IndyCar - Resolving latency issues of the message flow and implementing a data persistence layer

Arpit Bansal
Indiana University Bloomington
arbansal@iu.edu

Nishant Jain
Indiana University Bloomington
nishjain@iu.edu

1. INTRODUCTION

IndyCar is one of the top-level car racing series in America. During the period of each race, different kind of events happen, including pit stops, crashes, mechanical breakdown, drivers ranking changes, and that's only the tip of the iceberg. With the timing and scoring log data, the overall task is to detect events of interest which can be categorized as an anomaly detection task, and to predict whether this event will happen soon which can be categorized as a classification task.

IndyCar's initial prototype was developed using MQTT based pub/sub broker to interconnect individual modules. There are high latency issues incurred throughout the message flow in the current system as MQTT was designed as an extremely lightweight publish/subscribe messaging transport and is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. It is not designed to handle ingestion of massive data with high velocity. So, we are looking for an alternative technology to replace MQTT brokers.

Also, currently we don't have a data persistent layer attached to the main message flow. There is an immediate requirement of implementing a data persistence layer in order to achieve following goals:

a. Persist results of anomaly detection for offline analysis.
b. Persist historical data of racing events to support querying and analyzing.

This project consists of the following components:

1.1. Propose and justify an alternative message broker to integrate IndyCar modules: First, we would benchmark the performance of the existing broker and calculate timing information for the round-trip time of the message. Next, we would explore available message brokers (like RabbitMQ, Kafka, etc.), understand which environments and configurations they would they be well suited for and provide the best performance in, and then implement and benchmark them in the existing environment, comparing their performance. Once the use of such an alternate broker is justified, we need to replace existing publisher and consumer packages of IndyCar modules to support the new broker.

1.2. Implement the data persistence layer and develop modules to query and analyze historical data: We need to analyze the type of data we would be working on and see whether it is better to use a relational database instead of NoSQL. Again, we would run benchmarks on each technology to compare the performance in querying the data, handling resources, read-write speeds and which one is better to handle real time data. We need to research available options for a NoSQL (or SQL) database and understand their pros and cons in the current environment. The data persistence layer should be decoupled from the main message flow via the broker proposed in Step 1.

For both the problems, the tools being used should have active support (or online community in case of open source tool), ease of implementation and access, require low maintenance, low latency, cost effective, scalable in future, and justifiable in terms of resource utilization

Data Set: Data captured from the previous year's race will be utilized for analysis.

2. RELATED WORK

There have been numerous studies on comparison of different broker services like ActiveMQ, RabbitMQ and Kafka.

In one of the papers, ActiveMQ and RabbitMQ were compared based on the performance. *(Ionescu, 2015)* [3]

As per the conclusions, after testing the broker performance, it was seen that ActiveMQ is faster on message reception (the client sends the message to the broker). But the same tests showed that RabbitMQ is faster on producing messages (the client receiving messages from the broker) compared to ActiveMQ.

In another research work, the histories, design goals, and distinct features of RabbitMQ and Kafka are compared. *(Dobbelaere & Esmaili, 2017)* [5]

It was inferred that RabbitMQ offers flexible routing mechanism, using the exchanges/binding notions. It is much closer to the classic messaging systems and its main design goal is to handle messages in memory, and its queue logic is optimized for empty or nearly-empty queues. Kafka, on the other hand, is designed around a distributed commit log, aiming at high-throughput and consumers of varying speeds. In case of RabbitMQ, the difference between at most once and at least once delivery modes is not significant. For Kafka, on the other hand, latency increases about twice as large for the at least once mode. Additionally, if it needs to read from disk, its latency can grow by up to an order of magnitude. Hence, in order to conclude, it is suggested to consider all other dimensions of the problem including the architecture and requirements of the system being implemented.

We also went through few published articles for the options we had for storing our data. In one of these articles *(S. Chickerur et al., 2015)* [4] , a conventional relational database management system (RDBMS) - MySQL was compared with a NoSQL database (MongoDB) and it was found through experiments that selections, insertions, updations and deletions in MongoDB were much faster than those in MySQL. Moreover, the values (time taken in seconds) for these operations increase as the number of tuples in the database increase, hence demonstrating that not only is the performance of such NoSQL databases in real-world scenarios better, but the fact that they are distributed, horizontally scalable, schema-free, no join, provide easy replication support, have simple API and are eventually consistent make them more suitable for modern day applications.

Another article *(Bhardwaj, 2016)* [9] compared the performance of two such NoSQL Document Oriented Databases - MongoDB and CouchDB and it was found that the average response time and throughput of MongoDB for 100 samples was much better than that of CouchDB. Also, the document insertion rate in MongoDB is approximately 10 times better than Apache CouchDB under the stated conditions and environment, making MongoDB the best candidate for our needs and the existing architecture that we have.

3. ARCHITECTURE AND IMPLEMENTATION

We used juliet nodes to implement our architecture and perform our experimentation. We are using one node to host the Apache Apollo (ActiveMQ) or RabbitMQ message broker and the other node as publisher-subscriber system which is writing and listening to the same topic, and test the time taken to push data from publisher and receive data at a subscriber (round trip time). We record the time taken for each record to pass through and receive and write the results in a csv file.

Prerequisites:

- Java
- Maven
- IndyCar log files

Software Components:

1. Record Streamer: This component can be used to replay races at the real speed by reading log files.

2. MongoDB Loader: This component can be used to load log files to MongoDB tables.

3. Flux generator: This can be used to generate the flux template for storm topology

4. WebSocket Server: WebSockets based server, which acts as the backend for the web application
5. Web Application: ReactJS based web application that connects to WebSocket server to fetch and display events in real-time.

We begin testing the network using the previous years' (2018) IndyCar log files. We test by sending messages for 1, 8 and 33 cars and for 50,000 records in total to be able to test the robustness of system in handling high amount of data. As each race lasts for 3 hours, it takes that much time to complete a test for 1 car.

The output is written in a csv file containing all the latency times (in nanoseconds) for each message. We convert the nanoseconds to milliseconds first as that is our base of measurement. As it takes around 10,000 messages for the warm-up, we are using the remaining 40,000 records for all the analysis.

Below is the architecture used to test the pub-sub broker system as explained before.

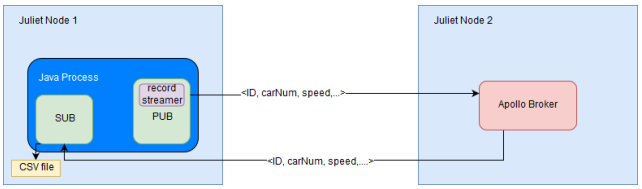


Figure 1: Broker testing architecture

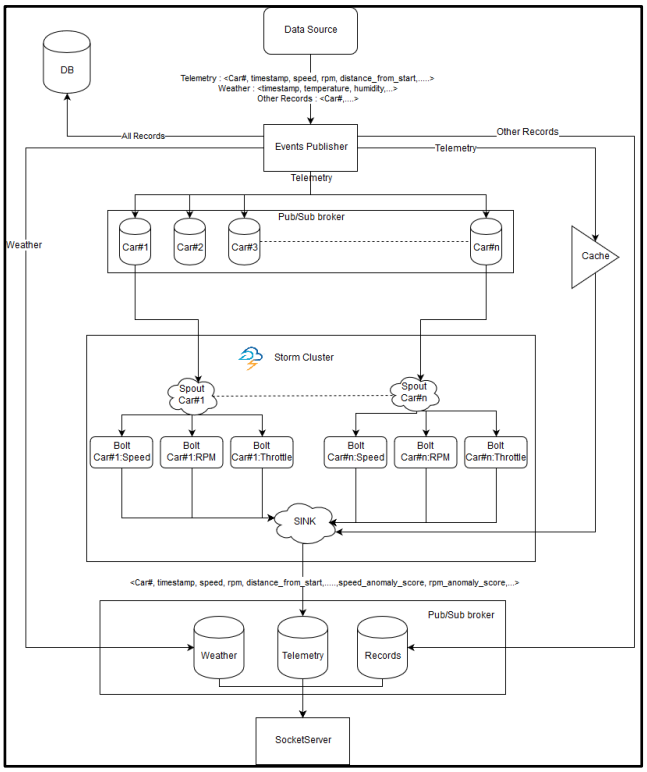


Figure 2: IndyCar Architecture

IndyCar project in overall has a bigger architecture consisting of several components including, broker system, data persistence layer (MongoDB), multiple data streams, and a storm cluster to run the HTM algorithm (anomaly detection). This is shown in figure 2.

The analysis involves calculating the following parameters: average, standard deviation, standard error, min and max of the latency times. We use the same architecture for both ActiveMQ and RabbitMQ.

We used SSH tunneling to access the dashboard (GUI) for both the broker services via Juliet node to check the live status of services, messages processed, queue details, memory usage and some other basic statistics. Snapshots of the details being monitored on the dashboard are given hereby for reference:

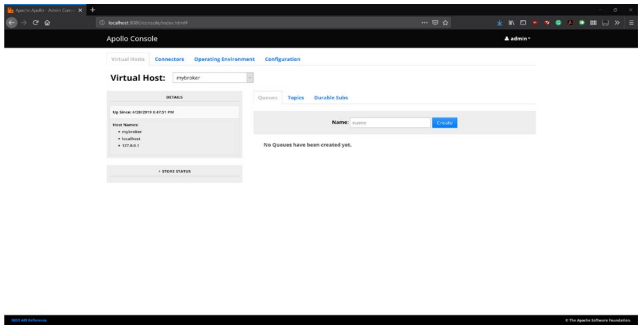


Figure 3: Snapshot of ActiveMQ web client

RabbitMQ dashboard features some additional features like live statistics reporting in the form of charts which gives a better representation when monitoring the broker service during test runs.

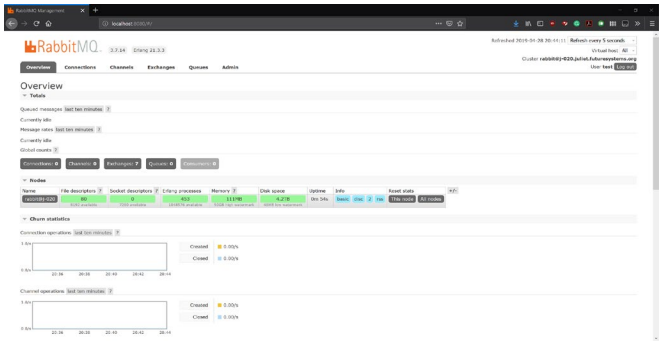


Figure 4: Snapshot of RabbitMQ web client

4. EXPERIMENT

The experiment involves setting up the above architecture on Juliet nodes and running it for both, ActiveMQ and RabbitMQ, to get the required results in csv format, and then perform the analysis as explained above.

Input Data: We used the latest log file "IPBroadcaster_Input_2018-05-27_0.log" for benchmarking. Below is a list of all the log files available from previous year race data, and all of them were used for the MongoDB experiments.

1. IPBroadcaster_Input_2018-05-15_0.log
2. IPBroadcaster_Input_2018-05-16_0.log
3. IPBroadcaster_Input_2018-05-17_0.log
4. IPBroadcaster_Input_2018-05-18_0.log
5. IPBroadcaster_Input_2018-05-19_0.log
6. IPBroadcaster_Input_2018-05-20_0.log
7. IPBroadcaster_Input_2018-05-21_0.log
8. IPBroadcaster_Input_2018-05-25_0.log
9. IPBroadcaster_Input_2018-05-27_0.log

We created an R Notebook using R MarkDown in Rstudio to analyze the results obtained in the csv files. We used "ggplot2" package to generate the charts. Below are the results obtained:

1. ActiveMQ

We observe that the almost all latency values are well below 1 ms. And the mean values are around ~0.5ms in all three cases. The variation is on a bit higher side and there are some outliers with very high latency values.

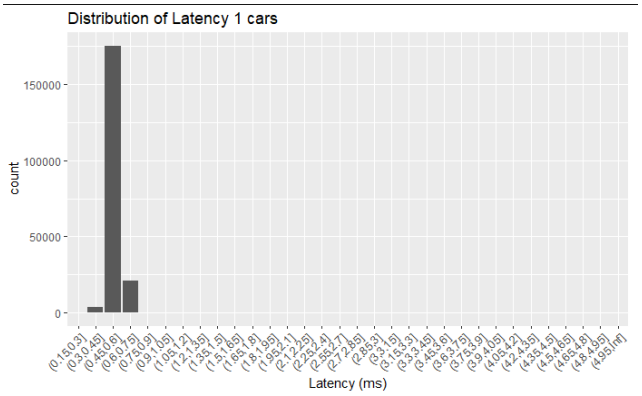


Chart 1: Latency distribution in milliseconds for 1 car using ActiveMQ

	Milliseconds
Mean	0.56
Std Dev	0.33
Std error	0.00
Min	0.29
Max	116.41

Table 1: Statistics for 1 car using ActiveMQ

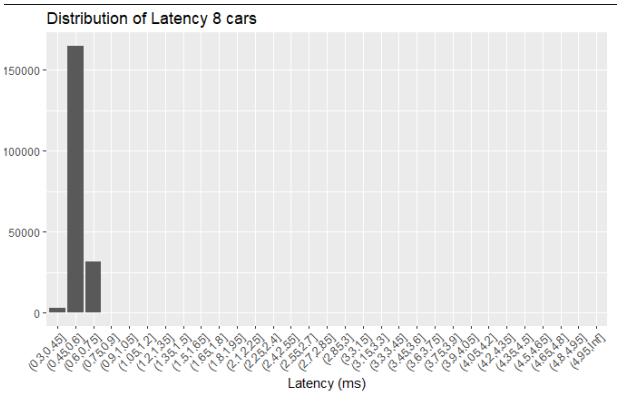


Chart 2: Latency distribution in milliseconds for 8 cars using ActiveMQ

	Milliseconds
Mean	0.57
Std Dev	0.35
Std error	0.00
Min	0.31
Max	87.98

Table 2: Statistics for 8 cars using ActiveMQ

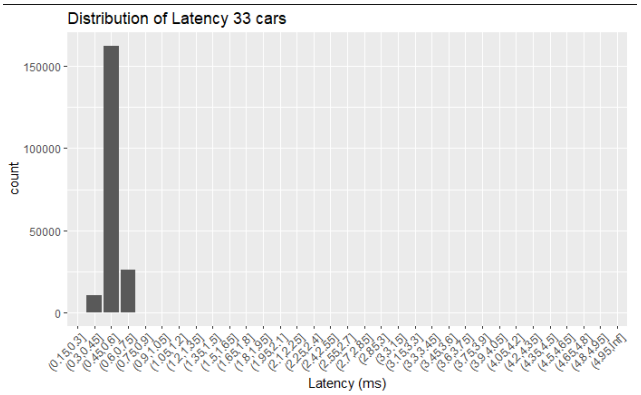


Chart 3: Latency distribution in milliseconds for 33 cars using ActiveMQ

	Milliseconds
Mean	0.55
Std Dev	0.33
Std error	0.00
Min	0.19
Max	82.5

Table 3: Statistics for 33 cars using ActiveMQ

2. RabbitMQ:

Again, we observe that the latency values are well below 1ms. But the mean values are around ~0.8ms in all three cases, which is not high as its still well under 1ms. Hence, there is no significant difference in mean latency time of ActiveMQ and RabbitMQ. For e.g., a difference of 1ms vs 5ms would have been considered as significant. The variation is lower as compared to ActiveMQ, hence this shows the robustness of RabbitMQ. Also during testing we found RabbitMQ to be more reliable as it didn't result in any crashes whereas there were some random crashes associated with ActiveMQ.

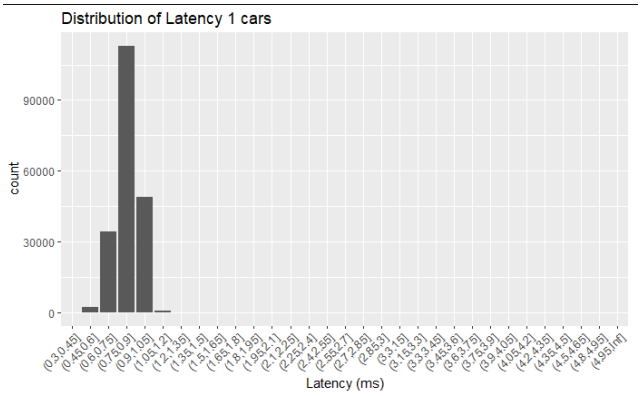


Chart 4: Latency distribution in milliseconds for 1 car using RabbitMQ

	Milliseconds
Mean	0.83
Std Dev	0.11
Std error	0.00
Min	0.41
Max	6.27

Table 4: Statistics for 1 car using RabbitMQ

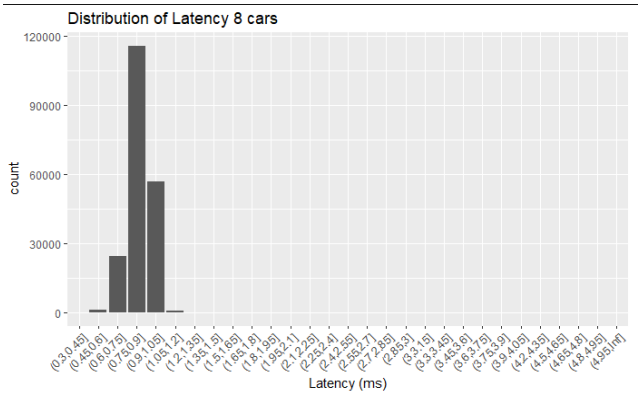


Chart 5: Latency distribution in milliseconds for 8 cars using RabbitMQ

	Milliseconds
Mean	0.85
Std Dev	0.32
Std error	0.00
Min	0.43
Max	100.48

Table 5: Statistics for 8 cars using RabbitMQ

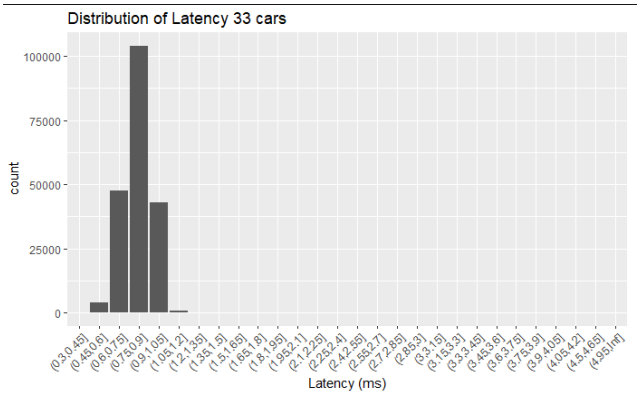


Chart 6: Latency distribution in milliseconds for 33 cars using RabbitMQ

	Milliseconds
Mean	0.82
Std Dev	0.25
Std error	0.00
Min	0.41
Max	71.1

Table 6: Statistics for 33 cars using RabbitMQ

3. Data persistence layer using MongoDB

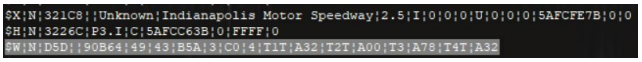
We installed a standalone instance of MongoDB (v4.0.6) and read log records available from a local folder to create our "Offline Data loader". The program written uses Java (v1.8.211) and mongo-java-driver (v3.4.3) to store and retrieve documents from the 8 collections that we create under this instance of MongoDB.

Load Data: Data is loaded from multiple log files listed at the start of this section. The java code loads all the log files by specifying the folder path containing all the log files.

Process Data: Following headers are present in the log files, and we process each of them to a separate collection in MongoDB:

1. $O - Overall results information
2. $E - Entry information
3. $R - Run information
4. $T - Track information
5. $F - Flag information
6. $P - Telemetry information
7. $S - Completed section results
8. $W - Weather information

Each field in the entry record in the log file is separated by "¦" operator. Hence, we split each record using that field as a delimiter/separator using necessary conditional operators and utilize the header information in the record to determine which collection does it belongs to. (For example, $P is used to retrieve a telemetry record from the log file and writes to the "telemetry" collection.). One small section of the log records is given hereby as an example:



We create 8 collections under a database named "indycar" - "driverinfo", "trackinfo", "runinfo", "trackinfo", "telemetry", "weatherinfo", "sectionresults", "overallresults".

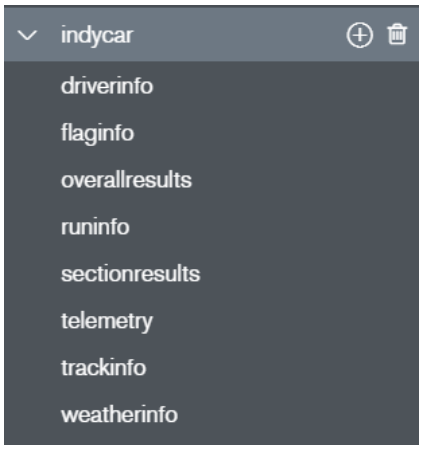


Figure 5: Snapshot of MongoDB Compass UI listing the collections

Post segregating each entry field using the delimiter and given header value, we refer the IndyCar timing data document to understand what information that field provides, and we next make an entry for the record as a new document in the MongoDB collection.We provide necessary "column_names" to avoid ambiguity and also convert the stored hexadecimal values in log files to standard values which are stored as a "String" data_type in the MongoDB collection, which make our data querying and retrieval process easier. We have also made use of hashmaps in Java to create in-memory dictionaries which make our data storage process easier. For each new log file read, we have used a new value for the "race_id" column so that information from separate log files can be easily recognized.

For reading all the log files given to us, our "MongoDriver" class takes around 10 minutes to store all data to the MongoDB instance.

Next, for data retrieval, we have hidden our database calls inside service classes which in turn call separate methods to fetch details like the following:

1. Drivers:

- List the names of all the drivers
- Search the driver by name. (eg. Typing Ca, should return Ed Carpenter, Carl, etc)
- Get set of drivers for a particular race
- Get the driver's profile
- Get Lap records and lap section records when driver name and race id is given

2. Races:

- List all races (just the name and id)
- Get all race metadata
- Get all the ranks of the racers in the race
- Get a snapshot of a race at given time. (Including speed, rpm, throttle etc. of each car)
- Get all the flags in a given race

3. Track:

- List all the tracks
- Get all the information of a track, when track name or id is given

(The data from the above functions is returned in the format of List<Document> or List<String>, whichever is most appropriate for the type of query being performed and expected results.)

This implementation serves as a Data API which can be imported at any other layer of the IndyCar application (Storm bolts, Dashboard Server, etc.).

We have created new service classes for each logical entity:

- DriversService.java will contain methods to query records related to Drivers
- RaceService.java will contain methods to query records related to race
- TrackService.java will contain methods to query records related to race

The functionality of the data API can be extended in future with the use of new classes and methods, depending on user requirements.

5. CONCLUSION

Broker Service:

Using MQTT protocol, the ActiveMQ and RabbitMQ message brokers provide similar performance, with ActiveMQ average latency being slightly better. Hence, in case of single consumer, Apache Apollo ActiveMQ should be chosen.

However, previous research from cited papers show that RabbitMQ is capable of producing messages at a faster rate and also our experiment results suggest that it is a more robust system. As these brokers will be used in conjunction with the data persistence layer (MongoDB), which will be a slow consumer, and there will be multiple consumers as per the proposed architecture, hence, we recommend using RabbitMQ instead of ActiveMQ.

Both ActiveMQ and RabbitMQ offer a lightweight and fast experience. Other brokers like Kafka could also be explored as they offer to better handle consumers of varying speeds, but that would be heavier in terms of resource utilization and might not be relevant for our use case.

Data Persistence:

We were successfully able to implement MongoDB for persistence of IndyCar race data. Though loading of data for the first time takes some time, but once the system is in place, adding more data to it would be faster as MongoDB is optimized for insertions and updates (in comparison to other NoSQL document databases) and also there will be a continuous feed of small data segments as compared to loading the whole data at once.

We were also able to create a custom Data API to handle and query data using Mongo Java Driver which perfectly integrates with current system components, all of which are also based on Java. Querying and retrieving data is very fast in MongoDB and it offers wide variety of inbuilt functionalities to implement different kind of queries. All the current queries have been implemented accurately and further queries can be added as per the requirements.

6. ACKNOWLEDGMENTS

We would like to thank Chathura Widanage for being our mentor through this project. Also, we would like to thank Prof. Judy Qiu and Selahattin Akkas for guiding us throughout the semester and helping us whenever required.

7. REFERENCES

[1] http://activemq.apache.org/slow-consumer-handling.html

[2] https://activemq.apache.org/apollo/documentation/getting-started.html

[3] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, Craiova, 2015, pp. 132-137

[4] S. Chickerur, A. Goudar and A. Kinnerkar, "Comparison of Relational Database with Document-Oriented Database (MongoDB) for Big Data Applications," *2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*, Jeju, 2015, pp. 41-47

[5] Dobbelaere, Philippe & Sheykh Esmaili, Kyumars. (2017). Kafka versus RabbitMQ

[6] https://www.rabbitmq.com/getstarted.html

[7] https://www.rabbitmq.com/install-generic-unix.html

[8] https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1

[9] Bhardwaj, Niteshwar. (2016). Comparative Study of CouchDB and MongoDB – NoSQL Document Oriented Databases. International Journal of Computer Applications. 136. 24-26. 10.5120/ijca2016908457.

8. APPENDIX

8.1. Work done on Message Brokers:

a. Research on Apollo ActiveMQ: Nishant Jain

b. Implementation\Testing of Apollo ActiveMQ: Arpit Bansal

c. Research on RabbitMQ: Arpit Bansal

d. Implementation\Testing of RabbitMQ: Nishant Jain

e. Setting up Java and configuration\execution of code on server for benchmarking: Arpit Bansal

f. Benchmarking (Analysis and Reporting) for Apollo ActiveMQ and RabbitMQ: Arpit Bansal, Nishant Jain

8.2. Work done on persistence layer (NoSQL\MongoDB):

a. Research: Arpit Bansal, Nishant Jain

b. Implementation and configuration: Arpit Bansal

c. Move historical data to the persistence layer for querying and analysis: Arpit Bansal

d. Testing: Nishant Jain

8.4. Final Report: Arpit Bansal, Nishant Jain

8.5. Final Presentation: Arpit Bansal, Nishant Jain