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

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

Abstract: **In today's world, distributed message queues are used in many systems and play different roles (e.g. content delivery, notification system and message delivery tools). It is important for the queue services to be able to deliver messages at large scales with a variety of message sizes with high concurrency. 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 omits some of features that are provided by traditional queues. SQS does not guarantee the order of the messages, nor does it guarantee the exactly once delivery. 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 utilizes a round robin local load balancer to save the message and scale across the area region accordingly. HDMQ provides replication for high reliability of messages. It also 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 the commonly used commercial distributed queues measuring throughput, latency and price per request. We found HDMQ to outperform SQS, Windows Azure Service bus, and IronMQ by up to 2-15x times in throughput, 1.6-39x times in latency, and all this for 13%-80% less costs.**

Keywords-distributed message queues, FIFO message delivery, exactly-once delivery

I. INTRODUCTION

Computing capacity of large-scale system is increasing at an exponential rate and is expected to be on the order of exascale computing by 2019; millions of nodes and billions of threads of execution will be powering these future systems [1]. We argue that message queues are a fundamental building block for future distributed services and applications that aim to operate at these levels of concurrency. These message queues will likely have to be distributed, be asynchronous, support a variety of message sizes, guarantee message delivery, and support a variety of delivery ordering. As these systems grow in size, the number and size of messages will also grow. There is a need for an effective message queue service to provide all the features needed by an application at an effective cost that is architected for tomorrow's scales.

There are many effective ways available to manage these messages. But as we have found out, they all compromise on certain feature of messaging. The main criteria that we considered while designing our system were a. Throughput, b. Latency, c. Cost, d. Message Order, e. Reliability, f. Scalability and g. Single Delivery. We found one or more of these features to be missing from queuing system out there [1][19][22]. The most popular message queue system Amazon SQS does not ensure message order and has a significant cost associated with it as the size of the system grow larger [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 messages 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 the study of the available systems as discussed above, we designed HDMQ (Hierarchical Distributed message Queue Service) a highly scalable and reliable message queue service. The main goals of HDMQ are to provide high throughput, low latency, message order, high reliability and high scalability. 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 is to make this system highly scalable and provide all the features discussed earlier.

The main contributions of this work are:

1. ***Design and implement a highly scalable distributed queue service using hierarchical architecture that supports exactly once delivery, message order, large message size, and message resilience.***
2. ***Outperform SQS by 10-20 % in throughput and 2X in latency, with 50% less cost, all while providing a richer set of features.***
3. ***Performance evaluation comparing HDMQ with Amazon SQS, Windows Azure Service Bus and IronMQ.***

The remaining sections of this paper are as follows. Section II discusses about the design and implementation details of HDMQ as well as operations in HDMQ. Section III evaluates the performance of the HDMQ in different

aspects using different metrics. Section IV studies the related work in the area of distributed queuing systems. Section V discusses conclusion, limitations of the current work, and ideas for future work.

II. 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 is 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 implementing "Area" style approach, which allows us to maintain message order and high throughput.

A. Architecture Overview

In the figure we see 5 front-end nodes, 1 region containing 100 storage nodes, 10 sub-regions each containing 10 storage node and 1 router node. It also has 1 Queue id/Manager node.

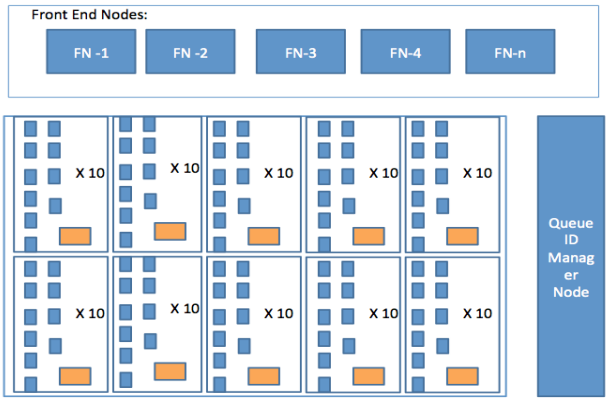

Figure 1 Hierarchical Style Message queue system

We organized our system in three components:

1) Storage Nodes

All the storage in two hierarchical regions, where a sub region consists of O (10) nodes and a router node, the main region consists of multiple sub regions. All the regions together make up the storage node system. The router node behaves like a real physical router with some intelligence to distribute the request according to request type either put or get message. This will ensure the order of the message ingoing and outgoing from the router.

2) Front End Nodes

These are the nodes that clients interact with and make request to. Each front-end node maintains a local hash-table that contains information about each queue location. Local hash table also receives updates for change in "sub-region" for queue ID. They always have the list of codes used by queue. Currently we are using 10:1 ratio for number of storage nodes vs. front-end nodes. We use load balancer for the front-end node to balance the load between the front-end nodes. The front-end node has a multithreaded connection with the router node inside the sub-regions to add and retrieve messages.

3) 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. This node also stores information about the already existing queue like area location. The queue id node is accessed at the time when a new queue is created. It is also accessed at the time when the nodes start adding the message or at the time when nodes start retrieving messages.

4) Area

It defines the address for a set of nodes that are part of a sub region. For example assume we have 1,000 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 contains O (10) nodes. In this case we can divide 1,000 nodes in 10 regions of 100 nodes (1 to 10), then each 100 node in each region will be set of 10 nodes. So we have 10 sub-regions per region. For example node 228 will have queue code – 2, 2, 8. If replication is on, each regions is divided into two parts, the upper half of the region is used to store the messages, the lower half is use to replicate the upper half. If replication is off, each region provides all 10 sub-regions to store the message.

B. Operation Overview

1) Write Operation

For insert operation the front-end node will use the queue-id to determine the region and sub-region and route the messages to the given sub-region where the router for the sub-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 sub-region are full in which case incoming insert message will be routed to next available sub-regions (to sub-region 9 in above example). Front-end nodes will also maintain a hash table and when the write operation overflows to next available sub- regions they will also be updated (In above example to 2,2,9 but the queue ID will remain the same and will act as the key in the front end node).

2) Read Operation

For read operation, front-end nodes use the queue-id to determine the region and sub-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 the sub-region get empty, front-end nodes will figure out whether the queue still have messages in other sub-region or not, if it has the messages then front-end nodes will reroute the read requests to another sub-region otherwise it will say empty queue.

3) Queue ID Operation

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.

4) 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. The ways the sub-regions are configured inside the region are based on whether replication is on/off.

5) Node Failure/ sub-region failure

While read or write operation, if any node fails, then router will provide the exact details about the location of the particular node to the front end node, and then front end node can access the replication node if available to fetch the messages. If the sub-region fails then the router node will no longer talk with the front-end node, and by this way the front-end node can directly move to the replication node if available to fetch the messages.

6) Concurrent Read/Write

Since the read request and the write request are handled separately, for e.g. a read request and a write request at the same time will work like this, A write request will go to the router node and ask for the location from where to start storing of messages. At the same time the read request will also ask the router node from which node the reading should start from. If there is no message in the system, read request will return null. This is how concurrent read/write is maintained.

C. Refinements

1) Exactly once Delivery

Only single copy of message is saved. 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. There is no chance of getting two get requests for the same message. When a HTTP message request comes in, a message is sent through HTTP response and the message is deleted at the same time. Since the read of the message is done on the atomic base, once the message is read, the message will be deleted from both the storage node and the replica node. This is all done on atomic base. At the same time a copy of message is locked in the front-end node until the message is successfully delivered to the client. In a case if router didn't get the message from the storage node, it will get it from replica node. 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]. Windows Azure service bus provides ***At Least Once Processing*** in case of failure. IronMQ provides exactly once delivery.

2) 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 retrieving 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 sub-region. So if the incoming of message is so much or the sub-region has high load and if the sub-region nodes are full of messages, then the next incoming message will be saved in another sub-region of the area. This is done in two steps, (I) When the sub-region is full of messages, the sub-region is changed to the next available sub-region (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 still providing exactly once message delivery [2].

3) 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].

4) Mirrored Sub-Region Behavior

Each sub-region is mirrored for the high reliability of the message. So if any node fails or any sub-region fails, we still have the message safe on another sub-region or node.

III. PERFORMANCE EVALUATION

We evaluated Amazon SQS system using 20 clients 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 clients start receiving the message from the very next time.

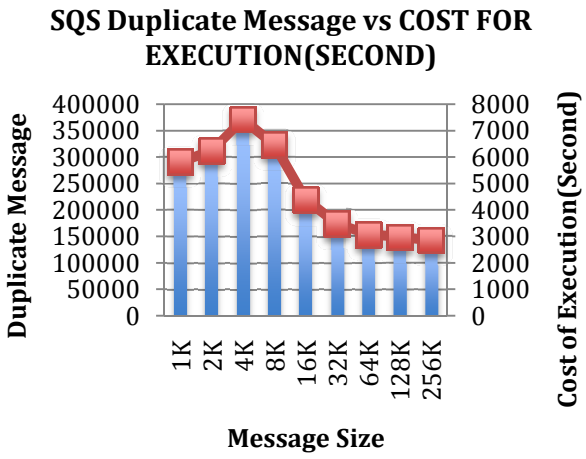

Figure 2 SQS Duplicate Message vs. Cost for Execution

The figure shows us the comparison between the SQS repeated messages and the overhead for the execution of the messages in seconds. According to the figure 2 we observed that the overhead shown here in the graph is only the

overhead of the SQS. 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 granularities, 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 DyanmoDB for handling the single delivery of message but it will probably decrease the performance of the whole system by 30% as shown in CloudKon [8].

We also evaluated Amazon SQS, HDMQ, IronMQ and Windows Azure service bus by adding and retrieving 1 M messages using 20 clients and with granularity of 1KB-512KB. Note that some system does not support the highest message sizes tested. The next several figures show the performance (measured in latency) comparison between running 1M message operations (adding+retrieving) to SQS, HDMQ, Windows Azure Service Bus, and IronMQ.

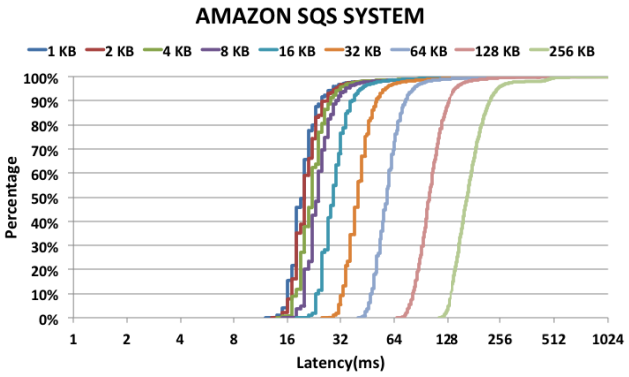

Figure 3 Amazon SQS System Latency

Figure 3 is a CDF graph for Amazon SQS. 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 is a CDF graph for HDMQ. We evaluated HDMQ system on Amazon cloud using 20 clients running on m2.4xlarge and granularity from 1KB – 512KB message size, submitting 1M messages. We had 10 front-end nodes, which were running on Amazon EC2 m1.xlarge Instance. We used the elastic load balancer to balance the load between the front-end nodes. We used the 20 M3 double extra large instances, 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.

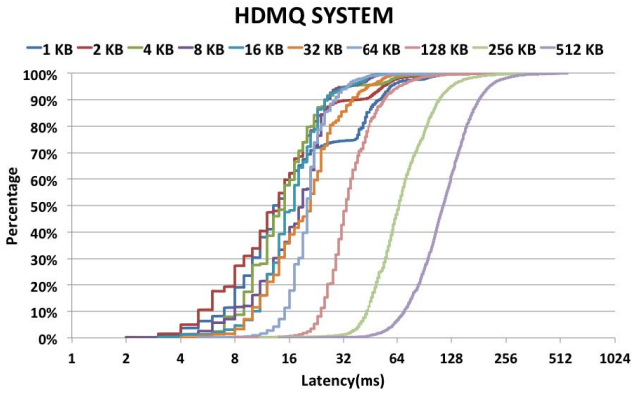

Figure 4 HDMQ System Latency

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. HDMQ has significantly less latency and still provides double the message size.

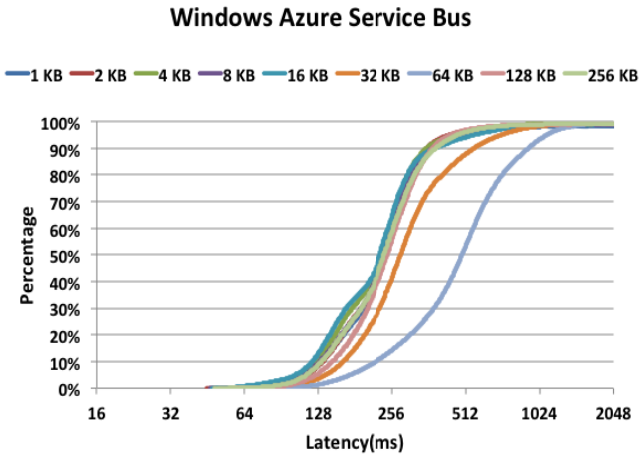

Figure 5 Windows Azure Service Bus System Latency

Figure 5 is a CDF graph for Windows Azure Service Bus. We evaluated Windows Azure Service Bus system using 20 clients running on medium (A2) instance from Windows azure virtual machine and granularity from 1KB – 256 KB message size, submitting 1 M messages. We observed that the 1KB latency starts from 46ms and reaches 239ms at 50%, which is like much more than 13ms for HDMQ at 50% for 1KB. We also observed that the latency

for 256KB starts from 48ms and goes to 243ms at 50%, which is likely much more than 93ms for HDMQ at 50% for 256KB. We also observed that the latency of 64KB is more than the other message sizes. 64 KB is the limit for the message size. We noticed that sending 64 KB messages, the system adds overhead (e.g. base64 encoding) to the messages. Therefore a message with 64 KB data size will not fit in one single packet and will be divided into two messages. Therefore the latency of the underlying process will be almost double of the latency of other message sizes.

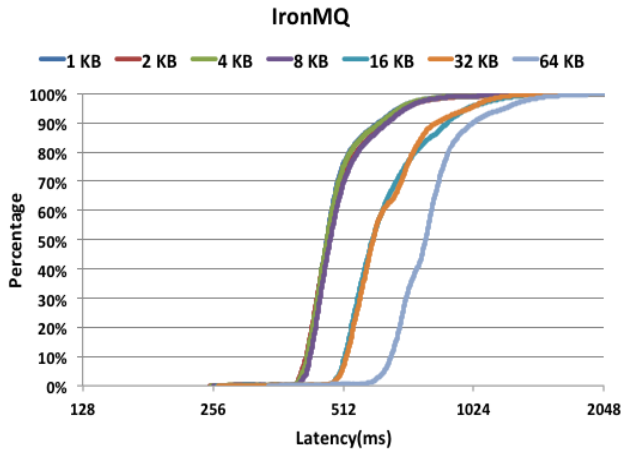

Figure 6 is a CDF graph for IronMQ.

We evaluated IronMQ system using 20 clients running on Amazon. IronMQ was configured to run on Amazon instances so it's better to put the clients on amazon. We tested messages with size from 1KB-64KB. We observed that 1KB latency starts from 253ms and reaches to 476ms at 50%, which is like much more than 13ms for HDMQ at 50% for 1KB. We also observed that the latency for IronMQ for 64KB starts from 342ms and reaches to 790ms at 50%, which is like much more than 35ms for HDMQ at 50%.

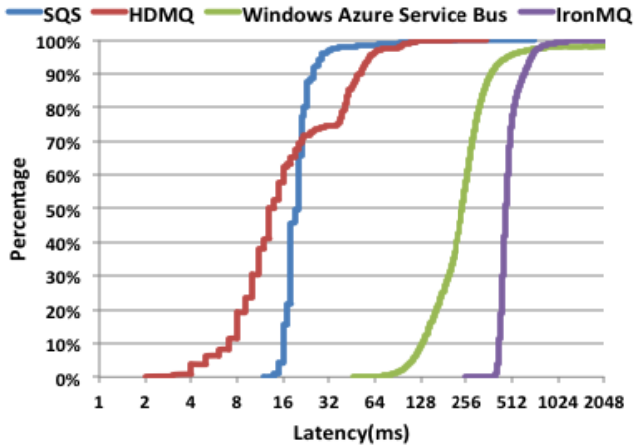

Figure 7 Comparison of HDMQ, SQS, Windows Azure Service Bus, and IronMQ with 1 KB message Size

What we observed was the overall latency for IronMQ is much more than HDMQ, Amazon SQS and Windows Azure Service Bus.

Figure 7 represent the CDF of 1KB message size comparing HDMQ to SQS, Windows Azure Service Bus and IronMQ. You can clearly see that HDMQ outperforms all other systems with small message sizes (e.g. 1 KB message size.

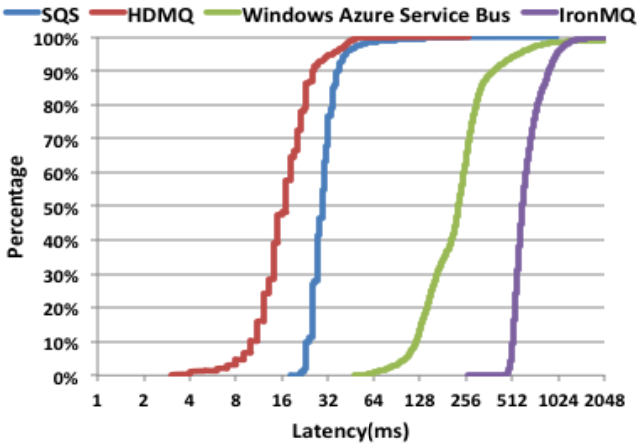

Figure 8 Comparison of HDMQ, SQS, Windows Azure Service Bus, and IronMQ with 16 KB message Size

Figure 8 represent the CDF of 16KB message size comparing four systems. We can observe from the graph that HDMQ is better than all other systems in the graph.

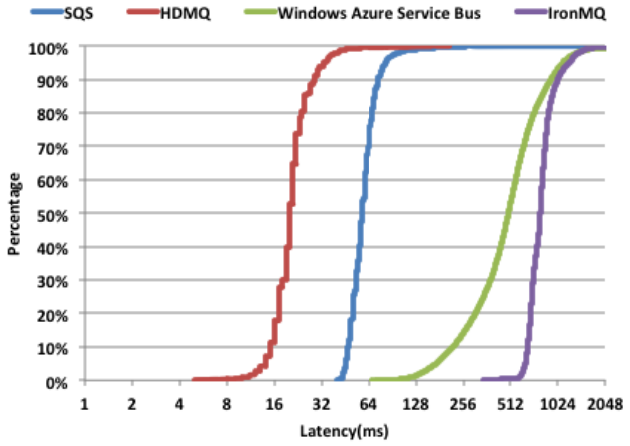

Figure 9 Comparison of HDMQ, SQS, Windows Azure Service Bus, and IronMQ with 64 KB message Size

Figure 9 shows the CDF comparison of four systems for 64KB message size. HDMQ is much more faster and has very low latency then other systems.

After comparing HDMQ with SQS, Windows Azure Service Bus and IronMQ 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 messages it has. HDMQ offers message size from 1KB-

512KB while SQS offers 1KB-256KB, Windows Azure Service Bus offers 1KB-256KB and IronMQ offers the lowest range of 1KB-64KB.

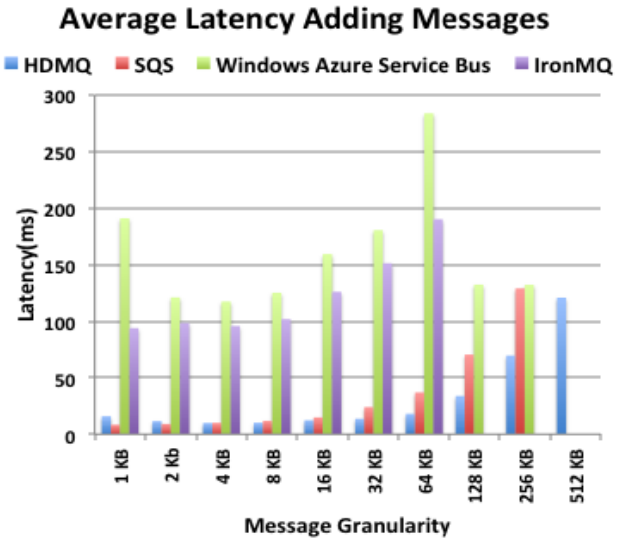

Figure 10 Average Latency HDMQ vs. SQS vs. IronMQ vs. Windows Azure Service Bus for Adding Message

Figure 10 shows the comparison of average latency for adding messages for HDMQ, SQS, Windows Azure Service Bus and IronMQ. The average latency for adding the messages in HDMQ is less than SQS other than 1KB and 2KB message size. We can also observe that the average latency for Windows Azure Service Bus and IronMQ is much more than HDMQ and SQS. We also observed that adding 256KB message size is like 70ms for HDMQ against 129ms for SQS and 132ms for Windows Azure Service Bus, which is almost double of HDMQ.

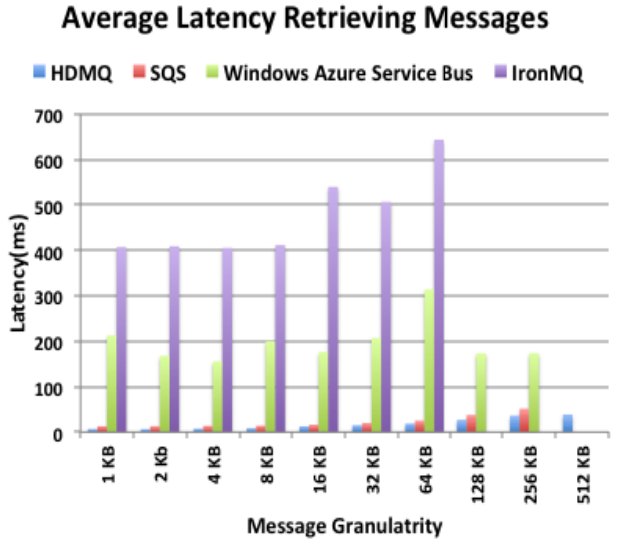

Figure 11 Average Latency HDMQ vs. SQS vs. IronMQ vs. Windows Azure Service Bus for Retrieving Message

Figure 11 shows the comparison of average latency for retrieving messages for HDMQ, SQS, Windows Azure Service Bus and IronMQ. 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 operations that is retrieve and delete, IronMQ two operations retrieve and delete and Windows Azure Service bus has two in one operation ***ReceiveandDelete***.

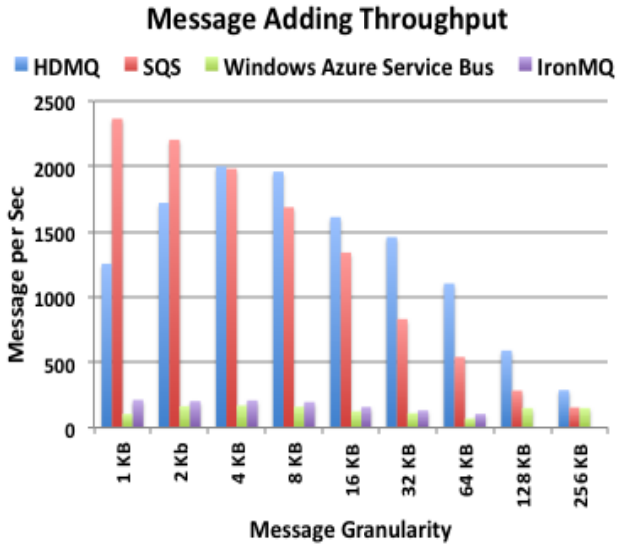

Figure 12 Message Adding Throughput HDMQ vs. SQS vs. IronMQ vs. Windows Azure Service Bus

We also observed that HDMQ latency is almost below 39ms as compared to 52ms for SQS, 313ms for Windows Azure Service Bus. If we compare the 1k-message latency, HDMQ get 7ms while SQS gets nearly 13ms, Windows Azure Service bus 212ms and IronMQ with 407ms. The average latency for retrieving the message in HDMQ on an average is 30% less than the latency found in SQS.

Figure 12 shows the comparison of message adding throughput for HDMQ, SQS, Windows Azure Service Bus and IronMQ. The message-adding throughput in HDMQ system is less than the message-adding throughput in SQS for 1KB; 2KB 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. Overall HDMQ is faster than Windows Azure Service Bus and IronMQ. They don't event compete with the HDMQ as well as Amazon SQS.

Figure 13 shows the comparison of average latency for retrieving messages for HDMQ, SQS, Windows Azure Service Bus and IronMQ. The message retrieving throughput is much higher and consistent than the amazon SQS, Windows Azure Service Bus and IronMQ because there are two operations in Amazon SQS, one is retrieve and second is delete, there are also two operation in IronMQ and Windows Azure Service bus has two in one operation ***ReceiveandDelete*** where else in our system we have only one call that is retrieve call. This greatly increases the

combined throughput of our system. We also observed that Windows Azure Service bus and IronMQ didn't compete with HDMQ and Amazon SQS, as their throughput is very low.

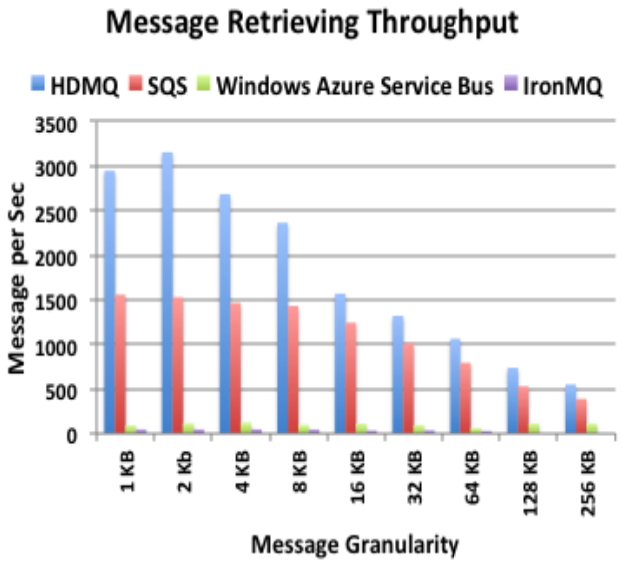

Figure 13 Message Adding Throughput HDMQ vs. SQS vs. IronMQ vs. Windows Azure Service Bus

Figure 14 shows the throughput of the message by increasing number of nodes. The throughput here is calculated by summation of adding and receiving throughput of particular system. This experiment was carried with considering the message size of 32 KB. The client was single node on which multiple numbers of threads were running. This experiment was carried to know the upper limit of the threads that can communicate with the system.

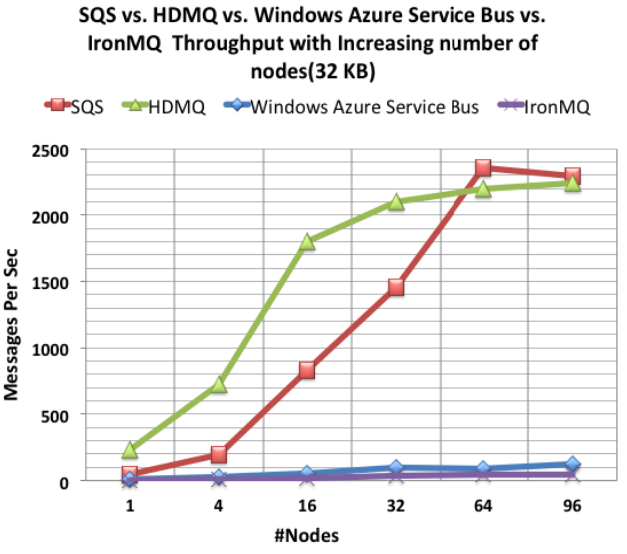

Figure 14 Throughput with increasing number of nodes with message size 32KB for HDMQ, IronMQ, Amazon SQS and Windows Azure Service Bus

From the figure 14 we can observe that the throughput of the 32 KB message size 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 vs. 124 for Windows Azure Service Bus vs. 42 for IronMQ 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. We also observed that Amazon SQS throughput starts sleeping down after 64 nodes.

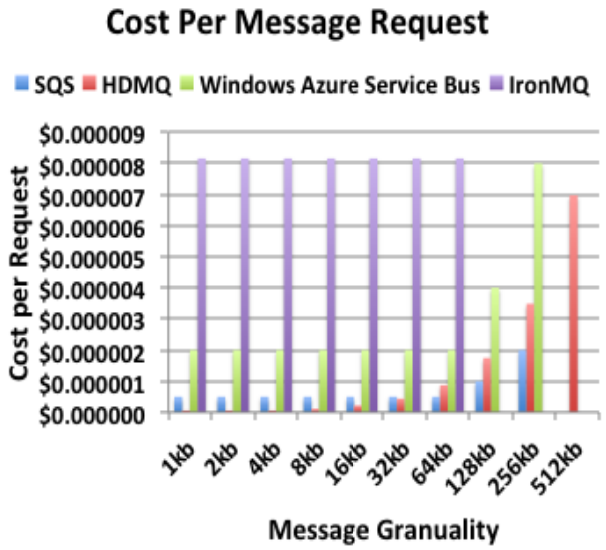

Figure 15 Cost per Request HDMQ vs. SQS

Figure 15 shows the message cost per request. As per our observation, we pay more than SQS for 64 KB – 512 KB, 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. We observed that IronMQ cost remains constant for any message size from 1KB- 64KB. The cost for Windows Azure Service Bus remains constant until 64KB and then increases for 128KB and 256KB. The reason for this is each message size is divided by 64 KB and then those numbers of request is taken. Windows Azure Service Bus also has a parameter called relay hours, which doubles the cost. 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. At last IronMQ is the costliest service followed by Windows Azure Service Bus followed by Amazon SQS and HDMQ.

IV. RELATED WORK

In this section we study each type of queue service available and point out their benefits and weaknesses compared to HDMQ. These distributed queue services discussed are Amazon SQS, Windows Azure Service Bus, IronMQ, 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 an open source message broker written in JAVA that was built for enterprise level application to provide enterprise features with full support JMS client [11][6]. Active MQ is a message-oriented library, which uses its own communication protocol, to ensure speed and reliability between distributed processes. It is optimized to avoid overhead with a P2P or server client model for pushing message to the receiver [6]. 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 [13]. 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 [12]. Active MQ performance and scalability mostly depends upon the topology used to configure it. As per the SPECjms2007, the result shows that horizontal configuration is performing very lower than the vertical configuration [14][15]. It was designed to support multiple languages using multiple protocols like AMQP, Stomp and OpenWire [12]. Active MQ can locked up or freeze for issues like JVM memory, broker memory, prefetching limit, producer flow control, and message cursors.

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 message delivery service, which is highly reliable, scalable, simple, secure and distributed over multiple data centers [2]. It 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. It 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 Amazon SQS 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 as well as it doesn't ensure single delivery [2].

Hedwig on the other side is a publish-subscribe system designed to carry large amounts of data from those who produce it (publishers) to those who are interested in it (subscribers) with the goal to give guaranteed delivery, topic based publisher and subscriber, incremental scalability and high availability [3]. 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 is an open source queue system built on top of Couch DB, a robust, fast and easy to use document-oriented database providing guaranteed message delivery [16]. Couch-RQS was inspired from Amazon SQS [2]. It provides FIFO, provides exactly once delivery [7]. It comes with the price tag of $0. It is ready to use service. It supports very large message payload size up to 4GB limited by available ram on the server [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]. 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 [17].

Apache Kafka is a high throughput distributed messaging system. It 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. It provides strong ordering of messages. It also provides three different kinds of delivery guarantees that are at most once, at least once and exactly once. Each Kafka fiber maintains a partitioned log, Kafka cluster retains all messages whether they have been published or not. Each partition can only handle one client in the group. So there cannot be more clients than the available number of partition. It relies heavily on the file system for storing cache messages. It is built 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]. The problem is Kafka only provides a total order over messages within a partition. So if we have thousands of clients we need thousands of partition. This leads to so many number of resources utilized.

Rabbit MQ is an open platform robust messaging system for applications, 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 [10]. It offers flexible routing, user can setup simple routing or use bind exchanges or even use custom exchange type for routing [4]. It also provides clustering, which helps RabbitMQ servers on a local network clustered together. 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].

Windows azure service bus is a messaging service from Microsoft that provides messaging channel for connecting cloud applications to the on-premises applications, service and systems providing 99.9% monthly SLA [18]. It exchanges messages in a loosely coupled way for improved scale and resiliency. Service Bus offers simple first in first out guaranteed message delivery and supports a range of standard protocols like REST, AMQP, WS* and APIs to put/pull messages on/off the queue. Service Bus Topics deliver messages to multiple subscriptions and easily fan out messages delivery at scale to downstream systems. Service Bus Relay allows on premise web services to project public endpoints. Queues offer First In, First Out (FIFO) message delivery to one or more competing consumers but FIFO behavior isn't guaranteed [22]. Messages can be received in any order [22]. Messages sent to the queue are in plain text or binary format, but are always received in Base64 encoded format [18]. A queue can contain an unlimited number of messages, each of which can be up to 256KB in size, with a maximum header size of 64KB [22]. Messages are stored only for seven days, after seven days, the messages are garbage-collected [22]. Receiving of messages can be work out in two different modes: ***ReceiveAndDelete*** and ***PeekLock***. For handling application crashes, it provides ***At Least Once Processing*** of messages that is it will redeliver the message again in case of failure. Service Bus provides both "relayed" and "brokered" messaging capabilities. In the relayed messaging pattern, the relay service supports direct one-way messaging, request/response messaging, and peer-to-peer messaging [18]. Brokered messaging provides durable, asynchronous messaging components such as Queues, Topics, and Subscriptions, with features that support publish-subscribe and temporal decoupling: senders and receivers do not have to be online at the same time; the messaging infrastructure reliably stores messages until the receiving party is ready to receive them. It comes with a price tag of $0.01 for every 10,000 messages. Messages exceeding 64KB in size will result in an additional message being charged for every 64KB in message [21]. They also have relay hours, which cost $0.10 for every 100 relay hours and $0.01 for every 10,000 messages. Relay hours start when the first listener connects to a given relay address and end when the last listener disconnects from the address and are rounded up to the next clock hour. The problem with this queuing service is that the queue size cannot be greater than 5GB[22].

IronMQ is a reliable message queue service that lets you connect systems and build distributed apps that scale effortlessly and eliminate any single point of failure [19]. It is easy to use highly available message queuing service that is built for distributed cloud application with critical messaging needs. It provides on-demand message queues with HTTPS transport, one-time FIFO delivery, message persistence, real time monitoring and cloud-optimized performance. It runs on cloud infrastructures like Amazon and Rackspace, uses multiple data centers for high-availability [20]. It uses reliable data stores for message durability and persistence. It is interoperable/No lock-in that provides maximum flexibility. It is highly scalable and has high performance. It also provides secure gateway using https and SSL. OAuth2 provides flexibility, scalability and security. It provides large set of libraries for different languages. It provides message size up to 64KB. It is billed by message request [19][20]. The price per message request varies from $0.00000096 to 0.00000816 as per the monthly plan.

V. CONCLUSIONS

From the above work we conclude that, the HDMQ adding and retrieving latency is lower than the SQS, Windows Azure Service Bus and IronMQ latency. We also observed that throughput for adding in HDMQ is little lower than the SQS system for 1KB and 2KB but if we implement the router level load balancer then the throughput would be much higher than SQS. HDMQ is also faster than Windows Azure Service Bus and IronMQ in message adding throughput. We also observed that the average receiving throughput of HDMQ is significantly higher than the average throughput of Amazon SQS, Windows Azure Service Bus and IronMQ. If we combine the average throughput of adding and receiving, HDMQ would be faster than Amazon SQS, Windows Azure Service Bus and IronMQ. We also observed that the throughput of HDMQ with increasing number of nodes is also higher than the Amazon SQS, Windows Azure Service Bus and IronMQ. We also conclude that the cost for implementing the system right now is little higher for message size greater than 64 KB as we are implementing the system on top of Amazon Web Services using EC2 instance, but if we have message aware queue and/or our own private cloud, we can reduce that price by a great amount. By this way we offer the cheapest distributed queue service, still offering lower latency and high throughput at the same time single delivery of each message still providing ordering of messages with high message size and providing replication for higher availability of messages. The only trade-off we observed was the limit of the network traffic both incoming and outgoing traffic that can be handled by router node.

As future work, we will be implementing our own load balancer in future so that our framework is completely independent from Amazon Web Services. We plan on extending a distributed NoSQL key/value store ZHT [23] to implement distributed message queues. We believe distributed message queues can be used to design future decentralized scheduling systems such as MATRIX [24, 25], which are very much at the heart of the realization of scalable support for a wide range of applications from Many-Task Computing [26, 27, 28, 29, 30, 31].

ACKNOWLEDGEMENTS

This work was possible in part due to the Amazon AWS Research Grants. This work was also possible in part due to the Windows Azure free trial credits and IronMQ free requests per month.

REFERENCES

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

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

[3] Hedwig, [online] 2014, http://wiki.apache.org/hadoop/HedWig

[4] Videla, Alvaro, and Jason JW Williams. RabbitMQ in action. Manning, 2012

[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] 2014, https://code.google.com/p/couch-rqs/

[8] Iman Sadooghi, et al. "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing", 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2014

[9] Apache Hedwig [online] 2014, http://zookeeper.apache.org/bookkeeper/docs/r4.0.0/hedwigUser.html

[10] Dawar, Sumit, et al. "Building a Scalable Event Processing System with Messaging and Policies–Test and Evaluation of RabbitMQ and Drools Expert."

[11] Henjes, Robert, et al. "Throughput performance of the ActiveMQ JMS server." Kommunikation in Verteilten Systemen (KiVS). Springer Berlin Heidelberg, 2007.

[12] Henjes, Robert, Michael Menth, and Valentin Himmler. "Impact of complex filters on the message throughput of the ActiveMQ JMS server." Managing Traffic Performance in Converged Networks. Springer Berlin Heidelberg, 2007. 192-203.

[13] Sachs, Kai, et al. "Benchmarking of message-oriented middleware." Proceedings of the Third ACM International Conference on Distributed Event-Based Systems. ACM, 2009.

[14] SPECjms2007 ® Horizontal Result, [online] 2007, http://www.spec.org/jms2007/results/res2009q4/jms2007-20090921-00010.html

[15] SPECjms2007 ® Vertical Result, [online] 2007, http://www.spec.org/jms2007/results/res2009q4/jms2007-20090921-00008.html

[16] Anderson, J. Chris, Jan Lehnardt, and Noah Slater. CouchDB: the definitive guide. O'Reilly, 2010.

[17] Comparison Couch-RQS vs. Amazon SQS, [online], 2010 https://code.google.com/p/couch-rqs/wiki/CompareRQSWithSQS

[18] Padhy, Rabi Prasad, Manas Ranjan Patra, and Suresh Chandra Satapathy. "Windows Azure Paas Cloud: An Overview." International Journal of Computer Application 2 (2012).

[19] IronMQ, [online] 2014, http://www.iron.io/mq

[20] IronMQ Pricing, [online] 2014, http://www.iron.io/pricing#mq

[21] Zhang, Qi, Lu Cheng, and Raouf Boutaba. "Cloud computing: state-of-the-art and research challenges." Journal of Internet Services and Applications 1.1 (2010): 7-18.

[22] Redkar, Tejaswi, and Tony Guidici. Windows Azure Platform. Apress, 2011

[23] Tonglin Li, et al. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013

[24] Anupam Rajendran, Ioan Raicu. "MATRIX: Many-Task Computing Execution Fabric for Extreme Scales", Department of Computer Science, Illinois Institute of Technology, MS Thesis, 2013

[25] Ke Wang, et al. "Paving the Road to Exascale with Many-Task Computing", Doctoral Showcase, IEEE/ACM Supercomputing 2012

[26] Yong Zhao, et al. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, ISBN: 978-1-60456-404-4, Nova Publisher 2008.

[27] Ioan Raicu, et al. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[28] Ioan Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", University of Chicago, Doctorate Dissertation, March 2009

[29] Ioan Raicu, et al. "Towards Data Intensive Many-Task Computing", book chapter in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", IGI Global Publishers, 2009

[30] Y. Zhao, I. Raicu, S. Lu, X. Fei. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE CyberC 2011

[31] Michael Wilde, et al. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", Poster Presentation, Scientific Discovery through Advanced Computing Conference (SciDAC09) 2009