Benchmarking Modern AMQP Message Brokers through Load Testing

Andrew Palmer and Diya Burman

August 7, 2017

Abstract

Message broker systems have become a popular architectural choice in distributed systems, providing greater network scalability and flexibility than the traditional client-server model. This recent rise in popularity has caused countless options of broker system configurations to emerge. Broker systems vary on dimensions such as message passing scheme, broker setup, and protocols used. Most brokers claim to be robust, fast and reliable, but sometimes these claims are unsubstantiated. In addition, it is difficult to figure out the right broker that satisfies an application's requirements, thus, it becomes necessary to paint a more vivid picture of the many options available.

This paper aims to give a clear view on the myriad of message broker system options available through a survey and also by benchmarking two popular systems, ActiveMQ and RabbitMQ under various workloads. These two brokers are investigated on their quality of service features such as tail latency and scalability.

1 Introduction

Network connected applications require a communication channel that enables data transfer. A lot of times applications may go down which creates the need to store the data to enable communication. In addition, there may be several communication bottlenecks due to the tight coupling of systems. One way to decouple these systems and allow greater levels of scalability is through a type of Message Oriented Middleware (MOM), message queues. A message queue allows programs to communicate with one another via message passing. A message queue stores data from the receiver when the sender is busy or down, and can provide quality of service mechanisms to reduce message delivery overload. A given queue holds data in sequential order and delivers to the receiver's end in the same order, although global ordering of messages is typically not guaranteed.

In software engineering, the publish–subscribe paradigm is one where message senders, i.e. publishers, do not directly send the messages to the receivers i.e. subscribers. Rather, publishers categorize the data into classes without the knowledge of the subscribers, if any. Similarly, subscribers express interest in one or more classes of data without any knowledge of the publishers if any. The publisher-subscriber paradigm is a brother paradigm of the message queue mechanism. Most messaging systems support both pub/sub mechanism and message queue models in their API, e.g. Java Message Service (JMS).

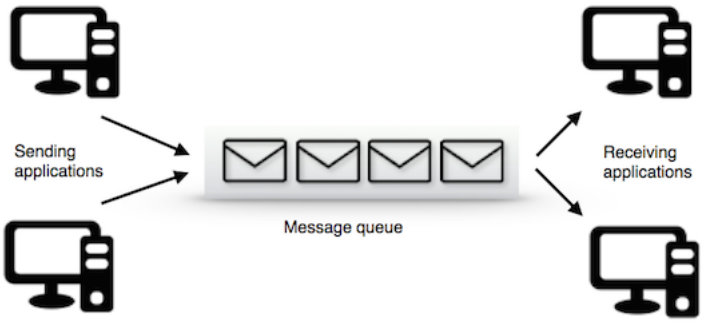

Figure 1: A basic view of a message queue between systems.

2 Research Questions

This report is guided by the following research questions:

1. What are the popular choices of message broker systems being used today?
2. How do existing implementations differ in their configurations?
3. How do two popular systems, ActiveMQ and RabbitMQ, handle throughput and latency under various workloads?

3 RabbitMQ and ActiveMQ

The basic architecture of a message queue can be so defined: a client called the producer sends messages to the queue. Another component called the consumer connects to the queue in order to retrieve the messages. The queue stores the messages until a consumer retrieves them. Refer Fig.1.

RabbitMQ is a popular open source message broker software that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and fail-over. Client libraries to interface with the broker are available for all major programming languages. RabbitMQ is one of the leading implementation of the AMQP protocol. The AMQP 0-9-1 Model has the following view of the world: messages are published to exchanges, which are often compared to post offices or mailboxes. Exchanges then distribute message copies to queues using rules called bindings. Then AMQP brokers either deliver messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand. This approach makes RabbitMQ very easy to use and deploy, because advanced scenarios like routing, load balancing or persistent message queuing are supported in just a few lines of code.

Apache ActiveMQ is an open source message broker developed using Java, providing libraries to create cross-language clients in any other language. It provides "Enterprise Features" which means aiding communication between any combination and number of clients and servers. ActiveMQ is enriched with features such as computer clustering and freedom to use any database as a JMS

	ActiveMQ / Apollo	RabbitMQ	ZeroMQ	Kafka	IronMQ	Apcahe Qpid
Brokerless/ Decentralized	No	No	Yes	Distributed	Distributed & Cloud Based	No
Clients	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others
Transaction	Yes	Yes	No	No. But can be implemented with plugin	No	Yes
Persistence/ Reliability	Yes (configurable)	Yes (built-in)	No persistence – requiring higher layers to manage persistence	Yes (built-in - File system)	Yes (built-in)	Yes (additional Plugin required)
Routing	Yes (easier to implment)	Yes (easier to implment)	Yes (complex to implment)	Can be implemented	No	No
Failover/ HA	Yes	Yes	No	Yes	Yes	Yes
Unlimited Queue	Yes	Yes	Yes	Yes	Yes	Yes
Scalabilty	Yes	Yes	Yes	Yes	Yes	Yes
Users	FuseSource, CSC, GatherPlace, UW Tech, Enterprise Carshare	Mozilla , AT&T, UIDAI		Linkedin,Twitter, Link Smart, Mozilla, Urban Airship,DataShift		
Licence/ Community	Apache (openSource)	Spring Source.Licenced under Mozilla Public License	IMatix . Licensed General Public License	Apache (openSource) . Initially developed by Linkedin.	Iron.io. Commercial	Apache (openSource)

Figure 2: Message Queue Comparison[9]

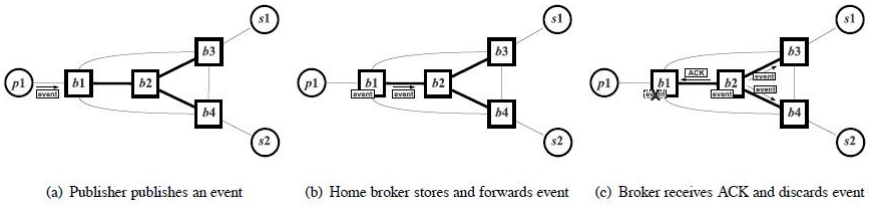

Figure 3: Example of event buffering/re-transmission scheme

persistence provider along with virtual memory and cache. It can deployed in both Peer-to-Peer and broker topologies. Similar to RabbitMQ, it's easier to implement advanced scenarios but it comes at the cost of raw performance. ActiveMQ can be operated in a Master/Slave configuration which has a benefit of keeping up operations even in the event of failure of a broker. ActiveMQ has a distinct feature of being very easy to configure in comparison to a lot of other (especially newer) message queues: subscribers and publishers connect to the broker and depending on the protocol being implemented data is sent from pubs to subs using the pull or push method.For the most part ActiveMQ is reliable and fast for Enterprise needs but seems to be getting increasing competition from newer and better performing brokers. Refer Fig.2 for a view of the comparison between some of the popular message queues.

4 Recent trends

In this section we are going to have a look at some of the research done in the last couple of years in the area of publish subscribe systems with regards to performance and quality of service features:

Pongthawornkamol et. al[1] propose a model that abstracts the basic publish/subscribe protocol along with several common fault tolerant techniques over large-scale, distributed networks. The overall goal is to predict the quality of service with regard to probability and timeliness of event delivery. In order to do so they propose a mathematical model to formulate the subscriber real-time reliability estimation problem. They then present a protocol-independent generic estimation algorithm to solve the problem. They further propose a set of protocol-dependent, pairwise publisher-subscriber reliability estimation models for each fault tolerance/recovery mechanism such as:

- Periodic Subscription: In periodic subscription scheme each subscriber periodically re-propagates its subscription message to its home broker who tends transmits it to the other brokers in the network. This helps avoid subscription losses but not event losses. More details about periodic subscription could be found in some of their previous work[5][6].

- Event Buffering/Re-transmission: Through this scheme each broker ensures event delivery to its next hop neighbor. While the mechanism ensures that events will not be lost due to broker or link failures, if the route is disconnected for too long the event may expire before it can be delivered to the subscribers. Refer Fig.3.

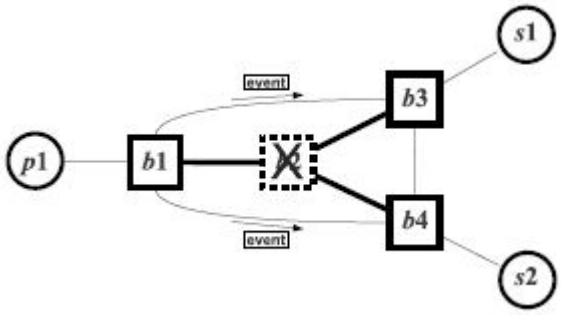

Figure 4: Example of path bypassing scheme where broker b1 bypasses failed broker b2

- Redundant Path Bypassing: This scheme perpetrates that if the routing path between a publisher and subscriber is interrupted due to a link or broker failure it might be possible to discover another publisher-subscriber path which excludes the failed link/broker[7][8]. Refer Fig.4.

Ionescu at the University of Pitesti[2], presents a comparison of ActiveMQ and RabbitMQ after testing their performance for both sending and receiving messages. As a result of the tests, he contends that ActiveMQ is faster on message reception i.e client sends message to broker and on the other hand, RabbitMQ is faster on producing messages i.e. client receiving messages from the broker. He attaches examples to help the reader understand under what circumstances which of the broker would be a good choice:

- GPS Applications for taxi companies should use RabbitMQ. In order to display the location of the automobile, the application transmits the co-ordinates to a database motor and from there back to the administrator when needed. This is an application which sends a lot of messages but the server is not interrogated as often.
- In medical systems which record and reference networks, the queues need to carry data to a database and store it there. When needed by the doctor/staff the data is pulled from the database. This application should use ActiveMQ because it wants better performance while loading messages to the broker.

Bellavista, et al.[3] survey state-of-the-art academic and industrial publisher-subscriber solutions with a particular focus on the scalability and reliability of these solutions. This is executed through a detailed technical analysis of different mechanisms and techniques employed to ensure and increment scalability and other QoS (Quality of Service) features, such as:

- **Subscription Models** such as the topic-based and *topic-based content-based* pub-sub systems. In topic-based, subscribers are partitioned into groups and each group is associated with a topic; thus making them simple to implement and enhancing routing efficiency whereas with content-based pub-sub systems give the subscribers a lot more flexibility but complicated the routing and matching functions.

- **Routing Models** such as *Centralized broker* which while simplifies the implementation of QoS properties, does not scale very well since a single broker has too much load and affects fault tolerance of the system as well. In contrast, an *Overlay of Brokers* organizes the MOM middleware as a network of peer brokers with matching-routing functions being executed via distributed algorithms - this improves system scalability.

  Et cetera.

The paper classifies QoS properties for pub-sub systems into three main dimensions from whose perspective to see each QoS property - which forms the basis for their analysis:

- **Granularity** Is the property enforceable with the granularity of a single publication or does it require consideration of a sequence of events?
- **Agreement Mode** Does the property require three-way agreements (e.g. publisher offered – subscriber requested – middleware confirmed) or just the end points (publisher offered – subscriber requested)
- **Quality Domain** What aspect of the pub-sub architecture does the property influence?

The research provides deep insight on delivery semantics, persistence in systems, latency, priorities and weak timing indications and event ordering.

5 Experiments

5.1 Setup

All experiments were conducted on 6 nodes of equivalent system specifications. Prior to evaluation, all nodes were synced to UTC through an Internet standard time server via the Network Time Protocol (NTP) protocol.

CPU	12 x AMD Opteron 4170 HE CPU @ 2.1GHz
Memory	16 GB
iPerf bandwidth	946 Mbps
Avg. ping over 1000 messages (64B)	0.116 ms
Avg. ping over 1000 messages (1KB)	0.157 ms
Disk write: 1M x 64B files	33.3 MB/s
Disk write: 1M x 1KB files	79.1 MB/s

5.2 Tail latency

Tail latency was benchmarked by consuming 1,000,000 messages of variable sizes, through the following procedure:

1. Synchronize publishers and subscribers
2. Send a message and record its timestamp t_0
3. Record the timestamp t_1 on the occurrence of the event being evaluated
4. Record the latency $t_1 - t_0$

The disk is only explicitly touched after the last message is sent/received and is done for logging purposes.

5.3 Scalability

For scalability, different configurations were considered as well. All experiments were also tested through consumption of 1,000,000 messages, and also examined under different ratios of producers to consumers, and different payload sizes. For scalability the average throughput of the 1,000,000 messages consumed was the value of interest.

5.4 Results

5.4.1 Tail Latency

The results of tail latency experiments with varying message sizes are as follows:

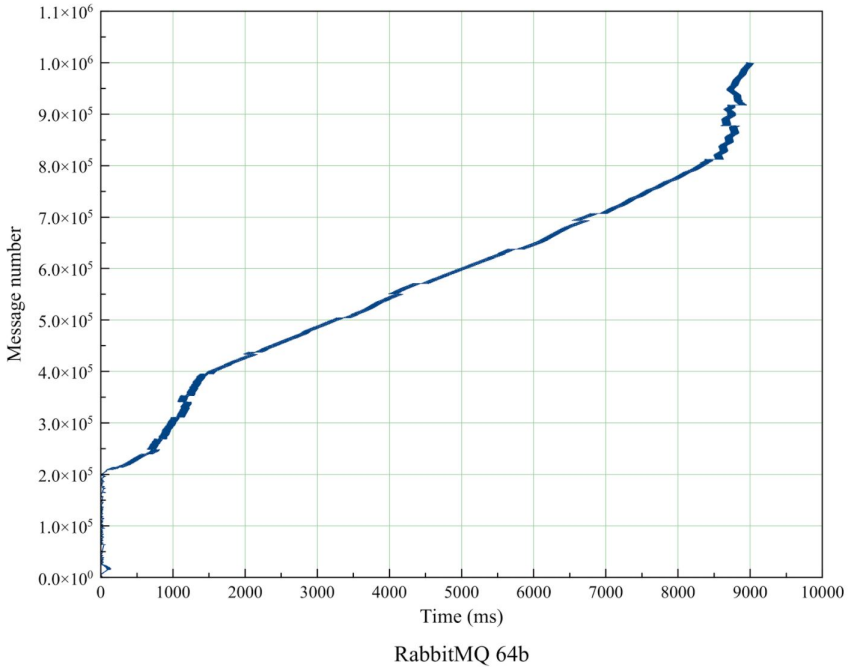

Figure 5: RabbitMQ Tail Latency with 1 million 64 byte messages

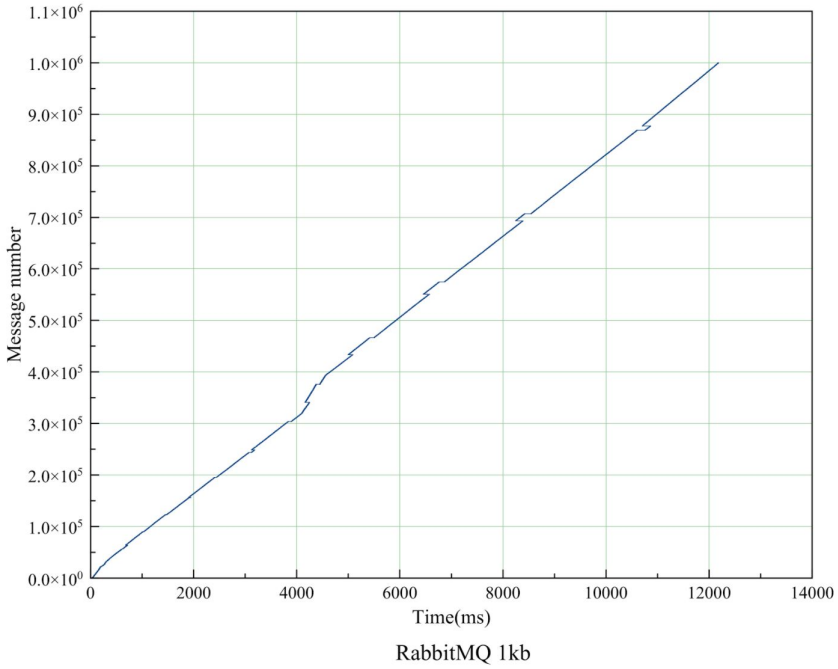

Figure 6: RabbitMQ Tail Latency with 1 million 1000 byte messages

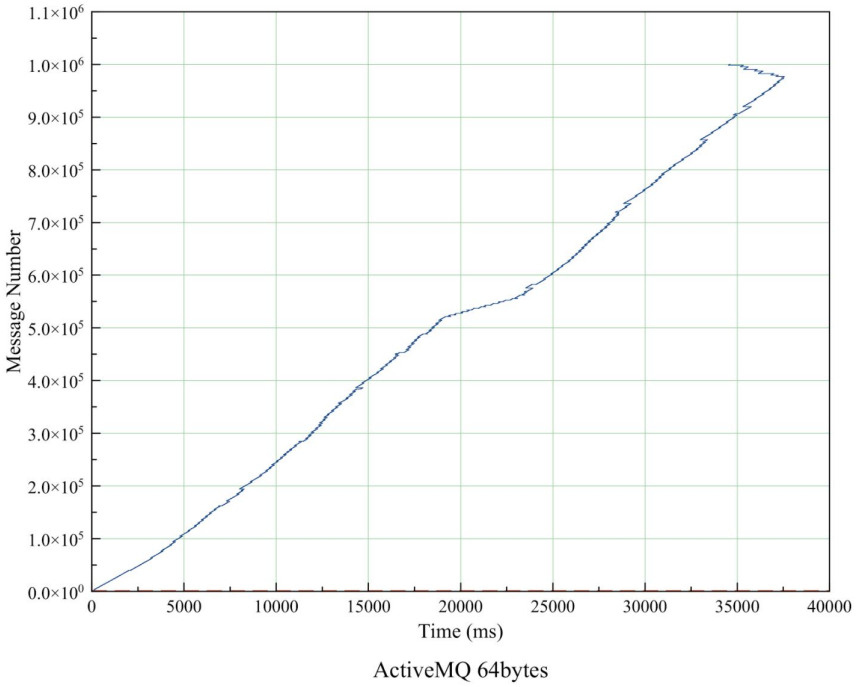

Figure 7: ActiveMQ Tail Latency with 1 million 64 byte messages-

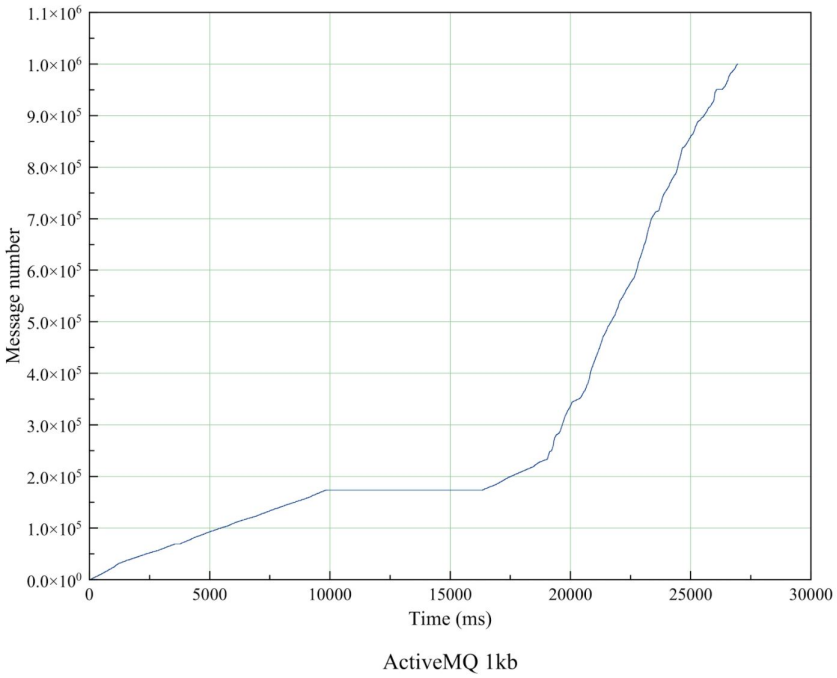

Figure 8: ActiveMQ Tail Latency with 1 million 1000 bytes messages

5.4.2 Scalability

The results of benchmarking RabbitMQ and ActiveMQ are as follows:

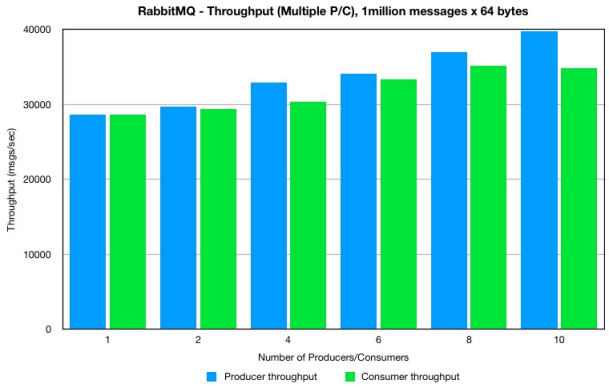

Figure 9: RabbitMQ throughput with 1 million 64 byte messages

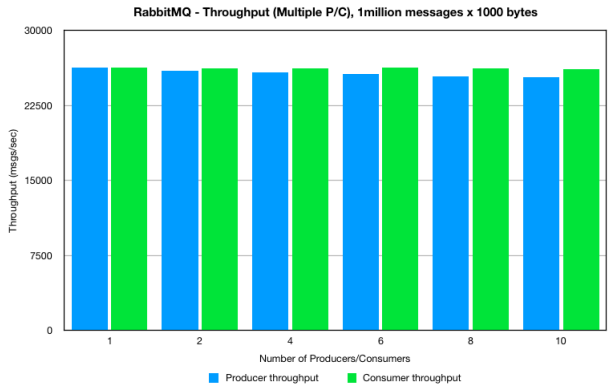

Figure 10: RabbitMQ throughput with 1 million 1000 byte messages

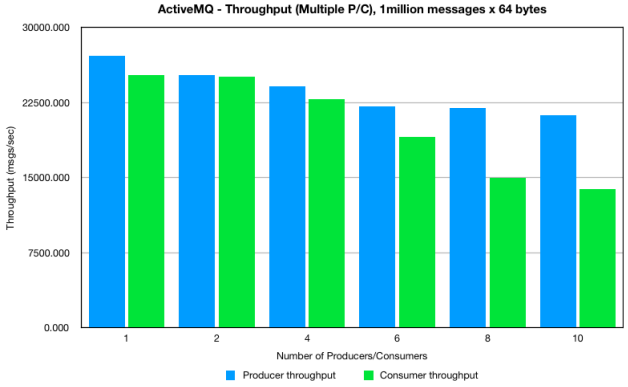

Figure 11: ActiveMQ throughput with 1 million 64 byte messages-

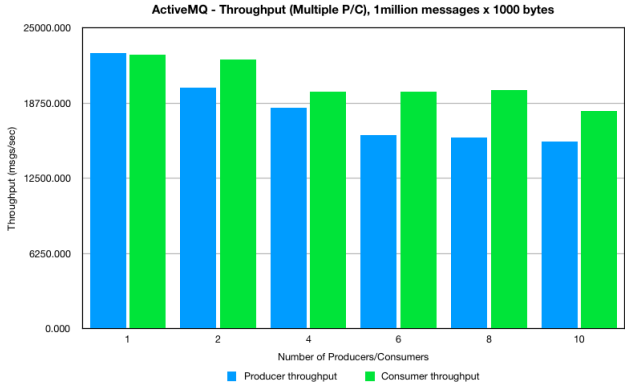

Figure 12: ActiveMQ throughput with 1 million 1000 bytes messages

5.5 Discussion

In case of the results of testing tail latency in Section 5.4.1:

- ActiveMQ shows some unexpected results of extremely slow times between sending and receiving. One possible explanation for this could be in what the ActiveMQ documentation speaks of as the slow consumers problem. Slow consumer especially affects non-durable queues i.e. non-persistent queues. Since messages are not persisted, messages are constantly sent to all the consumers and there needs to be extra configuration to keep the broker from overloading the slow consumer with messages which was not implemented in the current set of experiments.

- RabbitMQ also produced results contrary to our hypothesis. Although persistence was switched off across the board, the results indicate a general linear increase in latency as the number of messages increased. Our hypothesis is that this was due to a synchronous consumer, combined with RabbitMQ's default routing scheme. RabbitMQ has the option to route via header values which enables a much faster routing method.

In case of the results of testing scalability in Section 5.4.2:

- In RabbitMQ (Figures 9 and 10), there is an advantage to increasing the ratio of producers to consumers, which is more pronounced for small, bursty messages of size 64B. For these small messages, the total throughput at producers steadily increases with the quantity. Unfortunately it remains unclear as to whether or not this trend continues beyond 10 nodes, however the intention is to explore scalability for more extreme numbers of producers. This is less advantageous when the message size is significantly larger at 1kB, however an interesting thing to note here is that the throughput at both the producer and the consumer both decrease.

  With a smaller size file of 64 bytes, the throughput increases consistently with the increasing number of producers and as the push and pull rate of individual producer and consumer is pretty much the same. However, in case of the larger files of 1000 bytes, because there's a lot more time spent just transferring data rather than pulling messages, the rate of push by the individual producer suffers as compared to the rate of pull by individual consumer.

- In ActiveMQ, (Figures 11 and 12), for the small-sized messages, the trend is opposite that of RabbitMQ, i.e. as the number of producers increases, the worse is the throughput of both the producer and the consumer. However, in the case of larger-sized messages, expected results are seen wherein with increasing number of producers, the throughput of producers goes down more drastically as compared to that of the consumer, as more time is spent pushing data than pulling data in case of larger files. Again it remains unclear as to whether these trends will continue for nodes greater than 10 or if there are equal ratios of producers and consumers each turn.

6 Conclusion

While both ActiveMQ and RabbitMQ are quite comparable in terms of the results in our set of studies, we do believe that exploring different permutations of configurations in terms of persistency, routing schemes, and exchange types

(in the case of RabbitMQ) would yield more favourable results. In addition, there are many different combinations of Producer-Consumer setups that are left unexplored, but which could potentially yield more interesting results. The review of the recent trends reveal that message queues have quite a history and are now more than capable of aiding communication between large scale applications, with each open source/proprietary system offering its own set of rich features.

References

[1] Pongthawornkamol, Thadpong, Klara Nahrstedt, and Guijun Wang. "Reliability and Timeliness Analysis of Fault-tolerant Distributed Publish/Subscribe Systems." ICAC. 2013.

[2] Ionescu, Valeriu Manuel. "The analysis of the performance of RabbitMQ and ActiveMQ." RoEduNet International Conference-Networking in Education and Research (RoEduNet NER), 2015 14th. IEEE, 2015.

[3] Bellavista, Paolo, Antonio Corradi, and Andrea Reale. "Quality of Service in Wide Scale Publish—Subscribe Systems." IEEE Communications Surveys & Tutorials 16.3 (2014): 1591-1616.

[4] Esposito, Christian, Domenico Cotroneo, and Stefano Russo. "On reliability in publish/subscribe services." Computer Networks 57.5 (2013): 1318-1343.

[5] Jaeger, Michael A. "Self-managing publish/subscribe systems." (2008).

[6] JERZAK, Z., AND FETZER, C. Soft state in publish/subscribe. In DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (New York, NY, USA, 2009), ACM, pp. 1–12.

[7] Kazemzadeh, Reza Sherafat, and Hans-Arno Jacobsen. "Reliable and highly available distributed publish/subscribe service." Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on. IEEE, 2009.

[8] Kazemzadeh, Reza Sherafat, and Hans-Arno Jacobsen. "Opportunistic multipath forwarding in content-based publish/subscribe overlays." Proceedings of the 13th International Middleware Conference. Springer-Verlag New York, Inc., 2012.

[9] Message Queue Comparison - kuntalganguly.blogspot.ca/2014/08/message-queue-comparision.html