HDMQ: Towards In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues

Dharmit Patel, Faraj Khasib, Shiva Srivastava, Iman Sadooghi, Ioan Raicu
dpatel74@hawk.iit.edu, fkhasib@hawk.iit.edu, ssriva10@hawk.iit.edu, isadoogh@iit.edu, iraicu@cs.iit.edu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

Abstract: **In today's world distributed message queues is used in many systems and play different roles such as content delivery, notification system and message delivery tools. It is important for the queue services to be able to deliver messages in larger scales, at the same time it must be highly scalable and provide parallel access at the same time. An example of a commercial state of the art distributed message queue is Amazon Simple Queuing Service (SQS). SQS is a distributed message delivery fabric that is highly scalable. It can queue unlimited number of short messages (maximum size: 256 KB) and deliver them to multiple users in parallel. In order to be able to provide such high throughput at large scales, SQS confines some of the features that are provided by traditional queues. SQS does not guarantee the order of the messages. Furthermore, it also does not guarantee the exactly once delivery as duplicate messages can be delivered. This paper addresses these limitations through the design and implementation of HDMQ, a hierarchical distributed message queue. HDMQ consist of collection of area message nodes that can be used to store messages up to 512 KB. It utilized round robin local load balancer to save the message and scale across the area region accordingly. HDMQ provides 1 replica for high reliability of messages. HDMQ provides SQS-like APIs in order to provide compatibility with current systems that currently use SQS. We performed a detailed performance evaluation and compared HDMQ to SQS measuring throughput, latency and price per request. We found HDMQ to outperform SQS by up to 10-20% in throughput, 100% in latency, and 50% less in costs.**

1 INTRODUCTION

Computing capacity of large-scale system is increasing at an exponential rate today and is expected to be in the order of Exascale Computing by 2018. Million of nodes and billion of threads of execution will be producing millions of messages [1]. As the size of these systems grow, the number and size of messages will also grow exponentially. There is a need for an effective message queue service to provide all the features needed by an application at an effective cost.

There are many effective ways available to manage these messages that rely different ways to manage but based on research they all compromise on certain feature of messaging, main criteria's that we considered while designing our system were a. Throughput, b. Latency, c. Cost, d. Message Order e. Reliability and f. Scalability and we found one or more of these to be missing from available system out there. The most popular message queue system Amazon SQS does not ensure message order and has a significant cost associated with it especially as the size of the systems grow larger to Exascale level [2]. We also looked at Hedwig [3] which is a publish-subscribe system designed to carry large amount of data across the Internet in a guaranteed-delivery fashion from those who produce it (publishers) to those who are interested in it (subscribers) [3]. Hedwig offers a lot of features but on system design analysis we found that all the message go through a single hub server (zookeeper) that save messages in a region where the order is maintained but messages could be stored in different regions and order is not maintained between regions. Also the hub nodes could limit the scalability of the system.

Based on our study on the available systems as discussed above we designed HDMQ (Hierarchical Distributed message Queue Service). The main goals of HDMQ are to provide high throughput, latency, message order, and reliability and be scalable. Our inspirations were primarily Hedwig and SQS. We designed this system that stores messages in storage nodes that are structured in an area style organization where each node is a part of a hierarchal region where the queue address would allow the front end nodes to direct the message to respective regions in hop where the lowest region level would maintain message order consistency for read and write operations. Our goal was also to make this system highly scalable and provide all the other features which we were able to do so as discussed in the results section.

2 BACKGROUND INFORMATION AND RELATED WORK

Distributed Message Queues now a days is used in many systems and play different roles such as content delivery, notification system and message delivery tools. It is important for the queue services to be able to deliver messages in larger scales and provide parallel access at the same time.

2.1 BACKGROUND INFORMATION

ActiveMQ is a message-oriented library, which ensures reliability between distributed processes. It is optimized to avoid overhead with a P2P or Server Client Model for pushing message to the receiver [6]. It uses its own communication protocol to ensure speed and reliability. They do communication between servers by simple message communication. With each node launch, node launches the server to listen to any incoming messages and handle them. Active MQ is highly configurable but it's slow and has issue

of lost/duplicate message. There are three kind of scaling available in Active MQ like Default Transport, Horizontal Scaling and Partitioning. It eventually crashes once per month [6].

Amazon SQS is a distributed, message delivery service, which is highly reliable, scalable, simple and secure [2]. SQS is distributed over multiple data centers so there is no single point of failure. SQS delivers and guarantees extremely high availability. It can deliver unlimited number of messages at any time. The size of the message cannot be more than 256 KB. And it ensures at least 1 delivery of the message. This tells us that every operation you do with the message is assumed as idempotent. SQS retains message up to 14 days. It also provides batching of messages up to 10 messages or 256 KB in total whichever is higher is applicable [2]. When a message is received, it becomes locked while being processed. This keeps other computer from processing the message simultaneously. If the message processing fails, the lock will expire and the message will be available again. In the case where the application needs more time for processing the lock timeout can be changed dynamically via the change message visibility operation. But it comes with a price tag of $0.50 for every 1M requests. It's not high price but it certainly isn't free either. It doesn't deliver message ordering [2].

Hedwig is a publish-subscribe system designed to carry large amounts of data across the Internet in a guaranteed-delivery fashion from those who produce it (publishers) to those who are interested in it (subscribers) [3]. The Hedwig is designed with the goal to give Guaranteed Delivery, Topic Based publisher and subscriber, Incremental Scalability and High availability. In Hedwig, clients publish messages associated with a topic, and they subscribe to a topic to receive all messages published with that topic. Clients are associated with (publish to and subscribe from) a Hedwig *instance* (also referred to as a *region*), which consists of a number of servers called *hubs*. The hubs partition up topic ownership among themselves, and all publishes and subscribes to a topic must be done to its owning hub [9]. When a client doesn't know the owning hub, it tries a default hub, which may redirect the client. Running a Hedwig instance requires a Zookeeper server and at least three Bookkeeper servers. Because all messages on a topic go through a single hub per region, all messages within a region are ordered. Providing global ordering is prohibitively expensive in the wide area. Hedwig client such as PNUTS, lack of global ordering is not a problem, as PNUTS serializes all updates to table row at a single designated master for that row. There is no ordering between different topics, as topics are independent. Version vectors are associated with each topic and serve as the identifiers for each message. Vectors consist of one component per region. A component value is the region's local sequence number on the topic, and is incremented each time a hub persists a message (published either locally or remotely) to BookKeeper[9]. They still need to implement more on how version vectors are to be used, and on maintaining vector-maxes [9].

Couch-RQS Queue system is based on database system, which is called Couch DB, which is basically a fast light weigh NOSQL DB [7]. The problem with this Library is that it is a primitive application and doesn't have significant components. It uses database to store its information and that's not going to give us better performance. It might be faster than any SQL Or NO-SQL database but that's not useful in commercial area where we deal with distributed environment. As their limitation is that Couch-RQS cannot run safely in a distributed/replicated environment and cannot scale high, cannot provide high availability [7].

Apache Kafka is publish subscribe messaging rethought as distributed commit log. It is very fast as a single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients [5]. It is also highly scalable as it is designed to allow single cluster to serve as the central backbone for large organization. It takes message from producers and feeds them to consumers. Each Kafka fiber maintains a partitioned log, Kafka cluster retains all messages whether they have been published or not. It relies heavily on the file system for storing cache messages. It is build on top of JVM [5]. Kafka nodes perform load balancing. It uses asynchronous messages sending. It uses traditional push pull model for messaging where data is pushed to the broker from the producer and pulled from the broker by the consumer. Kafka replicates its log information for each topic across a configurable number of servers to recover from failures. It performs cleaner log aggregation as it abstracts away the details of files and gives a cleaner abstraction of log or event as stream of messages. It is platform independent as it runs on JVM. The bottleneck of this system is not CPU or disk but network bandwidth particularly in the case of data pipeline that needs to send over data centers that is distributed over wide area network. It supports batch compression of messages [5].

Rabbit MQ is a robust messaging system for applications, it is open platform, which runs on all operating systems and supports a large number of client developer platforms [4]. It allows application to connect and scale using asynchronous messaging. It allows options to do tradeoff between performance, reliability, including persistence, delivery acknowledgements, publisher confirms and high availability. It offers Flexible Routing, user can setup simple routing or use bind exchanges or even use custom exchange type for routing [4]. It offers 'Mirroring' where queues can be mirrored across several machines ensuring that in the event of hardware failure, messages are safe. It offers management UI to monitor and control every aspect of message broker. It offers client in a variety of languages (C#, Java, clojure, erlang, Perl, python, ruby, PHP). It can report memory usage information for connections, queues, plugins and other processes in memory [4]. It can detect memory usage and can raise the memory alarm and block all connections until the memory alarm is cleared, and normal services are resumed. It ships in the ready to use state, and can be customized in environment variables, configuration file, runtime parameters and policies [4].

2.2 Related Work

There have been many distributed queue service implementations proposed over the years. We discuss Amazon

SQS in this section due to its wide use in commercial application. Amazon SQS is a distributed message service from Amazon. It is highly scalable and fast. Client is allowed to send message up to size 256 KB [2]. It ensures at least 1 delivery of message. Some of the other distributed queue services are RabbitMQ, Apache Kafka, Hedwig, Couch-RQS and Active MQ [3][4][5][6][7]. Most of these services are built and inspired from Amazon SQS.

Active MQ is a message broker written in JAVA together with a full support JMS client [6]. It was designed to support multiple languages using multiple protocols like AMQP, Stomp and OpenWire. This protocols together support multiple languages. Active MQ is highly configurable but it's slow and has issue of lost/duplicate messages. You have three kind of scaling available in Active MQ like Default Transport, Horizontal Scaling and Partitioning [6]. It eventually crashes once per month.

Couch-RQS solves all the limitations Amazon SQS provides but at the expense of requiring that you maintain Couch instance and that it only supports a single access-point (single master Couch DB instance), which limits the potential availability [7].

Apache Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design [5]. At a high level, producers send messages over the network to the Kafka cluster, which in turn serves them up to consumers [5].

Hedwig on other side is a publish-subscribe system designed to carry large amounts of data across the Internet in a guaranteed-delivery fashion from those who produce it (publishers) to those who are interested in it (subscribers). It has incremental scalability, high availability of messages, guaranteed delivery of messages and publishers and subscribers are topic based [3].

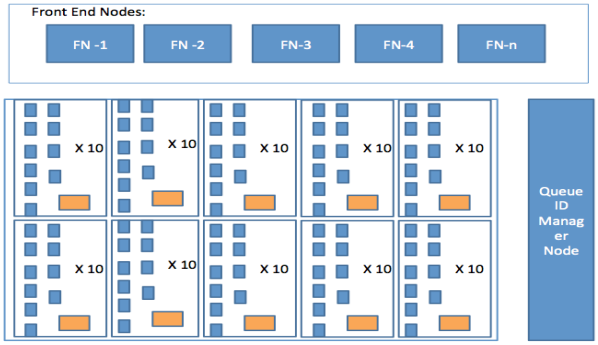
RabbitMQ is not a highly scalable queue. It also delivers message in unordered format and not FIFO [4]. Message can be delivered twice to subscribers. All the instances have same amount of overhead due to queues on every node in a cluster. From our point of view none of them provides a complete solution. All of them have some trade offs and are developed based on the requirement of the client [4].

3 DESIGNS AND IMPLEMENTATION OF HDMQ

We believe that by creating relationship between storage nodes and message queue we can provide features such as message order while still maintaining throughput and latency. In our design we have organized the storage nodes in an "Area" style hierarchy, where each node are part of hierarchal region. The main value of our design lies in the fact that we are able to achieve message localization of message storage for a queue within a sub region using "Area" style approach, which allows us to maintain message order and high throughput.

3.1 Architecture Overview

Figure 1: Hierarchical Style Message queue system:



We organized our system in three components:

A. Storage Nodes: All the storage in two hierarchical regions, where a sub region consists of ~10 nodes and a router node, the main region consists of multiple sub regions. All the main regions together make up the storage node system.

B. Front End Nodes: These are the nodes that clients interact with and make request to. Each front-end node maintains a local hash-table for that contains updates for "Area" for each queue ID. Currently we are using 10:1 ratio for number of storage nodes vs. front-end nodes.

C. Queue ID Manager Node: We use one queue ID node in the system that determines the storage region for new queues and generate area (queue ID) for the new nodes

Area: It defines the address for a set of nodes that are part of a sub region.

For example assume we have 10,0000 total storage nodes and x number of front-end nodes. This system will break down the nodes in regions and sub regions down to where each of lowest hierarchy region contain ~ 10 nodes. In this case we can divide 10,000 nodes in 10 regions of 1000 nodes (1 to 10), then each 1000 node in region of 100 nodes and this 100 node regions in set of 10 nodes. So for example node 2287 will have area – 2, 2, 8

3.2 Operation Overview

Write Operation: For insert operation the front-end node will route the messages to the given area where the router for the region will determine which node will be next for insert. This router will follow round robin insert strategy until all the 10 nodes in the region are full in which case incoming insert message will be routed to next available regions (to region 9 in above example). Front-end nodes will also maintain a hash table and when the write operation overflows to next regions they will be updated (In above example to 2,2,8:9, but the queue ID will remain the same and will act as the key in the front end node).

Read Operation: For read operation, front-end nodes use the area to determine the region where messages are stored for that queue, then they initiate read request to the router for that region to read messages. The messages are read again by the router using a round robin strategy hence maintaining the message order among different storage nodes, each storage node also follows round robin strategy to read messages hence maintaining overall message order. If there is overflow of messages to another region, then using updated queue id, front-end nodes are able to forward the read request to the overflow region.

Queue ID Manager Node: We will also have a queue ID manager node that will maintain the list of queue ID and generates new ID based on system load and assign initial area. We believe that this node will be low stress node and we only need 1 ~ 3 nodes to manage the system.

Replication: Synchronous Replication is provided for higher reliability. It can be configurable by the user whether one wants replication or not for the reliability of the message. Every message store on the original node is also copied in the replication node. As of right now there is only one replica of the message.

4 REFINEMENTS

Exactly one Delivery:
Only single copy of message is saved. There is no chance of getting two get requests for the same message. Once the message is delivered, the message is locked inside the node until it is delivered to the client. This doesn't mean that we don't store multiple copies of message. We store multiple copies of message for high reliability, but retrieve the other message when there is failure of a node. This is how the reliability is maintained in the system. Compared to Amazon SQS, our system offers Exactly one Delivery [2]. If we have Exactly one delivery functionality in Amazon SQS using DynamoDB as used in CloudKon, the performance of the Amazon SQS decreases by 30% [8].

Ordering of Message:
When the message comes in, the Router put the message inside the nodes that are in the section in round-robin fashion. So when there is a get request, the Router starts the delivery of message from the first node. If the first node doesn't have the message, then it will say empty queue. When the message is fetch from the queue, the information about where to get the next message is stored in the router. By default when the first message is fetched, the message is always fetched from the first node in the section. So if the incoming of message is so much or the section has high load and if the section nodes are full of messages, then the next incoming message will be saved in another section of the Area. This is done in two steps, (i) When the section is full of messages, the section is changed to the next available section (ii) An Atomic operation is performed where all the front end nodes are updated and paused for a small amount of time to get updated. Compared to Amazon SQS our system offers ordering of message while delivering [2].

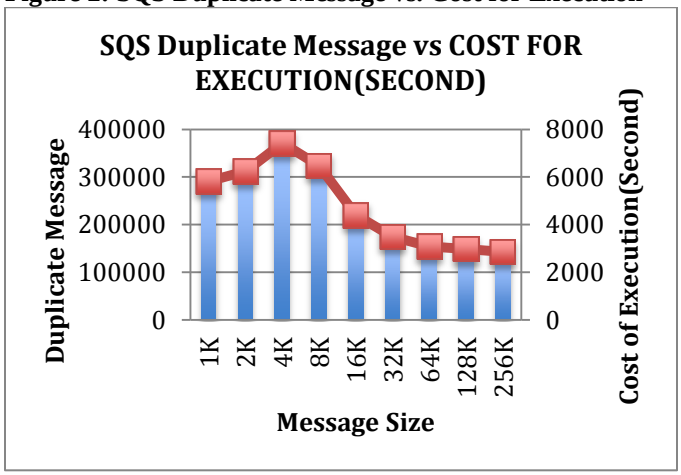
Large Message Size
Our System support a larger message size of 512 KB, as our design all depends upon the type of the nodes you select and the number of nodes you keep in one section. It doesn't depend upon the number of front-end nodes, or the number of section. Compared to Amazon SQS, our system offers double message size [2].

Mirrored Section Behavior
Each section is mirrored for the High Reliability of the message. So if any node fail or any section fail, we still have the message safe on another section or node.

5 PERFORMANCE EVALUATION

We evaluated Amazon SQS system using 20 client running on M1.xlarge and granularity from 1KB – 256 KB message size, submitted 1 million messages, and after submitting all the 1 million messages, the 20 Client start receiving the message from the very next time. The figure shows us the comparison between the SQS repeated messages and the overhead for the execution of the messages in seconds.

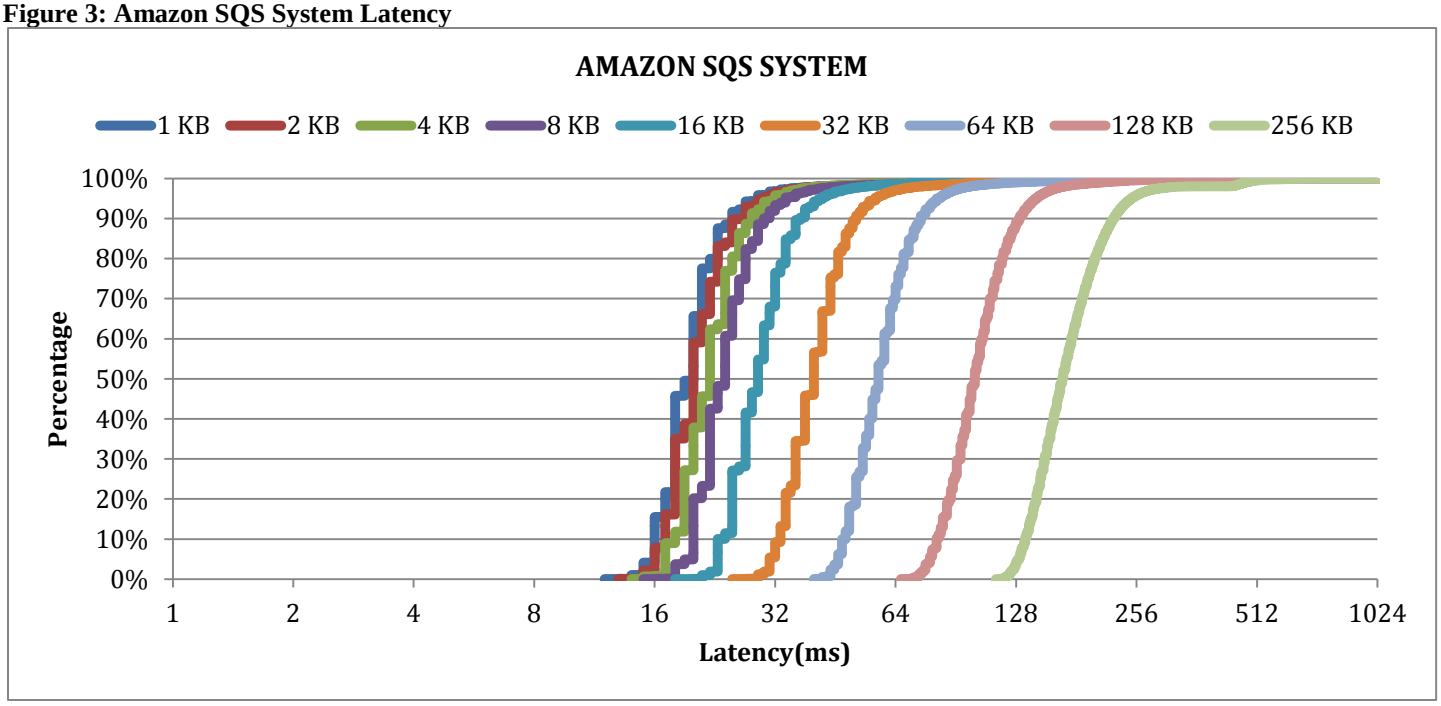
Figure 2: SQS Duplicate Message vs. Cost for Execution



According to the figure we observed that the overhead shown here in the graph is only the overhead of the SQS. If in a real system, if 1 message takes on an average 5 sec to execute, then this many number of message * 5 + SQS overhead for processing that message will give you the exact overhead of the whole system utilizing the SQS. If we take the average of all the repeated messages for all the granularity, we found that on an average 23.73 % of total messages are found in SQS as repeated messages, which is a big overhead to the system. This is just for the 1st million messages. After the delivery of repeated message we still will be having the repeated messages. So if we want to stop these repeated messages from SQS, we can use DynamoDB for handling the single delivery of message but it will probably decrease the performance of the whole system by 30 % as shown in the CloudKon[8].

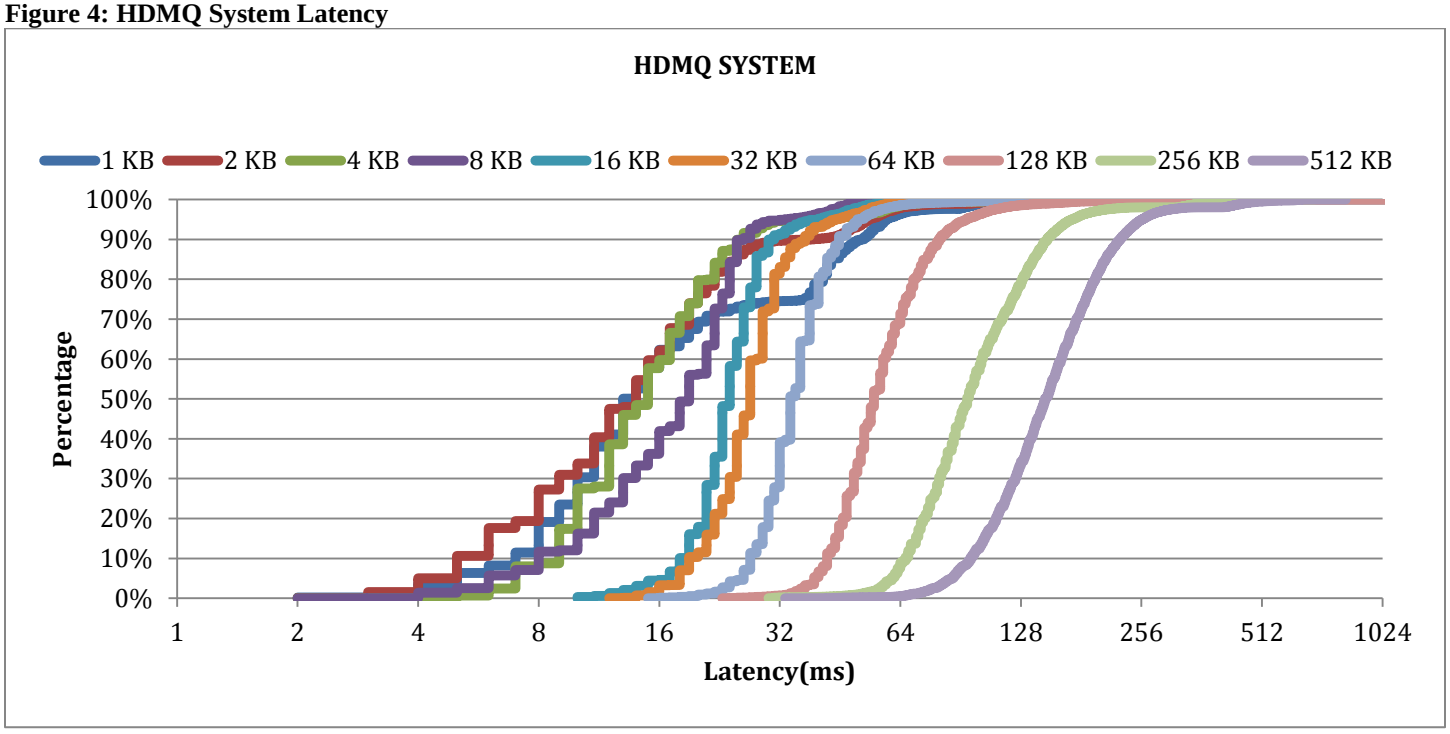
Comparison Between Adding 1 M Messages (Adding + Retrieving) to SQS vs. HDMQ

Figure 3: Amazon SQS System Latency



We evaluated Amazon SQS system using 20 clients running on M1.xlarge and granularity from 1KB – 256 KB message size, submitting 1 million messages. We observed that the 1KB message starts with 12ms minimum latency and reaches 20ms at 50%. But what is interesting is that the latency increases very fast after the 16KB message size. As you can see that 16KB starts with 18ms, 32KB with 27ms, 64KB with 40ms, 128KB with 68ms and 256KB with 114ms.

Figure 4: HDMQ System Latency



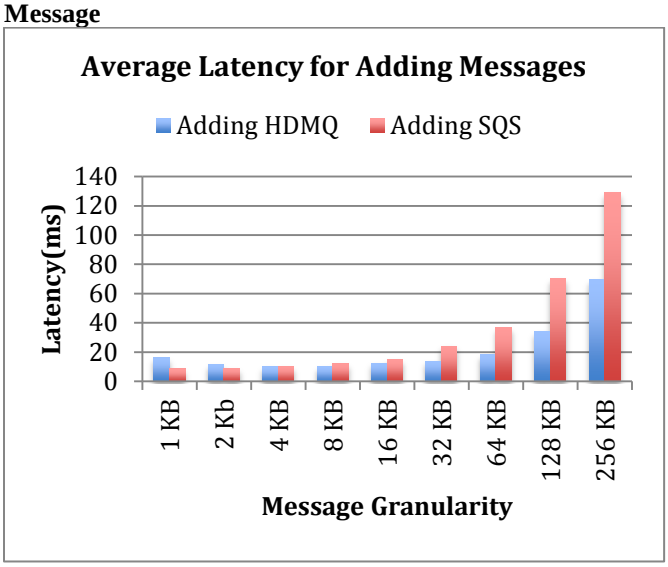
We evaluated HDMQ system using 20 clients running on m2.4xlarge and granularity from 1KB – 512 KB message Size, submitting 1 Million messages. We had 10 Front-end nodes, which were running on Amazon EC2 m1.xlarge Instance. We were using the Elastic load balancer to balance the load between the front-end nodes. We used the 20 M3 Double Extra Large Instance, 10 for the Storage nodes, and another 10 for backup storage nodes, which act as a RAID 1 to the actual storage node for replication. For configuration without replication we use only 10 M3 Double Extra Large Instance. We also used one m3.xlarge instance for local load Balancer. The above graph shown is the result for the system without replication. From the result we observed that our system has very less latency as compared to Amazon SQS. For e.g. for

1KB our system has latency as low as 2ms compared to 12ms of Amazon SQS. We also observed that the latency of our system as compared to Amazon SQS for 16KB, 32KB, 64KB, 128 KB, and 256 KB is less than their respective latency for SQS. We also observed that our system latency for 512KB message size is like 14ms at starting and goes up to 521ms at the end of the run. But SQS latency for 256KB message size starts from 114ms and goes up to 1019ms at the end of the run. We have like 100% less latency and still provide double the size of the message.

After comparing HDMQ with SQS we found out that, HDMQ did well at latency. We are also ensuring single delivery of message, we are also ensuring ordering of message, we are also not getting repeated message as we get on Amazon SQS. We also observed that if we compute the total time to execute this entire set of message, Amazon SQS will take 23.73 % more time to finish than HDMQ due to repeated task it has.

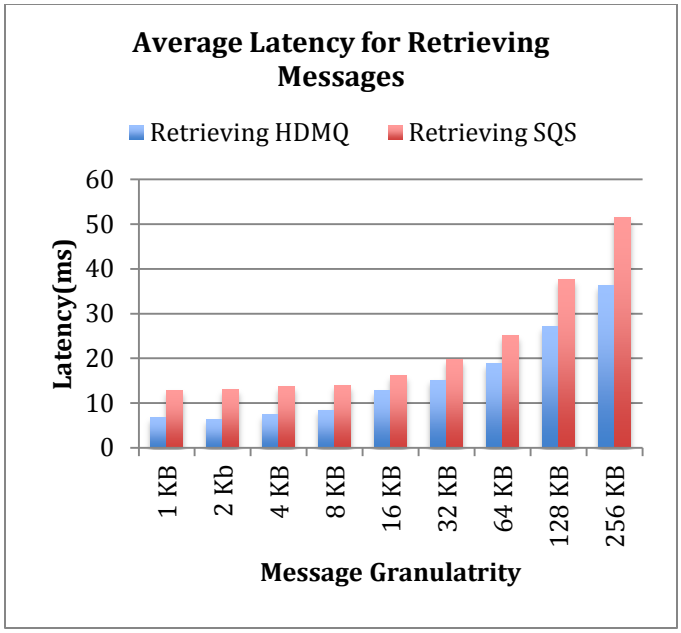
Comparison Between 1 M Messages to SQS vs. HDMQ in Latency, Throughput and Cost per request

Figure 5: Average Latency HDMQ vs. SQS for Adding Message



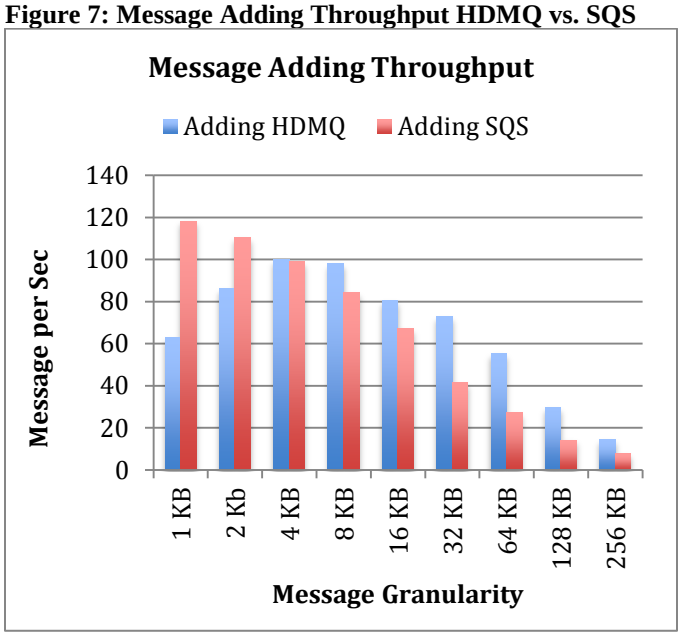
The average latency for adding the messages in HDMQ is also less than SQS other than 1 KB and 2 KB message size. We observed that adding 256KB message size is like 70ms for HDMQ against 129ms for SQS, which is almost double of HDMQ.

Figure 6: Average Latency HDMQ vs. SQS for Retrieving Message



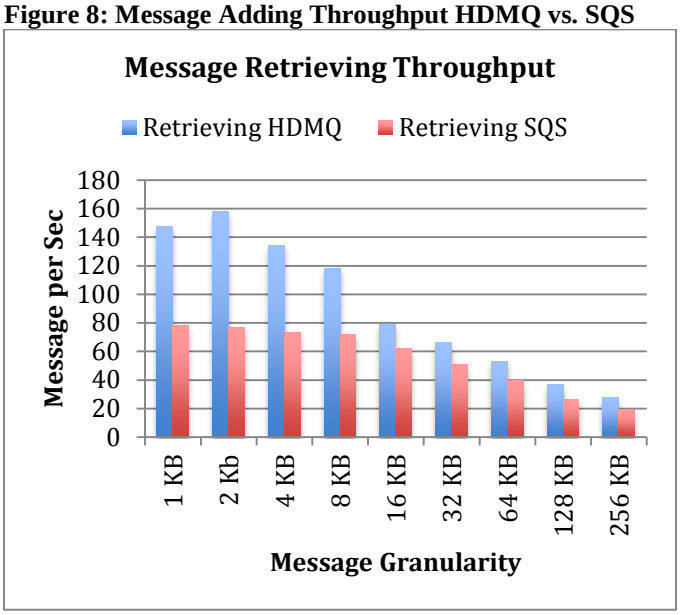
After comparing HDMQ and SQS for retrieving messages, we found out that HDMQ latency for the retrieving messages is so low due to one operation rather than SQS two operation that is retrieve and delete. We also observed that HDMQ latency is almost below 40ms as compared to 51ms of SQS. If we compare the 1k-message latency, HDMQ get 7ms while SQS gets nearly 13ms. The average latency for retrieving the message in HDMQ on and average is 30% less than the latency found in SQS.

Figure 7: Message Adding Throughput HDMQ vs. SQS



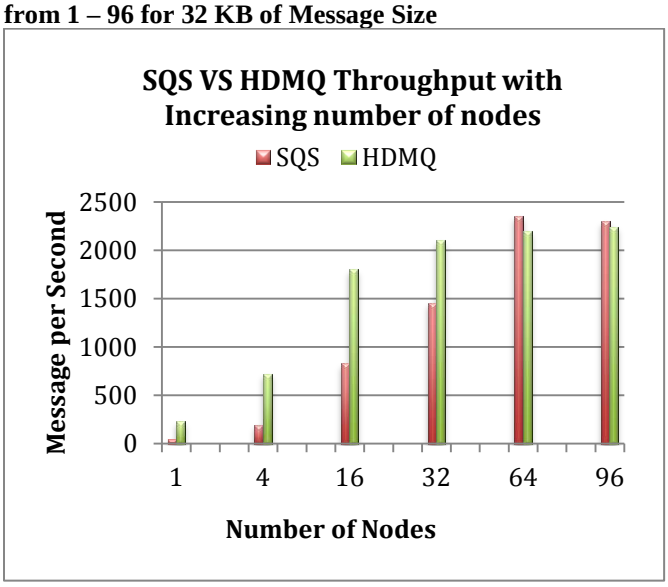
The Message Adding Throughput in HDMQ system is less than the message-adding throughput in SQS because our local load balancer is a node based load balancer. If we implement a router level load balancer, HDMQ would be much more faster than the SQS.

Figure 8: Message Adding Throughput HDMQ vs. SQS



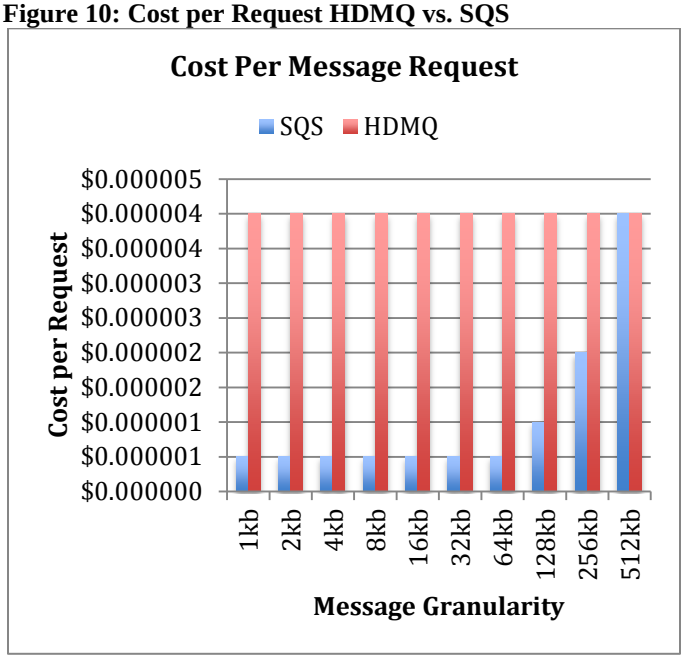
The message retrieving throughput is much higher and consistent than the amazon SQS because there are two operations in Amazon SQS, one is retrieve and second is delete, where else in our system we have only one call that is retrieve call. This greatly increases the combined throughput of our system.

Figure 9: Throughput for Different Number of nodes from 1 – 96 for 32 KB of Message Size



This graph shows the throughput of the message by summation of adding and receiving. From the above graph we can see that the throughput of the 32 KB message increases as the number of nodes increases. We got a max throughput of 2241 messages per second for HDMQ system vs. 2295 for Amazon SQS for 96 nodes. We also observed that the peak throughput for Amazon SQS is around 2352 Messages per sec for 64 nodes, but on the other side we observe that our system scales as the number of nodes increases rather than sleeping down.

Figure 10: Cost per Request HDMQ vs. SQS



As per our knowledge, we pay more than SQS, but SQS cost will remain constant for any granularity up to 64 KB, but after that the cost doubles with the double of message size. Our system cost more because we run our system on top of Amazon EC2 instances. So we end up paying more. But if we have our own hardware, we will probably cost less than SQS. On the other side, if we have knowledge based messaging service our cost can greatly reduce as compare to SQS, because then we would be having knowledge of the incoming message size and we can optimize the cost by having the low cost hardware machines from Amazon. We can further reduce the price by using the private cloud.

6 CONCLUSIONS AND FUTURE WORK

From the above work we conclude that, the HDMQ adding and retrieving latency is lower than the SQS latency. We also observed that Throughput for adding in HDMQ is little lower than the SQS system but if we implement the router level load balancer then the throughput would be much higher than SQS. We also observed that the average receiving throughput of HDMQ is much more higher than the average throughput of Amazon SQS. If we combine the average throughput of adding and receiving, HDMQ would be much more faster than Amazon SQS. We also observed that the throughput of HDMQ with increasing number of nodes is also higher than the Amazon SQS. We also conclude that the cost for implementing the system right now is little higher as we are implementing the system on top of Amazon Web Services using EC2 instance, but if we have message aware queue and our own private cloud, we can reduce that price by a great amount.

We will be implementing our own load balancer in future so that our framework is completely independent from Amazon Web Services. We will also implement queue-monitoring

service. We will also try to increase the Adding message throughput by implementing local router level load balancer. We will also provide more throughputs still maintaining the reliability by providing asynchronous replication. We also want to design the framework message aware so that it can scale according to the incoming message size. This will not only reduce the cost of operating per request but will also help us to be aware about the right storage node for the incoming messages size so that the system can scale itself. We will also try to configure the number of replicas for the message nodes. As of right now its by default 1 if you start the system with replication.

7 REFERENCES

[1] Dongfang Zhao and Ioan Raicu, Supporting Large Scale Data-Intensive Computing with the FusionFS Distributed File System, Technical Report, Illinois Institute of Technology, 2013

[2] Amazon SQS, [online] 2013, http://aws.amazon.com/sqs/

[3] Hedwig, [online] 2013, http://wiki.apache.org/hadoop/HedWig

[4] RabbitMQ in action: distributed messaging for everyone, Videla, Alvaro; Williams, Jason J W, Shelter Island NY : Manning, 2012. - 1288 p.

[5] Jay Kreps, Neha Narkhede and Jun Rao, Kafka: a Distributed Messaging System for Log Processing, 2011.

[6] Snyder, Bruce, Dejan Bosanac, and Rob Davies. "Introduction to Apache ActiveMQ." Active MQ in Action: 6-16.

[7] Couch-RQS, [online] 2013, https://code.google.com/p/couch-rqs/

[8] Iman Sadooghi, Ioan Raicu. "CloudKon: a Cloud enabled Distributed tasK executiON framework", Illinois Institute of Technology, Department of Computer Science, PhD Oral Qualifier, 2013

[9] Apache Hedwig [online] 2013, http://zookeeper.apache.org/bookkeeper/docs/r4.0.0/hedwigUser.html