August 2019

Performance Comparison of Message Queue Methods

Sanika Raje
rajesanika7@gmail.com

PERFORMANCE COMPARISON OF MESSAGE QUEUE METHODS

By

Sanika N. Raje

Bachelor of Science - Information Technology
University of Mumbai
2012

Master in Computer Applications
University of Mumbai
2015

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2019

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

May 16, 2019

This thesis prepared by

Sanika N. Raje

entitled

Performance Comparison of Message Queue Methods

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Yoohwan Kim, Ph.D.
Examination Committee Chair

Ju-Yeon Jo, Ph.D.
Examination Committee Member

Fatma Nasoz, Ph.D.
Examination Committee Member

Sean Mulvenón, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

ABSTRACT

Message queues are queues of messages that facilitate communication between applications. A queue is a line of messages or events waiting to be handled in a sequential manner. A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed. For a distributed system to work, it needs to pass information between various machines. No single machine is responsible for the entire system, but all information is interrelated. Hence a major concern of distributed systems is this transfer of data. Which also proves to be one of the most significant challenges. Message Queues provide this asynchronous communication between applications. Major factors behind the success of an application is the ability to decouple and scale it.

In this thesis, we focus on analyzing and comparing the performance of three most widely used open source message brokers namely Apache ActiveMQ, RabbitMQ and Apache Kafka which help in creating a distributed system. An end to end message queuing model is setup for each of the brokers to mimic real world application models. The producers, consumers and brokers that make up the message queuing system are then put through rigorous benchmarking tests to analyze their performance. The performance is evaluated based on major factors like throughput, latency and total time taken by the transaction. Based on the benchmarking results, it was observed that Apache Kafka which was initially developed to be a message queue but later enhanced to be a streaming platform outdid RabbitMQ and Apache ActiveMQ in almost all the performance factors. It was also observed that the larger the message size, more constant is the performance of all message brokers. Hence, for gauging the performance in hard times, the message sizes considered for the experiments is very small. This gives us a glimpse of the actual performance capabilities of the message queuing brokers.

ACKNOWLEDGEMENTS

I would like to express my gratitude towards Dr. Yoohwan Kim for his continuous guidance throughout my graduate studies at University of Nevada, Las Vegas. I thank you for being my thesis committee chair. I derived confidence from the way you motivated me to move forward in the right direction of research. This immensely helped to broaden my work. Without your advice this thesis would not have had such a rich content.

I am indebted to Dr. Ju-Yeon Jo, Dr. Fatma Nasoz, and Dr. Sean Mulvenon for being my thesis committee members. I am highly grateful for teaching me the complex concepts in a simplified way. With your support when I needed it, this thesis has taken a more meaningful shape. For this and for being generous with the availability during the hours of my need.

Special thanks to the faculty at the Department of Computer Science, University of Nevada Las Vegas for providing me advanced knowledge essential for a master’s degree student along with the financial support.

I would like to extend sincere appreciation towards my family for being the pillars of strength through thick and thin and always encouraging me to dream big and then strive for it. Without their help I could never have the capacity to reach where I am today in my life. Last but not the least; I thank my friends for their help and support towards completion of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION

Message queuing has been used in processing of data for many years. One of its most common and widespread use is in E-mails. When you talk with someone on the phone, it is a synchronous communication. Both parties must be present and have a connection end to end for that to work. On the other hand, sending emails is asynchronous. The message is handed off to an intermediary who manages transport, routing, storage and delivery. Message queuing allows communication between applications by exchanging messages. It provides a place to store messages temporarily when the destination is busy or not connected. [1].

Message Queuing is part of a larger Message Oriented Middleware (MOM). Message Oriented Middleware (MOM) is a crucial part when it comes to the development of distributed applications. For an e-business to be successful it is important that applications that are based on different architectures seamlessly integrate with each other. MOM is used to help applications across multiple platforms communicate with one another, creating a much more seamless business operation [2].

Message queues facilitate asynchronous communication between applications. This means that applications can communicate with each other without having to be online attached to the queue at the same time. Once a message is pushed onto a queue, it stays there till the receiver connects and consumes it.

Queuing is the mechanism which keeps the messages stored onto a queue till a receiver connects to the queue and pulls it. Queuing allows you to:

- Communicate between programs without having to write any logic that connects the programs. These programs may or may not be built in the same environment.
- The order of messages to be processed in can be selected.
- When the number of messages exceed a threshold limit load balancing can be done.
- Create master slave architecture of senders and receivers to increase the availability of your applications [3].

1.1 Objective

When multiple software applications are connected via a network, there comes a time when there is a need for the clients and servers to communicate with each other. Moreover, the clients and servers may always not be available. This is where message queues come into picture. But every application has different needs and it is important to evaluate different message queuing technologies to find one that best fits your needs. The objective of this thesis is to study the various message queuing applications available and evaluate them based on their performance.

1.2 Outline

In Chapter 2, we will present some basic background of Message Queues and all the information required to better understand them, why are they needed and why will they be relevant in the future as well. Chapter 3 discusses Open Source Message Queues in depth as the focus of this study is to better understand open source message queues. Chapter 4 explains the setup and underlying environment used to test the various applications of message queues. In Chapter 5, the results of the test are mapped and explained. Chapter 6 draws the conclusion based on the results.

CHAPTER 2.

BACKGROUND

For a distributed system to work, it needs to pass information between the machines. No single machine is responsible for the entire system, but all information is interrelated. Hence a major concern of distributed systems is this transfer of data. Which also proves to be one of the most significant challenges.

2.1 What are Message Queues

In simple words, message queues are queues of messages that facilitate communication between applications. A **queue** is a line of things waiting to be handled in a sequential manner. A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed [4]. A **message** is the actual data that is transported between applications. For example, it can be a start process command for a task.

Figure 1 shows the basic architecture of a message queue. The producer, which is the client application, creates messages and pushes them onto a queue. The consumer on the other end of the queue receives the message that is to be processed. Messages stay on the queue till the consumer receives them. It acts a s a buffer between the applications, to queue messages coming from the source application until the destination application is ready to receive them.

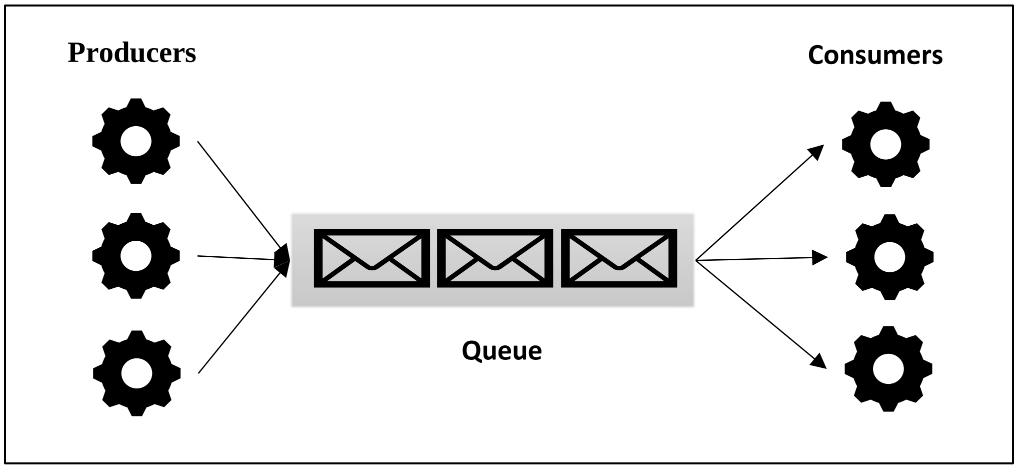

Figure 1. Message Queue Architecture.

2.2Before Message Queues

Message Queues work on peer to peer communication model. This allows programs to operate independently and use message queues to exchange information. The communication can be either synchronous or asynchronous. Before the use of message queues, communication between programs happened mainly using the client server communication model.

2.2.1 Client Server Communication Model

The client server model was developed in the 1980s. It was one way of achieving distributed systems. Following this model, two programs are used in creating a distributed system; one programs is assigned to generate requests and the other programs fulfills those requests [5]. Client programs provide an interface to the user to request for information or services. The Server processes the request and sends back information. This provides synchronous communication. Client server model has technologies like Remote Procedural calls that is used for building distributed architecture.

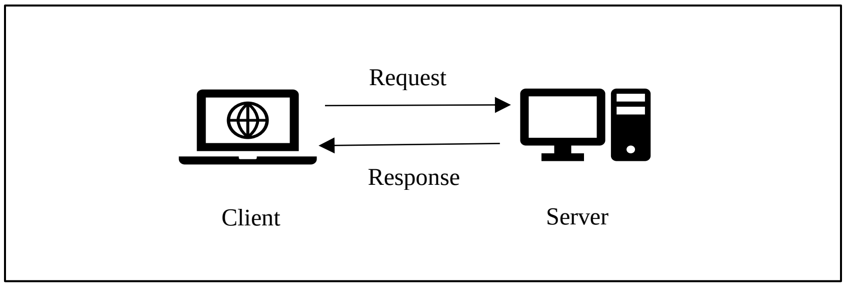

Figure 2. Client Server Communication Model.

Remote Procedural Call

A remote procedural call is the simplest way to exchange information between two applications. The way functions in a program are called is modeled by RPC. A packet of information is passed to the recipient as parameters. It then waits for a response from a recipient. It is not concerned with what the recipient does. The recipient then returns a result to the sender as a packet of information. Even though this model works well for normal programs, it has some drawbacks when it comes to distributed systems [6].

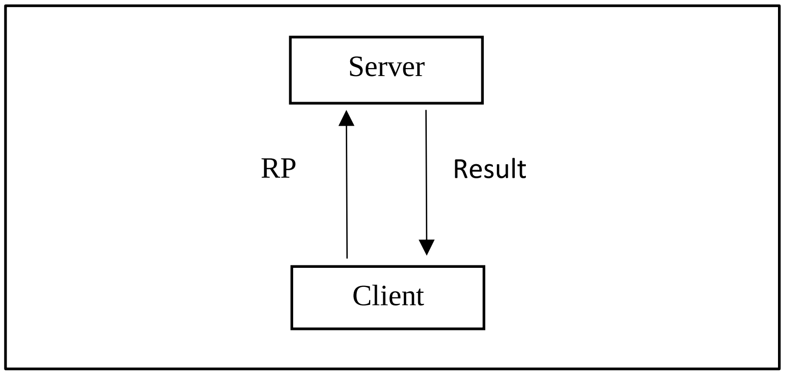

Figure 3. Remote Procedural Call.

One major drawback being that RPCs are synchronous. Both the participating applications need to assign dedicated resources that wait listening for a response from either end. The request can be a query for which the resource needs a result, or it can be a command asking the resource to act on it. The call can even be a command that returns a void and the resource must listen dedicatedly for it. If it does not receive a response, it does not know if the recipient receives the call or not. So, it must either fail the request or retry the RPC.

RPCs cannot be relied on. As opposed to a local method call that cannot fail to reach the receiver, an RPC can get dropped, timed out or corrupted. This can happen to both the request and response. If there is a failure, there is no way to find out if it’s a request or response that is lost. A request can be re-sent but a response can lead to duplicate data.

2.3 What is the need for Message Queues?

In today’s application development, decoupling and scalability are of utmost importance. Instead of building one large application, it is beneficial to decouple various parts of it and establish an asynchronous communication between them. This helps each part of the application to evolve independently, be developed in different environments that are more efficient for the application module and be self-contained in its functionality.

Decoupling is a sign of an application that is well structured which makes it easier to maintain, extend and debug [7]. Decoupling of systems can be achieved when the systems can be completely autonomous and unaware of each other but are still able to communicate without being connected. When the system is decoupled, communication needs to be asynchronous.

Message queues provide asynchronous communication. Once the producer sends a message to the queue, the consumer consumes the message when it starts. The message remains in the buffer till the consumer is ready. Once the producer sends a message onto the queue, it need not follow up on the status of the message and can continue with its next process. If the message does not go through, it can be redelivered until it is processed.

2.4 Advantages of Message Queues

- **Asynchronous Messaging**

When an application needs a task to be done but its not needed immediately or if the result doesn’t matter, Queues are very useful. Instead of waiting for that task to complete, a message can be added to the queue to perform the task later.

- **Decoupling by Data Contracts**

Hard dependencies can be decoupled by using a queue between different parts of the applications. The message in the queue becomes a data contract and any application that understands it, can process the transaction.

- **Concurrency**

If multiple producers are sending messages at a time, there could be a problem with concurrency to ensure that the first message is consumed first. By using a queue, it guarantees that the first message is the first to be consumed.

- **Scalability**

Message queues facilitate decoupling of applications. This helps in improving the scalability of the application.

- **Monitoring**

Message queuing systems enable you to monitor queues. This helps to know how many items are in the queue, the rate at which messages are processed and other statistics.

- **Break large tasks into small ones**

A large task can be divided into much smaller tasks and pushed onto the queue to occur in a sequence on the other system.

- **Persistence**

One of the most important things in asynchronous communication is to make sure that messages are received by the receiver. Queues make sure that the transaction has gone through and it is not safe to discard a message.

- **Guarantee that transaction occurs once**

As a message queue waits to confirm that a transaction has been processed before deleting the message, this helps to ensure that the transaction has happened only once.

2.5 Types of Message Queue Implementations

Message queuing can be implemented as a Service, Hardware or Open Source.

- **Hardware:** Vendors like Solace, Apigee and Tervela provide hardware-based messaging middleware. Queuing is offered through silicon data paths [8].
- **Service (SaaS):** Cloud based message services include IronMQ, StormMQ and Amazon Simple Queue Service (SQS).

- **Open Source:** Most widely used open source message queues are Apache ActiveMQ, Apache Qpid, Apache RocketMQ, RabbitMQ. Apache Kafka is primarily a streaming tool that also can be implemented as a message queue.

2.6 Message Queuing Protocols

Message Queuing protocols started being implemented when Open Source Message Queues came into existence. The stages of standardization and adoption for these protocols is different. The first two operate at the same level as HTTP, MQTT at the level of TCP/IP [9]. The three main protocols are:

- **Advanced Message Queuing Protocol (AMQP)** – This message queuing protocol is rich in features. It has been approved since April 2014 as ISO/IEC 19464.
- **Streaming Text Oriented Messaging Protocol (STOMP)** – It is text oriented and simple.
- **MQTT (formerly MQ Telemetry Transport)** – Used specially for embedded devices and is lightweight.

2.7 Message Queues Model

Message Queues can be modeled in two basic ways. They can have a Broker, or they can be Broker less.

2.7.1 Brokered Message Queues

Most messaging systems have a broker i.e., a messaging server in the middle. This is like a hub or star architecture. No two applications connect directly to each other. They all connect through the broker [10].

This model has various advantages. Applications do not need to now the actual location of the other applications. Just knowing the queue or topic name and network IP is enough. The sender and receiver lifetimes do not need to overlap. The application that sends message to broker can push the message and terminate. The pushed message is available to be received anytime. As brokers store data on the disk, messages are never lost and can be available even after a failure.

The two main drawback of brokered system are that it requires a lot of network bandwidth for communication and there can be a traffic bottleneck at the broker. The broker will be overworked as all applications connect to it but the application themselves maybe idle.

Types of Brokered Message Queues

- **ActiveMQ and RabbitMQ**

They are both based on AMQP. They make sure that the message is delivered as they are brokers. Persistent and non-persistent delivery and synchronous and asynchronous messaging is supported by both the brokers. A server restart does not result in loss of data as by default messages are written to the disk. The latency is high when using synchronous messaging. To add to it, latency increases as the brokers use message acknowledgement to guarantee message delivery. Clustering is supported through shared storage and shared nothing for fault tolerance. To ensure that there is no message loss or failure, queues can be replicated across clustered nodes [11].

- **NATS and Ruby NATS**

NATS is a pure Go implementation of the ruby-nats messaging system. It is fast and simple to use. Message transactions and persistence are not done by NATS. However, it does support clustering so you can build the system keeping in mine high availability and failover. TLS and SSL are supported in ruby nats but not NATS.

- **Kafka**

Kafka has been developed by LinkedIn. It makes use of a distributed commit log which is persistent to implement publish-subscribe messaging. It is specially designed to operate in clusters so that multiple clients can access it. It makes use of ZooKeeper which helps the brokers to integrate seamlessly and it internally take care of cluster rebalancing. Messages can be easily replayed if there is a failure at the customer end. Kafka clusters can be easily maintained using ZooKeeper but that also means that we have an additional module to maintain [12].

- **Kestrel**

Kestrel is developed by Twitter. It is a distributed, open source message queue. As it intends to be lightweight and fast there is no failover or clustering implemented. There is no cross-communication between nodes. Its queues are durable. There is item expiration and reads for every transaction.

- **NSQ**

NSQ is a messaging platform built by Bitly. The daemon is NSQD and is standalone. It is responsible for receiving, queueing and delivering messages to the client. The topology on which NSQ runs is decentralized and distributed. This is achieved by another daemon called nsqlookupd. This acts like a service discovery mechanism. It also provides nsqadmin. It acts as a front end to display real-time cluster statistics. It also executes tasks like managing topics and clearing queues. Messages are non-durable by default. It is an in-memory message queue. The size of the queue can be configured. This means that after a certain point, messages will be written to the disk. There is no built-in message replication. In order to guarantee delivery of messages, it makes use of acknowledgements. But this does not guarantee the order of messages delivered. Idempotency is the responsibility of the develop as messages can be delivered more than once. NSQ provides the functionality of adding clusters just like Kafka

- **Redis**

Transient storage and lightweight messaging are provided by Redis. Even though its publish and subscriber capability is fast, its capability is limited.

2.7.2 Broker-less Message Queues

In Broker less messaging there is no broker meaning the queues connect the two peers directly. There is no involvement of a middleman. As seen in Figure 4, the number of hops decreases as there are no brokers. There is no bottleneck on the network. It is ideal when there is more emphasis on low latency but high transaction rate. This leads the system to not be easily managed. Each application must know the network address of each application it wishes to communicate with. This model looks good on paper but is not easily managed in the real world.

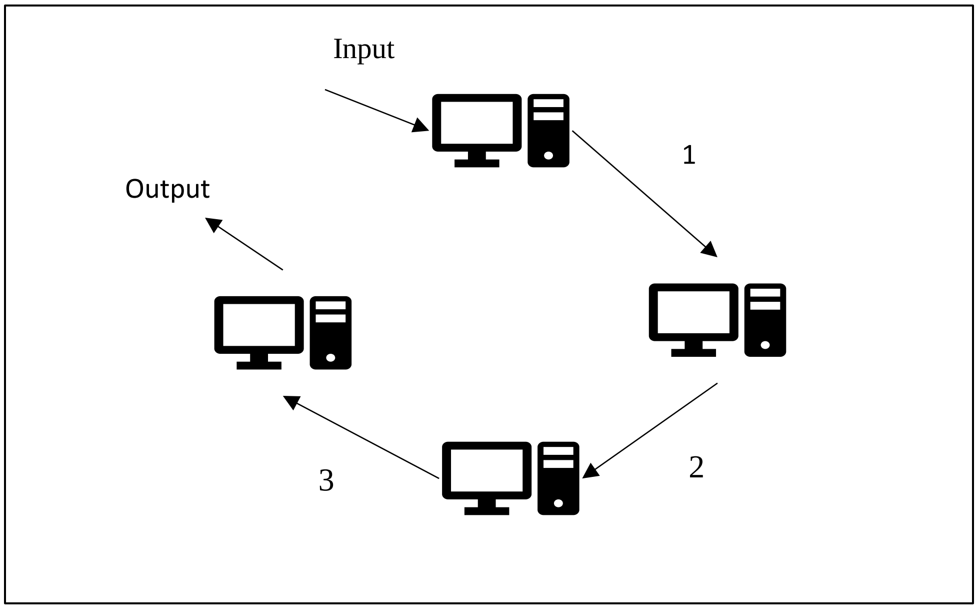

Figure 4. Broker-less system.

Types of Broker less Message Queues

- **ZeroMQ and Nanomsg**

Nanomsg is a socket style library that makes use of convenient patters to perform distributed messaging. This means that, apart from embedding the library, there is nothing to deploy. The working of Nanomsg is similar to that of ZeroMQ as it is written by one of its authors and provides a cleaner API. Unlike ZeroMQ, there is no notion of a context in which sockets are bound to. Nanomsg is more open to extension as it also provides pluggable transport and messaging protocols. It also has built in scalability protocols. Like ZeroMQ, it guarantees the delivery of messages in the given order but there is no guarantee of delivery itself.

ZeroMQ has been around since 2007 and is battle tested. Like nanomsg, ZeroMQ acts as a socket abstraction and is not a message-oriented middleware. It is almost similar to Nanomsg when it comes to usability.

CHAPTER 3.

OPEN SOURCE MESSAGE QUEUES

3.1 Apache ActiveMQ

Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (JMS) client. The latest version appeared in March 2019. Its basic function is to send messages between various applications and help them communicate with each other. It also includes additional features like OpenWire, STOMP and JMS. ActiveMQ is written in Java and it translates and passes messages between the sender and receiver.

3.1.1 ActiveMQ working

ActiveMQ sends messages between applications and has three main components.

- Producers: Client applications that create and send messages.
- Consumers: Application that receives and processes the message
- Destination: ActiveMQ broker routes messages between from producers to consumers through two types of destinations:
    - Queue: Used in point to point setup.
    - Topic: Used in Publisher Subscriber setup.

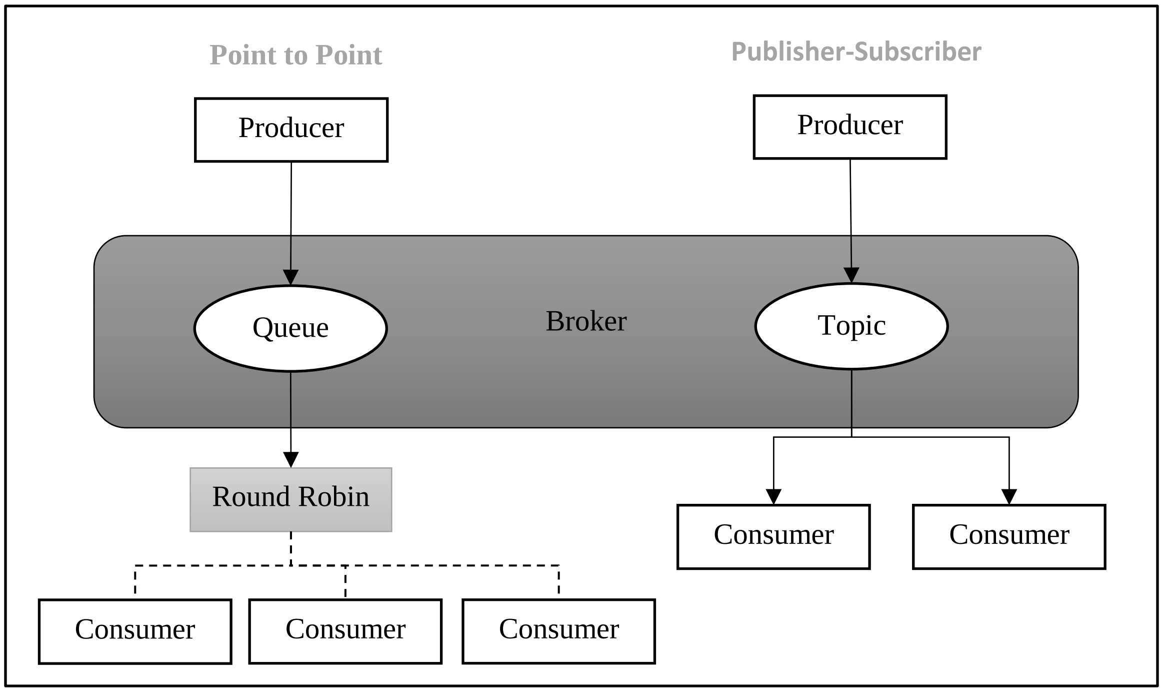

Figure 5. ActiveMQ Architecture.

Messages can be sent over either a queue or topic based on the ActiveMQ setup. In point to point messaging, one queue has one or more consumers attached to it. The broker acts as a load balancer and routes messages to consumers in a round robin fashion. In Publisher Subscriber setup, broker delivers each message to all consumers attached to that topic.

A simple ActiveMQ connection can be setup between two applications using the steps given in Figure 6. One Queue or Topic can have multiple producers and/or consumers attached to it at the same time.

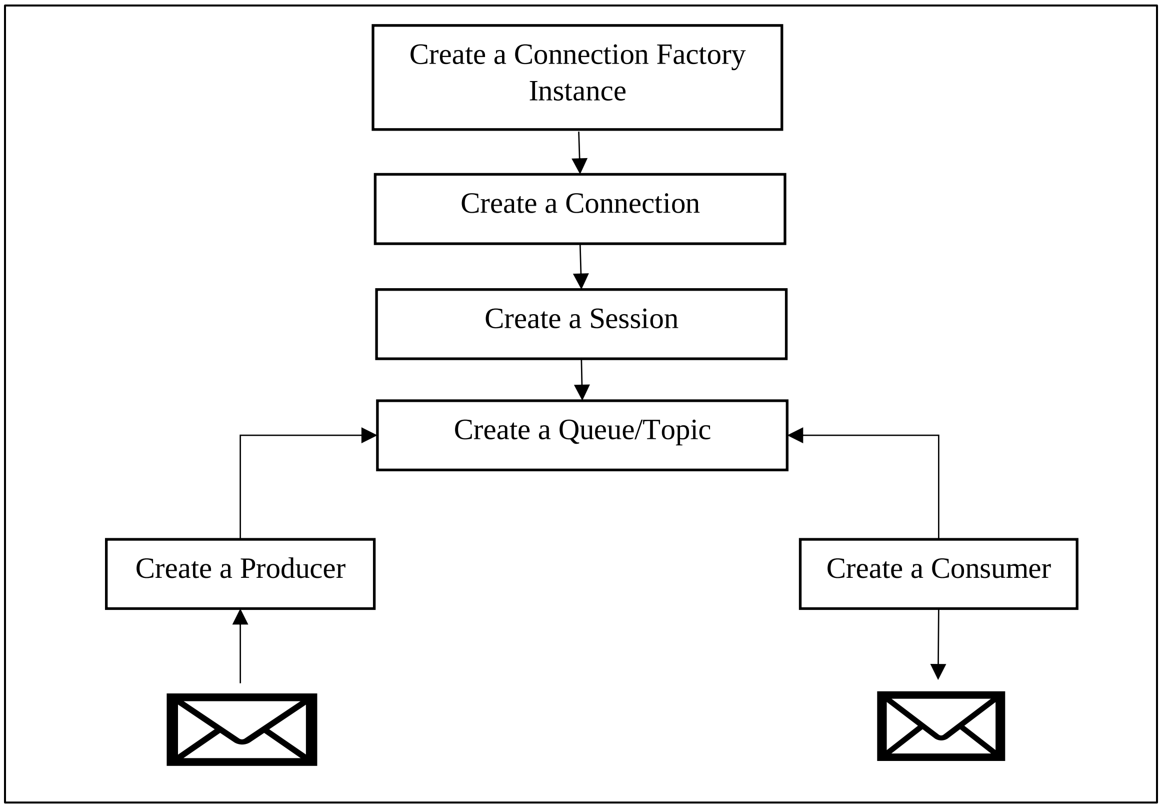

Figure 6. Steps to create basic ActiveMQ Application.

3.1.2 Why ActiveMQ

ActiveMQ is one of the most widely used brokers when creating a distributed system. Following are some of the reasons why ActiveMQ is popular.

- **Transactional Messaging**

ActiveMQ is persistent when it comes to messages. Irrespective of system failures, it processes each message exactly once and does not miss a single message.

Persistence is achieved through servers and availability is achieved through clustering of servers.

If a node is offline, the message remains in the queue the entire time till the node comes back online.

- **High Performance Market Data Distribution**

ActiveMQ is highly efficient in its routing and throughput. It provides a very high sender and receiver throughput.

- **Asynchronous Messaging Model**

ActiveMQ provides low latency for bigger messages. Its latency is challenged a little when the messes are small and larger in number.

- **Web Streaming Data**

ActiveMQ has AJAX support. This helps to integrate ActiveMQ with a web container. This helps in publishing messages using Http Post.

- **RESTful API**

The message broker can be provided with an Http interface. This allows simple cross language APIs to exchange messages. An Http Post request is made to post a message to the broker and an Http Get request is made to receive messages from the broker. The destination is specified using the URI and its associated parameters.

3.1.3 Who Uses ActiveMQ?

- **FuseSource:** FuseSource provides enterprise class training, mentoring ans support for ActiveMQ
- **Dopplr**: ActiveMQ is used as a message provider at Dopplr.

- **Gnip**: Low latency MOM messaging at Gnip is achieved using ActiveMQ.
- **RomTrac:** Load balancing of requests across a server cluster is achieved using ActiveMQ.
- **University of Washington:** Uses ActiveMQ as a backend layer of messaging for distributed applications.
- **Daugherty Systems:** used within in-house application for reliable, asynchronous messaging.
- **Document archiving/flow systems:** Use ActiveMQ for connectivity between front and back end. Once frontend receives a message it processes it and sends it forward to the backend. It then creates an ID and sends it to the user.
- **CSC**: the Finnish IT center for science is building a bioinformatics system for DNA-microarray data storage and analysis. System contains rich graphical clients, a large database and heavy server machinery for processing analysis jobs. We are trying to use ActiveMQ as a JMS implementation to shuttle data between the servers and clients in an event-based manner.
- **Gather Place:** use ActiveMQ to gather billing and usage data that is real time from servers that are distributed across the world.
- **Golconde:** A distributed postgresql replication system is implemented using ActiveMQ.

3.2 RabbitMQ

RabbitMQ is an open source message broker written in Erlang. It implements message queuing protocols such as AMQP, STOMP and MQTT. The latest version appeared in March 2019. Its basic function is to send messages between various applications and help them communicate with each other.

3.2.1 RabbitMQ Working

Like ActiveMQ, RabbitMQ acts as a middleman to exchange messages between various applications. It consists of three main components: Producer, Consumer and Queue. Unlike ActiveMQ, it does not directly push messages onto queues, instead, producers send message to an exchange. Messages from the producer application are accepted by an exchange. This message is then routed to the message queue. Binding and routing keys are used to achieve this. A link between an exchange and queue is the binding [13].

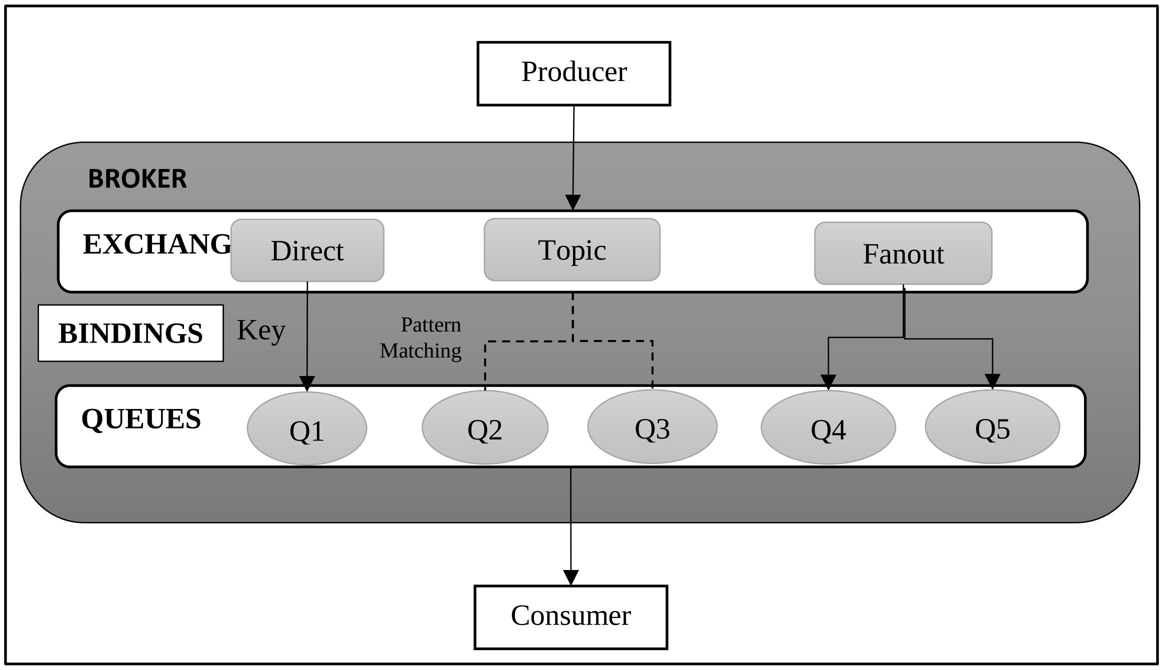

Figure 7. RabbitMQ Architecture.

Direct: Based on a message routing key, a direct exchange delivers messages to queues. The binging and routing key should match in order to perform a direct exchange.

Topic: For a topic exchange, the routing patter specified in the binding is patter matched against the routing key.

Fanout: A fanout exchange is like a publisher-subscriber setup. All the queues attached to it receive the message.

A simple RabbitMQ connection can be setup between two applications using the steps given in Figure 8. One Queue or Topic can have multiple producers and/or consumers attached to it at the same time.

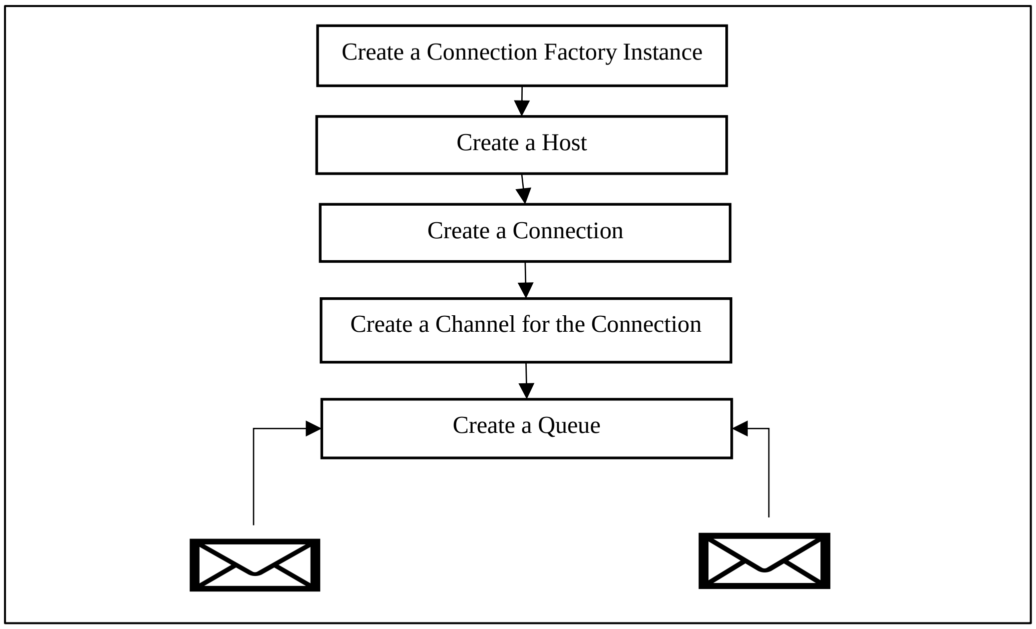
Create a Connection Factory Instance
Create a Host
Create a Connection
Create a Channel for the Connection
Create a Queue

Figure 8. Steps to create RabbitMQ Application.

3.2.2 Why RabbitMQ

- RabbitMQ helps to facilitate in connecting and scaling applications. Applications connect to form a larger application or connect to user devices. Messaging is asynchronous, decoupling applications by separating sending and receiving data.
- RabbitMQ is a messaging broker - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

Feature Highlights

- **Reliability**

RabbitMQ is highly reliable. It provides features like persistence of messages achieved using delivery acknowledgements and high availability.

- **Flexible Routing**

Messages first route through exchanges and then arrive at the queue. Various exchange types are built-in within RabbitMQ for simple routing. When it comes to complex routing, you can write your own exchange type and use it as a plugin.

- **Clustering**

A single logical broker can be created by clustering several RabbitMQ servers on a local network.

- **Federation**

A federation model is available is servers need to be loosely and unreliably connected.

- **Highly Available Queues**

To ensure the safety of messages during a hardware failure, queues can be mirrored across multiple machines in a cluster.

- **Multi-protocol**

Messages are supported over a number of messaging protocols.

- **Many Clients**

RabbitMQ clients are available across multiple clients.

- **Management UI**

The management UI available along with RabbitMQ allows the control and monitoring of every aspect of the message broker.

- **Tracing**

Tracing support is available within RabbitMQ.

- **Plugin System**

RabbitMQ comes with a lot of plugins, you can also customize plugins or create them from scratch.

3.2.3 Who uses RabbitMQ

Many companies make use of RabbitMQ such as:

- **The Deutsche Börse**

RabbitMQ is used as a standard protocol on their system. It helps in monitoring the positions and risk related data of its members.

- **JPMorgan**

JPMorgan sends billions of AMQP data per day.

- **National Science Foundation**

RabbitMQ is also used by The Ocean Observatories Initiative infrastructure in their sensors. AMQP to bring readings ashore from ocean platforms and a global pub-sub network to disseminate readings.

- **NASA**

It is used in the control plane of the Nebula Cloud Computing.

- **Red Hat**

RedHat uses RabbitMQ to control all its internal operations.

- **VMware**

RabbitMQ is used in cloud services and virtualization of their product.

- **Google**

Google has a project called Rocksteady that makes use of RabbitMQ to analyze user defined metrics. The goal of the service is to allow diagnosis of root causes.

- **UIDAI, Government of India**

UIDAI is the largest online identity project in the world aiming to provide each of India's 1.2 billion residents with a unique identity number. UIDAI uses RabbitMQ to decouple sub-components of its application allowing it to scale.

- **Mozilla**

RabbitMQ is used in their in-house eventing called Pulse.

- **OpenStack**

Openstack uses RabbitMQ for messaging.

- **AT&T**

The local search provides, AT&T interactive makes use of RabbitMQ.

- **INETCO**

It uses RabbitMQ to pass real time data in cloud environments.

- **Smith Electric Vehicles**

SmithLink service uses RabbitMQ to transfer 2 billion data points per day for automotive based in 90 countries across the world.

3.3 Apache Kafka

Apache Kafka is a distributed streaming platform. It has the following three capabilities:

- Publish and subscribe to streams of records, like a message queue or enterprise messaging system.
- Storage of streams of records in a fault-tolerant durable way.
- Processing of streams of records as they occur.

Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

3.3.1 Kafka Working

Kafka topics are divided into several partitions as shown in Figure 8. You can parallelize a topic using partitions. This lets you split the data across multiple brokers. Partitions can be placed on different machines. This allows the consumers to read topics in parallel. Parallelizing of consumers is also possible. This lets multiple consumers to read from multiple partitions. This gives a high throughput for message processing.

An offset is assigned to each message that is within a partition. This message ordering is maintained by Kafka. Consumers can read messages starting from a specific offset and can read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit [14].

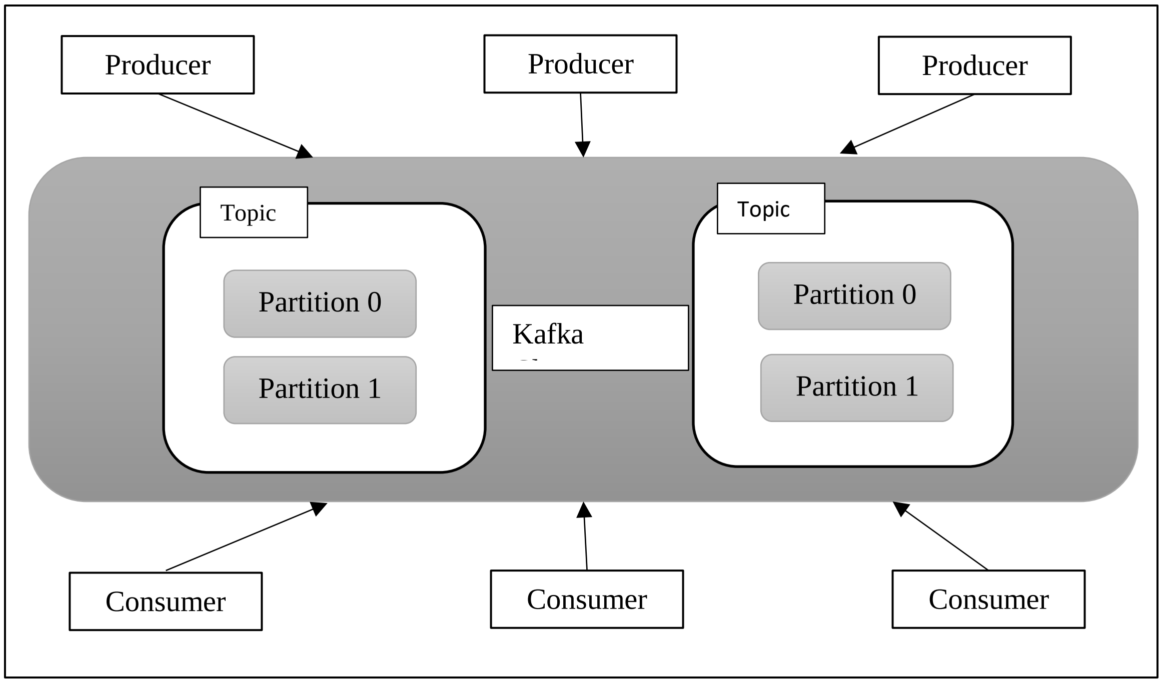

Figure 9. Kafka Architecture.

3.3.2 Why Kafka

To provide real-time analytics, Kafka is often used in real time streaming of data architectures. There are cases when ActiveMQ and RabbitMQ are not considered for message queueing, but Kafka is. This is because Kafka is comparatively reliable and has a higher throughput. Kafka is used for metrics collection and monitoring, log aggregation, real-time analytics, stream processing, website activity tracking, CEP, ingesting data into Spark, ingesting data into Hadoop, CQRS, replay messages, error recovery, and guaranteed distributed commit log for in-memory computing. [15]

3.3.3 Who Uses Kafka

Large companies that handle tons of data use Kafka. It was developed at LinkedIn where it was used to track data and operational metrics. In order to provide stream processing infrastructure, it is used by Twitter as a part of Storm. Square uses Kafka as a bus to move all system events to various Square data centers (logs, custom events, metrics, and so on), outputs to Splunk, for Graphite (dashboards), and to implement Esper-like/CEP alerting systems. It's also used by other companies like Goldman Sachs, Spotify, Cisco, Uber, PayPal, Tumbler, Netflix, Box and CloudFlare.

3.4 Comparison of Open Source Message Queues

	Apache ActiveMQ	**RabbitMQ**	**Apache Kafka**
Main Concept	One of the most often used open source products for messaging. ActiveMQ is commonly used in enterprise projects, due to its support of advanced features such as multiple instances for storing messages, and clustering environments.	RabbitMQ is a messaging broker, implementing low-level AMQP protocol and Producer-Consumer pattern. It is intermediary between two applications when in the procedure of processing communication.	Apache Kafka is a community distributed event streaming platform capable of handling trillions of events a day. Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit

	Apache ActiveMQ	RabbitMQ	Apache Kafka
	The basis of ActiveMQ is JMS – the Java Messaging Service. JMS is an API implementation within J2EE (Java Enterprise).		log. Kafka has quickly evolved from messaging queue to a full-fledged event streaming platform, since its creation by LinkedIn in 2011. [16]
Language	Java	Erlang	Scala
Cross Platform	Yes	Yes	Yes
Opensource	Yes	Yes	Yes
Multiple Languages	Yes, it supports Pearl, Ruby on Rails, C++, C, C#, Jekejeke Prolog, Haskell, Go, Erlang, Python, Pike, Racket, Netlogo, Haxe, Node.js	Yes, it supports C, C++, PHP, Python, Ruby, Erlang, Objective-C, Perl, Haskell, Go, Java, Javascript, Rust	Yes, it supports Go, Haskell, OCaml, Python, PHP, C#, Node.js, Ruby
Protocols	XMPP, WS, WebSocket, WSIF, OpenWire, MQTT, AMQP, REST, RSS,	HTTP, MQTT, AMQP, STOMP	Does not support message protocols

	Apache ActiveMQ	**RabbitMQ**	**Apache Kafka**
	Stomp, Atom, AUTO, WS Notification		
Brokers	Can be deployed with P2P topologies as well as brokers.	Only Broker	Only Broker
Tools	ActiveMQ Admin	The RabbitMQ admin which is a browser based UI management plugin and command line tool	
Synchronous/ Asynchronous	By default it is synchronous. It can be made asynchronous by setting the useAsyncSend property	Supports both asynchronous and synchronous methods	Inherently asynchronous
Basic Message Patterns	Publisher – Subscriber and Queues	Message Queue, PUB-SUB, ROUTING, RPC like REQ-REP, but not the same.	Publisher - Subscriber
Who uses it	FuseSource, Dopplr, gnip, RomTrac, University of	Reddit, Vine, CircleCI, Trivago, 9GAG, Code	Linkedin, Spotify, Uber, Tumbler, Goldman Sachs,

	Apache ActiveMQ	**RabbitMQ**	**Apache Kafka**
	Washington, CSC, STG Technologies.	School, 500px, HeadHunter	PayPal, Box, Cisco, CloudFlare, and Netflix

Table 1. Comparison between ActiveMQ, RabbitMQ and Kafka.

CHAPTER 4.

PERFORMANCE TEST ENVIRONMENT

This chapter explains the various environments that are setup for measuring the performance of the message queuing brokers. The hardware used in the setup along with how the software environment was setup is explained. The various performance measurement parameters used to evaluate the performance have been explained. The performance of the brokers is evaluated across different model architectures that can be setup for the brokers. The brokers are evaluated based on all the above-mentioned factors.

4.1 Hardware Environment

The underlying environment is common for all the three message Queues:

- Operating System - Ubuntu (64 bit)
- RAM – 4 GB
- Hard Drive – 32 GB

4.2 Software Environment

The setup of the brokers and all the supporting tools used for the benchmarking is explained in this section.

4.2.1 ActiveMQ

- Version: 5.15.9
- Setup: For the setup of ActiveMQ, the broker as well as the Producers and Consumers were setup on the same machine. The model used for passing messages is Publisher-Subscriber. In a Publisher-Subscriber model, the producers are decoupled from the consumer. This means

that the producer does not know how many consumers are interested in the message that it has published [17].

4.2.2 RabbitMQ

- Version: 3.7.14
- Setup: The broker, producers and consumers for RabbitMQ were all setup on a single machine. The publisher-subscriber model is used. This a pattern in which an application publishes messages which are consumed by several subscribers [18]. RabbitMQ is developed using Erlang and is required to run the tool.
- Erlang Version: 21.3

4.2.3 Kafka

- Version: 2.2.0
- Setup: For Kafka, the broker, producers and consumers are setup on a single machine. Kafka supports only the publisher-subscriber model. It also requires Zookeeper.
- Zookeeper Version: 3.4.13

4.3 Performance Measurement Parameters

To measure the performance of a broker, four aspects where considered:

A. Throughput

Throughput is measured as the total number of messages produced or consumed per second. Here the message size and number of messages passed is constant across the three brokers as 50B and 1000000 messages respectively. There is no delay between the production of two messages.

B. Latency by Number of Messages

Latency is the mean latency in milli-seconds for a message of 50B and is calculated over the number of messages sent.

C. Latency by Message Size

Latency by message size is the mean latency in milli seconds for a total of 1GB data distributed according to message sizes of 256 Bytes, 1 Kilo Byte, 5 Kilo Byte and 1 MB. Unlike the previous latency, the number of messages is not constant across all message sizes. Instead, it is 3906250, 1000000, 200000, 1000 messages respectively.

D. Total time taken for the entire transfer

Total time taken for the entire transfer of data is the time measured in seconds for the entire transaction of 1 GB data.

4.4 Latency Calculation

HDR Histogram is used for the calculation of Latency. Coordinated omission is attempted to be corrected by filling in additional samples when a request falls outside of its expected interval.

For example, latency is normally calculated as follows:

1. Timestamp before request is noted, t0.
2. Synchronous request is made.
3. Take a note of the timestamp after request, t1.
4. Calculate Latency t1-t0.
5. Repeat as needed for request schedule.

This approach works as expected if our requests fit within the specified request schedule. Suppose we are issuing 100 requests per second and each request takes 10 ms to complete, it works. However, if one request takes 100 ms to complete, this means only one request was issued during those 100 ms when, according to our schedule, we should have issued 10 requests in that window. Nine other requests should have been issued, but the benchmark effectively coordinated with the system under test by backing off. Those nine requests waited in line—one for 100 ms, one for 90 ms, one for 80 ms, etc. Most benchmarks don't capture this time spent waiting in line, yet it can have a dramatic effect on the results. HDR Histogram handles this issue.

4.5 Performance evaluation against various models

Along with the Performance Measurement Parameters mentioned in Section 4.4, the performance of the brokers was also evaluated against various types of models that the brokers support namely:

- **Single Producer and Consumer**

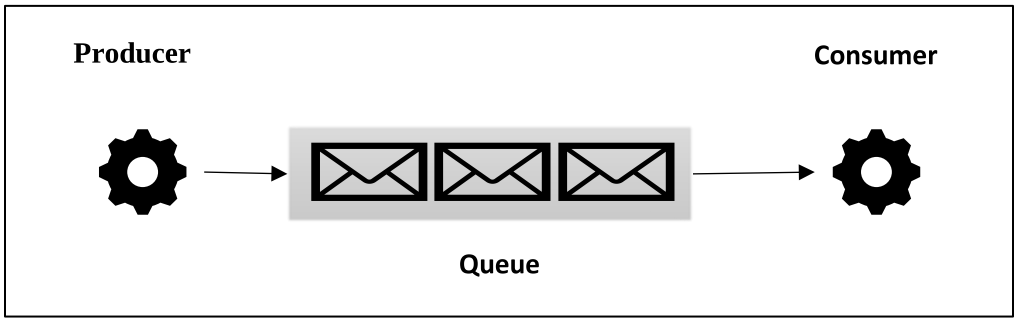

Figure 10. Single Producer-Consumer Setup.

Here, as shown in Figure 10 a queue/topic has a single pair of producer and consumer attached to it. This model can be used when there is not much data to be passed between two applications.

- **Single Producer – Multiple Consumers**

Here as shown in Figure 11, a queue/topic has two or more consumers listening to it. Once a message is pushed to the queue, both the consumers pull the same message. This model is useful when the same message needs to go to two separate applications.

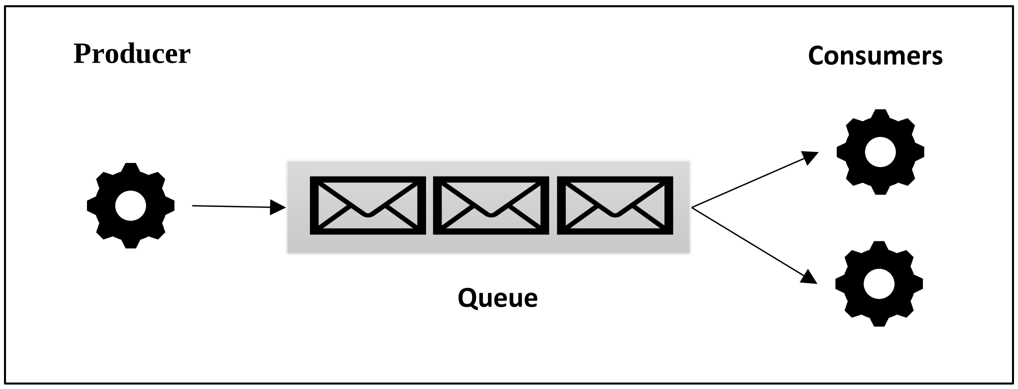

Figure 11. Single Producer-Multiple Consumer Setup.

- **Multiple Producers – Single Consumer**

Here as shown in Figure 12, two or more producers push messages onto the queue and there is one consumer that pulls all the data. This model is used when the listening application works with multiple other applications and needs data from them.

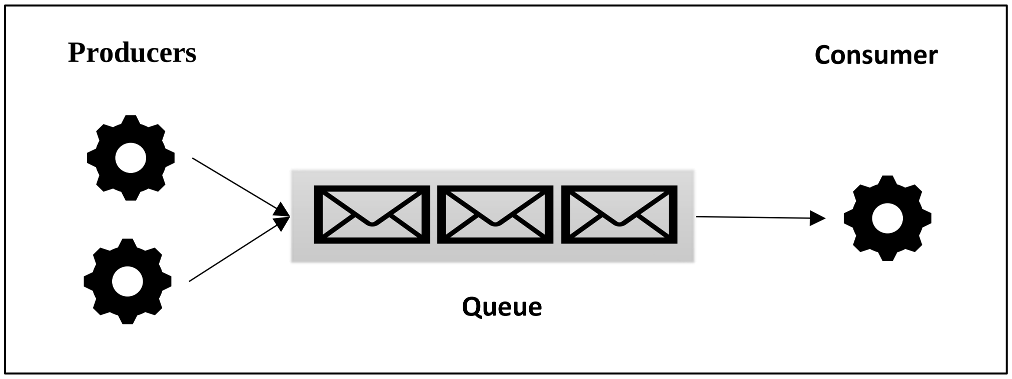

Figure 12. Multiple Producers-Single Consumer Setup.

- **Multiple Producer and Consumer**

Here as shown in Figure 13, there are multiple producers pushing messages to the queue and multiple consumers listening and consuming all the incoming messages. This is needed when there are multiple applications working with each other. There is no limit to the number of producers and consumers that can be attached to a queue.

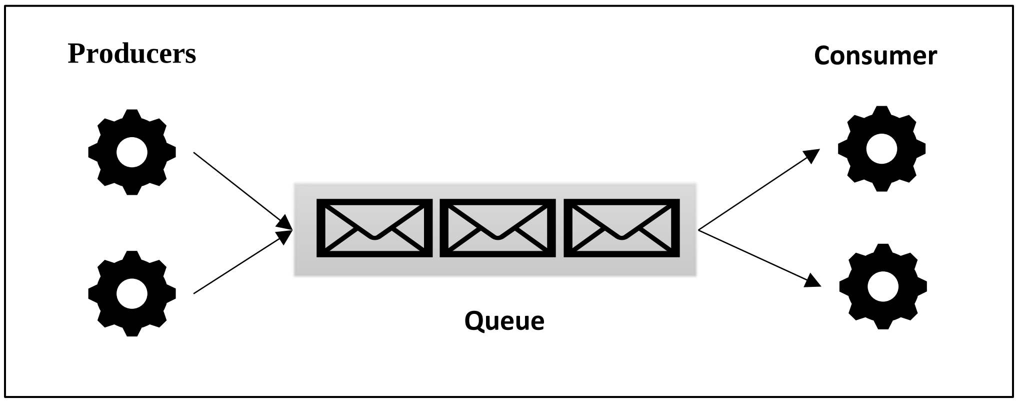

Figure 13. Multiple Producer-Consumer Setup.

CHAPTER 5.

PERFORMANCE ANALYSYS

5.1 Single Producer and Consumer

In Single Producer-Consumer model, one producer pushes messages to the queue and at the other end there is only one consumer listening at a time. There can be multiple consumers at the other end, but they will be configured as master-slave i.e. the secondary consumer only listens to the queue when its master shuts down thus becoming the master itself.

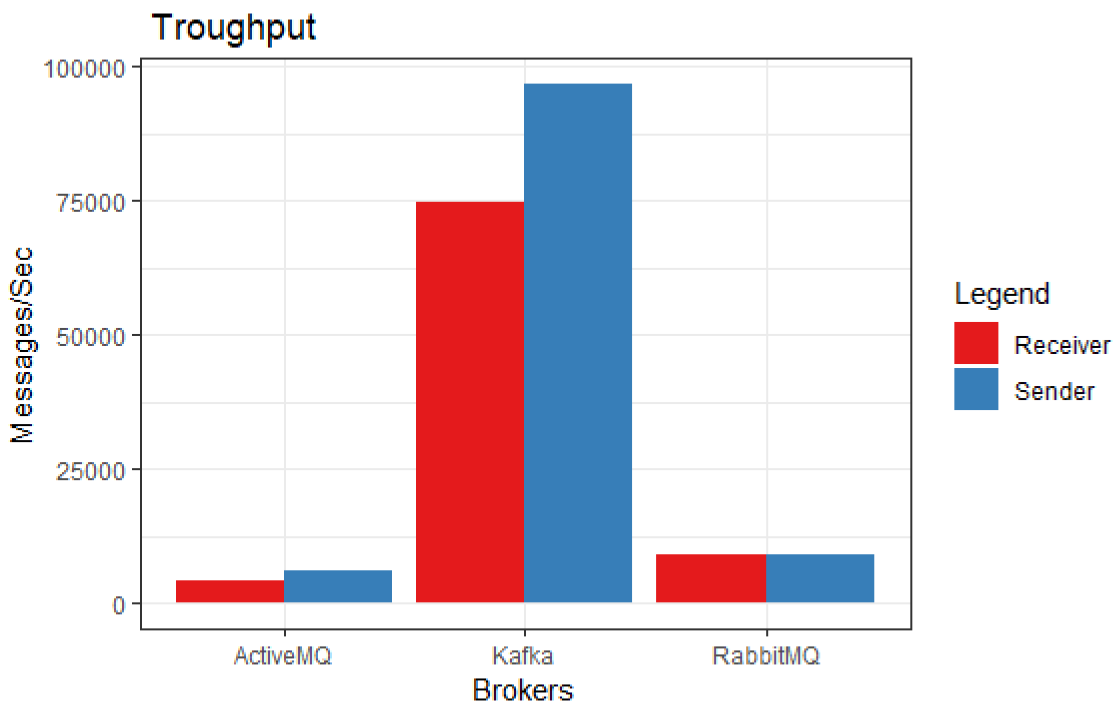

Figure 14. Single Producer-Consumer Throughput.

Figure 14 shows the throughput for producers and consumers. For measuring the throughput 1000000 messages each of 50 Bytes are sent in succession without any delay. As the graph shows, for ActiveMQ the throughput for sending and receiving messages is lower than the other two message queues. Kafka sends the highest number of messages sent per second, but the consumption of messages is not as fast as pushing messages to the topic. Even though RabbitMQ does not publish messages as fast as Kafka, the sending and receiving throughput is almost the same.

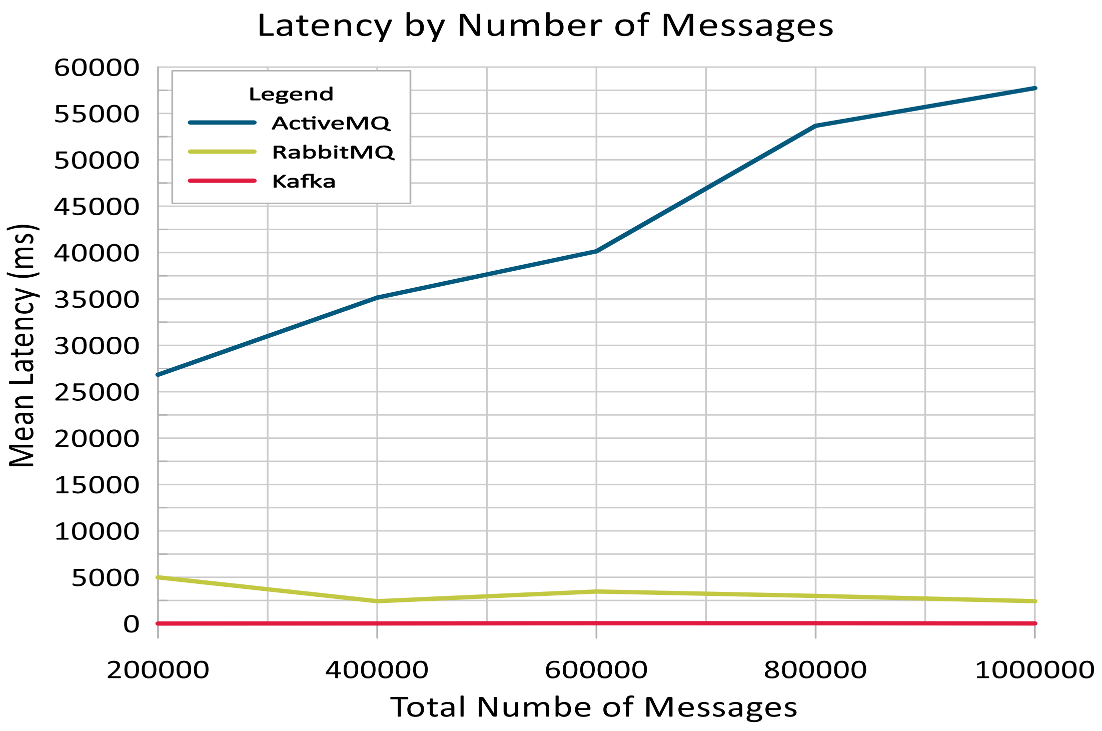

Figure 15. Latency by Number of Messages 1:1.

Figure 15 measures the latency of the message queues based on the number of messages it sends over the queue/topic. Here, each message sent is of 50 Bytes. As shown in the chart, the mean latency for ActiveMQ goes on increasing as the number of messages increases. The Latency for Kafka is somewhat constant and the latency for RabbitMQ fluctuates initially but becomes constant with the increase in the number of messages.

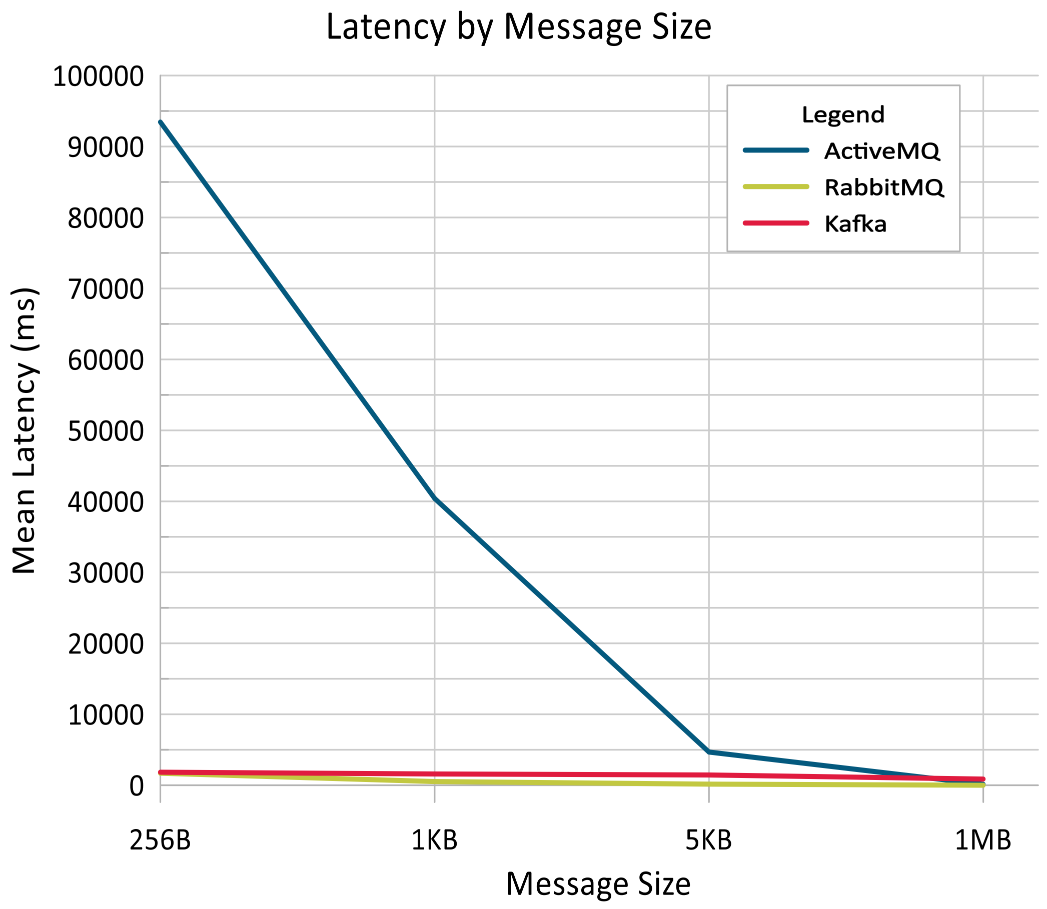

Figure 16. Latency by Message Size 1:1.

Figure 16 Measures the latency based on the size of the message. Here, messages of various sizes namely, 256B, 1KB, 5KB and 1MB are taken into consideration. The total size of data sent over is 1 GB. Hence, the number of messages sent for each message size varies i.e. 3906250, 1000000, 200000, 1000 respectively. The Mean Latency for ActiveMQ decreases with an increase in the message size. This is because of the increased number of overheads associated with smaller size data and its respective number of messages. The latency for Kafka and RabbitMQ remains constant irrespective of the message size.

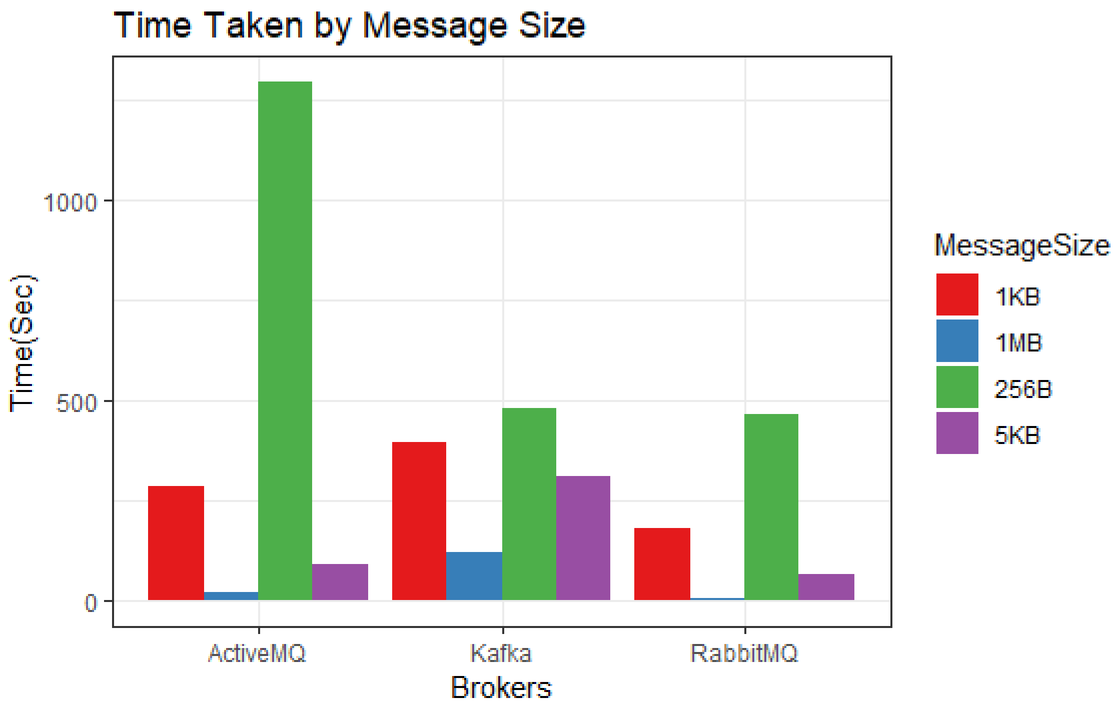

Figure 17. Time Taken by Message Size 1:1.

Figure 17 shows the time taken for an entire exchange of messages to happen based on the message size and number of messages. In this, the total data transferred between the producer and consumer is 1GB hence the number of messages sent varies depending on the size of the message. As shown in the chart, the time taken is maximum for ActiveMQ when the size of the message is the smallest i.e. 256 Bytes. As the size of the messages increase the difference between the time is not much for all three brokers.

5.2 One Producer – Two Consumers

In One Producer - Two Consumers model, one producer pushes messages to the queue and at the other end there are two consumers listening at a time. This means that any message pushed to the queue, will be consumed by both the consumers.

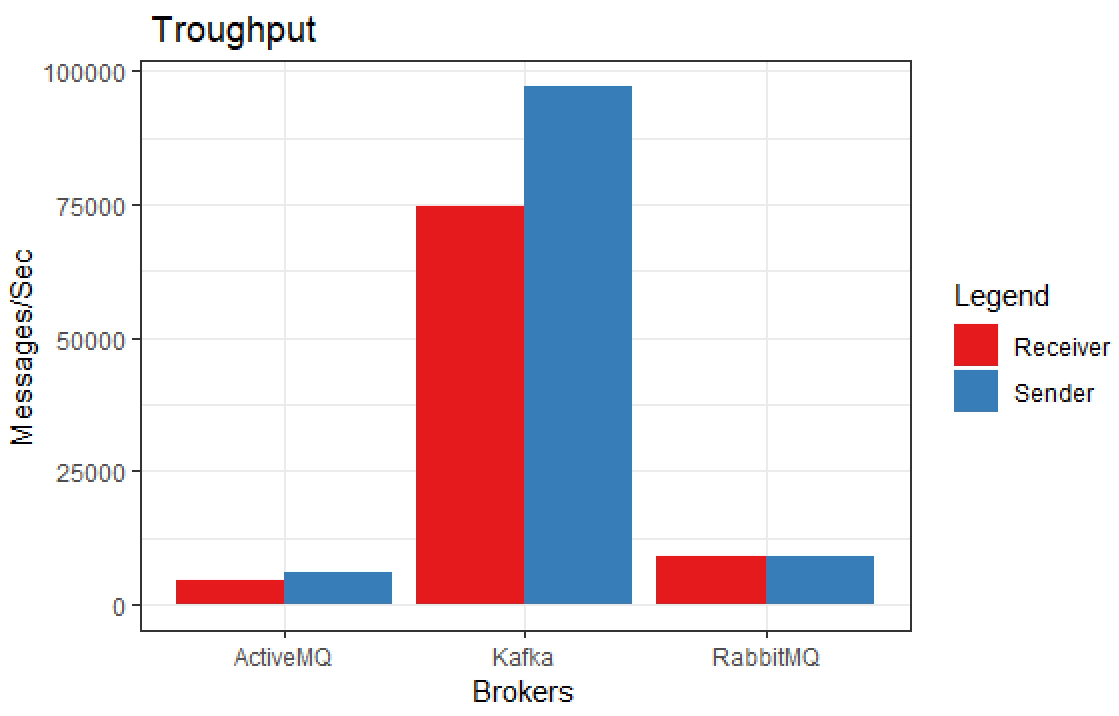

Figure 18. One Producer- Two Consumer Throughput.

Figure 18 shows the throughput for single producer and two consumers. For measuring the throughput 1000000 messages each of 50 Bytes are sent in succession without any delay. The throughput at consumer end is the average throughput of the two consumers. As the graph shows, for ActiveMQ the throughput for sending and receiving messages is lower than the other two message queues. Kafka sends the highest number of messages sent per second, but the consumption of messages is not as fast as pushing messages to the topic. Even though RabbitMQ does not publish messages as fast as Kafka, the sending and receiving throughput is almost the same.

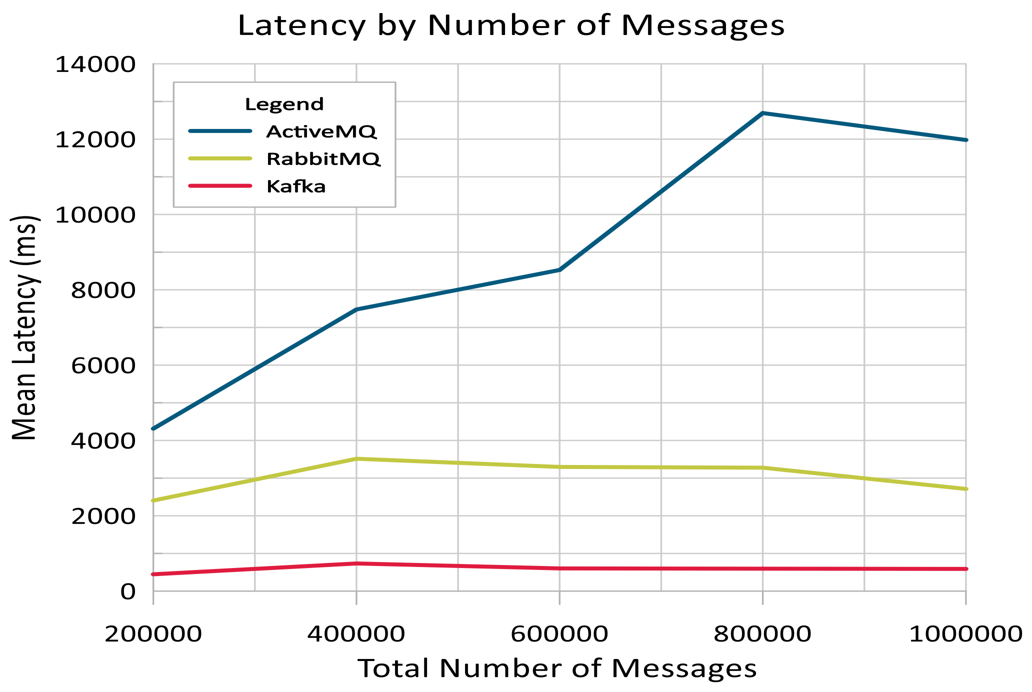

Figure 19. Mean Latency by Number of Messages 1:2.

Figure 19 measures the latency of the message queues based on the number of messages it sends over the queue/topic. Here, each message sent is of 50 Bytes. The mean latency calculated in the average latency between the two consumers. The Mean Latency for ActiveMQ increases up to a certain number of messages and then starts decreasing. The latency for Kafka is constant whereas RabbitMQ like ActiveMQ shows a decrease after an initial increase.

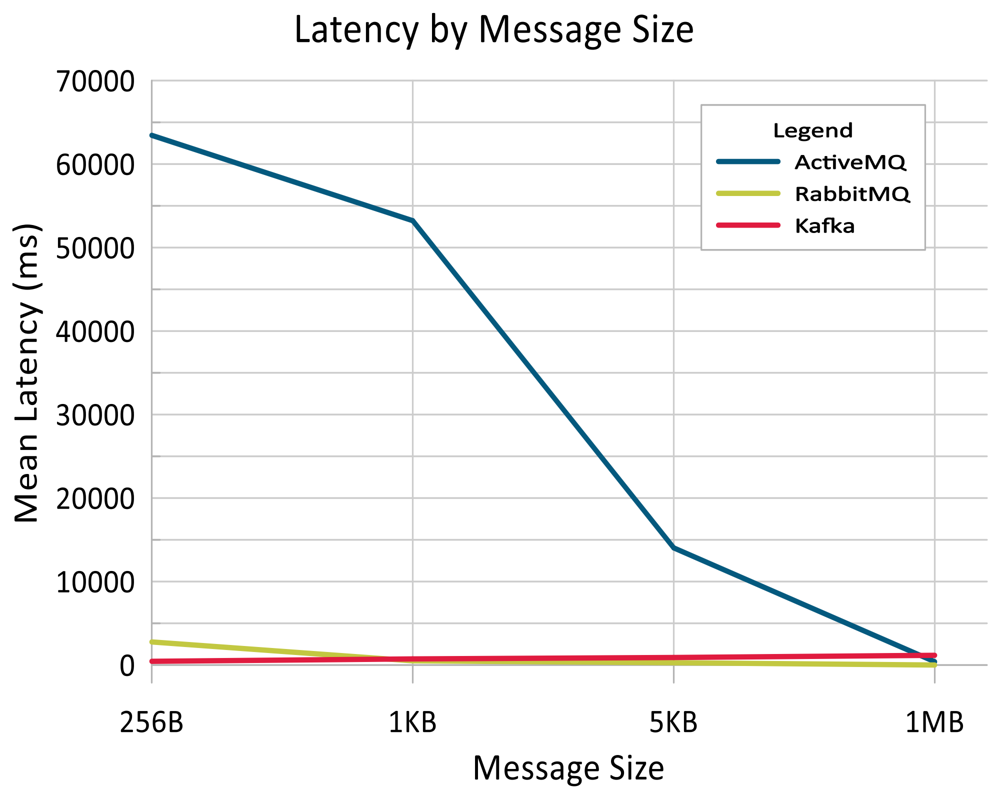

Figure 20. Mean Latency by Message Size 1:2.

Figure 20 Measures the latency based on the size of the message. Here, messages of various sizes namely, 256B, 1KB, 5KB and 1MB are taken into consideration. The total size of data sent over is 1 GB hence the number of messages sent for each message size varies. The mean latency calculated in the average latency between the two consumers. The Mean Latency for ActiveMQ decreases with an increase in the message size. The latency for Kafka and RabbitMQ remains constant irrespective of the message size.

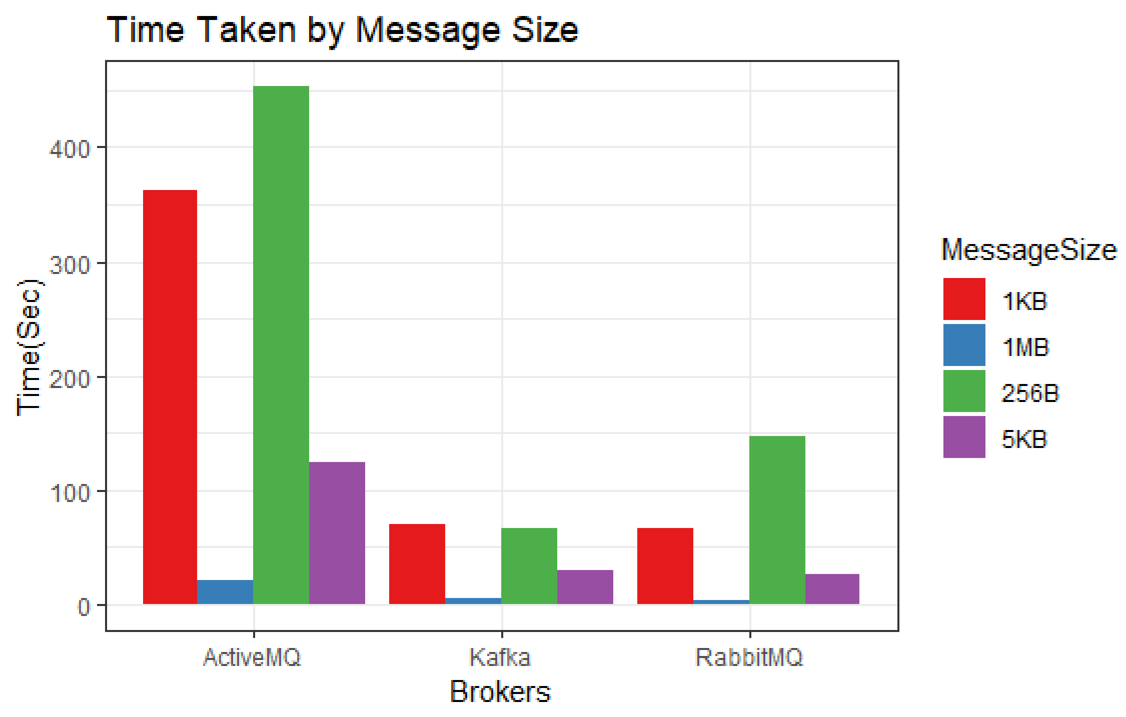

Figure 21. Time taken by Message Size 1:2.

Figure 21 shows the time taken for an entire exchange of messages to happen based on the message size and number of messages. In this, the total data transferred between the producer and consumer is 1GB hence the number of messages sent varies depending on the size of the message. As shown in the chart, the time taken is maximum for ActiveMQ when the size of the message is the smallest i.e. 256 Bytes. As the size of the messages increase the difference between the time is not much for all three brokers.

5.3 Two Producers – One Consumer

In Two Producer - One Consumers model, two producers push messages to the same queue and at the other end there is only one consumer listening. In this case, both the producers push the same message onto the queue.

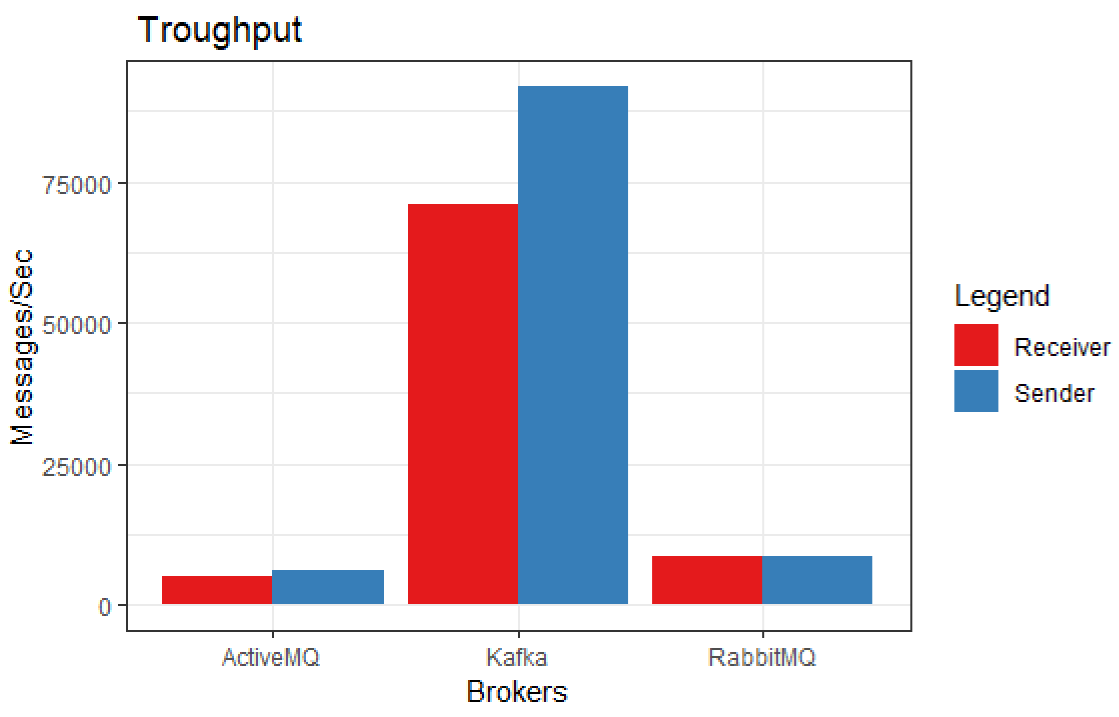

Figure 22. Two Producer - One Consumer Throughput.

Figure 22 shows the throughput for two producers and one consumer. For measuring the throughput 1000000 messages each of 50 Bytes are sent in succession without any delay. The throughput at producer end is the average throughput of the two producers. As the graph shows, for ActiveMQ the throughput for sending and receiving messages is lower than the other two message queues. Kafka sends the highest number of messages sent per second, but the consumption of messages is not as fast as pushing messages to the topic. Even though RabbitMQ does not publish messages as fast as Kafka, the sending and receiving throughput is almost the same. The sender throughput is the average throughput of the two producers.

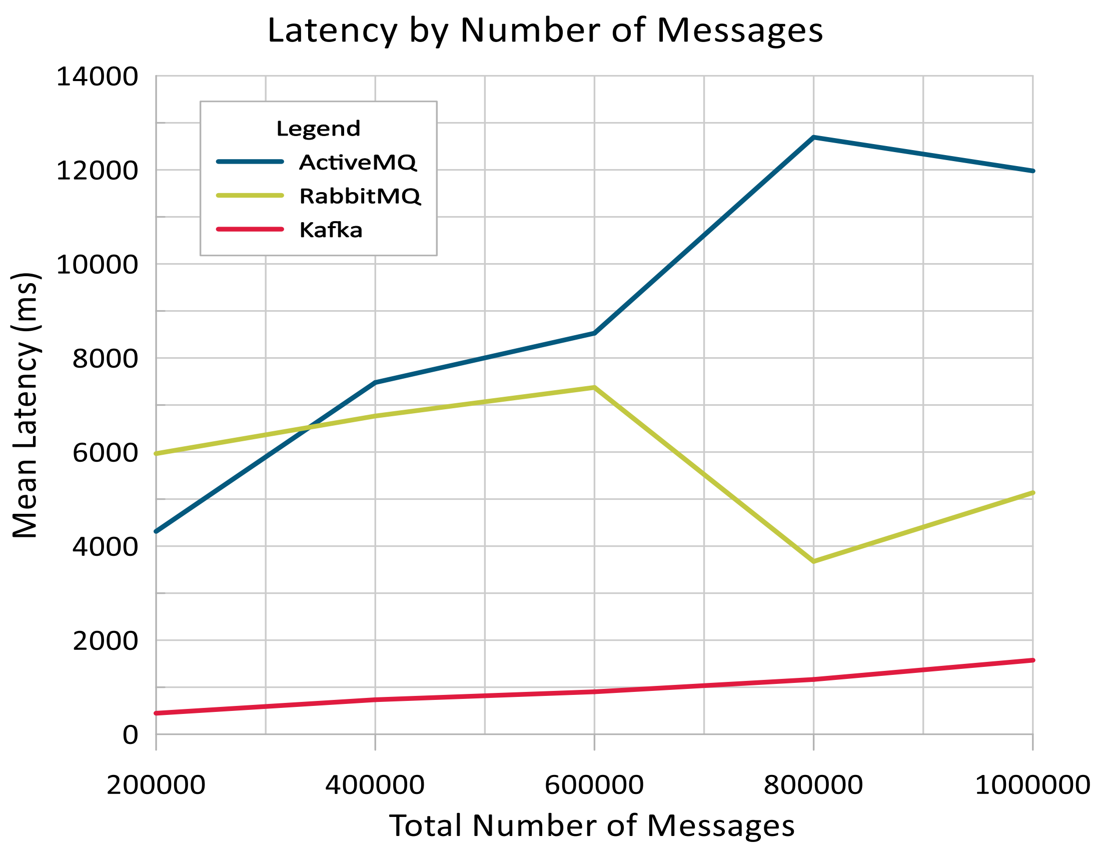

Figure 23. Mean Latency by Number of Messages 2:1.

Figure 23 measures the latency of the message queues based on the number of messages it sends over the queue/topic. Here, each message sent is of 50 Bytes. The mean latency calculated is the average latency between the two producers. The Mean Latency for ActiveMQ increases up to a certain number of messages and then starts decreasing. The latency for Kafka shows a linear increase whereas RabbitMQ shows fluctuations.

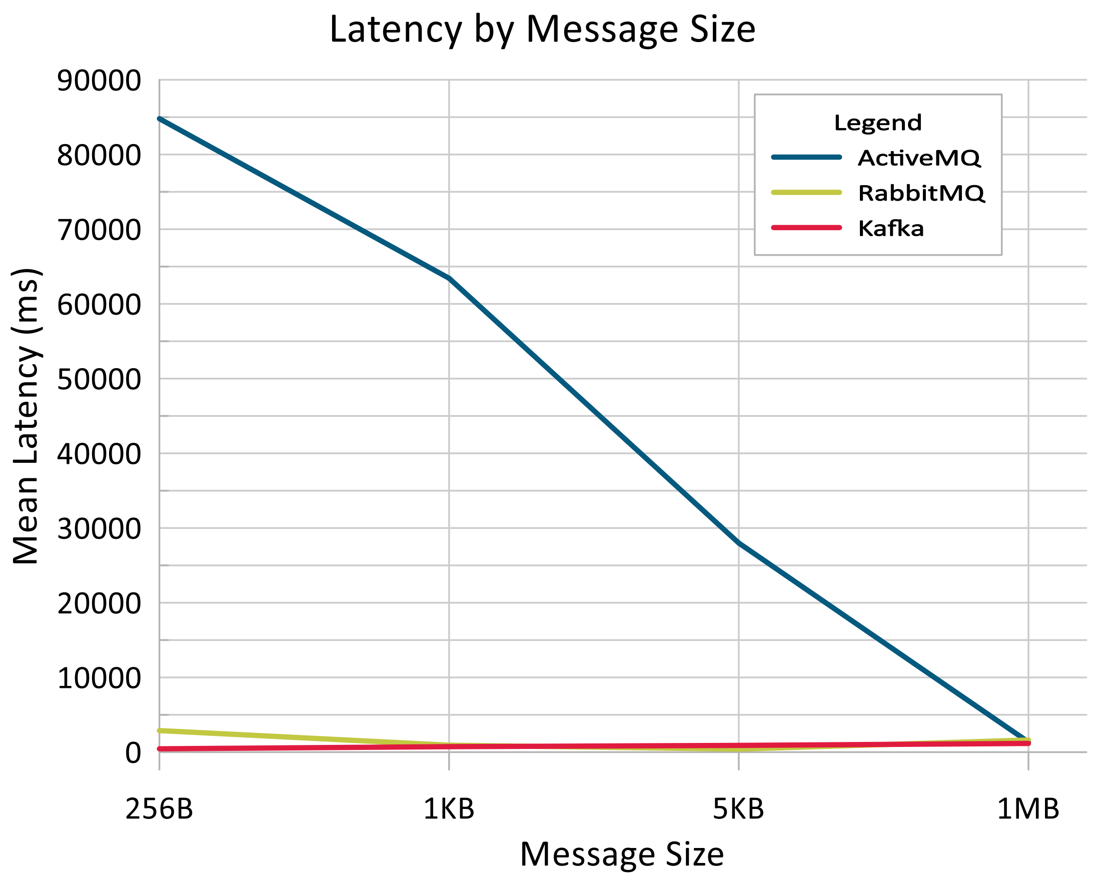

Figure 24. Mean Latency by Message Size 2:1.

Figure 24 Measures the latency based on the size of the message. Here, messages of various sizes namely, 256B, 1KB, 5KB and 1MB are taken into consideration. The total size of data sent over is 1 GB hence the number of messages sent for each message size varies. The mean latency calculated in the average latency between the two producers. The Mean Latency for ActiveMQ decreases with an increase in the message size. The latency for Kafka and RabbitMQ remains constant irrespective of the message size.

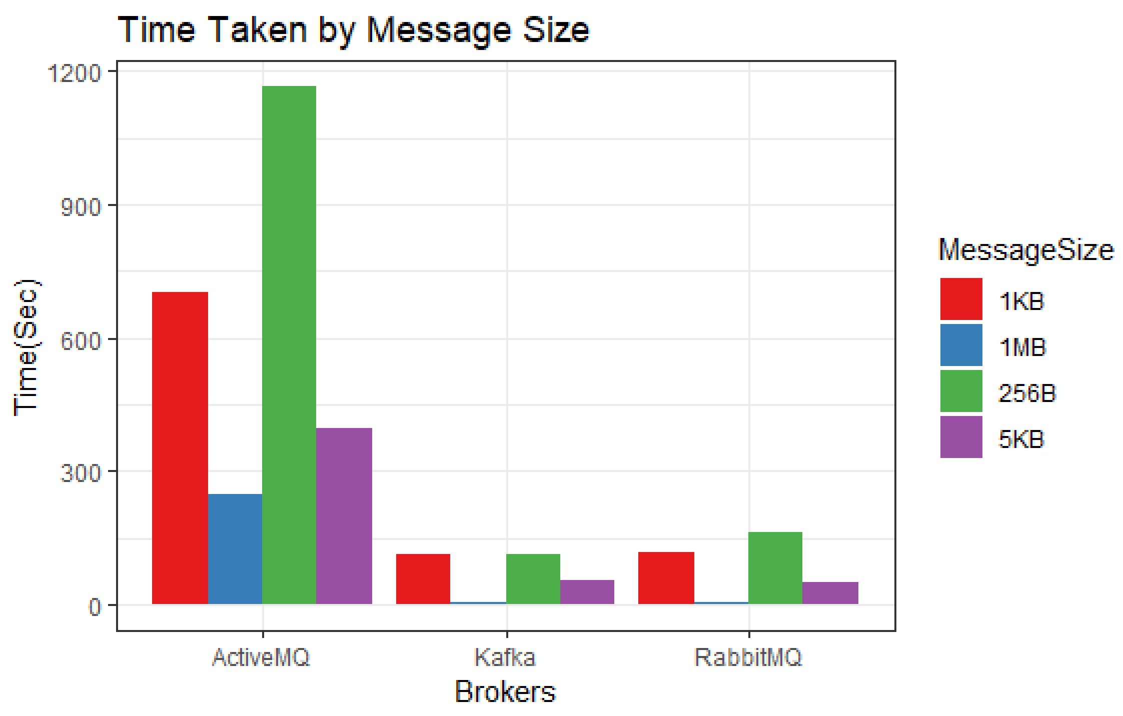

Figure 25. Time taken by Message Size 2:1.

Figure 25 shows the time taken for an entire exchange of messages to happen based on the message size and number of messages. In this, the total data transferred between the producer and consumer is 1GB hence the number of messages sent varies depending on the size of the message. As shown in the chart, the time taken is maximum for ActiveMQ when the size of the message is the smallest i.e. 256 Bytes and decreases as the message size increases. As the size of the messages increase the difference between the time is not much for all three brokers.

5.4 Two Producers – Two Consumers

In Two Producer - Two Consumers model, two producers push messages to the same queue and at the other end there are two consumers listening. In this case, both the producers push the same message onto the queue and both the consumers consume the same message.

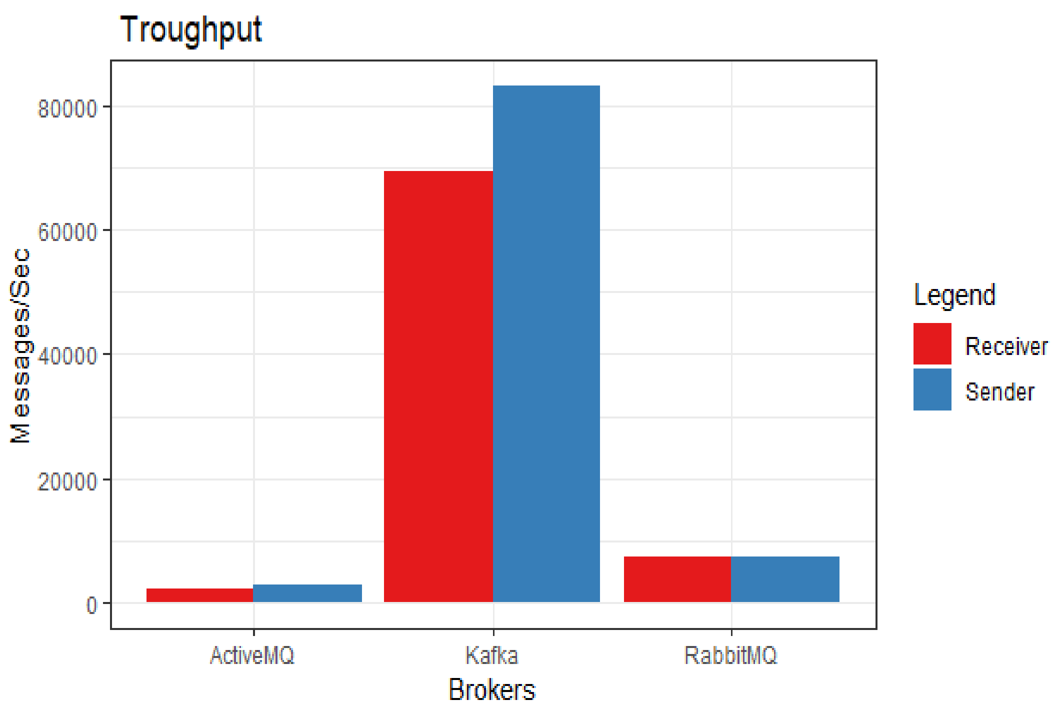

Figure 26. Multiple Producer – Consumer Throughput.

Figure 26 shows the throughput for two producers and one consumer. For measuring the throughput 1000000 messages each of 50 Bytes are sent in succession without any delay. The throughput at producer end is the average throughput of the two producers. As the graph shows, for ActiveMQ the throughput for sending and receiving messages is lower than the other two message queues. Kafka sends the highest number of messages sent per second, but the consumption of messages is not as fast as pushing messages to the topic. Even though RabbitMQ does not publish messages as fast as Kafka, the sending and receiving throughput is almost the same. The sender throughput is the average throughput of the two producers.

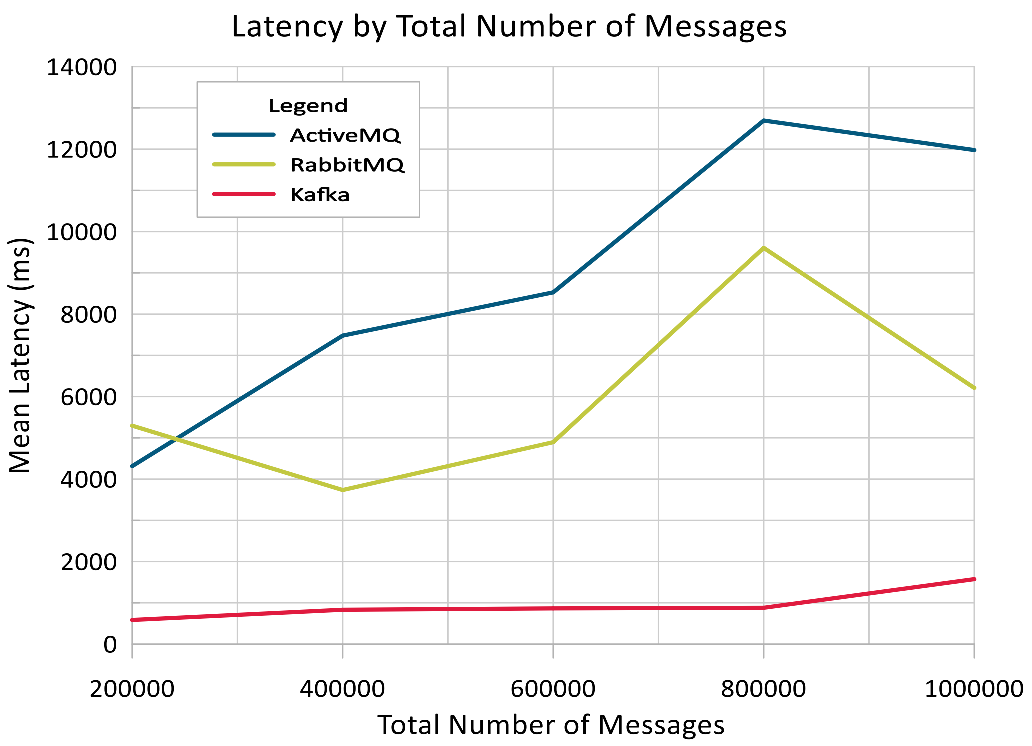

Figure 27. Latency by Number of Messages 2:2.

Figure 27 measures the latency of the message queues based on the number of messages it sends over the queue/topic. Here, each message sent is of 50 Bytes. The mean latency calculated is the average latency between the two producers. The Mean Latency for ActiveMQ increases up to a certain number of messages and then starts decreasing. The latency for Kafka shows a linear increase whereas RabbitMQ shows fluctuations.

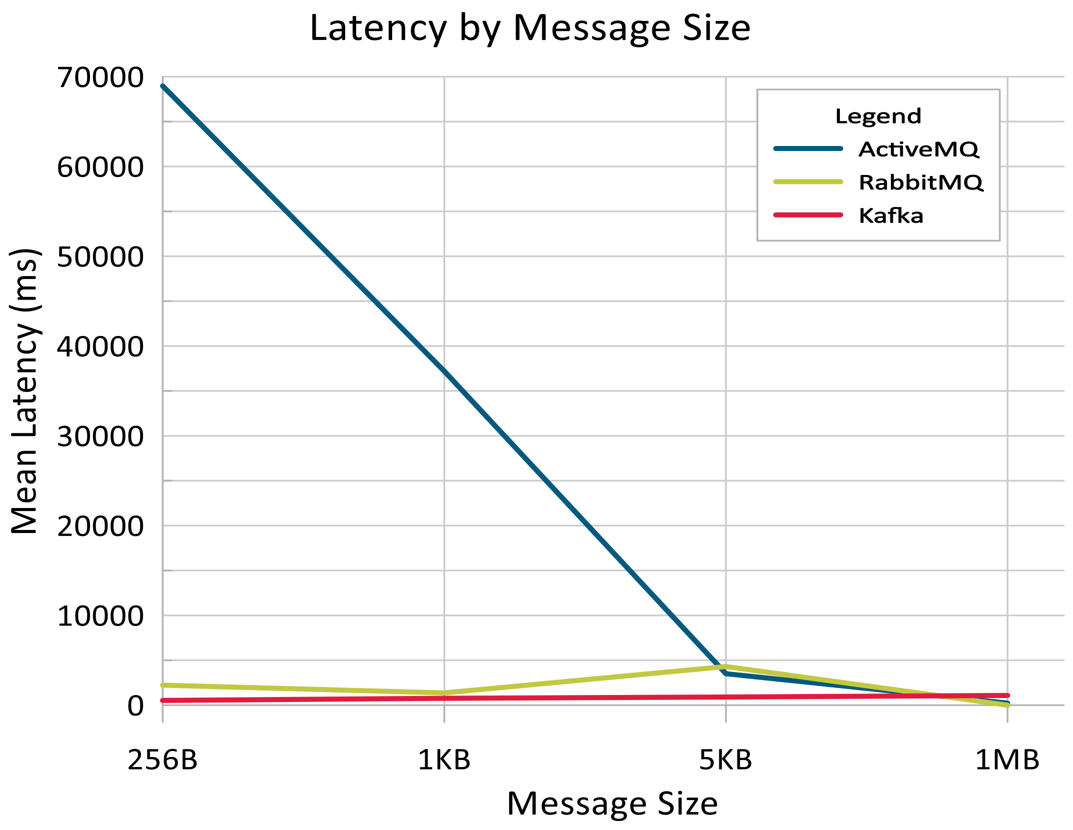

Figure 28. Latency by Message Size 2:2.

Figure 28 Measures the latency based on the size of the message. Here, messages of various sizes namely, 256B, 1KB, 5KB and 1MB are taken into consideration. The total size of data sent over is 1 GB hence the number of messages sent for each message size varies. The mean latency calculated in the average latency between the two producers. The Mean Latency for ActiveMQ decreases with an increase in the message size. The latency for Kafka and RabbitMQ remains constant irrespective of the message size.

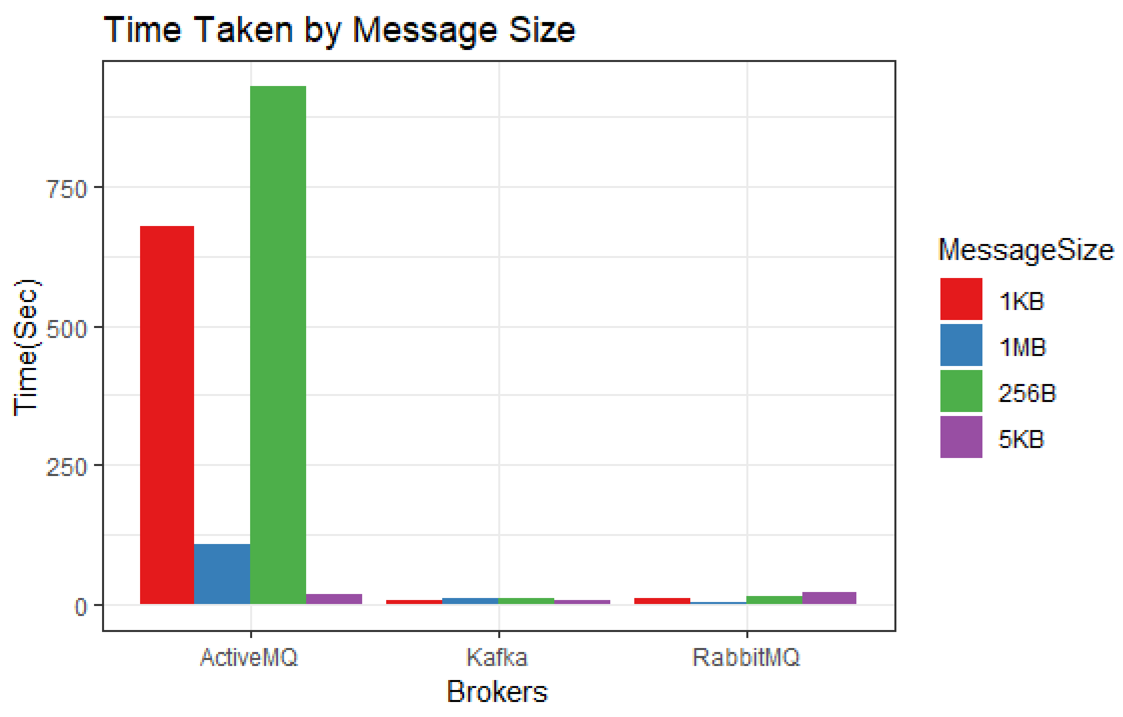

Figure 29. Time taken by Message Size 2:2.

Figure 29 shows the time taken for an entire exchange of messages to happen based on the message size and number of messages. In this, the total data transferred between the producer and consumer is 1GB hence the number of messages sent varies depending on the size of the message. As shown in the chart, the time taken is maximum for ActiveMQ when the size of the message is the smallest i.e. 256 Bytes and decreases as the message size increases. As the size of the messages increase the latency is almost the same for RabbitMQ and Kafka but higher for ActiveMQ.

CHAPTER 6.

CONCLUSION AND FUTURE WORK

On studying the graphs in Chapter 5, the following things have come forward for each of the message broker.

6.1 ActiveMQ

ActiveMQ is an implementation of AMQP. It ensures message delivery. Supports both persistent and non-persistent message delivery. By default, it is asynchronous but can be setup as synchronous. Delivery of messages is guaranteed by using acknowledgements due to which its latency is high. The sender and receiver throughput are good as it does not have much of a difference across all four models. However, as compared to Kafka the number of messages sent per second is very low. Due to guaranteed acknowledgements, the latency is higher for smaller size messages with many messages. The latency goes on decreasing as the message size increases and the total number of messages to be exchanged is less. This is observed across all the four models.

6.2 RabbitMQ

RabbitMQ is an implementation of AMQP. It ensures message delivery. Supports both persistent and non-persistent message delivery. By default, it is asynchronous but can be setup as synchronous. The sender and receiver throughput almost match hence making it fast. But like ActiveMQ, due to guaranteed acknowledgement the throughput is not a match to Kafka. Latency keeps fluctuating when the message size is low, and the number of messages goes on increasing across all the four models. Latency is also high when the number of producers increases, irrespective of the number of consumers listening. Performs best when the message size is high and the total number of messages to be sent is comparatively lower.

6.3 Kafka

Kafka was originally developed by LinkedIn for Message Queuing but was later built up to be a data streaming platform. It is specially designed to operate in clusters so that multiple clients can access it. It makes use of ZooKeeper which helps the brokers to integrate seamlessly and it internally take care of cluster rebalancing. Messages can be easily replayed if there is a failure at the customer end. Kafka clusters can be easily maintained using ZooKeeper but that also means that we have an additional module to maintain. Although the receiver lags in consuming messages as compared to the producer, the throughput of Kafka is unmatched. This is not hindered by the size of the message no matter how small or larger or the total number of messages. The latency for smaller messages can show a slight linear increase as the number of messages increase but is still way less than ActiveMQ or RabbitMQ. The latency is almost constant as the message size increases.

The performance tests conducted in this thesis do not include adding any restrictions onto the brokers or the queues to test performance. There are various ways to hinder the performance of message queues and then benchmark them. There is a lot of scope to explore and work on those aspects of Message Queues.

REFERENCES

[1] Johansson L., “What is message queuing” (2014), [Online] Available: https:// www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html

[2] Mupparaju N., "Performance Evaluation and Comparison of Distributed Messaging Using Message Oriented Middleware" (2013), University of North Florida Graduate Theses and Dissertations.

[3] IBM, “Introduction to Message Queuing” (2019), [Online] Available: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.pro.doc/q00262. htm

[4] Chen, G., Du, Y., Qin, P., & Zhang, L., “Research of JMS Based Message Oriented Middleware for Cluster” (2013), International Conference on Computational and Information Sciences, 1628-1631.

[5] Oracle, “What is BEQ Message Queue” [Online] Available: https://docs.oracle.com /cd/E13203_01/tuxedo/msgq/msgq40a/intro/html/i_chap1.htm

[6] Perry M., “Message Queues” (2011), [Online] Available: https://historicalmodeling.com/ distributed-systems/message-queues.html

[7] Kato, Kasumi & Takefusa, Atsuko & Nakada, Hidemoto & Oguchi, Masato., “A Study of a Scalable Distributed Stream Processing Infrastructure Using Ray and Apache Kafka” (2018), 5351-5353. 10.1109/BigData.2018.8622415.

[8] Wikipedia, “Message Queue”, [Online] Available: https://en.wikipedia.org/wiki/ Message_queue

[9] Linagora Engineering, “Hot to choose a message queue” (2018), [Online] Available: https://medium.com/linagora-engineering/how-to-choose-a-message-queue-247dde46e66c

[10] Sustrik M., “Broker vs. Brokerless” (2017), [Online] Available: http://zeromq.org /whitepapers:brokerless

[11] Treat T, “Dissecting Message Queues” (2014), [Online] Available: https:// bravenewgeek.com/dissecting-message-queues/

[12] John, V., & Liu, X., “A Survey of Distributed Message Broker Queues” (2017) CoRR, abs/1704.00411.

[13] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ" (2015), 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), Craiova, 2015, pp. 132-137. doi: 10.1109/RoEduNet.2015.7311982

[14] Sookocheff K., “Kafka in a nutshell” (2015), [Online] Available: https:// sookocheff.com/post/kafka/kafka-in-a-nutshell/

[15] Azhar JP., “What is Kafka” (2017), [Online] Available: https://dzone.com/articles/what-is-kafka

[16] Artyom N., “Comparison of ActiveMQ, RabbitMQ and ZeroMQ Message Oriented Middleware” (2019), [Online] Available: https://vironit.com/comparison-of-activemq-rabbitmq-and-zeromq-message-oriented-middleware/

[16] Confluent, “What is Apache Kafka” (2019), [Online] Available: https://www.confluent.io/what-is-apache-kafka/

[17] Apache ActiveMQ, “Virtual Destinations” (2019), [Online] Available: https://activemq.apache.org/virtual-destinations.html

[18] Kelly B., “Message Confusion pub-sub vs. multicast vs. fanout” (2011), [Online] Available: https://stackoverflow.com/questions/8261654/messaging-confusion-pub-sub-vs-multicast-vs-fan-out

CURRICULUM VITAE

GRADUATE COLLEGE

UNIVERSITY OF NEVADA, LAS VEGAS

SANIKA N. RAJE

rajesanika7@gmail.com

Degrees:

- Bachelor of Science – Information Technology, 2012

  University of Mumbai, India

- Master in Computer Applications, 2015

  University of Mumbai, India

- Master of Science in Computer Science, 2019

  University of Nevada, Las Vegas

Thesis Title: **Performance Comparison of Message Queue Methods**

Thesis Examination Committee:

- Committee Chair, Dr. Yoohwan Kim, Ph.D.
- Committee Member, Dr. Ju-Yeon Jo, Ph.D.
- Committee Member, Dr. Fatma Nasoz Ph.D.
- Graduate College Representative, Dr. Sean Mulvenon Ph.D.