Towards Distributed Message Queues using Distributed Key/Value Stores

Dharmit Patel, Iman Sadooghi, Ioan Raicu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA
dpatel74@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 report addresses these limitations through the design and implementation of ZDMQ, a distributed message queue using distributed key-value store. ZDMQ consist of collection of ZHT server that can be used to store messages up to 1 MB message size. ZDMQ provides replication of messages for high reliability. In the preliminary testing we performed evaluation and compared ZDMQ to the commonly used commercial distributed queues measuring throughput and latency. We found ZDMQ to outperform SQS, HDMQ, Windows Azure Service bus, and IronMQ by up to 1.86-351x times in throughput and 3.6-177x times in latency.**

Keywords-distributed message queues, exactly-once delivery, distributed key-value store, zero-hop distributed hash table.

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. 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. One domain that will greatly benefit from message queues is Many-Task Computing [1, 2], which aims to bridge the gap between HPC and High-Throughput Computing. Run-time systems to support parallel programming systems (e.g. Swift [3]) would greatly benefit from scalable message queue building blocks.

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. Single Delivery, e. Reliability and f. Scalability. We found one or more of these features to be missing from queuing system out there. 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. We also looked at HDMQ [4, 6], which is a distributed message queuing service, which offers single delivery of messages as well as ordering of message. HDMQ offers a lot of features but on system design analysis we found that all the messages go through a single router node that save messages in a region in round robin fashion where the order is maintained. Also bandwidth of the router nodes could limit the scalability of the system.

Based on the study of the available systems as discussed above, we designed ZDMQ (Zero Hop Distributed Message Queue) a highly scalable and reliable message queue service. The main goals of ZDMQ are to provide high throughput, low latency, single delivery high reliability and high scalability. Our inspirations were primarily SQS and HDMQ. We designed this system that stores messages in distributed key-value store that are structured in an multiple ZHT [5, 7] server where each ZHT server is a part of a storage where the queue messages will be stored in key-value manner. Our goal is to make this system highly scalable with very low latency and provide all the features discussed earlier.

II. Design

Data Structure:

- **a) UUID QUEUE:** UUID queue is used to store the uuid of the messages store in that ZHT server. Every message comes in to the server brings in the uuid with the message, same time the uuid is pushed to this queue, so that later we can pop that queue when a client do a pop operation and get the message. The max size of this queue can go upto the limit of the messages the ZHT server can store. It uses Queue collection.
- **b) META DATA LIST:** It is used to store information about the ZHT server containing the messages of QUEUE. It also store the first UUID of the message pushed to the ZHT Server. The ZHT Server also has the same UUID store in a variable called ***firstUUID***. We use list collection. There is only 1 Meta Data list per queue. The size of metadata queue can go up to the number of ZHT servers available.

Operation: There are mainly four operations in ZDMQ

1) ***Create Queue:*** Create queue operation in ZHTMQ takes queue name as an parameter, based on the hash of the queue-name, it will go on one of the ZHT server, where it will create the meta data list for that queue. Two queues with the same name are not allowed in ZHTMQ.
2) ***Push (Best 1 operation, Worst 2 operations):*** The Push operation in ZHTMQ will take message and queue-name as argument. Each push operation will generate a unique UUID. This UUID will get hashed and based on the hash value; it will send the message on one if the ZHT server. Now there are two possibilities possible

- **UUID Queue Present:** if UUID queue is present for that particular queue-name then it will push the UUID in to the UUID-queue and store the key-value pair in ZHT server. This is atomic operation. Both of this operation is done in one push call from client.
- **UUID Queue not present:** if UUID queue is not present then, it will create UUID queue in that ZHT server for that queue-name, at the same time, it will also update the firstUUID variable to identify that ZHT server for this specific queue. It will also push the UUID into the UUID-queue. It will lastly store the key-value into the ZHT. It will also return with the return code, which will notify the client to make another call to the ZHT server, which contains the Meta data list. This call will update the Meta data list with the first UUID pushed to the empty ZHT server. By this way we will always have information about the ZHT server storing the messages for particular queue.

3) ***Pop (Best 1 operation, Worst 3 operations):*** The Pop operation in ZHTMQ will take queue-name as an argument. Each pop operation will also create a unique UUID, which will get hashed, and it will send the pop request to the particular server. Thereby randomizing the pop operation and load balancing the pop request between the servers. Once the pop operation reached the server, there are three possibilities:

- **No UUID Queue->No Message found:** If no message is found and if there is no UUID queue, that means there was no message pushed to this server. So now it will call fetch_node() operation, which is internal to the pop() operation from client.
- **UUID Queue Present and Queue_size==0 => No Message found:** if no message is found and fi the UUID queue is present but it is empty, at this time, it will remove the UUID queue, it will reset the firstUUID variable and ZHT server will return with the firstUUID. Then client will make another call to the ZHT Server that contains Meta data list and pass firstUUID as an argument, it will remove the firstUUID entry from that Meta data list as that server no longer has the messages stored for that queue. Also in return it will fetch any random UUID from Meta data server for their next pop() call. After this the pop() will request this server for the messages until that server gets empty.
- **UUID Queue Present and Queue_size!=0 => Message found**: if UUID queue is present and the size is greater than zero, which means the messages are present on that ZHT Server. At this point, it will pop the UUID queue, and then do the lookup() operation of the message using the UUID, get the message, and then do the remove() operation and return the message to the client.
- ***Fetch Node Operation*:** This operation is of two types as follows
  - **Fetch node from Meta data List:** This operation is called only when the (a) type of possibility occurs in pop operation on server side.
  - **Remove node and Fetch node from Meta data List:** This operation is called only when (b) type of possibility occurs in pop operation on server side, when you have to remove the node, and fetch new node from the Meta data list to make further pop operation until that ZHT server is empty.

Delete Queue: Delete queue operation will remove all the UUID queues and the meta data list from the ZHT servers as well as it will remove all the messages using remove() operation inside the ZHT server.

III. Conclusion

- Light-weighted: cost less than 10MB memory/node
- Low latency: about 2-4ms
- Wide range of use: open source
- Very high Throughput
- Single Delivery
- Decentralized Architecture

IV. Future Work

- Benchmark ZDMQ from 1-1000 node scale on Amazon Cloud by varying number of threads from 1-96 threads per node with 1 million messages for message size from 1kb – 1mb.
- Integrate ZDMQ in Cloudkon.
- Add monitoring on ZDMQ.
- Compare ZDMQ with HDMQ, Amazon SQS, Windows Azure Service Bus and IronMQ.

V. References

[1] Ioan Raicu. Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing, University of Chicago, Computer Science Department, PhD Dissertation, 2009

[2] Ioan Raicu, Ian Foster, Yong Zhao, Alex Szalay, Philip Little, Christopher M. Moretti, Amitabh Chaudhary, Douglas Thain. "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

[3] Michael Wilde, Ioan Raicu, Allan Espinosa, Zhao Zhang, Ben Clifford, Mihael Hategan, Kamil Iskra, Pete Beckman, Ian Foster. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", Scientific Discovery through Advanced Computing Conference (SciDAC09) 2009

[4] Dharmit Patel, Faraj Khasib, Iman Sadooghi, Ioan Raicu. "Towards In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues", 1st International Workshop on Scalable Computing For Real-Time Big Data Applications (SCRAMBL'14) at IEEE/ACM CCGrid 2014

[5] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, Ioan Raicu. “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013

[6] Dharmit Patel, Faraj Khasib, Shiva Srivastava, Iman Sadooghi, Ioan Raicu. "HDMQ: Towards In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2013

[7] Tonglin Li, Raman Verma, Xi Duan, Hui Jin, Ioan Raicu. “Exploring Distributed Hash Tables in High-End Computing”, ACM Performance Evaluation Review (PER), 2011