RETINAA: A Real Time App Analytics Framework

Code Base https://github.com/bourneagain/retina
Demo https://www.youtube.com/watch?v=f__ihYnPPfk&hd=1

Rajendran, Shyam
Masters Computer Science
University of Illinois, Urbana Champaign
srajend2@illinois.edu

Roychowdhury, Saikat
Masters Computer Science
University of Illinois, Urbana Champaign
rychwdh2@illinois.edu

Sharma, Abhinav
Masters Computer Science
University of Illinois, Urbana Champaign
sharma55@illinois.edu

***Abstract*—The mobile application market is booming at a rapid rate. Application development has become pervasive. Problem arises when these applications, without proper testing, bring down the platform stability and overall reputation. The application developers would greatly benefit from a real time system, which can gather the crash reports and other application statistics such as user base demography, click streams in real time and help them fix bugs and launch features proactively. This can be used for monetization and tweaks in the UI depending on the user's interaction stats. Although platform providers (Apple and Google) do provide basic analytics information to developer they do not meet specific application developer's requirements, as the infrastructure provided is very proprietary and restrictive.**

Keywords— Druid, Kafka, Storm, Real Time Analytics

I. INTRODUCTION

Our goal is to build a generic real time events analytics platform. In this project we are using the platform as a real time app analytic dashboard to give insight into the application and user statistics with drill down. The RETINAA aims to help mobile application developers understand their app statistics in **real time** across millions of users around the world and provide in depth detailed analysis of crash and usage statistics to improve the development process. We plan to gather stream of application logs and help the developers in understanding the application statistics including top open bugs to be fixed, overall app stability and the application reach.

The usual procedure for analysis happens to be to post process the collected log in batches to gain insight rather than in real time and the primary means of data observations, in terms of viewing stats, are very limited to having to write our own code logic. This has been proven to be slow and and we plan to leverage systems which are designed specifically for this purpose to build our RETINAA infrastructure.

II. BACKGROUND AND CHALLENGES

First, the usual means to solve large scale metrics analysis problems (at scales petaByte), the preferred option would be use Spark with Parquet format. Apache Parquet[1] is a columnar storage format available to any project in the Hadoop ecosystem. Spark's Resilient Distributed Dataset (takes care of keeping the data in memory for repeated queries on the same data set and Parquet takes care of the columnar format. Other options would be Dremel clones like Impala, Drill or Druid – Impala is very much tied to Hadoop and doesn't do it in memory like Spark, and Drill looks conceptually promising but keeps us waiting for GA. Meanwhile Hive on Tez is picking up steam and appears to comprise a more stable option – but again not an in-memory solution.

The implementation with Druid over Spark or any other analytics gives us the ability to scale to millions of clients with very strict query latency restrictions and TBs of event logs being ingested in real time having very minimal or no influence on the query latency. We have studied the use of Druid in the industry where they have been deployed to handle billions of clients. (This has been tested with a simple experiment comparing MongoDB implementation Vs Druid for aggregation detailed under Section IX).

We did a survey of prevalent technologies that is used in the industry and is the state of the art in real time analytics. Some of such systems are deployed at Pinterest, Netflix, VMware. Pinterest[1] has a version of real time analytics where all event logs are collected and shipped to a centralized repository. The infrastructure has adaptive log processing intervals but the development has largely being proprietary in house development activity which cannot be used as a plug and play product in the industry. Netflix[2] has Suro, an infrastructure to collect and analyse real time events such as log messages, user activity records, system operational data, or any arbitrary data that their systems need to collect their business, product, and operational analysis with minimal latency. They generate real-time trends are built with a pipeline of Hadoop jobs and they also dispatch to designated Kafka cluster. They then make use of Druid cluster to make the indexes available immediately for querying. VMware[3] has a real-time log management with machine learning-based Intelligent Grouping, high performance search and better troubleshooting across physical, virtual, and cloud environments. Other than that ebay and metamarkets are using DRUID based implementation heavily. Druid[4] is used for real-time user

behavior analytics by ingesting up at a very high rate(over 100,000 events/second at ebay and over 30 billion events per day at metamarket. Jolata calculates a billion metrics every minute to visualize precise network metrics in real-time with drill down options. These industry reports indicated us to strongly use Druid for our application over other database analytics platforms with the view of scalability. The trend in the industry that we see companies making system which target specific use cases[6] [7] [8] [9] [10] [11] [12] [13] [14]. We are trying to build a framework generic enough to handle any type of streaming events, be it crash logs, click streams, streaming data from sensor networks.

III. RETINAA DESIGN

We have our platform built over 3 main stages.

Ingestors

In the first stage are our distributed log ingestors. These servers collect the event data pushed from the mobiles. The ingestor servers dump data into a time ordered Kafka[15] queue.

Storm Cluster

To process the streaming event messages, we have built a Storm cluster. The cluster feeds data off a Kafka spout[16] into a set of bolts dedicated to parsing the different types of event messages. The processed event stream is pushed onto a Kafka Queue.

Druid Network.

The Druid implementation forms the core of our RETINAA. DRUID ingests the processed data off the Kafka queue. This data feeds our WEBUI for gathering app analytic and metrics information.

A simple call flow of RETINAA pipeline works as follows:

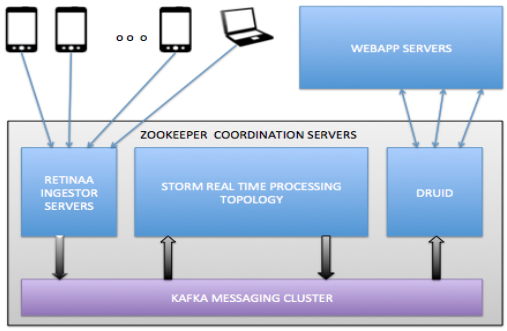
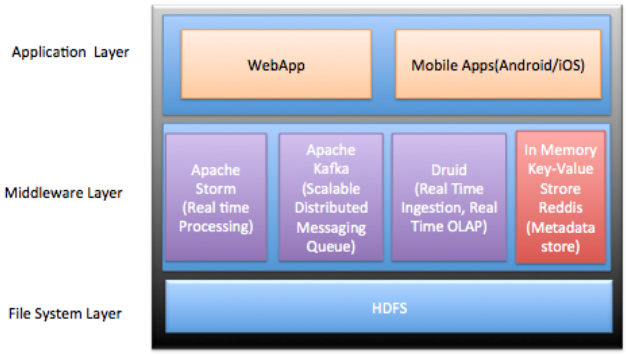


Fig 1.1

1. RETINAA event collection service, which runs on mobile devices/desktops/laptops, sends the logs periodically over the one of the servers of the RETINAA Ingestor Cluster.
2. The Ingestor server dumps the raw events to the Kafka queue under a topic name say ‘A’.
3. A Storm spout consumer group listens for messages listed under topic ‘A’ in the Kafka queue.
4. Once a message is retrieved by the spout, it is passed on to one of the worker bolts for further processing.
5. The processed event is the dumped to the Kafka queue again under a different topic say ‘B’.
6. Druid listens for messages posted under topic ‘B’ and ingests that into one of its real-time nodes and then subsequently passes it to the historical nodes, when the time segment (as per configuration) expires, which then stores the event logs in the HDFS.
7. When a developer logs in to RETINAA webapp, aggregations, filter queries are fired to the Druid broker node which then retrieves the data. This data is presented as an intuitive real time graphical display on the webapp dashboard for the developer to view.
8. All the components Storm, Apache and Druid use its own instance of zookeeper cluster. We plan to avoid a single cluster for all the three components as this will overburden the zookeeper cluster (More details on this in the section X)

IV. IMPLEMENTATION

RETINAA stack is divided into three primary layers based on the core functionality of the layer.



RETINAA STACK

Fig 1.2

A. APPLICATION LAYER

This layer directly interacts with the users. The interaction is primarily at two places. Mobile App side, where retina service runs in the background and collects log events from the developer apps. On the web application side, the developer is presented with a dashboard view of the real time trends and changes in the various app analytics metric. We also integrate **cross-region load balancing**[17] for our client facing web application.

B. MIDDLE LAYER

This is the core pipeline of RETINAA where the processing of app event streams is done. The primary components of this layer are Apache Storm, Apache Kafka, Druid and Redis[18]. Kafka is the distributed messaging queue over which Storm and Druid communicate. The workflow of the three components is explained in the next section.

C. FILESYSTEM LAYER

This is the storage layer of RETINAA, which is HDFS file system, and is utilized by Druid to store the event logs.

V. RETINAA COMPONENTS

A. Ingestor

Ingestors handle main entry point of logs from mobiles**.** The ingestor load is distributed based on the mobile event originating by deploying region wise ingestor servers connected to a central Kafka queue. The ingestor servers are web servers which process the HTTP POSTed event logs into the Kafka.

B. Storm[19] Topology

There are multiple spouts (= number of partition of a Kafka topic) running in parallel on different machines which reads data from Kafka Queue. Each spout is a part of consumer group to consume logs from Kafka topic. This is scalable as spouts are running on different machines handling different data, each from a topic's partition, so logs coming to spout are parallelized and getting data in totally ordered manner.(each Storm spout reads from a single partition of a Kafka topic, which is itself totally ordered)

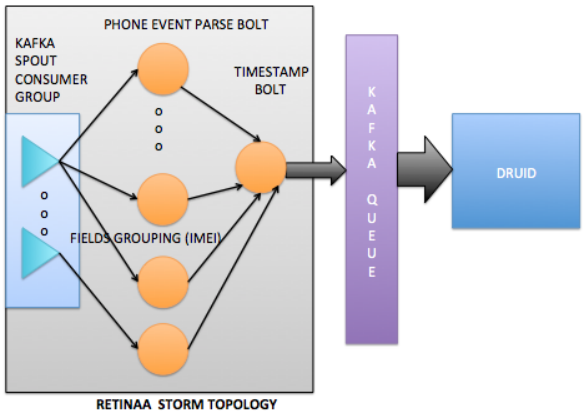


Fig 1.3

Next, there are multiple bolts that are connected to spouts using Field's grouping base on phone's IMEI number. They will all process the incoming logs in parallel and transform them to retina event format as expected by Druid. Parse bolts are also responsible for performing partial aggregation of error counts and click stream counts on the received log tuple. This aggregation is explained in Section VI. The parallel processing based on Hash Partitioning to multiple bolts on different machines is again scalable. These Parse bolts can easily scale out as system load increases.

At the end, a single Timestamp bolt receives formatted Json in retina's format from using all grouping and dump it into Kafka queue by assigning a timestamp to them so that they are totally ordered. We discuss of limitation of Timestamp bolt in Section X.

C. Apache Kafka

Kafka is a distributed, partitioned, replicated commit log service. Kafka basically divides the incoming messages into different partitions for scalability. Each partition has different replicas for fault tolerance. Each partition has a leader and multiple followers, where reads and writes go to leader first and followers try to catch up leader. Producers are processes (can be on different machine or servers) that pushes the data in Kafka cluster. Consumers can be in a consumer group and there can be multiple consumer groups that are fetching data from Kafka in parallel. Each consumer in a group maintains a offset in the queue and fetch the logs in linear fashion. Each consumer within in a group fetches log from a fixed partition so that it receives all the logs in totally ordered manner.

Scalability : Overall architecture of Kafka is highly scalable as it can hold huge amount of data in several partitions and performance is constant in amount of data since consumer maintains offset by itself.

D. Druid

Druid[20] is an open-source analytics data store designed for business intelligence (OLAP) queries on timeseries immutable data. Druid provides low latency (real-time) data ingestion, flexible data exploration, and fast data aggregation. In Druid, data is divided into segments where each segment defines data collected over some defined interval. Druid stores data in form of segments. Druid also requires deep storage (HDFS) where segment resides, metadata storage (relational DBMS) where metadata of each segment is present and zookeeper which helps in coordination b/w different type of servers. Druid consist of 4 type of servers each with different purposes.

- **Real Time Nodes** - These are connected to some data source which can be Kafka, irc channel or some kind of firehose which is continuously pushing data to Druid. It segments the data, it receives, on a well defined interval; calculates the metadata of that segment; push the segment into deep storage and also after some interval pushes the metadata of segment to Zookeeper for Historical node to fetch.

a. **Historic Nodes -** These nodes receives segment metadata information from zookeeper about which segments to load or drop from their local disk (cache). When it receives metadata, it knows from the metadata about where to download the main segment from the deep storage, how to decompress the data, process it and load into local cache. Upon receival of segment metadata, if the segment information is already present historic node may just load the data from the disk. It also receives messages related to dropping of segment or shifting to another coordinator from the coordinator node and it acts accordingly.

- **Coordinator Nodes -** They are primarily responsible for segment management and distribution. The Druid coordinator is responsible for loading new segments, dropping outdated segments, managing segment replication, and balancing segment load. They are connected to some sort of metadata storage like MySQL so that they know about the segments present in the system and whether they need to be dropped or not. Also, it checks the distribution of segments over historical nodes and try to keep them uniform by shifting some of the segments from highly loaded to other historic nodes.

- **Broker Nodes -** They are primary contact point for client. Client contacts the broker node with some query over some interval. It first get the view of all the segments present in historic nodes and get the relevant segments that overlaps with the queried interval. Also, it contacts real time nodes explicitly because their segment information may not be present in zookeeper yet. It then maintains a table of all the fetched segments and on which node they are present. It finds

out the relevant segments and saves all that information in memcache for further reuse.

Use Case of Druid

Druid is built for exploratory analytics for OLAP workflows. It supports a variety of filters, aggregators and query types and provides a framework for plugging in new functionality.

- **Highly Available** Druid is used to back SaaS implementations that need to be up all the time. Your data is still available and queryable during system updates. Scale up or down without data loss.
- **Scalable** Druid's real-time nodes employ lock-free ingestion of append-heavy data sets to allow for simultaneous ingestion and querying of 10,000+ events per second. Existing Druid deployments handle billions of events and terabytes of data per day.

VI. RETINAA MESSAGE

In order to reduce the traffic from phone and keep the bandwidth usage to a minimum, we have three types of event messages pushed from mobile: Registration Event, Heartbeat Event, Logs Events. All retina events are encapsulated in JSON format. These events can be pushed from target devices to RETINAA Ingestor servers, which ingest the events in the main processing unit (Storm), where the events are processed and filtered further in real time. A, per-device, RETINAA event reporting daemon is responsible for constructing these events and pushing to the ingestor servers.

b. Registration event:

This event is used by devices to register phone and app metadata with RETINAA. The important fields of this event are local timestamp of event w.r.t the device, Phone metadata like Phone IMEI, Phone model, Phone Kernel Version, Operating system version, App Meta data like App version, App ID. When RETINAA receives this metadata, it stores the metadata in an in-memory look-up table. This is important since all future events will be correlated with an entry in this lookup table to perform necessary join operation on event logs.

A sample Registration looks as follows:

```
{
"phonetimestamp" :"2015-04-29T22:45:09Z",
"eventid":"1",
"eventtype":"M",
"phoneimei":"9900000000000000",
"appid":"appid0",
"logs":"Lollypop Nexus4 m.x.y.z LRX22C\nappid0 0.1
app0\n"
}
```

[Note - "logs" field contains space separated meta data information for easy parsing]

c. Heartbeat Event:

Heartbeat event sent periodically from the devices to RETINAA. This event is used to track which app is currently active in which phones. Since this is a periodic message, we designed the JSON schema to contain only bare minimum information namely phone IMEI, and App ID, to capitalize on network bandwidth savings.

When RETINAA receives Heartbeats, the phone and app meta data is correlated to an entry in the lookup table. If a match exists the event is passed onto Druid, else the heartbeat is just ignored. We have designed this to be very generic with the possibility to relay information about the percentage time the app spends in various process states like foreground, background, paused, stop states.

```
{
"phonetimestamp":"2015-04-29T22:45:14Z",
"eventid":"1",
"eventtype":"H",
"phoneimei":"9900000000000000",
"appid":"appid0",
"process state": <>    // for future use
}
```

d. Logs Event:

This is the core event type of RETINAA, which encapsulates various log events that we want to capture about an App. The "logs" field of the log event contains "\n" separated lines of various types of logs supported by retina i.e various debug level logs like error, warn, alert. We also support various click event logs for various components of the app which the developer might be interested in. The design facilitates easy addition new event types by registration of a keyword with RETINAA. This helps in scaling the framework to support new event types in future.

```
{
"phonetimestamp":"2015-04-29T22:45:26Z",
"eventid":"1",
"eventtype":"L",
"phoneimei":"9900000000000000",
"appid":"appid0",
"logs":"Error:<error logs>\n
Warn: <Warn logs>t\n
Click:component1\n
Crash:<Crash logs>\n
Error:error print1\n
Crash:<Crash logs>\n
Error:error print2\n
Crash:<Crash logs>\n
Error:error print3\n
Click:component11\n"
}
```

VII. SCHEMA DESIGN:

Current version of Druid supports only single level schema definitions and fixed number of column definitions. So we had to flatten RETINAA event information into one level schema definition. This flattening of schema is done in RETINAA's Storm processing layer. Storm is also responsible for partial aggregation of click stream counts and error counts received in the "logs" Final Druid schema is as follows:

```
{
{"phonetimestamp":,
"eventid":,
"appid":
"phoneversion":
"phonemodel":
"phonebaseband":
"phonebuild": "myphonebuild"
"phonebaseband": "myphoneversion"
"appversion": "myappversion"
"appname": "myapp"
"eventtype": "<M/H/R>"
```

```
   “eventerror”: “<>error logs”
   “eventwarn”: “<warn logs>”
   “eventclick”:“component_name1:50
component_name2:100 ” // Storm generated partial
aggregation per component
   “crashcount”: “1”  // Storm generated partial
aggregation
   “errorcount”: “10” // Storm generated partial
aggregation
   “warncount”: “30” // Storm generated partial
aggregation
   }
```

This JSON is dumped into the Kafka queue under a topic which Druid is listening to. As we can see, a timeseries query to return aggregation result over any specific event can now be efficiently performed by Druid using fileds “errorcount”, “warncount”, “clickcount” etc.

VIII. EXPERIMENTAL SETUP

We have deployed the RETINAA stack on the Google Compute Engine Cloud platform and have designed to have it scale well with increased user load through the use of cross region load balancing features available in the compute stack.

a. Simulated Phone Data Generators:

We have two dedicated servers that pump the simulated event traffic into to our RETINAA cluster from different regions of the cloud platform. These servers form the data source for our project. . We can programmatically control the number of unique devices, unique apps, app version and the rate at which the logs are generated from these servers.

Simulated Phone Data Server 1 and Simulated Phone Data Server 2

Region	**asia-east1-a** and **europe-west1-b**
OS	Ubuntu 14.02 LTS
CPU	Single core @ 2.5 GHz
RAM	4 GB

Table 1

b. Ingestor server

We have region-wise ingestor servers which are the only point of contact to the mobiles (simulated phone servers in our case) to dump the event logs. The Asia region traffic is diverted to www-2 ingestor while the Europe region mobile traffic is diverted www-1 ingestor. Ingestor servers were implemented as Java based web server listening for POST requests.

Region	**asia-east1-a** and **europe-west1-b**
OS	Ubuntu 14.02 LTS
CPU	Single core @ 2.5 GHz
RAM	4 GB

Table 2

c. RetinaWebUI:

We developed a simple web application, which is accessible to the registered developers where the developers can see the usage stats and application metrics in real time for their deployed application.

Presently we support only Android platform, which provides a special permission request upon app installation, which allows users to accept to sharing the usage stats to the app developer.

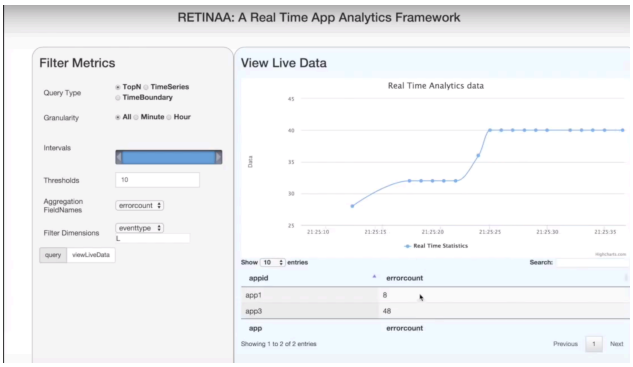


Fig 1.4

The interface provides a basic view of per application trending timeseries event. These could be top N applications of the developer as per error count as observed in real time (Fig 1.4)

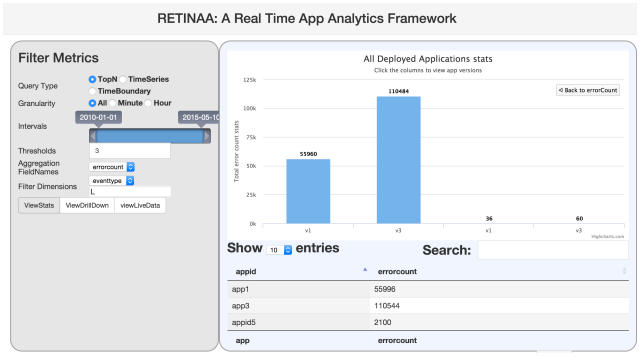


Fig 1.5

We plan to extend support to view application usage pattern across the globe in real time.

We also have provided a drill feature (Fig 1.5), which will show the metrics, based on the chosen application’s different versions (Fig 1.6) or the mobile instrument type as a second level of statistics.

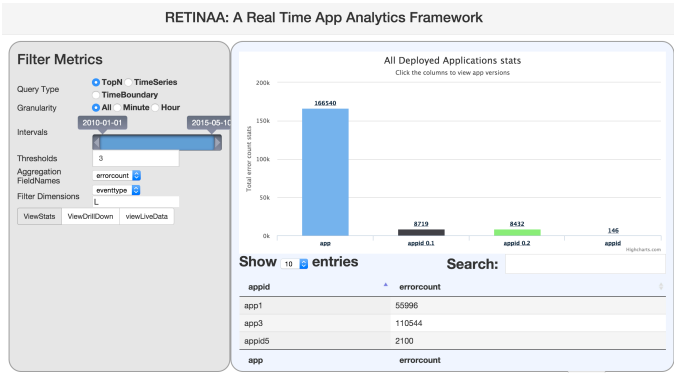


Fig 1.6

Application Server deployment

The server is configured to utilize the Google Cross Region load balancing feature. The webUI servers are created in different regions and zones and we assign a global proxy ip address. We tested the load-balancing feature by pumping data from **US** (*www-1*) and **Europe** (*www-4*) regions (Fig 1.7) to the application server on global IP to view the traffic being diverted to appropriate balancing servers.

```
   www-1    us-central1-b    n1-standard-1    10.240.220.118
104.197.94.21    RUNNING

   www-4    europe-west1-b    n1-standard-1    10.240.211.148
104.155.110.252 RUNNING


   Global    Proxy    IP    accessible    RETINA    WEBUI:
http://107.178.246.128/a.php
```

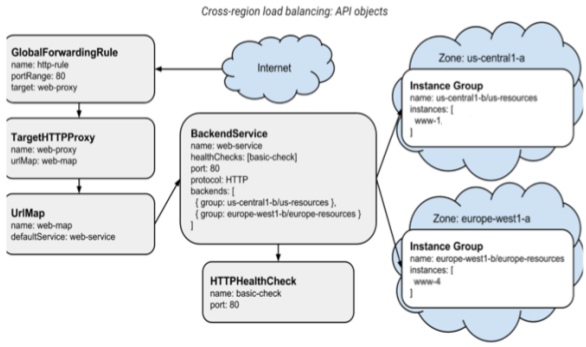


Fig 1.7

d. Main Topology Cluster

As a prototype, we developed our main event-processing cluster, which consists of Storm, Kafka and Druid configured on single high performance server managed by central ZOOKEEPER. This is due to the limitation of the Google Compute engine that provides a maximum of only 8 cpus allocated for US region servers. But we tested individually multi cluster configurations of Storm, Kafka, Druid and Zookeeper prior to building the final processing cluster consisting of these individual units.

Google Compute Server Details

PhoneSimulatorServer1	asia-east1-a	n1-standard-1	10.240.22.238	104.155.225.233	RUNNING
PhoneSimulatorServers2	europe-west1-b	n1-standard-1	10.240.84.94	130.211.103.142	RUNNING
Ingestor Servers					
www-2	asia-east1-a	n1-standard-1	10.240.222.16	104.199.142.123	RUNNING
www-3	europe-west1-b	n1-standard-1	10.240.25.249	104.155.79.128	RUNNING
Load Balancers for Web Application					
www-1	us-central1-b	n1-standard-1	10.240.220.118	104.197.94.21	RUNNING
www-4	europe-west1-b	n1-standard-1	10.240.211.148	104.155.110.252	RUNNING
Main Topology	us-central1-a	n1-standard-1	10.240.176.94	104.154.46.154	RUNNING

Table 3

e. Firewall setup.

Google Compute requires specific ports to be opened up for external access. Following firewall rules were added as per our project requirements. (Table 4)

default-allow-rdp	default	0.0.0.0/0	tcp:3389
default-allow-ssh	default	0.0.0.0/0	tcp:22
Druid	default	0.0.0.0/0	tcp:8082
ingestor	default	0.0.0.0/0	tcp:1234
Kafka	default	0.0.0.0/0	tcp:9092
stormui	default	0.0.0.0/0	tcp:8080
webapp	default	0.0.0.0/0	tcp:80
www-firewall	default	0.0.0.0/0	tcp:1234
zk	default	0.0.0.0/0	tcp:2181
zkleader	default	0.0.0.0/0	tcp:2888
zkquorum	default	0.0.0.0/0	tcp:3888

Table 4

IX. PERFORMANCE EVALUATION.

We started to first evaluate the use of Druid over other available database systems especially for our use of with Json. Our requirement was to do interactive analysis on nested data. We restricted ourselves to using in-memory data for our analysis. We also eliminated use of any SQL database for obvious reasons of unstructured data use case.

A. MongoDB

We posted the test data consisting 1 million Json event log into MongoDB[21] and ran the query to fetch top N=3 applications based on gathered error count. Though this is not entirely in-memory analysis but we assumed near cache performance with Mongodb.

B. Druid

Druid requires flattened Json to work. We posted the same Json after creating Druid specific columnar data structure and in the spec file. We assumed that there is no query to the Druid deep storage and query acts on the in-memory data alone to be fair with MongoDB.

Both the query tests were performed on the same machine (with hard reboots to clear cache between the tests) and observe the below numbers.

For MongoDB, we wrote a small JavaScript code to calculate the Top 3 applications by error count. The Druid has Top N query support in built and needed no additional code other than specifying the query format in a Druid specific Json query file.

```
MongoDB
MilliSecs for query exec: ~10s

Druid
MilliSecs for query exec: under 1s
```

We could a 10x improvement in top N specific query with Druid and best suited our use case of analytics. Further Druid also provides support good query that leverages timestamps (specific duration interval). MongoDB requires us to code for such uses.

X. CHALLENGES AND LIMITATION

Flattening the schema to meet Druid compatibility requirements

The Timestamp Bolt in RETINAA's Storm topology right now is a single processing unit, which assigns totally ordered timestamp to RETINAA events. All the Parse bolts are connected by a global grouping scheme to TimeStamp bolt. This is because, in present implementation of Druid, if we pump any event, which is older than the latest timestamp event seen, the event will be simply ignored by Druid and not

ingested by the real time servers. There is ongoing work to fix this, and later version of Druid will have this enhancement. This will enable RETINAA to use multiple timestamp bolts, where we can have some kind of fields grouping scheme to parallelize the assignment of timestamps.

We discovered that Druid supports single level flat schema for data objects. This requirement meant that we needed to the schema flattening processing in Storm before sending the modified schema to Druid. If we take the example of Click stream event counts, in the present schema format, a simple filter and aggregation query on Druid will efficiently return all the rows which contains the matched click events and the aggregation field "clickcount". So if we need a further drill down to show the actual click events, we need one separate stage of aggregation on the Druid query output. This can be done efficiently using a NoSQL database(e.g. MongoDB) to store the query results of Druid. Modular design of RETINAA enables easy integration with MongoDB.

In present Implementation of prototype RETINAA, the processing bolts of RETINAA's Storm layer have in-memory hash table to store the metadata information about apps. This information is used to perform streaming join operations on the incoming event streams. If the metadata is large enough such that it heavily loads the processing bolts, we can have a distributed key-value cache like REDIS to store the metadata.

CONCLUSION

We developed a generic event analytics framework that would help application developers gather app insights in real time. We gave prime importance to the scalability features including cross region balancer and portability of our implementation. The event logs can be swapped with any streaming data. To demonstrate the generic use case of our RETINAA infrastructure, we in fact built an application on top of this infrastructure, NewsUP[22],[23]. RETINNA can be extended to gather finer granularity events such as per application per state click stats and region wise app usage in real time.

ACKNOWLEDGMENT

We thank the Advanced Distributed Systems course professor; Dr. Indranil Gupta, (Indy) and the TA, Mainak Ghosh, for their guidance and support throughout the course period and valuable review comments from fellow classmates, which helped us immensely to evolve the project.

REFERENCES

[1] http://parquet.apache.org/

[2] http://engineering.pinterest.com/post/111380432054/real-time-analytics-at-pinterest

[3] http://www.slideshare.net/DiscoverPinterest/singer-pinterests-logging-infrastructure

[4] http://techblog.netflix.com/2013/12/announcing-suro-backbone-of-netflixs.html

[5] http://www.vmware.com/products/vrealize-log-insight

[6] "Making machine data accessible and usable" http://www.splunk.com/

[7] http://www.splunk.com/en_us/products/splunk-mint/splunk-mint-express.html

[8] Simplify Log Management Forever https://www.loggly.com

[9] http://wiki.apache.org/hadoop/Chukwa

[10] Logscape : http://www.logscape.com/product.html#infrastructure-monitoring

[11] http://www.adremsoft.com/netcrunch/monitoring/alerts-and-logs

[12] https://engineering.twitter.com/research/publication/the-unified-logging-infrastructure-for-data-analytics-at-twitter

[13] http://vldb.org/pvldb/vol5/p1771_georgelee_vldb2012.pdf

[14] Apache Flume http://flume.apache.org/FlumeDeveloperGuide.html

[15] http://kafka.apache.org/

[16] https://github.com/apache/storm/tree/master/external/storm-kafka

[17] https://cloud.google.com/compute/docs/load-balancing/http/cross-region-example

[18] http://redis.io/

[19] https://storm.apache.org/

[20] http://druid.io/druid-powered.html

[21] https://www.mongodb.org/

[22] NewsUP : https://github.com/jalatif/NewsUp-News_Real_Time_Analysis_And_Generation

[23] Android Application : https://github.com/jalatif/NewsUp_Android_App

BUSINESS PLAN

Why RETINAA?

Application event logs are usually sent to the platform providers (such as Google or Apple) which the developers use to analyze. With our system, we create a direct link between user base and the application developer. Our system interface will help developers to view the top open bugs/crash and usage patterns and have drill downs per demography and gain insights into the app usage and problem areas of the application. The main advantage of our platform is the option to gather application specific stats as per developer's need rather than being bounded by the platform providers.

Who will want to use RETINAA?

Developers also will benefit from our aggregation and prioritization infrastructure to help reduce the development and debugging time especially when the platform support is very restrictive and proprietary. Working on highest priority bugs can help in stabilizing the app faster and enable faster TTM for new features. The system can also categorize reports on software version and other categories and provide tools for in depth analysis. This feature is not just restricted to crash analytics, developers can use our APIs to track event analytics also like click streams from the app.

How do we plan to capture the market?

Our product will be available as a beta product for app developers to use. Currently we will target the Android developers from Google Play Store. But in the future, our customer base can extend to anyone who requires to analyze streaming events and do analytics for gathering data insights.

How much will the service cost?

As the user-base grows, we plan to add more features and improve our codebase and make the features available as a paid service. Only registered developers who pay a one time small registration fee can use our product.

Cost per datapoint usage : Initially around 5 Million data points (which include error/warn counts, app crash stats, specific clickstream action feeds). We plan to have pay as you model and pull in our customer base by offering free trial for a week with unlimited data point capture.