CS237 PROJECT REPORT

A Platform Design and Implementation for Real-time Visitor Flow Visualization

Yuan Zhang, ID: 91123766
Haikang Chen, ID: 15002417
Shiye Yan, ID: 91728643

I. Motivation and Goals

Visitor flow visualization provides intuitionistic view of the position and moving trend of crowds of people in real-time on the map. It is a useful tool in tourism, transportation, and security areas. Especially in today's context, when horrible infectious diseases such as COVID-19 spread around the world, keeping a distance with a dense crowd according to a real-time reference is increasingly important.

The data source of such application can be collected anonymously from mobile signaling data, which contains a rough physical location of each user, then data processing and analysis is performed to generate data for visualization[1]. Comparing with other methods like collecting data from GPS, such method provides more stable data stream with satisfying precision.

Data visualization is the graphic representation of data. It involves producing images that communicate relationships among the represented data to viewers of the images. This communication is achieved using a systematic mapping between graphic marks and data values in the creation of the visualization. Real-time data visualization performs dynamic charts and graphs to the user based a persistent data stream, which gives the user an intuitive understanding of the on-going changes in the data, because graphic displays are often very effective at communicating information[2].

The main technique lies behind the application is real-time data monitoring and analysis, which is a popular topic in the era of big data and mobile Internet. Traditional analytics is based on offline analysis of historical data. whereas real-time analytics involves comparing current events with historical patterns in real time, to detect problems or opportunities. The input of the system is usually unbounded streams of data, and thus the main challenge is to reliably process the data stream in a relatively small delay, as well as do real-time processing. The system should be reliable, fault-tolerant, and scalable.

Apache Storm is a distributed real-time computation system which helps reliably process unbounded streams of data. It is widely used in real-time analytics, online machine learning, continuous computation, etc. It is fast, fault-tolerant, and scalable. An Apache Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.

In this project, we try to design and implement a visitor flow visualization application mainly based on Storm, which consumes a persistent data stream containing the location of visitors, and performs real-time computation and analytics to support the visualization of position and moving trend of crowds of people in the form of combining with real-world map on a web page.

II. System Design

a. Scenario Modeling

The first problem we need to solve is how to model the scenario. After reading some materials, we decide to introduce a concept in telecommunications called signaling. In telecommunication, signaling is the use of signals for controlling communications. This may

constitute an information exchange concerning the establishment and control of a telecommunication circuit and the management of the network, in contrast to manual setup of circuits by users or administrators[3] .

With this concept, the scenario we choose is simulating users use mobile networks to communicate with the base station. Our system collects the real-time logs from the base stations, tries to do the statistics and analysis works according to the real-time logs.

In our simulation model, we develop a pseudo data source to generate real-time log messages. The messages contain the following fields. (1) Phone number, which is also an identification of one person. (2) Base station location, which contains both the value of its latitude and longitude. The value of the location also acts as our standard of visitor flow statistics. (3) Time, which is in the format as "year – month – day hour: minute: second". (4) Latency, which means the latency in the communications between the users' mobile phones and the base stations.

b. System Architecture

After modeling the scenario, we need to design a system to support our calculation. Firstly, for collecting the real-time logs, we decide to use Logstash and Apache Kafka.

We use Logstash to help us collect the real-time logs from the data source. Logstash is an open source data collection engine with real-time pipelining capabilities. Logstash can dynamically unify data from disparate sources and normalize the data into destinations of user's choice. Cleanse and democratize all user's data for diverse advanced downstream analytics and visualization use cases. Because Logstash acts like an event forwarder with multiple features and supports plugin, it usually be used to gain data from data sources simultaneously, then handle the results to another place[4].

In fact, Logstash does not act an important role in our system, which is used as an input for Kafka. Considering we need to process real-time data, which means that it may have a very large amount of data in a short time which beyond the computation ability of our cluster. Therefore, as a message queue, Kafka is necessary and important to act as the cache for input. Kafka is a distributed, partition-based and multi-replica-based distributed messaging system coordinated by Zookeeper. Its biggest feature is that it can process large amounts of data in real time to meet various demand scenarios: such as Hadoop-based batch processing system, low-latency real-time system, Storm / Spark streaming processing engine, web / Nginx log, access log, message service, etc., written in Scala language[5][5]. With Kafka, the computation cluster can read the input data when it is able to process it.

The next part of our system is Apache Storm, which is also the core cluster for computation. Storm has good performance on real-time streaming processing, which is very suitable in our scenario model. Apache Storm is a free and open source distributed real-time computation system, which is currently being used to run various critical computations in Twitter at scale, and in real-time. Apache Storm makes it easy to reliably process unbounded streams of data, doing for real-time processing[7].

Additionally, we use Redis for caching the data we need in processing the logs (the coordinate) and the output data. The reasons we use Redis instead of other databases such as MySQL are the three main points as following. (1) Redis is a NoSQL database, which means that we do not need to design tables and its fields. (2) Redis is an in-memory data structure store[8], which is fast in inserting and searching with a key. (3) Redis offers a function to expire

the items in a specific time period automatically, which can help us to ensure the real-time validity of data in the cache.

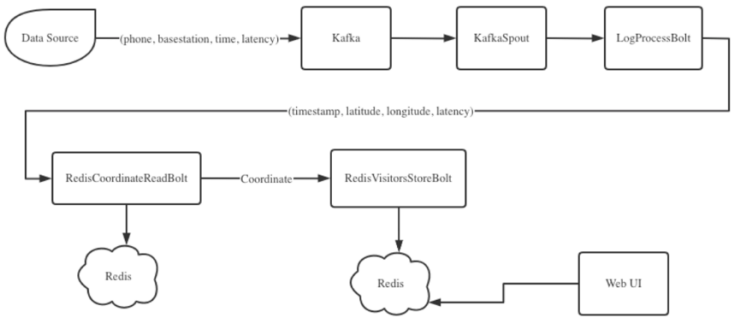

Figure 1. System Design Diagram

c. Storm Topology Implementation

Since we use Apache Storm as the core of our system, in this part we will introduce the Storm topology implementation.

The first part of the topology is the *KafkaSpout*, which not only acts as the interface to connect to the message queue Kafka, but also acts as the data source in the Storm topology. In order to connect to Kafka and read the real-time data cached in Kafka, we need to configure the Zookeeper address used by Kafka into an instance of a class called *BrokerHosts*. Additionally, we need to put the topic which Kafka stores data, and a root directory which stores the location information for *KafkaSpout* (which is also means the offset), as well as an id, together with an instance of *BrokerHosts* into the configuration of Spout. The last step is that we need to set the start offset time of the Spout, which is used for reading the offset. In other words, if we set the offset as the latest time, Spout will read the latest data instead of reading the data from the beginning and get all the history data.

The second part is called *LogProcessBolt*, which is used to process the real-time logs from Kafka. In Storm, the output of the spout connected to Kafka is in bytes. Therefore, we should use a byte array to store the data get from the *KafkaSpout*. In *LogProcessBolt*, we get the fields we want and filter the others. Then we emit the data in the fields of time, latitude, longitude, and latency to the next bolt.

The third part is called *RedisCoordinateReadBolt*. This bolt is used to look up the Redis database which stores the coordinate and get the coordinates we need according to the base station location and the latency. About the details in our coordinate design, you can read part d. This bolt extends *AbstractRedisBolt*, which is flexible and can be used for complex business logics. In *AbstractRedisBolt*, we use an instance of class J*edisCommands* to read the data from our Redis cache. The data we stored in the Redis cache for this bolt is our coordinate, which type is list. After finishing the all these works, it will emit the data in fields of location and visitors to the next bolt.

The fourth part is *RedisVisitorsStoreBolt*. This bolt is used to do the statistics works. To be specific, it will count the locations and store the location as well as the counts into another Redis cache for our front-end. The structure of this bolt and the way we connect to Redis is similar to *RedisCoordinateReadBolt*, but the logic is more complex. We use the location as the

key. If the key does not exist when we increase the value of the key, its value will be initialized as 0 and set the key as persistent. Since we need to set the expire time (by default is 30 seconds in our system), we should first judge whether the key is timeout. If the key is timeout, we can increase the key and set the expire time. If not, and if the key exists for the first time, what we should do is only to set the expire time.

d. Coordinate Design

Since the emphasis of this project is not on analysis algorithm, we simplify the problem of estimating population density. We map the real world into a grid, as shown in Figure 2, each square has a counter representing the relative population density in the corresponding area. The input of the system is real-time anonymous user connection info (bid, l), where bid denotes the base station the user connects to and l the communication latency.

For example, lets' say the two blue squares in Figure 2 denote the location of two base stations, the two light gray circle area denotes the area covered by each station with the communication latency in range [0,50] millisecond, the overlapping area of the two circles is filled by dark gray. Now, two users, located in the overlapping area in real world, each connects with one of the two stations. Each connection results in the incrementation of the counter of each square in the corresponding circle, causing the counter of squares in the middle dark gray area to be 2, which denotes that there is a higher population density in that area.

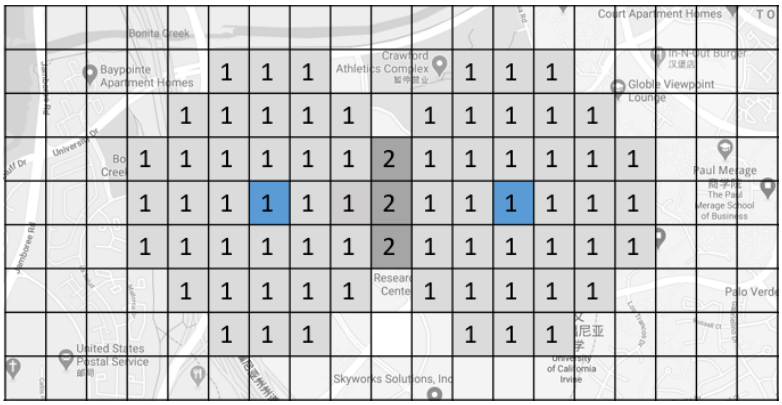

Figure 2. Example of the method we use to estimate population density

If the distribution of base stations and connections are dense enough, this method can generate a reasonable estimation of the distribution of population density. Note that the process of incrementing the counter of each square is independent of each other, thus the whole method is highly parallelizable. Figure 3 shows the dataflow of one of the parallel processes. To further improve the performance, we precompute a list of coordinates of covered squares corresponding to each base station in each latency range.

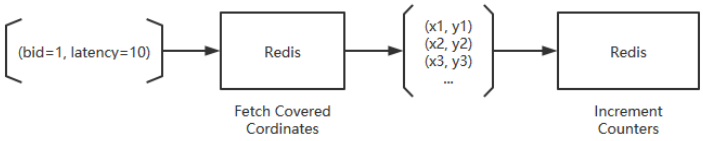

Figure 3. Flowchart of the method

e. Front-End

In the web front-end, we use Spring Boot as our framework. To connect to Redis, we use a class called *StringRedisTemplate*, because the type in the Redis cache used by the front-end is String. For heap-map rendering, we choose Google Map APIs.

III. Test and Evaluation

a. Scalability

With the help of the flexibility and scalability of Apache Storm, it is not a difficult thing to add machines into our system cluster if we need to improve the computation ability.

In our test environment, we set a cluster on AWS with 3 EC2 t2.medium instances. Table 1 shows the roles of the nodes.

Node Number	Zookeeper	Nimbus	Supervisor
0	☑	☑	☑
1	☑		☑
2	☑		☑

Table 1. Cluster Nodes

Here we are going to show the steps of adding a node into the cluster to certify the scalability of our system, which mainly contains the following simple steps.

(1) Installing JDK and Python. Our system in Storm relies on these two languages. JDK 8 and Python 2.7 are the best choice.

(2) Configuring the Zookeeper cluster. Storm cluster must rely on a Zookeeper cluster, and our Storm cluster will also read data from Kafka with the help of a Zookeeper cluster. What we should do for the new machine is modifying its Zookeeper configuration file and adding the Zookeeper hosts and ports of other machines in the cluster. Besides, we should set the id in the cluster called *myid*, which must be unique across the cluster.

(3) Configuring the Storm cluster. After installing Storm, we should modify its configuration file to set the Zookeeper servers in the cluster (host names) and the supervisor ports in this new node. Another important step is that we must set the nimbus seeds as all the hosts in the machine. In Storm 1.1.1, it will scan all the hosts in nimbus seeds to find the nimbus node.

b. Fault Tolerance

(1) Process-level fault tolerance. Our system in Storm has several different daemon processes. For example, nimbus that schedules workers and supervisors launch and kill workers. According to this, we did some tests in process-level. When we let a worker die, we found the supervisor will restart it. The new worker will restart on the original supervisor node initially. However, we found that it the worker continuously fails, nimbus will reschedule the worker to another node. Then we test the situation when a node dies. We found in the logs that there were several messages to show that the tasks on the machine was timeout, and then nimbus

reassigned those tasks to other machines. We also did tests on the situations when nimbus or supervisor daemons die, and the result is that these demons will restart automatically. That is because the nimbus and supervisor daemons are designed to be fail-fast (process self-destructs whenever any unexpected situation is encountered) and stateless (all state is kept in Zookeeper or on disk)[9].

(2) Zookeeper fault tolerance. We also tested the situations that if Zookeeper fails, and we found that if more than half of the Zookeeper nodes survive, the Zookeeper cluster can run normally.

c. Optimization

To improve the performance and reduce the system latency, we need to change some parameters in Kafka and our Storm cluster to try to find out the best configuration.

In fact, optimization works for a Storm cluster needs experience, which has not a fixed way and need to be analyzed according to the specific situation, such as the peek input data volume and the cluster hardware performance.

Before the optimization works, we run the topology by all default configurations, which means that the number of workers is 1, and the number of executors (which also means the parameter parallelism) of each spout or bolt in the Storm cluster is also 1. Figure 4 shows the latencies of different part of the topology by all default configurations in one test before optimization. Notice that it is a result of one test, the latency fluctuates in different tests.

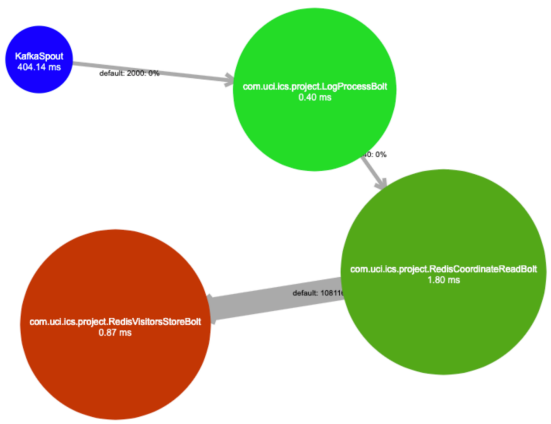

Figure 4. The Topology Performance Before Optimization

There are many configurations we can change in our system. To simplify the problem, we set some parameters fixed. First, we do not change our test environment, which is three EC2 t2.medium instances on AWS, which is showed in part a.

Name	vCPUs	RAM(GiB)
t2.medium	2	4.0

Table 2. t2.medium details[10]

According to the materials we read, it is a better choice to set number of workers in a Storm cluster the same as the number of cores in the cluster. One t2.medium instance

has a CPU with 2 cores. Therefore, totally, we have 6 cores in our test cluster. As a result, we set the number of workers, which is our second fixed parameter, as 6.

The following test and experiments we did is based on the fact that we input 2000 real-time logs at one time. From the test we did before optimization, we find that the bottleneck of the cluster is the *KafkaSpout*, which takes 404.14ms. And the other parts in relatively fast, which means that we do not need to improve their performance.

To reduce the latency of our spout which connects to Kafka, we try to improve the parallelism of it. Figure 5 shows our results, each parallelism (spout number) we test for 10 times and get the average latency. It shows that in our cluster, when the parallelism of *KafkaSpout* is 2, the performance is the best.

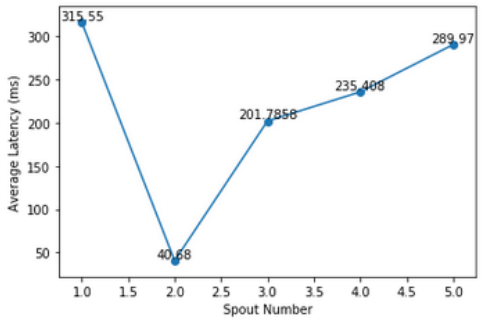

Figure 5. Experiment Results for the Parallelism of *KafkaSpout*

Another method come to our mind is to improve the partition of Kafka (we set this parameter as 1 initially in our Kafka topic). Unfortunately, it seems do not work for our cluster. It might be related to the number of disk partitions in one EC2 instance. Because of the imitation of time, we do not conduct in-depth researches on this phenomenon. After the experiments, we still set the partition number of our Kafka topic as 1. Figure 6 shows the results of the experiments.

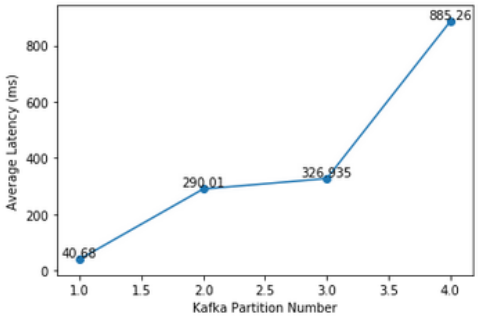

Figure 6. Experiment Results for the Kafka Partition Number

After our optimization works, we successfully reduce the latency of *KafkaSpout*. Figure 7 shows the best performance we meet in our tests after optimization works. The latency of *KafkaSpout* is reduced to 16.66 ms.

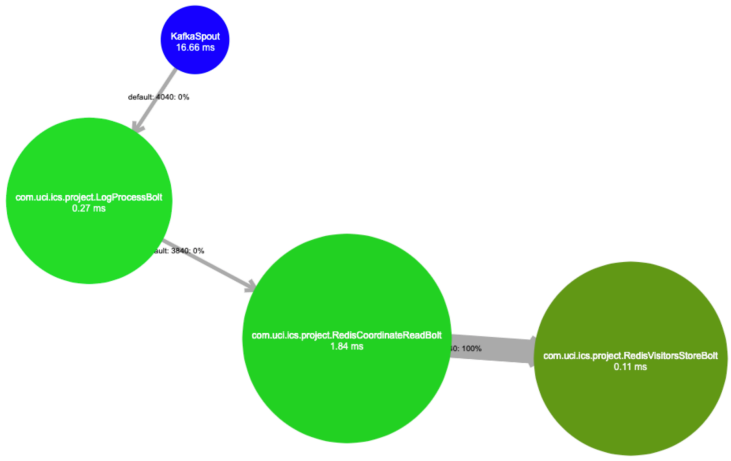

Figure 7. One of the Best Performances after Optimization

d. Web UI Results

To ensure that our results are real-time, we set the expire time of the data we render on the web UI will be expired in 30 seconds. We use the heap map in Google Map APIs to show the real-time visitors flow. The areas with the darker colors mean that there are more visitors in that areas. Figure 8 shows the results in our web UI and its changes.

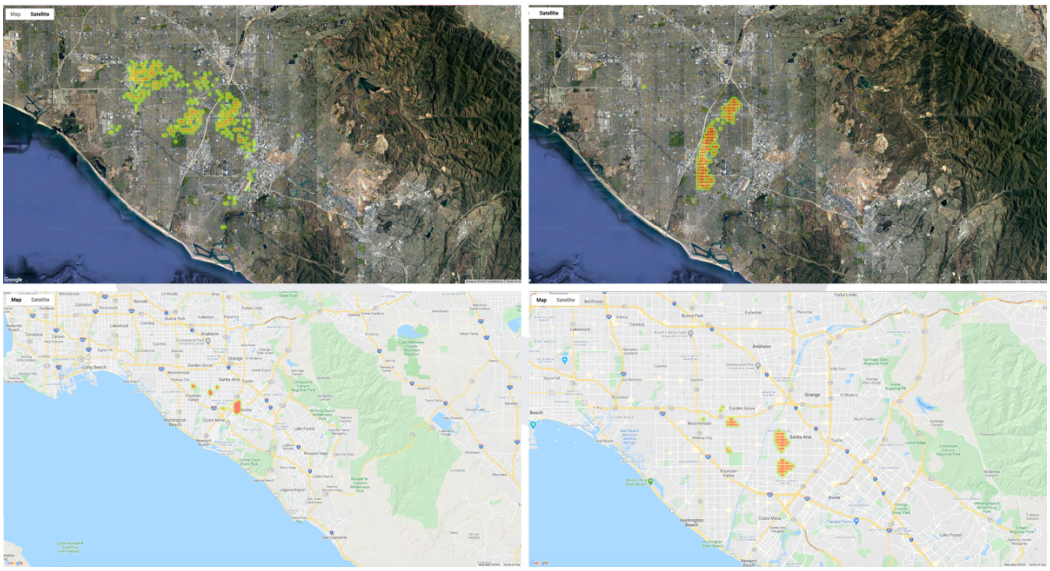

Figure 8. Web UI Results

IV. Future Works

To improve the performance of the system continuously, we can deploy a Kafka cluster to connect to our data source to improve the performance of our message queue. Additionally, we can also add some application modules and functions in our web application, such as adding a

warning function which will be triggered when the visitors flow exceeds a threshold in a specific area.

Reference:

[1]. Ramachandran, “Systems, methods, and computer program products for estimating crowd sizes using information collected from mobile devices in a wireless communications network”, US8442807B2, 2010.

[2]. Chen, Chun-houh, Wolfgang Karl Härdle, and Antony Unwin, eds. *Handbook of data visualization*. Springer Science & Business Media, 2007.

[3]. Wikipedia, https://en.wikipedia.org/wiki/Signaling_(telecommunications).

[4]. Marcin Bajer, “Building an IoT Data Hub with Elasticsearch, Logstash and Kibana”, 2017 5th International Conference on Future Internet of Things and Cloud Workshops.

[5]. Apache Kafka online official document, https://kafka.apache.org/documentation (accessed 05. 07. 2020).

[6]. Jay Kreps, Neha Narkhede, Jun Rao, “Kafka: A Distributed Messaging System for Log Processing”.

[7]. Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. “Storm@twitter”. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD ’14).

[8]. Redis Official Documentation, https://redis.io/documentation.

[9]. Apache Storm Official Documentation, http://storm.apache.org/releases/1.2.3/index.html .

[10]. Amazon Web Services, Product Details, https://aws.amazon.com/ec2/instance-types/t2/ .