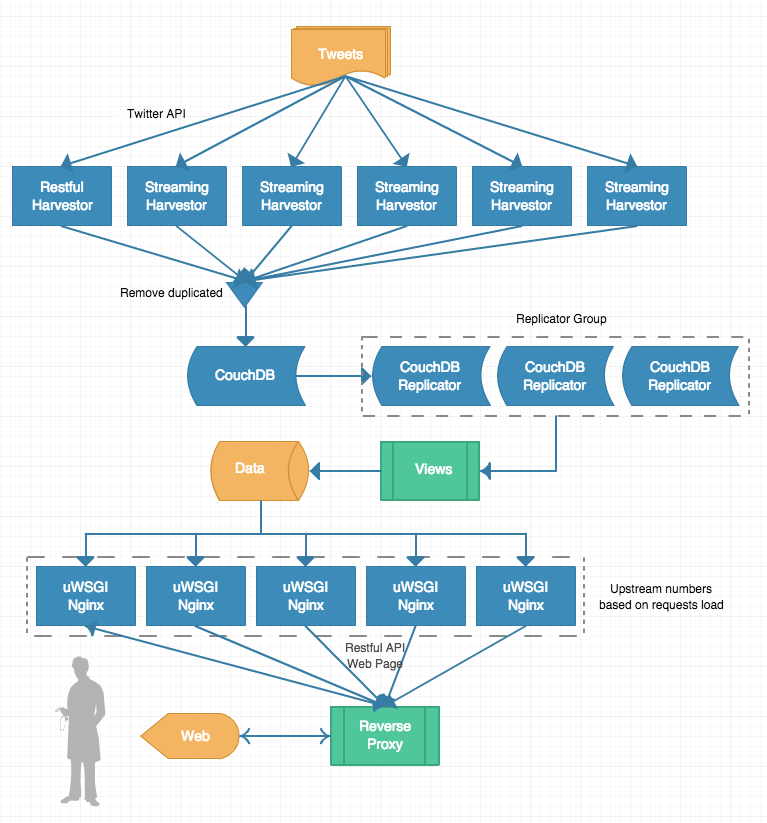
Architecture



The graph illustrate the whole system architecture of the application for global twittering.

### Data Collecting (crawler)

As we need tons of tweets to do analysis, we created two kinds of crawler to access both REST and Streaming APIs which provided by twitter. Both of them will fetch tweets continuously from twitter, and remove some non-using key-value pairs, then put them to couchdb.

REST APIs

We mainly use the search API to get tweets from Chicago, as tweets from other cities should be excluded from our responses, the requests are submitted with geocode parameter. Not like streaming api supporting bounding box parameter, for limiting the location, we specified the central latitude and longitude of Chicago with a radius of 15 miles, in that case, we may cover more areas besides the main city of Chicago.

To retrieve as many tweets as we could, our script recursively sent http request with result\_type and max\_id parameters. The result\_type is set to recent, in that case, the API returns only the most recent results in the response, to make sure we could get tweets back forward, the max\_id will be set, each time when script got 100 tweets, it will save the minimum id of these tweets, it will be used for next request as max\_id, thus, we can make sure the crawler could go back as much as the API can provide. After some experiments, we found that the API can return tweets one week before the request day at the most, in this case, the crawler will get no old tweets, the script will then fetch a newest tweets and reset max\_id, then repeat sending requests to make sure we do not miss any one of the responses.

As there are rate limits exist, we mainly handled two kinds of them. To resolve the ’15 minute window’, we only need to add adequate interval between each request, this does not commit some issue. Another limit we got is ‘Over Capacity’, this seems to be a server side limit of twitter API, they reduce the capacity of current requesting user. If we use one IP address and same app authentication for a long time, we get this faulty response, and no new tweets will respond no matter how long you wait or how many times you request, in this situation, we had to change restful crawler to another virtual machine with new IP address, the app need to be configured to a new one as well.

Streaming APIs

The Streaming APIs give developers low latency access to Twitter’s global stream of Tweet data. Connecting to the streaming API requires keeping a persistent HTTP connection open. We use streaming API to maintain the completeness of tweets, it can retrieve much more tweets than REST API.

The Chicago is divided into five bounding boxes based on geo location, then we created five apps to run on five different virtual machines. The locations parameter is provided to get correct tweets from specified area, however, according to our logging system, twitter usually pushes same tweets to our five harvesters. Those tweets usually contains no coordinates information, but the user’s profile showing the location is Chicago, this might be the design of Streaming API, its kind of compromise to the completeness as a tweet without accurate geo location, it will be defined the same place as the user’s profile.

|  | REST | Streaming |
| --- | --- | --- |
| Completeness | No | Yes |
| Accuracy | 90% | < 50% |
| Speed | Low | High |
| Persistent Connection | No | Yes |

*table: compare rest and streaming*

### Web Services

For data visualisation, a simple web server was created with flask, which is a micro web framework written by Python. It includes html templates for browser viewing as well as RESTful apis to access. All charts or maps object drew by javascript will do async request to backend restful api.

All data for visualisation is retrieved from couchdb map reduce views, we stored different views to generate desired data for various charts. For example, we add a sorting to CouchDB list functions, when the data need to be ranked, we call the sorting list function other than fetch all data from specific view and then sort them locally on web server, this saves a lot of time for transmitting data. With large amount of data, transmission load is extremely heavy when there are lot of requests, reducing this time greatly enhanced user experience.

To pick a suitable application server container, we used vegeta(HTTP load testing tool) to simulate 50 requests per second, the result is illustrated in the bar chart.

It is obvious that, pure python can not handle concurrent requests very well, as the native multithreads can not utilise all computing resources. While uWSGI has much better performance running this application. Still with one Nginx and one uWSGI, it is slightly slower than pure uWSGI, this situation would be changed once the load goes much higher and adding more upstreams to Nginx. And some articles mentioned that, the performance is better when nginx running together with multiple uWSGI master process, but we haven’t test that approach yet.

From the system architecture diagram, when request arrives, the reverse proxy server will retrieve resources on behalf of a client from servers. These resources are then returned to the clients.On every instance, server application is started by uWSGI with 1 master process and 8 worker processes, to utilise all the processing cores of server. The local Nginx on each instance could be used as reverse proxy as well as http server handling other kind of requests, for example, static files, redirection and so forth.

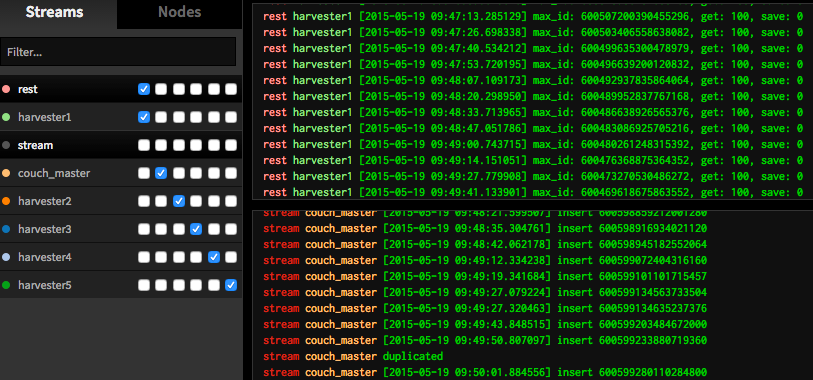
The main reverse proxy server also plays the role of load balancer, it has three ways to handle loads:

* + round-robin — requests to the application servers are distributed in a round-robin fashion
  + least-connected — next request is assigned to the server with the least number of active connections
  + ip-hash — a hash-function is used to determine what server should be selected for the next request (based on the client’s IP address)

This server model should enhance the performance very well and give a better fault tolerance.

### Error Handling

realtime log monitor system

The crawler are deployed to six virtual machines on Nectar, although we can use boto and ansible to secure all the environment be installed easily, we need to monitor all the crawlers to make sure no error blocks them. However, logging into each instance and check the processes and logs is kind of wasting extra time, we use log.io to monitor all instances via web interface, all logs on different instances will be streamed to web browser, six machines at one time. With log.io, we could identify if the crawler is running correctly or not.

*realtime* *logging*

removal of duplication

As twitter may return or response duplicated tweets to all the crawlers, we need an approach to identify them and make sure no redundant tweets are saved to CouchDB, this is one important key to save storage spaces and improve the speed of retrieving data, as well as indexing when requesting views.

The solution to this part is quite simple, each tweets from twitter API has an unique id and id\_str, to make sure CouchDB does not lose precision of number, we use id\_str to replace the build-in \_id key. Each document in CouchDB has same definition of \_id and twitter’s id\_str, in this way, when a new tweet returns, it can be identified if it is already existed in CouchDB with extremely low time complexity, as \_id could be used for searching and indexing.

fault tolerance

The whole system has three layers of fault tolerance.

* + Crawler level: as we used both REST and Streaming APIs, once one node of Streaming crawler crashes or stops, the rest crawler can still get those tweets from those area, as REST crawler is querying recursively. Same mechanism is designed for REST crawler, no tweet is missed with both crawlers.
  + Database level: although we have only one CouchDB master server in this implement, the crawler write to two different databases, one for data processing, the other for data backup. Moreover, with setting continuous replication, those replicators also store one copy of data, basically, there is no chance all nodes are dead, so safety of data could be guaranteed.
  + Web level: Nginx could be configured with multiple upstreams, for example, if we got five instances running web services, the main reverse proxy server have five different options to pass clients’ requests. Still, to make sure the upstream is live, a health check module must be configured to poll status of all instances, no request will be passed to a blocking instance. Still, uWSGI launches several worker processes, the request could be respond even if some of them are blocked. Generally, we have enough services providers, it can be calculated by [num\_of\_instances \* num\_of\_workers \* num\_of\_threads].

### References

1. <https://dev.twitter.com/overview/documentation>
2. <http://flask.pocoo.org/>
3. <https://flask-restless.readthedocs.org/en/latest/>
4. <https://uwsgi-docs.readthedocs.org/en/latest/>
5. <http://nginx.org/en/>