FaBRiQ: Leveraging Distributed Hash Tables towards Distributed Publish-Subscribe Message Queues

Iman Sadooghi, Ke Wang, Dharmit Patel, Dongfang Zhao, Tonglin Li, Shiva Srivastava, Ioan Raicu
isadoogh@iit.edu, {kwang22, dpatel74, dzhao8, tli13, ssriva10}@hawk.iit.edu, iraicu@cs.iit.edu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

***Abstract*— The advent of Big Data has brought many challenges and opportunities in distributed systems, which have only amplified with the rate of growth of data. There is a need to rethink the software stack for supporting data intensive computing and big data analytics. Over the past decade, the data analytics applications have turned to finer granular tasks which are shorter in duration and much more in quantity. Such applications require new frameworks to handle their data flow. Distributed Message Queues have proven to be essential building blocks in distributed computing towards the support for fine granular workloads. Distributed message queues such as Amazon's SQS or Apache's Kafka have been used in handling massive data volumes, content delivery, and many more. They have also been used in large scale job scheduling on public clouds. However, even these frameworks have some limitations that make them incapable of handling large scale data with high efficiency. Those are not suitable for High Performance Computing (HPC) applications that require lower latency than what is available on the cloud. We propose Fabriq, a distributed message queue that runs on top of a Distributed Hash Table. The design goal of Fabriq is to achieve lower latency and higher efficiency while being able to handle large scales. Moreover, Fabriq is persistent, reliable and consistent. Also, unlike other state-of-the-art systems, Fabriq guarantees exactly once delivery of the messages. The results show that Fabriq was able to achieve high throughput in both small and large messages. At the scale of 128 nodes, Fabriq's throughput was as high as 1.8 Gigabytes/sec for 1 Megabytes messages, and more than 90,000 messages/sec for 50 bytes messages. At the same scale, Fabriq's latency was less than 1 millisecond. Our framework outperforms other state of the art systems including Kafka and SQS in throughput and latency. Furthermore, our experiments show that Fabriq provides a significantly better load balancing than Kafka. The load difference between Fabriq servers was less than 9.5% (compared to the even share), while in Kafka this difference was 100%, meaning that some servers did not receive any messages and remained idle.**

Keywords*— *Message Queue, Data Analytics, Distributed Hash Tables, Distributed Systems

I. INTRODUCTION

The advent of Big Data and the exascale computing has changed many paradigms in the computing science area. More than 2.5 exabytes of data is generated every day, and more than 70% of it is unstructured [19]. Various organizations including governments and big companies generate massive amounts of data in different formats including logs, and other unstructured raw data every day. Experts predict that by the end of 2018, the exascale computers will start to work [22]. With the growth of the data at the current rate, it is unlikely for the traditional data processing systems to be able to handle the requirement of Big Data processing. There is a need to reinvent the wheel instead of using the traditional systems. Traditional data processing middleware and tools such as SQL databases and file system are being replaced by No-SQL data-stores and key-value storage systems in order to be able to handle the data processing at the current scale. Another key tool that is getting more attention from the industry is distributed queuing service.

A Distributed Message Queue (DMQ) could be an important building block for a reliable distributed system. Message Queues could be useful in various data movement and communication scenarios. In High Throughput Computing (HTC), message queues can help decouple different components of a bigger system that aims to run in larger scales. Using distributed queues, different components can communicate without dealing with the blocking calls and tightly coupled communication.

Over the past few years, distributed queuing services have been used in both industrial and scientific applications and frameworks [1][2][20][21][26][28]. SQS is a distributed queue service by Amazon AWS, which is being leveraged by various commercial applications. Some systems have used SQS as a buffer for their server to handle massive number of requests. Other applications have used SQS in monitoring, workflow applications, big data analytics, log processing and many other distributed systems scenarios [1][23][36].

The large scale log generation and processing is another example that has become a major challenge on companies that have to deal with the big data. Many companies have chosen to use distributed queue services to address this challenge. Companies like LinkedIn, Facebook [6], Cloudera [4] and Yahoo have developed similar queuing solutions to handle gathering and processing of terabytes of log data on their servers [5]. For example LinkedIn's Kafka [3] feeds hundreds of gigabytes of data into Hadoop [25] clusters and other servers every day.

Distributed Queues can play an important role in Many Task Computing (MTC) [8] and High Performance Computing (HPC). Modern Distributed Queues can handle data movement on HPC and MTC workloads in larger scales without adding significant overhead to the execution [7].

CloudKon is a Distributed Job Scheduling system that is optimized to handle MTC and HPC jobs. It leverages SQS as a task delivery fabric that could be accessed simultaneously and achieve load balancing at scale [2][29]. CloudKon has proved to outperform other state-of-the-art schedulers like Sparrow [18] by more than 2X in throughput. One of the main motivations of this work is to provide a DMQ that can replace SQS in future versions of the CloudKon. There are a few

limitations with SQS including having duplicate messages, and getting the system tied to AWS cloud environment. CloudKon [2] uses DynamoDB [27] to filter out the duplicate messages. Among the various DMQs, Fabriq and Kafka are the only alternatives that can provide the acceptable performance at larger scales required by CloudKon. Kafka is mainly optimized for large scale log delivery. It does not support multiple clients read from one broker at the same time. Moreover, it does not have a notion of independent messages or tasks. These limitations can significantly degrade the performance of CloudKon. Fabriq has none of those limitations. Leveraging Fabriq, CloudKon can run independently on any generic distributed system without being tied to SQS, DynamoDB, or the Amazon AWS Cloud in general [30]. Moreover, our results on section V show that Fabriq provides a much higher throughput and much lower latency than SQS. According to our comparison results between SQS and Fabriq, and based on the fact that the future version of CloudKon will not have the overhead of DynamoDB, we expect about a 20X performance improvement (13X for using Fabriq and 1.5X for not using DynamoDB) on future version of CloudKon.

There are various commercial and open sourced queuing services available [9][10][11][12]. However, they have many limitations. Traditional queue services usually have centralized architecture and cannot scale well to handle today's big data requirements. Providing features such as transactional support or consumption acknowledgement makes it almost impossible for these queues to achieve low latency. Another important feature is persistence. Many of the currently available options are in memory queues and cannot guarantee persistence. There are only a few DMQs that can scale to today's data analytics requirement. Kafka is one of those that provide large scale message delivery with high throughput. However, as it is shown in Fig. 8 and Fig. 9, Kafka has a long message delivery latency range. Moreover, as we have shown in Fig. 6, Kafka cannot provide a good load balance among its nodes. That could cause Kafka to perform inefficiently in larger scales.

Today's data analytics applications have moved from coarse granular tasks to fine granular tasks which are shorter in duration and much more in number [18]. Such applications cannot tolerate a data delivery middleware with an overhead in the order of seconds. It is necessary for a DMQ to be as efficient as possible without adding substantial overhead to the workflow.

We propose Fast, Balanced and Reliable Distributed Message Queue (Fabriq), a persistent reliable message queue that aims to achieve high throughput and low latency while keeping the near perfect load balance even on large scales. Fabriq uses ZHT as its building block. ZHT is a persistent distributed hash table that allows low latency operations and is able to scale up to more than 8k-nodes [13]. Fabriq leverages ZHT components to support persistence, consistency and reliable messaging. Another unique feature of Fabriq is the guarantee of exactly once delivery. To our best knowledge, no other DMQ provides such guarantee. This requirement becomes a challenge for the systems that aim to support large scale delivery. A common practice is to keep multiple copies of the message on multiple servers of a DMQ. Once the message gets delivered by a server, it will asynchronously inform other servers to remove their local copies. However, since the informing process is asynchronous, there is a change of having a message delivered to multiple clients before getting removed from the servers. Hence, the systems with such procedure can generate duplicate messages.

The fact that Fabriq provides low latency makes it a good fit for HPC and MTC workloads that are sensitive to latency and require high performance. Also, unlike the other compared systems (Fig. 8 and Fig. 9), Fabriq provides a very stable delivery in terms of latency variance. Providing a stable latency could be substantial for MTC applications, as well as towards having predictable performance. Finally, Fabriq supports dynamic scale up/down during the operation. In summary, the contributions of Fabriq are:

- It uses ZHT as its building block to implement a scalable DMQ.
- Leveraging ZHT components, it supports persistence, consistency, reliable messaging and dynamic scalability.
- It guarantees exactly once delivery of the messages.
- It achieves a near perfect load balance among its servers.
- It provides high throughput and low latency outperforming Kafka and SQS. It also provides a shorter latency variance than the other two systems.
- It could be used on HPC environments that do not support Java (e.g. Blue Gene L/P supercomputers).

The rest of the paper is organized as follows. We review the related works on section II. Section III discusses the Fabriq's architecture. We first briefly go over the architecture of ZHT and explain how Fabriq leverages ZHT to provide an efficient and scalable DMQ. Later on section IV, we analyze the communication costs on Fabriq. Section V evaluates the performance of the Fabriq in different metrics. Finally, section VI concludes the paper and discusses about the future work.

II. Related Work

Enterprise queue systems are not new in the distributed computing area. They have been around for quite a long time and have played a major role in asynchronous data movement [32][33][34]. Systems like JMS [12] and IBM Websphere MQ [11] have been used in distributed applications. However, these systems have some limitations that make them unusable for today's big data computing systems. First, these systems usually add significant overhead to the message flow that makes them incapable of handling large scale data flows. JMS supports delivery acknowledgement for each message. IBM Websphere MQ provides atomic transaction support that lets the publisher submit a message to all of the clients. These features can add significant overhead to the process. It is not trivial to handle these features for the larger scale systems. In general, traditional queuing services have many assumptions that prevent them from scaling well. Also, many of these traditional services do not support persistence [35].

ActiveMQ [10] is a message broker in Java that supports AMQP protocol. It also provides a JMS client. ActiveMQ provides many configurations and features. But it does not support any message delivery guarantee. Messages could be delivered twice or even get lost. Other researches have shown that it cannot scale very well in larger scales [3].

RabbitMQ [9] is a robust enterprise queuing system with a centralized manager. The platform provides option to choose

between performance and reliability. That means enabling persistence would highly degrade the performance. Other than the persistence, the platform also provides options like delivery acknowledgement and mirroring of the servers. The message latency on RabbitMQ is large and not tolerable for any application that is sensitive to the efficiency. Being centralized makes RabbitMQ not scale very well. It also makes it unreliable because of having a single point of failure. Other researches have shown that it cannot perform well in larger scales as compared to scalable systems like Kafka [3].

Besides the traditional queuing systems, there are two modern queuing services that have got quite popular among commercial and open source user community. Those two are Apache Kafka [2] and Amazon Simple Queue Service (SQS) [14]. Kafka is an open source, distributed publish and consume service which is introduced by LinkedIn. The design goal of Kafka is to provide a system that gathers the logs from a large number of servers, and feeds it into HDFS [15] or other analysis clusters. Other log management systems that were provided by other big companies are usually saving data to offline file systems and data warehouses. That means they do not have to provide low latency. However, Kafka can deliver data to both offline and online systems. Therefore, it needs to provide low latency on message delivery. Kafka is fully distributed and provides high throughput. We discuss more about Kafka later in a separate section.

Amazon SQS is a well-known commercial service which provides reliable message delivery in large scales. SQS is persistent. Like many other Amazon AWS services [16], SQS is reliable and highly available. It is fully distributed and highly scalable. We discuss more about SQS and compare its features to Fabriq in another section.

III. FABRIQ ARCHITECTURE AND DESIGN PRINCIPLES

This section discusses the Fabriq design goals and demonstrates its architecture. As we discussed in the previous section, many of the available alternative solutions do not guarantee exactly once delivery, persistence and reliability. There are many ways to implement a distributed queue. A distributed queue should be able to guarantee message delivery. It should also be reliable. Finally, a distributed queue has to be highly scalable. Most of the straight forward design options include centralized manager component that limit the scalability. Depending on the architecture, a Distributed Hash Table (DHT) could achieve high scalability as well as maintaining other benefits. Fabriq uses a DHT as its building block of our queuing services. The simple put and get methods of a DHT could be similar to the push and pop methods on a DMQ. We chose to use ZHT which is a low overhead and low latency DHT, and has a constant routing time. It also supports persistence. Before discussing the design details of Fabriq, we briefly review the architecture and the key features of ZHT.

A. ZHT overview

ZHT has a simple API with 4 major methods: 1. *insert*(key, value); 2. *lookup*(key); 3. *remove*(key), and 4. *append*(key,value). The key in ZHT is a simple ASCII character string, and the value can be a complex object. A key look up in ZHT can take from 0 (if the key exists in the local server) to 2 network communications. This helps providing the fastest possible look up in a scalable DHT. The following sections discuss main features of ZHT.

1) Network Communication

ZHT supports both TCP and UDP protocols. In order to optimize the communication speed, the TCP connections will be cached by a LRU cache. That will make TCP connections almost as fast as UDP. In Fabriq, we rely on the ZHT for the network communications. Having optimized TCP communications enables Fabriq to achieve low latency on its operations.

2) Consistency

ZHT supports replication to provide a reliable service. In order to achieve high throughput ZHT follows a weak consistency model. The first two replications for each dataset are strongly consistent. That means the data will be written to the primary and the secondary replicas. After completion of the write on the secondary replica, the replication to the following replicas happens in an asynchronous fashion [31].

3) Fault Tolerance

ZHT supports fault tolerance by lazily tagging the servers that are not being responsive. In case of failure, the secondary replica will take the place of the primary replica. Since each ZHT server operates independently from the other servers, the failure of a single server does not affect the system performance. Every change to the in-memory DHT data is also written to the disk. Therefore, in case of system shut down (e.g. reboot, power outage, maintenance, etc.) the entire data could be retrieved from the local disk of the servers.

4) Persistence

ZHT is an in-memory data-structure. In order to provide persistence, ZHT uses its own Non-Volatile Hash Table (NoVoHT). NoVoHT uses a log based persistence mechanism with periodic check-pointing.

5) Dynamic Scalability (membership)

ZHT supports dynamic membership. That means server nodes can join or leave the system any time during the operation. Hence, the system scale can be dynamically changed on ZHT (and also Fabriq) during the operation. Although dynamic scalability of Fabriq is supported, due to space limitation, we will explore the evaluation of dynamic membership in Fabriq in future work.

B. Fabriq Design and Architecture

The main design goal of Fabriq is achieving high scalability, efficiency and perfect load balance. Since Fabriq is using ZHT as its building block for saving messages and the communication purposes, and ZHT has proven to be able to scale more than 8k-nodes, we can expect Fabriq to also scale as much as ZHT [13].

Fabriq distributes the queue load of each of the user queue among all of its servers. That means user queues can co-exist on multiple servers. When a single server is down due to any reason such as failure or maintenance, the system can continue serving all of the users with other servers. That enables the system to provide a very high availability and reliability. Fig. 1 depicts the message delivery of multiple user queues in Fabriq.

Like any other message queue, Fabriq has the simple push and pop functions. In addition to those, Fabriq also supports peek method which is reading the contents of a message

without removing it. In order to implement the queue functionalities, we have used ZHT as our system building block and extended the queue functionalities to it. Fig. 2 shows the structure of a single Fabriq server. Besides the local NoVoHT hash table, there are two different data structures on each server.

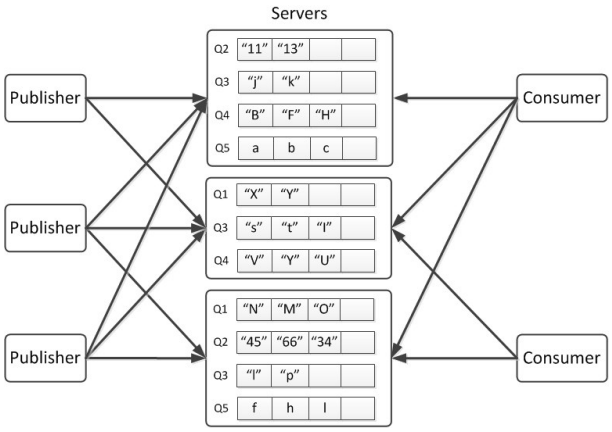

Fig. 1. Fabriq servers and clients with many user queues.

MessageId Queue: it is a local in-memory queue, used to keep the message IDs of a user queue that are saved on the local NoVoHT on this server. The purpose of having this queue is to be able to get messages of a user queue from the local hash table without having the message Ids, and also to distinguish the messages of different user queues from each other. This way, each Fabriq server can independently serve the clients without having to get the message Ids of a user queue from a central server.

Metadata List: each user queue has a unique Metadata list in the whole system which keeps the address of the servers which have messages of this certain queue. The Metadata list only exists in one server. The purpose of having this list is to reduce the chance of accesses to the servers that don't have messages for a user queue.

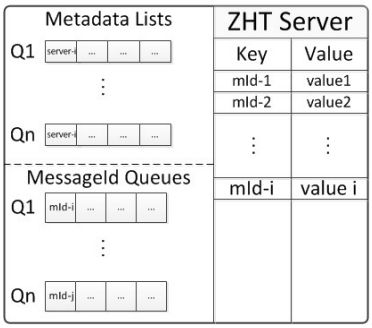

Fig. 2. Structure of a Fabriq server.

Next, we discuss about the process of delivering messages by explaining the major methods on Fabriq. Besides the below mentioned methods, Fabriq has peek and deleteQueue.

1) createQueue

This method lets users define their own queue. The method gets a unique name for the queue (assume it is "Qx"), and hashes the name. Based on the hashing value, the client sends a createQueue request to the destination server. Then it will define a unique Metadata List for "Qx". The Metadata List is supposed to keep the address of the servers that keep the messages of "Qx". It will also create a MessageId queue for "Qx" for the future incoming messages to this server. A user queue can have more than one MessageId queue in the whole system, but it has only one Metadata List. The Metadata List of a user queue resides on the server with the same address as the hash value of that user queue name.

2) push

Once a user queue has been defined, the client can push messages to it. The method has two inputs: the queue name and the message contents. Fabriq uses Google Protocol Buffer for message serialization and encoding. Therefore, the message contents input supports both string or user defined objects.

Once the push method is called, the client first generates a message Id using its IP Address, port, and a counter. The message Id is unique on the whole system. Then, the client hashes the message Id and chooses the destination server based on the hash value. Since the hashing function in Fabriq distributes the signature uniformly among all of the servers, the message could land on any of the collaborating servers.

Fig. 3 depicts the push procedure on Fabriq. After receiving the push request, the destination server performs one of the following based on the queue name:

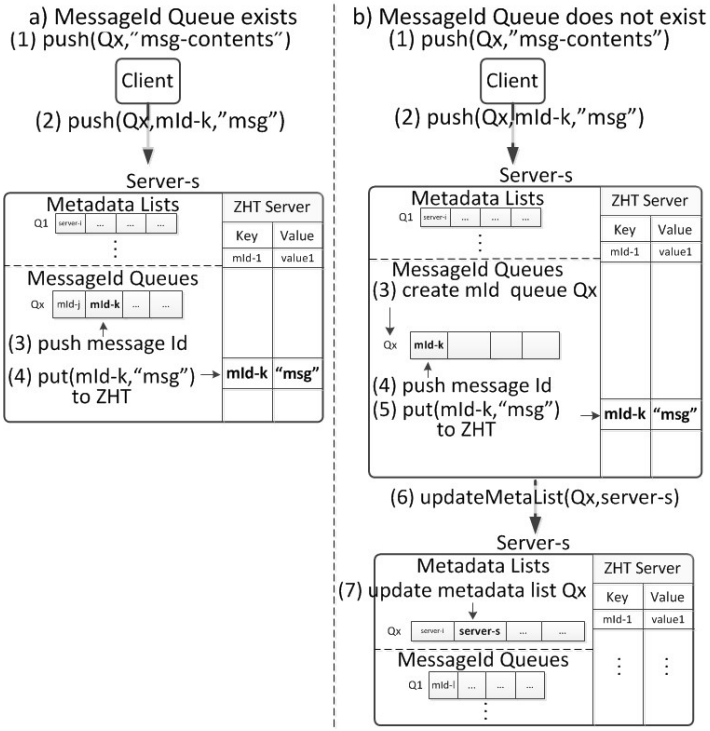

Fig. 3. Push operation.

a) If the MessageId queue exists in this server, it will add the new MessageId to the queue and then it will make a put request to the underlying ZHT server. Since the hashing function used to hash the message Id on the client side is the same as the ZHT server's hashing function, the hash value will again determine the local server itself as the destination. Thus the ZHT server will add the message to its local NoVoHT server and there will be no additional network communications involved.

b) If the destination server does not have a MessageId queue with the name of this user queue, the server first creates a new MessageId queue for the user queue on this server, and then it will push the message to the MessageId queue and the local NoVoHT. Meanwhile, the Metadata List of this user queue has to be updated with the information of the new server that keeps its messages. The server makes a request to the server that keeps Metadata List of the user queue and adds its own address to that list. The address of the destination server that keeps the Metadata list will be retrieved by hashing the name of the user queue.

3) pop

The pop method requests a message from a user queue on a local or remote Fabriq server. We want to make sure to retrieve a message from a Fabriq server with the lowest latency and the minimum network communication overhead.

A message of a certain queue may reside in any of the servers. The client can always refer to the Metadata list of a certain queue to get the address of a server that keeps messages of that queue. However, referring to the owner of the Metadata list in order to find a destination server adds network communication overhead and degrades the performance. Moreover, on larger scales, accessing the single metadata list owner could become a bottleneck for the whole system. In order to avoid the communication overhead, the client first tries the following ways before directly going to the metadata List owner:

(1) When a client starts to run, it first checks if there is a local Fabriq server running on the current node. The pop method first gets all of the messages on the local Fabriq server. The method sends the pop request to the local server and keeps getting messages until the mId queue is empty. After that the server returns a null value meaning that there is nothing left for this user queue on this server.

(2) After getting the null value, the client uses the second approach. It generates a random string and makes a pop request to a random server based on its hash value. Please note that the random string is not used as the message Id to be retrieved and it is only used to choose a remote server. If the destination server has messages, the client saves the random string as the last known server for the later accesses of this user queue. The client keeps popping messages from the last known server until it runs out of the messages for this user queue and returns null value.

(3) Finally, after client finds out that the last know server has returned null, using the hash value of the user queue name, it sends a request to the metadata list and gets the address of a server that has messages for this queue. Once a server returns null, the client again goes back to the metadata list owner and asks for a new server address.

Fig. 4 shows a remote pop operation that only takes 1 hop. On the server side, the pop method looks for the MessageId queue of the requested user queue: **a)** If the mId queue does not exist in this server or if it is empty, the pop method returns a null value to the client. **b)** If the mId queue exists and has at least one message Id, it will retrieve a mId from the queue and makes a ZHT get request. Since the message Ids on the local queue have the same hash value as the local server's Id, the get request which is supposed to hash the message Id to find the server's address will get the value from the local ZHT server. Then the pop method will return that message to the client. If the retrieved mId was last one on the mId queue, the server calls a thread to asynchronously update the Metadata List of this user queue and remove the server Id from it.

C. Features

In this section, we discuss about some of the important features of Fabriq that makes it superior to other state-of-the-art message queues.

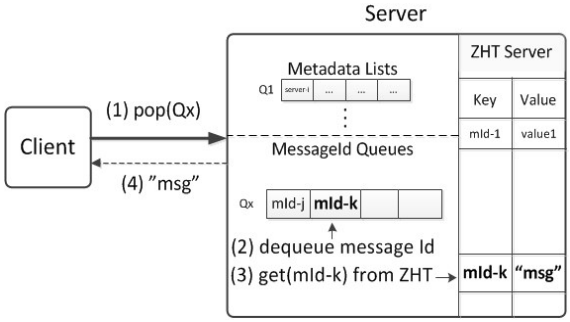

Fig. 4. A remote pop operation with a single hop cost.

1) Load Balancing

One of the design goals of Fabriq is to achieve a near perfect load balance. We want to make sure that the load from multiple queues gets distributed on all of the servers.

The load balancing of a system can highly depend on its message routing strategy. The systems with deterministic message routing usually have a static load distribution. That means the messages of multiple queues are statically split among all of the servers. This design is more convenient for the centralized and hierarchical architectures. However, there are many limitations with such design. In these architectures, the load balance on the system can fluctuate depending on the submission rate on different queues. On the other hand, the systems with non-deterministic routing have a more dynamic load on the servers. In order to have a dynamic load distribution, Fabriq Client generates a randomly generated key for each message. Based on the hash value of the key, the message will be sent to a Fabriq server. Fabriq uses a uniformly distributed hash function.

2) Order of messages

Like many other distributed message queues, Fabriq cannot guarantee to keep the order of messages in the whole queue [2][13]. However, it can guarantee the order of the messages in a single server. The messages of a user queue are written in a local message queue on each server and the order of the messages is kept in that queue. Therefore the order of messages delivery in the server will be kept.

The message delivery order can be important for some workflows in scientific applications or HPC, we have provided a specific mode to define queues in a way that it keeps the message order. In this mode the messages of the user queue are only submitted to a single server. Since the order is kept in the single server, the order of the delivery will be kept as submitted.

3) Message delivery guarantee

When it comes to large scale distributed systems, the delivery of the content becomes a real challenge. Distributed systems cannot easily deal with this problem. On most of the loosely coupled systems where each node controls its own

state, the delivery is not guaranteed. In distributed Message Queues, the delivery of the messages is an inevitable requirement. Therefore, most of the state of the art systems guarantee of the delivery. It is hard for the independent servers to synchronize with each other at large scales. Therefore they guarantee the delivery at the cost of producing duplicate messages. Thus, they guarantee at least once delivery. However, Fabriq guarantees exactly once delivery.

Fabriq benefits from using a persistent DHT as its backbone. The push method in Fabriq makes a put request on the ZHT. The data in each server is persistent. ZHT also provides replication. Replication can prevent the loss of data in case of losing the hard disk on a node. The push and pop functions are both blocking functions. The client only removes the message from the memory after the server returns a success notification. There are two possible scenarios in case of the network or the server failure. If the message somehow does not get delivered, the server will not send a success notification. Therefore the push function times out and the message will be sent again. However, there is a possibility that the message gets delivered and the notification signal gets lost. In such scenario, the client will again send the message. This behavior could lead to duplicate messages on the system. However, the message destination is determined by the hash value of its message Id (mId). That means a message with a certain mId will always deliver to the same destination server. In case of the duplicate delivery, the ZHT destination server will notice a rewrite on the same Id and throws an exception. Therefore the messages are pushed with no duplicate messages.

Likewise, on the pop operation, the server only removes the message from ZHT when the client returns a success notification. The pop method first performs a ZHT get on the server side. It only performs a ZHT remove after it gets a notification from the client. Since the network proxy is TCP, if the client fails to receive the message or if it fails to notify the server, the TCP proxy throws an error and server notifies the client to ensure the delivery. Also, since Fabriq guarantees strong consistency between the primary and the secondary servers, the secondary server will not deliver a message that was removed from the primary server. In the rare case of the network temporary partition when the primary server gets disconnected, the secondary server delivers the message. When the primary server comes back online, it might deliver the message that was already popped. This could be prevented by attempting to detect the network partition, and issue a protocol for verifying that all replicas are consistent. We leave that as a future work.

Therefore, Fabriq can guarantee exactly once delivery except when temporary network partitions take place, in which case we could implement a protocol to validate consistency.

4) Persistence

Fabriq extends the ZHT's persistence strategy to provide persistence. In ZHT, a background thread periodically writes the hash table data into the disk. Using ZHT, we can make sure the messages are safe in the hash table. But Fabriq still needs to keep its own data structure persistent in the disk. Otherwise, in case of system shut down of memory fail, Fabriq will not be able to retrieve messages from the hash table. In order to save the MessageId Queues and the Metadata List on each server, we have defined a few key-value pairs in the local NoVoHT table of each server. We save the list of the Metadata Lists and the MessageId Queues in two key-value pairs. We also save the contents of each single queue or list on an object and save those separately in the hash table. The background thread periodically updates the values of the data structures on the hash table. In case of failure, the data structures could be rebuilt using the key for the list of queues and lists in the hash table.

5) Consistency and Fault tolerance

Fabriq extends the ZHT strategies for its fault tolerance and consistency. It supports a strong consistency model on the first two replicas. The consistency is weak after the second replica. Fabriq also implements the lazy tagging of failed servers. In case of failure the secondary replica will take over the delivery. The metadata lists and the MessageId queues of each Fabriq server are locally saved on its ZHT. Therefore they are automatically replicated on different servers. In case of the failure of a server, they can be easily regenerated from the replica server.

Another strategy which helps Fabriq provide better fault tolerance is spreading each user queue over all of the servers. In case of the failure of a server, without any need to link the client, the client will randomly choose any other server and continue pushing/retrieving messages from the system. Meanwhile, the secondary replica takes over and fills the gap.

6) Multithreading

Fabriq supports multithreading on the client side. The client can do push or pop using multiple threads. On the server side, Fabriq can handle simultaneous requests. But it does not use multithreading. The Fabriq server uses an event-driven model based on epoll which is able to outperform the multithreaded model by 3x. The event-driven model also achieves a much better scalability compared to the multithreading approach [13].

IV. NETWORK COMMUNICATION COST

In order to achieve low latency and high efficiency, it is important to keep the number of network communications low. In Fabriq, we design our push and pop operation with the minimum possible number of network communications. In this paper, we consider each network communication as one hop.

Push cost: As shown in Fig. 3, the push operation takes only one hop to complete. Since the update of the metadata list is executed by a separate thread in a non-blacking fashion, we don't count it as an extra hop. Moreover, it only happens in the first push of each server. Therefore it does not count as an extra hop for the push operation.

Pop cost: A pop operation communication cost varies depending on the situation of both the client and the server. In order to be able to model the cost, we make a few assumptions and simplify our model. We assume that the uniform hash function works perfectly, and evenly distributes the messages among all of the servers. We analyze this assumption in practice in section V.C.

We model the total cost of the pop operation in a system with a single consumer and multiple servers. s shows the number of servers and m shows the total number of messages

that was produced by the clients. We model the cost in two situations: (a) when the total number of messages is more than the number of servers ($m>s$); and (b) when the number of messages is less than the number of servers ($m<s$). The total cost when $m>s$ is shown below:

$$Total\ Cost_{(m>s)} = \frac{m}{s} \times 0 + \left(\frac{m}{s} - 1\right)(s-1) \times 1 + \sum_{i=1}^{s-1} \frac{i \times 3 + (s-i) \times 1}{s}$$

Based on the assumption of having perfect uniform distribution, we can assume that each server has m/s messages at the beginning of the consumption. Since the consumer first consumes all of the messages on its local server, the cost of the first m/s messages is going to be zero hop. After that, the consumer randomly chooses a server among the $s-1$ that are left. The cost of finding a server with messages can be either 1 or 3. The client saves the id of the last known server and only makes a random call when the last known server has no messages left. After finding a new server, the client fetches all of the messages on the last known server until the server is empty. The cost of all of these messages ($(m/s)-1$) is 1 hop. This process continues until all of the messages of each server are consumed. We can conclude that on each of the $s-1$ remote servers there will be a single message that is going to be retrieved with the cost of 1 or 3 hops and $(m/s)-1$ messages that are retrieved with the cost of exactly 1 hop. Having the total cost of the retrieval, we can calculate the average cost of each pop operation by dividing the total cost by the number of total messages:

$$Average\ Cost_{(m>s)} = 1 - \frac{1}{s} + \frac{s-1}{2m}$$

We can induce the range of the cost from the average cost formula. The average cost ranges from <1 to <1.5 hops. In the second scenario where the total number of messages is less than the number of servers, the total cost is:

$$Total\ Cost_{(m<s)} = \sum_{i=1}^{m} \frac{(s + i - (m+1)) \times 3 + ((m+1) - i) \times 1}{s}$$

In this case, since each server gets one message at most, the cost of retrieving each message can be either 1 or 3. The average cost analysis is provided below:

$$Average\ Cost_{(m<s)} = 3 - \frac{m+1}{s}$$

Again, we can induce that the average cost of pop in this case ranges from 2 to 3 hops.

In order to confirm our analysis, we ran an experiment with a single consumer and counted the average number of hops on each pop operation. Fig. 5 shows the hop count in an experiment with 1 client and 64 servers. The total number of messages in this run was 64,000 messages. The results show that there were 1,079 messages on the local queue with the cost of 0 hops. Based on the cost model the average cost of hops in this experiment is 0.984 and the actual average cost is 1.053 hops, which means the model is fairly accurate.

The maximum communication cost in a system with multiple clients could be more than 3 hops. Since multiple clients can request a queue metadata owner for a message server at the same time, there is a chance that they both receive the same message server address from the metadata owner. Assuming the message server has only 1 message for this queue, the first client can get that last message, and the second client gets a null return value. In that case the client has to request the owner server again for another message server. This process can be repeated for s times until the client gets a message. However the chances of this occasion are very low. In fact, we have ran experiments in up to 128 instances scale and have not experienced a pop operation with more than 5 hops.

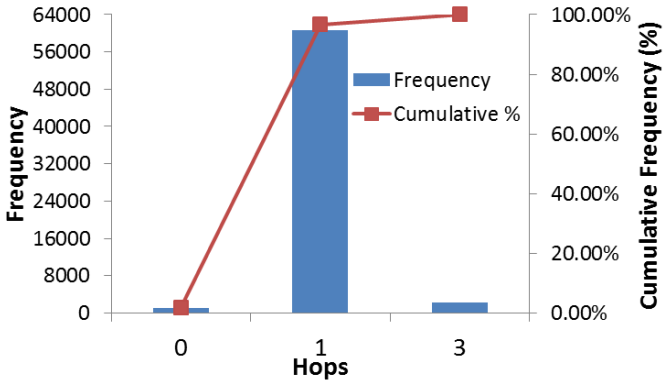

Fig. 5. Cumulative Distribution of 1 client and 64 servers.

V. Performance Evaluation

This section analyzes the performance of Fabriq in different scenarios, compared with the other state of the two art Message Queue systems. But first, we summarize different features of Fabriq, compared with Kafka and SQS. We compare the performance of the three systems in terms of throughput and latency. We also compare the load balancing of the Kafka and Fabriq.

A. Fabriq, Kafka, and SQS

All three of the compared systems are fully distributed and are able to scale very well. However, they use different techniques in their architecture. Fabriq uses a DHT as its building block, while Kafka uses Zookeeper [17] to handle the metadata management. SQS is closed source and there is minimal information available about its architecture.

One of the important features of a distributed queue is its message retrieval policy. All of the Fabriq servers act as a shared pool of messages together. That means all of the clients have equal chance of accessing a message at the same time. This feature enables the system to provide better load balancing. Moreover, having this feature, the producer can make sure that its messages are not going only to a specific consumer, but all of the consumers. SQS provides this feature as well. In Kafka, messages that reside in a broker (server) are only consumed by a single consumer at a time. The messages of that broker will only be available when the consumer gets the number of messages it requires. This can cause load imbalance when there is not enough messages in all of the brokers and degrade the system performance. This design goal in Kafka was a tradeoff to provide the rewind feature. Unlike other conventional queue systems including Fabriq and SQS, Fabriq provides message rewind feature that lets consumers to re-consume a message that was already consumed. However, s mentioned before, having this feature means only one consumer can access a broker at a time.

TABLE I. summarizes the features of the three queuing services. One of the major benefits of Fabriq over the other systems is providing exactly once delivery. Unlike the other

two systems, Fabriq does not support message batching yet. However this feature is currently supported in the latest version of ZHT and can be easily integrated with Fabriq. We expect that batching is going to improve the throughput significantly.

In Kafka brokers, messages are written as a continuous record and are only separated by the offset number. This feature helps Kafka provides better throughput for continues log writing and reading from producers and consumers. However, as mentioned before, this makes it impossible for multiple consumers to access the same broker at the same time. SQS and Fabriq save messages as separate blocks of data that enables those to provide simultaneous access on a single broker. All three of the systems provide the queue abstraction for multiple clients. In Fabriq and SQS, the client can achieve this by creating new queues. In Kafka, the client achieves this by defining new topics. Another important feature of Fabriq is the fact that it is able to run on different types of supercomputers including Blue Gene series that don't support Java. Kafka is written in Java, and SQS is closed source. Scientists are unable to use those two systems for HPC applications that run on such supercomputers.

TABLE I. COMPARISON OF FABRIQ, SQS AND KAFKA

Feature	Fabriq	Kafka	SQS
Persistence	Yes	Yes	Yes
Delivery Guarantee	Exactly Once	At least Once	At least Once
Message Order	Inside Node	Inside Node	-
Replication	Customizable	Mirroring	3x
Shared Pool	Yes	No	Yes
Batching	No (Future work)	Yes	Yes

B. Testbed and Configurations

Since SQS runs on AWS, in order to keep our comparisons fair, we chose Amazon EC2 as our testbed. The experiments scale from 1 to 128 instances. We chose m3.medium instances. Each instance has a single CPU core, a 1 Gigabit network card, and 16 GB of SSD storage.

C. Load Balance

As discussed before, we believe that Fabriq provides a very good load balance. In this section we compare the load balancing of Fabriq with Kafka by checking the message distribution on the server of both systems. Since we don't have access to the servers on SQS, we cannot include this system on this experiment.

Fig. 6 shows the number of messages received on each server of the two systems. In this experiment, each producer has sent 1000 messages. The total number of messages is 64000. The results show a very good load balance on Fabriq. The number of messages range from 940 to 1088 messages on 64 servers. We ran the experiment 5 times and found out that the error rate is less than 5% for at least 89% of the servers, and is less than 9.5% in worst case. In Kafka, we observe a major load imbalance. The number of messages per server ranged from 0 to 6352. More than half of the servers got less than 350 messages.

Considering the fact that each server can only be accessed by one consumer at a time, we can notice that there will be a major load imbalance in the system. In a system with a 1 to 1 mapping between the servers and the consumers, more than half of the consumers go idle after finishing the messages of the underutilized servers and will wait for the rest of consumers to finish consuming their messages. Only after that, they can consume the rest of the messages and finish the workload.

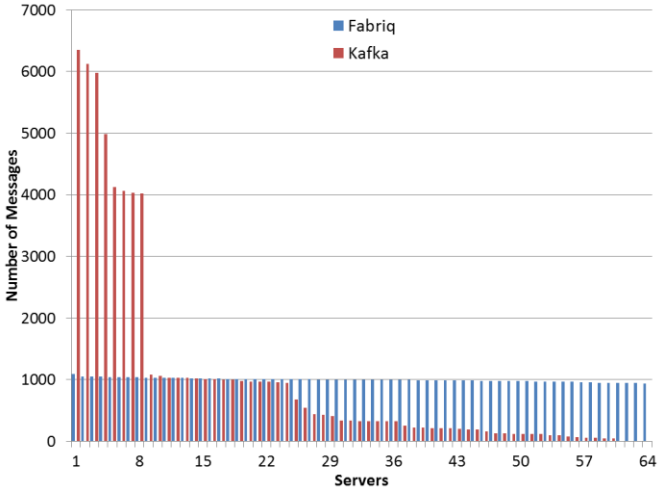

Fig. 6. Load Balance of Fabriq vs. Kafka on 64 instances.

D. Latency

The latency of the message delivery is a very important metric for a distributed message queue. It is important for a DMQ to provide low latency on larger scales in order to be able to achieve high efficiency. Nowadays, many of the modern scientific and data analytics applications run tasks with the granularity of sub-seconds [18]. Therefore, such systems will not be able to exploit a message queue service that delivers messages in the order of seconds.

We measured latency by sending and receiving 1000, 50 bytes messages. Each instance ran 1 client and 1 server. Fig. 7 shows the average latency of the three systems in push and pop operations.

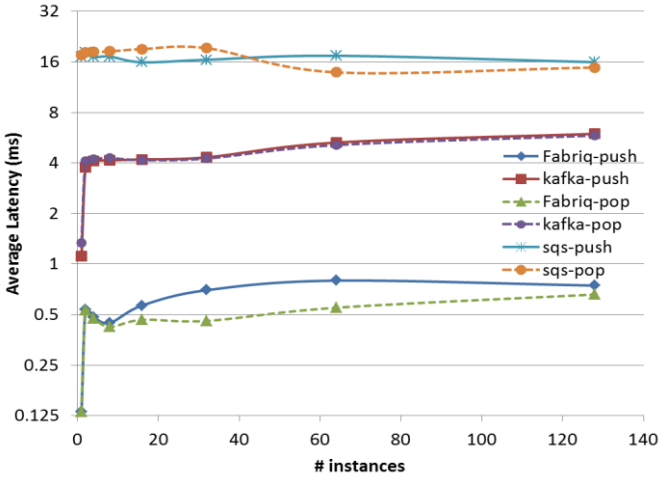

Fig. 7. Average latency of push and pop operations.

All the three systems show stable latency in larger scale. Fabriq provides the best latency among the three systems.

Since the communications are local at the scale of 1 for Kafka and Fabriq, they both show significantly lower latency than the other scales. We can notice that there is almost an order of magnitude difference between the average latency of Fabriq and the other two systems. In order to find out the reason behind this difference, we have generated the cumulative distribution on both push and pop operations for the scales of 64 and 128 instances. According to Fig. 8, at the 50 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.42ms, 1.03ms, and 11ms. However, the problem with the Kafka is having a long tail on latency. At the 90 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.89ms, 10.4ms, and 10.8ms. We can notice that the range of latency on Fabriq significantly shorter than the Kafka. At the 99.9 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 11.98ms, 543ms, and 202ms.

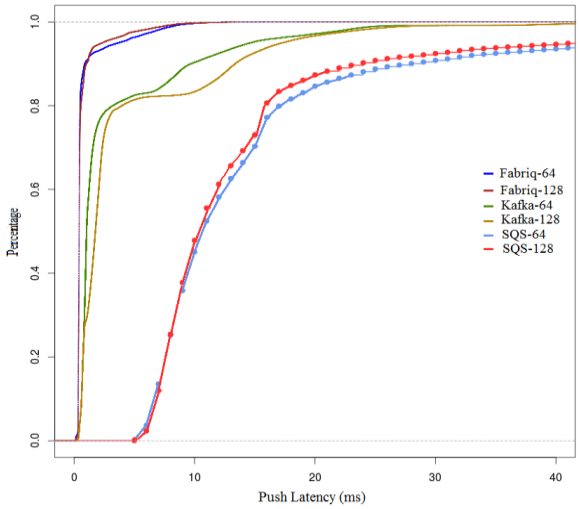

Fig. 8. Cumulative distribution of the push latency.

Similarly, Fig. 9 shows a long range on the pop operations for Kafka and SQS. The maximum pop operation time on the on Fabriq, Kafka, and SQS were respectively 25.5ms, 3221ms, and 512ms.

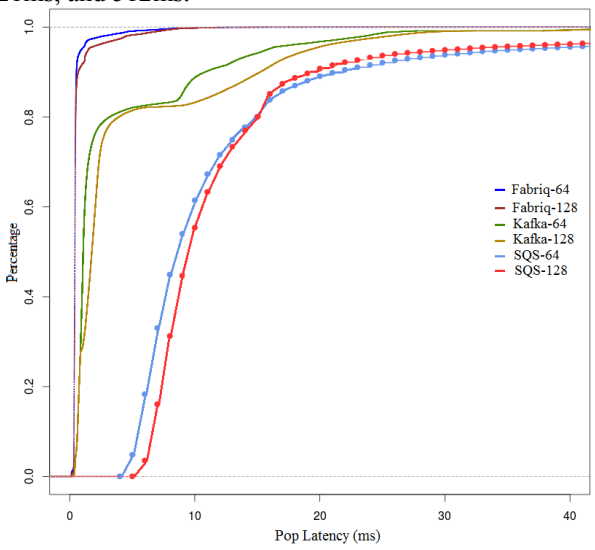

Fig. 9. Cumulative distribution of the pop latency.

As we observed on from the plots, Fabriq provides a more stable latency with a shorter range than the other two systems. Among the three systems, Kafka has the longest range of latency. There could be many reasons for the poor performance of Kafka. Before starting to produce or consume, each node needs to get the broker information from a centralized Zookeeper. In larger scales, this could cause a long wait for some of the nodes. Another reason for the long range of message delivery is the load imbalance. We have already discussed about it on the previous sections.

E. Throughput

It is substantial for a DMQ to provide high throughput in different scales. In this section, we compare the throughput of the three systems. We have chosen three different message sizes to cover small, medium and large messages. All of the experiments were run on 1 to 128 instances with a 1 to 1 mapping between the clients and servers in Fabriq and Kafka. In SQS, since the server is handled by AWS, we only run the client that includes producer and consumer on each instance.

Fig. 10 shows the throughput of both push and pop operations for the short messages. Each client sends and receives 1000 messages that are each 50 bytes long. Among the three systems, Fabriq provides the best throughput on both push and pop operations. As mentioned before, due to problems such as bad load distribution, and the problem of single access to the broker by the consumers, the throughput of Kafka is almost an order of magnitude lower than the Fabriq.

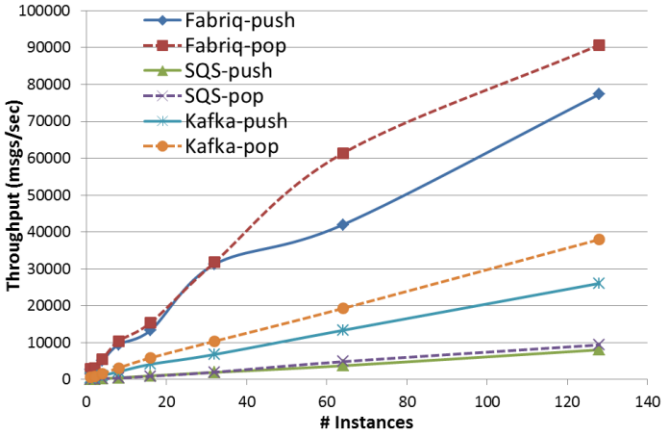

Fig. 10. Throughput for short (50 bytes) messages (msgs/sec).

All of the three systems are scaling almost nearly up to the scale of 128 instances. We can also notice that the throughput of pop operation is higher than the push operation. The reason for that in Fabriq is that the consumers first try to fetch the local messages. Also, in general we know that in a local system, the read operation is usually faster than the write operation. Fig. 11 compares the throughput of push and pop operations for medium (256KB) and large (1MB) messages. At the largest scale, Fabriq could achieve 1091 MB/sec on push operation and 1793 MB/sec on pop operation. We notice that the throughput of the Kafka for push and pop operations is respectively 759 MB/sec and 1433 MB/sec which is relatively close to what Fabriq can achieve. The reason for that is the continuous writing and reading on the same block of file instead of having separate files for different messages. This way, Kafka is able to deliver large messages with the minimum overhead. Therefore it performs well while delivering larger messages.

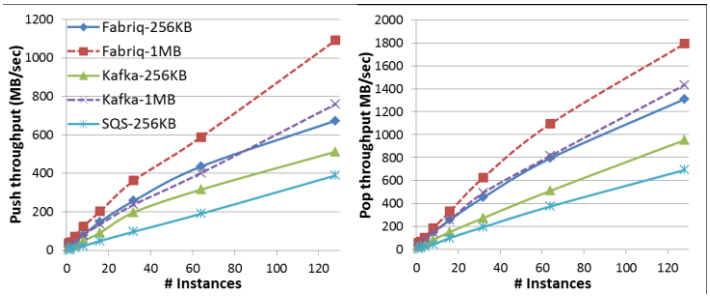

Fig. 11. Push and pop throughput for large messages (MB/sec).

VI. Conclusion and Future Work

A Distributed Message Queue can be an essential building block for distributed systems. A DMQ can be used as a middleware in a large scale distributed system that decouples different components from each other. It is essential for a DMQ to reduce the complexity of the workflow and to provide low overhead message delivery. We proposed Fabriq, a distributed message queue that runs on top a Distributed Hash Table. Fabriq was designed with the goal of achieving low latency and high throughput while maintaining the perfect load balance among its nodes. Servers in Fabriq are fully independent. The load of each queue is shared among all of the nodes of the Fabriq. This makes Fabriq achieve good load balance and high availability. The network communication protocol in Fabriq is tuned to provide low latency. A push operation could take 0 to 1 roundtrip communication between the servers. A pop operation takes 0, 1 or 3 operations for more than 99% of the operations.

The results show that Fabriq achieve higher efficiency and lower overhead than Kafka and SQS. The message delivery latency on SQS and Kafka is orders of magnitude larger than Fabriq. Moreover, they have a long range of push and pop latency which makes them unsuitable for applications that are sensitive to operations with long tails. Fabriq provides a very stable latency throughout the delivery. Results show that more than 90% of the operations take less than 0.9ms and more than 99% percent of the operations take less than 8.3ms in Fabriq. Fabriq also achieves high throughput is large scales for both small and large messages. At the scale of 128, Fabriq was able to achieve more than 90000 msgs/sec for small messages. At the same scale, Fabriq was able to deliver large messages at the speed of 1.8 GB/sec.

There are many directions for the future work of Fabriq. One of the directions is to provide message batching support in Fabriq. The latest version of ZHT which is under development supports message batching. We are going to integrate Fabriq with the latest version of ZHT and enable the batching support [38]. Another future direction of this work is to enable our network protocol to support two modes for different workflow scenarios. In this feature, the user will be able to choose between the two modes of heavy workflows with lots of messages, and a moderate workflow with less number of messages. We are going to optimize Fabriq for task scheduling purposes and leverage it in CloudKon [2] and MATRIX [26][39] which are both task scheduling and execution systems optimized for different environments and workflows. Finally, inspired by the work stealing technique used in MATRIX [37], we are planning to implement message-stealing on the servers in order to support pro-active dynamic load balancing of messages. Pro-active load balancing of the messages helps balancing the server loads when the message consumption is uneven.

References

[1] W. Vogels. " Improving the Cloud - More Efficient Queuing with SQS" [online] 2012, http://www.allthingsdistributed.com/2012/11/efficient-queueing-sqs.html

[2] I. Sadooghi, S. Palur, et al. "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing", Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID), 2014.

[3] J. Kreps, N. Narkhede, and J. Rao. "Kafka: A distributed messaging system for log processing". NetDB, 2011.

[4] A. Alten-Lorenz, Apache Flume, [online] 2013, https://cwiki.apache.org/FLUME/

[5] A. Thusoo, Z. Shao, et al, "Data warehousing and analytics infrastructure at facebook," in SIGMOD Conference, 2010, pp. 1013–1020.

[6] J. Pearce, Scribe, [online] https://github.com/facebookarchive/scribe

[7] T. J. Hacker and Z. Meglicki, "Using queue structures to improve job reliability," in Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC), 2007, pp. 43–54.

[8] I. Raicu, I. Foster, and Y. Zhao, " Many-task computing for grids and supercomputers", In MTAGS 2008, p. 1-11

[9] A. Videla and J. J. Williams, "RabbitMQ in action". Manning, 2012.

[10] B. Snyder, D. Bosanac, And R. Davies, "ActiveMQ in action" Manning, 2011.

[11] S. Davies, and P. Broadhurst, "WebSphere MQ V6 Fundamentals", IBM Redbooks, 2005

[12] Java Message Service Concepts, Oracle, [online] 2013, http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

[13] T. Li, X. Zhou, et. Al. "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in Proceedings of the IEEE IPDPS, 2013.

[14] Amazon SQS, [online] 2014, http://aws.amazon.com/sqs/

[15] D.Borthakur, HDFS architecture. Tech. rep., Apache Software Foundation, 2008.

[16] Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/

[17] P. Hunt, et. al., "ZooKeeper: wait-free coordination for internet-scale systems," Proceedings of USENIXATC'10, 2010.

[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. "Sparrow: distributed, low latency scheduling". Proceedings SOSP '13.

[19] M. Wall, "Big Data: Are you ready for blast-off", BBC Business, March 2014

[20] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: The twitter experience. SIGKDD Explorations, 14(2), 2012.

[21] A. Reda, Y. Park, M. Tiwari, C. Posse, and S. Shah. Metaphor: a system for related search recommendations. In CIKM, 2012.

[22] I. Raicu, P. Beckman, I. Foster. "Making a Case for Distributed File Systems at Exascale", Invited Paper, LSAP, 2011

[23] R. Ramesh, L. Hu, and K. Schwan. "Project Hoover: auto-scaling streaming map-reduce applications". Proceedings of (MBDS '12). ACM, USA, 7-12. 2012

[24] H. Liu, "Cutting mapreduce cost with spot market". 3rd USENIX Workshop on Hot Topics in Cloud Computing (2011).

[25] T. White, "Hadoop: The Definitive Guide." O'Reilly Media, Inc., 2009

[26] K. Wang, et. al. "Optimizing Load Balancing and Data-Locality with Data-aware Scheduling", IEEE Big Data 2014.

[27] Amazon DynamoDB. [online] 2014 http://aws.amazon.com/dynamodb/

[28] N. Liu, A. Haider, X.-H. Sun and D. Jin. "FatTreeSim: Modeling a Large-scale Fat-Tree Network for HPC Systems and Data Centers Using Parallel and Discrete Event Simulation," ACM SIGSIM PADS, 2015.

[29] N. Liu, et. al. "On the role of burst buffers in leadership-class storage systems," IEEE MSST conference, 2012

[30] I. Sadooghi, J. Martin, T. Li, I. Raicu, et. al. "Understanding the Performance and Potential of Cloud Computing for Scientific Applications", IEEE Transactions on Cloud Computing (TCC), 2015

[31] T. Li, et. al. "A Convergence of Distributed Key-Value Storage in Cloud Computing and Supercomputing", Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

[32] K. Wang, K. Qiao, I. Sadooghi, et. al. "Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales", Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

[33] T. Li, I. Raicu, L. Ramakrishnan, "Scalable State Management for Scientific Applications in the Cloud", BigData 2014

[34] T. Li, K. Keahey, K. Wang, D. Zhao, I. Raicu, "A Dynamically Scalable Cloud Data Infrastructure for Sensor Networks", ScienceCloud 2015

[35] D. Patel, F. Khasib, I. Sadooghi, I. Raicu. "Towards In-Order and Exactly Once Delivery using Hierarchical Distributed Message Queues", (SCRAMBL'14) 2014

[36] C. Dumitrescu, I. Raicu, I. Foster. "The Design, Usage, and Performance of GRUBER: A Grid uSLA-based Brokering Infrastructure", International Journal of Grid Computing, 2007

[37] K. Wang, K. Brandstatter, I. Raicu. "SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales", ACM HPC 2013

[38] T. Li, R. Verma, X. Duan, H. Jin, I. Raicu. "Exploring Distributed Hash Tables in High-End Computing", ACM Performance Evaluation Review (PER), 2011

[39] A. Rajendran, I. Raicu. "MATRIX: Many-Task Computing Execution Fabric for Extreme Scales", Illinois Institute of Technology, MS Thesis, 2013