to the redis-py library and can be used to manipulate anything within your Redis instance.

Settings

This page covers the various settings contained within the Redis Monitor. The sections are broken down by functional component.

Core

SLEEP_TIME

Default: `0.1`

The number of seconds the main process will sleep between checking for new actions to take care of.

RETRY_FAILURES

Default: `True`

Retry an action if there was an unexpected failure while computing the result.

RETRY_FAILURES_MAX

Default: `3`

The number of times to retry a failed action before giving up. Only applied when `RETRY_FAILURES` is enabled.

HEARTBEAT_TIMEOUT

Default: `120`

The amount of time the statistics key the Redis Monitor instance lives to self identify to the rest of the cluster. Used for retrieving stats about the number of Redis Monitor instances currently running.

Note: On actions that take longer than the timeout, the key will expire and your stats may not be accurate until the main thread can heart beat again.

Redis

REDIS_HOST

Default: `'localhost'`

The Redis host.

REDIS_PORT

Default: `6379`

The port to use when connecting to the `REDIS_HOST`.

REDIS_DB

Default: `0`

The Redis database to use when connecting to the `REDIS_HOST`.

REDIS_LOCK_EXPIRATION

Default: `6`

The number of seconds a vacant worker lock will stay within Redis before becoming available to a new worker

Kafka

KAFKA_HOSTS

Default: `'localhost:9092'`

The Kafka host. May have multiple hosts separated by commas within the single string like `'h1:9092,h2:9092'`.

KAFKA_TOPIC_PREFIX

Default: `'demo'`

The Kafka Topic prefix to use when generating the outbound Kafka topics.

KAFKA_CONN_TIMEOUT

Default: `5`

How long to wait (in seconds) before timing out when trying to connect to the Kafka cluster.
KAFKA_APPID_TOPICS

Default: `False`

Flag to send data to both the firehose and Application ID specific Kafka topics. If set to `True`, results will be sent to both the `demo.outbound_firehose` **and** `demo.outbound_<appid>` Kafka topics, where `<appid>` is the Application ID used to submit the request. This is useful if you have many applications utilizing your cluster but only would like to listen to results for your specific application.

KAFKA_PRODUCER_BATCH_LINGER_MS

Default: `25`

The time to wait between batching multiple requests into a single one sent to the Kafka cluster.

KAFKA_PRODUCER_BUFFER_BYTES

Default: `4 * 1024 * 1024`

The size of the TCP send buffer when transmitting data to Kafka

Zookeeper

ZOOKEEPER_ASSIGN_PATH

Default: `/scrapy-cluster/crawler/`

The location to store Scrapy Cluster domain specific configuration within Zookeeper. Should be the same as the crawler *settings*.

ZOOKEEPER_ID

Default: `all`

The file identifier to read crawler specific configuration from. This file is located within the `ZOOKEEPER_ASSIGN_PATH` folder above. Should be the same as the crawler *settings*.

ZOOKEEPER_HOSTS

Default: `localhost:2181`

The zookeeper host to connect to. Should be the same as the crawler *settings*.

Plugins

PLUGIN_DIR

Default: `'plugins/'`

The folder containing all of the Kafka Monitor plugins. **PLUGINS**

Default:

```
{
    'plugins.info_monitor.InfoMonitor': 100,
    'plugins.stop_monitor.StopMonitor': 200,
    'plugins.expire_monitor.ExpireMonitor': 300,
    'plugins.stats_monitor.StatsMonitor': 400,
    'plugins.zookeeper_monitor.ZookeeperMonitor': 500,
}
```

The default plugins loaded for the Redis Monitor. The syntax for this dictionary of settings is `'<folder>.<file>.<class_name>': <rank>`. Where lower ranked plugin API's are validated first.

Logging

LOGGER_NAME

Default: `'redis-monitor'`

The logger name.

LOG_DIR

Default: `'logs'`

The directory to write logs into. Only applicable when `LOG_STDOUT` is set to `False`.

LOG_FILE

Default: `'redis_monitor.log'`

The file to write the logs into. When this file rolls it will have `.1` or `.2` appended to the file name. Only applicable when `LOG_STDOUT` is set to `False`.

LOG_MAX_BYTES

Default: `10 * 1024 * 1024`

The maximum number of bytes to keep in the file based log before it is rolled.

LOG_BACKUPS

Default: `5`

The number of rolled file logs to keep before data is discarded. A setting of `5` here means that there will be one main log and five rolled logs on the system, generating six log files total.

LOG_STDOUT

Default: `True`

Log to standard out. If set to `False`, will write logs to the file given by the `LOG_DIR/LOG_FILE`

LOG_JSON

Default: `False`

Log messages will be written in JSON instead of standard text messages.

LOG_LEVEL

Default: `'INFO'`

The log level designated to the logger. Will write all logs of a certain level and higher.

Note: More information about logging can be found in the utilities *Log Factory* documentation.

Stats

STATS_TOTAL

Default: `True`

Calculate total receive and fail stats for the Redis Monitor.

STATS_PLUGINS

Default: `True`

Calculate total receive and fail stats for each individual plugin within the Redis Monitor.

STATS_CYCLE

Default: `5`

How often to check for expired keys and to roll the time window when doing stats collection.

STATS_DUMP

Default: `60`

Dump stats to the logger every X seconds. If set to `0` will not dump statistics.

STATS_DUMP_CRAWL

Default: `True`

Dump *statistics* collected by the Scrapy Cluster Crawlers. The crawlers may be spread out across many machines, and the log dump of their statistics is consolidated and done in a single place where the Redis Monitor is installed. Will be dumped at the same interval the `STATS_DUMP` is set to.

STATS_DUMP_QUEUE

Default: `True`

Dump queue metrics about the real time backlog of the Scrapy Cluster Crawlers. This includes queue length, and total number of domains currently in the backlog. Will be dumped at the same interval the `STATS_DUMP` is set to.

STATS_TIMES

Default:

```
[
    'SECONDS_15_MINUTE',
    'SECONDS_1_HOUR',
    'SECONDS_6_HOUR',
    'SECONDS_12_HOUR',
    'SECONDS_1_DAY',
    'SECONDS_1_WEEK',
]
```

Rolling time window settings for statistics collection, the above settings indicate stats will be collected for the past 15 minutes, the past hour, the past 6 hours, etc.

Note: For more information about stats collection, please see the *Stats Collector* documentation.

Design Learn about the design considerations for the Redis Monitor

Quick Start How to use and run the Redis Monitor

Plugins Gives an overview of the different plugin components within the Redis Monitor, and how to make your own.

Settings Explains all of the settings used by the Redis Monitor

CHAPTER 5

Rest

The Rest service allows for restful interactions with Scrapy Cluster, and utilitizes Kafka to allow you to submit crawl requests or get information about its current state.

Design

The Rest service is designed to act as a gateway into your Scrapy Cluster in order to provide services for requests and information that otherwise may not speak to Kafka. It provides a pass-through style interface, allowing the *Kafka Monitor* to handle the actual validation of the request.

The Rest service is one of the newer components to Scrapy Cluster, and provides the ability for a service to provide arbitrary JSON to be passed to the cluster, and to request information for longer jobs that traditionally time out during the execution of a call.

The ability for Scrapy Cluster to provide restful interaction opens up the door to User Interfaces, http/https requests, and generally more usability of the system. Built upon a simple Flask framework, the Rest service provides a foundation for Restful Kafka interaction and with Scrapy Cluster.

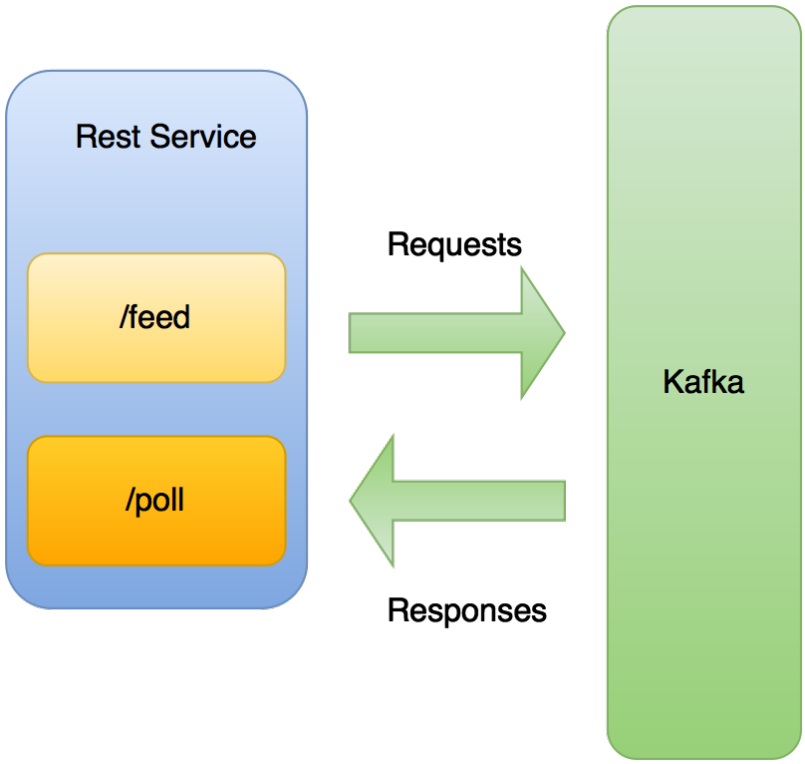
Quick Start

First, create a `localsettings.py` to track your overridden custom settings. You can override any setting you find within the Rest service's `settings.py`, and a typical file may look like the following:

```
REDIS_HOST = 'scdev'
KAFKA_HOSTS = 'scdev:9092'
```

rest_service.py

The main mode of operation for the Rest service is to generate a Restful endpoint for external applications to interact with your cluster.




```
$ python rest_service.py
```

Typical Usage

Typical usage of the Rest service is done in conjunction with both the Kafka Monitor and Redis Monitor. The Rest service is a gateway for interacting with both of those components, and provides little functionality when used by itself.

Warning: This guide assumes you have both a running Kafka Monitor and Redis Monitor already. Please see the Kafka Monitor *Quick Start* or the Redis Monitor *Quick Start* for more information.

Open two terminals.

Terminal 1:

Run the rest service

```
$ python rest_service.py
```

Terminal 2:

Curl the basic endpoint of the component.

```
$ curl scdev:5343
{
  "kafka_connected": true,
  "my_id": "34806877e13f",
  "node_health": "GREEN",
```

```
  "redis_connected": true,
  "uptime_sec": 23
}
```

You should see a log message come through Terminal 1 stating the message was received and the data transmitted back.

```
2016-11-11 09:51:26,358 [rest-service] INFO: 'index' endpoint called
```

Feed a request

```
$ curl scdev:5343/feed -H "Content-Type: application/json" -d '{"uuid":"abc123",
↪"appid":"stuff"}'
```

The request provided is outlined at the Kafka Monitor's *API*, when using the `feed` endpoint you need to ensure your request conforms to specification outlined on that page.

You should see a log message come through Terminal 1 stating the message was received and the data transmitted back.

```
2016-11-06 14:26:05,621 [rest-service] INFO: 'feed' endpoint called
```

After a short time, you should see the result come through your curl command in terminal 2.

```
{
  "data": {
    "appid": "stuff",
    "crawler": {
      "machines": {
        "count": 0
      },
      "queue": {
        "total_backlog": 0
      },
      "spiders": {
        "link": {
          "count": 1
        },
        "total_spider_count": 1,
        "unique_spider_count": 1
      }
    },
    "kafka-monitor": {
      "fail": {
        "21600": 1,
        "3600": 1,
        "43200": 1,
        "604800": 1,
        "86400": 1,
        "900": 1,
        "lifetime": 1
      },
      "plugins": {
        "StatsHandler": {
          "21600": 2,
          "3600": 2,
          "43200": 2,
          "604800": 2,
```

```
          "86400": 2,
          "900": 2,
          "lifetime": 2
        }
      },
      "total": {
        "21600": 3,
        "3600": 3,
        "43200": 3,
        "604800": 3,
        "86400": 3,
        "900": 3,
        "lifetime": 3
      }
    },
    "redis-monitor": {
      "nodes": {
        "afa660f7e348": [
          "3333a4d63704"
        ]
      },
      "plugins": {
        "StatsMonitor": {
          "21600": 2,
          "3600": 2,
          "43200": 2,
          "604800": 2,
          "86400": 2,
          "900": 2,
          "lifetime": 2
        }
      },
      "total": {
        "21600": 2,
        "3600": 2,
        "43200": 2,
        "604800": 2,
        "86400": 2,
        "900": 2,
        "lifetime": 2
      }
    },
    "server_time": 1478714930,
    "stats": "all",
    "uuid": "abc123"
  },
  "error": null,
  "status": "SUCCESS"
}
```


At this point, your Rest service is operational.

API

The Rest service component exposes both the Kafka Monitor *API* and a mechanism for getting information that may take a significant amount of time.

Standardization

Aside from the `Index` endpoint which provides information about the service itself, all objects receive the following wrapper:

```
{
    "status": <status>,
    "data": <data>,
    "error" <error>
}
```

The `status` key is a human readable `SUCCESS` or `FAILURE` string, used for quickly understanding what happened. The `data` key provides the data that was requested, and the `error` object contains information about any internal errors that occurred while processing your request.

Accompanying this object are standard `200`, `400`, `404`, or `500` response codes. Whenever you encounter non-standard response codes from the Rest service you should receive the accompanying information provided both in the response from the service itself and/or in the service logs. Problems might include:

- Improperly formatted JSON or content headers are incorrect

```
{
  "data": null,
  "error": {
    "message": "The payload must be valid JSON."
  },
  "status": "FAILURE"
}
```

- Invalid JSON structure for desired endpoint

```
{
  "data": null,
  "error": {
    "cause": "Additional properties are not allowed (u'crazykey' was␣
↪unexpected)",
    "message": "JSON did not validate against schema."
  },
  "status": "FAILURE"
}
```

- Unexpected Exception within the service itself

```
{
  "data": null,
  "error": {
    "message": "Unable to connect to Kafka"
  },
  "status": "FAILURE"
}
```

- The desired endpoint does not exist

```
{
  "data": null,
  "error": {
    "message": "The desired endpoint does not exist"
  },
```

```
  "status": "FAILURE"
}
```

For all of these and any other error, you should expect diagnostic information to be either in the response or in the logs.

Index Endpoint

The index endpoint allow you to obtain basic information about the status of the Rest Service, its uptime, and connection to critical components.

Headers Expected: `None`

Method Types: `GET`, `POST`

URI: `/`

Example

```
$ curl scdev:5343
```

Responses

Unable to connect to Redis or Kafka

```
{
  "kafka_connected": false,
  "my_id": "d209adf2aa01",
  "node_health": "RED",
  "redis_connected": false,
  "uptime_sec": 143
}
```

Able to connect to Redis or Kafka, but not both at the same time

```
{
  "kafka_connected": false,
  "my_id": "d209adf2aa01",
  "node_health": "YELLOW",
  "redis_connected": true,
  "uptime_sec": 148
}
```

Able to connect to both Redis and Kafka, fully operational

```
{
  "kafka_connected": true,
  "my_id": "d209adf2aa01",
  "node_health": "GREEN",
  "redis_connected": true,
  "uptime_sec": 156
}
```

Here, a human readable `node_health` field is provided, as well as information about which service is unavailable at the moment. If the component is not `GREEN` in health you should troubleshoot your configuration.

Feed Endpoint

The feed endpoint transmits your request into JSON that will be sent to Scrapy Cluster. It follows the *API* exposed by the Kafka Monitor, and acts as a pass-through to that service. The assumptions made are as follows:

- Crawl requests made to the cluster do not expect a response back via Kafka
- Other requests like Action or Stat expect a response within a designated period of time. If a response is expected but not received, a *Poll* is used to further poll for the desired response.

Headers Expected: `Content-Type: application/json`

Method Types: `POST`

URI: `/feed`

Data: Valid JSON data for the request

Examples

Feed a crawl request

```
$ curl scdev:5343/feed -H "Content-Type: application/json" -d '{"url":"istresearch.com
↪", "appid":"madisonTest", "crawlid":"abc123"}'
```

Feed a Stats request

```
$ curl scdev:5343/feed -H "Content-Type: application/json" -d '{"uuid":"abc123",
↪"appid":"stuff"}'
```

In both of these cases, we are translating the JSON required by the Kafka Monitor into a Restful interface request. You may use all of the API's exposed by the Kafka Monitor here when creating your request.

Responses

The responses from the feed endpoint should match both the standardized object and the expected return value from the Kafka Monitor API.

Successful submission of a crawl request.

```
{
  "data": null,
  "error": null,
  "status": "SUCCESS"
}
```

Successful response from a Redis Monitor request

```
{
  "data": {... data here ...},
  "error": null,
  "status": "SUCCESS"
}
```

Unsuccessful response from a Redis Monitor request

```
{
  "data": {
    "poll_id": <uuid of request>
  },
  "error": null,
  "status": "SUCCESS"
}
```

In this case, the response was unable to be obtained within the *response time* and the `poll_id` should be used in the *poll* request below.

Poll Endpoint

The Poll endpoint provides the ability to retrieve data from long running requests that might take longer than the desired *response time* configured for the service. This is useful when conducting statistics gathering or pruning data via action requests.

Headers Expected: `Content-Type: application/json`

Method Types: `POST`

URI: `/poll`

Data: Valid JSON data for the request

JSON Schema

```
{
    "type": "object",
    "properties": {
        "poll_id": {
            "type": "string",
            "minLength": 1,
            "maxLength": 100,
            "description": "The poll id to retrieve"
        }
    },
    "required": [
        "poll_id"
    ],
    "additionalProperties": false
}
```

Example

```
$ curl scdev:5343/poll -XPOST -H "Content-Type: application/json" -d '{"poll_id":
↪"abc123"}'
```

Responses

Successfully found a poll that has been completed, but was not returned initially during the request

```
{
  "data": {... data here ...},
  "error": null,
  "status": "SUCCESS"
}
```

Did not find the results for the `poll_id`

```
{
  "data": null,
  "error": {
    "message": "Could not find matching poll_id"
  },
  "status": "FAILURE"
}
```

Note that a failure to find the `poll_id` may indicate one of two things:

- The request has not completed yet
- The request incurred a failure within another component of Scrapy Cluster

Settings

This page covers the various settings contained within the Rest service. The sections are broken down by functional component.

Core

FLASK_LOGGING_ENABLED

Default: `True`

Enable Flask application logging, independent of Scrapy Cluster logging.

FLASK_PORT

Default: `5343`

The default port for the Rest service to listen on. The abbreviation `SC` equals `5343` in hexidecimal.

SLEEP_TIME

Default: `0.1`

The number of seconds the main threads will sleep between checking for items.

HEARTBEAT_TIMEOUT

Default: `120`

The amount of time the heartbeat key the Rest service instance lives to self identify to the rest of the cluster. Used for retrieving stats about the number of Rest service instances currently running.

DAEMON_THREAD_JOIN_TIMEOUT

Default: `10`

The amount of time provded to the daemon threads to safely close when the instance is shut down.
WAIT_FOR_RESPONSE_TIME

Default: `5`

The amount of time the Rest service will wait for a response from Kafka before converting the request into a `poll`.

SCHEMA_DIR

Default: `'schemas/'`

The directory for the Rest Service to automatically load JSON Schemas that can be used by the application.

Redis

REDIS_HOST

Default: `'localhost'`

The Redis host.

REDIS_PORT

Default: `6379`

The port to use when connecting to the `REDIS_HOST`.

REDIS_DB

Default: `0`

The Redis database to use when connecting to the `REDIS_HOST`.

Kafka

KAFKA_HOSTS

Default: `'localhost:9092'`

The Kafka host. May have multiple hosts separated by commas within the single string like `'h1:9092,h2:9092'`.

KAFKA_TOPIC_PREFIX

Default: `'demo'`

The Kafka Topic prefix used for listening for Redis Monitor results.

KAFKA_FEED_TIMEOUT

Default: `10`

How long to wait (in seconds) before timing out when trying to feed a JSON string into the `KAFKA_INCOMING_TOPIC`

KAFKA_CONSUMER_AUTO_OFFSET_RESET

Default: `'latest'`

When the Kafka Consumer encounters and unexpected error, move the consumer offset to the 'latest' new message, or the 'earliest' available.

KAFKA_CONSUMER_TIMEOUT

Default: `50`

Time in ms spent to wait for a new message during a `feed` call that expects a response from the Redis Monitor

KAFKA_CONSUMER_COMMIT_INTERVAL_MS

Default: `5000`

How often to commit Kafka Consumer offsets to the Kafka Cluster

KAFKA_CONSUMER_AUTO_COMMIT_ENABLE

Default: `True`

Automatically commit Kafka Consumer offsets.

KAFKA_CONSUMER_FETCH_MESSAGE_MAX_BYTES

Default: `10 * 1024 * 1024`

The maximum size of a single message to be consumed by the Kafka Consumer. Defaults to 10 MB

KAFKA_CONSUMER_SLEEP_TIME

Default: `1`

The length of time to sleep by the main thread before checking for new Kafka messages

KAFKA_PRODUCER_TOPIC

Default: `demo.incoming`

The topic that the Kafka Monitor is litening for requests on.

KAFKA_PRODUCER_BATCH_LINGER_MS

Default: `25`

The time to wait between batching multiple requests into a single one sent to the Kafka cluster.

KAFKA_PRODUCER_BUFFER_BYTES

Default: `4 * 1024 * 1024`

The size of the TCP send buffer when transmitting data to Kafka

Logging

LOGGER_NAME

Default: `'rest-service'`

The logger name.

LOG_DIR

Default: `'logs'`

The directory to write logs into. Only applicable when `LOG_STDOUT` is set to `False`.

LOG_FILE

Default: `'rest_service.log'`

The file to write the logs into. When this file rolls it will have `.1` or `.2` appended to the file name. Only applicable when `LOG_STDOUT` is set to `False`.

LOG_MAX_BYTES

Default: `10 * 1024 * 1024`

The maximum number of bytes to keep in the file based log before it is rolled.

LOG_BACKUPS

Default: `5`

The number of rolled file logs to keep before data is discarded. A setting of `5` here means that there will be one main log and five rolled logs on the system, generating six log files total.

LOG_STDOUT

Default: `True`

Log to standard out. If set to `False`, will write logs to the file given by the `LOG_DIR/LOG_FILE`

LOG_JSON

Default: `False`

Log messages will be written in JSON instead of standard text messages.

LOG_LEVEL

Default: `'INFO'`

The log level designated to the logger. Will write all logs of a certain level and higher.

Note: More information about logging can be found in the utilities *Log Factory* documentation.

Design Learn about the design considerations for the Rest service

Quick Start How to use and run the Rest service

API The API the comes with the endpoint

Settings Explains all of the settings used by the Rest component

CHAPTER 6

Utilites

These documents provide information about the agnostic utility classes used within Scrapy Cluster that may be useful in other applications.

Argparse Helper

The Argparse Helper class is a small utility designed to help developers create cleaner and more maintainable `--help` messages when using Argparse sub-commands.

The default behavior when passing a `-h` or `--help` flag to your script prints out only minimal information about the subcommands. This utility class allows you to print out more information about how to run each of those subcommands from your main parser invocation.

The ArgparseHelper class does not have any direct method calls. Instead, the class is passed as a parameter when setting up your argument parser.

Usage

Using the ArgparseHelper class to display help information requires modification to the following two lines when setting up the parser.

```
import argparse
from scutils.argparse_helper import ArgparseHelper

parser = argparse.ArgumentParser(
    description='my_script.py: Used for processing widgets', add_help=False)
parser.add_argument('-h', '--help', action=ArgparseHelper,
                    help='show this help message and exit')
```

After the imports, we state that we do not want to use the default Argparse help functionality by specifying `add_help=False`. Next, we our own custom help message and pass the ArgparseHelper class for the `action` parameter.

You are now free to add subcommands with whatever parameters you like, and when users run `my_script.py -h` they will see the full list of acceptable ways to run your program.

Example

Put the following code into `example_ah.py`, or use the file located at `utils/examples/example_ah.py`

```
import argparse
from scutils.argparse_helper import ArgparseHelper

parser = argparse.ArgumentParser(
    description='example_ah.py: Prints various family members', add_help=False)
parser.add_argument('-h', '--help', action=ArgparseHelper,
                    help='show this help message and exit')
# use the default argparse setup, comment out the lines above
#parser = argparse.ArgumentParser(
#    description='example_ah.py: Prints various family members')

subparsers = parser.add_subparsers(help='commands', dest='command')

# args here are applied to all sub commands using the `parents` parameter
base_parser = argparse.ArgumentParser(add_help=False)
base_parser.add_argument('-n', '--name', action='store', required=True,
                         help="The name of the person running the program")

# subcommand 1, requires name of brother
bro_parser = subparsers.add_parser('bro', help='Prints only the brother\'s name',
                                   parents=[base_parser])
bro_parser.add_argument('-b', '--brother', action='store', required=True,
                        help="The brother's name")

# subcommand 2, requires name of sister and optionally mom
fam_parser = subparsers.add_parser('fam', help='Prints mom and sister\'s name',
                                   parents=[base_parser])
fam_parser.add_argument('-s', '--sister', action='store', required=True,
                        help="The sister's name")
fam_parser.add_argument('-m', '--mom', action='store', required=False,
                        default='Mom', help="The sister's name")

args = vars(parser.parse_args())

if args['command'] == 'bro':
    print "Your brother's name is " + args['brother']
elif args['command'] == 'fam':
    print "Your sister's name is " + args['sister'] + " and you call your "\
        "Mother '" + args['mom'] + "'"
```

Running `-h` from the base command prints out nicely formatted statements for all your subcommands.

```
$ python example_ah.py -h
usage: example_ah.py [-h] {bro,fam} ...

example_ah.py: Prints various family members

positional arguments:
  {bro,fam}   commands
    bro       Prints only the brother's name
```

```
    fam       Prints mom and sister's name

optional arguments:
  -h, --help  show this help message and exit

Command 'bro'
usage: example_ah.py bro [-h] -n NAME -b BROTHER

Command 'fam'
usage: example_ah.py fam [-h] -n NAME -s SISTER [-m MOM]
```

If you comment out the first two lines, and replace them with the simpler commented out line below it, you get the default Argparse behavior like shown below.

```
$ python example_ah.py -h
usage: example_ah.py [-h] {bro,fam} ...

example_ah.py: Prints various family members

positional arguments:
  {bro,fam}   commands
    bro       Prints only the brother's name
    fam       Prints mom and sister's name

optional arguments:
  -h, --help  show this help message and exit
```

You can see that this does not actually display to the user how to run your script sub-commands, and they have to type another `python example_ah.py bro -h` to see the arguments they need. Of course, you can always create your own `description` string for your default help message, but now you have to maintain the arguments to your commands in two places (the description string and in the code) instead of one.

The ArgparseHelper class allows you to keep your parameter documentation in one place, while allowing users running your script to see more detail about each of your subcommands.

Log Factory

The LogFactory is a Singleton Factory design that allows you to use a single logger across your python application.

This single log instance allows you to:

- Write to a file or standard out
- Write your logs in JSON or in a standard print format
- Can handle multiple independent processes writing to the same file
- Automatically handle rotating the file logs so they do not build up
- Not worry about multi threaded applications creating too many loggers resulting in duplicated data
- Register callbacks to provide additional functionality when certain logging levels or criteria are met

It is designed to be easy to use throughout your application, giving you a standard and flexible way to write your log data.

LogFactory.**get_instance**(*json=False*, *stdout=True*, *name='scrapy-cluster'*, *dir='logs'*, *file='main.log'*, *bytes=25000000*, *backups=5*, *level='INFO'*, *format='%(asctime)s [%(name)s] %(levelname)s: %(message)s'*, *propagate=False*, *include_extra=False*)

Parameters

- **stdout** – Flag to write logs to stdout or file
- **json** – Flag to write json logs with objects or just the messages
- **name** – The logger name
- **dir** – The directory to write logs into
- **file** – The file name
- **bytes** – The max file size in bytes
- **backups** – The number of backups to keep of the file
- **level** – The logging level string ['DEBUG', 'INFO', 'WARNING', 'ERROR' 'CRITICAL']
- **format** – The log format
- **propagate** – Allow the log to propagate to other ancestor loggers
- **include_extra** – When not logging json, include the 'extra' param in the output

Returns The log instance to use

Usage

Basic Usage of the LogFactory:

```
>>> from scutils.log_factory import LogFactory
>>> logger = LogFactory.get_instance()
>>> logger.info("Hello")
2015-11-17 15:55:29,807 [scrapy-cluster] INFO: Hello
```

Please keep in mind the default values above, and be sure to set them correctly the **very first time** the LogFactory is called. Below is an example of **incorrect** usage.

```
>>> from scutils.log_factory import LogFactory
>>> logger = LogFactory.get_instance(json=True, name='logger1', level='DEBUG')
>>> logger.debug("test")
{"message": "Logging to stdout", "logger": "logger1", "timestamp": "2015-11-
↪17T21:13:59.816441Z", "level": "DEBUG"}
>>> # incorrectly changing the logger in mid program
...
>>> new_logger = LogFactory.get_instance(name='newlogger', json=False)
>>> new_logger.info("test")
{"message": "test", "logger": "logger1", "timestamp": "2015-11-17T21:14:47.657768Z",
↪"level": "INFO"}
```

Why does this happen? Behind the scenes, the LogFactory is trying to ensure that no matter where or when you call the `get_instance()` method, it returns to you the same logger you asked for originally. This allows multithreaded applications to instantiate the logger all the same way, without having to worry about duplicate logs showing up in your output. Once you have your logger object, the following standard logger methods are available:

debug(*message*, *extra={}*)
Tries to write a debug level log message

Parameters

- **message** – The message to write to the logs
- **extra** – A dictionary of extra fields to include with the written object if writing in json

info(*message*, *extra={}*)
Tries to write an info level log message

Parameters

- **message** – The message to write to the logs
- **extra** – A dictionary of extra fields to include with the written object if writing in json

warn(*message*, *extra={}*)
Tries to write a warning level log message

Parameters

- **message** – The message to write to the logs
- **extra** – A dictionary of extra fields to include with the written object if writing in json

warning(*message*, *extra={}*)
Tries to write a warning level log message. Both of these warning methods are only supplied for convenience

Parameters

- **message** – The message to write to the logs
- **extra** – A dictionary of extra fields to include with the written object if writing in json

error(*message*, *extra={}*)
Tries to write an error level log message

Parameters

- **message** – The message to write to the logs
- **extra** – A dictionary of extra fields to include with the written object if writing in json

critical(*message*, *extra={}*)
Tries to write a critical level log message

Parameters

- **message** – The message to write to the logs
- **extra** – A dictionary of extra fields to include with the written object if writing in json

When setting you application log level, you determine what amount of logs it will produce. Typically the most verbose logging is done in `DEBUG`, and increasing the log level decreases the amount of logs generated.

App Log Level	Output Log levels
DEBUG	DEBUG, INFO, WARNING, ERROR, CRITICAL
INFO	INFO, WARNING, ERROR, CRITICAL
WARNING	WARNING, ERROR, CRITICAL
ERROR	ERROR, CRITICAL
CRITICAL	CRITICAL

Scrapy Cluster's many components use arguments from the command line to set common properties of the its logger. You may want to use an argument parser in your application to set common things like:

- The log level
- Whether to write in JSON output or formatted print statements
- Whether to write to a file or not

If you would like to register a callback for extra functionality, you can use the following function on your logger object.

register_callback(log_level, fn, criteria=None):

Parameters

- **log_level** – The log level expression the callback should act on
- **fn** – the function to call
- **criteria** – a dictionary of values providing additional matching criteria before the callback is called

The callback takes arguments in the form of the log level and a criteria dictionary. The log level may be of the following forms:

- `'INFO'`, `'ERROR'`, `'DEBUG'`, etc - the callback is only called when the exact log level message is used
- `'<=INFO'`, `'>DEBUG'`, `'>=WARNING'`, etc - parameterized callbacks, allowing the callback to execute in greater than or less than the desired log level
- `'*'` - the callback is called regardless of log level

For additional filtering requirements you may also use the `criteria` parameter, in which the criteria **must be a subset** of the `extras` originally passed into the logging method of choice.

For example:

```
{'a': 1} subset of {'a': 1, 'b': 2, 'c': 3}
```

Both the log level and criteria are considered before executing the callback, basic usage is as follows:

```
>>> from scutils.log_factory import LogFactory
>>> logger = LogFactory.get_instance()
>>> def call_me(message, extras):
...   print("callback!", message, extras)
...
>>> logger.register_callback('>=INFO', call_me)
>>> logger.debug("no callback")
>>> logger.info("check callback")
2017-02-01 15:19:48,466 [scrapy-cluster] INFO: check callback
('callback!', 'check callback', {})
```

In this example both the log message was written, the callback was called, and the three values were printed.

Note: Your callback function must be declared outside of your Class scope, or use the `@staticmethod` decorator. Typically this means putting the function at the top of your python file outside of any class declarations.

Example

Add the following python code to a new file, or use the script located at `utils/examples/example_lf.py`:

```
import argparse
from scutils.log_factory import LogFactory
parser = argparse.ArgumentParser(description='Example logger.')
parser.add_argument('-ll', '--log-level', action='store', required=False,
                    help="The log level", default='INFO',
                    choices=['DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'])
parser.add_argument('-lf', '--log-file', action='store_const',
                    required=False, const=False, default=True,
                    help='Log the output to the file. Otherwise logs to stdout')
parser.add_argument('-lj', '--log-json', action='store_const',
                    required=False, const=True, default=False,
                    help="Log the data in JSON format")
parser.add_argument('-ie', '--include-extra', action='store_const', const=True,
                        default=False, help="Print the 'extra' dict if not logging"
                        " to json")
args = vars(parser.parse_args())
logger = LogFactory.get_instance(level=args['log_level'], stdout=args['log_file'],
                    json=args['log_json'], include_extra=args['include_extra'])

my_var = 1

def the_callback(log_message, log_extras):
    global my_var
    my_var += 5

def the_callback_2(log_message, log_extras):
    global my_var
    my_var *= 2

logger.register_callback('DEBUG', the_callback)
logger.register_callback('WARN', the_callback_2, {'key':"value"})

logger.debug("debug output 1")
logger.warn("warn output", extra={"key":"value", "key2":"value2"})
logger.warn("warn output 2")
logger.debug("debug output 2")
logger.critical("critical fault, closing")
logger.debug("debug output 3")
sum = 2 + 2
logger.info("Info output closing.", extra={"sum":sum})
logger.error("Final var value", extra={"value": my_var})
```

Let's assume you now have a file named `example_lf.py`, run the following commands:

```
$ python example_lf.py --help
usage: example_lf.py [-h] [-ll {DEBUG,INFO,WARNING,ERROR,CRITICAL}] [-lf]
                     [-lj]

Example logger.

optional arguments:
  -h, --help            show this help message and exit
  -ll {DEBUG,INFO,WARNING,ERROR,CRITICAL}, --log-level {DEBUG,INFO,WARNING,ERROR,
↪CRITICAL}
                        The log level
  -lf, --log-file       Log the output to the file. Otherwise logs to stdout
  -lj, --log-json       Log the data in JSON format
```

```
$ python example_lf.py --log-level DEBUG
# Should write all log messages above
2017-02-01 15:27:48,814 [scrapy-cluster] DEBUG: Logging to stdout
2017-02-01 15:27:48,814 [scrapy-cluster] DEBUG: debug output 1
2017-02-01 15:27:48,814 [scrapy-cluster] WARNING: warn output
2017-02-01 15:27:48,814 [scrapy-cluster] WARNING: warn output 2
2017-02-01 15:27:48,814 [scrapy-cluster] DEBUG: debug output 2
2017-02-01 15:27:48,814 [scrapy-cluster] CRITICAL: critical fault, closing
2017-02-01 15:27:48,814 [scrapy-cluster] DEBUG: debug output 3
2017-02-01 15:27:48,814 [scrapy-cluster] INFO: Info output closing.
2017-02-01 15:27:48,814 [scrapy-cluster] ERROR: Final var value
```

```
$ python example_lf.py --log-level INFO --log-json
# Should log json object of "INFO" level or higher
{"message": "warn output", "key2": "value2", "logger": "scrapy-cluster", "timestamp":
↪"2017-02-01T20:29:24.548895Z", "key": "value", "level": "WARNING"}
{"message": "warn output 2", "logger": "scrapy-cluster", "timestamp": "2017-02-
↪01T20:29:24.549302Z", "level": "WARNING"}
{"message": "critical fault, closing", "logger": "scrapy-cluster", "timestamp": "2017-
↪02-01T20:29:24.549420Z", "level": "CRITICAL"}
{"message": "Info output closing.", "sum": 4, "logger": "scrapy-cluster", "timestamp
↪": "2017-02-01T20:29:24.549510Z", "level": "INFO"}
{"message": "Final var value", "logger": "scrapy-cluster", "timestamp": "2017-02-
↪01T20:29:24.549642Z", "level": "ERROR", "value": 2}
```

Notice that the extra dictionary object we passed into the two logs above is now in our json logging output

```
$ python example_lf.py --log-level CRITICAL --log-json --log-file
# Should log only one critical message to our file located at logs/
$ tail logs/main.log
{"message": "critical fault, closing", "logger": "scrapy-cluster", "timestamp": "2017-
↪02-01T20:30:05.484337Z", "level": "CRITICAL"}
```

You can also use the `include_extra` flag when instantiating the logger to print the dictionary even if you are not logging via json.

```
$ python example_lf.py -ll DEBUG -ie
2017-02-01 15:31:07,284 [scrapy-cluster] DEBUG: Logging to stdout
2017-02-01 15:31:07,284 [scrapy-cluster] DEBUG: debug output 1
2017-02-01 15:31:07,284 [scrapy-cluster] WARNING: warn output {'key2': 'value2', 'key
↪': 'value'}
2017-02-01 15:31:07,284 [scrapy-cluster] WARNING: warn output 2
2017-02-01 15:31:07,284 [scrapy-cluster] DEBUG: debug output 2
2017-02-01 15:31:07,284 [scrapy-cluster] CRITICAL: critical fault, closing
2017-02-01 15:31:07,284 [scrapy-cluster] DEBUG: debug output 3
2017-02-01 15:31:07,284 [scrapy-cluster] INFO: Info output closing. {'sum': 4}
2017-02-01 15:31:07,285 [scrapy-cluster] ERROR: Final var value {'value': 22}
```

Notice here that both our `DEBUG` callback was called three times, and the single `WARN` callback was called once, causing our final `my_var` variable to equal `22`.

The LogFactory hopefully will allow you to easily debug your application while at the same time be compatible with JSON based log architectures and production based deployments. For more information please refer to *Integration with ELK*

Method Timer

The Method Timer module is for executing python methods that need to be cancelled after a certain period of execution. This module acts as a decorator to existing functions with or without arguments, allowing you to escape out of the method if it does not finish within your desired time period.

timeout(*timeout_time*, *default*)

Parameters

- **timeout_time** – The number of seconds to wait before returning the default value
- **default** – The default value returned by the method if the timeout is called

Usage

Use above a function definition like so:

```
>>> from scutils.method_timer import MethodTimer
>>> @MethodTimer.timeout(5, "timeout")
>>> def my_function():
>>>     # your complex function here
```

The Method Timer module relies on python signals to alert the process that the method has not completed within the desired time window. This means that it is not designed to be run within a multi-threaded application, as the signals raised are not depended on the thread that called it.

Use this class as a convenience for connection establishment, large processing that is time dependent, or any other use case where you need to ensure your function completes within a desired time window.

Example

Put the following code into a file named `example_mt.py`, or use the one located at `utils/examples/example_my.py`

```
from scutils.method_timer import MethodTimer
from time import sleep

@MethodTimer.timeout(3, "did not finish")
def timeout():
    sleep(5)
    return "finished!"

return_value = timeout()
print return_value
```

```
$ python example_mt.py
did not finish
```

Within the decorator declaration if you set the timeout value to 6, you will see `finished!` displayed instead.

You can also dynamically adjust how long you would like your method to sleep, by defining your function underneath an existing one. Take the following code as an example:

```
from scutils.method_timer import MethodTimer
from time import sleep

def timeout(sleep_time, wait_time, default, mul_value):
    # define a hidden method to sleep and wait based on parameters
    @MethodTimer.timeout(wait_time, default)
    def _hidden(m_value):
        sleep(sleep_time)
        return m_value * m_value
    # call the newly declared function
    return _hidden(mul_value)

print timeout(5, 3, "did not finish 2*2", 2)
print timeout(3, 5, "did not finish 3*3", 3)
print timeout(2, 1, "did not finish 4*4", 4)
```

Now we have a hidden method underneath our main one that will adjust both its timeout period and return value based on parameters passed into the parent function. In this case, we try to compute the square of `mul_value` and return the result.

```
$ python example_mt.py
did not finish 2*2
9
did not finish 4*4
```

Redis Queue

A utility class that utilizes Pickle serialization by default to store and retrieve arbitrary sets of data in Redis. You may use another encoding mechanism as long as it supports both `dumps()` and `loads()` methods. The queues come in three basic forms:

- `RedisQueue` - A FIFO queue utilizing a Redis List
- `RedisStack` - A Stack implementation utilizing a Redis List
- `RedisPriorityQueue` - A Priority Queue utilizing a Redis Sorted Set. This is the queue utilized by the scheduler for prioritized crawls

All three of these classes can handle arbitrary sets of data, and handle the encoding and decoding for you.

class `RedisQueue` (*server*, *key*, *encoding=pickle*)

Parameters

- **`server`** – The established redis connection
- **`key`** – The key to use in Redis to store your data
- **`encoding`** – The serialization module to use

`push` (*item*)

Pushes an item into the Queue

Parameters **`item`** – The item to insert into the Queue

Returns None

`pop` (*timeout=0*)

Removes and returns an item from the Queue

Parameters **timeout** (*int*) – If greater than 0, use the Redis blocking pop method for the specified timeout.

Returns None if no object can be found, otherwise returns the object.

clear()
Removes all data in the Queue.

Returns None

__len__()
Get the number of items in the Queue.

Returns The number of items in the RedisQueue

Usage `len(my_queue_instance)`

class RedisStack(*server*, *key*, *encoding=pickle*)

Parameters

- **server** – The established redis connection
- **key** – The key to use in Redis to store your data
- **encoding** – The serialization module to use

push(*item*)
Pushes an item into the Stack

Parameters **item** – The item to insert into the Stack

Returns None

pop(*timeout=0*)
Removes and returns an item from the Stack

Parameters **timeout** (*int*) – If greater than 0, use the Redis blocking pop method for the specified timeout.

Returns None if no object can be found, otherwise returns the object.

clear()
Removes all data in the Stack.

Returns None

__len__()
Get the number of items in the Stack.

Returns The number of items in the RedisStack

Usage `len(my_stack_instance)`

class RedisPriorityQueue(*server*, *key*, *encoding=pickle*)

Parameters

- **server** – The established redis connection
- **key** – The key to use in Redis to store your data
- **encoding** – The serialization module to use

push(*item*, *priority*)
Pushes an item into the PriorityQueue

Parameters

- **item** – The item to insert into the Priority Queue
- **priority** (*int*) – The priority of the item. Higher numbered items take precedence over lower priority items.

Returns None

pop(*timeout=0*)
Removes and returns an item from the PriorityQueue

Parameters **timeout** (*int*) – Not used

Returns None if no object can be found, otherwise returns the object.

clear()
Removes all data in the PriorityQueue.

Returns None

__len__()
Get the number of items in the PriorityQueue.

Returns The number of items in the RedisPriorityQueue

Usage `len(my_pqueue_instance)`

Usage

You can use any of the three classes in the following way, you just need to have a valid Redis connection variable.

```
>>> import redis
>>> import ujson
>>> from scutils.redis_queue import RedisStack
>>> redis_conn = redis.Redis(host='scdev', port=6379)
>>> queue = RedisStack(redis_conn, "stack_key", encoding=ujson))
>>> queue.push('item1')
>>> queue.push(['my', 'array', 'here'])
>>> queue.pop()
[u'my', u'array', u'here']
>>> queue.pop()
u'item1'
```

In the above example, we now have a host at `scdev` that is using the key called `stack_key` to store our data encoded using the `ujson` module.

Example

In this example lets create a simple script that changes what type of Queue we use when pushing three items into it.

```
import redis
from scutils.redis_queue import RedisStack, RedisQueue, RedisPriorityQueue
import argparse

# change these for your Redis host
host = 'scdev'
port = 6379
redis_conn = redis.Redis(host=host, port=port)

parser = argparse.ArgumentParser(description='Example Redis Queues.')
```

```
group = parser.add_mutually_exclusive_group(required=True)
group.add_argument('-q', '--queue', action='store_true', help="Use a RedisQueue")
group.add_argument('-s', '--stack', action='store_true',
                        help="Use a RedisStack")
group.add_argument('-p', '--priority', action='store_true',
                        help="Use a RedisPriorityQueue")

args = vars(parser.parse_args())

if args['queue']:
    queue = RedisQueue(redis_conn, "my_key")
elif args['stack']:
    queue = RedisStack(redis_conn, "my_key")
elif args['priority']:
    queue = RedisPriorityQueue(redis_conn, "my_key")

print "Using " + queue.__class__.__name__

if isinstance(queue, RedisPriorityQueue):
    queue.push("item1", 50)
    queue.push("item2", 100)
    queue.push("item3", 20)
else:
    queue.push("item1")
    queue.push("item2")
    queue.push("item3")

print "Pop 1 " + queue.pop()
print "Pop 2 " + queue.pop()
print "Pop 3 " + queue.pop()
```

Save the file as `example_rq.py` or use the one located at `utils/examples/example_rq.py`, and now lets run the different tests.

As a queue:

```
$ python example_rq.py -q
Using RedisQueue
Pop 1 item1
Pop 2 item2
Pop 3 item3
```

As a stack:

```
$ python example_rq.py -s
Using RedisStack
Pop 1 item3
Pop 2 item2
Pop 3 item1
```

As a priority queue:

```
$ python example_rq.py -p
Using RedisPriorityQueue
Pop 1 item2
Pop 2 item1
Pop 3 item3
```

The great thing about these Queue classes is that if your process dies, your data still remains in Redis! This allows you to restart your process and it can continue where it left off.

Redis Throttled Queue

This utility class is a wrapper around the *Redis Queue* utility class. Using Redis as a Queue is great, but how do we protect against concurrent data access, if we want more than one process to be pushing and popping from the queue? This is where the RedisThrottleQueue comes in handy.

The `RedisThrottleQueue` allows for multi-threaded and/or multiprocess access to the same Redis key being used as a `RedisQueue`. The throttle acts as a wrapper around the core queue's `pop()` method, and permits access to the data only when the throttle allows it. This is highly utilized by Scrapy Cluster's Distributed Scheduler for *controlling* how fast the cluster crawls a particular site.

Redis already has an official RedLock implementation for doing distributed locks, so why reinvent the wheel? Our locking implementation focuses around Scrapy Cluster's use case, which is rate limiting Request pops to the spiders. We do not need to lock the key when pushing new requests into the queue, only for when we wish to read from it.

If you lock the key, it prevents all other processes from *both* pushing into and popping from the queue. The only guarantee we need to make is that for any particular queue, the transaction to `pop()` is atomic. This means that only one process at a time can pop the queue, and it guarantees that no other process was able to pop that same item.

In Scrapy Cluster's use case, it means that when a Spider process says "Give me a Request", we can be certain that no other Spider process has received that exact request. To get a new request, the Throttled Queue tries to execute a series of operations against the lock before any other process can. If another process succeeds in performing the same series of operations before it, that process is returned the item and former is denied. These steps are repeated for all processes trying to pop from the queue at any given time.

In practice, the Redlock algorithm has many safeguards in place to do true concurrent process locking on keys, but as explained above, Scrapy Cluster does not need all of those extra features. Because those features significantly slow down the `pop()` mechanism, the `RedisThrottledQueue` was born.

class **RedisThrottledQueue**(*redisConn*, *myQueue*, *throttleWindow*, *throttleLimit*, *moderate=False*, *windowName=None*, *modName=None*, *elastic=False*, *elastic_buffer=0*)

Parameters

- **redisConn** – The Redis connection
- **myQueue** – The RedisQueue class instance
- **throttleWindow** (*int*) – The time window to throttle pop requests (in seconds).
- **throttleLimit** (*int*) – The number of queue pops allows in the time window
- **moderation** (*int*) – Queue pop requests are moderated to be evenly distributed throughout the window
- **windowName** (*str*) – Use a custom rolling window Redis key name
- **modName** (*str*) – Use a custom moderation key name in Redis
- **elastic** – When moderated and falling behind, break moderation to catch up to desired limit
- **elastic_buffer** – The threshold number for how close we should get to the limit when using the elastic catch up

push(**args*)

Parameters **args** – The arguments to pass through to the queue's `push()` method

Returns None

pop(**args*)

Parameters **args** – Pop arguments to pass through the the queue's `pop()` method

Returns None if no object can be found, otherwise returns the object.

clear()

Removes all data associated with the Throttle and Queue.

Returns None

__len__()

Returns The number of items in the Queue

Usage `len(my_throttled_queue)`

Usage

If you would like to throttle your Redis queue, you need to pass the queue in as part of the `RedisThrottleQueue` constructor.

```
>>> import redis
>>> from scutils.redis_queue import RedisPriorityQueue
>>> from scutils.redis_throttled_queue import RedisThrottledQueue
>>> redis_conn = redis.Redis(host='scdev', port=6379)
>>> queue = RedisPriorityQueue(redis_conn, 'my_key')
>>> t = RedisThrottledQueue(redis_conn, queue, 10, 5)
>>> t.push('item', 5)
>>> t.push('item2', 10)
>>> t.pop()
'item2'
>>> t.pop()
'item'
```

The throttle merely acts as a wrapper around your queue, returning items only when allowed. You can use the same methods the original `RedisQueue` provides, like `push()`, `pop()`, `clear()`, and `__len__`.

Note: Due to the distributed nature of the throttled queue, when using the `elastic=True` argument the queue must successfully pop the number of `limit` items before the elastic catch up will take effect.

Example

The Redis Throttled Queue really shines when multiple processes are trying to pop from the queue. There is a small test script under `utils/examples/example_rtq.py` that allows you to tinker with all of the different settings the throttled queue provides. The script is shown below for convenience.

```
import sys


def main():

    import argparse
    import redis
    import time
```

```
    import random

    import sys
    from os import path
    sys.path.append(path.dirname(path.dirname(path.abspath(__file__))))

    from scutils.redis_queue import RedisPriorityQueue
    from scutils.redis_throttled_queue import RedisThrottledQueue

    parser = argparse.ArgumentParser(description="Throttled Queue Test Script."
                    " Start either a single or multiple processes to see the "
                " throttled queue mechanism in action.")
    parser.add_argument('-r', '--redis-host', action='store', required=True,
                        help="The Redis host ip")
    parser.add_argument('-p', '--redis-port', action='store', default='6379',
                        help="The Redis port")
    parser.add_argument('-m', '--moderate', action='store_const', const=True,
                        default=False, help="Moderate the outbound Queue")
    parser.add_argument('-w', '--window', action='store', default=60,
                        help="The window time to test")
    parser.add_argument('-n', '--num-hits', action='store', default=10,
                        help="The number of pops allowed in the given window")
    parser.add_argument('-q', '--queue', action='store', default='testqueue',
                        help="The Redis queue name")
    parser.add_argument('-e', '--elastic', action='store_const', const=True,
                        default=False, help="Test variable elastic catch up"
                        " with moderation")

    args = vars(parser.parse_args())

    window = int(args['window'])
    num = int(args['num_hits'])
    host = args['redis_host']
    port = args['redis_port']
    mod = args['moderate']
    queue = args['queue']
    elastic = args['elastic']

    conn = redis.Redis(host=host, port=port)

    q = RedisPriorityQueue(conn, queue)
    t = RedisThrottledQueue(conn, q, window, num, mod, elastic=elastic)

    def push_items(amount):
        for i in range(0, amount):
            t.push('item-'+str(i), i)

    print "Adding", num * 2, "items for testing"
    push_items(num * 2)

    def read_items():
        print "Kill when satisfied ^C"
        ti = time.time()
        count = 0
        while True:
            item = t.pop()
            if item:
                print "My item", item, "My time:", time.time() - ti
```

```
                count += 1

            if elastic:
                time.sleep(int(random.random() * (t.moderation * 3)))

    try:
        read_items()
    except KeyboardInterrupt:
        pass
    t.clear()
    print "Finished"

if __name__ == "__main__":
    sys.exit(main())
```

The majority of this script allows you to alter how the throttled queue is created, most importantly allowing you to change the window, hits, and moderation flag. If you spin up more than one process, you will find that any single 'item' popped from the queue is given to only one process. The latter portion of the script either pushes items into the queue (`item-0` - `item-29`) or sits there and tries to `pop()` it.

Spinning up two instances with exactly the same settings will give you similar results to the following.

Warning: When spinning up multiple processes acting upon the same throttled queue, it is **extremely** important they have the exact same settings! Otherwise your processes will impose different restrictions on the throttle lock with undesired results.

Note: Note that each process inserts exactly the same items into the priority queue.

Process 1

```
$ python example_rtq.py -r scdev -w 30 -n 15 -m
Adding 30 items for testing
Kill when satisfied ^C
My item item-29 My time: 0.00285792350769
My item item-29 My time: 1.99865794182
My item item-27 My time: 6.05912590027
My item item-26 My time: 8.05791592598
My item item-23 My time: 14.0749168396
My item item-21 My time: 18.078263998
My item item-20 My time: 20.0878069401
My item item-19 My time: 22.0930709839
My item item-18 My time: 24.0957789421
My item item-14 My time: 36.1192228794
My item item-13 My time: 38.1225728989
My item item-11 My time: 42.1282589436
My item item-8 My time: 48.1387839317
My item item-5 My time: 54.1379349232
My item item-2 My time: 64.5046479702
My item item-1 My time: 66.508150816
My item item-0 My time: 68.5079059601
```

Process 2

```
# this script was started slightly after process 1
$ python example_rtq.py -r scdev -w 30 -n 15 -m
Adding 30 items for testing
Kill when satisfied ^C
My item item-28 My time: 2.95087885857
My item item-25 My time: 9.01049685478
My item item-24 My time: 11.023993969
My item item-22 My time: 15.0343868732
My item item-17 My time: 28.9568138123
My item item-16 My time: 31.0645618439
My item item-15 My time: 33.4570579529
My item item-12 My time: 39.0780348778
My item item-10 My time: 43.0874598026
My item item-9 My time: 45.0917098522
My item item-7 My time: 49.0903818607
My item item-6 My time: 51.0908298492
My item item-4 My time: 59.0306549072
My item item-3 My time: 61.0654230118
```

Notice there is a slight drift due to the queue being moderated (most noticeable in process 1), meaning that the throttle *only allows* the queue to be popped after the moderation time has passed. In our case, 30 seconds divided by 15 hits means that the queue should be popped only after 2 seconds has passed.

If we did not pass the `-m` for moderated flag, your process output may look like the following.

Process 1

```
$ python example_rtq.py -r scdev -w 10 -n 10
Adding 20 items for testing
Kill when satisfied ^C
My item item-19 My time: 0.00159978866577
My item item-18 My time: 0.0029239654541
My item item-17 My time: 0.00445079803467
My item item-16 My time: 0.00595998764038
My item item-15 My time: 0.00703096389771
My item item-14 My time: 0.00823283195496
My item item-13 My time: 0.00951099395752
My item item-12 My time: 0.0107297897339
My item item-11 My time: 0.0118489265442
My item item-10 My time: 0.0128898620605
My item item-13 My time: 10.0101749897
My item item-11 My time: 10.0123429298
My item item-10 My time: 10.0135369301
My item item-9 My time: 20.0031509399
My item item-8 My time: 20.0043399334
My item item-6 My time: 20.0072448254
My item item-5 My time: 20.0084438324
My item item-4 My time: 20.0097179413
```

Process 2

```
$ python example_rtq.py -r scdev -w 10 -n 10
Adding 20 items for testing
Kill when satisfied ^C
My item item-19 My time: 9.12855100632
My item item-18 My time: 9.12996697426
My item item-17 My time: 9.13133692741
My item item-16 My time: 9.13272404671
```

```
My item item-15 My time: 9.13406801224
My item item-14 My time: 9.13519310951
My item item-12 My time: 9.13753604889
My item item-7 My time: 19.1323649883
My item item-3 My time: 19.1368720531
My item item-2 My time: 19.1381940842
My item item-1 My time: 19.1394021511
My item item-0 My time: 19.1405911446
```

Notice that when unmoderated, Process 1 pops all available items in about one hundredth of a second. By the time we switched terminals, Process 2 doesn't have any items to pop and re-adds the 20 items to the queue. In the next 10 second increments, you can see each process receiving items when it is able to successfully pop from the same Redis Queue.

Feel free to mess with the arguments to `example_rtq.py`, and figure out what kind of pop throttling works best for your use case.

Settings Wrapper

The SettingsWrapper module provides a way to organize and override settings for you application in a consistent manner. Like many other larger frameworks that allow you to override particular settings, this class is designed to allow you to easily separate default settings from various custom configurations.

The SettingsWrapper was also created to clean up large amounts of import statements when loading different variables in from another file using the `import` python declaration. Instead, you point the SettingsWrapper to your settings file, and everything is loaded into a python dictionary for you.

class `SettingsWrapper`

`load`(*local='localsettings.py'*, *default='settings.py'*)
Loads the default settings and local override

Parameters

- **`local`** (*str*) – The local override file to keep custom settings in
- **`default`** (*str*) – The core default settings file

Returns The settings dictionary

`load_from_string(settings_string='', module_name='customsettings'):`
Loads your default program settings from an escaped string. Does not provide override capabilities from other strings or the `load()` method.

Parameters

- **`settings_string`** (*str*) – The string containing the settings
- **`module_name`** (*str*) – The module name to use when loading the settings

Returns The loaded settings dictionary

`settings():`

Returns The loaded settings dictionary

Note: When using a local settings file, the SettingsWrapper will override lists, dictionary keys, and variables, but does not recursively override settings buried deeper than 1 layer into dictionary objects.

Usage

Lets assume you have a `settings.py` file with the following setup:

```
NAME='Bill'
FAMILY = {
    'Friends': ['Joe', 'Mark'],
    'Sister': 'Kathy'
}
```

You can load your settings and access them like the following

```
>>> from scutils.settings_wrapper import SettingsWrapper
>>> settings = SettingsWrapper()
>>> my_settings = settings.load(default='settings.py')
No override settings found
>>> my_settings['NAME']
'Bill'
```

Example

Let's expand our use case into a working script that will accept both default settings and override settings. Use the following code as an example and save it as `example_sw.py`, or use the one located at `utils/examples/example_sw.py`

```
import argparse
from scutils.settings_wrapper import SettingsWrapper

# set up arg parser
parser = argparse.ArgumentParser(
    description='Example SettingsWrapper parser.\n')
parser.add_argument('-s', '--settings', action='store', required=False,
                    help="The default settings file",
                    default="settings.py")
parser.add_argument('-o', '--override-settings', action='store', required=False,
                    help="The override settings file",
                    default="localsettings.py")
parser.add_argument('-v', '--variable', action='store', required=False,
                    help="The variable to print out",
                    default=None)
args = vars(parser.parse_args())

# load up settings
wrapper = SettingsWrapper()
my_settings = wrapper.load(default=args['settings'],
                           local=args['override_settings'])

if args['variable'] is not None:
    if args['variable'] in my_settings:
        print args['variable'], '=', my_settings[args['variable']]
```

```
    else:
        print args['variable'], "not in loaded settings"
else:
    print "Full settings:", my_settings
```

Now create a `settings.py` file containing the same settings described above. This will be our **default** settings.

```
NAME='Bill'
FAMILY = {
    'Friends': ['Joe', 'Mark'],
    'Sister': 'Kathy'
}
```

Loading these settings is easy.

```
$ python example_sw.py
No override settings found
Full settings: {'NAME': 'Bill', 'FAMILY': {'Sister': 'Kathy', 'Friends': ['Joe', 'Mark
↪']}}
```

Now we want to alter our application settings for "Joe" and his family. He has the same friends as Bill, but has a different Sister and a house. This is `joe_settings.py` and will overlay on top of our default `settings.py`. This override settings file only needs to contain settings we wish to add or alter from the default settings file, not the whole thing.

```
NAME='Joe'
FAMILY={
  'Sister':'Kim'
}
HOUSE=True
```

Using the override, we can see how things change.

```
$ python example_sw.py -o joe_settings.py
Full settings: {'HOUSE': True, 'NAME': 'Joe', 'FAMILY': {'Sister': 'Kim', 'Friends': [
↪'Joe', 'Mark']}}
```

Notice how we were able to override a specific key in our `FAMILY` dictionary, without touching the other keys. If we wanted to add a final application settings file, we could add Bill's twin brother who is identical to him in every way except his name, `ben_settings.py`.

```
NAME='Ben'
```

```
$ python example_sw.py -o ben_settings.py
Full settings: {'NAME': 'Ben', 'FAMILY': {'Sister': 'Kathy', 'Friends': ['Joe', 'Mark
↪']}}
```

If you would like to further play with this script, you can also use the `-v` flag to print out only a specific variable from your settings dictionary.

```
$ python example_sw.py -o ben_settings.py -v NAME
NAME = Ben
```

Hopefully by now you can see how nice it is to keep custom application settings distinct from the core default settings you may have. With just a couple of lines of code you now have a working settings manager for your application's different use cases.

Stats Collector

The Stats Collector Utility consists of a series of Redis based counting mechanisms, that allow a program to do distributed counting for particular time periods.

There are many useful types of keys within Redis, and this counting Stats Collector allow you to use the following styles of keys:

- Integer values
- Unique values
- HyperLogLog values
- Bitmap values

You can also specify the style of time window you wish to do your collection in. The Stats Collect currently supports the following use cases:

- Sliding window integer counter
- Step based counter for all other types

The sliding window counter allows you to determine *"How many hits have occurred in the last X seconds?"*. This is useful if you wish to know how many hits your API has had in the last hour, or how many times you have handled a particular exception in the past day.

The step based counter allows you to to collect counts based on a rounded time range chunks. This allows you to collect counts of things in meaningful time ranges, like from `9-10 am`, `10-11 am`, `11-12 pm`, etc. The counter incrementally steps through the day, mapping your counter to the aggregated key for your desired time range. If you wanted to collect in 15 minute chunks, the counter steps through any particular hour from `:00-:15`, `:15-:30`, `:30-:45`, and `:45-:00`. This applies to all time ranges available. When using the step style counters, you can also specify the number of previous steps to keep.

Note: The step based counter does not map to the same key once all possible steps have been accounted for. `9:00 - 9:15 am` is **not** the same thing as `10:00 - 10:15am`. or `9-10 am` on Monday is **not** the same thing as `9-10am` on Tuesday (or next Monday). All steps have a unique key associated with them.

You should use the following static class methods to generate your counter objects.

class `StatsCollector`

These easy to use variables are provided for convenience for setting up your collection windows. Note that some are duplicates for naming convention only.

Variables

- **`SECONDS_1_MINUTE`** – The number of seconds in 1 minute
- **`SECONDS_15_MINUTE`** – The number of seconds in 15 minutes
- **`SECONDS_30_MINUTE`** – The number of seconds in 30 minutes
- **`SECONDS_1_HOUR`** – The number of seconds in 1 hour
- **`SECONDS_2_HOUR`** – The number of seconds in 2 hours
- **`SECONDS_4_HOUR`** – The number of seconds in 4 hours
- **`SECONDS_6_HOUR`** – The number of seconds in 6 hours
- **`SECONDS_12_HOUR`** – The number of seconds in 12 hours
- **`SECONDS_24_HOUR`** – The number of seconds in 24 hours

- **SECONDS_48_HOUR** – The number of seconds in 48 hours
- **SECONDS_1_DAY** – The number of seconds in 1 day
- **SECONDS_2_DAY** – The number of seconds in 2 days
- **SECONDS_3_DAY** – The number of seconds in 3 day
- **SECONDS_7_DAY** – The number of seconds in 7 days
- **SECONDS_1_WEEK** – The number of seconds in 1 week
- **SECONDS_30_DAY** – The number of seconds in 30 days

get_time_window(*redis_conn=None*, *host='localhost'*, *port=6379*, *key='time_window_counter'*, *cycle_time=5*, *start_time=None*, *window=SECONDS_1_HOUR*, *roll=True*, *keep_max=12*)

Generates a new TimeWindow Counter. Useful for collecting number of hits generated between certain times

Parameters

- **redis_conn** – A premade redis connection (overrides host and port)
- **host** (*str*) – the redis host
- **port** (*int*) – the redis port
- **key** (*str*) – the key for your stats collection
- **cycle_time** (*int*) – how often to check for expiring counts
- **start_time** (*int*) – the time to start valid collection
- **window** (*int*) – how long to collect data for in seconds (if rolling)
- **roll** (*bool*) – Roll the window after it expires, to continue collecting on a new date based key.
- **keep_max** (*bool*) – If rolling the static window, the max number of prior windows to keep

Returns A *TimeWindow* counter object.

get_rolling_time_window(*redis_conn=None*, *host='localhost'*, *port=6379*, *key='rolling_time_window_counter'*, *cycle_time=5*, *window=SECONDS_1_HOUR*)

Generates a new RollingTimeWindow. Useful for collect data about the number of hits in the past X seconds

Parameters

- **redis_conn** – A premade redis connection (overrides host and port)
- **host** (*str*) – the redis host
- **port** (*int*) – the redis port
- **key** (*str*) – the key for your stats collection
- **cycle_time** (*int*) – how often to check for expiring counts
- **window** (*int*) – the number of seconds behind now() to keep data for

Returns A *RollingTimeWindow* counter object.

get_counter(*redis_conn=None*, *host='localhost'*, *port=6379*, *key='counter'*, *cycle_time=5*, *start_time=None*, *window=SECONDS_1_HOUR*, *roll=True*, *keep_max=12*, *start_at=0*)

Generate a new Counter. Useful for generic distributed counters

Parameters

- **redis_conn** – A premade redis connection (overrides host and port)
- **host** (*str*) – the redis host
- **port** (*int*) – the redis port
- **key** (*str*) – the key for your stats collection
- **cycle_time** (*int*) – how often to check for expiring counts
- **start_time** (*int*) – the time to start valid collection
- **window** (*int*) – how long to collect data for in seconds (if rolling)
- **roll** (*bool*) – Roll the window after it expires, to continue collecting on a new date based key.
- **keep_max** (*int*) – If rolling the static window, the max number of prior windows to keep
- **start_at** (*int*) – The integer to start counting at

Returns A *Counter* object.

get_unique_counter(*redis_conn=None*, *host='localhost'*, *port=6379*, *key='unique_counter'*, *cycle_time=5*, *start_time=None*, *window=SECONDS_1_HOUR*, *roll=True*, *keep_max=12*)

Generate a new UniqueCounter. Useful for exactly counting unique objects

Parameters

- **redis_conn** – A premade redis connection (overrides host and port)
- **host** (*str*) – the redis host
- **port** (*int*) – the redis port
- **key** (*str*) – the key for your stats collection
- **cycle_time** (*int*) – how often to check for expiring counts
- **start_time** (*int*) – the time to start valid collection
- **window** (*int*) – how long to collect data for in seconds (if rolling)
- **roll** (*bool*) – Roll the window after it expires, to continue collecting on a new date based key.
- **keep_max** (*int*) – If rolling the static window, the max number of prior windows to keep

Returns A *UniqueCounter* object.

get_hll_counter(*redis_conn=None*, *host='localhost'*, *port=6379*, *key='hyperloglog_counter'*, *cycle_time=5*, *start_time=None*, *window=SECONDS_1_HOUR*, *roll=True*, *keep_max=12*)

Generate a new HyperLogLogCounter. Useful for approximating extremely large counts of unique items

Parameters

- **redis_conn** – A premade redis connection (overrides host and port)

- **host** (*str*) – the redis host
- **port** (*int*) – the redis port
- **key** (*str*) – the key for your stats collection
- **cycle_time** (*int*) – how often to check for expiring counts
- **start_time** (*int*) – the time to start valid collection
- **window** (*int*) – how long to collect data for in seconds (if rolling)
- **roll** (*bool*) – Roll the window after it expires, to continue collecting on a new date based key.
- **keep_max** (*int*) – If rolling the static window, the max number of prior windows to keep

Returns A *HyperLogLogCounter* object.

get_bitmap_counter(*redis_conn=None*, *host='localhost'*, *port=6379*, *key='bitmap_counter'*, *cycle_time=5*, *start_time=None*, *window=SECONDS_1_HOUR*, *roll=True*, *keep_max=12*)

Generate a new BitMapCounter. Useful for creating different bitsets about users/items that have unique indices.

Parameters

- **redis_conn** – A premade redis connection (overrides host and port)
- **host** (*str*) – the redis host
- **port** (*int*) – the redis port
- **key** (*str*) – the key for your stats collection
- **cycle_time** (*int*) – how often to check for expiring counts
- **start_time** (*int*) – the time to start valid collection
- **window** (*int*) – how long to collect data for in seconds (if rolling)
- **roll** (*bool*) – Roll the window after it expires, to continue collecting on a new date based key.
- **keep_max** (*int*) – If rolling the static window, the max number of prior windows to keep

Returns A *BitmapCounter* object.

Each of the above methods generates a counter object that works in slightly different ways.

class TimeWindow

increment()

Increments the counter by 1.

value()

Returns The value of the counter

get_key()

Returns The string of the key being used

delete_key()

Deletes the key being used from Redis

class `RollingTimeWindow`

`increment`()
Increments the counter by 1.

`value`()

Returns The value of the counter

`get_key`()

Returns The string of the key being used

`delete_key`()
Deletes the key being used from Redis

class `Counter`

`increment`()
Increments the counter by 1.

`value`()

Returns The value of the counter

`get_key`()

Returns The string of the key being used

`delete_key`()
Deletes the key being used from Redis

class `UniqueCounter`

`increment`(*item*)
Tries to increment the counter by 1, if the item is unique

Parameters `item` – the potentially unique item

`value`()

Returns The value of the counter

`get_key`()

Returns The string of the key being used

`delete_key`()
Deletes the key being used from Redis

class `HyperLogLogCounter`

`increment`(*item*)
Tries to increment the counter by 1, if the item is unique

Parameters `item` – the potentially unique item

`value`()

Returns The value of the counter

`get_key`()

Returns The string of the key being used

delete_key()
Deletes the key being used from Redis

class BitmapCounter

increment(*index*)
Sets the bit at the particular index to 1

Parameters **item** – the potentially unique item

value()

Returns The number of bits set to 1 in the key

get_key()

Returns The string of the key being used

delete_key()
Deletes the key being used from Redis

Usage

To use any counter, you should import the StatsCollector and use one of the static methods to generate your counting object. From there you can call `increment()` to increment the counter and `value()` to get the current count of the Redis key being used.

```
>>> from scutils.stats_collector import StatsCollector
>>> counter = StatsCollector.get_counter(host='scdev')
>>> counter.increment()
>>> counter.increment()
>>> counter.increment()
>>> counter.value()
3
>>> counter.get_key()
'counter:2016-01-31_19:00:00'
```

The key generated by the counter is based off of the UTC time of the machine it is running on. Note here since the default `window` time range is `SECONDS_1_HOUR`, the counter rounded the key down to the appropriate step.

Warning: When doing multi-threaded or multi-process counting on the **same key**, all counters operating on that key should be created with the counter style and the same parameters to avoid unintended behavior.

Example

In this example we are going count the number of times a user presses the Space bar while our program continuously runs.

Note: You will need the `py-getch` module from pip to run this example. `pip install py-getch`

```
import argparse
from getch import getch
from time import time
```

```
from scutils.stats_collector import StatsCollector

# set up arg parser
parser = argparse.ArgumentParser(
    description='Example key press stats collector.\n')
parser.add_argument('-rw', '--rolling-window', action='store_true',
                    required=False, help="Use a RollingTimeWindow counter",
                    default=False)
parser.add_argument('-r', '--redis-host', action='store', required=True,
                    help="The Redis host ip")
parser.add_argument('-p', '--redis-port', action='store', default='6379',
                    help="The Redis port")

args = vars(parser.parse_args())

the_window = StatsCollector.SECONDS_1_MINUTE

if args['rolling_window']:
    counter = StatsCollector.get_rolling_time_window(host=args['redis_host'],
                                                     port=args['redis_port'],
                                                     window=the_window,
                                                     cycle_time=1)
else:
    counter = StatsCollector.get_time_window(host=args['redis_host'],
                                             port=args['redis_port'],
                                             window=the_window,
                                             keep_max=3)

print "Kill this program by pressing `ENTER` when done"

the_time = int(time())
floor_time = the_time % the_window
final_time = the_time - floor_time

pressed_enter = False
while not pressed_enter:
    print "The current counter value is " + str(counter.value())
    key = getch()

    if key == '\r':
        pressed_enter = True
    elif key == ' ':
        counter.increment()

    if not args['rolling_window']:
        new_time = int(time())
        floor_time = new_time % the_window
        new_final_time = new_time - floor_time

        if new_final_time != final_time:
            print "The counter window will roll soon"
            final_time = new_final_time

print "The final counter value is " + str(counter.value())
counter.delete_key()
```

This code either creates a *TimeWindow* counter, or a *RollingTimeWindow* counter to collect the number of space bar presses that occurs while the program is running (press `Enter` to exit). With these two different settings,

you can view the count for a specific minute or the count from the last 60 seconds.

Save the above code snippet, or use the example at `utils/examples/example_sc.py`. When running this example you will get similar results to the following.

```
$ python example_sc.py -r scdev
Kill this program by pressing `ENTER` when done
The current counter value is 0
The current counter value is 1
The current counter value is 2
The current counter value is 3
The current counter value is 4
The current counter value is 5
The current counter value is 6
The current counter value is 7
The final counter value is 7
```

It is fairly straightforward to increment the counter and to get the current value, and with only a bit of code tweaking you could use the other counters that the StatsCollector provides.

Zookeeper Watcher

The Zookeeper Watcher utility class is a circuit breaker design pattern around Apache Zookeeper that allows you to watch a one or two files in order to be notified when when they are updated. The circuit breaker allows your application the ability to function normally while your connection is interrupted to Zookeeper, and reestablish the connection and watches automatically.

Behind the scenes, the Zookeeper Watcher utility uses kazoo, and extends Kazoo's functionality by allowing automatic reconnection, automatic file watch reestablishment, and management.

In practice, the Zookeeper Watcher is used to watch configuration files stored within Zookeeper, so that your application may update its configuration automatically. In two-file mode, the utility assumes the first file it is watching actually has a full path to another Zookeeper file where the actual configuration is stored, enabling you to dynamically switch to distinct configurations files stored in Zookeeper as well as see updates to those files.

Within this configuration file watcher, the Zookeeper Watcher also has the ability to function in both **event driven** and **polling mode** (or a combination thereof). Depending on your application setup, it may be more beneficial to use one or the other types of modes.

class **ZookeeperWatcher**(*hosts*, *filepath*, *valid_handler=None*, *config_handler=None*, *error_handler=None*, *pointer=False*, *ensure=False*, *valid_init=True*)

Zookeeper file watcher, used to tell a program their Zookeeper file has changed. Can be used to watch a single file, or both a file and path of its contents. Manages all connections, drops, reconnections for you.

Parameters

- **hosts** (*str*) – The zookeeper hosts to use
- **filepath** (*str*) – The full path to the file to watch
- **valid_handler** (*func*) – The method to call for a 'is valid' state change
- **config_handler** (*func*) – The method to call when a content change occurs
- **error_handler** (*func*) – The method to call when an error occurs
- **pointer** (*bool*) – Set to true if the file contents are actually a path to another zookeeper file, where the real config resides

- **ensure** (*bool*) – Set to true for the ZookeeperWatcher to create the watched file if none exists
- **valid_init** (*bool*) – Ensure the client can connect to Zookeeper first try

is_valid()

Returns A boolean indicating if the current series of watched files is valid. Use this for basic polling usage.

get_file_contents(*pointer=False*)

Gets any file contents you care about. Defaults to the main file

Parameters **pointer** (*bool*) – Get the contents of the file pointer, not the pointed at file

Returns A string of the contents

ping()

A ping to determine if the Zookeeper connection is working

Returns True if the connection is alive, otherwise False

close(*kill_restart=True*)

Gracefully shuts down the Zookeeper Watcher

Parameters **kill_restart** – Set to False to allow the daemon process to respawn. This parameter is not recommended and may be deprecated in future versions.

Usage

The Zookeeper Watcher spawns a daemon process that runs in the background along side your application, so you only need to initialize and keep the variable around in your program.

Assuming you have pushed some kind of content into Zookeeper already, for example `stuff here` or `Some kind of config string here`.

In polling mode:

```
>>> from scutils.zookeeper_watcher import ZookeeperWatcher
>>> from time import sleep
>>> file = "/tmp/myconfig"
>>> zoo_watcher = ZookeeperWatcher("scdev", file)
>>> hosts = "scdev"
>>> zoo_watcher = ZookeeperWatcher(hosts, file)
>>> while True:
...       print "Valid File?", zoo_watcher.is_valid()
...       print "Contents:", zoo_watcher.get_file_contents()
...       sleep(1)
...
Valid File? True
Contents: stuff here
Valid File? True
Contents: stuff here
```

For event driven polling, you need to define any of the three event handlers within your application. The below example defines only the the `config_handler`, which allows you to monitor changes to your desired file.

```
>>> from scutils.zookeeper_watcher import ZookeeperWatcher
>>> from time import sleep
>>> file = "/tmp/myconfig"
>>> def change_file(conf_string):
```

```
...     print "Your file contents:", conf_string
...
>>> zoo_watcher = ZookeeperWatcher("scdev", file, config_handler=change_file)
Your file contents: Some kind of config string here
```

You can manually alter your file contents if you have access to the Zookeeper's command line interface, but otherwise can use the `file_pusher.py` file located within the `crawling/config` directory of this project. For more information about the `file_pusher.py` script and its uses, please see *here*.

Example

In this example, we will create a fully functional file watcher that allows us to flip between various states of usage by passing different command line arguments.

```
from scutils.zookeeper_watcher import ZookeeperWatcher
from time import sleep
import argparse

parser = argparse.ArgumentParser(
            description="Zookeeper file watcher")
parser.add_argument('-z', '--zoo-keeper', action='store', required=True,
                    help="The Zookeeper connection <host>:<port>")
parser.add_argument('-f', '--file', action='store', required=True,
                    help="The full path to the file to watch in Zookeeper")
parser.add_argument('-p', '--pointer', action='store_const', const=True,
                    help="The file contents point to another file")
parser.add_argument('-s', '--sleep', nargs='?', const=1, default=1,
                    type=int, help="The time to sleep between poll checks")
parser.add_argument('-v', '--valid-init', action='store_false',
                    help="Do not ensure zookeeper is up upon initial setup",
                    default=True)
group = parser.add_mutually_exclusive_group(required=True)
group.add_argument('--poll', action='store_true', help="Polling example")
group.add_argument('--event', action='store_true',
                   help="Event driven example")

args = vars(parser.parse_args())

hosts = args['zoo_keeper']
file = args['file']
pointer = args['pointer']
sleep_time = args['sleep']
poll = args['poll']
event = args['event']
valid = args['valid_init']

def valid_file(state):
    print "The valid state is now", state

def change_file(conf_string):
    print "Your file contents:", conf_string

def error_file(message):
    print "An error was thrown:", message

# You can use any or all of these, polling + handlers, some handlers, etc
```

```
if pointer:
    if poll:
        zoo_watcher = ZookeeperWatcher(hosts, file, ensure=True,pointer=True)
    elif event:
        zoo_watcher = ZookeeperWatcher(hosts, file,
                                       valid_handler=valid_file,
                                       config_handler=change_file,
                                       error_handler=error_file,
                                       pointer=True, ensure=True, valid_init=valid)
else:
    if poll:
        zoo_watcher = ZookeeperWatcher(hosts, file, ensure=True)
    elif event:
        zoo_watcher = ZookeeperWatcher(hosts, file,
                                       valid_handler=valid_file,
                                       config_handler=change_file,
                                       error_handler=error_file,
                                       valid_init=valid, ensure=True)

print "Use a keyboard interrupt to shut down the process."
try:
    while True:
        if poll:
            print "Valid File?", zoo_watcher.is_valid()
            print "Contents:", zoo_watcher.get_file_contents()
        sleep(sleep_time)
except:
    pass
zoo_watcher.close()
```

This file allows us to test out the different capabilities of the Zookeeper Watcher. Now, save this file as `example_zw.py` or use the one located at `utils/examples/example_zw.py` and then in a new terminal use the `file_pusher.py` to push a sample settings file like shown below.

settings.txt

```
My configuration string here. Typically YAML or JSON
```

Push the configuration into Zookeeper

```
$ python file_pusher.py -f settings.txt -i myconfig -p /tmp/ -z scdev
creaing conf node
```

Run the file watcher.

```
$ python example_zw.py -z scdev -f /tmp/myconfig --poll
Valid File? True
Contents: My configuration string here. Typically YAML or JSON
```

You can see it already grabbed the contents of your file. Lets now change the file while the process is still running. Update your **settings.txt** file.

```
NEW My configuration string here. Typically YAML or JSON
```

Now push it up with the same above command. In your watcher window, you will see the text output flip over to the following.

```
Valid File? True
Contents: My configuration string here. Typically YAML or JSON

Valid File? True
Contents: NEW My configuration string here. Typically YAML or JSON
```

Now, lets try it in event mode. Stop your initial process and restart it with these new parameters

```
$ python example_zw.py -z scdev -f /tmp/myconfig --event
Your file contents: NEW My configuration string here. Typically YAML or JSON

The valid state is now True
Use a keyboard interrupt to shut down the process.
```

Notice both the valid state handler and the file string were triggered once and only once. Lets now update that configuration file one more time.

settings.txt

```
EVENT TRIGGER My configuration string here. Typically YAML or JSON
```

When you push that up via the same file pusher command prior, an event will fire in your watcher window only once.

```
Your file contents: EVENT TRIGGER My configuration string here. Typically YAML or JSON
```

Lastly, lets change our example to be *pointer* based, meaning that the initial file we push up should have a single line inside it that points to another location in Zookeeper. Stop your initial Zookeeper Watcher process and update the settings file.

settings.txt

```
/tmp/myconfig
```

We should put this somewhere else within zookeeper, as it stores a pointer to the file we actually care about. Push this into a new pointers folder

```
$ python file_pusher.py -f settings.txt -i pointer1 -p /tmp_pointers/ -z scdev
```

Now we have a pointer and the old config file. Let us make another config file for fun.

settings2.txt

```
this is another config
```

Push it up.

```
$ python file_pusher.py -f settings2.txt -i myconfig2 -p /tmp/ -z scdev
creaing conf node
```

So now we have two configuration files located at `/tmp/`, and one pointer file located in `/tmp_pointers/`. Now lets run our watcher in pointer mode, with the file path specifying the pointer path instead.

```
$ python example_zw.py -z scdev -f /tmp_pointers/pointer1 --event -p
Your file contents: EVENT TRIGGER My configuration string here. Typically YAML or JSON

The valid state is now True
```

Neat! It shows the actual configuration we care about, instead of the path pointer.

Warning: If you receive `The valid state is now False, An error was thrown: Invalid pointer path` error, you most likely have a newline in your pointer file! This is very easy to eliminate with code, but can be hard to detect in typical text editors. **You should have one line only in the pointer file**

Now that we have the configuration we care about, with the watcher process still running we will change the configuration *pointer*, not the configuration file.

settings.txt

```
/tmp/myconfig2
```

Lets push it up, and watch our configuration change.

```
$ python file_pusher.py -f settings.txt -i pointer1 -p /tmp_pointers/ -z scdev
```

The change

```
Your file contents: this is another config
```

We also have the ability to update that configuration too.

settings2.txt

```
this is another config I ADDED NEW STUFF
```

Push it up

```
$ python file_pusher.py -f settings2.txt -i mfig2 -p /tmp/ -z scdev
```

The change in the watcher process

```
Your file contents: this is another config I ADDED NEW STUFF
```

The example application received the updated changes of the file it is pointed at!

What we have here is the ability to have a 'bank' of configuration files within a centrally located place (Zookeeper). This allows us to only care where our 'pointer' sits, and be directed to the appropriate real configuration file. We will get the changes when the pointed at file is updated, or we need to point to a different file (updates to the pointer file).

Hopefully this example has shown the power behind the Zookeeper Watcher utility class. It allows your application to receive and act on updates to files stored within Zookeeper in an easy to use manner, without the overhead and complexity of setting up individual file notifications and management of them.

Argparse Helper Simple module to assist in argument parsing with subparsers.

Log Factory Module for logging multithreaded or concurrent processes to files, stdout, and/or json.

Method Timer A method decorator to timeout function calls.

Redis Queue A module for creating easy redis based FIFO, Stack, and Priority Queues.

Redis Throttled Queue A wrapper around the *Redis Queue* module to enable distributed throttled pops from the queue.

Settings Wrapper Easy to use module to load both default and local settings for your python application and provides a dictionary object in return.

Stats Collector Module for statistics based collection in Redis, including counters, rolling time windows, and hyperloglog counters.

Zookeeper Watcher Module for watching a zookeeper file and handles zookeeper session connection troubles and re-establishment of watches.

CHAPTER 7

Advanced Topics

These documents cover more advanced topics within Scrapy Cluster in no particular order.

Upgrade Scrapy Cluster

This page will walk you through how to upgrade your Scrapy Cluster from one version to the next. This assumes you have a fully functioning, stable, and deployed cluster already and would like to run the latest version of Scrapy Cluster.

Upgrades

Pre Upgrade

Before you shut down your current crawling cluster. You should build and test the desired version of Scrapy Cluster you wish to upgrade to. This means looking over documentation, or trying the cluster out in a Virtual Machine. You should be familiar with the tutorials on how to stand a new cluster version up, and be prepared for when the time comes to upgrade your existing cluster.

Preparation is key so you do not lose data or get malformed data into your cluster.

The Upgrade

For all upgrades you should use the `migrate.py` script at the root level of the Scrapy Cluster project.

```
$ python migrate.py -h
usage: migrate.py [-h] -ir INPUT_REDIS_HOST [-ip INPUT_REDIS_PORT]
                  [-id INPUT_REDIS_DB] [-or OUTPUT_REDIS_HOST]
                  [-op OUTPUT_REDIS_PORT] [-od OUTPUT_REDIS_DB] -sv {1.0,1.1}
                  -ev {1.1,1.2} [-v {0,1,2}] [-y]

Scrapy Cluster Migration script. Use to upgrade any part of Scrapy Cluster.
Not recommended for use while your cluster is running.
```

```
optional arguments:
  -h, --help            show this help message and exit
  -ir INPUT_REDIS_HOST, --input-redis-host INPUT_REDIS_HOST
                        The input Redis host ip
  -ip INPUT_REDIS_PORT, --input-redis-port INPUT_REDIS_PORT
                        The input Redis port
  -id INPUT_REDIS_DB, --input-redis-db INPUT_REDIS_DB
                        The input Redis db
  -or OUTPUT_REDIS_HOST, --output-redis-host OUTPUT_REDIS_HOST
                        The output Redis host ip, defaults to input
  -op OUTPUT_REDIS_PORT, --output-redis-port OUTPUT_REDIS_PORT
                        The output Redis port, defaults to input
  -od OUTPUT_REDIS_DB, --output-redis-db OUTPUT_REDIS_DB
                        The output Redis db, defaults to input
  -sv {1.0,1.1}, --start-version {1.0,1.1}
                        The current cluster version
  -ev {1.1,1.2}, --end-version {1.1,1.2}
                        The desired cluster version
  -v {0,1,2}, --verbosity {0,1,2}
                        Increases output text verbosity
  -y, --yes             Answer 'yes' to any prompt
```

The script also provides the ability to migrate between Redis instances, change the verbosity of the logging within the migration script, and can also prompt the user in case of failure or potentially dangerous situations.

This script **does not** upgrade any of the actual code or applications Scrapy Cluster uses! Only behind the scenes keys and datastores used by a deployed cluster. You are still responsible for deploying the new Crawlers, Kafka Monitor, and Redis Monitor.

Warning: It is **highly** recommended you shut down all three components of your Scrapy Cluster when upgrading. This is due to the volatile and distributed nature of Scrapy Cluster, as there can be race conditions that develop when the cluster is undergoing an upgrade while scrapers are still inserting new Requests, or the Kafka Monitor is still adding Requests to the Redis Queues.

If you do not chose to shut down the cluster, you need to ensure it is 'quiet'. Meaning, there is no data going into Kafka or into Redis.

Once your cluster is quiet or halted, run the upgrade script to ensure everything is compatible with the newer version. For example, upgrading from Scrapy Cluster 1.0 to Scrapy Cluster 1.1:

```
$ python migrate.py -ir scdev -sv 1.0 -ev 1.1 -y
Upgrading Cluster from 1.0 to 1.1
Cluster upgrade complete in 0.01 seconds.
Upgraded cluster from 1.0 to 1.1
```

In the future, this script will manage all upgrades to any component needed. For the time being, we only need to upgrade items within Redis.

Post Upgrade

After upgrading, you should use the various version quick start documentations for each component, to ensure it is properly configured, restarted, and running. Afterwards in your production level run mode, you should begin to monitor log output to check for any anomalies within your cluster.

Upgrade Notes

This section holds any comments or notes between versions.

1.0 -> 1.1

The primary upgrade here is splitting the individual spider queues with keys like `<spider>:queue` into a more generic queue system `<spider>:<domain>:queue`. This allows us to do controlled domain throttling across the cluster, better explained *here*.

1.1 -> 1.2

Upgrades the cluster for an inefficient use of `pickle` encoding to a more efficient use of `ujson`. The primary keys impacted by this upgrade are the spider domain queues.

Integration with ELK

Scrapy Cluster's *Log Factory* has the ability to change the log output from human readable to JSON, which integrates very nicely with tools like Elasticsearch, Logstash, and Kibana. This page will guide you on how to set up basic monitoring and visualizations of your Scrapy Cluster through the ELK stack.

> **Warning:** This guide does not cover how to set up or maintain any of the technologies used within. You should already be familiar with the ELK stack and have a working Scrapy Cluster before you begin.

Scrapy Cluster Setup

You should alter a couple of settings used within each of the main components of Scrapy Cluster. Change the `localsettings.py` files to be updated with the following.

Kafka Monitor

- `LOG_STDOUT = False` - Forces the Kafka Monitor logs to be written to a file on the machine
- `LOG_JSON = True` - Flips logging output from human readable to JSON.
- `LOG_DIR = '/var/log/scrapy-cluster'` - Logs the file into a particular directory on your machines

Redis Monitor

- `LOG_STDOUT = False` - Forces the Redis Monitor logs to be written to a file on the machine
- `LOG_JSON = True` - Flips logging output from human readable to JSON.
- `LOG_DIR = '/var/log/scrapy-cluster'` - Logs the file into a particular directory on your machines

Crawler

- `SC_LOG_STDOUT = False` - Forces the Crawler logs to be written to a file on the machine
- `SC_LOG_JSON = True` - Flips logging output from human readable to JSON.
- `SC_LOG_DIR = '/var/log/scrapy-cluster'` - Logs the file into a particular directory on your machines

Rest

- `LOG_STDOUT = False` - Forces the Rest service logs to be written to a file on the machine
- `LOG_JSON = True` - Flips logging output from human readable to JSON.
- `LOG_DIR = '/var/log/scrapy-cluster'` - Logs the file into a particular directory on your machines

---

Note: Depending on your machine's `/var/log` folder permissions, you may need to create the `scrapy-cluster` directory manually and ensure your Scrapy Cluster processes have permission to write to the directory.

You should now be able to `tail -f` any of your logs and see the JSON objects flowing by.

Logstash

This setup assumes you have Logstash or FileBeat set up, in order to forward your logs to Elasticsearch. For this, we will assume Logstash is installed on every machine, but the set up should be similar enough to work with File Beat.

We will use Logstash to automatically store and manage Scrapy Cluster's logs via a Mapping Template. This allows us to automatically format our logs within Elasticsearch so Kibana can display them properly. This is the mapping template we will use:

```
{
  "template" : "logs-*",
  "order" : 0,
  "settings" : {
    "index.refresh_interval" : "5s"
  },
  "mappings" : {
    "_default_" : {
      "dynamic_templates" : [ {
        "message_field" : {
          "mapping" : {
            "omit_norms" : true,
            "type" : "keyword"
          },
          "match_mapping_type" : "keyword",
          "match" : "message"
        }
      }, {
        "keyword_fields" : {
          "mapping" : {
            "omit_norms" : true,
            "type" : "keyword"
          },
          "match_mapping_type" : "keyword",
          "match" : "*"
        }
      } ],
      "properties" : {
        "@version" : {
          "index" : "not_analyzed",
          "type" : "keyword"
        }
      },
      "_all" : {
        "enabled" : true
      }
    }
```

```
  },
  "aliases" : { }
}
```

Save this file as `logs-template.json` in your logstash templates directory, here we will assume it is located at `/etc/logstash/templates/`.

Now we need to configure Logstash to use the template, and read from our logs. For this we will use a simple file input and elasticsearch output available via Logstash.

```
input {
  file {
    path => ['/var/log/scrapy-cluster/*.log']
    codec => json
    tags => ['scrapy-cluster']
  }
}

output {
  if 'scrapy-cluster' in [tags]{
    elasticsearch {
      hosts => "<your es hosts here>"
      template => "/etc/logstash/templates/logs-template.json"
      template_name => "logs-*"
      template_overwrite => true
      index => "logs-scrapy-cluster"
      document_type => "%{[logger]}"
    }
  }
}
```

Save this file as `scrapy-cluster-logstash.conf`, and put it into the folder where Logstash reads its configuration files. This logstash template says that we are going to read from any file that matches our pattern `*.log` within the Scrapy Cluster log folder we defined prior. The output of this operation says to ship that log to our Elasticsearch hosts, using the template we created one step above. This will write our logs to the Elasticsearch index `logs-scrapy-cluster`, with the document type defined as the logger name.

What we end up with is one single index where our logs are stored, and each type of log (Kafka Monitor, Redis Monitor, and Crawler) split into a different series of documents.

You will need to restart your Logstash instance to read the new settings, but once running you should end up with any new logs being written both to disk and to your Elasticsearch cluster.

`http://<your es host>:9200/logs-scrapy-cluster/_count?pretty`

```
{
  "count": 19922,
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  }
}
```

Here, we have done a bit of crawling already and have around 20,000 log records in our index.

At this point you should now have your logs indexed in Elasticsearch, and we can use Kibana to visualize them.

Kibana

In your Kibana instance, you now need to configure a new index pattern. If you would like to be exact, use `logs-scrapy-cluster`, or if you plan on using the provided templates in other projects you can use `logs-*`. Configure the time value to be `timestamp`, **NOT** `@timestamp`. The latter is an auto-generated timestamp by logstash, and does not reflect the real time the log was written by the process.

From here, you can play around with the different searching and visualization functions provided by Kibana.

If you would like to use some preconfigured searches and visualizations, go to **Settings** and (at time of writing) click **Saved Objects**, then **Import**. We are going to import a sample set of visualizations and searches from the Scrapy Cluster project under the folder `elk`. Select the `export.json` file to import everything in.

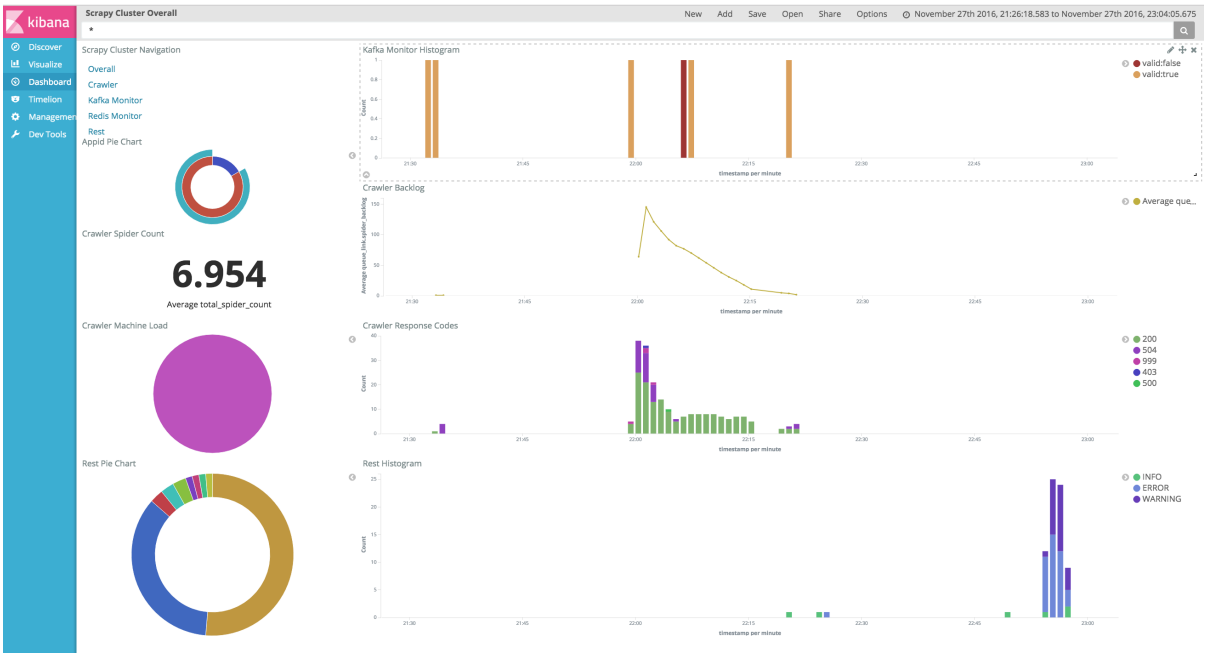
Note: It is important you actually use your cluster before you try to upload the preconfigured visualizations. This ensures the defined mappings within Elasticsearch are present for the widgets. You can check this by looking at the number of fields in your index defined above - if is has over **170** different fields you should be ok to import, otherwise refresh it, use the cluster more, or exercise a different component.

You should now have a number of different Visualizations, Dashboards, and Searches so you can better understand how your cluster is operating at scale.

Note: The graphs below only show a sample series of three or four crawl requests over a span of four hours. A typical cluster will have hundreds or thousands of requests per minute!

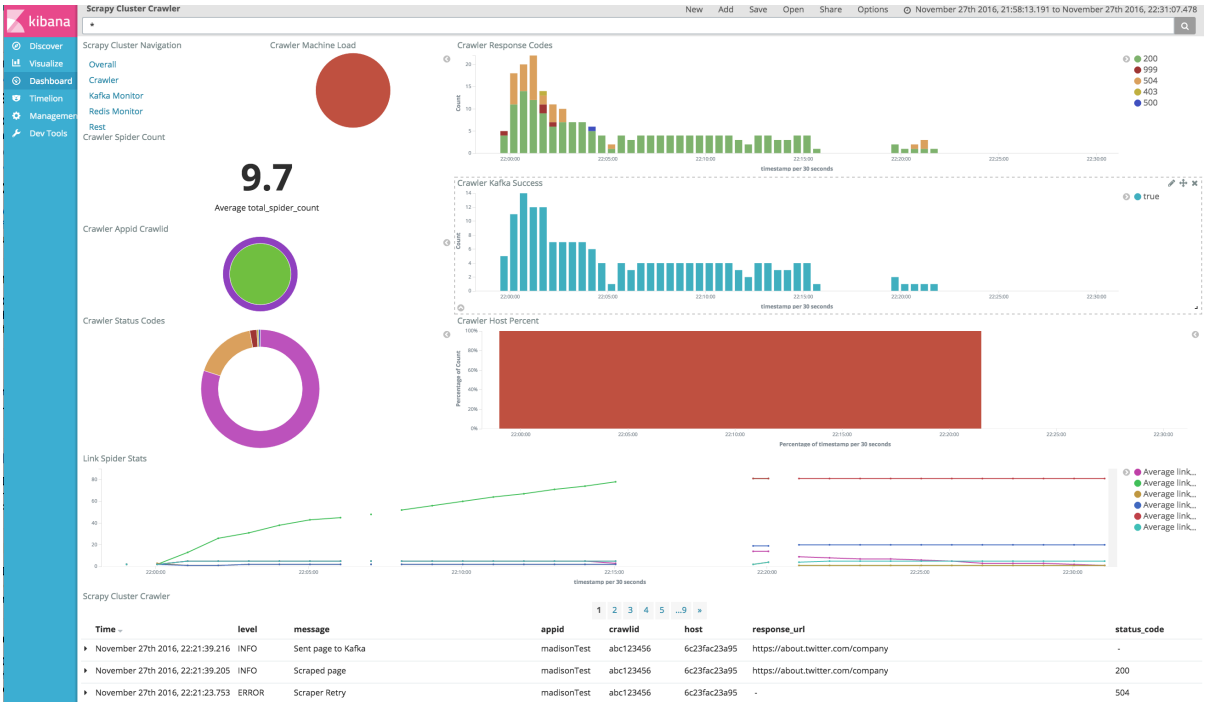
Overall

This is a high level overview dashboard of all three components of your Scrapy Cluster. This is the typical view to go to when you would like to know what is going on across all of your different components.



Crawler

The Crawler dashboard view shows you a much more in depth view of your current Scrapy Crawlers. Here you see breakdowns of response codes, machine load balances, and successful outbound Kafka messages.



Kafka Monitor

This view gives you better insight into the Kafka Monitor and the APIs in which it is testing against. It shows a breakdown of application requests and overall usage of the Kafka Monitor.

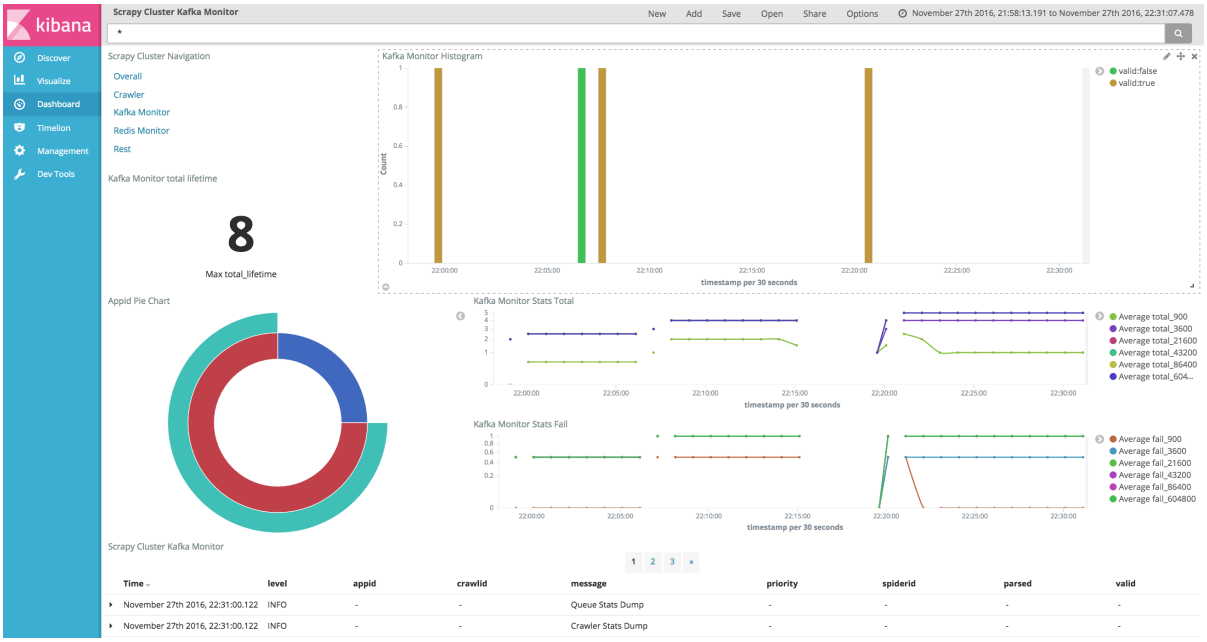
Redis Monitor

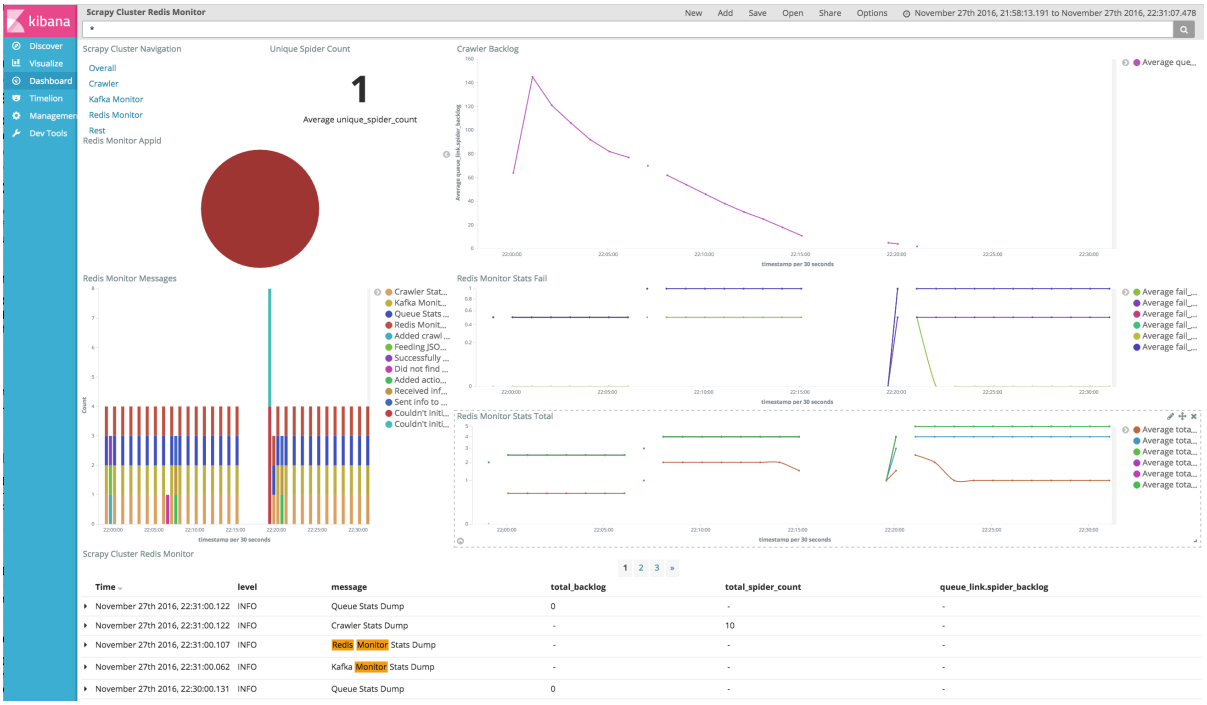
The Redis Monitor breakdown shows you the backlog of your current spiders, and the different requests the Redis Monitor has had to process from your cluster.

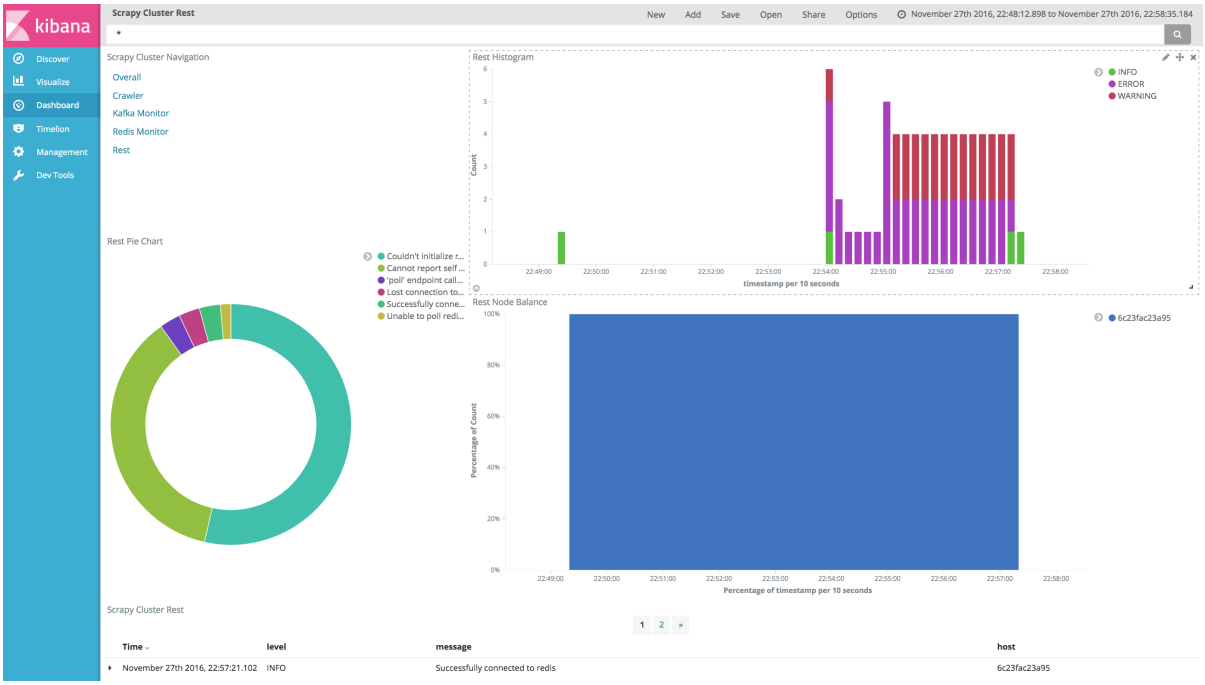
Rest

The Rest breakdown gives you a view of the endpoints and logs being generated by the Rest components in your cluster. It shows a basic breakdown over time and by log type.

Feel free to add to or tinker with the visualizations provided! You should now have a much better understanding about what is going on within your Scrapy Cluster.


kibana
Scrapy Cluster Kafka Monitor
New Add Save Open Share Options
November 27th 2016, 21:58:13.191 to November 27th 2016, 22:31:07.478
Discover
Visualize
Dashboard
Timelion
Management
Dev Tools
Scrapy Cluster Navigation
Overall
Crawler
Kafka Monitor
Redis Monitor
Rest
Kafka Monitor total lifetime
8
Max total_lifetime
Kafka Monitor Histogram
valid:false
valid:true
Count
timestamp per 30 seconds
Appid Pie Chart
Kafka Monitor Stats Total
Average total_900
Average total_3600
Average total_21600
Average total_43200
Average total_86400
Average total_604...
Kafka Monitor Stats Fail
Average fail_900
Average fail_3600
Average fail_21600
Average fail_43200
Average fail_86400
Average fail_604800
Scrapy Cluster Kafka Monitor
Time
level
appid
crawlid
message
priority
spiderid
parsed
valid
November 27th 2016, 22:31:00.122 INFO - - Queue Stats Dump - - - -
November 27th 2016, 22:31:00.122 INFO - - Crawler Stats Dump - - - -


kibana
Scrapy Cluster Redis Monitor
New Add Save Open Share Options
November 27th 2016, 21:58:13.191 to November 27th 2016, 22:31:07.478
Discover
Visualize
Dashboard
Timelion
Management
Dev Tools
Scrapy Cluster Navigation
Overall
Crawler
Kafka Monitor
Redis Monitor
Rest
Unique Spider Count
1
Average unique_spider_count
Redis Monitor Appid
Crawler Backlog
Average que...
timestamp per 30 seconds
Redis Monitor Messages
Crawler Stat...
Kafka Monit...
Queue Stats ...
Redis Monit...
Added crawl ...
Feeding JSO...
Successfully ...
Did not find ...
Added actio...
Received Inf...
Sent info to ...
Couldn't init...
Couldn't init...
Redis Monitor Stats Fail
Average fail_...
Redis Monitor Stats Total
Average tota...
Scrapy Cluster Redis Monitor
Time
level
message
total_backlog
total_spider_count
queue_link.spider_backlog
November 27th 2016, 22:31:00.122 INFO Queue Stats Dump 0 - -
November 27th 2016, 22:31:00.122 INFO Crawler Stats Dump - 10 -
November 27th 2016, 22:31:00.107 INFO Redis Monitor Stats Dump - - -
November 27th 2016, 22:31:00.062 INFO Kafka Monitor Stats Dump - - -
November 27th 2016, 22:30:00.131 INFO Queue Stats Dump 0 - -



Docker

Scrapy Cluster supports Docker by ensuring each individual component is contained within a a different docker image. You can find the docker compose files in the root of the project, and the Dockerfiles themselves and related configuration is located within the `/docker/` folder. This page is not meant as an introduction to Docker, but as a supplement for those comfortable working with Docker already.

A container could either contain:

- A single spider
- A single kafka monitor
- A single redis monitor
- A single rest service

Thanks to the ability to scale each component independently of each other, we utilize Docker Compose to allow for scaling of Scrapy Cluster and management of the containers. You can use this in conjunction with things like Docker Swarm, Apache Mesos, Kubernetes, Amazon EC2 Container Service, or any other container manager of your choice.

You can find the latest images on the Scrapy Cluster Docker hub page here.

Note: Docker support for Scrapy Cluster is fairly new, and large scale deployments of Scrapy Cluster have not been fully tested at time of writing.

Images

Each component for Scrapy Cluster is designated as a tag within the root docker repository. Unlike a lot of projects, we chose to keep the dockerized Scrapy Cluster within the same github repository in order to stay consistent with how the project is used. This means that there will be no `latest` tag for Scrapy Cluster, instead the tags are defined as follows.

Kafka Monitor: `istresearch/scrapy-cluster:kafka-monitor-{release/build}`

Redis Monitor: `istresearch/scrapy-cluster:redis-monitor-{release/build}`

Crawler: `istresearch/scrapy-cluster:crawler-{release/build}`

Rest: `istresearch/scrapy-cluster:rest-{release/build}`

For example `istresearch/scrapy-cluster:redis-monitor-1.2` would be the official stable 1.2 release of the Redis Monitor, but `istresearch/scrapy-cluster:redis-monitor-dev` would be tied to the latest `dev` branch release. Typically numeric releases will be paired with the `master` branch, while `-dev` releases will be paired with the `dev` branch.

Code Layout

Each container contains only code for that particular component, located at `/usr/src/app`. By default a `localsettings.py` file will supply the override to provide the container the configuration to work with the native docker compose files. You are free to use docker volumes to mount a different settings file on top of the `localsettings.py` file in order to configure the component with your particular setup.

However, if you look at the component's local settings file, you may see some slight alterations based on environment variable overrides. The following snippet shows some of the alterations you would find in the project's `docker/*/settings.py` files, or within the container at `/usr/src/app/localsettings.py`:

```
REDIS_HOST = os.getenv('REDIS_HOST', 'redis')
REDIS_DB = int(os.getenv('REDIS_DB', 0))
KAFKA_INCOMING_TOPIC = os.getenv('KAFKA_INCOMING_TOPIC', 'demo.incoming')
LOG_JSON = str2bool(os.getenv('LOG_JSON', False))
```

Commonly altered variables can be overridden with a docker environment variable of the same name. Here, you can override `REDIS_DB` with `1` to specify a different redis database.

If you find a variable you would like to override that does not already have an environment variable applied, you can use the following to make it available.

String: `os.getenv('PARAM_NAME', 'default_value')`

Integer: `int(os.getenv('PARAM_NAME', default_value))`

Boolean: `str2bool(os.getenv('PARAM_NAME', default_value))`

Where `default_value` is your specified default, and `PARAM_NAME` is the name of the configuration you are overriding. Note that you will need to rebuild your containers if you alter code within the project.

The environment overrides can then be applied to your component within the normal `-e` flag from Docker, or in the `docker-compose.yml` files like show below.

```
environment:
  - REDIS_HOST=other-redis
  - REDIS_DB=1
  - LOG_JSON=True
```

It is important to either look at the `localsettings.py` files within the container or at the `settings.py` files as denoted above to find the environment variables you wish to tweak.

Running

It is recommended you use docker compose to orchestrate your cluster with all of the other components, and simply issuing `docker-compose up` will bring your cluster online. If you wish to build the containers from scratch or

insert custom code, please use add the following lines, customized for each component.

```
image: istresearch/scrapy-cluster:kafka-monitor-1.2
build:
  context: .
  dockerfile: docker/kafka-monitor/Dockerfile
```

You will then be able to issue the following:

```
docker-compose -f docker-compose.yml up --build --force-recreate
```

This will rebuild your containers and bring everything online.

Warning: The Kafka container does not like being stopped and started over and over again, and will sometimes fail to start cleanly. You can mitigate this by issuing `docker-compose down` (note this will clean out all data).

Compose ELK Stack

You can use Docker Compose to configure both Scrapy Cluster and an associated ELK stack to view your log data from the 4 core components.

The docker compose file is located in `elk/docker-compose.elk.yml`, and contains all of the necessary ingredients to bring up

- Scrapy Cluster
  - Kafka Monitor
  - Redis Monitor
  - Crawler
  - Rest
- Infrastructure
  - Kafka
  - Zookeeper
  - Redis
- ELK
  - Elasticsearch
  - Logstash
  - Kibana

Bring it up by issuing the following command from within the `elk` folder:

```
$ docker-compose -f docker-compose.elk.yml up -d
```

You can ensure everything started up via:

```
$ docker-compose -f docker-compose.elk.yml ps
        Name                     Command                  State                    Ports
-------------------------------------------------------------------------------------------
↪-------
elk_crawler_1           scrapy                   Up
```

```
                        runspider c ...
elk_elasticsearch_1     /docker-entrypoint.sh   Up                      0.0.0.0:9200->
↪9200/tc
                        elas ...                                        p, 0.0.0.
↪0:9300->9300
                                                                        /tcp
elk_kafka_1             start-kafka.sh          Up                      0.0.0.0:9092->
↪9092/tc
                                                                        p
elk_kafka_monitor_1     python                  Up
                        kafka_monit ...
elk_kibana_1            /docker-entrypoint.sh   Up                      0.0.0.0:5601->
↪5601/tc
                        kibana                                          p
elk_logstash_1          /docker-entrypoint.sh   Up                      0.0.0.0:5000->
↪5000/tc
                        logs ...                                        p
elk_redis_1             docker-entrypoint.sh    Up                      0.0.0.0:32776-
↪>6379/t
                        redis ...                                       cp
elk_redis_monitor_1     python                  Up
                        redis_monit ...
elk_rest_1              python rest_service.py  Up                      0.0.0.0:5343->
↪5343/tcp
elk_zookeeper_1         /bin/sh -c              Up                      0.0.0.0:2181->
↪2181/tc
                        /usr/sbin/sshd  ...                             p, 22/tcp,␣
↪2888/tcp,
                                                                        3888/tcp
```

From here, please continue to the *Kibana* portion of the *ELK* integration guide.

As we continue to to expand into the docker world this page is subject to change. If you have a novel or different way you would like to use Scrapy Cluster in your container based application we would love to hear about it.

Crawling Responsibly

Scrapy Cluster is a very high throughput web crawling architecture that allows you to spread the web crawling load across an arbitrary number of machines. It is up the the end user to scrape pages responsibly, and to not crawl sites that do not want to be crawled. As a start, most sites today have a robots.txt file that will tell you how to crawl the site, how often, and where not to go.

You can very easily max out your internet pipe(s) on your crawling machines by feeding them high amounts of crawl requests. In regular use we have seen a single machine with five crawlers running sustain almost 1000 requests per minute! We were not banned from any site during that test, simply because we obeyed every site's robots.txt file and only crawled at a rate that was safe for any single site.

Abuse of Scrapy Cluster can have the following things occur:

- An investigation from your ISP about the amount of data you are using
- Exceeding the data cap of your ISP
- Semi-permanent and permanent bans of your IP Address from sites you crawl too fast

With great power comes great responsibility.

Production Setup

System Fragility

Scrapy Cluster is built on top of many moving parts, and likely you will want some kind of assurance that you cluster is continually up and running. Instead of manually ensuring the processes are running on each machine, it is highly recommended that you run every component under some kind of process supervision.

We have had very good luck with Supervisord but feel free to put the processes under your process monitor of choice.

As a friendly reminder, the following processes should be monitored:

- Zookeeper
- Kafka
- Redis
- Crawler(s)
- Kafka Monitor(s)
- Redis Monitor(s)
- Rest service(s)

Spider Complexity

Spider complexity plays a bigger role than you might think in how fast your cluster can crawl. The two single most important things to worry about are:

How fast your Spider can process a Response

Any single response from Scrapy is going to hog that spider process until all of the items are yielded appropriately. How fast your literal `spider.py` script can execute cleanly is important to getting high throughput from your Spiders. Sure, you can up the number of concurrent requests but in the end an inefficient Spider is going to slow you down.

How fast your Item Pipeline can process an Item

Scrapy item pipelines are not distributed, and you should not be doing processing with your items that requires that kind of scalability. You can up the maximum number of items in your item pipeline, but in the end if your pipeline is trying to do too much your whole crawling architecture will begin to slow down. That is why Scrapy Cluster is built around Kafka, so that you may do really large and scaled processing of your items in another architecture, like Storm.

Hardware

Many that are new to Scrapy Cluster do not quite understand how to get the most out of it. Improperly configured core data components like Zookeeper, Kafka, and Redis, or even where you put your crawling machines can have an impact on how fast you can crawl. **The most important thing you can do is to have a massive internet pipe available to your crawler machines**, with a good enough network card(s) to keep up.

Nothing here can substitute for a poor network connection, so no matter if you are crawling behind a proxy, through some service, or to the open internet, your rate of collection is depending on how fast you can get the data.

In production, we typically run everything on their own boxes, or a little overlap depending on your limitations.

Zookeeper - 3 nodes to help ensure Kafka is running smoothly, with a little overhead for the crawlers. These boxes typically can be all around or 'general' boxes; with decent IO, RAM, and disk.

Kafka - 5 nodes to ensure your cluster is able to pump and read as much data as you can throw at it. Setting up Kafka can be its own art form, so you want to make sure your data packet size is fairly large, and that you make sure your settings on how much data you store are properly configured. These boxes rely heavily on the internal network, so high IO and no crawlers on these machines.

Redis - 1 node, since Redis is an in memory database you will want a lot of RAM on this box. Your database should only become a problem if your periodic back up files are too large to fit on your machine. This means either that you cannot crawl fast enough and your database is filling up, or you have that much stuff to where it fills your disk. Either way, disk and RAM are the important things here

---

Note: Scrapy Cluster has not been tested against Redis Cluster. If you would like to run Redis as a Cluster used by Scrapy Cluster please take caution as there may be key manipulations that do not scale well across Redis instances.

Kafka Monitors - This is a lightweight process that reads from your Kafka topic of choice. It can be sprinkled on any number of machines you see fit, as long as you have the number of Kafka topics paritions to scale. We would recommend deployment on machines that are close to either Kafka or Redis.

Redis Monitors - Another very lightweight process that can be sprinkled around on any number of machines. Prefers to be close to Redis.

Crawlers - As many small machines as you see fit. Your crawlers need to hit the internet, so put them on machines that have really good network cards or lots of IO. Scale these machines wider, rather than trying to stack a large number of processes on each machine.

Rests - If you need multiple rest endpoints, put them behind a load balancer that allows the work to be spread across multiple hosts. Prefers to be close to Kafka.

Crawler Configuration

Crawler configuration is very important, and it can take some time to find settings that are correct for your setup. Here are some general guidelines for a large Scrapy Cluster:

- Spiders should be spread **thin**, not thick across your machines. How thin? Consider only running 5 to 10 spider process on each machine. This allows you to keep your machine size small, and to scale horizontally on the number of machines you run. This allows for better per-ip rate limits, and to enable the cluster to be more efficient when crawling sites while at the same time not getting your IP blocked. You would be surprised how fast a 5 machine 5 process setup can crawl!
- Have as many different IP addresses as you can. If you can get one IP Address per machine - awesome. The more you stack machines out the same IP Address, the lower the throughput will be on your cluster due to the domain throttling.
- Run the IP Address throttle only (no Spider Type) if you have many different spiders coming from your cluster. It will allow them to orchestrate themselves across spider types to crawl at your desired rate limits. Turning on the Spider Type throttle will eliminate this benefit.
- Don't set your `QUEUE_WINDOW` and `QUEUE_HITS` too high. There is a reason the defaults are `60` and `10`. If you can scale horizontally, you can get your throughput to `10 * (num machines)` and should be able to fit your throughput needs.
- Flip the *Base 64* encode flag on your crawlers. You **will** run across malformed utf-8 characters that breaks `json.dumps()`. It will save you headaches in the long run by ensuring your html is always transmitted to Kafka.

Stats Collection

Stats Collection by Scrapy Cluster is meant to allow users to get a better understanding of what is happening in their cluster without adding additional components. In production, it may be redundant to have *both* Stats Collection and Elasticsearch logging, and you will want to turn off or greatly reduce the number of stats collected by the cluster.

Retaining lots of different stats about every machine and every process is very memory intensive. We recommend reducing the default `STATS_TIMES` to eliminate `SECONDS_1_WEEK` and `SECONDS_1_DAY` at the very least in order to reduce your Redis memory footprint.

The last point about Stats Collection is that it becomes inconsistent with *Docker* style deployments without a defined `hostname` for your container. In some circumstances the stats collection needs the hostname, and when docker autogenerates hostanames for the containers this can create excess data within Redis. In this case, we recommend eliminating Stats Collection from your cluster if you plan on continuously redeploying, or bringing your docker container up and down frequently without a defined hostname.

DNS Cache

The default for Scrapy is to cache DNS queries in memory, but there is no TTL handling as of Scrapy v1.0. While this is fine for short-lived spiders, any persistent spiders can accumulate stale DNS data until the next time they are restarted, potentially resulting in bad page crawls. An example scenario is with Amazon's Elastic Load Balancer, which auto-adds public IPs to the load balancer during high activity. These IPs could later be re-allocated to another website, causing your crawler to get the wrong data.

Scrapy Cluster spiders have the potential to stay up for weeks or months at a time, causing this backlog of bad IPs to grow significantly. We recommend that you set `DNSCACHE_ENABLED = False` in your `localsettings.py` for all of your crawlers in order to prevent this caching and use the system DNS resolver for all queries.

Your system will instead need a DNS cache on the localhost, such as `nscd` or `dnsmasq`, as this will increase requests to your DNS server.

> **Warning:** Ubuntu 12.04 and 14.04 server editions do not come with a preconfigured DNS cache. Turning this setting off without a DNS cache will result in errors from Scrapy.

Response Time

The Scrapy Cluster Response time is dependent on a number of factors:

- How often the Kafka Monitor polls for new messages
- How often any one spider polls redis for new requests
- How many spiders are polling
- How fast the spider can fetch the request
- How fast your item pipeline can get the response into Kafka

With the Kafka Monitor constantly monitoring the incoming topic, there is very little latency for getting a request into the system. The bottleneck occurs mainly in the core Scrapy crawler code.

The more crawlers you have running and spread across the cluster, the lower the average response time will be for a crawler to receive a request. For example if a single spider goes idle and then polls every 5 seconds, you would expect a your maximum response time to be 5 seconds, the minimum response time to be 0 seconds, but on average your

response time should be 2.5 seconds for one spider. As you increase the number of spiders in the system the likelihood that one spider is polling also increases, and the cluster performance will go up.

The next bottleneck in response time is how quickly the request can be conducted by Scrapy, which depends on the speed of the internet connection(s) you are running the Scrapy Cluster behind. This is out of control of the Scrapy Cluster itself, but relies heavily on your ISP or other network configuration.

Once your spider has processed the response and yields an item, your item pipeline before the response gets to Kafka may slow your item down. If you are doing complex processing here, it is recommended you move it out of Scrapy and into a larger architecture.

Overall, your cluster response time for brand new crawls on a domain not yet seen is a lot slower than a domain that is already in the crawl backlog. The more Crawlers you have running, the bigger throughput you will be able to achieve.

Kafka Topics

This page explains the different Kafka Topics that will be created in your Kafka Cluster.

Production

For production deployments you will have at a minimum three operational Kafka Topics.

- **demo.incoming** - The incoming Kafka topic to receive valid JSON requests, as read by the Kafka Monitor
- **demo.crawled_firehose** - All of the crawl data collected from your Crawlers is fed into this topic.
- **demo.outbound_firehose** - All of the Action, Stop, Expire, and Statistics based request results will come out of this topic.

If you have configured Application ID specific topics for the *Redis Monitor* or the *Crawler*, you will have these topics.

- **demo.crawled_<appid>** - Crawl data specific to the `appid` used to submit the request
- **demo.outbound_<appid>** - Info, Stop, Expire, and Stat data specific to the `appid` used to submit the request.

Testing

If you run the online integration tests, there will be dummy Kafka Topics that are created, so as to not interfere with production topics.

- **demo.incoming_test** - Used in the online integration test for the Kafka Monitor
- **demo_test.crawled_firehose** - Used for conducting the online integration test for the Crawlers
- **demo_test.outbound_firehose** - Used in the online integration test for the Redis Monitor

Redis Keys

The following keys within Redis are used by the Scrapy Cluster:

Production

- **timeout:<spiderid>:<appid>:<crawlid>** - The timeout value of the crawl in the system, used by the Redis Monitor. The actual value of the key is the date in seconds since epoch that the crawl with that particular `spiderid`, `appid`, and `crawlid` will expire.
- **<spiderid>:<domain>:queue** - The queue that holds all of the url requests for a spider type of a particular domain. Within this sorted set is any other data associated with the request to be crawled, which is stored as a Json object that is ujson encoded.
- **<spiderid>:dupefilter:<crawlid>** - The duplication filter for the spider type and `crawlid`. This Redis Set stores a scrapy url hash of the urls the crawl request has already seen. This is useful for coordinating the ignoring of urls already seen by the current crawl request.
- **<spiderid>:blacklist** - A permanent blacklist of all stopped and expired `crawlid`'s . This is used by the Scrapy scheduler prevent crawls from continuing once they have been halted via a stop request or an expiring crawl. Any subsequent crawl requests with a `crawlid` in this list will not be crawled past the initial request url.

Warning: The Duplication Filter is only temporary, otherwise every single crawl request will continue to fill up the Redis instance! Since the the goal is to utilize Redis in a way that does not consume too much memory, the filter utilizes Redis's EXPIRE feature, and the key will self delete after a specified time window. The *default* window provided is 600 seconds, which means that if your crawl job with a unique `crawlid` goes for more than 600 seconds without a request, the key will be deleted and you may end up crawling pages you have already seen. Since there is an obvious increase in memory used with an increased timeout window, that is up to the application using Scrapy Cluster to determine what a safe tradeoff is.

- **<spiderid>:<ip_address>:<domain>:throttle_time** - Stores the value for the future calculation on when the next time a moderated throttle key is available to pop. Both `<spiderid>` and `<ip_address>` are dependent on the *throttle style*, and may not be present depending on configuration.
- **<spiderid>:<ip_address>:<domain>:throttle_window** - Stores the number of hits for a particular domain given the Type and IP Throttle Style. Is used by the Scrapy Scheduler to do coordinated throttling across a particular domain. Both `<spiderid>` and `<ip_address>` are dependent on the *throttle style*, and may not be present depending on configuration.
- **rest:poll:<uuid>** - Allows a response caught by a separate Rest service to be stored within Redis to be later retrived by a `poll` request to another. Useful when running multiple Rest processes behind a load balancer with jobs that are longer than the initial call timeout.

Redis Monitor Jobs

- **zk:<action>:<domain>:<appid>** - Stores job info for the Zookeeper plugin
- **stop:<spiderid>:<appid>:<crawlid>** - Used to stop a crawl via the Stop plugin
- **statsrequest:<stats-type>:<appid>** - Used by the Stats plugin to fetch statistics requests
- **info:<spiderid>:<appid>:<crawlid>** - Used by the Info plugin to gather information about an active crawl
- **timeout:<spiderid>:<appid>:<crawlid>** - Used to stop an active crawl when the crawl time window has expired

Statistics

- **stats:kafka-monitor:<plugin>:<window>** - Used to collect plugin statistics of requests that are received by the Kafka Monitor. These keys hold information about the number of successful hits on that plugin in a specific time window.
- **stats:kafka-monitor:total:<window>** - Holds statistics on the total number of requests received by the Kafka Monitor. This contains both successful and unsuccessful API requests.
- **stats:kafka-monitor:fail:<window>** - Holds statistics on the total number of failed API validation attempts by the Kafka Monitor for requests it receives. The value here is based on the defined statistics collection time window.
- **stats:kafka-monitor:self:<machine>:<id>** - Self reporting mechanism for the Kafka Monitor
- **stats:redis-monitor:<plugin>:<window>** - Used to collect plugin statistics of requests that are received by the Redis Monitor. These keys hold information about the number of successful hits on that plugin in a specific time window.
- **stats:redis-monitor:total:<window>** - Holds statistics on the total number of requests received by the Redis Monitor. This contains both successful and unsuccessful attempts at processing in monitored key.
- **stats:redis-monitor:fail:<window>** - Holds statistics on the total number of failed attempts by the Redis Monitor for requests it receives. The value here is based on the defined statistics collection time window.
- **stats:redis-monitor:self:<machine>:<id>** - Self reporting mechanism for the Redis Monitor
- **stats:crawler:<hostname>:<spiderid>:<status_code>** - Holds metrics for the different response codes the crawlers have seen. This key is split by the machine, spider, and response code type to allow you to further examine your different crawler success rates.
- **stats:crawler:self:<machine>:<spiderid>:<id>** - Self reporting mechanism for each individual crawler
- **stats:rest:self:<machine>:<id>** - Self reporting mechanism for each individual Rest service

Testing

If you run the integration tests, there may be temporary Redis keys created that do not interfere with production level deployments.

- **info-test:blah** - Used when testing Redis Monitor has the ability to process and send messages to kafka
- **cluster:test** - Used when testing the Kafka Monitor can act and set a key in Redis
- **test-spider:istresearch.com:queue** - Used when testing the crawler installation can interact with Redis and Kafka
- **stats:crawler:<hostname>:test-spider:<window>** - Automatically created and destoryed during crawler testing by the stats collection mechanism settings.

Other Distributed Scrapy Projects

Scrapy Cluster is not the only project that attempts to use Scrapy in a distributed fashion. Here are some other notable projects:

Scrapy Redis

Github: https://github.com/rolando/scrapy-redis

Description: Scrapy Cluster was born from Scrapy Redis, which offloads Requests to a Redis instance. This allows for spiders to coordinate their crawling efforts via Redis, and pushes the results back into Redis. Scrapy Cluster ended up going in a different enough direction that it was no longer the same project, but if you look closely enough you will see reminents still there.

Frontera

Github: https://github.com/scrapinghub/frontera

Description: Built by the Scrapinghub Team, Frontera is designed to allow agnostic crawl frontier expansion. In particular, it helps determine what a series of crawlers should crawl next via some kind of logic or processing. This shares many of the same features as Scrapy Cluster, including a message bus and the ability to use many popular big data stack applications.

Scrapy RT

Github: https://github.com/scrapinghub/scrapyrt

Description: Scrapy Real Time provides an HTTP interface for working with and scheduling crawls. It provides an HTTP rest server with a series of endpoints to submit and retrieve crawls. Scrapy RT is similar to Scrapy Cluster in that it provides a single API to interact with your crawlers.

Scrapyd

Github: https://github.com/scrapy/scrapyd

Description: Scrapyd is a daemon service for running spiders. It allows you the unique ability to deploy whole spider projects to your Scrapyd instance and run or monitor your crawls. This is similar to Scrapy Cluster in that the spiders are spread across machines, but inherently do not do any orchestration with other crawler machines.

Arachnado

Github: https://github.com/TeamHG-Memex/arachnado

Description: Arachnado is a Tornado based HTTP API and Web UI for using a Scrapy spider to crawl a target website. It allows you to create crawl jobs, execute them, and see aggregate statistics based on your Spider results. It has similar themes to Scrapy Cluster, like statistics and crawl jobs, but does not appear to orchestrate multiple spiders without additional work.

Others

Do you know of another distributed Scrapy project not listed here? Please feel free to *contribute* to make our documentation better.

Upgrade Scrapy Cluster How to update an older version of Scrapy Cluster to the latest

Integration with ELK Visualizing your cluster with the ELK stack gives you new insight into your cluster

Docker Use docker to provision and scale your Scrapy Cluster

Crawling Responsibly Responsible Crawling with Scrapy Cluster

Production Setup Thoughts on Production Scale Deployments

DNS Cache DNS Caching is bad for long lived spiders

Response Time How the production setup influences cluster response times

Kafka Topics The Kafka Topics generated when typically running the cluster

Redis Keys The keys generated when running a Scrapy Cluster in production

Other Distributed Scrapy Projects A comparison with other Scrapy projects that are distributed in nature

CHAPTER 8

Frequently Asked Questions

Common questions about Scrapy Cluster organized by category.

General

I can't get my cluster working, where do I begin?

We always recommend using the latest stable release or commit from the `master` branch. Code pulled from here should be able to successfully complete all `./offline_tests.sh`.

You can then test your online integration settings by running `./online_tests.sh` and determining at what point your tests fail. If all of the online tests *pass* then your cluster appears to be ready for use.

Normally, online test failure is a result of improper settings configuration or network ports not being properly configured. Triple check these!

If you are still stuck please refer the the *Vagrant Quickstart* guide for setting up an example cluster, or the *Troubleshooting* page for more information.

How do I debug a component?

Both the Kafka Monitor and Redis Monitor can have their log level altered by passing the `--log-level` flag to the command of choice. For example, you can see more verbose debug output from the Kafka Monitor's run command with the following command.

```
$ python kafka_monitor.py run --log-level DEBUG
```

You can also alter the `LOG_LEVEL` setting in your `localsettings.py` file to achieve the same effect.

If you wish to debug Scrapy Cluster based components in your Scrapy Spiders, use the `SC_LOG_LEVEL` setting in your `localsettings.py` file to see scrapy cluster based debug output. Normal Scrapy debugging techniques can be applied here as well, as the scrapy cluster debugging is designed to not interfere with Scrapy based debugging.

What branch should I work from?

If you wish to have stable, tested, and documented code please use the `master` branch. For untested, undocumented bleeding edge developer code please see the `dev` branch. All other branches should be offshoots of the `dev` branch and will be merged back in at some point.

Why do you recommend using a `localsettings.py` file instead of altering the `settings.py` file that comes with the components?

Local settings allow you to keep your custom settings for your cluster separate from those provided by default. If we decide to change variable names, add new settings, or alter the default value you now have a merge conflict if you pull that new commit down.

By keeping your settings separate, you can also have more than one setting configuration at a time! For example, you can use the Kafka Monitor to push json into to different Kafka topics for various testing, or have a local debugging vs production setup on your machines. Use the `--settings` flag for either the Kafka Monitor or Redis Monitor to alter their configuration.

Note: The local settings override flag does not apply to the Scrapy settings, Scrapy uses its own style for settings overrides that can be found on this page

---

How do I deploy my cluster in an automated fashion?

Deploying a scrapy cluster in an automated fashion is highly dependent on the environment **you** are working in. Because we cannot control the OS you are running, packages installed, or network setup, it is best recommended you use an automated deployment framework that fits your needs. Some suggestions include Ansible, Puppet, Chef, Salt, Anaconda Cluster, etc.

Do you support Docker?

Docker support is new with the `1.2` release, please see the *Docker* guide for more information.

Are there other distributed Scrapy projects?

Yes! Please see our breakdown at *Other Distributed Scrapy Projects*

I would like to contribute but do not know where to begin, how can I help?

You can find suggestions of things we could use help on *here*.

How do I contact the community surrounding Scrapy Cluster?

Feel free to reach out by joining the Gitter chat room, send an email to scrapy-cluster-tech@istresearch.com, or for more formal issues please *raise an issue*.

Kafka Monitor

How do I extend the Kafka Monitor to fit my needs?

Please see the plugin documentation *here* for adding new plugins to the Kafka Monitor. If you would like to contribute to core Kafka Monitor development please consider looking at our guide for *Submitting Pull Requests*.

Crawler

How do I create a Scrapy Spider that works with the cluster?

To use everything scrapy cluster has to offer with your new Spider, you need your class to inherit from our `RedisSpider` base class.

You can also yield new Requests or items like a normal Scrapy Spider. For more information see the *crawl extension* documentation.

Can I use everything else that the original Scrapy has to offer, like middlewares, pipelines, etc?

Yes, you can. Our core logic relies on a heavily customized Scheduler which is not normally exposed to users. If Scrapy Cluster hinders use of a Scrapy ability you need please let us know.

Do I have to restart my Scrapy Cluster Crawlers when I push a new domain specific configuration?

No, the crawlers will receive a notification from Zookeeper that their configuration has changed. They will then automatically update to the new desired settings, without a restart. For more information please see *here*.

How do I use Scrapy `start_urls` with Scrapy Cluster?

Don't put `start_urls` within your Scrapy Cluster spiders! Use the *Crawl API* to feed those initial urls into your cluster. This will ensure the crawl is not duplicated by many spiders running and the same time, and that the crawl has all the meta-data it needs to be successful.

Redis Monitor

How do I extend the Redis Monitor to fit my needs?

Please see the plugin documentation *here* for adding new plugins to the Redis Monitor. If you would like to contribute to core Redis Monitor development please consider looking at our guide for *Submitting Pull Requests*.

Rest

My rest endpoint reports RED or YELLOW, how do I fix this?

A red or yellow status indicates the service cannot connect to one or more components. There should be error logs indicating a failure or loss of connection to a particular component.

Utilities

Are the utilities dependent on Scrapy Cluster?

No! The utilities package is located on PyPi here and can be downloaded and used independently of this project.

Have a question that isn't answered here or in our documentation? Feel free to read our *Raising Issues* guidelines about opening an issue.

CHAPTER 9

Troubleshooting

This document will help you in debugging problems seen in the components of Scrapy Cluster, by utilizing the *Log Factory* utility class.

General Debugging

Logging

For all three components, you may set the Scrapy Cluster log level for the component to `DEBUG` in order to see more verbose output. These logs are hopefully verbose enough to help you figure out where things are breaking, or help you trace through the code to find where the bug lies.

- **Kafka Monitor** - use the `--log-level DEBUG` flag when executing either the `run` or `feed` command, or in your `localsettings.py` set `LOG_LEVEL="DEBUG"`
- **Redis Monitor** - use the `--log-level DEBUG` flag when executing either the main command, or in your `localsettings.py` set `LOG_LEVEL="DEBUG"`
- **Crawler** - use the `localsettings.py` file to set `SC_LOG_LEVEL="DEBUG"`. Note this is **different** so as to not interfere with normal Scrapy log levels.
- **Rest** - use the `--log-level DEBUG` flag when executing either the main command, or in your `localsettings.py` set `LOG_LEVEL="DEBUG"`

You can alter the Log Factory settings to log your data in JSON, to write your Scrapy Cluster logs to a file, or to write only important `CRITICAL` log messages to your desired output. JSON based logs will show all of the extra data passed into the log message, and can be useful for debugging python dictionaries without contaminating your log message itself.

Note: Command line arguments take precedence over `localsettings.py`, which take precedence over the default settings. This is useful if you need quick command line changes to your logging output, but want to keep production settings the same.

Offline Testing

All of the different components in Scrapy Cluster have offline tests, to ensure correct method functionality and error handling. You can run all of the offline tests in the top level of this project by issuing the following command.

```
$ ./run_offline_tests.sh
```

If you are making modifications to core components, or wish to add new functionality to Scrapy Cluster, please ensure that all the tests pass for your component. Upgrades and changes happen, but these tests should always pass in the end.

If you are modifying a single component, you can run its individual offline test by using nose. The following commands should be run from within the component you are interested in, and will run both the nosetests and provide code coverage information to you:

```
# utilities
nosetests -v --with-coverage --cover-erase

# kafka monitor
nosetests -v --with-coverage --cover-erase --cover-package=../kafka-monitor/

# redis monitor
nosetests -v --with-coverage --cover-erase --cover-package=../redis-monitor/

# crawler
nosetests -v --with-coverage --cover-erase --cover-package=crawling/

# rest
nosetests -v --with-coverage --cover-erase --cover-package=../rest/
```

This runs the individual component's offline tests. You can do this in the Kafka Monitor, Crawler, Redis Monitor, Rest, and the Utilities folders.

Online Testing

Online tests serve to ensure functionality between components is working smoothly. These 'integration tests' are strictly for testing and debugging new setups one component at a time.

Running `./run_online_tests.sh` right away when you download Scrapy Cluster will most likely result in a failure. Instead, you should create small `localsettings.py` files in the three core components that will override Scrapy Cluster's built in default settings. A typical file might look like this:

```
REDIS_HOST = '<your_host>'
KAFKA_HOSTS = '<your_host>:9092'
```

Where `<your_host>` is the machine where Redis, Kafka, or Zookeeper would reside. Once you set this up, each of the three main components can run their online tests like so:

```
$ python tests/online.py -v
```

If your system is properly configured you will see the test pass, otherwise, the debug and error log output should indicate what is failing.

Note: The Crawler online test takes a little over 20 seconds to complete, so be patient!

If you would like to debug the Utilities package, the online test is slightly different as we need to pass the redis host as a flag.

```
python tests/online.py -r <your_host>
```

If all your online tests pass, that means that the Scrapy Cluster component was successfully able to talk with its dependencies and deems itself operational.

Kafka Monitor

If you need to add extra lines to debug the Kafka Monitor, the LogFactory logger is in the following variables.

- **Core**: `self.logger`
- **Plugins**: `self.logger`

Typical Issues

- Cannot connect to Redis/Kafka, look into your network configuration.
- Kafka is in an unhappy state, debugging should be done for Kafka.

Crawler

If you need to add extra lines to debug an item within the Scrapy Project, you can find the LogFactory logger in the following variables.

- **Scheduler**: `self.logger`
- **Kafka and Logging Pipelines**: `self.logger`
- **Spiders**: `self._logger` (We can't override the spider's `self.logger`)
- **Log Retry Middleware**: `self.logger`

Note: It is important that you always use the `LogFactory.get_instance()` method if you need another Scrapy Cluster logger elsewhere in your project. Due to the way Twisted instantiates each of the threads you can end up with multiple loggers which ends up duplicating your log data.

The LogFactory does not interfere with the Scrapy based logger, so if you are more comfortable using it then you are free to tinker with the Scrapy logging settings here.

Typical Issues

- Continuous 504 Timeouts may indicate your spider machines cannot reach the public internet
- Cannot connect to Redis/Kafka/Zookeeper, look into your network configuration.
- Lots of errors when writing to a Kafka topic - Kafka is in an unhappy state and should be looked at.

Redis Monitor

To add further debug lines within the Redis Monitor, you can use the following variables within the classes.

- **Core**: `self.logger`
- **Plugins**: `self.logger`

Typical Issues

- Cannot connect to Redis/Kafka, look into your network configuration.
- Lots of errors when writing to a Kafka topic - Kafka is in an unhappy state and should be looked at.

Rest

To add further debug lines within the Rest, you can use the following variables within the classes.

- **Core**: `self.logger`

Typical Issues

- Cannot connect to Redis/Kafka, look into your network configuration.
- Improperly formatted requests - please ensure your request matches either the Kafka Monitor or Rest service API
- All Redis Monitor requests come back as a `poll_id` - Ensure you have the Kafka Monitor and Redis Monitor properly set up and running.

Utilities

The utilities do not instantiate a LogFactory based logger, as that would create a cyclic dependency on itself. Instead, you can use your standard logging methods or print statements to debug things you think are not working within the utilities.

If you wish to test your changes, you can run the offline/online tests and then run

```
python setup.py install
```

To overwrite your existing pip package installation with your updated code.

Data Stack

This project is not meant to help users in debugging the big data applications it relies upon, as they do it best. You should refer to the following references for more help.

Zookeeper

You should refer to the official Zookeeper documentation for help in setting up Zookeeper. From all your machines, you should be able to run the following command to talk to Zookeeper

```
$ echo ruok | nc scdev 2181
imok
```

Main Port: 2181

Kafka

Please refer to the official Kafka documentation for instructions and help on setting up your Kafka cluster. You can use the following command in the Kafka Monitor to check access to your Kafka machines. For example:

```
$ python kafkadump.py list
2016-01-04 17:30:03,634 [kafkadump] INFO: Connected to scdev:9092
Topics:
- demo.outbound_firehose
- demo.outbound_testapp
- demo.crawled_firehose
- demo.outbound_testApp
- demo_test.outbound_firehose
- demo_test.crawled_firehose
- demo.outbound_testapp
- demo.incoming
```

Main Port: 9092

Redis

Refer to the official Redis documentation for more information on how to set up your Redis instance. A simple test of your redis instance can be done with the following commands.

```
$ vagrant ssh
vagrant@scdev:~$ /opt/redis/default/bin/redis-cli
127.0.0.1:6379> info
# Server
redis_version:3.0.5
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:71f1349fddec31b1
redis_mode:standalone
os:Linux 3.13.0-66-generic x86_64
...
```

Main Port: 6379

CHAPTER 10

Contributing

There are a number of ways to contribute to Scrapy Cluster.

Raising Issues

If you're thinking about raising an issue because you think you've found a problem with Scrapy Cluster, you'd like to make a request for a new feature in the codebase, or any other reason... please read this first.

The GitHub issue tracker is the preferred channel for *Bug Reports*, Feature Requests, or *Change Requests*, but please respect the following restrictions:

- Please **search for existing issues**. Help us keep duplicate issues to a minimum by checking to see if someone has already reported your problem or requested your idea.
- Please **do not** use the issue tracker for personal setup or support requests. If the symptom doesn't appear in the *Vagrant Quickstart* but appears in your production setup, it is not a Scrapy Cluster issue.

For all other chatter, feel free to join the *Community Chat* set up for Scrapy Cluster.

Bug Reports

A bug is a *demonstrable problem* that is caused by the code in the repository. Good bug reports are extremely helpful - thank you!

Guidelines for bug reports:

1. **Use the GitHub issue search**, check if the issue has already been reported.
2. **Check if the issue has been fixed**, try to reproduce it using the latest *dev* branch or look for closed issues in the current milestone.
3. **Isolate the problem**, ideally create a minimal test case to demonstrate a problem with the current build.

4. **Include as much info as possible!** This includes but is not limited to, version of Kafka, Zookeeper, Redis, Scrapy Cluster (release, branch, or commit), OS, Pip Packages, or anything else that may impact the issue you are raising.

A good bug report shouldn't leave others needing to chase you up for more information.

Feature Requests

If you've got a great idea, we want to hear about it! Please label the issue you create with `feature request` to distinguish it from other issues.

Before making a suggestion, here are a few handy tips on what to consider:

1. Visit the Milestones and search through the known feature requests to see if the feature has already been requested, or is on our roadmap already.
2. Check out *Working on Scrapy Cluster Core* - this explains the guidelines for what fits into the scope and aims of the project.
3. Think about whether your feature is for the Kafka Monitor, Redis Monitor, the Crawlers, or does it affect multiple areas? This can help when describing your idea.
4. Remember, it's up to *you* to make a strong case to convince the project's leaders of the merits of a new feature. Please provide as much detail and context as possible - this means explaining the use case and why it is likely to be common.

Change Requests

Change requests cover both architectural and functional changes to how Scrapy Cluster works. If you have an idea for a new or different dependency, a refactor, or an improvement to a feature, etc - please be sure to:

1. **Use the GitHub search** and check someone else didn't get there first
2. Take a moment to think about the best way to make a case for, and explain what you're thinking as it's up to you to convince the project's leaders the change is worthwhile. Some questions to consider are:
    - Is it really one idea or is it many?
    - What problem are you solving?
    - Why is what you are suggesting better than what's already there?
3. Label your issue with `change request` to help identify it within the issue tracker.

Community Chat

If you would like to add a comment about Scrapy Cluster or to interact with the community surrounding and supporting Scrapy Cluster, feel free to join our Gitter chat room. This is a place where both developers and community users can get together to talk about and discuss Scrapy Cluster.

Submitting Pull Requests

Pull requests are awesome! Sometimes we simply do not have enough time in the day to get to everything we want to cover, or did not think of a different use case for Scrapy Cluster.

If you're looking to raise a PR for something which doesn't have an open issue, please think carefully about raising an issue which your PR can close, especially if you're fixing a bug. This makes it more likely that there will be enough information available for your PR to be properly tested and merged.

To make sure your PR is accepted as quickly as possible, please ensure you hit the following points:

- You can run `./run_tests_offline.sh` **and** `./run_tests_online.sh` in your Vagrant test environment to ensure nothing is broken.
- Did you add new code? That means it is probably unit-testable and should have a new unit test associated with it.
- Your PR encapsulates a single alteration or enhancement to Scrapy Cluster, please do not try to fix multiple issues with a single PR without raising it on Github first.
- If you are adding a significant feature, please put it in the *Change Log* for the current Milestone we are driving towards.

Once you have a minimal pull request please submit it to the `dev` branch. That is where our core development work is conducted and we would love to have it in the latest bleeding edge build.

Testing and Quality Assurance

Never underestimate just how useful quality assurance is. If you're looking to get involved with the code base and don't know where to start, checking out and testing a pull requests or the latest `dev` branch is one of the most useful things you can help with.

Essentially:

1. Checkout the latest `dev` branch.
2. Follow one of our *Quick Start* guides to get your cluster up and running.
3. Poke around our documentation, try to follow any of the other guides or ensure that we are explaining ourselves as clear as possible.
4. Find anything odd? Please follow the *Bug Reports* guidelines and let us know!

Documentation

Scrapy Cluster's documentation can be found on Read the Docs. If you have feedback or would like to write some user documentation, please let us know in our *Community Chat* room or by raising and issue and submitting a PR on how our documentation could be improved.

Working on Scrapy Cluster Core

Are you looking to help develop core functionality for Scrapy Cluster? Awesome! Please see the *Vagrant Quickstart* guide for a Vagrant Image to use to test small scale deployments of Scrapy Cluster. If you are looking to do large scale testing and development, please first ensure you can work with the Vagrant Image first.

If something goes wrong, please see the *Troubleshooting* guide first.

Looking for something to work on?

If you're interested in contributing to Scrapy Cluster and don't know where to start, here's a few things to consider.

- **We are trying to build an generic framework for large scale, distributed web crawling.** Code that is applicable only to your setup, installation, or use case may not be helpful to everyone else. The framework and code we create should be extendable, helpful, and improve the ability to succeed in this mission.
- Look for issues that are labeled with the **current** milestone and see if someone is already working on it. Leave a comment stating why you would like to work on this or the skills you can contribute to help complete the task.
- **Do not** begin development on features or issues outside of the current milestone. If you must, please submit an issue or comment and explain your motivation for working on something that we haven't quite gotten to yet.
- Do you have a neat idea or implementation for a new plugin or extenstion to any of the three core areas? We would love to hear about it or help guide you in building it.
- Test and use the Scrapy Cluster Utils Package in your other projects! We would love feedback on how to improve the utilities package, it has been extremely helpful to us in developing Scrapy Cluster. More documentation can be found in the *Utilites* section.
- Feel like you have a pretty good grasp with Scrapy Cluster? Please consider doing **large scale testing** of crawlers (10-20 machines at least, with 10-20 spiders per machine), and have the cluster crawl what ever your heart desires. Where are the runtime bottlenecks? Where can our algorithms be improved? Does certain cluster setups slow down crawling considerably? We are always looking to improve.
- Are you an expert in some other field where we lack? (Docker, Mesos, Conda, Python 3, etc) Please consider how you you can contribute to the project and talk with us on where we think you can best help.

If you're still stuck, feel free to send any of the core developers an message in the *Community Chat* as we are always happy to help.

Key Branches

- `master` - (link to master) this branch reflects the lastest stable release. Hotfixes done to this branch should also be reflected in the `dev` branch
- `dev` - (link to dev) the main developer branch. Go here for the latest bleeding edge code

Other branches represent other features core developers are working on and will be merged back into the main `dev` branch once the feature is complete.

CHAPTER 11

Change Log

This page serves to document any changes made between releases.

Scrapy Cluster 1.2

Date: 03/29/2017

- Added Coveralls code coverage integration
- Added full stack offline unit tests and online integration testing in Travis CI
- Upgraded all components to newest Python packages
- Switched example Virtual Machine from Miniconda to Virtualenv
- Add setting to specify Redis db across all components
- Docker support
- Improved RedisThrottledQueue implementation to allow for rubber band catch up while under moderation
- Added support for Centos and Ubuntu Virtual Machines

Kafka Monitor Changes

- Updated stats schema to accept `queue`, `rest` statistics requests.
- Added plugin API for managing Zookeeper domain configuration
- Added ability to scale horizontally with multiple processes for redundancy

Redis Monitor Changes

- Added `close()` method to plugin base class for clean shutdown

- Added queue statistics response object
- Added plugin for executing Zookeeper configuration updates
- Added ability to scale horizontally with multiple processes for redundancy
- New limited retry ability for failed actions

Crawler Changes

- Added ability to control cluster wide blacklists via Zookeeper
- Improved memory management in scheduler for domain based queues
- Added two new spider middlewares for stats collection and meta field passthrough
- Removed excess pipeline middleware

Rest Service

- New component that allows for restful integration with Scrapy Cluster

Scrapy Cluster 1.1

Date: 02/23/2016

- Added domain based queue mechanism for better management and control across all components of the cluster
- Added easy offline bash script for running all offline tests
- Added online bash test script for testing if your cluster is integrated with all other external components
- New Vagrant virtual machine for easier development and testing.
- Modified demo incoming kafka topic from `demo.incoming_urls` to just `demo.incoming` as now all crawl/info/stop requests are serviced through a single topic
- Added new `scutils` package for accessing modules across different components.
- Added `scutils` documentation
- Added significantly more documentation and improved layout
- Created new `elk` folder for sample Elasticsearch, Logstash, Kibana integration

Kafka Monitor Changes

- Condensed the Crawler and Actions monitor into a single script
- Renamed `kafka-monitor.py` to `kafka_monitor.py` for better PEP 8 standards
- Added plugin functionality for easier extension creation
- Improved kafka topic dump utility
- Added both offline and online unit tests
- Improved logging
- Added defaults to `scraper_schema.json`

- Added Stats Collection and interface for retrieving stats

Redis Monitor Changes

- Added plugin functionality for easier extension creation
- Added both offline and online unit tests
- Improved logging
- Added Stats Collection

Crawler Changes

- Upgraded Crawler to be compatible with Scrapy 1.0
- Improved code structure for overriding url.encode in default LxmlParserLinkExtractor
- Improved logging
- Added ability for the crawling rate to be controlled in a manner that will rate limit the whole crawling cluster based upon the domain, spider type, and public ip address the crawlers have.
- Added ability for the crawl rate to be explicitly defined per domain in Zookeeper, with the ability to dynamically update them on the fly
- Created manual crawler Zookeeper configuration pusher
- Updated offline and added online unit tests
- Added response code stats collection
- Added example Wandering Spider

Scrapy Cluster 1.0

Date: 5/21/2015

- Initial Release

CHAPTER 12

License

```
The MIT License (MIT)

Copyright (c) 2017 IST Research

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 13

Introduction

Overview Learn about the Scrapy Cluster Architecture.

Quick Start A Quick Start guide to those who want to jump right in.

CHAPTER 14

Architectural Components

Kafka Monitor

Design Learn about the design considerations for the Kafka Monitor

Quick Start How to use and run the Kafka Monitor

API The default Kafka API the comes with Scrapy Cluster

Plugins Gives an overview of the different plugin components within the Kafka Monitor, and how to make your own.

Settings Explains all of the settings used by the Kafka Monitor

Crawler

Design Learn about the design considerations for the Scrapy Cluster Crawler

Quick Start How to use and run the distributed crawlers

Controlling Learning how to control your Scrapy Cluster will enable you to get the most out of it

Extension How to use both Scrapy and Scrapy Cluster to enhance your crawling capabilites

Settings Explains all of the settings used by the Crawler

Redis Monitor

Design Learn about the design considerations for the Redis Monitor

Quick Start How to use and run the Redis Monitor

Plugins Gives an overview of the different plugin components within the Redis Monitor, and how to make your own.

Settings Explains all of the settings used by the Redis Monitor

Rest

Design Learn about the design considerations for the Rest service

Quick Start How to use and run the Rest service

API The API the comes with the endpoint

Settings Explains all of the settings used by the Rest component

Utilities

Argparse Helper Simple module to assist in argument parsing with subparsers.

Log Factory Module for logging multithreaded or concurrent processes to files, stdout, and/or json.

Method Timer A method decorator to timeout function calls.

Redis Queue A module for creating easy redis based FIFO, Stack, and Priority Queues.

Redis Throttled Queue A wrapper around the `redis_queue` module to enable distributed throttled pops from the queue.

Settings Wrapper Easy to use module to load both default and local settings for your python application and provides a dictionary object in return.

Stats Collector Module for statistics based collection in Redis, including counters, rolling time windows, and hyperloglog counters.

Zookeeper Watcher Module for watching a zookeeper file and handles zookeeper session connection troubles and re-establishment of watches.

CHAPTER 15

Advanced Topics

Upgrade Scrapy Cluster How to update an older version of Scrapy Cluster to the latest

Integration with ELK Visualizing your cluster with the ELK stack gives you new insight into your cluster

Docker Use docker to provision and scale your Scrapy Cluster

Crawling Responsibly Responsible Crawling with Scrapy Cluster

Production Setup Thoughts on Production Scale Deployments

DNS Cache DNS Caching is bad for long lived spiders

Response Time How the production setup influences cluster response times

Kafka Topics The Kafka Topics generated when typically running the cluster

Redis Keys The keys generated when running a Scrapy Cluster in production

Other Distributed Scrapy Projects A comparison with other Scrapy projects that are distributed in nature

CHAPTER 16

Miscellaneous

Frequently Asked Questions Scrapy Cluster FAQ

Troubleshooting Debugging distributed applications is hard, learn how easy it is to debug Scrapy Cluster.

Contributing Learn how to contribute to Scrapy Cluster

Change Log View the changes between versions of Scrapy Cluster.

License Scrapy Cluster is licensed under the MIT License.

Index

Symbols

B

C

D

E

G

H

I

L

P

R

S

T

U

V

W

Z