Performance Benchmark of Apache Kafka on AWS

Dipika Pandey[1] and Sapna Pratik Gandhi[2]
Guided by Dr. Amy Apon[3]

Abstract— **Every day, huge volume of transactional and web data are continuously generated in form of streams and need to be analyzed online when they arrive. As we are moving from periodic data collection to real-time streaming of data, we seek a reliable and scalable streaming platform that can stream real-time data with high throughput and low latency. To build such streaming applications, organizations can deploy real-time messaging system such as Apache Kafka on AWS to build their system scalable and reliable. In this paper, we establish the right platform for discussion by evaluating the performance benchmark of Kafka on AWS. Our experimental results show that Apache Kafka on Amazon EC2 can turn out to be the right combination for its benefits.**

General Terms. Real-time streaming, Distributed System, Log Aggregators, AWS.

Keywords. throughput, latency, distributed.

I. INTRODUCTION

In a world of real-time data, online service providers like Facebook, LinkedIn process an enormous volume of data. For instance, Facebook has more than 1.32 billion active users generating 3 million gigabyte of data on a daily basis[10]. This data is typically classified as user's activity data and operational metric data. User activities include login, search, share, like, etc. while operational metrics cover parameters like latency, CPU, memory and disk utilization on each machine. Real time collection and analysis of these data makes it possible to offer user services such as advertisements, customized search results, recommendations and also helps in monitoring system health. This is only possible as we are able to find relevant information on a fast scale. In order to collect and transmit such huge volume of data, applications use integrated messaging system and log servers. Facebook's Scribe, Yahoo's Data Highway and Cloudera's Flume are few such systems. All traditional messaging systems are primarily designed with objective of collecting and loading the log data into a data warehouse or Hadoop for offline consumption. These traditional log messaging system are characterized with high-latency and high delivery success rate. In order to move from such high-latency batch system to a low-latency streaming system, we need a streaming engine that can stream and process unbounded sequence of data with low latency and high throughput.

School of Computing
[1]Dipika Pandey is Graduate Student of Computer Science, Clemson University, SC, USA 29634 dpandey@g.clemson.edu
[2]Sapna Pratik Gandhi is Graduate Student of Computer Science, Clemson University, SC, USA 29634 sapnapg@g.clemson.edu
[3]Dr. Amy Apon, is with Faculty of School of Computing, Clemson University, Clemson, SC 29634 USA aapon@clemson.edu

Apache Kafka is a distributed, open source publish/subscribe messaging system developed at LinkedIn[1].It combines the features of both traditional log aggregators and messaging systems. Kafka is designed for distributed high throughput systems and tends to work well, in comparison with conventional messaging system. In this paper, we deployed Apache Kafka on AWS(Amazon Web Service) to evaluate its performance by ingesting streaming data, during peak hours and off peak hours. AWS is a secure cloud service platform that provides data storage, compute power and content delivery facility along with other functionality to help organizations to grow and scale[13].Kafka acts as a messaging system where not only it plays the role of log aggregator but also provides real-time streaming of data with high throughput, scalability and performance.

II. RELATED WORK

A review of previous studies has been introduced in this section.The section involves characteristics of traditional log servers, deployment of Kafka at Linkedin followed by data delivery infrastructure to support connected vehicle applications.

A. Log aggregation and management in traditional log servers

Traditional log servers were primarily used to collect, aggregate and load copious amount of data into data warehouses. However, data collection and information retrieval from these log aggregators pose several challenges, affecting performance: (i) Specialized log aggregators such as Facebook's Scribe are designed for offline consumption of data where data warehouse does periodic log collection rather than continuous consumption of data[11]. (ii) These log servers consider reliability as major design criteria over throughput. Some log aggregators such as JMS(Java Message Service)[8], requires full TCP/IP round-trip for each message. With the increase in real-time applications, it is not a viable solution as the enormous amount of data and throughput are of paramount importance. (iii)These log servers work on static configuration and are weak in distributed support. Thus, it is difficult to partition and store messages on multiple nodes. With these challenges and real world applications that generates tremendous amount of data, we need a messaging system with low-latency and high throughput capabilities.

B. Kafka deployment and usage at LinkedIn

LinkedIn is using Kafka to move clickstream data from front-end users to back-end cloud servers to make services

more personalized for the subscribers. Each day, LinkedIn processes almost 10 million messages per day, during peak hours with an average of 172,000 messages per second[9]. LinkedIn supports 367 topics which cover user's activity(such as likes, comments and share pages) as well as operational data (such as log and traces). The largest topic adds an average of 92GB compressed messages and smallest adds an average of few hundred KB messages per day[9]. The conceptual view of Kafka deployment at LinkedIn is shown in Fig. 1. Front-end services generate log data and publish this data to Kafka brokers. LinkedIn uses hardware load balancer to evenly distribute the data to the set of brokers. LinkedIn has deployed a dedicated Kafka cluster for offline analysis which is located geographically near to the Hadoop cluster. Kafka instance which runs for offline analysis works as a consumer that pulls data from the Kafka cluster at live data centre. All the data that is pulled from the Kafka cluster to Hadoop cluster is stored to data warehouse for further analysis. Loading data from Kafka cluster to Hadoop cluster is done by implementing a special Kafka input format that allows MapReduce jobs to directly read from Kafka. Furthermore, MapReduce processes the raw data and compress it for future use. LinkedIn uses Avro as its serialization protocol to support schema evolution[1][12]. This protocol stores the id of its Avro schema and its serializable bytes in the payload. Therefore, it plays the critical role of enforcing the contract to ensure the compatibility between producers and consumers.

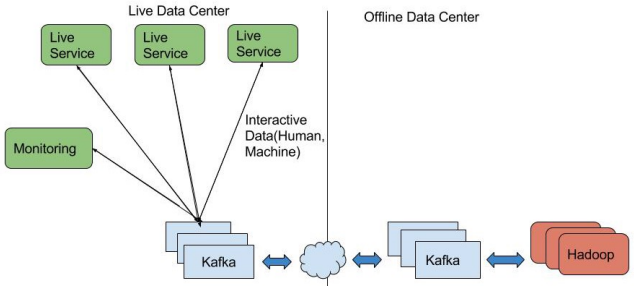

Fig. 1. Conceptual view of Kafka deployment at LinkedIn

C. Distributed Data Delivery infrastructure to support connected vehicle applications

In connected vehicle environment, various mobile devices, wireless networks and social media sources interact and generate massive amount of data. This will create bottlenecks in the analysis center as they are simultaneously trying to process and analyze data along with retrieving it from multiple sources. This study enforces on deploying additional hops at analysis center to process and stream data in a distributed environment. It deployed Kafka on Palmetto Cluster and Holocron in Clemson to serve the purpose of centralizing the communication between producer node and consumer node, with help of broker node. This paper focuses on running different experiments in terms of latency requirements for connected vehicle environment under different simulation scenarios. The results showed that measured latency is significantly less than the recommended latency by US Department of Transportation[3].

III. Physical and Application Architecture of Kafka

In this section, we will introduce Kafka architecture, its terminologies and application on AWS. A stream of messages of particular type is defined as 'Topic'. Producer applications publish messages to a dedicated topic. These published messages are then stored into a set of servers that act as multiple brokers. Consumer applications can subscribe to topic and pull data from the brokers[1]. Before subscribing to a topic, consumer needs to create one or more message streams. The topic will be evenly distributed into these sub streams. Kafka provides a stream iterator API in each message stream that enforces on producing continuous stream of messages. The consumer iterates over the messages and consumes the payload. If there are no more messages to consume, the iterator blocks until new messages come in. In comparison with traditional log aggregators, this messaging system provides continuous streaming of data[1][14].

The overall architecture of Kafka is shown in Fig. 2. As shown in figure, producer node publishes messages to topic. These published messages are stored into a set of servers called broker. Consumer application subscribes to topic and consumes the data from broker. The black dashed lines coordinate cluster membership and other represent consumer offset.

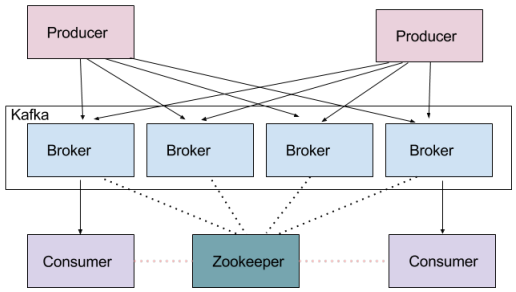

Fig. 2. Kafka Architecture

The rest of the section covers important terminologies of Kafka and role of Kafka in real-time streaming and analysis in AWS cloud.

Kafka has a storage layout where each partition is represented as a logical log. Unlike conventional messaging system, each message in the partition does not have an explicit message id[1]. Instead, each message is addressed by the **Logical Offset** in the log[1][14]. This way, we can avoid the random access indexes to map the message id to the location where messages are actually located. Consumer always consumes messages in a sequential way. If consumer acknowledges a specific offset then it signifies that it has consumed all the message prior to the offset in that partition. When consumer sends a pull request to the broker requesting it to keep the buffer ready to send the data, this pull request

consists of offset of the message from where the consumption will begin and an acceptable number of bytes to fetch. On the other side, the broker maintains a sorted list of offset, including the offset of the first message in every segment file. The broker finds the segment file by searching the offset list of the messages and sends the data to the consumer. After consumer receives the data, it calculates the offset of the next message to consume and uses it in next pull request to fetch data in the next iteration[1].

Kafka uses a term known as **Consumer Group**[1][14]. Each consumer group consists of several consumers that jointly consume the subscribed messages within a group. Each message is delivered to only one consumer in the group. Different consumer group consume a full set of messages and there is no coordination involved between them. All messages in the partition are consumed by only one consumer in the group to avoid coordination overhead. There is no master node involved in Kafka architecture. Instead, consumer nodes coordinate among themselves in a distributed way to reduce complexity. Since there are multiple consumers in group, each one will be notified when a broker or a consumer changes. However, notifications may come at different times to consumers. In that scenario, some consumer may try to take ownership for the messages that are already owned by other consumers. In this case, consumer releases those messages, waits and tries the re-balance process which may stabilize after two or three iterations.

Many distributed applications use **Zookeeper** i.e. a file system like API[1]. It acts as a configuration manager and keeps tracks of all the producers and consumers. In case of any failure, zookeeper detects the fail-over at the consumer or producer end. It also manages the re-balance between them and manages the consumption relationship when such events happen. Furthermore, it also keeps track of consumed offset of each partition in the topic.

In traditional messaging system, a server manages the clients by keeping information about them. However, in Kafka, the broker is not responsible for managing the producer and consumer as it acts as a **Stateless Broker**[1]. It does not keep information about producers and consumers. Instead, consumer itself manages this information. This design reduces the complexity and overhead on the broker[1][4]. However, this approach has a downside that the broker does not know whether consumers have consumed all the messages or not. To overcome this problem, Kafka uses a time-based SLA for the retention policy. A message will automatically be deleted if its retained more than the specified time, typically 7 days as mentioned in the retention policy[1]. This solution works well in practice as most servers including offline servers can consume data either in hourly, daily or in real time. Consumer can go back to the old offset and consume data again. For instance, if a consumer crashes and data is lost, it can checkpoint the smallest offset for those lost messages and re-consume the data from the offset. Checkpointing and rewinding back the consumer is much easier in the pull model rather than the push model[1].

In cloud computing, AWS is one of the largest cloud providers and Kafka is a distributed messaging system which fits well for building real-time streaming applications in AWS cloud. Data loss is significantly less in Apache Kafka due to its storage replication and the acks involved.

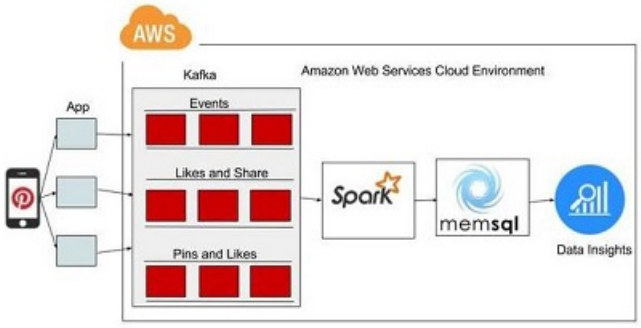

Fig. 3. All elements of real-time streaming and analytics in AWS Cloud.

As shown in Fig. 3, Real-time applications and web services generate data at massive scale which is being streamed to Kafka Cluster on AWS cloud[5]. This real-time streamed data is fed to Apache Spark for real-time processing and analyzing the data to get some insights, for instance, user behavior towards a specific product or service.

IV. Experimental Setup and Analysis of Results

This section focuses on methodology and results obtained from benchmarking experiments of Kafka on AWS. The results indicate the measured latency, throughput and impact of message size on Kafka's performance.

A. Platform Description

We ran our experiments on cloud platform provided by AWS. Overall, we used 8 EC2 m3xlarge amazon instances for running brokers, producers and consumers. The M3 instance type provides a balance between compute, memory, and network resources which proves to be a better fit for many applications.

Each EC2 m3xlarge machine has 4 vCPU, 15 GB of memory, 240 GB SSD storage for fast I/O performance, 2.5 GHz, Intel Xeon E5-2670v2 processors and high network performance i.e. the rate of data transfer[13].

We used Red Hat Enterprise Linux 7.3 (HVM) as a platform for installing software's such as JDK version 8 update 10, Zookeeper framework version 3.4.6 and Kafka version 0.9. Out of the 8 instances, we configured 9 brokers on 3 instances, deployed 20 producers on 1 and rest were used for deploying consumers.

B. Test Scenarios

We conducted 12 experimental scenarios for performance measurement such as latency and throughput with publishers publishing and consumers subscribing the data at the same time. Additionally, we also conducted 4 scenarios to know the impact of message size on performance of the system. The list of experiments conducted is outlined in TABLE I.

TABLE I
EXPERIMENTAL SCENARIOS

Scenario I: Performance of Data Streaming with Broker		
No. of Producers	No. of Brokers	No. of Consumers
20	3, 6, 9 3, 6, 9 3, 6, 9 3, 6, 9	10 20 30 40

Scenario II: Impact of Message size	
No. of Producers	Message Size(bytes)
20	10 100 1000 10000

As a producer node, EC2 instance runs 20 producers which is fixed for all the scenarios. These producers publish data for 10 different topics. We have varied cluster size of brokers in the count of 3, 6 and 9 for measuring the impact of performance with increase in number of brokers. Zookeeper is started on one of the broker instance for managing brokers, electing a new leader among them and providing fault tolerance. Each of the broker instance runs 3 brokers which are registered with the Zookeeper. For 3 brokers test case, only one instance is started and responsible for running 3 brokers. On the other hand, for 9 brokers test case, all the brokers are evenly deployed on 3 EC2 instances which are registered with Zookeeper.

As specified in the TABLE I, a number of consumers such as 10, 20, 30 and 40 are executed for subscribing data for each cluster size. For measuring the performance during load, we are considering an on-peak scenario in which 40 consumers try to consume data at the same time and an off-peak scenario in which only 10 consumers are consuming the data, reducing the load. Each EC2 instance is running only 10 consumers subscribing to data.

We have performed similar test cases as specified in paper [3]. But due to resource limitations, we have reduced the number of test cases and have performed a comparable set of scenarios. For this purpose, we created 20 publishers, each of them publishing 100,000 messages at the same time. All of these messages have fixed payload of 256 bytes and are published to 10 different topics.

The publishing of data requires a set of configuration which plays a crucial role for performance evaluation in Kafka. The Kafka producer publishes messages in terms of batches and uses a queuing system for better throughput. If send buffer memory is large enough than batch size then producer keeps buffering the data until the configured memory size. This results in high throughput but may lead to increase in latency. Thus, it is important to have an appropriate value based on the application requirement. We have configured the send buffer memory size to be twice the batch size where batch size is 8192 bytes as we are mainly focused on achieving low latency.

For scenario II in TABLE I, we are measuring the impact of message size that are being published to broker. In this scenario, we have deployed 20 producers on one EC2 instance. These producers are publishing data to 3 brokers with varied records size such as 10, 100, 1000 and 10000 bytes. We measured throughput and latency to evaluate the impact of message size.

C. Analysis of Experimental Results

The obtained results provide insights that Kafka's performance on AWS is comparatively acceptable as the average latency observed for 100,000 messages(each 250 bytes) is less than 1 sec. Indeed, in a peak hour scenario for 9 broker, it is approximately 6 ms. This delay can be due to the small message overhead of distributed log-message broker system when it is serving multiple data consumers simultaneously[3].

Kafka broker immediately writes the messages to file system when they are received which results in persistence storage of log data. These messages are not deleted on consumption of data. However, they are stored based on retention policy, as described in previous section. Unlike conventional system, where a single queue is maintained per consumer causing increase in data storage size with consumer, the persistence storage in Kafka allows to support space-efficient messaging system as there is only one single shared log regardless of number of consumers subscribing to data.

The experimental results for system are presented in the TABLE II. and Fig. 4, 5, 6, 7. These results represent the latency and throughput observed during this experiment.

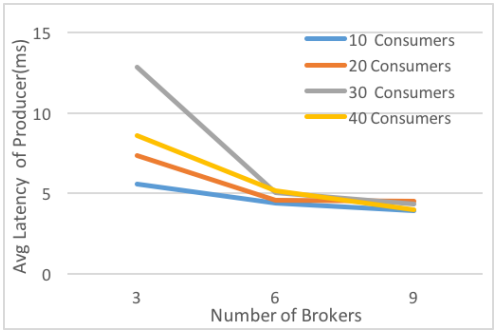

Fig. 4. Average latency of Producer with varying number of consumer and broker

As we can observe from Fig. 4 & 5, the latency for both producer and consumer decreases with increase in cluster size of broker. This is more significant at peak hours where 30 to 40 consumers subscribe for data at the same time. During peak hours, the average latency with cluster size of 3 brokers is approximately 12.8 ms. However, the same is reduced to 6 ms. with cluster of 9 brokers. This

TABLE II
EXPERIMENTAL RESULTS

No. of Consumers	Producer Latency (ms)	Consumer Latency(ms) (ms)	Producer Throughput (MB/s)	Consumer Throughput (MB/s)
No. of Brokers - 3				
10	5.594	3.1141	0.129	6.95447
20	7.346	3.7261	0.304	5.9227
30	12.8395	4.9005	0.094	6.9202
40	8.5695	4.2105	0.0855	6.7412
No. of Brokers - 6				
10	4.3845	2.8648	0.164	7.24931
20	4.587	3.2869	0.1475	8.77
30	5.077	2.9624	0.1415	7.3394
40	5.161	2.648	0.1425	8.3093
No. of Brokers - 9				
10	3.91125	1.9807	0.1825	7.9544
20	4.544	2.694	0.189	8.3784
30	4.3505	2.7275	0.1845	8.1012
40	3.9875	2.0325	0.18625	8.0826

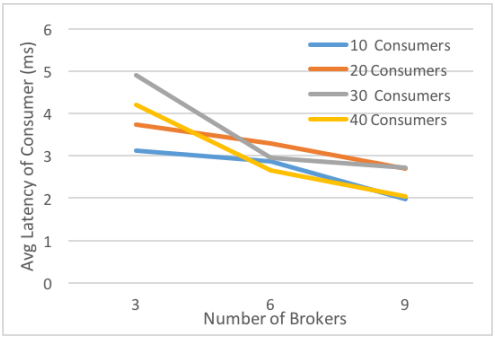

Fig. 5. Average latency of Consumer with varying number of consumer and broker

reduction in delay may be due to the data distributed across multiple partitions among brokers and consumers request being executed in parallel on multiple brokers. From Fig. 6 & 7, it can be inferred that both the producer and consumer throughput scales up with the increase in number of brokers.

During our experiment, we found that the latency at the consumer end is comparatively lesser than producer, if the consumer is consuming data while producer is producing the same. This coordinated fetching of data between both entities happen due to Kafka's efficient consumer implementation[7]. Kafka directly fetches chunks of log using consumer offset directly from the file system, as explained in "Consumer Offset" in previous section. It uses the sendfile API (Linux API) to transfer data directly through the operating system without the incurring the cost of overhead of copying or buffering the messages. But, at the same time, observed throughout of consumer is not of much significance as Kafka is sending the messages when it is available to it from publisher, where publisher is publishing data with low throughput(based on batch size and send buffer memory configuration).

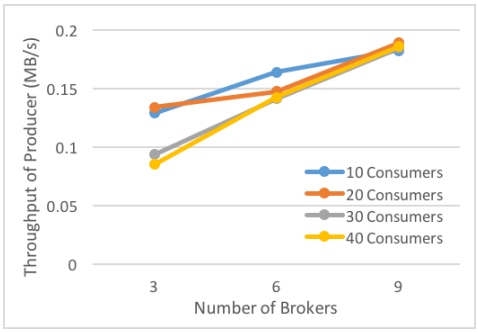

Fig. 6. Throughput of Producer with varying number of consumer and broker

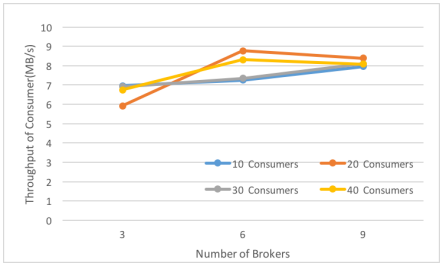

Fig. 7. Throughput of Consumer with varying number of consumer and broker

If data is consumed after the publishing is completed, it may lead to increase in latency as fetching may start at the beginning of the log. As the offset initially, is at 0 it results in performing a real read I/O operations. Although, in production environment, the data will be read data in real time which will be written by any producer and therefore it's still being cached. Due to this, the consumer will read mostly out of the OS pagecache. In the former case(after publishing is done), the throughput observed by the consumer will be greater than the latter one [7]. We can speculate that in the latter case the producer is publishing data in batches which are being fetched by consumer. However, in the former case, the data is published and more number of bytes are read by consumer per second.

The performance result produced on AWS has similar behavior to the results presented in paper[3]. Yet, there is difference in the latency and throughput produced. The latency difference can be seen in Fig. 8. The latency produced is more in AWS compared to Palmetto/Holocron might be due to the uncontrolled environment and hardware difference of AWS. Even with these differences, observed latency of 8.5695 ms. with 3 brokers and 3.9875 ms. with 9 brokers for 100,000 messages(256 bytes each) published by 20 producers and consumed by 40 consumers is an acceptable metric.

Finally, we analyzed the impact of message size on Kafka's performance with varying message size from 10,

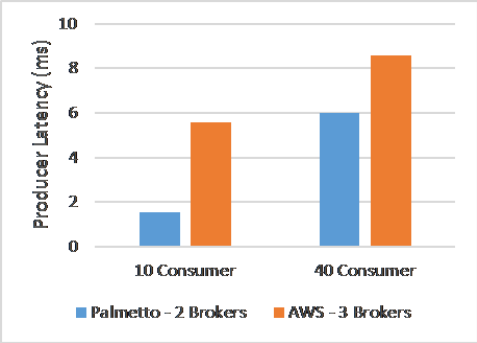

Fig. 8. Comparision between Latency observed on Palmetto and AWS

100, 1000 to 10000 bytes. From the graphs in Fig. 9 & 10, we learned that with the increase in message size, the number of records/sec(records sent per sec) decreases though throughput increases. Kafka sends messages in batches(records), where the batch size is configurable. With small message size, the data is sent in form of batches which results in less number of bytes being sent but more number of records actually sent. In this case, the data is not queued for longer duration which results in low latency. For larger message size, the data is getting en-queued for defined batch size which leads to increase in latency and increase in throughput as number of bytes being sent are more (in less number of records).

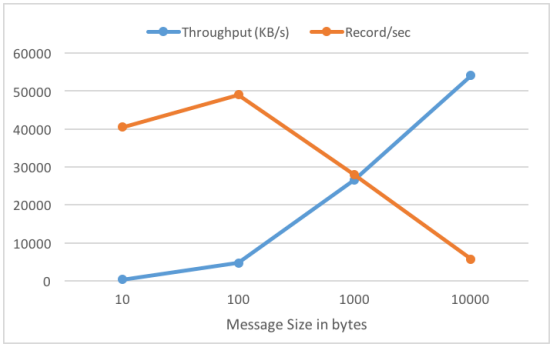

Fig. 9. Comparison between Throughput(KB/s) and Records/sec with varying message size

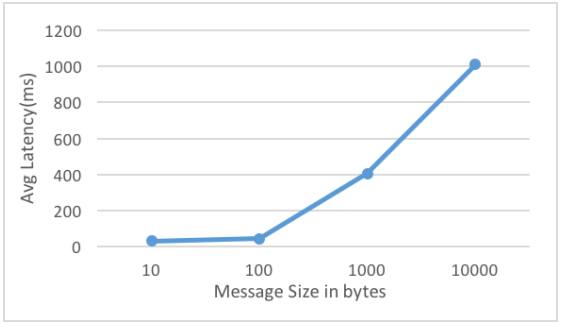

Fig. 10. Avg Latency with varying message size

V. CONCLUSIONS

In this paper, we performed a set of experiments to study performance evaluation of Kafka on AWS. The presented results indicate that Kafka is fast and scalable distributed messaging system on AWS. For performance measurement, two important metrics were involved such as latency and throughput. Most of the systems are either optimized for latency or throughput but Kafka provides balance for both. As shown in the results, Kafka cluster can be tuned fairly with enough brokers along with other configurations to provide the required throughput for the required latency to process information. In our experiment, we have compared our results on AWS with results obtained in paper[3] on Palmetto. Based on this comparison, we infer that even with increased latency, Kafka can be an ideal platform for real-time streaming on AWS, combined with benefits of AWS. Kafka is simple messaging system for usage but involves complex study for its performance due to multiple tuning parameters. We have covered basic performance evaluation of the system and aim to cover major real time scenarios in future.

REFERENCES

[1] Kafka: a Distributed Messaging System for Log Processing, Jay Kreps, Neha Narkhede, Jun Rao, LinkedIn Corp.
[2] Kafka, Samza and the Unix Philosophy of Distributed Data, Martin Kleppmann, University of Cambridge, Computer Laboratory, Jay Kreps, Confluent, Inc.
[3] A Distributed Data Delivery Infrastructure for Supporting Connected Vehicle Technology Applications, Yuheng Du, Dr.Amy Apon, School of Computing, Clemson University, SC.(still work is going on)
[4] Wikipedia for logging solution[Online]: https://wikitech.wikimedia.org/wiki/Analytics/Cluster/Logging_Solutions_Recommendation
[5] Apache Kafka on AWS[Online]: https://aws.amazon.com/kafka/
[6] Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ, M. Rostanski, K. Grochla, and A. Seman, In Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on, pp. 879-884. IEEE, 2014.
[7] Benchmarking Apache Kafka: 2 Million Writes Per Second, Jay Kreps, Co-founder and CEO at Confluent.
[8] JAVA Message Service[Online]: http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html
[9] Building LinkedIns Real-time Activity Data Pipeline, Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede,Richard Park, Jun Rao, Victor Yang Ye, LinkedIn.
[10] Stanford facebook data[Online]: https://web.stanford.edu/class/cs101/bits-gigabytes.html
[11] Facebook Scribe GitHub[Online]: https://github.com/facebookarchive/scribe, Thomas Porschberg, Michael Tindal, and Daniel Casimiro
[12] Apache Avro website[Online]: http://avro.apache.org/
[13] AWS Official website[Online]: https://aws.amazon.com/
[14] Kafka Official website[Online]: https://kafka.apache.org/