Scrapy Cluster Documentation

Release 1.0

IST Research

May 21, 2015

Contents

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

CHAPTER 1

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 of the crawl requests.

1.1 Dependencies

Please see `requirements.txt` for Pip package dependencies across the different sub projects.

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

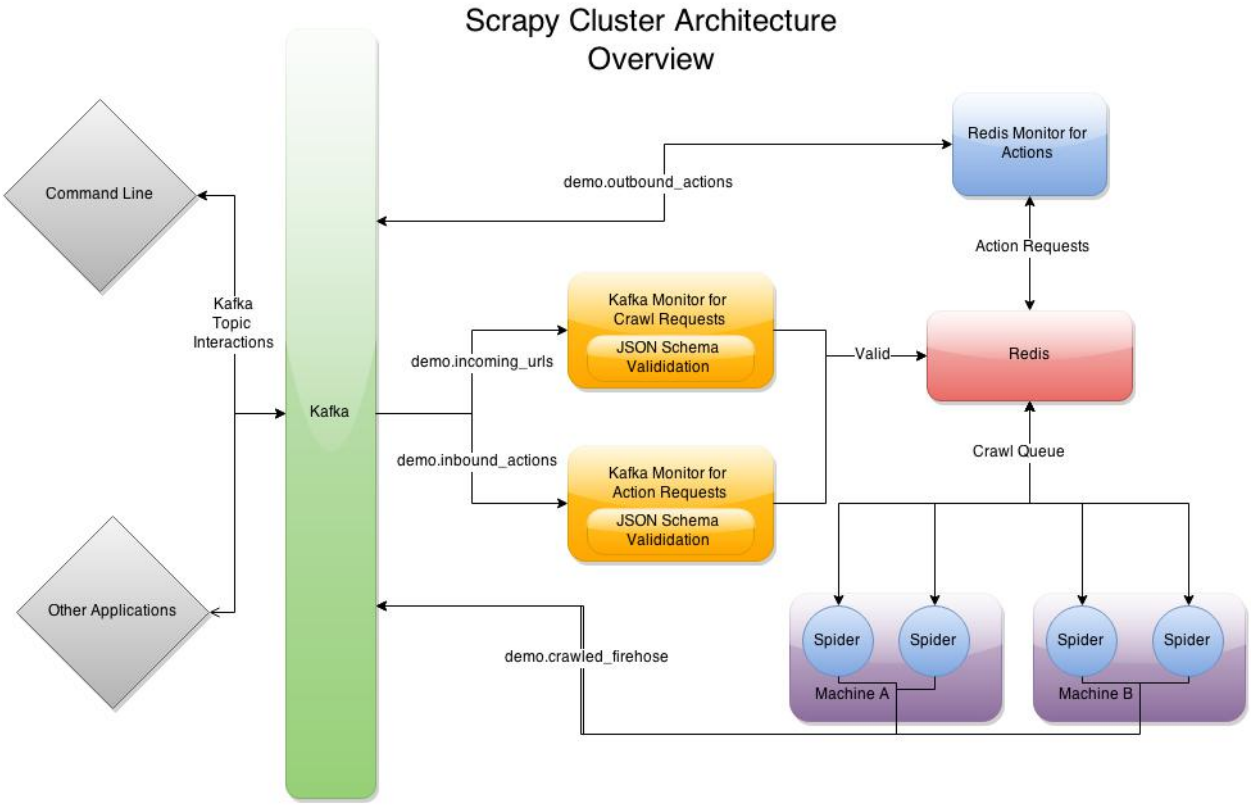
1.2 Core Concepts

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 across scraping jobs or have multiple scraping jobs going at the same time
- Allows for unparalleled 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)

1.3 Architecture Diagram



CHAPTER 2

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.

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 project here.

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

3. You will now need to configure the following four settings files:

1. `~/scrapy-cluster/kafka-monitor/settings-crawling.py`

Add your specific configuraiton to `REDIS_HOST`, `REDIS_PORT`, and `KAFKA_HOSTS`.

For example:

```
REDIS_HOST = 'server-1'
REDIS_PORT = 6379
KAFKA_HOSTS = 'server-2:9092'
```

This is used to determine where the Kafka Monitor will listen to incoming crawl requests and where send them. For this example, lets assume `server-1` houses Redis and `server-2` houses Kafka.

2. `~/scrapy-cluster/kafka-monitor/settings-actions.py`

Add your specific configuraiton to `REDIS_HOST`, `REDIS_PORT`, and `KAFKA_HOSTS`.

For example:

```
REDIS_HOST = 'server-1'
REDIS_PORT = 6379
KAFKA_HOSTS = 'server-2:9092'
```

Notice this is very similar to the first settings file, but the other parameters are different. This settings file is used to listen for action requests and determines where to send them.

3. `~/scrapy-cluster/crawler/crawling/settings.py`

Add your specific configuraiton to `REDIS_HOST`, `REDIS_PORT`, and `KAFKA_HOSTS`.

For example:

```
REDIS_HOST = 'server-1'
REDIS_PORT = 6379
KAFKA_HOSTS = 'server-2:9092'
```

This settings file is used to configure Scrapy. It uses all of the configurations already built in with the project, with a few more to get us into cluster mode. The new settings are utilized by the scheduler and item pipeline.

4. `~/scrapy-cluster/redis-monitor/settings.py`

Add your specific configuraiton to `REDIS_HOST`, `REDIS_PORT`, and `KAFKA_HOSTS`.

For example:

```
REDIS_HOST = 'server-1'
REDIS_PORT = 6379
KAFKA_HOSTS = 'server-2:9092'
```

This last settings file is used to get information out of the redis queue, and tells the redis monitor where to point.

4. At this point we can 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.

Bare Bones:

- The Kafka Monitor for Crawling:

```
python kafka-monitor.py run -s settings_crawling.py
```

- A crawler:

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

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

```
python kafkadump.py dump demo.crawled_firehose --host=server-2:9092
```

Fully Operational:

- The Kafka Monitor for Crawling:

```
python kafka-monitor.py run -s settings_crawling.py
```

- The Kafka Monitor for Actions:

```
python kafka-monitor.py run -s settings_actions.py
```

- The Redis Monitor:

```
python redis-monitor.py
```

- A crawler (1+):

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

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

```
python kafkadump.py dump demo.crawled_firehose --host=server-2:9092
```

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

```
python kafkadump.py dump demo.outbound_firehose --host=server-2:9092
```

5. 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:

```
=> feeding JSON request into demo.incoming_urls...
{
    "url": "http://istresearch.com",
    "crawlid": "abc123",
    "appid": "testapp"
}
=> done feeding request.
```

- If this command hangs, it means the script cannot connect to Kafka

6. 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
7. 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", "max
```

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

```
python kafka-monitor.py feed -s settings_actions.py '{"action":"info", "appid":"testapp", "uuid":"som
```

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': 1430170027, u'crawlid': u'abc1234', u'total_pending': 48, u'low_priority': -19, u'hi
```

In this case we had 48 urls pending in the queue, so yours may be slightly different.

8. If the crawl from step 7 is still running, lets stop it by issuing a `stop` action request (this requires a full deployment).

Action Request:

```
python kafka-monitor.py feed -s settings_actions.py '{"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'action': u'stop', u'total_purged': 48, u'spiderid': u'link', u'crawlid': u'abc1234', u'appid': u't
```

In this case we had 48 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 at each of the components, please continue reading the documentation for each component.

CHAPTER 3

Kafka Monitor

The Kafka Monitor serves as the entry point into the crawler architecture. It validates both incoming crawl requests and incoming action requests. The files included within can be used to submit new crawl requests, gather information about currently running crawls, and dump results to the command line.

3.1 Quick Start

First, make sure your *settings_crawler.py* and *settings_actions.py* are updated with your Kafka and Redis hosts.

Then run the kafka monitor for crawl requests:

```
python kafka-monitor.py run -s settings_crawling.py
```

Then run the kafka monitor for action requests:

```
python kafka-monitor.py run -s settings_actions.py
```

Finally, submit a new crawl request:

```
python kafka-monitor.py feed -s settings_crawling.py '{"url": "http://istresearch.com", "appid":"test
```

If you have everything else in the pipeline set up correctly you should now see the raw html of the IST Research home page come through. You need both of these to run the full cluster, but at a minimum you should have the `-s settings_crawling` monitor running.

- For how to set up the Scrapy crawlers, refer to the Crawler documentation
- To learn more about how to see crawl info, please see the Redis Monitor documentation

3.2 Design Considerations

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 and stop their crawls from that same interface, so we decided to make a dynamic interface that could support all of the request needs, but utilize the same base code. In the future this base code could expanded to handle any different style of request, as long as there was a validation of the request and a place to send the result to.

From our own internal debugging and ensuring other applications were working properly, a utility program was also created on the side in order to be able to interact and monitor the kafka messages coming through. This dump utility can be used to monitor any of the Kafka topics within the cluster.

3.3 Components

This section explains the individual files located within the kafka monitor project.

3.3.1 kafka_monitor.py

The Kafka Monitor consists of the following two different functionalities

```
python kafka-monitor.py run
Usage:
    monitor run --settings=<settings>
    monitor feed --settings=<settings> <json_req>
```

When in `run` mode, the class monitors the designated kafka topic and validates it against a formal Json Schema Specification. After the formal json request is validated, the designated `SCHEMA_METHOD` handler function is then called to process your data. We designed it in this fashion so you could deploy then same simple python file, but dynamically configure it to validate and process your request based on a tunable configuration file. The formal specifications for submitted crawl requests and action requests are outlined in below section.

When in `feed` mode, the class acts as a simple interface to validate and submit your command line json request. An example is shown in the above section when you submit a new crawl request. Anything in the formal json specification can be submitted via the command line or via a Kafka message generated by another application. The feeder is only used when you need to submit manual crawls to the API for debugging, and any major application should connect via Kafka.

Incoming Crawl Request Kafka Topic:

- `demo.incoming_urls` - The topic to feed properly formatted crawl requests to

Outbound Crawl 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: For more information about the topics generated and used by the Redis Monitor, please see the Redis Monitor documentation

---

Example Crawl Requests:

```
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",
```

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

```
python wat-scrapy-kafka-monitor.py feed '{"url": "http://aol.com/", "appid":"testapp", "crawlid":"a2
```

- 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.

Example Crawl Request Output from the kafkadump utility:

```
{
    u'body': u'<real raw html source here>',
    u'crawlid': u'abc1234',
    u'links': [],
    u'response_url': u'http://www.dmoz.org/Recreation/Food/',
    u'url': u'http://www.dmoz.org/Recreation/Food/',
    u'status_code': 200,
    u'status_msg': u'OK',
    u'appid': u'testapp',
    u'headers': {
        u'Cteonnt-Length': [u'40707'],
        u'Content-Language': [u'en'],
        u'Set-Cookie': [u'JSESSIONID=FB02F2BBDBDBDDE8FBE5E1B81B4219E6; Path=/'],
        u'Server': [u'Apache'],
        u'Date': [u'Mon, 27 Apr 2015 21:26:24 GMT'],
        u'Content-Type': [u'text/html;charset=UTF-8']
    },
    u'attrs': {},
    u'timestamp': u'2015-04-27T21:26:24.095468'
}
```

For a full specification as to how you can control the Scrapy Cluster crawl parameters, please refer to the *scraper_schema.json* documentation.

3.3.2 kafkadump.py

The Kafka dump utility stemmed from the need to quickly view the resulting kafka crawl results. This is a simple utility designed to do two things:

```
python kafkadump.py --help

Usage:
    kafkadump list --host=<host>
    kafkadump dump <topic> --host=<host> [--consumer=<consumer>]
```

When ran with the command `list` the utility will dump out all of the topics created on your cluster.

When ran with the `dump` command, the utility will connect to Kafka and dump every message in that topic starting from the beginning. Once it hits the end it will sit there and wait for new data to stream through. This is especially useful when doing command line debugging of the cluster to ensure that crawl results are flowing back out from the system, or for monitoring the results of information requests.

3.3.3 scraper_schema.json

The Scraper Schema defines the level of interaction an application gets with the Scrapy Cluster. 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.
- **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.

3.3.4 action_schema.json

The Action Schema allows for extra information to be gathered from the Scrapy Cluster, 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.

CHAPTER 4

Crawler

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.

4.1 Quick Start

This is a complete Scrapy crawling project.

First, make sure your `settings.py` is updated with your Kafka and Redis hosts.

To run the crawler offline test to make sure nothing is broken:

```
python tests/tests_offline.py -v
```

Then run the crawler:

```
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 in order to add crawling capability.

- To execute a crawl, please refer the Kafka Monitor documentation
- To learn more about how to see crawl info, please see the Redis Monitor documentation

4.2 Design Considerations

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 a priority, and for every subsequent level deep in the crawl that priority decreases by 10. An example diagram is shown below:

As you can see above. the initial seed url generates 4 new links. Since we are using a priority based queue, the spiders continue to pop from the highest priority crawl request, and then decrease the priority for level deep they are from the parent request. Any new links are fed back into the same exact queue mechanism but with a lower priority to allow for the equal leveled 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 spider based queue. This allows you to have multiple spiders doing vastly different link strategies coordinated via the same Redis instance, and even on the same 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 queue 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

4.3 Components

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

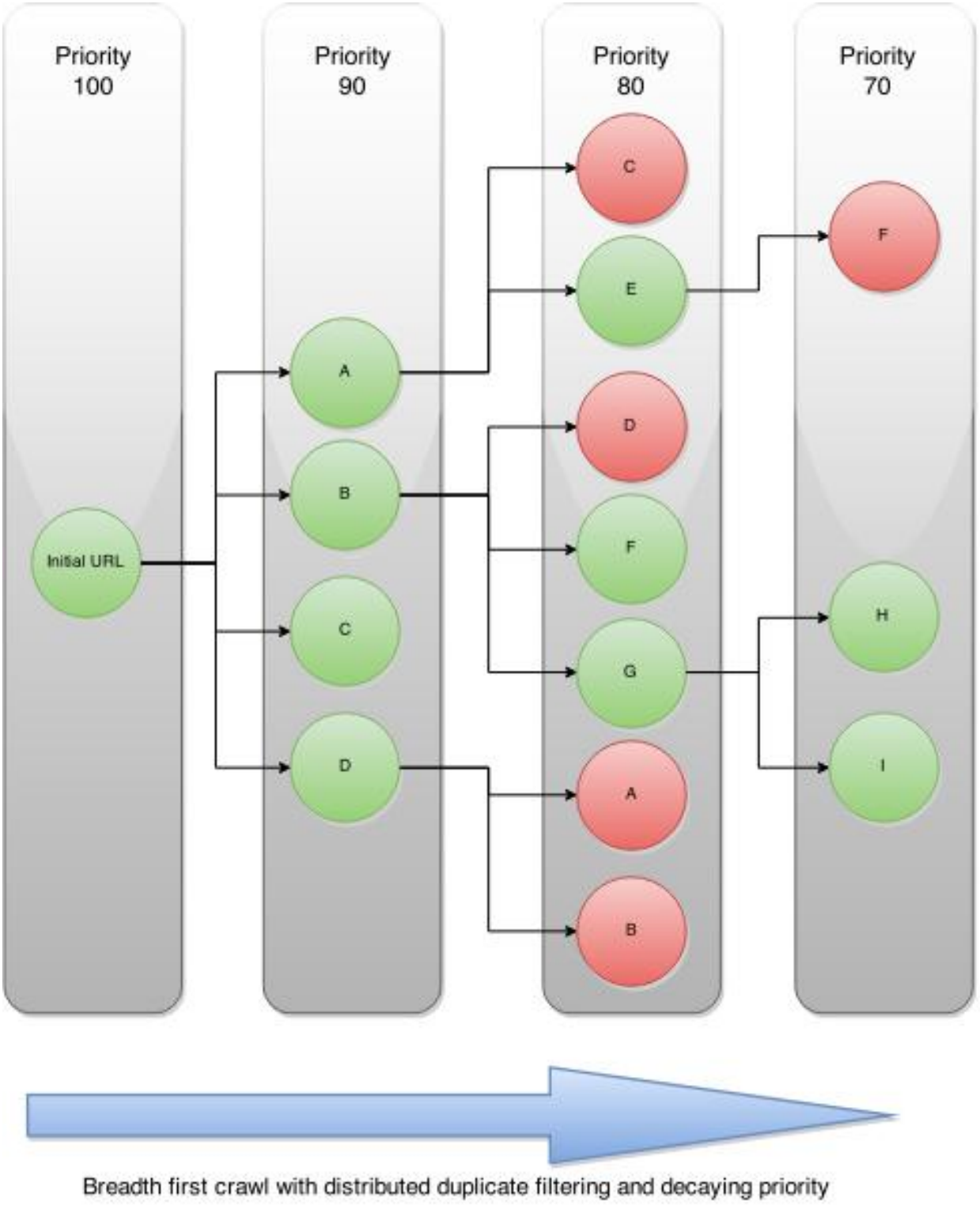
4.3.1 distributed_scheduler.py

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

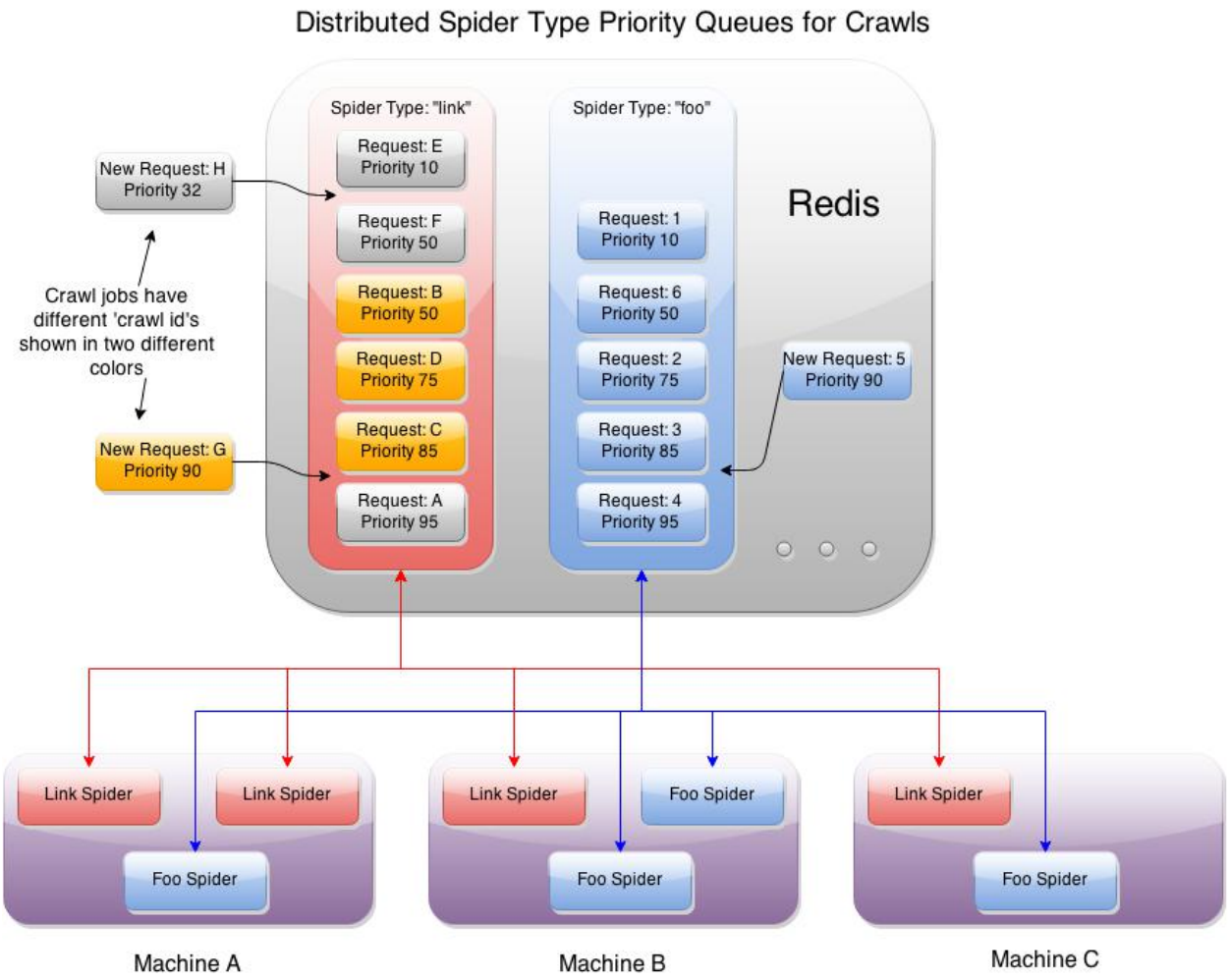
The scheduler has all of the same overridden functions you would expect, except for `has_pending_requests()`. In our tests we found inconsistencies in Scrapy when returning `True` when grabbing a new item from within that function or letting the item be returned from `next_request`. For now we return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

4.3.2 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:



Breadth first crawl with distributed duplicate filtering and decaying priority


Distributed Spider Type Priority Queues for Crawls
Spider Type: "link"
Spider Type: "foo"
Redis
Request: E
Priority 10
Request: F
Priority 50
Request: B
Priority 50
Request: D
Priority 75
Request: C
Priority 85
Request: A
Priority 95
Request: 1
Priority 10
Request: 6
Priority 50
Request: 2
Priority 75
Request: 3
Priority 85
Request: 4
Priority 95
New Request: H
Priority 32
New Request: G
Priority 90
New Request: 5
Priority 90
Crawl jobs have
different 'crawl id's
shown in two different
colors
Link Spider
Link Spider
Foo Spider
Link Spider
Foo Spider
Foo Spider
Link Spider
Foo Spider
Machine A
Machine B
Machine C

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

4.3.3 pipelines.py

The pipelines file is a basic Scrapy Item Pipeline with a single class to take the yielded item and send it back to Kafka. It also checks to make sure that the Kafka topic exists before sending the message to it.

4.3.4 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.

4.3.5 redis_queue.py

A utility class that utilizes Pickle encoding to store and retrieve arbitrary sets of data in Redis. 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 prioritized 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 pickle encoding and decoding for you.

4.3.6 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.

4.3.7 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.

4.3.8 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.

4.3.9 lxmlhtml.py

This is actually a duplicate 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.

In the future this may just be an extended class but for now it is the full copy/paste.

CHAPTER 5

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 current crawl jobs.

5.1 Quick Start

First, make sure your *settings.py* is updated with your Kafka and Redis hosts.

Then run the redis monitor to monitor your Scrapy Cluster:

```
python redis_monitor.py
```

This is a completely optional component to the Scrapy Cluster. If you do not care about getting information about crawls, stopping, or expiring crawls in the cluster then you can leave this component alone.

- For how to set up the Scrapy crawlers, refer to the Crawler documentation
- To learn more about how to submit crawls, please see the Kafka Monitor documentation

5.2 Design Considerations

The Redis Monitor is designed to act as a surgical instrument allows an application to peer into the queues and variables managed by Redis that act upon the Scrapy Cluster. It runs independently of the cluster, and interacts with the items within Redis in three distinct ways:

Information Retrieval

> All of the crawl information is contained within Redis, but stored in a way that makes it hard for a human to read. The Redis Monitor makes getting current crawl information about your application's overall crawl statistics or an individual crawl easy. If you would like to know how many urls are in a particular `crawlid` queue, or how many urls are in the overall `appid` queue then this will be useful.

Stop Crawls

> Sometimes crawls get out of control and you would like the Scrapy Cluster to immediately stop what it is doing, but without killing all of the other ongoing crawl jobs. The `crawlid` is used to identify exactly what crawl job needs to be stopped, and that `crawlid` will both be purged from the request queue, and blacklisted so the spider's know to never crawl urls with that particular `crawlid` again.

Expire Crawls

Very large and very broad crawls often can take longer than the time allotted to crawl the domain. In this case, you can set an expiration time for the crawl to automatically expire after X date. The particular crawl job will be purged from the spider's queue while allowing other crawls to continue onward.

These three use cases conducted by the Redis Monitor give a user or an application a great deal of control over their Scrapy Cluster jobs when executing various levels of crawling effort. At any one time the cluster could be doing an arbitrary number of different crawl styles, with different spiders or different crawl parameters, and the design goal of the Redis Monitor is to control the jobs through the same Kafka interface that all of the applications use.

5.3 Components

This section explains the individual files located within the redis monitor project.

5.3.1 redis_monitor.py

The Redis Monitor acts upon two different action requests, and also sends out notifications when a crawl job expires from the queues.

All requests adhere to the following three Kafka topics for input and output:

Incoming Action Request Kafka Topic:

- `demo.inbound_actions` - The topic to feed properly formatted action requests to

Outbound Action Result Kafka Topics:

- `demo.outbound_firehose` - A firehose topic of all resulting actions within the system. Any single action conducted by the Redis Monitor is guaranteed to come out this pipe.
- `demo.outbound_<appid>` - A special topic created for unique applications that submit action requests. Any application can listen to their own specific action results by listening to the the topic created under the `appid` they used to submit the request. These topics are a subset of the action firehose data and only contain the results that are applicable to the application who submitted it.

Information Action

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

Application Info Request

```
python kafka-monitor.py feed -s settings_actions.py '{"action":"info", "appid":"testapp", "uuid"
```

This returns back all information available about the `appid` in question. It is a summation of the various `crawlid` statistics.

Application Info Response

```
{
    u'server_time': 1429216294,
    u'uuid': u'someuuid',
    u'total_pending': 12,
    u'total_domains': 0,
    u'total_crawlids': 2,
    u'appid': u'testapp',
    u'crawlids': {
        u'2aaabbb': {
            u'low_priority': 29,
            u'high_priority': 29,
            u'expires': u'1429216389'
```

```
                u'total': 1
            },
            u'1aaabbb': {
                u'low_priority': 29,
                u'high_priority': 39,
                u'total': 11
            }
        }
    }
```

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 total, their high/low priority in the queue, and if they have an expiration.

Crawl ID Info Request

```
python kafka-monitor.py feed -s settings_actions.py '{"action":"info", "appid":"myapp", "uuid":"
```

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. The following example response is generated:

Crawl ID Info Response from Kafka

```
{
    u'server_time': 1429216864,
    u'crawlid': u'abc123',
    u'total_pending': 28,
    u'low_priority': 39,
    u'high_priority': 39,
    u'appid': u'testapp',
    u'uuid': u'someuuid'
}
```

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 28 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:

Stop Request

```
python kafka-monitor.py feed -s settings_actions.py '{"action":"stop", "appid":"testapp", "uuid"
```

After the request is processed, only current 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.

Stop Response from Kafka

```
{
    u'total_purged': 524,
    u'uuid': u'someuuid',
    u'spiderid': u'link',
    u'appid': u'testapp',
    u'action': u'stop',
    u'crawlid': u'ABC123'
}
```

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.

Expire Notification from Kafka

```
{
    u'total_expired': 75,
    u'crawlid': u'abcdef-1',
    u'spiderid': u'link',
    u'appid': u'testapp',
    u'action': u'expired'
}
```

This notification states that the `crawlid` of "abcdef-1" expired within the system, and that 75 pending requests were removed.

CHAPTER 6

Advanced Topics

This section describes more advanced topics about the Scrapy Cluster and other miscellaneous items.

6.1 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.

6.2 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 for Crawl Requests
- Kafka Monitor for Action Requests
- Redis Monitor

6.3 Scrapy Cluster 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

With the Kafka Monitor constantly monitoring the 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 final 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 final part is out of control of the Scrapy Cluster itself.

6.4 Redis Keys

The following keys within Redis are used by the Scrapy Cluster:

- `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>:queue` - The queue that holds all of the url requests for a spider type. Within this sorted set is any other data associated with the request to be crawled, which is stored as a Json object that is Pickle 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.

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 60 seconds, which means that if your crawl job with a unique `crawlid` goes for more than 60 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.

CHAPTER 7

License

```
The MIT License (MIT)

Copyright (c) 2015 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 8

Getting Started

Overview Learn about the Scrapy Cluster Architecture.

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

CHAPTER 9

Architectural Components

Kafka Monitor The gateway into your Scrapy Cluster.

Crawler The crawling plant behind the scenes.

Redis Monitor Provides the insight into how your crawls are doing.

CHAPTER 10

Miscellaneous

Advanced Topics Advanced items to consider when running Scrapy Cluster.

License Scrapy Cluster is licensed under the MIT License.