traptor Documentation

Release 1.0.0

Jason Haas

Jun 09, 2017

Contents

traptor is a framework to help manage your twitter data collection. What differentiates **traptor** from the many other Twitter libraries out there is that it does *real-time distributed streaming* of data based on rule sets using the Twitter Streaming API.

It uses a combination of Kafka, Redis, and the excellent birdy module. The goal is to have a convenient way to aggregate all of your twitter application data into one data stream and (optionally) a database. It uses birdy to make Twitter API connections, redis to handle the rule management among different traptor instances, and kafka to handle the data streams.

Please see http://traptor.readthedocs.org for documentation and the Quick Start guide.

CHAPTER 1

Overview

Dependencies

Components required to run a **traptor** cluster:

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

Please see `requirements.txt` for pip package dependencies.

Optional Dependencies

Depending on your implementation, you may want to consider also using these tools:

- Ansible: http://www.ansible.com/
- MySQL: https://www.mysql.com/
- ELK Stack: https://www.elastic.co/

Server provisioning

To provision your servers, it is helpful to use one of the many provisioning tools available, such as Ansible, Fabric, Chef, or Salt. Ansible and Fabric are both Python based and both work well – Fabric is preferred for simple deployments, Ansible for complex ones.

Data management

To manage your Twitter rule set, you may want to use a relational database system and parse out rules appropriately to your Redis database, which only stores key/value pairs of rules for each **traptor** type such as *track*, *follow* or *geo*. For storing the collected dataset, you may want to use one of many available NoSQL open source databases. Common choices are ElasticSearch (Lucene) and MongoDB.

In a currently deployed application, I am using a MySQL database for "user facing" rules, and the ELK stack (ElasticSeach, Logstash, Kibana) to do the data aggregation and visualization. A production level deployment data flow might look like this:

MySQL –> Redis –> Traptor –> Kafka –> Logstash –> Elasticsearch –> Kibana

CHAPTER 2

Quick Start

When first starting out with **traptor**, it is recommended that you test on your local machine to get a feel for how it works. To get a local **traptor** running, you will need to at minimum have:

- Python 2.7.x
- Redis
- Kafka (optional for testing)

You have the option of setting Redis and/or Kafka on your local machine or server, or using a pre-built vagrant machine for testing. I recommend using the Vagrant machine if you are testing on your local machine. If you are on a remote server somewhere (such as EC2), you will need to set up Redis and Kafka on that instance or somewhere else.

Traptor Test Environment

To set up a pre-canned Traptor test environment, make sure you have the latest Virtualbox + Vagrant >= 1.7.4 installed. Vagrant will automatically mount the base traptor directory to the /vagrant directory, so any code changes you make will be visible inside the VM.

Steps to launch the Vagrant VM:

1. `git clone git@github.com:istresearch/traptor` to pull down the latest code.
2. `cd traptor`
3. `vagrant up` in base **traptor** directory.
4. `vagrant ssh` to ssh into the VM.
5. `sudo su` to change to root user.
6. `supervisorctl status` to check that everything is running.
7. `cd /vagrant` to get to the **traptor** directory.

Now you will have all your dependencies installed and will be running Redis and Kafka on the default ports inside the VM.

Configuring Traptor

1. `pip install -r requirements.txt` to install Traptor dependencies.
2. `cd traptor` to get inside the module folder.
3. `cp settings.py localsettings.py` to create your `localsettings.py` file.
4. Remove the "Local Overrides" section from the `localsettings.py` file.
5. Fill in the `APIKEYS` and `TRAPTOR_TYPE` fields.
6. Optionally update the kafka and redis connection information if you are not running locally.
7. Optionally add a Redis pubsub channel if you are using pubsub to automatically refresh the rules Traptor uses.
8. Add your ruleset to Redis. This can be done any number of ways depending on where you are keeping your rules. In the in the `scripts/rule-extract.py` file there are examples of how to extract rules from a GNIP ruleset file and a MySQL database. You may wish to add a custom function to parse out rules from other sources.

Important: Be sure to insert each rule as a `hashmap` data type with the key format of `traptor-<traptor_type>:<traptor_id>:<rule_id>`.

Congratulations. You are all set to run **traptor**!

Running Traptor

To start, run it *without kafka* by running `python traptor.py --test` from the command line. The `--test` flag tells **traptor** to skip sending data to kafka and just print to stdout. You can pipe the output into jq (https://stedolan.github.io/jq/) like this `python traptor.py --test | jq .` to get a nicely colored JSON output.

traptor also accepts a `--info` or `--debug` flag if you wish to print out logging information.

Once that is working successfully, try writing your data to kafka by running `python traptor.py`. You can tail the Kafka output by running the following command in your Kafka installation directory:

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic␣
↪traptor
```

Tip: Check out kafkacat for a handy kafka debugging tool.

CHAPTER 3

Production

Running a collection of distributed Traptor streams.

Planning

To run **traptor** in a distributed environment, you'll need to figure out approximately what your collection needs are. The Twitter API offers different limits for different types of rules. As of this writing the following API limits are in place for the Public Streaming API.

- follow: 5000 rules
- track: 400 rules
- location: 25 rules

This means that you are *limited by how many rules you can add per traptor application*. For example, if you have 5,500 "follow" rules and 352 "track" rules, you will need 3 **traptor** connections (2 for "follow", 1 for "track"). These should be different API keys with different connection IP addresses.

Ansible

To handle a distributed deployment, you can use Ansible. Ansible lets you dynamically configure inventories based on roles to do semi-automated deployments.

Inventory

Using the example from above, my Ansible inventory may look something like this:

```
[traptor-follow-nodes]
server01
server02
```

```
[traptor-track-nodes]
server03

traptor-location-nodes]
server04
server05

[traptor-nodes:children]
traptor-follow-nodes
traptor-track-nodes
traptor-location-nodes
```

Group_vars

The best way to manage a pool of API keys is in a `traptor-nodes` groups_vars file. Since both `traptor-track-nodes` and `traptor-follow-nodes` are children of `traptor-nodes`, the API keys can be either by *any* traptor type. Continuing with the example above, the file might look like this:

```
---

traptor_kafka_topic: 'my_traptor'

apikeys:
  - consumer_key: 'YOUR_INFO'
    consumer_secret: 'YOUR_INFO'
    access_token: 'YOUR_INFO'
    access_token_secret: 'YOUR_INFO'
  - consumer_key: 'YOUR_INFO'
    consumer_secret: 'YOUR_INFO'
    access_token: 'YOUR_INFO'
    access_token_secret: 'YOUR_INFO'
  - consumer_key: 'YOUR_INFO'
    consumer_secret: 'YOUR_INFO'
    access_token: 'YOUR_INFO'
    access_token_secret: 'YOUR_INFO'
```

The `traptor_kafka_topic` is links to the traptor `localsettings` template to override the default `traptor` topic name with one of your choosing. The `apikeys` dictionary contains 3 sets of API connection info, one for each traptor node.

Tasks

Coming soon... how to set up Ansible tasks (link to sample code)

Redis PubSub for Automatic Rule Refresh

When your Twitter rule set changes, the Traptor to which rules have been either added or deleted can be automatically restarted. While running, Traptor continuously checks a Redis pubsub channel for a message for itself, in the following format:

<traptor-type>:<traptor-id>

An example message is:

track:0

In order to use this functionality, add a message as formatted above to the Redis pubsub channel for each Traptor for which the rules changed.

CHAPTER 4

Traptor API

class traptor.traptor.**MyBirdyClient**(*consumer_key*, *consumer_secret*, *access_token*, *access_token_secret*)

static get_json_object_hook(*data*)

class traptor.traptor.**Traptor**(*redis_conn*, *pubsub_conn*, *heartbeat_conn*, *traptor_notify_channel='traptor-notify'*, *rule_check_interval=60*, *traptor_type='track'*, *traptor_id=0*, *apikeys=None*, *kafka_enabled='true'*, *kafka_hosts='localhost:9092'*, *kafka_topic='traptor'*, *use_sentry='false'*, *sentry_url=None*, *test=False*, *enable_stats_collection='true'*)

_add_heartbeat_message_to_redis(**args*, ***kw*)
Add a heartbeat message to Redis.

_add_iso_created_at(*tweet_dict*)
Add the created_at_iso to the tweet.

Parameters **tweet_dict** – tweet in json format

Return **tweet_dict** with created_at_iso field

_check_redis_pubsub_for_restart()
Subscribe to Redis PubSub and restart if necessary.

Check the Redis PubSub channel and restart Traptor if a message for this Traptor is found.

_create_birdy_stream()
Create a birdy twitter stream. If there is a TwitterApiError it will exit with status code 3. This was done to prevent services like supervisor from automatically restart the process causing the twitter API to get locked out.

Creates `self.birdy_stream`.

_create_kafka_producer(**args*, ***kw*)
Create the Kafka producer

_create_rule_counter(*rule_id*)
Create a rule counter

Parameters **rule_id** – id of the rule to create a counter for

Returns stats_collector: StatsCollector rolling time window

_create_traptor_obj(*tweet_dict*)
Add the traptor dict and id to the tweet.

Parameters **tweet_dict** – tweet in json format

Return tweet_dict with additional traptor fields

_create_twitter_follow_stream(**args*, ***kw*)
Create a Twitter follow stream.

_create_twitter_locations_stream(**args*, ***kw*)
Create a Twitter locations stream.

_create_twitter_track_stream(**args*, ***kw*)
Create a Twitter follow stream.

_delete_rule_counters()
Stop and then delete the existing rule counters.

_enrich_tweet(*tweet*)
Enrich the tweet with additional fields, rule matching and stats collection.

Return dict enriched_data tweet dict with additional enrichments

Return dict tweet non-tweet message with no additional enrichments

_find_rule_matches(*tweet_dict*)
Find a rule match for the tweet.

This code only expects there to be one match. If there is more than one, it will use the last one it finds since the first match will be overwritten.

Parameters **tweet_dict** (*dict*) – The dictionary twitter object.

Returns a `dict` with the augmented data fields.

_gen_kafka_failure()

_gen_kafka_success()

_get_locations_traptor_rule()
Get the locations rule.

Create a dict with the single rule the locations traptor collects on.

_get_redis_rules(**args*, ***kw*)
Yields a traptor rule from redis. This function expects that the redis keys are set up like follows:

traptor-<traptor_type>:<traptor_id>:<rule_id>

For example,

traptor-follow:0:34

traptor-track:0:5

traptor-locations:0:2

For 'follow' twitter streaming, each traptor may only follow 5000 twitter ids, as per the Twitter API.

For 'track' twitter stream, each traptor may only track 400 keywords, as per the Twitter API.

For 'locations' twitter stream, each traptor may only track 25 bounding boxes, as per the Twitter API.

Returns Yields a traptor rule from redis.

_increment_limit_message_counter(**args*, ***kw*)
Increment the limit message counter

Parameters **limit_count** – the integer value from the limit message

_increment_rule_counter(**args*, ***kw*)
Increment a rule counter.

Parameters **rule_value** – the value of the rule to increment the counter for

_main_loop()
Main loop for iterating through the twitter data.

This method iterates through the birdy stream, does any pre-processing, and adds enrichments to the data. If kafka is enabled it will write to the kafka topic defined when instantiating the Traptor class.

_make_limit_message_counter()
Make a limit message counter to track the values of incoming limit messages.

_make_rule_counters()
Make the rule counters to collect stats on the rule matches.

Returns dict: rule_counters

_make_twitter_rules(*rules*)
Convert the rules from redis into a format compatible with the Twitter API.

Parameters **rules** (*list*) – The rules are expected to be a list of dictionaries that comes from redis.

Returns A `str` of twitter rules that can be loaded into the a birdy twitter stream.

_message_is_limit_message(*message*)
Check if the message is a limit message.

Parameters **message** – message to check

Returns True if yes, False if no

_message_is_tweet(*message*)
Check if the message is a tweet.

Parameters **message** – message to check

Returns True if yes, False if no

_send_enriched_data_to_kafka(**args*, ***kw*)
" Send the enriched data to Kafka

Parameters

- **tweet** – the original tweet
- **enriched_data** – the enriched data to send

_send_heartbeat_message()
Add an expiring key to Redis as a heartbeat on a timed basis.

_setup()
Set up Traptor.

Load everything up. Note that any arg here will override both default and custom settings.

_setup_birdy()
Set up a birdy twitter stream. If there is a TwitterApiError it will exit with status code 3. This was done to prevent services like supervisor from automatically restart the process causing the twitter API to get locked out.

Creates `self.birdy_conn`.

_setup_kafka()
Set up a Kafka connection.

static _tweet_time_to_iso(*tweet_time*)
Convert tweet created_at to ISO time format.

Parameters **tweet_time** – created_at date of a tweet

Returns A string of the ISO formatted time.

_wait_for_rules()
Wait for the Redis rules to appear

run()
Run method for running a traptor instance.

It sets up the logging, connections, grabs the rules from redis, and starts writing data to kafka if enabled.

traptor.traptor.**main**()
Command line interface to run a traptor instance.

Can pass it flags for debug levels and also –stdout mode, which means it will not write to kafka but stdout instread.

CHAPTER 5

Overview

Overview of Traptor and its dependencies.

CHAPTER 6

Quick Start

Get running with a local Traptor stream!

CHAPTER 7

Production

Deploying Traptor to a distributed environment.

CHAPTER 8

Traptor API

The Traptor API.

Python Module Index

t

Index

Symbols

G

M

R

T