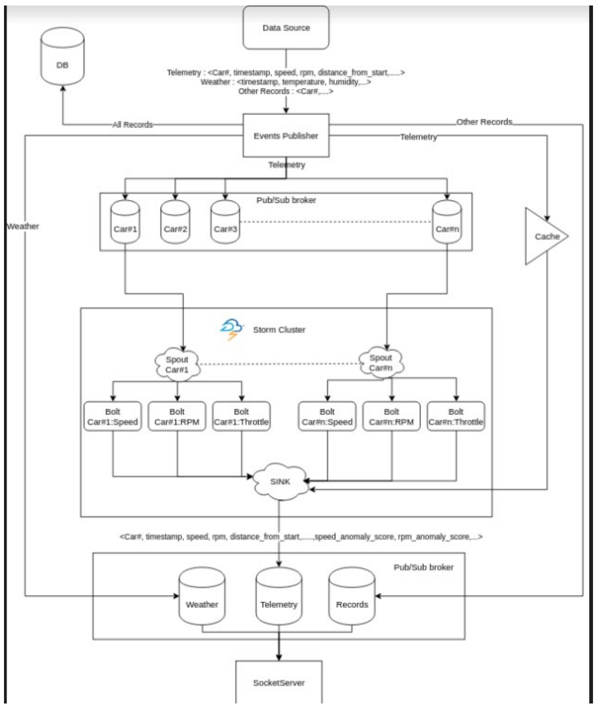
Resolving latency issues of the message flow and implementing the data persistence layer (IndyCar)

Ishneet Singh Arora
Graduate Student
Indiana University
United States
iarora@iu.edu

Sahaj Singh Maini
Graduate Student
Indiana University
United States
sahmaini@iu.edu

INTRODUCTION

Our project is contributing to the IndyCar project which is related to the real Indianapolis 500 race. The distance of a single lap in the race is of 500 miles. The number of cars that take part in the race is equal to 33, the race track contains 4 turns and the number of laps required for the race to be completed is equal to 200. Thus, for the race to be completed each car travels a distance of 100,000 miles. During the race there are a platitude of events that occur from time to time, for example-pitstops, crashes, mechanical breakdowns, changes in the rankings of drivers etc. With the help of the log data (in real time during the race), the IndyCar project aims to detect the events, which can be called anomaly detection and it aims to detect whether the event will happen soon, which can be called a classification task. The architecture of the Indycar project can be described by the below image-



Our role in this project is to find which message broker platform provides better latency performance and implement it for the flow of messages and to implement the data persistence layer. The two open-source message broker software platforms being used and compared in this project are Apache Apollo and RabbitMQ. Apache Apollo is written in Java along with a full Java Messaging Service Client and is developed by Apache Software Foundation whereas RabbitMQ is built on Open Telecom Platform, written in Erlang programming language and is developed by Pivotal Software. Both Apache Apollo and RabbitMQ are messaging technologies that are used to decouple processes and provide communication that is asynchronous.

The database technology being used in this project for implementing the data persistence layer is MongoDB which is a NoSQL database program and uses JSON like documents with schemata. In this project we will be using log files of all the races (including practice sessions) of Indy 500 series. These log files contain the raw log data that should be processed accordingly and later loaded into the database. The queueing, dequeuing and loading of data into the respective database takes place twice in entirety of the IndyCar project. Once when the data is initially streamed, and later after the anomaly detection section of the project is executed.

RELATED WORK

From the work done by Dobbelaere, P. & Esmaili, K. S[1], we can see that routing logic of RabbitMQ is much better than kafka while the time decoupling of Kafka is more efficient. In their work, they have provided information that RabbitMQ guarantees the delivery of packets while the same cannot be said about Apache Kafka. Both the brokers support replication which leads to increase in the availability. Apache Kafka makes use of Apache Zookeeper for scaling whereas in the case of RabbitMQ it is transparent to the end-user. Their work suggests that the latency performance of RabbitMQ is much better in comparison to Apache Kafka and also that the error rate in the case of Apache Kafka is much high compared to RabbitMQ. RabbitMQ is best suited for Pub/Sub Messaging, Request-Response Messaging, Operational Metrics Tracking, Underlying Layer for IoT Applications Platforms, Information-centric Networking etc. Whereas, use cases best suited for Apache Kafka are Pub/Sub Messaging, Scalable Ingestion System, Data-Layer Infrastructure, Capturing Change Feeds, Stream Processing.

According to the survey of distributed messaging queues conducted by Vineet John and Xia Liu[2], the Apache Kafka has higher throughput as compared to the RabbitMQ and RabbitMQ has lower latency as compared to the Apache Kafka. In this survey they have compared the performance and features of distributed messaging queues based on a particular solution. According to the survey, RabbitMQ is more reliable than Apache Kafka and RabbitMQ also has better security as compared to that of Apache Kafka.

Vidushi Jain and Aviral Upadhyay[3] in their work have compared MongoDB with traditional Databases and have listed its advantages and disadvantages. They have also listed details related to different kinds of NoSQL databases and mentioned major differences between SQL(Oracle) and NoSQL(MongoDB). One major observation made in their work was that SQL's (Oracle) performance in updating records was not as good as NoSQL (MongoDB) performance in updating records as the number of records kept increasing. According to their work NoSQL(MongoDB) provides better flexibility as compared to SQL(Oracle).

METHODOLOGY

While in actuality, the data is streamed in adherence to the real world events that take place as a part of the Indy 500 race in real-time. As there is no way to test the programs of this project in reality using real-time data, a program has already been written to stream data and simulate real-time data from log files that were created based on real-time data from Indy 500 races and practice sessions. This data is published to, and subscribed from, the message broker software platforms implemented in this project. The subscribed messages are then processed and stored in the MongoDB database, after the data is streamed, queued and dequeued. This happens for the first time when the data is streamed using the streamer program from the log files and the second time after the data is in the form of anomaly scores and is queued and dequeued post anomaly detection portion of the IndyCar project is executed.

In order to compare the results obtained from the different message broker software platform, we host a message broker software platform on one node and we publish as well as subscribe to the message broker software platform from the other node in order to calculate the latency. The latency is calculated at different times of the day to get assuring results. The latency is calculated for 50,000 messages which are queued to, and dequeued from, the respective messaging broker software platform. The results, which are the 50,000 latency values for the 50,000 messages that were sent and received are then stored and retrieved in a '.csv' file. The 50,000 latency values for the 50,000 messages are calculated multiple times for data being sent and received for different number of cars to the messaging broker software platform on the other node. All the latency values in a file are averaged and all the values received from averaging the '.csv' files created for 'n' number of cars are again averaged to find the average latency values for data sent and received to the messaging broker software platform on the other node for 'n' cars. We find the average latency values for data sent and received to and from the messaging broker software platform on the other node for '1' car, '8' cars and '33' cars. We will then compare the latency values to select the better messaging broker software platform as the solution to the problem.

Once, the messaging broker software platform for the IndyCar project is chosen based on better latency performance from all the experiments conducted as a part of this project. The streamer program that has already been developed will be used for streaming the data and sending it to the messaging broker software platform which will queue the data under two different topics. One topic to dequeue the messages to the Apache Storm element of the IndyCar project, and the other topic to dequeue the messages that are to be stored in the MongoDB database. The second time the messaging broker software platform will be used will be when the anomaly detection portion of the IndyCar project is executed and the anomaly scores are queued to the messaging software platform under two different topics, under one topic the messages will be dequeued to be sent to the client which will be displayed on a User Interface for viewing live streaming of data. Where as in the other topic the messages will be dequeued to be loaded into the MongoDB database for persistence.

ARCHITECTURE

Architecture of Message Broker Implementation

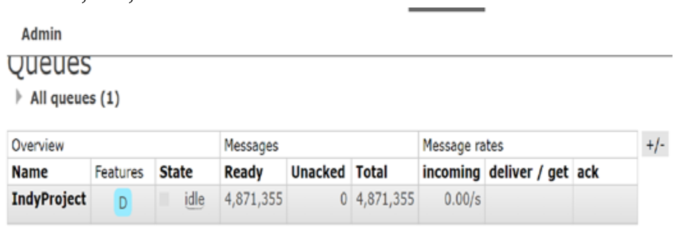
For the project, we implemented a real-time scenario which simulates the IndyCar Project. In the actual scenario, we will receive data in the form of messages on RabbitMQ queues. We will be subscribing to the RabbitMQ Queues and storing these messages into a MongoDB database for data persistence and querying.

For this project, we have used log files which contains the real data of the Indianapolis 500 race. We are reading the data from these files and sending each line of the log file which represents a record as a message to a RabbitMQ Queue. A consumer then subscribes to this queue and reads the messages, filters and preprocesses them before storing the messages into an array. After every minute, the messages from the queue are then sent for insertion into the database. For this particular setup, we have installed RabbitMQ on an Open Stack Server. We have also installed MongoDB on three OpenStack Servers. The data in the MongoDB database across the three servers is synchronized and replicated for better data availability as well as for making the system fault tolerant.

Configuration of the Servers which were used for setting up of RabbitMQ and MongoDB was –

Architecture: x86_64, CPU op-mode(s): 32-bit, 64-bit
CPU(s): 1, On-line CPU(s) list: 0, Thread(s) per core: 1
Core(s) per socket:1, Socket(s):1, Vendor ID: Genuine Intel, Model name: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

Fig 1 -Example of RabbitMQ Queue IndyProject having message count 4,871,355-



Admin

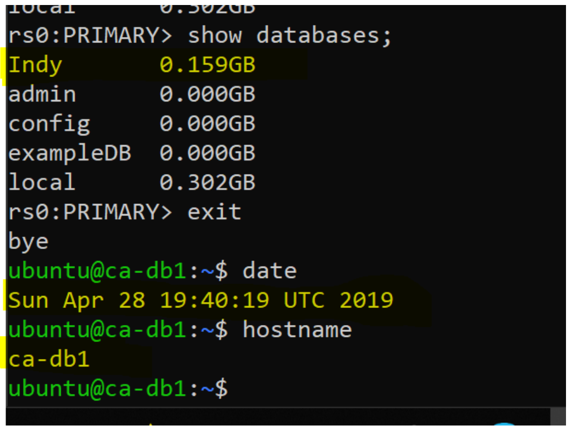
Queues

All queues (1)

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
IndyProject	D	idle	4,871,355	0	4,871,355	0.00/s			

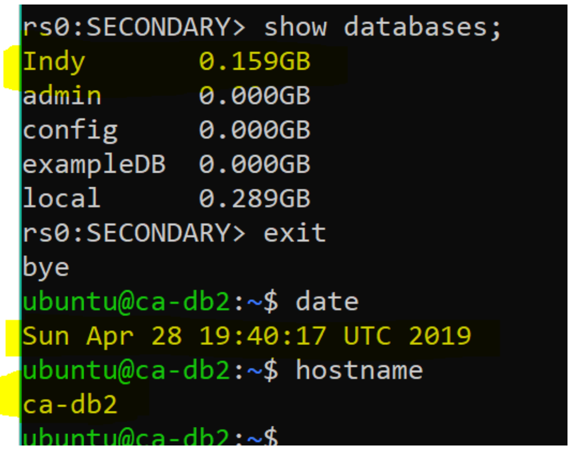
Fig 2a, 2b and 2c below show the increasing database size of Indy database that stores the relevant data in this project on all 3 nodes while the data is loading into the database after being received from RabbitMQ Queue.

Fig 2a showing the DB size while data is being loaded on Node 1-



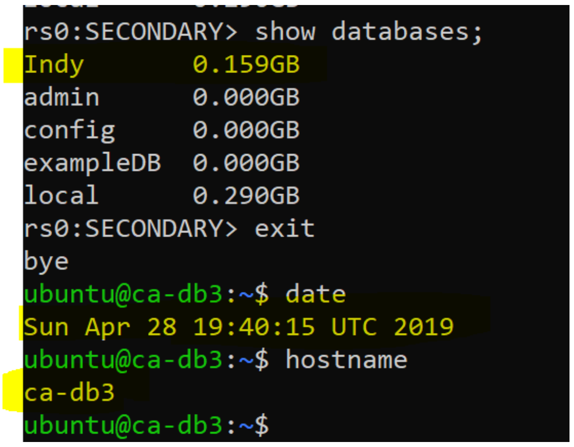
```
rs0:PRIMARY> show databases;
Indy         0.159GB
admin        0.000GB
config       0.000GB
exampleDB    0.000GB
local        0.302GB
rs0:PRIMARY> exit
bye
ubuntu@ca-db1:~$ date
Sun Apr 28 19:40:19 UTC 2019
ubuntu@ca-db1:~$ hostname
ca-db1
ubuntu@ca-db1:~$
```

Fig 2b showing the DB size while data is being loaded on Node 2-



```
rs0:SECONDARY> show databases;
Indy         0.159GB
admin        0.000GB
config       0.000GB
exampleDB    0.000GB
local        0.289GB
rs0:SECONDARY> exit
bye
ubuntu@ca-db2:~$ date
Sun Apr 28 19:40:17 UTC 2019
ubuntu@ca-db2:~$ hostname
ca-db2
ubuntu@ca-db2:~$
```

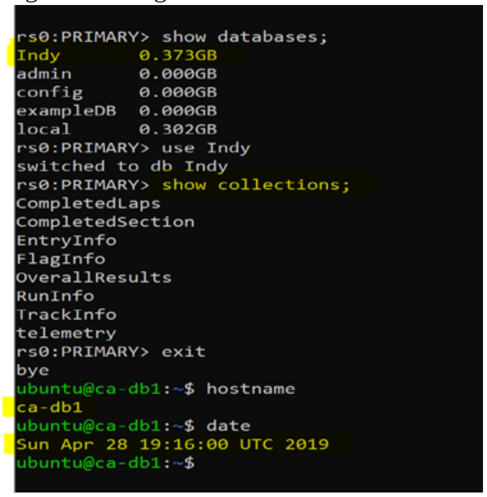
Fig 2c showing the DB size while data is being loaded on Node 3



```
rs0:SECONDARY> show databases;
Indy         0.159GB
admin        0.000GB
config       0.000GB
exampleDB    0.000GB
local        0.290GB
rs0:SECONDARY> exit
bye
ubuntu@ca-db3:~$ date
Sun Apr 28 19:40:15 UTC 2019
ubuntu@ca-db3:~$ hostname
ca-db3
ubuntu@ca-db3:~$
```

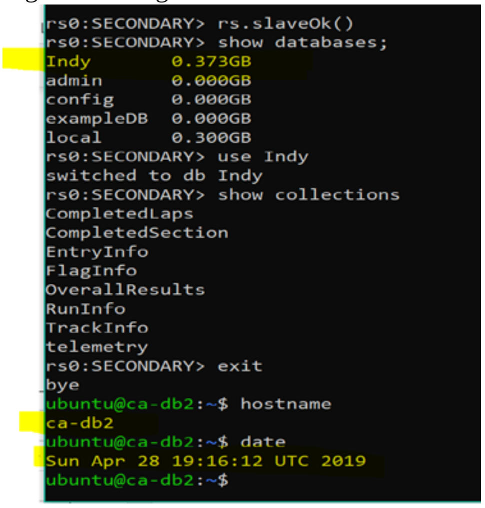
Fig 3a, 3b, 3c below show the same database size of Indy database on all 3 nodes after the Loading into DB is completed.

Fig 3a showing the DB size on Node 1



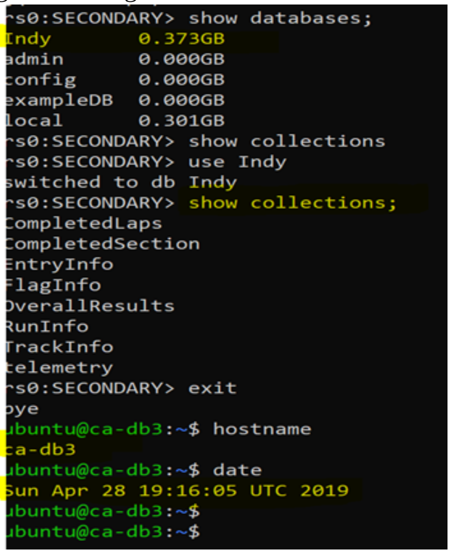
```
rs0:PRIMARY> show databases;
Indy         0.373GB
admin        0.000GB
config       0.000GB
exampleDB    0.000GB
local        0.302GB
rs0:PRIMARY> use Indy
switched to db Indy
rs0:PRIMARY> show collections;
CompletedLaps
CompletedSection
EntryInfo
FlagInfo
OverallResults
RunInfo
TrackInfo
telemetry
rs0:PRIMARY> exit
bye
ubuntu@ca-db1:~$ hostname
ca-db1
ubuntu@ca-db1:~$ date
Sun Apr 28 19:16:00 UTC 2019
ubuntu@ca-db1:~$
```

Fig 3b showing the DB size on node 2



```
rs0:SECONDARY> rs.slaveOk()
rs0:SECONDARY> show databases;
Indy         0.373GB
admin        0.000GB
config       0.000GB
exampleDB    0.000GB
local        0.300GB
rs0:SECONDARY> use Indy
switched to db Indy
rs0:SECONDARY> show collections
CompletedLaps
CompletedSection
EntryInfo
FlagInfo
OverallResults
RunInfo
TrackInfo
telemetry
rs0:SECONDARY> exit
bye
ubuntu@ca-db2:~$ hostname
ca-db2
ubuntu@ca-db2:~$ date
Sun Apr 28 19:16:12 UTC 2019
ubuntu@ca-db2:~$
```

Fig 3c showing the DB size on node 3-



```
s0:SECONDARY> show databases;
Indy        0.373GB
admin       0.000GB
config      0.000GB
exampleDB   0.000GB
local       0.301GB
s0:SECONDARY> show collections
s0:SECONDARY> use Indy
switched to db Indy
s0:SECONDARY> show collections;
CompletedLaps
CompletedSection
EntryInfo
FlagInfo
OverallResults
RunInfo
TrackInfo
telemetry
s0:SECONDARY> exit
bye
ubuntu@ca-db3:~$ hostname
ca-db3
ubuntu@ca-db3:~$ date
Sun Apr 28 19:16:05 UTC 2019
ubuntu@ca-db3:~$
ubuntu@ca-db3:~$
```

Architecture of Database Implementation

The messages from the messaging broker are pre-processed and arranged in the required order before being inserted in the respective MongoDB database. The version of the Mongo-Java used for the project is mongo-java-driver-3.7.1. The data is divided and stored in multiple collections depending upon the type of record that is received from the messaging broker and the information that the record contains. The record is inserted into its respective collection along with the race id of the race that the record contains information about. Every section in a record forms a field in the MongoDB document that is then inserted in the respective MongoDB collection.

The current implemented Database for the persistence layer contains eight different collections which are CompletedLaps, CompletedSection, EntryInfo, FlagInfo, OverallResults, RunInfo, TrackInfo, and telemetry.

The CompletedLaps collection contains the $C records(Completed Lap Results) and it shows car number, completed laps, elapsed time, last lap time, lap status, best lap time, best lap, time behind the leader, laps behind the leader, overall rank, current status, track status, pit stop counts, last pitted lap, start position, laps led, event name, run command that shows the status of the run, and race id (which is extracted from $R-run information records).

The CompletedSection collection contains $S records (Completed Section Results) that show car number, section identifier, elapsed time, last section time, last lap number, event name (Sweeps, Race, etc.), run command, race id and an unrelated counter that has been manually created and inserted in the collection along with the record and this can be used to sort the records in the order of which they were inserted for various query implementations.

The EntryInfo collection contains $E records (Entry Information) that show car number, unique identifier, driver name, start position, driver id, transponder control, equipment, license, team, team_id, engine, entrant id, points eligibility, hometown, competitor id, and race id.

The FlagInfo collection contains $F records (Flag Information) that show track status, lap number of track status change, green flag time, green flag laps, yellow flag time, yellow flag laps, red flag time, number of yellow flags, current leader, number of lead changes, average race speed, race id, and a counter which is inserted along with every record to sort according to the order of insertion in the collection for various queries.

The OverallResults collection contains the $O records (Overall Results) that belong to a group of enhanced results protocol (eRP) record type. This record is a level 2 record, which means that this is received only upon the occurrence of the event. The relevant fields in this record are inserted in the collection as a document along with race id of the race the record belongs to.

The RunInfo collection contains the $R record (Run Information) that show the name of the run, run command, start time and date, official name of the run and event round name.

The TrackInfo collection contains the $T record (Track Information) that show the track name of the racing track, length of the track, number of sections in the track, race id and the section information related to all the sections on the given track.

The telemetry collection contains the $P record (telemetry) that show the time of day, vehicle speed, engine rpm, throttle, date, and race id. Along with the record information we have also inserted ratio which has been defined as a geolocation index to find the closest $P value according to time, flag information extracted from previous $H record.

The queries implemented for the project can be divided into three modules that are DriverService, TrackService, and RaceService and all the functions return an ArrayList datatype in java containing the required information.

The DriverService module consists of five functions that are getAll, getDriversByRace, getdriver, search, getLapRecords. The getAll function returns all driver names in the series along with the driver id. The getDriversByRace function returns all the driver names and their driver id for a respective race. The getdriver function returns the driver details given the driver id that includes car number, engine, team name, etc. The search query uses regular expressions to search for names with the string fed to it as an argument which may be a partial name of the driver. The getLapRecords function returns two lists, the first one is the lap records for the given car number and the second list contains the section records. The DriverService module uses data from CompletedLaps, CompletedSection, and EntryInfo collections.

The RaceService module consists of five functions that are getAll, getRace, getFlags, getRanks, and getsnapshot_final. The getAll function returns all the event names along with race id. The getRace function returns the details like event name, event round, start date, time, etc. for a given race. The getFlags function returns the flag status (track status) at every timestamp that we receive $F (Flag Information record) during the race (Not during the warm-up sessions) along with the current lap number and counter that specifies the order in which the record was inserted. The getRanks function returns a list that contains the final race rankings of the race along with driver names. The getsnapshot_final function returns the snapshot of the race at a given time. When a timestamp is specified for the getsnapshot_final function it finds the nearest records for all the cars within the specified threshold value specified using the threshold_in_seconds argument using geolocation indexing in MongoDB which in this project has been performed on the ratio field of telemetry collection documents. The RaceService module uses data from RunInfo, FlagInfo, telemetry, OverallResults, and EntryInfo collections.

The TrackService module consists of two functions that are getAll and getTrack. The getAll function returns all the track names in the series and the getTrack function, given the track name returns the details regarding the track like the number of sections, and information pertaining to each section in the track.

EXPERIMENTS

RabbitMQ and Apache Apollo Benchmarks-

Here we have provided all the results of Latency Calculation for 50,000 messages which were calculated for Apache Apollo and RabbitMQ. For each number of cars (1, 8, 33) we have run the instance five times and then have calculated the average latency for ‘n’ number of cars that the messages were sent, Standard Deviation of all latency values for ‘n’ number of cars, Standard Error of all latency values for ‘n’ number of cars and have also calculated the Maximum and Minimum latency for all latency values of ‘n’ number of cars where n is equal to either 1, 8, or 33.

We have run this experiment on Juliet nodes which were provided to us. As we can see from the below results, the performance of Apollo MQTT was better than Rabbit MQ for a single subscriber. We have used the same procedure, files and Protocol-MQTT during the tests.

The configuration details of Juliet nodes are - Architecture: x86_64,CPU op-mode(s):32-bit, 64-bit, CPU(s):48,On-line CPU(s) list: 0-47, Thread(s) per core: 2, Core(s) per socket: 12, Socket(s): 2, Vendor ID: Genuine Intel, Model name: Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz

Fig 4-Displaying Error Bars in RabbitMQ and Apollo MQTT

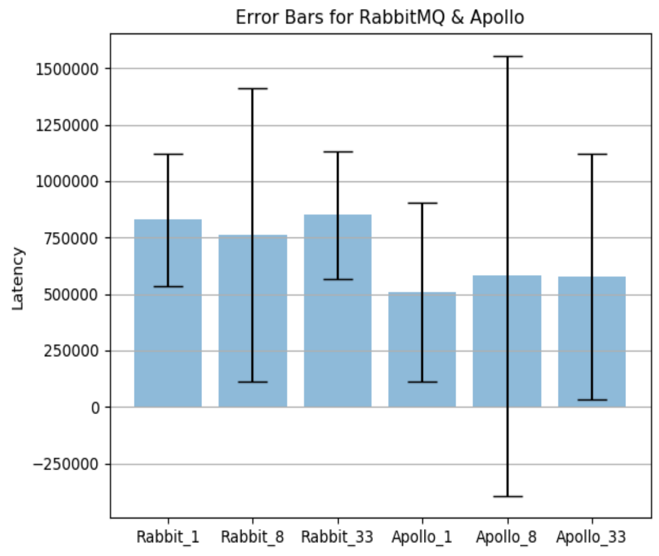


Fig 5- Mean of Latencies calculated during the experiments

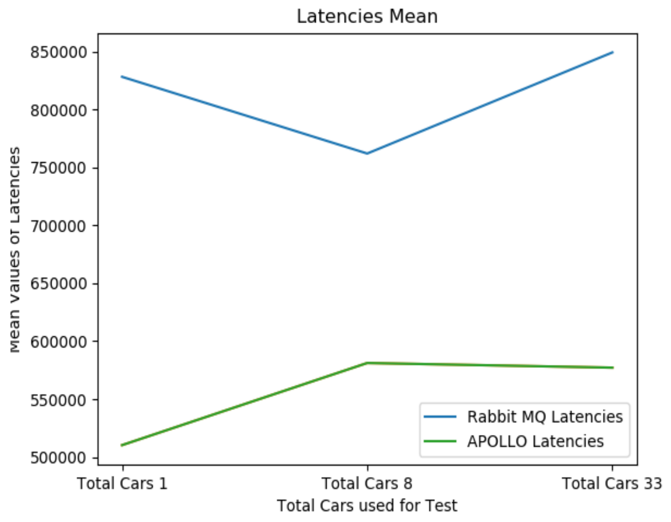
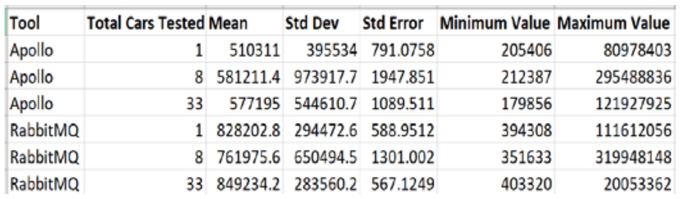


Fig 6 shows the Mean, Standard Deviation, Standard Error, Minimum Value, Maximum Value of the Latencies in Nano Seconds.



Tool	Total Cars Tested	Mean	Std Dev	Std Error	Minimum Value	Maximum Value
Apollo	1	510311	395534	791.0758	205406	80978403
Apollo	8	581211.4	973917.7	1947.851	212387	295488836
Apollo	33	577195	544610.7	1089.511	179856	121927925
RabbitMQ	1	828202.8	294472.6	588.9512	394308	111612056
RabbitMQ	8	761975.6	650494.5	1301.002	351633	319948148
RabbitMQ	33	849234.2	283560.2	567.1249	403320	20053362

Fig 7 -Histogram of Latencies for RabbitMQ for Car Number =1

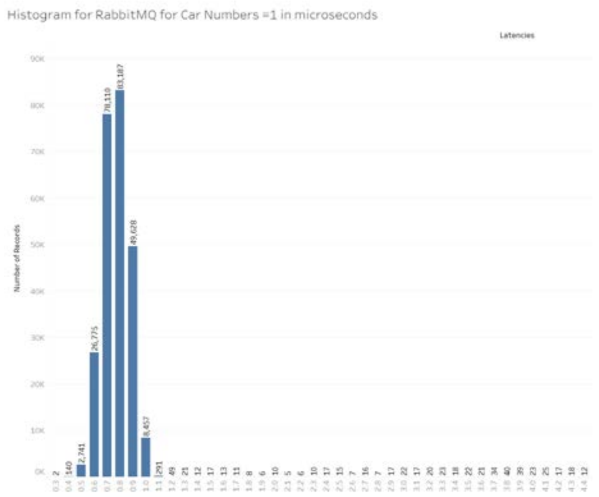


Fig 10-Histogram of Latencies for APOLLO for Car Number =1

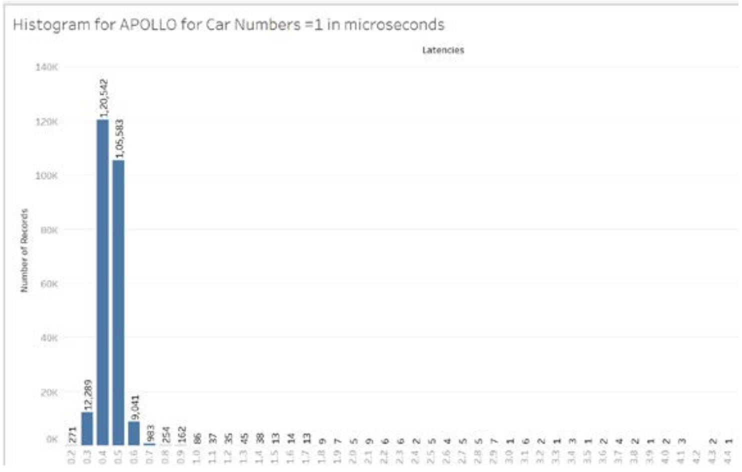


Fig 8-Histogram of Latencies for RabbitMQ for Car Number =8

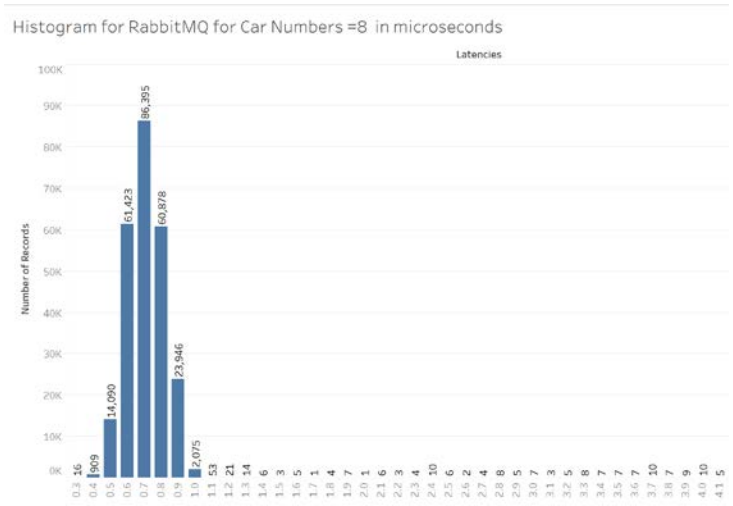


Fig 11 Histogram of Latencies for APOLLO for Car Number =8

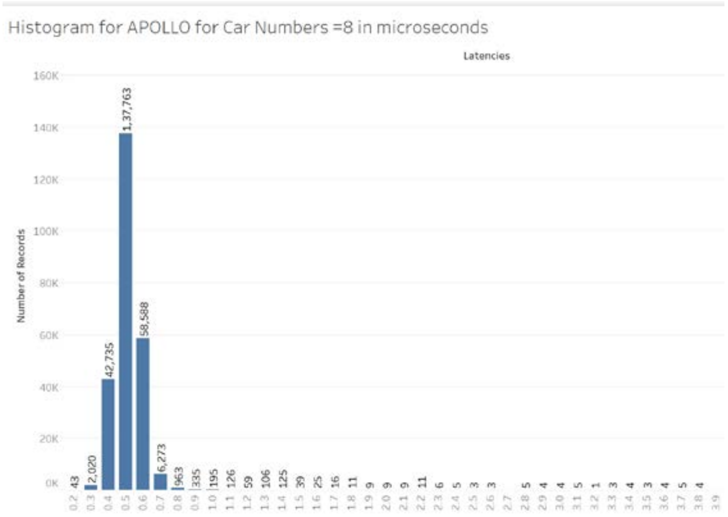


Fig 9-Histogram of Latencies for RabbitMQ for Car Number =33

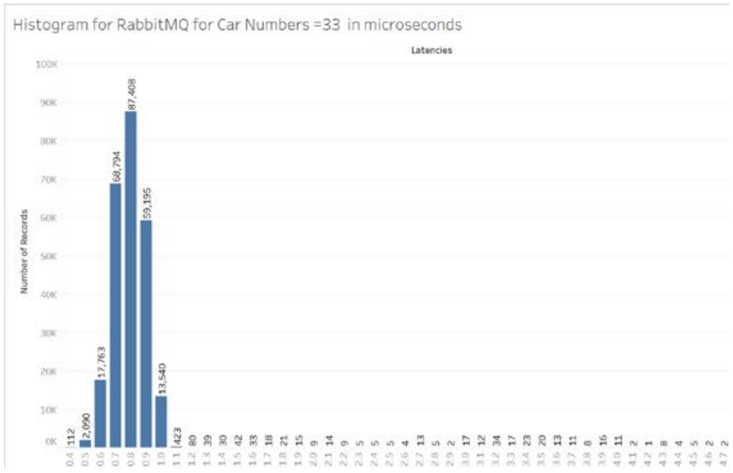
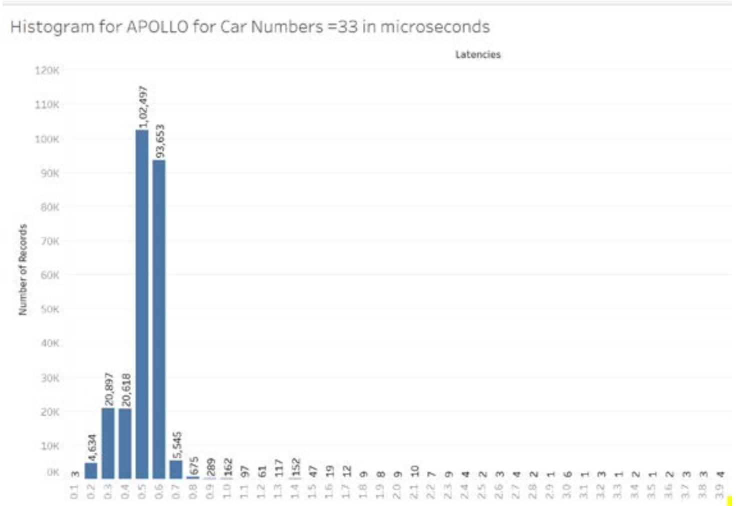


Fig 12-Histogram of Latencies for APOLLO for Car Number =33



MongoDB Queries

Demo implementation of MongoDB queries and their results.

- Suppose there are two drivers in the series, the getAll function from the DriverService module-
  indycarDBClient.drivers().getAll()
  Output-
  [["driver_name" : "Conor Daly", "driver_id" : "580"],["driver_name":"Alexander Rossi", "driver_id" : "522"]]

- Suppose there are two drivers in a race, the getDriversByRace function from the DriverService module-
  parameters- int race_id
  indycarDBClient.drivers().getDriversByRace(1)
  Output-
  [["driver_name" : "Conor Daly", "driver_id" : "580"],["driver_name":"Alexander Rossi", "driver_id" : "522"]]

- getdriver function from the DriverService module-
  parameters- String driver_id
  indycarDBClient.drivers().getdriver("747")
  Output-
  [["driver_name" : "Gabby Chaves", "driver_id" : "747", "car_number" : "88", "engine" : "Chevy", "team" : "Harding Racing", "team_id" : "145", "hometown" : "Bogota Colombia", "unique_identifier" : "15"]]

- search function from the DriverService module-
  parameters- String partial_string
  indycarDBClient.drivers().search("sa"))
  Output-
  [Takuma Sato, Sage Karam]

- getLapRecords function in DriverService module-
  parameters- int race_id, String car_num
  indycarDBClient.drivers().getLapRecords(1,"23");
  Output-
  [[["car_num" : "23", "completed_laps" : 0, "elapsed_time" : "5EC0", "last_laptime" : "C5839", "lap_status" : "T", "best_lap_time" : "0", "best_lap" : "0", "current_status" : "Active", "pit_stop_counts" : "0", "last_pitted_lap" : 0, "start_position" : 15, "laps_led" : "0", "run_command" : "R", "race_id" : "1"], . . .(more lap records)]
  [["car_num" : "23", "section_identifier" : "T1", "elapsed_time" : "1A535769", "last_section_time" : "3615", "last_lap" : 0, "run_command" : "R", "race_id" : "1", "unrelated_counter" : 62],...(more section records)]]

- getAll function in RaceService module-
  indycarDBClient.getRaceService().getAll()
  Output-
  [["event_name" : "102nd Indianapolis 500", "race_id" : "1"]]

- getRace function in RaceService module-
  parameters- int race_id
  indycarDBClient.getRaceService().getRace(1)
  Output-
  [["event_name" : "102nd Indianapolis 500", "event_round" : "IMS", "start_time_date" : "5B0AA2AC", "race_id" : "1"]]

- getFlags function from RaceService module-
  parameters- int race_id
  indycarDBClient.getRaceService().getFlags(1)
  Output-
  [["track_status" : "G", "lap_number" : "0", "timeOfDay" : "16:04:19.", "counter_$F" : 101], ... (more FlagInfo records)]

- getRanks function from RaceService module-
  parameters- int race_id
  indycarDBClient.getRaceService().getRanks(1)
  Output-
  [["overall_rank" : 1, "no" : "12", "first_name" : "Will", "last_name" : "Power"], ["overall_rank" : 2, "no" : "20", "first_name" : "Ed", "last_name" : "Carpenter"], ... (more records)]

- getsnapshot_final function in RaceService module-
  parameters- int hours, int minutes, double seconds, int race_id,int threshold_in_seconds
  indycarDBClient.getRaceService().getsnapshot_final(16,24,0.5,1,1)
  Output-
  [["$oid" : "5cc207e3f1da2e31c4bc7bc3" }, "command" : "$P", "car_num" : "9", "lap_distance" : "986.14", "time_of_day" : "16:24:00.526", "hours" : 16, "minutes" : 24, "seconds" : 0.526, "time_in_seconds" : 59040.526, "ratio" : [118.081052, 0.0], "time_for_comparison" : 1, "vehicle_speed" : "215.970", "engine_rpm" : "11267", "throttle" : "5", "date" : "2018-05-27", "race_id" : "1", "flag_from_$h" : "G", "run_command" : "R], ...(more records)]

- getAll function in TrackService module-
  indycarDBClient.getTrackService().getAll()
  Output-
  [Indianapolis Motor Speedway]

- getTrack function in TrackService module-
  parameters- String TrackName
  indycarDBClient.getTrackService().getTrack("Indianapolis Motor Speedway")
  Output-
  [["track_name" : "Indianapolis Motor Speedway", "venue" : "I",

"number_of_sections" : "20", "race_id" : "1", "section_information" : [{ "section_name" : "S1", "section_length" : "11856", "section_start" : "SF", "section_end" : "T1" }, { "section_name" : "S2A", "section_length" : "19800", "section_start" : "T1", "section_end" : "SS1" }, { "section_name" : "S2B", "section_length" : "19800", "section_start" : "SS1", "section_end" : "T2" }, { "section_name" : "S3A", "section_length" : "19800", "section_start" : "T2", "section_end" : "BS" }, { "section_name" : "S3B", "section_length" : "19800", "section_start" : "BS", "section_end" : "T3" }, { "section_name" : "S4A", "section_length" : "19800", "section_start" : "T3", "section_end" : "SS2" }, { "section_name" : "S4B", "section_length" : "19800", "section_start" : "SS2", "section_end" : "T4" }, { "section_name" : "S5A", "section_length" : "12672", "section_start" : "T4", "section_end" : "FS" }, { "section_name" : "S5B", "section_length" : "15072", "section_start" : "FS", "section_end" : "SF" }, { "section_name" : "S2", "section_length" : "39600", "section_start" : "T1", "section_end" : "T2" }, { "section_name" : "S3", "section_length" : "39600", "section_start" : "T2", "section_end" : "T3" }, { "section_name" : "S4", "section_length" : "39600", "section_start" : "T3", "section_end" : "T4" }, { "section_name" : "S5", "section_length" : "27744", "section_start" : "T4", "section_end" : "SF" }, { "section_name" : "T1", "section_length" : "1440", "section_start" : "T1T", "section_end" : "T1" }, { "section_name" : "T2", "section_length" : "1440", "section_start" : "T2T", "section_end" : "T2" }, { "section_name" : "T3", "section_length" : "1440", "section_start" : "T3T", "section_end" : "T3" }, { "section_name" : "T4", "section_length" : "1440", "section_start" : "T4T", "section_end" : "T4" }, { "section_name" : "TSF", "section_length" : "1974", "section_start" : "SF", "section_end" : "SFT" }, { "section_name" : "WI1", "section_length" : "16800", "section_start" : "PT3", "section_end" : "PSS2" }, { "section_name" : "WI2", "section_length" : "16800", "section_start" : "PSS2", "section_end" : "PT4" }, { "section_name" : "WI3", "section_length" : "15078", "section_start" : "PT4", "section_end" : "PI" }, { "section_name" : "WO1", "section_length" : "16800", "section_start" : "PO", "section_end" : "PSS1" }, { "section_name" : "WO2", "section_length" : "16800", "section_start" : "PSS1", "section_end" : "PT2" }, { "section_name" : "P1", "section_length" : "12690", "section_start" : "PI", "section_end" : "SFP" }, { "section_name" : "P2", "section_length" : "11874", "section_start" : "SFP", "section_end" : "PO" }, { "section_name" : "PIC", "section_length" : "5712", "section_start" : "PIC", "section_end" : "PI" }, { "section_name" : "L2", "section_length" : "24564", "section_start" : "PI", "section_end" : "PO" }, { "section_name" : "L3", "section_length" : "146544", "section_start" : "PO", "section_end" : "SF" }, { "section_name" : "L4", "section_length" : "145734", "section_start" : "SF", "section_end" : "PI" }, { "section_name" : "L5", "section_length" : "59400", "section_start" : "PO", "section_end" : "BS" }, { "section_name" : "L6", "section_length" : "74478", "section_start" : "BS", "section_end" : "PI" }, { "section_name" : "L7", "section_length" : "133878", "section_start" : "PO", "section_end" : "PI" }]]]

CONCLUSION AND OBSERVATIONS

From the log file data that we have, we can observe that the $O records (Overall Results) are not well ordered. We can also observe an inconsistency in the time section of the $P (telemetry) record type records. On looking closely at the time given in the $P records and the $H records, we can observe that there is a time difference of four hours which we assume may be due to different time settings. Along with the above-mentioned observations, we would like to add that the track record remains the same throughout the race.

As per the experiments conducted for calculating the Latencies for both RabbitMQ and Apache Apollo, it can be seen from the graph results that latencies are lower in the case for Apache Apollo. We have conducted the experiments only for a single subscriber. But we aren’t sure about performance details in the case of multiple subscribers subscribing to a single Queue. This can be appropriate future work. Further investigation can be done regarding the performance details in the case of multiple subscribers to a single queue. Additionally, we have successfully implemented database replication as suggested just to show that the architecture design can be fault tolerant as well as scalable in the case increase in load on the system.

ACKNOWLEDGEMENTS

We would like to thank our mentor, Chathura Widanage, the assistant instructor, Selahattin Akkas, and our Professor, Judy Qiu for giving us the opportunity to contribute to the IndyCar project and guiding us throughout the intended term of the project.

REFERENCES

https://www.bell-labs.com/our-research/publications/297579/
https://dl.acm.org/citation.cfm?id=3093908
https://www.ijcaonline.org/archives/volume167/number10/jain-2017-ijca- 914385.pdf