GeoTweet: Realtime Geolocation Analyzer for Tweets

Weijie Huo (whuo3), Shuo-Yang Wang (swang234), Jihui Yang (jyang75)

Abstract:

Social media has been the fastest growing industry since the beginning of this century. Social networking websites like facebook and twitter has more than one billion users and they share around fifteen billion new posts and tweets each month [1]. How to make sense of these vast amount of data is a valuable question that demands good solutions. However, every few services provides real-time twitter streaming based on specific geolocation. The convenience of real-time streaming geo-tagged tweets can be utilized by travel websites, finance companies and news agencies in analyzing data. We hence introduce GeoTweet, a geolocation-based tweet crawler that processes and analyzes streaming tweets. Our goal is to build a web service that analysis geolocation-based tweets in real-time streaming manner.

1. Introduction:

The social networking market has gone viral in the last decade and successfully utilizing data from social media can help analyze market and business [2]. There are similar work done on analyzing geolocations of streaming tweets. World Seer [3] is a real-time tweet photo mapping system that get pictures posted on twitter with their locations, however it did not provide any analytical data but instead merely displays the image on different geolocation. Geo-tagged Twitter collection [4] shows a heat map and tweet count in a specific predefined area, the analysis can show how popular twitter is among people at a location, however it did not analyze the contents inside the tweets. This paper [5] provides methods on analyze geolocation from tweet content from human-readable high-level information, however there should be more data to retrieve out of tweets than just the geolocation. Learning from similar work done in this field, we evaluate the advantages and disadvantages to better our product. Therefore, we present our product: GeoTweet, which is a web-service for real-time streaming tweets from Twitter. GeoTweet's goal is to not only deliver the tweet itself but also provide with sufficient information about the geolocation and analytical data from the tweet content. The features of GeoTweet include:

1. Display real-time tweets based on geolocation on a map.
2. Do hashtag ranking on tweets and present easy to understand information to users about hot topics.
3. Able to present hot events in different locations based on local hashtag rankings.

To process twitter data in real time, GeoTweet uses apache storm as streaming processing tool to analyze large chunk of data from twitter, and displays results of these tweets on a map according to their geolocation. We are unique from previous services and works in a way that we provide real time streaming data analysis based on its geolocation. The trending hashtag rankings includes global ranking as well as rankings for selected area. Companies that operates on

travelling, financing and news are in need for top trending data when they want to explore a local area. And that is exactly what we provide through GeoTweet. Our goal is to bring a new way to make people explore real-time local events, and let people find out the hottest events occurred in a specific area. People can quickly find out representative events in specific locations, view the hottest hashtags by using our system. With this system, people no longer need to manually search for the hottest events occurred in a specific area, which can be time consuming and the results are usually not real-time events. In conclusion, GeoTweet can guide users to interesting events around an area and explore new experiences.

In Section 2, we will go into depth about the implementation and architecture of our system including tools we evaluated and algorithms we selected. In Section 3, we will show experiment results and evaluate performance of different parts in our system. In Section 4, we will conclusion our strengths and talk about future work. Last but not the least, in section 5 we will present our business plan and expectation for our product in the next couple of years.

2. Implementation, Techniques

In this section, we will go through the implementation of GeoTweet from backend to frontend. We will introduce the architecture and algorithm used for implementing GeoTweet and how does that fit our needs.

2.1 Website display:

GeoTweet conveys several information to users, which includes:

1. Display real-time tweets on a map based on their geolocation
2. Display global hashtag ranking
3. Display local hashtag rankings in different locations based on tweets' geolocation

Clearly present all these information to users is not an easy task. There are some constraint that we need to take into account. First, the current frontend technology restricts us from doing courageous design, we needed to stick to the design a frontend framework provides. Second, the restriction on devices. As a user can use GeoTweet on different devices, it is necessary to make sure that all different devices in different resolutions can display the website correctly. Third, as a team of developers with restricted time to develop an easy-to-use website, we do not want to reinvent the wheel all the time. It is possible for us to create a website displaying information in a creative way. However, without careful experiments, we do not know if the new design would be suitable to our case or not, and if we did that, we already wasted bunch of time on doing so. Therefore, we determined to use some frontend frameworks and sticked to the design the frameworks provide. With the help of such frameworks, we can quickly develop the website with certain quality. Here we use Angular 2 as the whole frontend framework and Bootstrap as the CSS framework.

Figure 0a. Website snapshot of global view

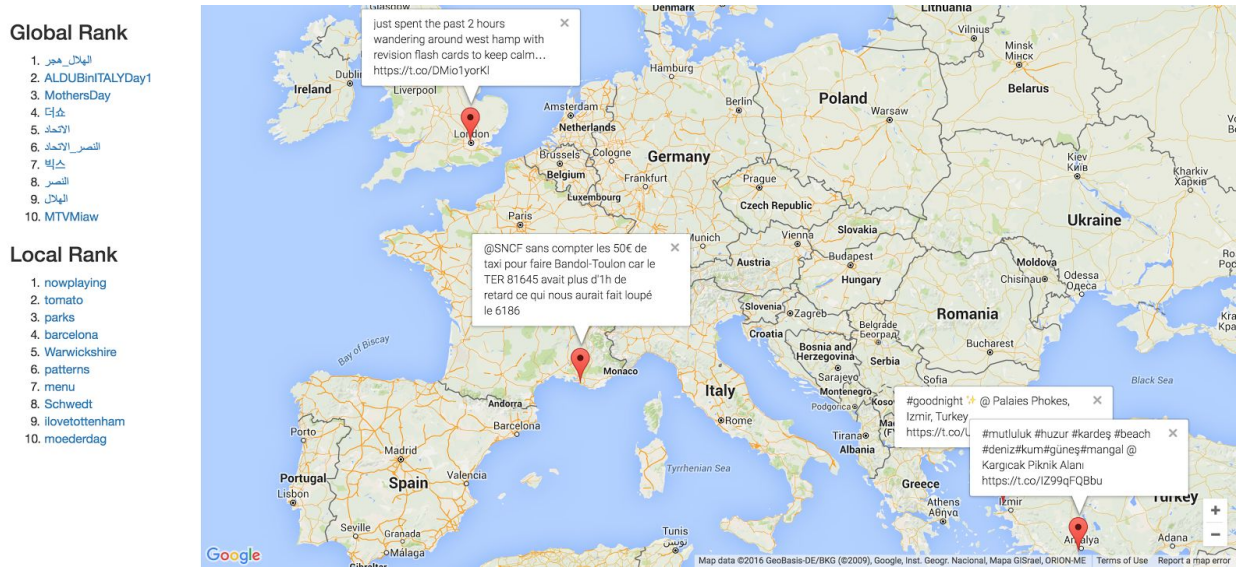

Figure 0b. Website snapshot of local view

Figure 0a and Figure 0b show two snapshots of the website. The left side of the website constitutes two lists, which are a global rank list and a local rank list, and the right side of the website is a map showing real-time tweets based on their geo-location. The global rank list represents the global hashtag ranking of all tweets, and the local rank list represents the local ranking of the tweets whose geo-locations are within the range of the map. For instance, Figure 0b shows the map view of Europe, and the local rank on the left side represents the hashtag ranking of all the tweets within the map. In addition, if users are interested in any hashtag on the list, users can click on the hashtag to open the corresponding hashtag web page on Twitter. Each tweet on the map will last for ten seconds so that the map can continuously update the newest tweet.

2.2 Architecture of the system:

The high-level architecture of the system is shown as below, from the frontend to backend servers. The frontend consists of "Load Balancer" and "Web Servers". The backend consists of "Redis Database", "Apache Storm":

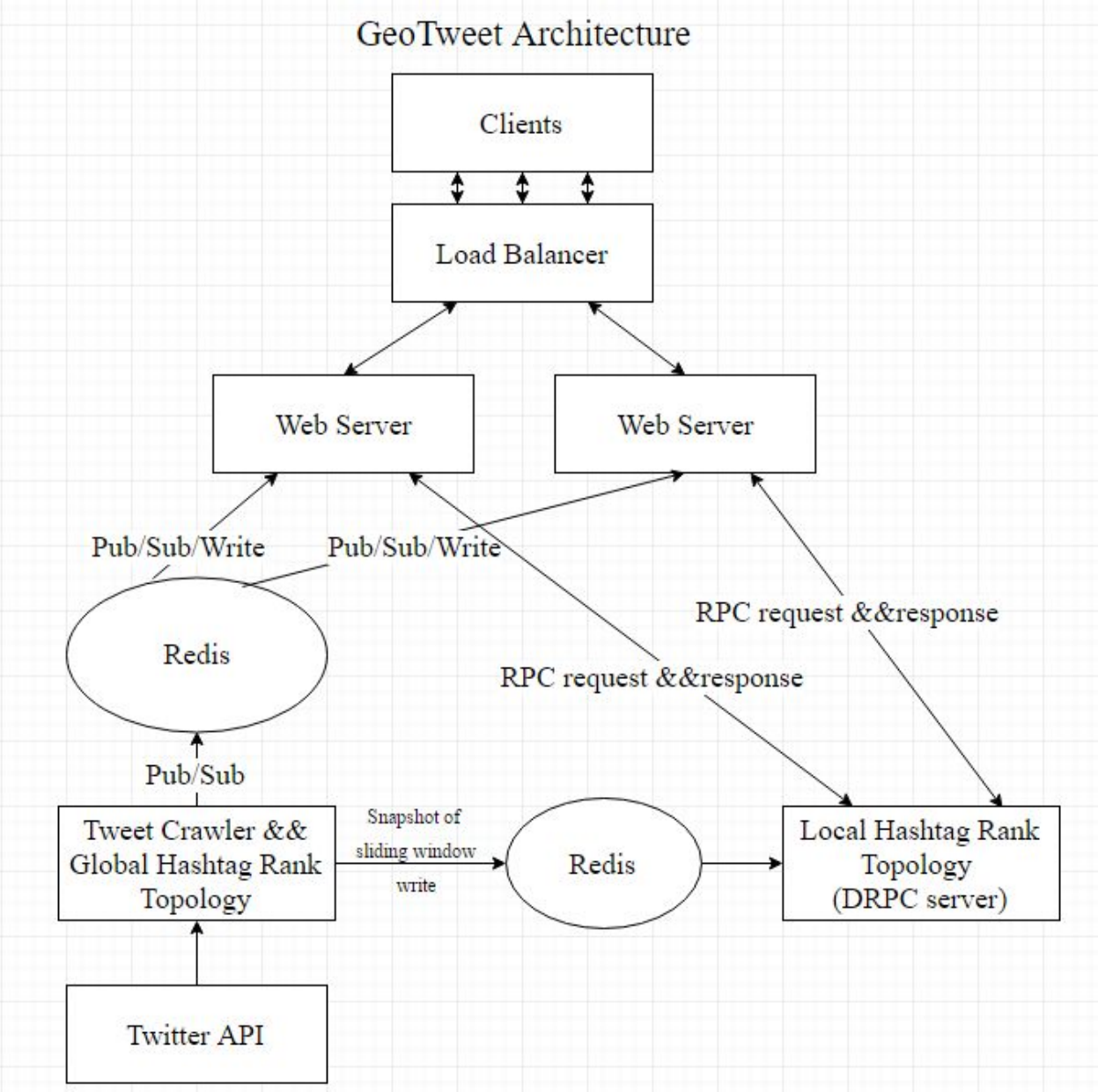

Figure 1. Flowchart of GeoTweet Architecture

Our system is running on 10 machines. We assigned 3 machines running Apache storm cluster, 3 machines running BSD Redis database cluster, 3 machines running Web Servers

services and 1 machine for Load Balancer. At the backend, servers running Storm will constantly crawl tweets from Twitter API. After processing those tweets Storm will write them to Redis database. At the frontend, users will access our web service through Load balancer, which distributed load equally among Web servers. Web servers will read data including tweets, hashtags and rankings directly from Redis database.

2.3 Storm topology:

Our Storm topology consists of three different parts of functionalities, they are:

1. Get tweets from twitter API and parse their latitude and longitude.
2. Get hashtag from all tweets and count global hashtag ranking.
3. Based on user remote procedure call, get local hashtag ranking for specific area.

Part one and two are combined and run as one topology job, part three is another topology scheme that run separately and awaits for request from user side.

2.3.1 Tweet crawler and Global hashtag rank

For this topology, tweets streaming comes in real-time from the Sprout and passed down to lower-level Bolts. As shown in Figure 2, the top half of Storm topology gets all tweets and filter all the tweets with hashtags, the bottom half of the Storm topology filters all tweets with geolocation data and stores them into Redis database. We will explain on how each one of the two system works.

1. Tweets with geolocation:

   Tweets flow in from Twitter API at Sprout, the Tweets are then sent to GeoParseBolt by shuffle grouping. At GeoParseBolt, it will parse the tweets and get all tweets with geolocation data, which is their latitude and longitude. At the end of GeoParseBolt it will send each tweet with its geolocation info to store in Redis database in the format *("latitude", "longitude", "tweet").* Tweets stored in Redis are then forwarded to Web server and displayed to user by publishing to subscriber (covered later in Section 2.6 Pub/Sub).

2. Global hashtag tanking:

   Tweets flow in from Twitter API at Sprout, the Tweets are then sent to GeoHashBolt by shuffle grouping. It will filter all the tweets with hashtags and forward them to RollingCountBolt by field grouping. RollingCountBolt will then apply sliding window to get these hashtags (covered later in Section 2.4 Sliding Windows for Hashtags). At this stage, RollingCountBolt will store hashtag it currently processes to Redis Server for later use by local hashtag rank. The format it is stored in *("latitude", "longitude", "hashtag").* Then RollingCountBolt forwards the hashtag to ImmediateRankBolt by field grouping. ImmediateRankBolt will combine hashtag with the same name by increasing its count, then it will sort it and get a rank of hashtags on each local bolt. Then all ImmediateRankBolt will forward their own rank to TotalRankBolt, which will combine all individual rank and sort the top 10 rank for global hashtag rank. The result will be sent to Web server in the format *(hashtag1&&hashtag2&&.....hashtag10&&)* and displayed to users.

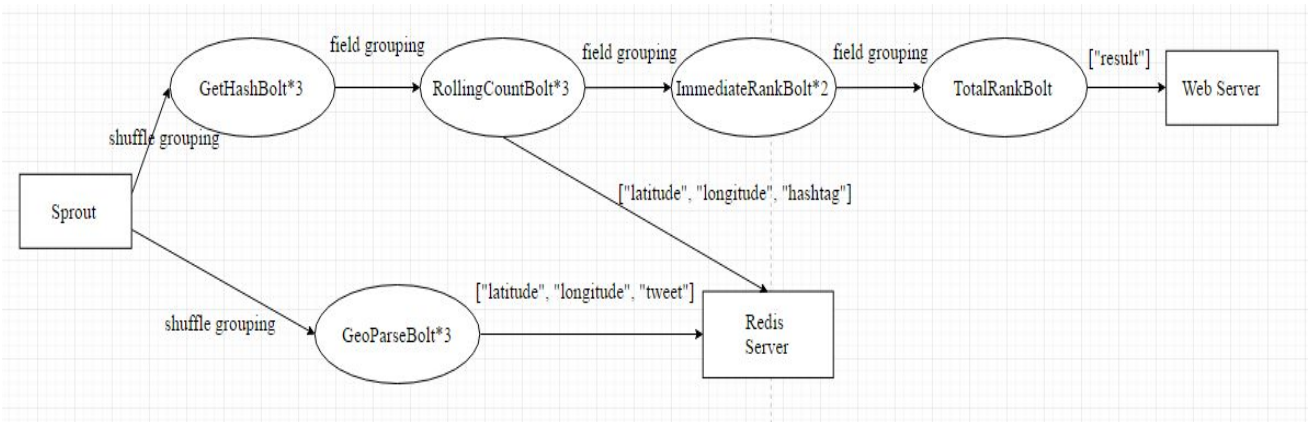

Figure 2. Tweet parse and global hashtag topology

2.3.2 Local hashtag ranking based on DRPC:

GeoTweet is able to display hashtag ranking based based on current location. The use of distributed remote procedure call(DRPC)[8] allows storm topology to process different request for different client by generate a distinct id for each new request. Hashtag rankings are provided from the sliding window for tweet content and local range based on latitude and longitude ranges. As shown in Figure 3, whenever frontend browser detects user zoom in or out, our node.js will send a request to DRPC server on storm with an argument in the format of *(west_bound, east_bound, north_bound, south_bound).* This format corresponds to the latitude and longitude ranges for local hashtag rank. The topology implementing that function uses a DRPCSprout to receive a function invocation stream from the DRPC server. Each function invocation is tagged with a unique id by the DRPC server and forward towards the topology. Inside the topology, GeoTweet filters tweets based on latitude and longitude ranges and counts occurrence of hashtags that bounds in the local area in UpdataBolt. The output is then forwarded into ImmediateRankBot where it sorts hashtags based on their count. The outputs from each ImmediateRankBolt are then combined inside TotalRankBolt and connects to the DRPC server with the result for the function invocation id. The result contains rank of top ten local hashtag ranking. The format for it is *(hashtag1&&hashtag2&&.....hashtag10&&)*, in which the *&&* symbols separates each hashtag and they rank 1 to 10 from left to right. The DRPC server then uses the id to match up that result with which client is waiting, unblocks the waiting client, and sends it the result.

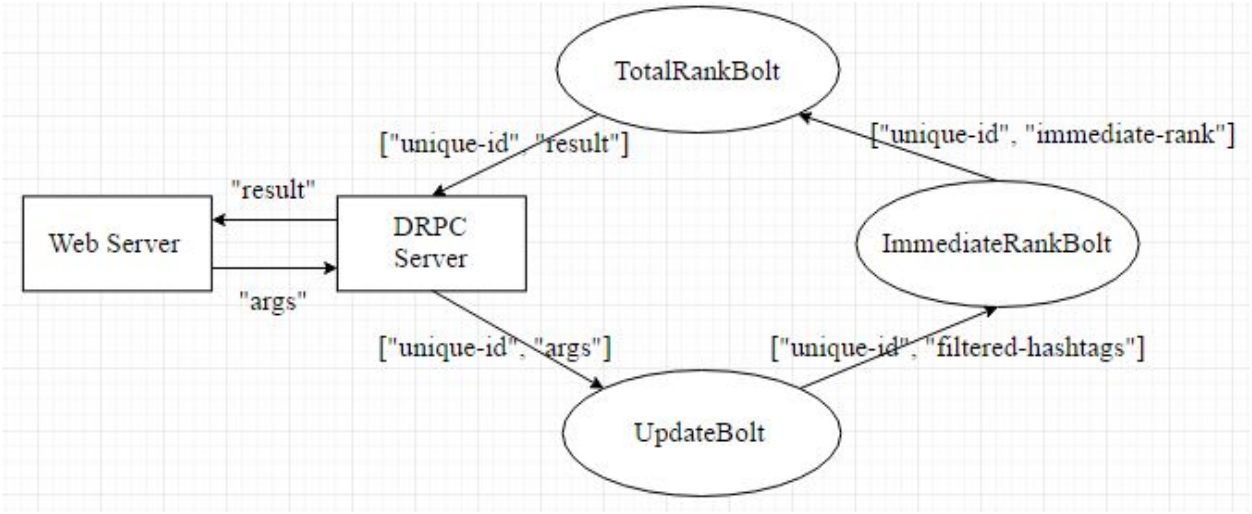

Figure 3. Local hashtag rank DRPC topology

2.4 Sliding Windows for Hashtags (Synchronization):

To display hashtags stream on the map and analyze it in real time, we need to be able to ensure the time gap between hashtags being posted and hashtags being displayed as short as possible. Therefore, we use Apache Storm to parse hashtags and NoSQL database Redis to store the tweet data.

1. We need to process the unlimited tweet data from Twitter API and Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing.
2. We compare the difference between Cassandra and Redis (More detail are covered below). As Redis support disk-backed in-memory database and it is good for rapidly changing data with a foreseeable database size (should fit mostly in memory).
3. We only want to show the new and available tweet data on the map. As sliding window advances, the slice of its input data changes. Therefore, we maintain a sliding window protocol to avoid web server reading the same data from database more than once.

There are about 2 million public geotagged hashtags every day, which is about 24 per second. Our topology in Apache Storm will use the twitter api to load, parse hashtags and store hashtags as key value pair in the Redis database. We only store the latest hashtags on the Redis cluster. To do that, every time storm write a key value pair on Redis, we set the key value pair TTL (time to live) to be 5 seconds(the database system will flush it 5 seconds after it was set). As a result, all the tweet data on the Redis are the latest (in 5 seconds). On the other hand, we need to avoid web server reading duplicated data from Redis server, web server will send a read request to Redis 5 seconds after it get the response from Redis. The proof can easily be seen in Figure 4 (Node that we only need to ensure the time between two read in database is larger than 5 seconds).

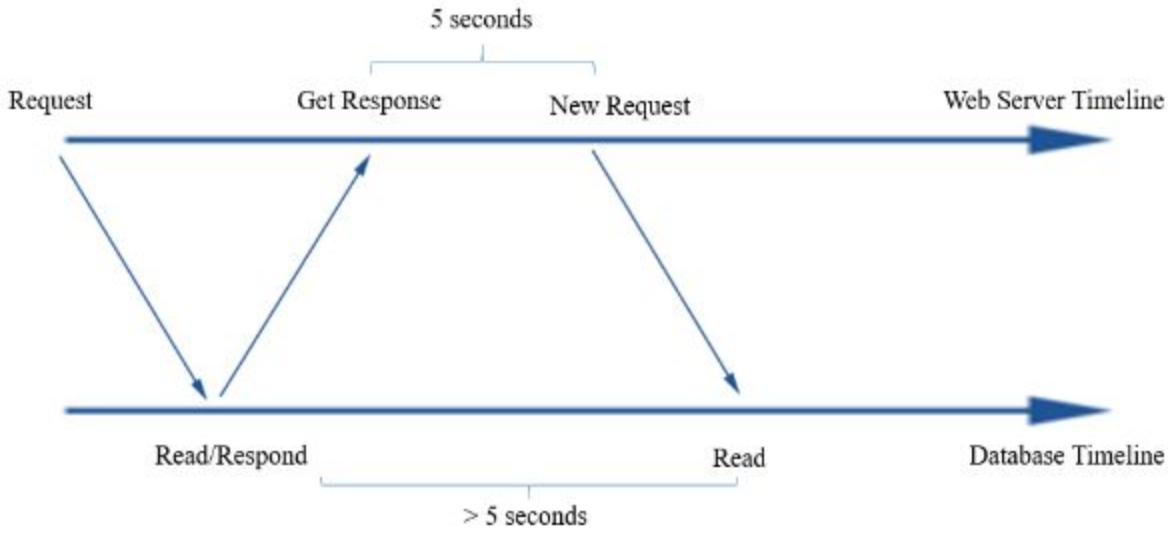

Figure 4. Request/response between Web server and Redis

2.5 Load Balancer:

In setting up our web servers, we pay a great amount of attention on the quality of service. As our web service launches, we need to be able to handle large incoming traffic on our web servers and provide seamless service to clients as if accessing one server that is available 24/7. Load balancing allows organizations to distribute inbound traffic across multiple back-end destinations. It is therefore a necessity to have the concept of a collection of back-end destinations. Clusters, as we will refer to them herein, although also known as pools or farms, are collections of similar services available on any number of hosts [6]. The basic load balancing transaction is as follows:

1. The client attempts to connect to the service on the load balancer
2. The load balancer accepts the connection. It then decides which host service should receive this connection, changes the IP destination to match the service of the selected host while keeping a record of the source IP from the client.
3. The host selected accepts the connection and responds directly to the client, via the load balancer as its default route.
4. The load balancer intercepts the return packet from the host and modifies the source IP to match the virtual server IP and port, and forwards the packet back to the client.
5. The client receives the return packet, believing it comes from the virtual server and continues the process.

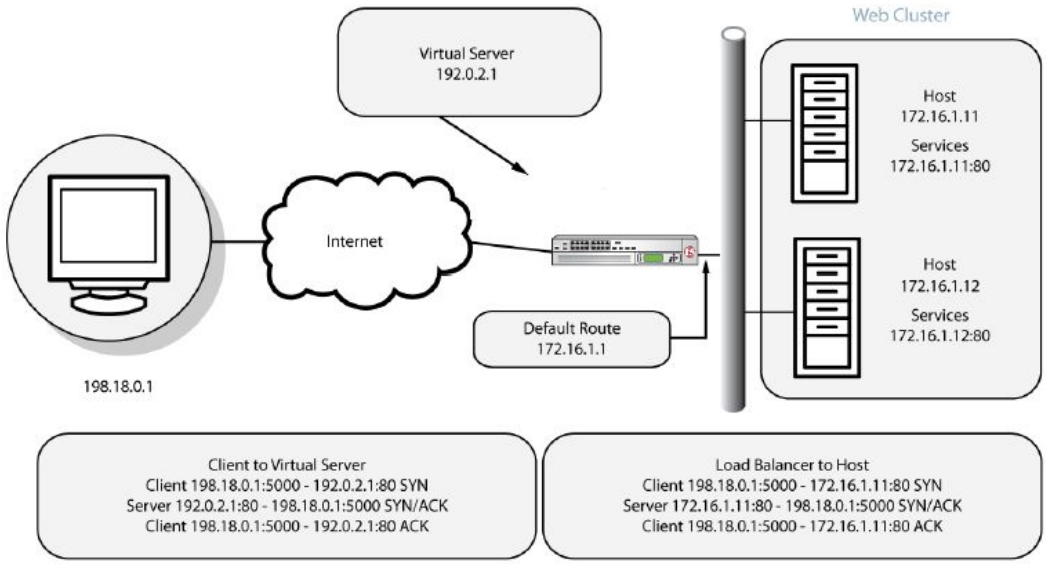

Figure 5. Load Balancing Transaction [6]

In setting up our backend web servers, we set both of these servers to serve exactly the same content. They are configured to only accept web connections only from the load balancer so that web servers delivers full throughput to clients through load balancer. Using a load balancer allows us to connect multiple web servers and hence avoid single point failures. But in our model there exists another single point failure at the load balancer's side. Since clients can only access load balancer to access our web service, if load balancer malfunctions single point failure still exists. To solve this problem, a generally recognized solution is to use two load balancers with Keepalived daemon and Floating IPs [7]. This additional implementation remains to be future work for our project.

2.6 Pub/Sub:

We chose Node.js for the website backend. The reason why we chose Node.js is that the community of Node.js is mature and Node.js is actually Javascript so that we can easily do the backend development without learning a new language.

The website backend is the bridge between frontend and our distributed database, which is a Redis cluster. At first, we thought that the frontend web client should actively query data from the backend server, and check if there is a new tweet from the Redis cluster or not. However, after digging into the official documentation of Redis, we found that Redis supports Publish/Subscribe (Pub/Sub) model. Pub/Sub model makes the data transfer in a reverse way, which means that now data can be actively transferred from Redis server to Node.js web server, and the Node.js server just need to wait for the data sending from the Redis cluster. As a result, by using Socket.io to develop the communication between the web client and the web server, now the frontend web client does not need to actively send request to the Node.js server, the frontend web client can jsut passively wait for the Node.js server to send data to it. Figure 6a

shows the Request/Response model which we are not using, and Figure 6b shows the Pub/Sub model we used in our system.

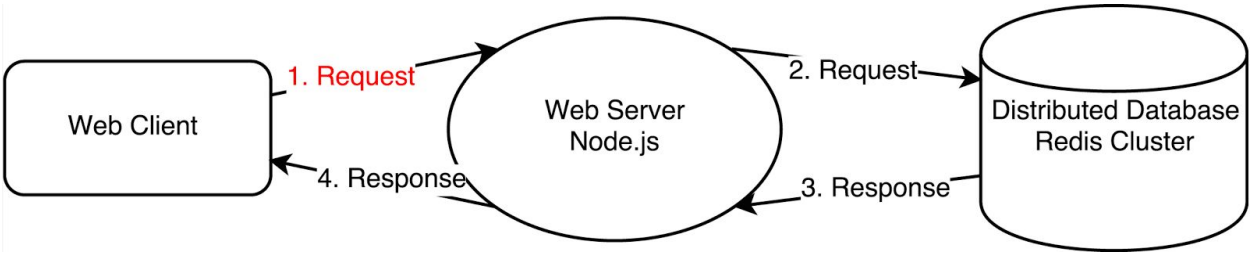

Figure 6a. Request/Response model

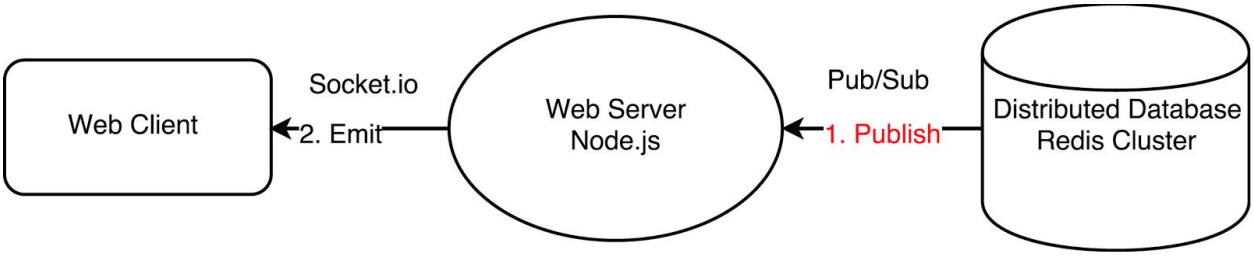

Figure 6b. Publish/Subscribe model

The reason why we chose Pub/Sub model for the system is that the whole application is much less complicated with Pub/Sub model. We have not fully evaluated and compared the performance between the two models, but if we chose the Request/Response model for the system, the frontend web client needs to make requests to the web server periodically. With some quick investigations, we learned that there will have lots of unnecessary requests under the case that frontend web client requests more frequently than the Redis cluster updates. In addition, the web server and the Redis cluster will be overloaded if there are a huge amount of web clients. Moreover, the simplicity to program using Pub/Sub model also attracted us to choose the Pub/Sub model for our system. Note that here we ignore the load balancer in Figure 6 to simplify the figure.

3. Tradeoff, Experiments:

3.1 Redis vs Cassandra

Redis: really fast compared to Cassandra as it has disk-backed in memory database. It satisfies Consistency and Partition Tolerance in the CAP problem. It is great to use when you have the rapid changing data and have approx. data size estimate (which can fit in memory).
Cassandra: an Apache product and has support for HiveQL (SQL like syntax. It satisfies Availability and Partition Tolerance in the CAP problem. It performs well when we are writing more than we are reading.
We need to choose a database system that dynamically stores data processed by storm and ready to be retrieved by our web servers. Results tested on 3 nodes model:

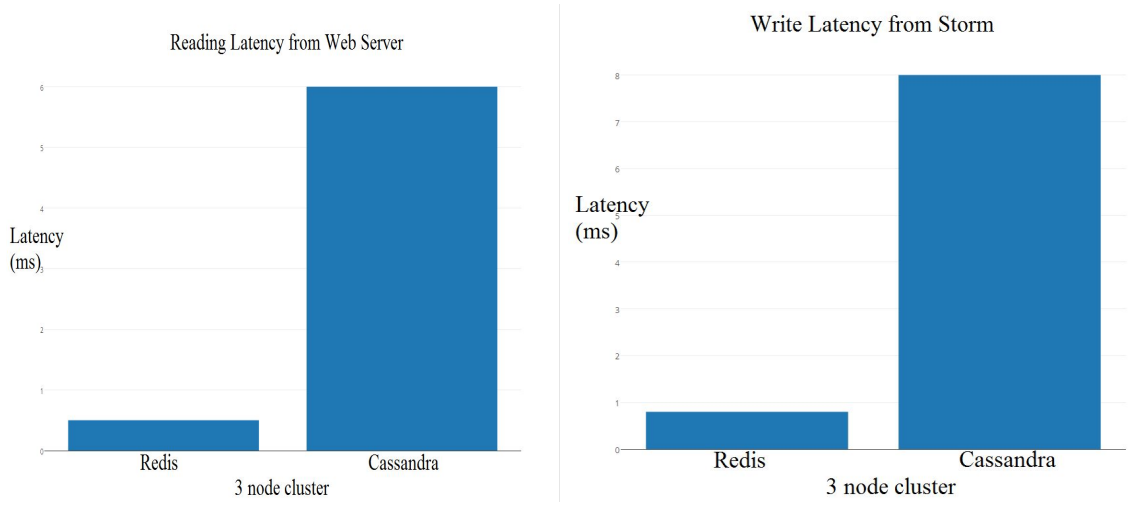

Figure 7. Redis, Cassandra latency comparison

3.2 Apache Storm real time processing vs Apache Spark real time processing

Storm is a very scalable, fast, fault-tolerant open source system for distributed computation, with a special focus on stream processing. Storm excels at event processing and incremental computation, calculating rolling metrics in real time over streams of data. Spark has more of a "functional" flavor, where working with the API is driven more by chaining successive method calls to invoke primitive operations, while storm tends to be driven by creating classes and implementing interfaces. Our product are primarily focused on stream processing and CEP-style processing. Therefore, we choose storm instead of Spark to implement it.

3.3 Why Node.js?

Node.js: Why we choose Node.js for back-end development is because that Node.js is actually written in Javascript, which would also be used in frontend development, and we do not want to spend additional time on learning a new language for back-end development. Another reason using Node.js for back-end development is that Node.js is an event driven language, which meets the needs of a server pretty well. Whenever a request arrives, the server can process the request accordingly, and because Node.js would handle each connection concurrently, the server can handle multiple requests at a time easily.

3.4 Why Angular 2?

Angular 2: When doing frontend web development, it would be a nightmare to just use plain HTML, CSS, and Javascript. All three components can mess up with each others. In other words, there is less boundary between these three components, and it would be hard to maintain if not using a suitable framework. Therefore, we use one of the famous frameworks for frontend development, that is, Angular 2 with typescript. Angular 2 is a MVC framework from Google that clearly distinguishes model, view, and controller in frontend development. Typescript is a Javascript transpiler from Microsoft that declares strict types for variables, which can make

developers detect type errors easily. With the selected frontend framework, we believe that we can create a robust web app with high maintainability efficiently.

3.5 Topology workload

There are about 500 million tweets posted every day. We are using 3 machine running Redis servers. Each machine has about 3GB RAM available. Each tweets extracted from Twitter API contains the content and additional overhead for processing which at most occupy about 2KB per tweet. In addition, we also store global hashtag and their count, each occupy at around 200B. By calculation, there are about 28935 tweets per 5 seconds (We only keep the latest 5 seconds tweets on Redis) and 9914 global hashtag. 29000 * 2KB + 10000 * 200B = 60 mb (on RAM). Figure 8 shows the workload throughout 24 hours a day.

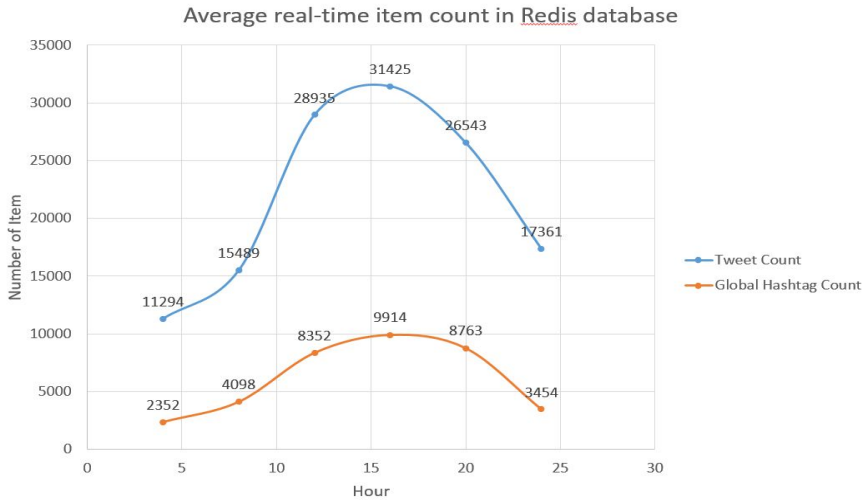

Figure 8. 24 hour workload from Storm to Redis

Due to Twitter security policies, we can only access a small portion sample of all tweets. Our current settings satisfy the current workload. If we have access to all the tweets we can exploit more machines to scale out and handle expanding workload. Later, we will deploy more feature and functions of our web services, we may add more machines to the cluster. But at this time, It is sufficient to use 3 machines for our Redis servers.

Business Plan Updated

Our market targets young users aging from 15 to 35, who are active internet social network users. According to recent report, this group of customers represents a stable ongoingly growing market. A conservative estimate predicts that this market will reach 300 million arrivals by 2020 and represent US$320 billion in market value. We want to be the pioneer who firstly exploits this growing market.

We plan to launch our product in May. Our initial launch will only provide service to champaign/urbana district because it is the neighborhood we are familiar about and we can get more feedback from fellow students. In the third quarter, we will gradually expand our product by improving it based on our feedback and launch in new cities. We plan to gradually launch in the US to cover all cities within a two year range.

We will promote our product via social networks and search engines. We can make our website shown among the top when user searches for “local events”, “travelling” or any relative keywords. Promoting on social network is also a good way because we use social network (twitter) as our data feed. And our targeted users are also those who uses social networks in their everyday life, these people are more likely to try out new things thus potential users for our project.

The main profit of our product comes from advertisement. Our advertisement pattern is similar to google. We will display our sponsors’ ads and links attached to the related tweets showing on the map. The profits from advertisements should be able to sustain our product. As it grows, we will cooperate with other websites to promote business. For example, if a user is interested in a hashtag about a local concert we will direct user to websites that can book shows and concerts.

We expect our product to take shape in around 3 years and have a relative stable user base. By that time our product will have quite some influence and reputation amongst travelling and local events planning providers. The next best opinion for our development probably is to get acquired by other big company with a large user database. Because a main disadvantage of our product is that we only provide service but has very few user engagement. This means that users only use our product without creating an account to become stable user. Companies with a large user database can easily utilize our product. Vise versa our product will also become more popular if users from those companies are willing to use our product. So we will communicate with companies like facebook and twitter to talk about acquisition, get paid and become millionaires.

Reference

1. Leading global social networks 2016 | Statistic "Leading social networks worldwide as of April 2016, ranked by number of active users (in millions)" *http://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/*
2. Can Tweets And Facebook Posts Predict Stock Behavior? *http://www.investopedia.com/articles/markets/031814/can-tweets-and-facebook-posts-predict-stock-behavior-and-rt-if-you-think-so.asp*
3. Hidetoshi Kawakubo , Keiji Yanai, GeoVisualRank: a ranking method of geotagged imagesconsidering visual similarity and geo-location proximity, *Proceedings of the 20th international conference companion on World wide web, March 28-April 01, 2011, Hyderabad, India*
4. Geo-tagged Twitter collection and visualization system. Hideyuki Fujita. *Cartography and Geographic Information Science. Vol. 40, Iss. 3, 2013*
5. The where in the tweet. Wen Li, Pavel Serdyukov. *Proceedings of the 20th ACM international conference on Information and knowledge management. Pages 2473-2476*
6. Load Balancing 101: Nuts and Bolts. *by KJ (Ken) Salchow, Jr Sr. Manager, Technical Marketing and Syndication. https://f5.com/resources/white-papers/load-balancing-101-nuts-and-bolts*
7. How To Set Up Highly Available HAProxy Servers with Keepalived and Floating IPs on Ubuntu 14.04. *Justin Ellingwood. Oct 23, 2015 Load Balancing, High Availability Ubuntu*
8. Distributed RPC. *Apache Storm Version 1.0.1 documentation. http://storm.apache.org/releases/current/Distributed-RPC.html*