Analyzing Parameter Sets For RabbitMQ and Apache Kafka On A Cloud Platform

Amir Rabiee
Master of Science - Software Engineering of Distributed Systems

19-6-2018
Supervisor - Lars Kroll
Examiner - Prof. Seif Haridi

Abstract

Applications found in both large enterprises and small needs a communication method in order to meet criteria of scalability and durability. There are many communication methods out there but one of the most well used is the use of message queues and message brokers. The problem is that it exist a plethora of different types of message queues and message brokers all unique regarding their design and implementation. These design choices results in different parameter sets that can be configured to fit different criterias such as high durability, throughput, availability etc.

This thesis tests two different message brokers, Apache Kafka and RabbitMQ with the purpose of discussing and showing the impact of using different parameters. The experiments conducted are focused on two primary metrics, latency, and throughput with secondary metrics such as disk usage and CPU usage. The parameters chosen for both RabbitMQ and Kafka is optimized with focus on the primary metrics. The experiments conducted are tested on a cloud platform, Amazon Web Services.

The results show that Kafka outshines RabbitMQ when it comes to throughput and latency but it also shows the impact that both Kafka and RabbitMQ has when it comes to the amount of written data, with RabbitMQ being the most efficient in terms of quantity of data being written while on the other hand being more CPU-heavy than Kafka.

Keywords: Kafka, RabbitMQ, throughput, latency, cloud platform, testing

Sammanfattning

Applikationer som finns i både komplexa och icke-komplexa system behöver en kommunikationsmetod för att uppfylla kriterierna för skalbarhet och hållbarhet. Det finns många kommunikationsmetoder där ute, men en av de mest använda är användningen av meddelandeköer och meddelandemäklare. Problemet är att det finns en uppsjö olika typer av meddelandeköer och meddelande mäklare som är unika med avseende på deras design och implementering. Dessa lösningar resulterar i olika parametersatser som kan konfigureras för att passa olika kriterier, exempelvis hög hållbarhet, genomströmning, tillgänglighet.

Denna avhandling testar två olika meddelandemäklare, Apache Kafka och RabbitMQ med syfte att diskutera och visa effekterna av att använda olika parametrar. De utförda experimenten är inriktade på två primära mätvärden, latens och genomströmning med sekundära mätvärden, diskanvändning och CPU-användning. De parametrar som valts för både RabbitMQ och Kafka optimeras med fokus på de primära mätvärdena. Experimenten som genomförs testades på en molnplattform, Amazon Web Services.

Resultaten visar att Kafka presterar bättre än RabbitMQ när det kommer till genomströmning och latens men det visar också den inverkan som både Kafka och RabbitMQ har när det gäller mängden skriven data, där RabbitMQ är den mest effektiva när det gäller datamängden som skrivs medan den å andra sidan är mer CPU-tung än Kafka.

Nyckelord: Apache Kafka, RabbitMQ, genomflöde, latens, molnplattform, testning

Contents

Chapter 1

Introduction

The world of applications and the data being generated and transfered to and from them are constantly evolving in a fast paced manner. The amount of data generated over the Internet and the applications running on it are estimated for 2020 to hit over 40 trillion gigabytes [1].

The applications handling this data has requirements that needs to be met such as having high reliability and high availability. Because of the high demands for such requirements, a natural outcome of this is to build a distributed system that can support these applications whether they be a load balancing system or a game application or anything in between [2, p. 1].

The evolving of distributed systems stems from the relative cheap hardware commodity that one has been able to utilize to build networks of computers communicating together for a specific purpose. These systems are in comparison to the client-server architecture constructed of different applications running on multiple separated machines. Because of the setup of having multiple machines coordinating together for a common task, an innately advantage for them is that distributed systems are more scalable, reliable and faster when architectured correctly in comparison to a client-server model. The advantages with these systems come with a cost, as designing, building and debugging distributed systems are more complicated than a client-server model [2, p. 2].

In order to meet the scalable part of a distributed system, appropriate methods have to be taken, to meet the need when the communication between different machines and their applications are put to the test. To help facilitate the problems that can occur when an application on one machine tries to communicate with a different application on another machine one can use **message queues** and **message brokers** [3, p. 2].

The message brokers works together with the message queues to help

deliver messages sent from different destinations and route them according to the correct route [4, p. 326].

1.1 Background

The distributed systems that are developed to meet the requirements of having high availability and reliability are built upon some abstract message form being sent from one process located on one machine to another process on a different machine. Therefore a distributed system needs an architecture or system that can distribute messages in order to achieve higher scalability and reliability.

The architecture used, that strives to fulfil the requirements, is called *message-oriented middleware* (MOM), this middleware is built on the basis of an asynchronous interaction model which enables users to continue with their execution after sending a message [5, p. 4]. More importantly a message oriented middleware is used for distributed communication between processes without having to adapt the source system to the destination system. This architecture is illustrated in Figure 1.1 where the apps in this context refers to the various systems using the MOM.

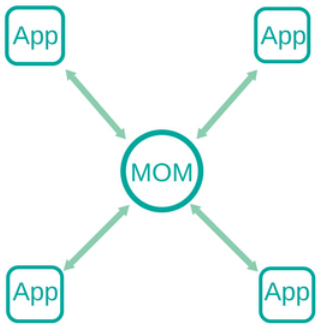

Figure 1.1: Structure of message oriented middleware.

A central structure when using a message oriented middleware is the usage of **message queues**, these queues are used to store messages on the MOM platform. The systems using the MOM platform will in turn use the queues to send and receive messages through them. These queues

have many configurable attributes, which include the name, size, the sorting algorithm for the queue and so forth [5, p. 7].

In order to efficiently distribute the messages to different queues one has to resort to a **message broker**, these message brokers can be seen as an architectural pattern according to [6, p. 288], the broker is responsible for routing and translating different message formats between applications.

An important part of message brokers is their usage of message queue protocols, there are many different types of protocols that can be used such as; **Streaming Text Oriented Messaging Protocol** (STOMP), **Extensible Messaging and Presence Protocol** (XMPP), **Message Queueing Telemetry Transport** (MQTT) and **OpenWire** and more [7, p.3].

The unique capabilities of the MOM-model comes from the messaging models that it uses, there are mainly two messaging models to be found, **point-to-point** and **publish/subscribe**.

The point-to-point model provides a communication link between the producer and consumer with the usage of a message queue, the consuming clients processes messages from the queue where only one receiver consumes the message albeit not being a strict requirement [5, p. 9], these messages are delivered **exactly once**.

In the publish/subscribe model, the producer produces a message to a specific topic, the consumers interested in these messages will subscribe to the topic, and thereafter be routed by a publish/subscribe engine.

1.2 Problem

With a plethora of different message brokers available to help with implementing a message oriented middleware, choosing one is a multifaceted question that has many different aspects that need to be taken in consideration. Every message broker has their own design and implementation goals and can therefore be used for different purposes and situations. An overview of some message brokers and the protocols supported can be seen in Table 1.1.
The message brokers found in Table 1.1 are widely used and can be deployed on different server architectures and platforms [8].

Table 1.1: Message brokers and their supported protocols

Message broker	**Protocols supported**
Apache Kafka	Uses own protocol over TCP
RabbitMQ	AMQP, STOMP, MQTT,
ActiveMQ	AMQP,XMPP, MQTT, OpenWire

The message brokers Apache Kafka and RabbitMQ, needs to be tested on their enqueueing performance with focus on different parameter sets to get a deeper understanding of the impact of the latency and throughput with the parameter sets. With this in mind the main problem and research question is: How does different parameter sets affect the performance of RabbitMQ and Kafka on a cloud platform?

1.3 Purpose

Because of the intricacy of the different message brokers and their corresponding protocols they use it is a relative difficulty in grasping both the fine grained differences between them as well as the coarse grained. This thesis discusses and shows an overview of the available messaging middleware solutions and aims to validate and verify the enqueueing performance for two message brokers. The enqueueing performance focuses on the throughput aspect versus the latency, moreover the thesis focuses on the resource usage of both the message brokers such as CPU and memory during load.

Furthermore the two different message brokers RabbitMQ and Apache Kafka is a debatable topic on deciding which one to use [9], this thesis work will try to shed a light on this subject.

The thesis work will present the designed testing experiments and the results of them, in order to visualize, interpret and explain the performance of the two message brokers running on the cloud platform Amazon Web Service.

1.4 Goal

The goals of this project is presented below:

- Design experiments for testing the enqueueing performance for Apache Kafka and RabbitMQ.
- Compare the results from each experiment and discuss the findings with regards to the the cloud platform and the respective message broker.
- Evaluate with general statistical analysis the results.
- Use the project as reference material when analyzing Apache Kafka and RabbitMQ for their enqueuing performance.

These goals presented lays the foundation for this thesis work and the results derived from the goals can be further used as a reference point for when to use RabbitMQ over Kafka and vice versa.

1.5 Benefits, Ethics and Sustainability

With the amount of data being generated in the present day one has to think of the ethical issues and aspects that can arise with processing and storing this much information and what the possibilities are with extracting valuable data from this content.

This thesis work is not focused on the data itself that is being stored, sent and processed, but rather the infrastructures which utilizes the communication passages of messages being sent from one producer to a consumer. Nonetheless the above mentioned aspects are important to discuss, because from these massive data-sets being generated one can mine and extract patterns and human behaviour [10], which in turn can be used to target more aggressive advertisements to different consumer groups.

Moreover another important aspect to be brought up is the privacy and security of peoples personal information being stored which can be exposed and used for malicious intent [11], this will be remedied to a degree with the introduction to the new changes in the General Data Protection Registration that comes into effect May 2018 [12] .

With the usage of large cloud platforms and their appropriate message

brokers that is used to send millions of messages between one destination to another, one has to think of the incumbent storage solutions for the data, which in this case has resulted in the building of large datacenters. These datacenters consumes massive amount of electricity, up to several megawatts [13].

The main benefitters of this thesis work are those standing at a crossroads of choosing a message broker for their platform or parties interested in a quantitative analysis focused on the enqueuing performance of two message brokers, Apache Kafka and RabbitMQ on a cloud platform such as Amazon Web Services.

1.6 Methodology

The focus of this thesis work is to analyze and test the enqueuing performance of two different message brokers, and with that in mind, one has to think of the different methodologies and research methods that are to avail and that are the most suitable to conduct such a thesis work. The fundamental choosings of a research method is based on either a *quantitative* or *qualitative* method, both of these have different goals and can be roughly divided into either a numerical or non-numerical project [14, p. 3].

The quantitative research method focuses on having a problem or hypothesis that can be measured with statistics and validified as well as verified, a qualitative research on the other hand focuses on opinions and behaviours to reach a conclusion and to form a hypothesis.

For this project the most suitable research method is of quantitative form because of the experiments and tests to be run gathers numerical data.

Another important aspect to be chosen for the thesis work is the *philosophical assumption* that can be made and there are several school of thoughts to be considered. Firstly the *positivism* element relies on the independency between the observer and the system to be observed as well from the tools to be measured with. Another philosophical school is the *realistic*, which collects data from observed phenomenons and thereafter develop knowledge. Thirdly a *criticalism* element is the one which focuses on learning how users can affect different types of computer sys-

tems. [14, p. 4]

There are several others philosophical principles but for this project the most fitting ones was to use a combination of both the positivism as well as the realistic.

Moreover to continue with the experimental testing of the enqueuing performance of the different message brokers a research method that fits the requirements of the thesis work had to be chosen. There are mainly two divisions of research methods, a *experimental* or a *non-experimental* research method.

Because of the nature of the thesis residing in experimenting with the correlation of variable changes and their relationship between one another in order to see how the message brokers becomes affected an obvious choosing would be a experimental research methodology. An *analytical* research method based on previous testing of both RabbitMQ and Apache Kafka is also showcased for more a comprehensive conclusion [14, p. 4].

1.7 Delimitations

This report will not go in to depth of how the operating system affects the message queues because the workings used for memory paging are to complex to track and needs a further explanation depending on what operating system being used. Furthermore this thesis will not explore how different protocols can affect the performance of the brokers.

1.8 Outline

In text, describe what is presented in Chapters 2 and forward. Exclude the first chapter and references as well as appendix.

- **Chapter 2** shows a more in-depth technical background of Apache Kafka and RabbitMQ and their protocols.
- **Chapter 3** presents the research methodologies of the thesis work with the appropriate statistical tools.
- **Chapter 4** will present the different experiments conducted

- **Chapter 5** will present and visualize the results from the experiments.
- **Chapter 6** will discuss the results and the conclusions that can be drawn from the thesis work.

Chapter 2

Background

In order for two different applications to communicate with each other a mutual interface must be agreed upon. With this interface a communication protocol and a message template must be chosen such that an exchange can happen between them. There are many different ways of this being done, one can define different types of schemas in a programming language or let two developers agree upon a request of some sort. Provided this mutual agreement between the two applications then it is of no importance for these applications to know the intricacies of each others system. Furthermore, the programming language and framework can over the years change for the applications but as long as the mutual interface exists between these two applications they will always be able to communicate with each other which results in lower coupling between the applications.

To further attain lower coupling one can introduce *messaging systems* or *message-oriented middleware* (interchangeable terms). The messaging system enables the communication between sender and receiver to be less conformative in a way where it is not necessary for the sender to store information on how many instances of receivers there are, where they are located or whether they are active [15, p. 3]. The message system is responsible for the coordination of message passing between the sender and receiver and has therefore a primary purpose of safeguarding the communication between two parties [4, p. 14].

Because of the unreliability of networks and computers, the reason message systems exists is, to help mitigate the problem of losing mesages when a receiver has gone down. The message system in this case will try to resubmit the message until it is successful.

There are different types of models that a message system can use but there are mainly three different types of communication models, a point-to-point, publish/subscribe or a hybrid of those two.

2.1 Point-to-Point

The point-to-point communication model works as such, a sender sends a message to a queue and can thereafter continue with its execution without having to wait for an acknowledgment from the receiver. The receiver does not have to be available when the message is sent and can process it whenever it wants to. This communication model is implemented with queues that uses a first in first out schema. This leads to only one of the subscribed consumers receiving the message, this can be seen in Figure 2.1.

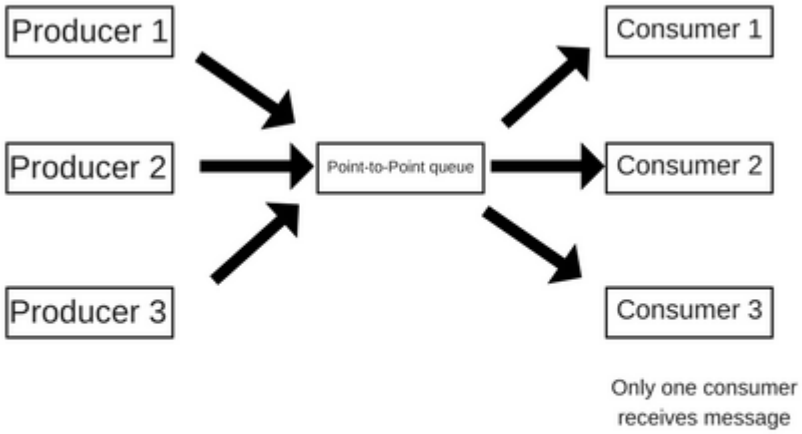

Figure 2.1: A point-to-point communication domain.

The usage of point-to-point communication is found in applications where it is important that you execute something once for example a load balancer or transfering money from one account to another.

Queues have important attributes, more specifically *persistence* and *durability*. The persistence attribute focuses on if a failure happens during

message processing, if this happens, then it is important that the message is not lost next time it is processed. This is done by storing the message to some other form than in-memory for example a temporary folder, a database or a file etc [16].

The durability attribute is when a message is sent to a queue and the queue is offline, and thereafter the queue comes back online, it is of importance that the queue fetches the messages that was lost during the downtime.

With persistence the reliability increases but at the expense of performance and it all boils down to design choices for the message systems to choose how many of the messages should be persisted and in what ways.

2.2 Publish/Subscribe

The publish/subscribe communication model works as such, a sender (publisher) can publish messages to a general message-holder from which multiple receivers (subscriber) can retrieve messages. The sender does not need to have information on how many receivers there are and the receiver does not need to keep track on how many senders there are, the receiver can process the messages anytime.

This type of model can be implemented with the help of *topics*. A topic can be seen as an event from which subscribers are interested in and whenever a message is sent to a topic from a publisher all the subscribers subscribed to that topic becomes notified of the update.The publishers can publish messages to a topic and the subscribers can either subscribe or unsubscribe from the topic. The publisher/subscribe communication model utilizes three dimensionalities the *temporal*, *spatial* and *synchronization*, which enhances message passing [17, p. 1]. This architecture is illustrated in Figure 2.2.

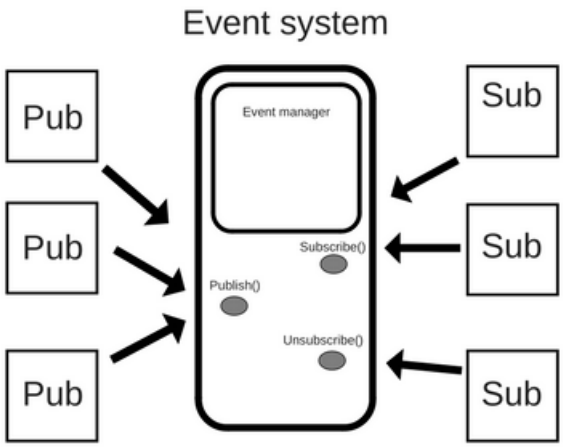

Figure 2.2: A publisher/subscriber domain.

With the *spatial* dimensionality, the requirement of two interacting parties needing to know each other is not present. Because the publishers needs only to publish their message to a topic with an event service as a middle hand and the subscribers need only to interact with event service [17, p. 2].

The *temporal* dimensionality is that there are no requirements of both the publisher and subscriber to be active at the same time, that is, the publisher can send events to a topic while a subscriber is offline which in turn infers that a subscriber can get an event that was sent after the publisher went offline.

The *synchronization* dimensionality means that a publisher does not need to wait for a subscriber to process the event in the topic in order to continue with the execution. This works in unity with how a subscriber can get notified asynchronously after doing some arbitrary work.

A secondary type of implementation is the *content-based* publish/subscribe model which can be seen as an extension of a topic based approach. What differs a content-based from a topic-based is that one is not bound to an already defined schema such as a topic name but rather to the attributes of the events themselves [18, p. 4]. To be able to filter out

certain events from a topic one can use a subscription language based on constraints of logical operators such as or, and, not etc [17, p. 9].

2.3 Communication protocols

There are many different communication protocols that can be used when you have to send a message from one destination to another, for example AMQP, XMPP, STOMP and MQTT. These protocols has their own design goals and use cases that are more fitting than others. There are several use cases to think of such as, how scalable is the implementation, how many users will be sending messages, how reliable is sending one message and what happens if a message can not be delivered etc.

2.3.1 Advanced Message Queueing Protocol

Advanced Message Queuing Protocol (AMQP) was initiated at the bank of JPMorgan-Chase with the goal of developing a protocol that provided high durability during intense volume messaging with a high degree of interoperability. This was of high importance in the environment of banking because there is an economic impact if a message is delayed, lost or processed incorrectly [19].

AMQP provides a rich set of features for messaging with a topic-based publish/subscribe domain messaging, flexible routing, security etc and is used by large companies that process over billion of messages a day ranging from JPMorgan Chase, NASA and Google [20].

The intricacies of how the AMQP protocol model is designed can be seen in Figure 2.3.

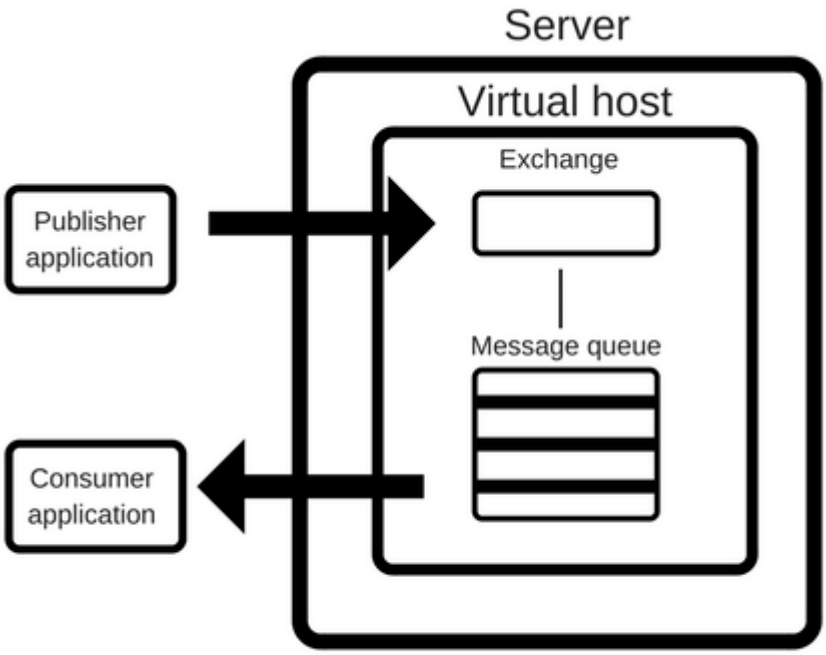

Figure 2.3: AMQP Protocol model

The exchange in Figure 2.3 accepts messages from producers and routes them to message queues, and the message queues stores the messages and forwards them to the consumer application [21].

The message queues in AMQP is defined as **weak FIFO** because if it exists multiple readers of a queue the one with the highest priority will take the message before the others. A message queue has the following attributes which can be configured:

- Name - Name of the queue.
- Durable - If the message queue can lose a message or not.
- Exclusive - that is if the message queue will be deleted after connection is closed.
- Auto-delete - the message queue can delete itself after the last consumer has unsubscribed.

In order to determine which queue to route a specific message from an exchange, a binding is used, this binding is determined with help of a routing key [22, p. 50].

There are several different types of exchanges found in AMQP, there is the direct type, the fan-out exchange type, topic and lastly the headers exchange type.

Direct type

This exchange type will bind a queue to the exchange using the routing key K, if a publisher sends a message to the Exchange with the routing key R, where K = R then the message is passed to the queue.

Fan-out

This exchange type will not bind any queue to an argument and therefore all the messages sent from a publisher will be sent to every queue.

Topic

This exchange will bind a queue to an exchange using a routing pattern P, a publisher will send a message with a routing key R, if R = P then the message is passed to the queue. The match will be determined for routing keys R that contain one or more words, where each word is delimited with a dot. The routing pattern P works in the same way as a regular expression pattern. This can be seen in figure 2.4.

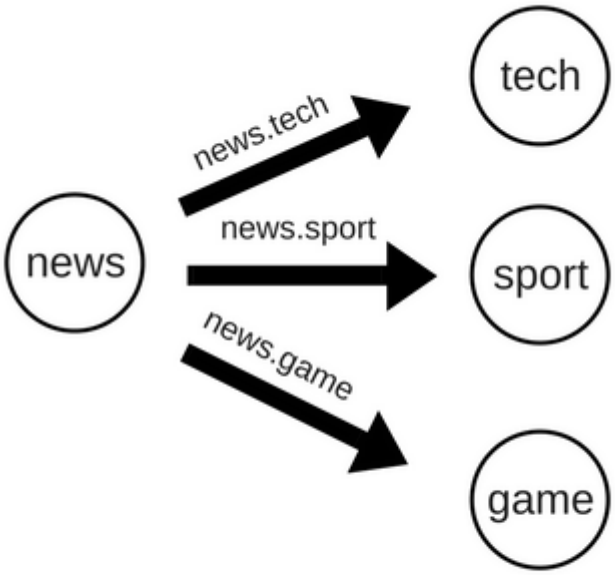

Figure 2.4: Topic exchange type

Headers

The headers exchange type will prioritize how the header of the message looks like and in turn ignores the routing key.

2.3.2 Extensible Messaging and Presence Protocol

The Extensible Messaging and Presence Protocol (XMPP) technologies were invented because of the abundance of different client applications for instant messaging services. The XMPP is an open technology in the same way as Hypertext Transfer Protocol (HTTP). The specifications of XMPP puts focus on the protocols and data entities that are used for real-time asynchronous communication such as instant messaging and streaming [23, p. 7]

XMPP makes use of the Extensible Markup Language (XML) to enable an exchangement of data from one point to another. The technologies of XMPP is based on a decentralized server-client architecture much alike

how the world wide web is deployed, this means, if a message is sent from one destination, the initial message is sent to an XMPP server and thereafter sent to the receivers XMPP server and finally to the receiver.

To be able to send a message to a person located somewhere else, XMPP sets up a XML-stream to a server and there after the two clients can exchange mesages with the help of three so called "stanzas", <message/>, <presence/> and <iq/> which are XML-elements. These stanzas can be seen as a data packet and are routed differently depending on what stanza it is.

The <message/> is what is used for pushing data from one destination to another, and is used for instant messaging, alerts and notifications [23, p. 18]. The <presence/> is one of the key concepts of real-time communications because this stanza enables others to know if a certain domain is online and ready to be communicated with. The only way for a person to see that someone is online is with the help of a presence subscription which employs a publisch-subscribe method. The info/query stanza <iq/> is used for implementing a structure for two clients to send and receive requests in the same way GET, POST and PUT methods are used within the HTTP.

What differs the iq stanza from the message stanza is that the iq has only one payload that the receiver must reply with and is often used to process a request. For error handling the XMPP does not acknowledge all of the packets sent over a communication link and XMPP assumes that a message or stanza is always delivered unless an error is received [23, p. 24].

The difference between a stanza error and a regular message error is that a stanza error can be recovered while other messages results in the closing of the XML stream that was opened in the start.

2.3.3 Simple/Streaming Text Oriented Messaging Protocol

The Simple/Streaming Text Oriented Messaging Protocol (STOMP) is a simple message exchange protocol aimed for asynchronous messaging between entities with servers acting as a middlehand. This protocol is not a fully pledged protocol in the same way as other protocols such as AMQP or XMPP, instead STOMP adheres to a subset of the most com-

mon used message operations [24].

STOMP is loosely modeled on HTTP and is built on frames, these frames is made of three different components, primarily a command, a set of optional headers and body. A server that makes use of STOMP can be configured in many different ways because STOMP leaves the handling of message syntax to the servers and not to the protocol itself. This means that one can have different delivery rules for servers as well as for destination specific messages.

STOMP employs a publisher/subscriber model where the client can both be a producer by sending frame containing SEND as well as being a consumer, this is done by sending a SUBSCRIBE frame.

For error handling and to stop malicious actions such as exploiting memory weaknesses on the server STOMP allows the servers to put a threshhold on how many headers there are in a frame, the lengths of a header and size of the body in the frame. If any of these are exceeded the server has to send an ERROR frame back to the client.

2.3.4 Message Queue Telemetry Transport

The Message Queue Telemtry Transport (MQTT) is a lightweight messaging protocol with design goals aimed to an easy implementation standard, having a high quality of service data delivery and being lightweight and bandwith efficient [25, p. 6]. The protocol uses a publish/subscribe model and is aimed primarily for machine to machine communication, more specifically embedded devices with sensor data.

The messages that are sent with a MQTT protocol are lightweight because it only consists of a header of 2 bytes and a payload of maximum 256 MB and a Quality of Service level (QoS) [26]. There are 3 types quality of service levels which are listed below

1. Level 0 - Employs a at-most-onec semantic where the publisher sends a message without an acknowledgement and where the broker does not save the message, more commonly used for sending non-critical messages.

2. Level 1 - Employs an at-least-once semantic where the publisher receives an acknowledgement atleast once from the intended recipient. This is done by sending a PUBACK message to the publishers

and until then the publisher will store the message and try to resend it. This type of message level could be used for shutting down nodes on different locations.

3. Level 2 - Employs an exactly-once semantic and is the most reliable level because it guarantees that the message is received, this is done by first sending a message stating that a level 2 message is inbound to the recipient, the recipient in this case replies that it is ready, the publisher relays the message and the recipient acknowledges it.

Moreover MQTT deploy something called a "Last Will and Testament" (LWT) for error handling, if a client disconnects abruptly which can be seen during power outages or unexepected network disturbances.

LWT is configured in the start for a client that connects to a broker, the broker will store it until a client disconnects abruptly, and broadcast the message to all subscribers that are connected to the topic that is published by the client. This ensures that the right precautions or actions are taken in the case of having a dead publisher.

Because of MQTT being a lightweight message protocol, the security aspect is flacking and the protocol does not include any security implementations of it its own as it is implemented on top of TCP, one is resorted to use SSL/TLS certifications on the client side for securing the traffic.

An indirect consequence of using SSL/TSL for encryptions is that it augments a significant overhead to the messages which in turn goes against the philosophy of MQTT being a lightweight protocol [27]. This is something that left for the developer to think about whether it is feasible to have more data being sent over the wire which can affect performance.

2.4 Apache Kafka

Apache Kafka is a distributed streaming platform used for processing streamed data, this central platform scales elastically instead of having an individual message broker for each application [28]. Kafka is also a system used for storaging as it replicates and persists data in infinite time, the data is stored in-order and is durable can be read deterministically [28, p. 4].

The messages written in Kafka are batch-processed, and not processed individually which is a design choice that favours throughput over latency, furthermore messages in Kafka are partitioned into *topics* and these are further divided into a set of *partitions*. The partitions contains messages that are augmented in an incremental way, and is read from lowest index to highest [28, p. 5].

The time-ordering of a message is only tied with one partition and not with rest of the partitions of a topic, each of these partitions can be allocated on different servers and is a key factor to enhancing scalability horizontally. This is illustrated in Figure 2.5

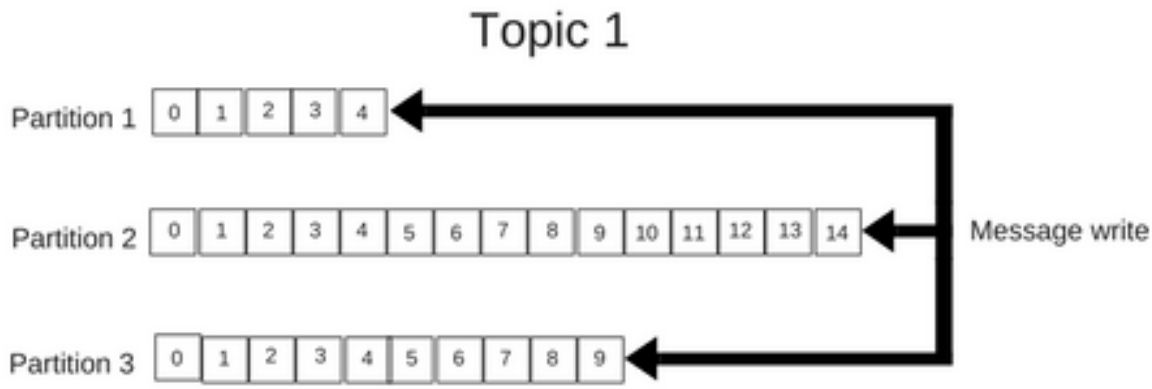

Figure 2.5: Topic partitioning

The messages within a partition is consumed by a reader and to keep track of which message has been read an *offset* is kept in the metadata if a reader stops and starts reading later on.

A consumer cooperates with other consumers in a group to deplete a topic and there can only be one consumer from a consumer group reading from a partition. If a consumer fails, the group will rebalance itself to take over the partition that was being read [28, p. 7].

A Kafka server also called a *broker*, is responsible for commiting messages to the disk whenever possible and giving offset measurements for producers, and responds to consumers requesting for messages of a partition. A broker will function with other brokers to form a *cluster*. This cluster deploys a leadership architecture where one broker will service as a leader for a partition and a so called *controller* is used for appointing

leaders and to work as a failure detector. To increase the durability multiple brokers can be appointed to a partition and the partition is therefore replicated to other brokers.

2.4.1 Apache Zookeeper

To help maintain a cluster of Kafka servers, Apache Kafka utilizes Zookeeper to help keep track on metadata of the clusters and information regarding consumers. Zookeeper works as key-value manager that can help out with the synchronization aspects for Kafka such as leader election, crash detection, group membership management and metadata management [29, p. 11].

Zookeeper and Kafka works in symbiosis, where ZooKeeper tries to offer more control to issues that arises in a system with multiple clusters. Examples of failures that can happen when you have distributed coordination of multiple servers is that messages from one process to another process can be delayed, the clock synchronization of different servers can lead to incorrect decisions on when a certain message has arrived [29, p. 8].

2.5 RabbitMQ

RabbitMQ is a message broker that utilizes AMQP in an efficient and scalable way alongside other protocols. RabbitMQ is implemented in Erlang which uses the Actor-Model model. The Actor-Model is a conceptual model used for distributed computing and message passing. Every entity in the model which are actors receives a message and acts upon them. These actors are seperated from one another and do not share memory, furthermore one actor can not change the state of another actor in a direct manner [8, p. 9][30]. The above mentioned reason is a key feature to why RabbitMQ is scalable and robust because all actors are considered independent.

RabbitMQ is in comparison to Apache Kafka mainly centered around and built upon AMQP. RabbitMQ makes use of the properties from AMQP and makes further extensions to the protocol.

The extension of the routing capabilities that RabbitMQ implements for AMQP is the **Exchange-to-Exchange** binding which means that

you can bind one exchange to another in order to create a more complex message topology. Another binding is the **Alternate-Exchange** that works similarly to a wildcard matcher where there are no defined matching bindings or queues for certain types of messages. The last routing enhancement is the **Sender-selected** binding which mitigates the problem where AMQP can not specify a specific receiver for a message [31].

A fundamental difference between RabbitMQ and Apache Kafka is that RabbitMQ tries to keep all messages in-memory instead of persisting them to secondary memory such as a disk. In Apache Kafka the retention mechanism of keeping messages for a set of time is usually done by writing to disk with regards to the partitons of a topic. In RabbitMQ consumers will consume messages directly and relies on a ***prefetch-limit*** which can be seen as a counter for how many messages that has been unread and is an indicator for a consumer that is starting to lag. This is a limitation to the scalement with RabbitMQ because this prefetch limiter will cut of the consumer if it hits the threshhold, resulting in stacking of messages.

RabbitMQ offers **at-most-once** delivery and **at-least-once** delivery but never exactly once, in comparison to Kafka that offers **exactly-once** [32].

RabbitMQ is also focused on optimizing near-empty queues or empty queues, that is because as soon as an empty queue receives a message, the message goes directly to a consumer. In the case of non-empty queues the messages has to be enqueued and dequeued which in turn results in a slower overall message processing.

Chapter 3

Prior work

The area of testing message brokers has a wide range of published materials ranging from white papers to blog entries. The amount of information from which one can read and gain knowledge is therefore plenty but the most important aspect is how relevant it is to the work being conducted. In this thesis project the quantative and ambigous variable enqueueing performance is analyzed and evaluated.

Finding related work and too see how others have tested the message brokers resulted in one related paper conducted in September 2017 by Dobbelaera et. al [33]. This paper is examined in its essence in section 3.3.

3.1 Metrics

Enqueueing performance focused on in this thesis can be measured in two ways, **throughput** vs **latency**. The throughput in this case is measured with how many messages that can be sent. Latency on the other hand can be measured in two different ways, either as one-way (end-to-end) or the round-trip time. For this thesis only the one-way latency is measured. This was done to see the impact during the experiments when tuning the parameters for how long each consumer has to wait before processing an event or message.

Furthermore because these two message brokers have their own unique architecture and design goals when it comes to how to they send and store messages, two other secondary metrics to measure was chosen, the resource usage of the CPU and memory. The CPU metric measured how much of the CPU is allocated to system and user while the memory metric focused on how much data is actually written to disk.
These two metrics was chosen over others such as the protocol overhead created when sending messages because it was deemed unfeasible to keep statistics of each message being sent over the network link and because

RabbitMQ employs different protocols in comparison to Kafka which has its own binary protocol over TCP.

3.2 Parameters

Because of how Kafka and RabbitMQ are designed the parameter sets between them both are not identical, that is, there is no 1 to 1 relation where parameter X in Kafka is the same to parameter Y in RabbitMQ. Therefore an investigation of which parameters for both architecture was made. The findings showed that Kafka is left with more configuration parameters than for RabbitMQ and these are described below.

3.2.1 Kafka parameters

- **Batch size** - The batch size is where the producer tries to collect many records in one request when there are many messages being sent to same partition in the broker, instead of sending each message individually. This parameter enhances the performance for both the client and server.

- **Linger** - This parameter works together with the batch size because batching happens during loadtime when there is alot of messages that cannot be sent out faster than they come. When this happens one can add a configurable delay (in milliseconds) for when to send the records instead of trying to send each record as fast as possible. By adding this delay the batch size can fill up and which results in more messages being sent.

- **Acks** - This parameter has three different levels that can be set for the producer. Acks in this case is the number of acknowledgements which the producer is requesting from the leader to have from the replicas before recognizing a request as finished.

  The first level is where the producer does not have to wait for an acknowledgement from the leader. This means that a message is considered delivered after it has left the network socket even if there is no acknowledgement from server that it has actually received it.

  The second level is where the leader will have the message being written locally and thereafter responding to the producer without

waiting for any acknowledgements from the other replicas. This can lead to messages being lost if the leader goes down before the replicas has written it to their local log.

The third level is where the leader will not send an acknowledgement to the producer before receiving all acknowledgements from the replicas. This means that the record can not be lost unless all of the replicas and the leader goes down.

- **Compression** - The compression parameter is used for the data being sent by the producer, Kafka supports three different compression algorithms, **snappy** and **gzip** and **lz4**.
- **Log flush interval messages** - This parameter decides how many messages in a partition should be kept before flushing it to the harddrive.
- **Partitions** - The partitions is how Kafka parallelize the workload in the broker.

3.2.2 RabbitMQ parameters

- **Queues** - This parameter can be configured to tell the broker how many queues should be used.
- **Lazy queue** - The lazy queue parameter changes the way RabbitMQ stores the messages, RabbitMQ will try to deload the RAM usage and instead store the messages to disk when it is possible.
- **Queue length limit** - This parameter can be tuned to either keep track of how many messages there are in the queue or how many bytes are stored.
- **Direct exchange** - The direct exchange parameter is used to declare that the queues being used should be have each message directly sent to its specific queue.
- **Fanout exchange** The fan-out exchange ignores the routing to specific queues and will instead broadcast it to every available queue.
- **Auto-ack** - Auto acknowledgement is used for when the message is considered to be sent directly after leaving the network socket, in similar fashion to the first level of Kafkas acknowledgement.

- **Manual ack** - A manual acknowledgement is when the client sends a response that it has received the message.
- **Persistent** - The persistent flag is used to write messages directly to disk when it enters the queue.

These parameters resulted in working with two different types of subsets which led to the conclusion that RabbitMQ and Kafka can not be compared on equal basis. This made the focus of the thesis work changed to see what degree the different parameters can achieve when it comes to throughput and latency.

3.3 Related work

Testing of RabbitMQ against Kafka has not been done before except for what was reported in [8] according to my findings. This report focused on two different deliverance semantic the **at-least-once** and **at-most-once** and was tested on RabbitMQ version 3.5 and Kafka version 0.10. Note that version 0.11 and higher for Kafka offers exactly-once and was planned initially in this thesis to have experiments conducted on, but without a counterpart found in RabbitMQ it was left out for testing. This report has its experiments conducted on Kafka version 1.1 and RabbitMQ version 3.7.3.

The experiments found in [8] was conducted on a single bare-metal server and not on a cloud platform which is what this report did and therefore making it more distributed.

From [8, p.13] they state that they only used the default configuration for their experiments which is in opposite to this thesis which tests different parameters and their impact on throughput and latency when running in the two different semantics. The report in question notes that three important factors for measuring throughput is the record size, partition and topic count, albeit true one can read from [34] that there are several other parameters that can factor in the throughput which the authors of [8] does not make use of, which is mainly the *batch size*, *lingerms*, *compression algorithm*, *acks*, and *buffer memory*. All of these are tested in this thesis except for the buffer memory which is used in combination when you have many partitions.

For measuring CPU and memory usage two different programs were used,

mpstat and **dstat**. The mpstat command displays many different types of information such as I/O utilization, hardware interrupts, software interrupts etcetera, but only two were considered the first was the CPU utilization of the system that occurs on kernel level. The other one was the user level and it shows how much of the CPU is utilized by applications. Dstat was used over other tools such as **vmstat** or **iostat** because of the layout of the output being more understandable as it tracks data in different columns.

Chapter 4

Experiment design

The work is divided in to four sections, section 4.1 presents a testing tool that was developed but was scraped in favour of existing tools from their respective distributions found in RabbitMQ and Kafka. Section 4.2 is where a suitable cloud platform was decided upon and the instances chosen on that platform and the reasons why these instances was chosen and it presents the setup of the testing suite. Section 4.3 presents all the experiments that was made for both Kafka and RabbitMQ. Section 4.4 shows the steps from setting up the servers to conducting the experiments to importing them for data processing.

4.1 Initial process

The testing phase started with a hasty decision of developing a tool for testing out Kafka on their platform. This tool was developed with the purpose of sending arbitrary messages to the broker and measuring the throughput and latency.

The tool that was created lacked some flexible features such as easily configuring the appropriate parameters of Kafka or adding measurement of multiple servers. In order to test a specific parameter one had to manually change the source code. Furthermore the logging of the CPU usage and memory usage showed unusual findings where the CPU usage was between 90-100%. The experiments conducted ran from a localhost to the cloud platform which could skewer the results since the distance to the cloud platform is longer.

With this in mind a more pragmatic solution was taken where tools provided by Kafka and RabbitMQ was used as they where inherently more flexibile in way where it was possible to connect to multiple servers, track the latency with better statistical measurements and offering more configuration parameters to choose from.

4.2 Setup

One of the key components of this thesis work was choosing a suitable cloud platform to conduct the experiments on, the chosen platform was Amazon Web Service.

AWS offers a varity of instances on their platform with focus on different test cases. These are comprised to a general purpose, compute optimization, memory optimization, accerelerated computing or storage optimized [35]. These categories has their own subset of hardware and the first choice to make was to pick which category to create an instance on. Instances on the general purpose section makes use of Burstable Performance Instances which means that if there is a CPU throttle on the host it can momentarily provide additional resources to it. The hardware specifications of these instances can be seen in Table 4.1.

Table 4.1: Instances on general purpose

Model	**vCPU**	**Memory (GiB)**	**Harddrive storage**
t2.nano	1	0.5	EBS-only
t2.micro	1	1	EBS-only
t2.small	**1**	**2**	**EBS-only**
t2.medium	2	4	EBS-only
t2.large	2	8	EBS-only
t2.xlarge	4	16	EBS-only
t2.2xlarge	8	32	EBS-only

EBS-only means that the storage is located on the Elastic Block Store which is a block-level storage that is orchestrated in a way where it replicates a physical storage drive which in comparison to an object store that stores data within a data-object model [36].

The instance chosen to conduct the experiments on was the t2.small because optimizing for throughput and latency for low (cheaper) hardware specifications from an economical perspective is more sustainable for users.

Four instances each of these where created for Kafka and RabbitMQ in total eight instances. Three of each instance where to be tested against and the fourth one acted as a testrunner in order to minimize external factors affecting the results as to testing it locally which was done with

the self-made tool. The setup for Kafka and RabbitMQ can be seen in figure 4.1.

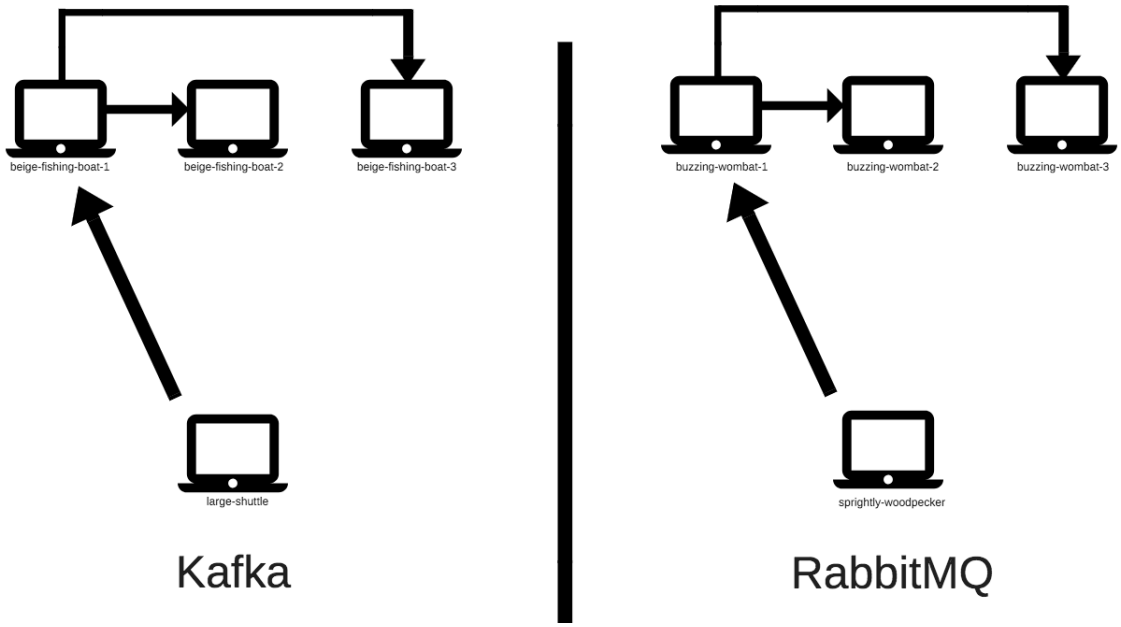

Figure 4.1: Test setup

The instance beige-fishing-boat-1 is the main broker and beige-fishing-boat-2 and 3 are replicas. For RabbitMQ buzzing-wombat-1 is the main server and buzzing-wombat-2 and 3 are replicas.

4.3 Experiments

The experiments that were conducted on RabbitMQ and Kafka is presented in section 4.3.1 and 4.3.2. The messages size was set to 500 bytes.

4.3.1 Kafka

Experiment one

The first experiment for Kafka was to configure the batch size and then trying it out with a number of partitions. The batch size was set to an interval from 10 000 to 200 000 with a step size of 10 000. The number of partitions were set to start with 5 and then iteratively being changed to 15, 30 and lastly 50. This experiment ran with a snappy compression.

The number of acknowledgements was set to level one for one set of experiments and level three for the other set.

Experiment two

The second experiment tested the linger parameter in combination with batch size. The batch size was set to 10 000 to 200 000 with a step size of 10 000. The linger parameter had an interval between 10 to 100 with a step size of 10. For each batch size the linger parameter was tested, that is, for batch size 10 000 the linger parameter interval tested was 10 to 100, for batch size 20 000 the linger parameter set was 10 to 100 and so forth. The compression for this experiment was snappy. Because of this extensive testing only 2 partitions was used, 5 and 15. The acknowledgement level was set to one for one set and level three for the other set.

Experiment three

The third experiment focused on the impact of compressing the messages, this experiment tested firstly a batch size interval of 10 000 to 200 000 with acknowledgement level two and snappy compression. The second experiment tested the same but without any compression at all.

Experiment four

The fourth experiment tested flush interval message parameter in order to see how it affected the disk usage. This experiment was conduected with two different partions, 5 and 15. The flush interval message parameter was tested in an interval as such 1, 1000, 5000, 10 000, 20 000, 30 000. The compression used was snappy.

4.3.2 RabbitMQ

Experiment one

This experiment tested the fanout exchange, with queues from 1 to 50. This test was conducted firstly with manual acknowledgement and thereafter auto acks. This was tested with persistent mode.

Experiment two

This experiment tested lazy queues with queues from 1 to 50, this was conducted with manual and auto acknowledgements. This was tested

with persistent mode.

Experiment three

The third experiments tested five different queue lengths ranging from the default length to 10, 100, 1000 and 10 000. This was tested with manual and auto acknowledgements and with the persistent mode.

4.4 Pipelining

The experiments was deemed unfeasible to conduct by manually changing the parameters for each experiment, this issue was solved by developing a script that can run each experiment automatically and log the findings.

This script works as such; it will connect to beige-fishing-boat or buzzing-wombat and start logging with dstat and mpstat to two text-files, after this, it connects to either the large-shuttle server or sprightly-woodpecker depending on which message broker to be tested and start sending messages to the message brokers and logging it to a text-file. After this step another connection to the beige-fishing-boat/buzzing-wombat needed to be done in order to terminate the processess that is logging with mpstat and dstat in order to not stack multiple process of the same function which could affect the result of the CPU and memory management.

With all of these tests being conducted a large amount of text-files was being generated and it needed to be processed in an efficient way. Therefore a parser was developed that could take in the data from these text-files and generate an Excel™sheet instead of manually filling each value from the text-file. This parser was put in front of the brokers in order for it to work, the whole architecture of the script and the servers and the parser can be seen in figure 4.2 with the appropriate steps on the whole run.

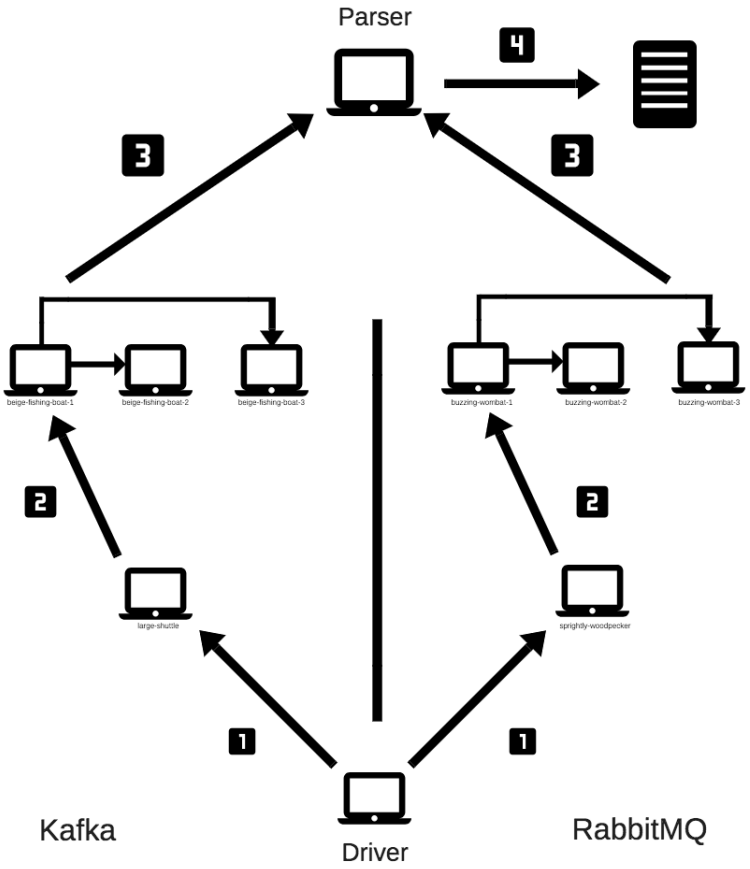

Figure 4.2: Architecture of system

Chapter 5

Result

A subset of results are presented in section 5.1 and 5.2, these results are chosen out of many because they show the most apparent difference between the experiments. Other results are found in the appropriate appendix.

5.1 Kafka

5.1.1 Experiment one

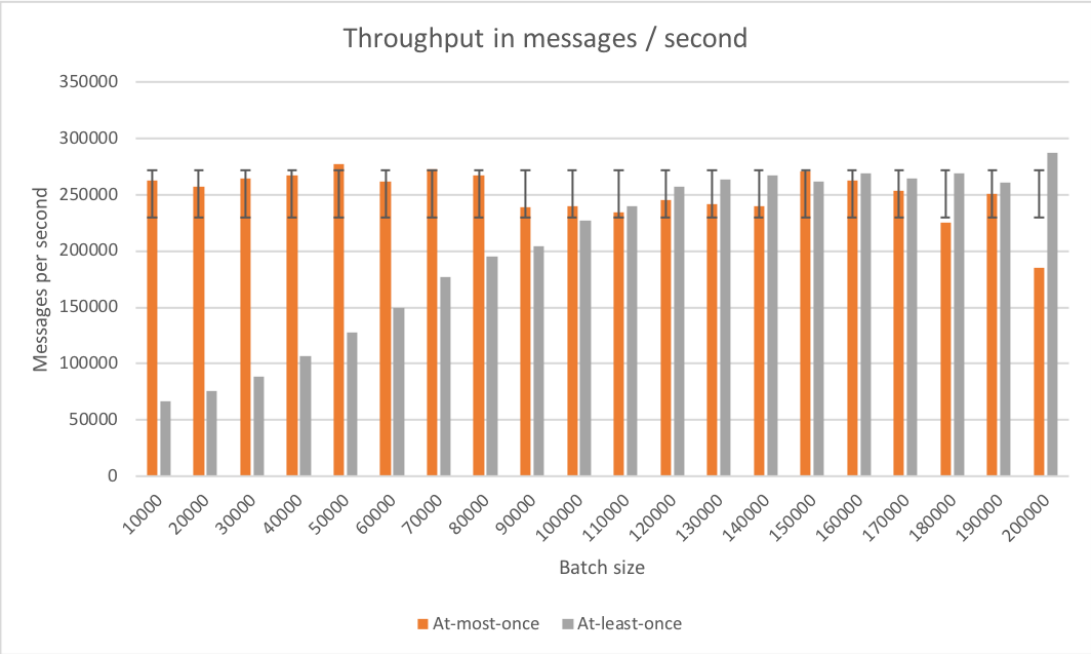

Figure 5.1: Throughput measured with 5 partitions

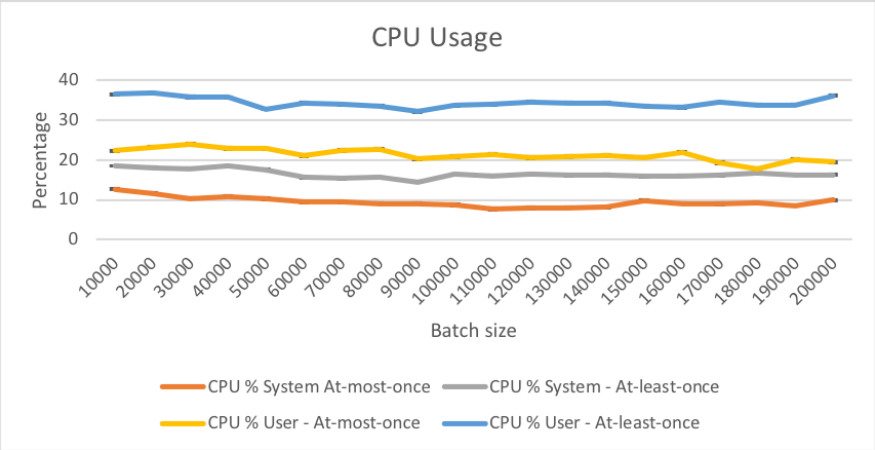

Figure 5.2: CPU Usage with 5 partitions

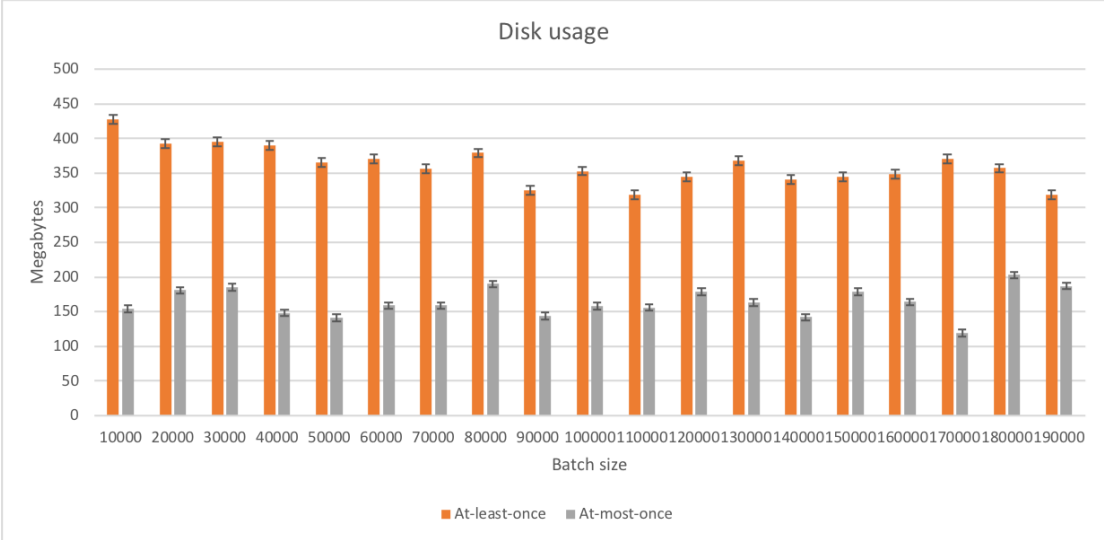

Figure 5.3: Diskusage with 5 partitions

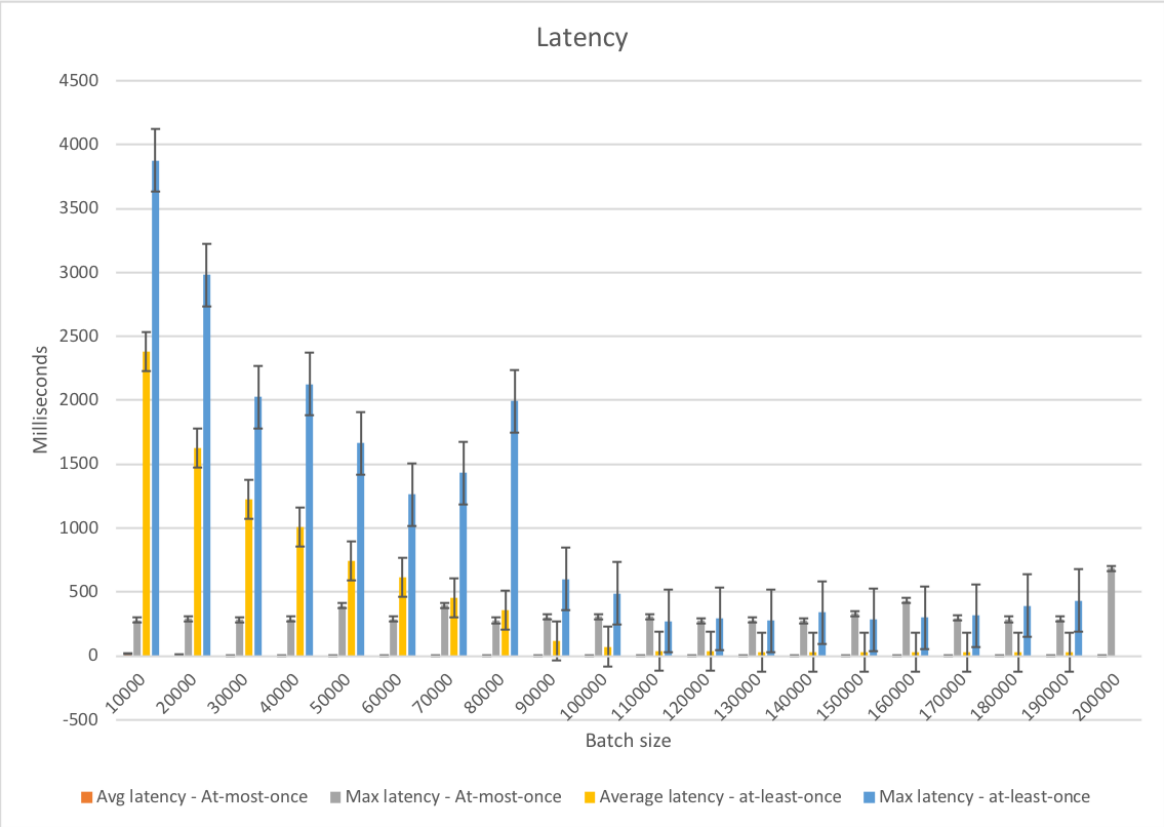

Figure 5.4: Latency measured with 5 partitions

This first experiment shows that with deliverence setting at-most-once, Kafka reaches 250 000 msg/s with a batch size of 10 000 and stays that way regardless of batch size. With at-least-once the throughput has a linear increasement for every increasing batch size, with batch size set to 10 000 the throughput is over 50 000 msg/s. The throughput increases for every change of batch size until it hits 120 000 where it stabilizes around 250 000 msg/s. One anomaly seen is for batch size 200 000, at-least-once performs better than at-most-once but the error margin is the same for the rest of the stabilized results so this can be dismissed. Kafkas CPU usage shows that it is utilized more for the application rather than for system calls.

With at-least-once mode Kafka writes between 300 to 350 megabytes and with at-most-once it only writes a maximum of 200 megabytes. The latency observed for the first experiment shows that the maximum latency for Kafka hits around 4 seconds with at-least-once mode while the max latency for at-most-once is around 0.5 seconds. The average latency between at-most-once and at-least-once is large with the former around

0-15 milliseconds and the latter starting with 2.5 seconds and stabilizes around 200 milliseconds.

5.1.2 Experiment two

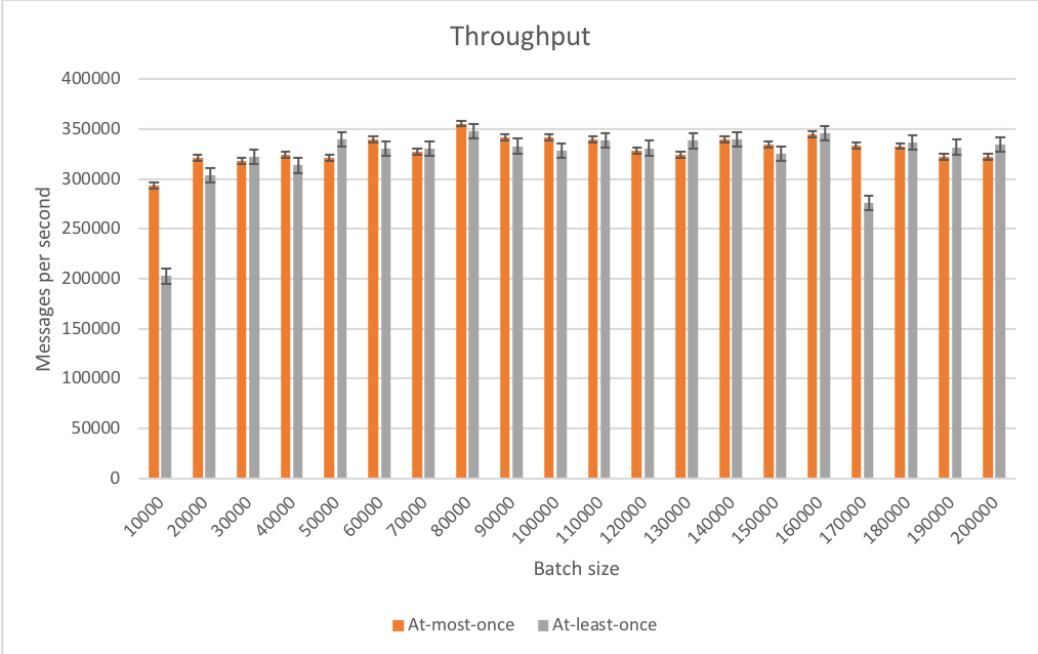

Figure 5.5: Throughput with linger set to 10

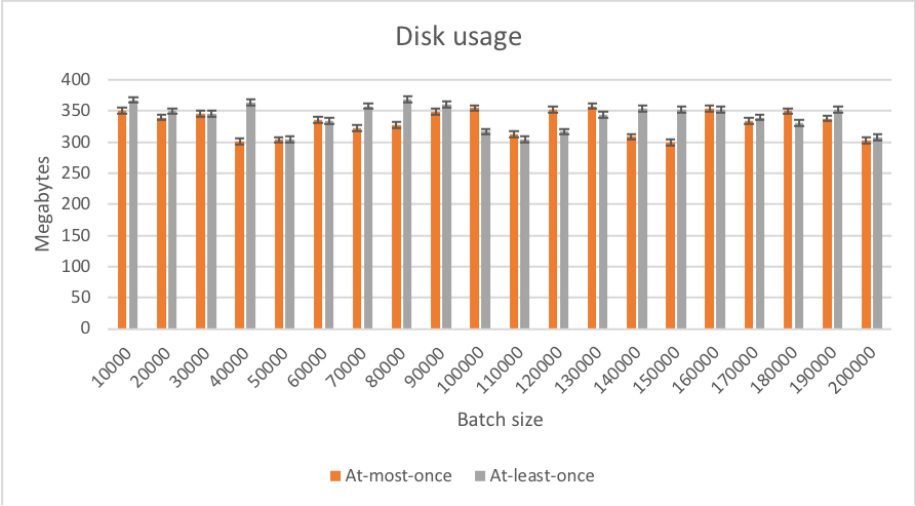

Figure 5.6: Disk usage with linger set to 10

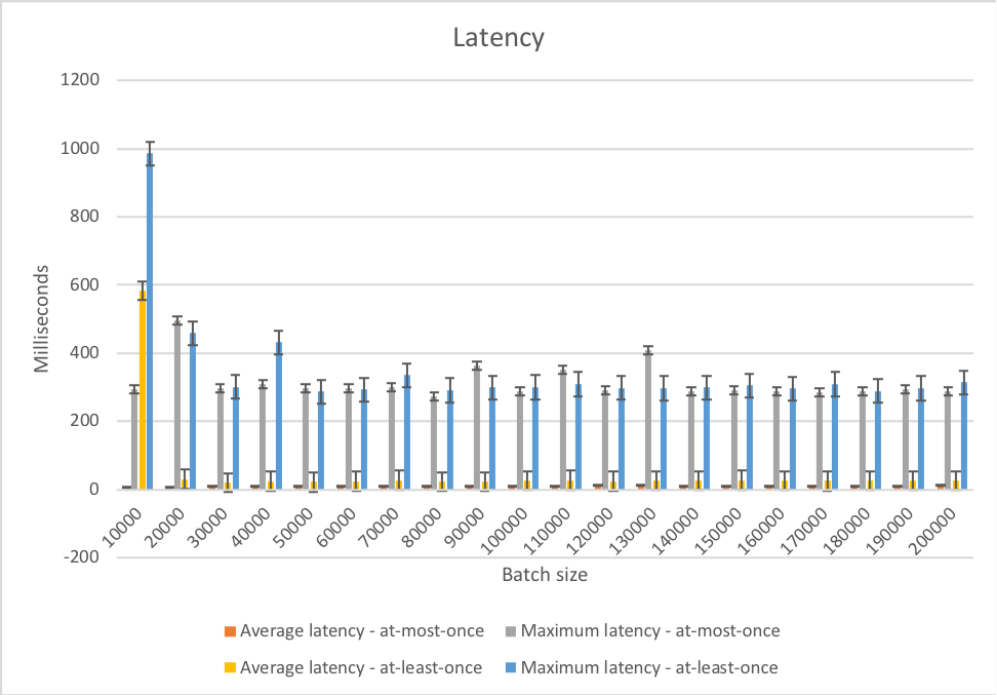

Figure 5.7: Latency with linger set as 10

With the addition of the linger parameter, the throughput increased from 250 000 msg/s to 350 000 msg/s in both deliverance modes. This points to Kafka waiting longer for a batch to reach it maximum size in comparison to only use the batch size parameter. The maximum latency for at-least once mode was 1 second, because the linger parameter was set to 10 milliseconds so Kafka only has to wait 10 milliseconds before sending off a batch.

5.1.3 Experiment three

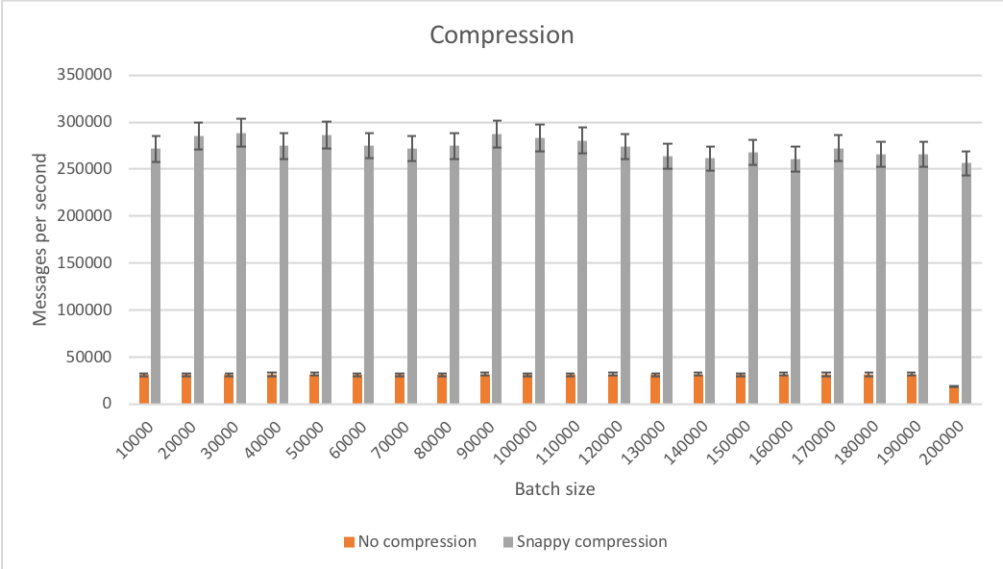

Figure 5.8: Throughput measurements with snappy compression and none compression

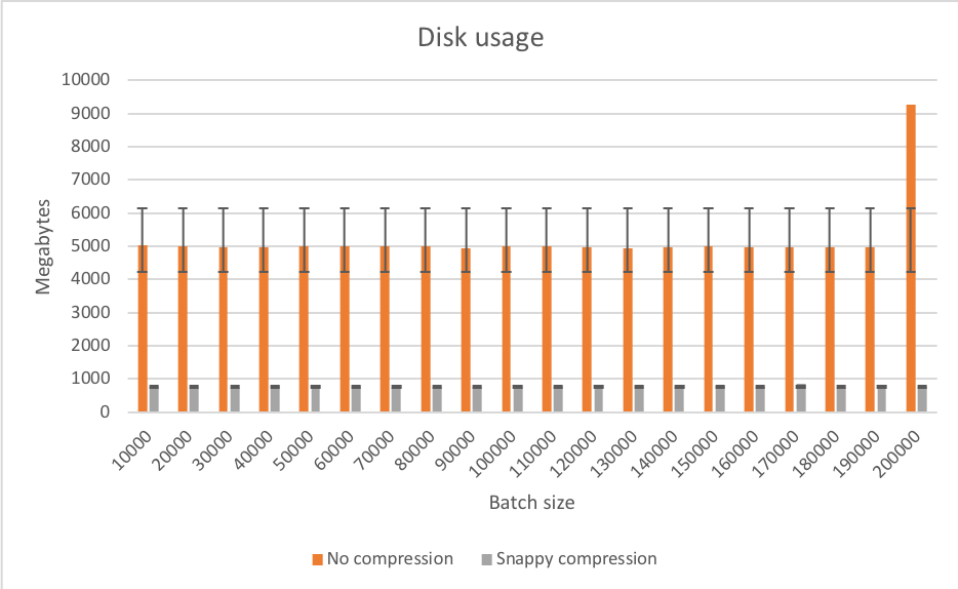

Figure 5.9: Disk usage with compression and none compression

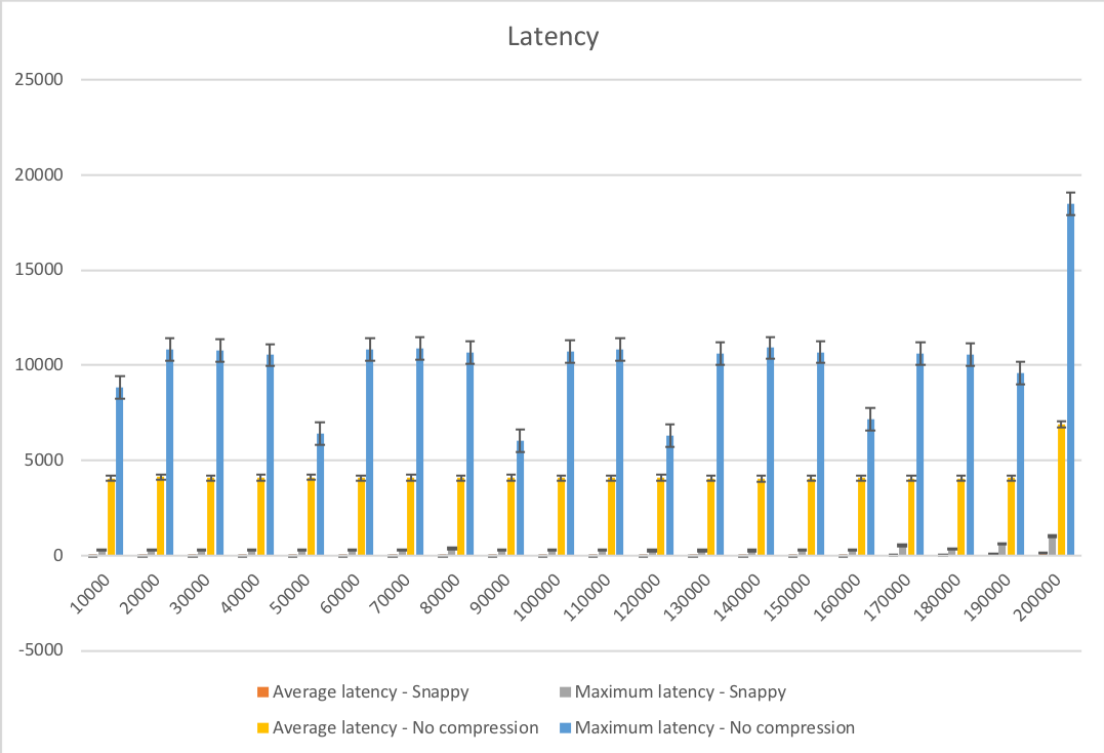

Figure 5.10: Latency measurements with compression and none compression

When not using compression the expected throughput would be decreased, the throughput reaches below 50 000 msg/s. With the compression Kafka reaches around 250 000 msg/s. The disk usage was expected to be higher because of sending non-compressed messages, and it shows that Kafka writes around 5Gb of data when not using compression. There is a spike of almost 9Gb being written when using batch size of 200 000 but error margin is within the earlier results.

5.1.4 Experiment four

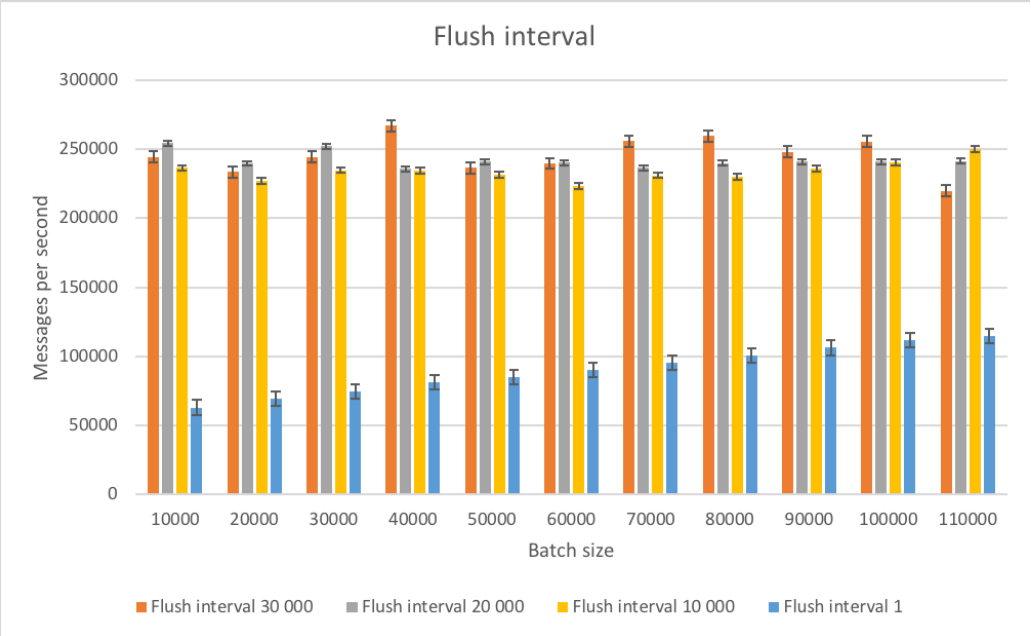

Figure 5.11: Throughput with configured flush interval

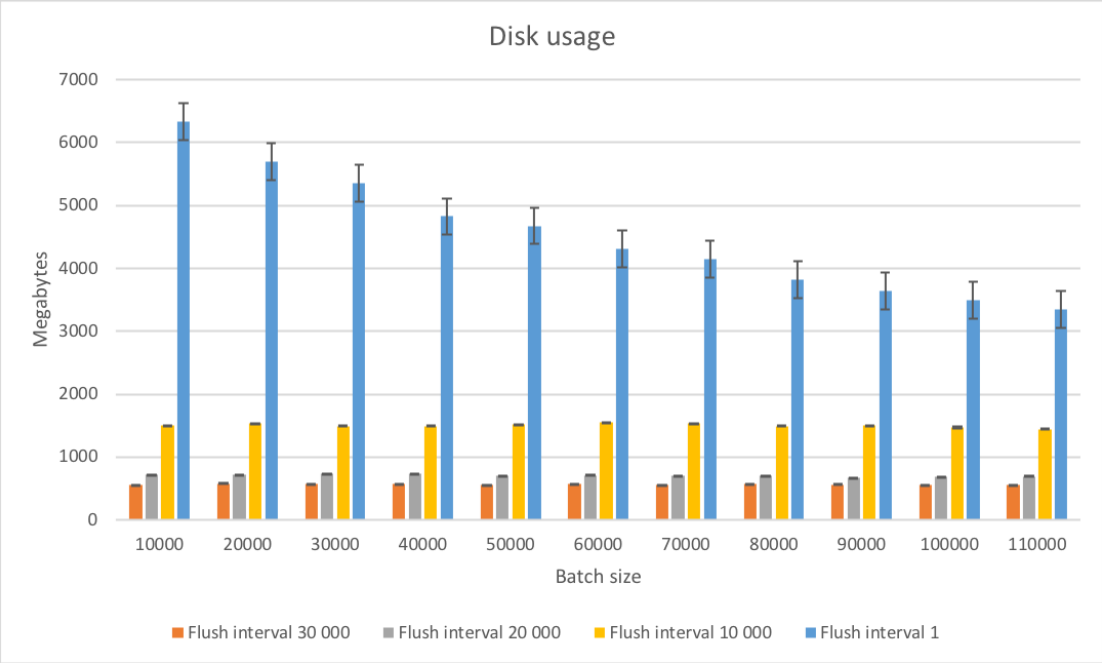

Figure 5.12: Disk usage with configured flush interval

When flushing each message to disk the throughput gets affected which was expected because it takes longer time to get a response. If you want to be sure that each message is stored to disk in favour for higher throughput the flush interval parameter must be used. For higher flush interval values the throughput does not get affected.

5.2 RabbitMQ

5.2.1 Experiment one

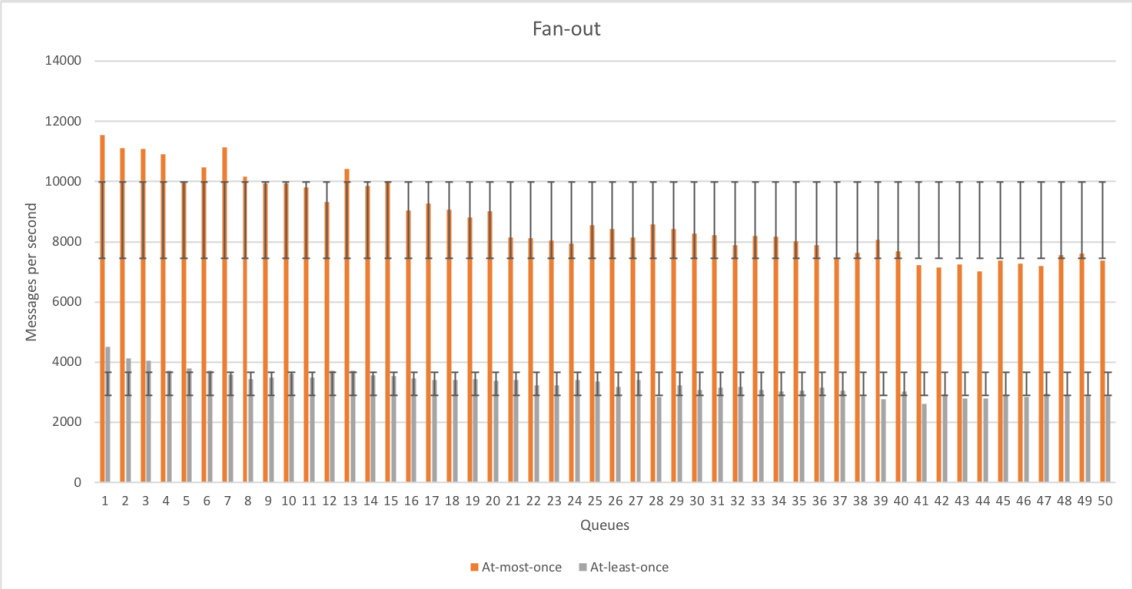

Figure 5.13: Throughput with fanout exchange

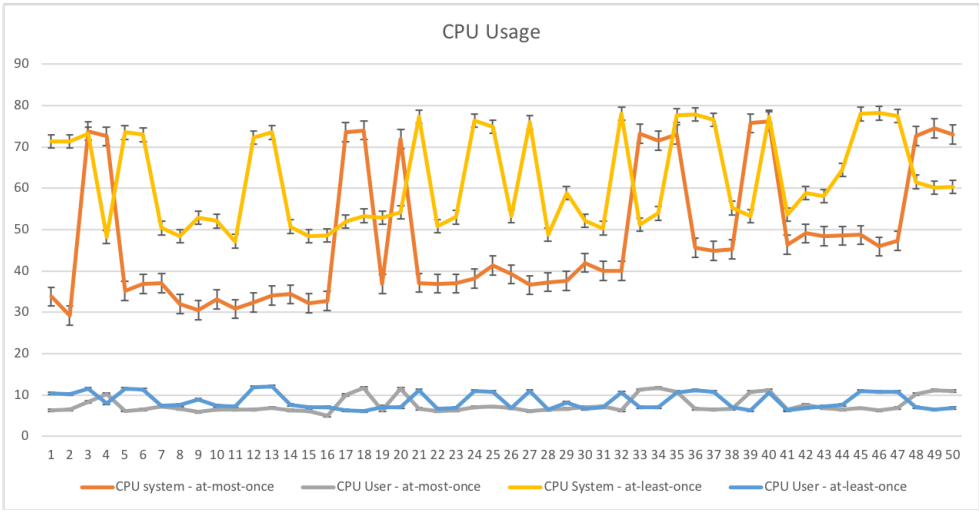

Figure 5.14: CPU usage with fanout exchange

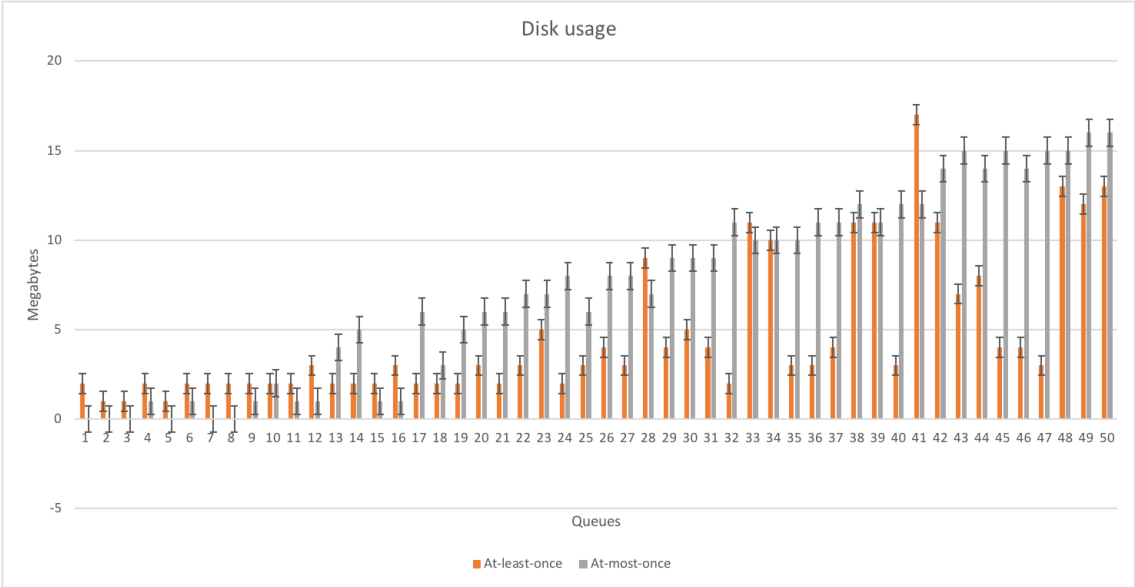

Figure 5.15: Disk usage with fanout exchange

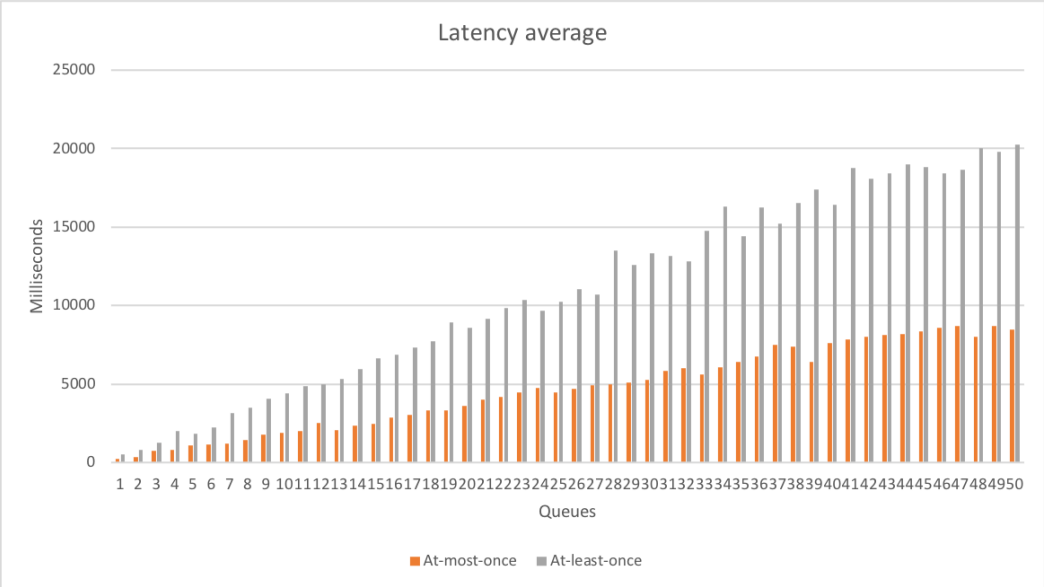

Figure 5.16: Latency measured with fanout exchange

The observed results for the first experiment for RabbitMQ shows the throughput is around 12 000 msg/s for at-most-once mode, and around 4000 msg/s for at-least-once. The amount of queues affects throughput for at-most-once mode but not at-least-once. The system usage for RabbitMQ is higher than the user usage. The anomalies found are discussed in section 5.3. The latency for at-most-once mode is increased for each queue to higher than 5 seconds but lower than 10, but at-least-once increase for each queue being created with a maximum of 20 seconds.

5.2.2 Experiment two & three

The results from experiment two and three shows similar results in both at-least-once and at-most-once mode to what is presented in experiment one and can be seen in the appendix section **??** as rawdata. This is discussed in section 5.3 on to why that is.

5.3 Discussion

When it comes to the result of the experiments one has to start with Kafka. This broker performs much better than RabbitMQ according to the tests when it comes to throughput and latency. The amount of messages being sent with Kafka is over 5 times more than with RabbitMQ. This can be seen in comparison to figure 5.13 and figure 5.1 with RabbitMQ reaching 12000 messages and Kafka reaching over 250 000.

Another interesting point to see is that impact of applying compression with Kafka, with compression Kafka reaches over 250 000 messages per second but without it, it only goes up to approximately 50 000 which can be seen in figure 5.8. Furthermore the disk usage when applying compression and non-compression is staggering, with compression it only writes 1Gb to the disk and without its almost 5 times more which can be seen in figure 5.9.

The linger parameter for Kafka combined with batch size gave the throughput an extra boost in comparison to not having it configured, by almost 50 000 more messages than by just using the batch size parameter, this can be seen in 5.7 which has over 300 000 messages being sent for the majority of batches.

The flush interval parameter for Kafka hampers the throughput when Kafka is forced to flush every message to the disk and it only reaches approximately 50 000 msg/s. With compression it still writes more to the disk (between 7Gb to 4Gb depending on what batch size is configured) in comparison to not applying compression which can be seen in figure 5.12.

Now as to why the experiments for RabbitMQ performed so poorly in comparison to Kafka is that the experiments conducted with many different queues only were configured to having 1 consumer/producer using them which is the default mode of the Kafka tool.

Another explanation to why Kafka outperforms RabbitMQ is that Kafka is optimized for stream-based data and does not offer the same configuring options as RabbitMQ with the potential workings of an exchange and binding messages with keys. RabbitMQ is more versatile when you want to configure specific messages to go to certain consumers.

The CPU utilization for Kafka and RabbitMQ are notably different which one can see from figure 5.14 and 5.2. The CPU utilization for RabbitMQ with the at-least-once mode staggers between 30-40% and spikes up to 70% for the system. A possible explanation to this behaviour is that the results from the queues are not automatically deleted which makes the CPU usage of the system go up.

A very notable different between Kafka and RabbitMQ is that Kafka makes much more use of the disk in terms of how much data is written in comparison to RabbitMQ. From figure 5.15 one can see that RabbitMQ writes maximum of around 20 megabytes in comparison to Kafka which in some experiments showed over 5 Gb of data being written.

Chapter 6

Conclusion

A number of parameters for Kafka and RabbitMQ has been tested in this thesis in two different delivery modes, at-least-once and at-most-once. This thesis shows the impact of different parameters on throughput and latency as well as the amount of data written and the usage of CPU for both Kafka and RabbitMQ. Interested parties that wants to know how Kafka compares to RabbitMQ can see the results of two deliverance semantic to better see the tradeoff of durability vs throughput i.e if it is important to store messages for lower throughput and higher latency. The goals for this thesis was to design experiments for Kafka and RabbitMQ which was done by analyzing different relevant parameters of both Kafka and RabbitMQ albeit not being a 1:1 parameter set between them. For each experiment made an appropriate error margin was delivered to see how accurate the results were which was one of the goals. The results from the different experiments were compared to each other as best as possible.

Kafka outperforms RabbitMQ when factoring both latency and throughput, RabbitMQ offers more advanced configurations to route messages with the help of exchange-to-exchange.

Possible future work could be to test it on other hardware configurations and try to set up a larger cluster of nodes. It would also be interesting to see the performance impact when running Kafka in exactly-once mode.

Bibliography

[1] EMC Digital Universe with Research Analysis by IDC. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*. 2014. URL: https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[2] Brendan Burns. *Designing Distributed System. Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly, 2017.

[3] Dotan Nahum Emrah Ayanoglu Yusuf Aytaş. *Mastering RabbitMQ. Master the art of developing message-based applications with RabbitMQ*. Packt Publishing Ltd, 2015.

[4] Gregor Hohpe. *Enterprise integration patterns : designing, building and deploying messaging solutions*. eng. The Addison-Wesley signature series. Boston: Addison-Wesley, 2004. ISBN: 0-321-20068-3.

[5] "Message-Oriented Middleware". In: *Middleware for Communications*. John Wiley Sons, Ltd, 2005, pp. 1–28. ISBN: 9780470862087. DOI: 10.1002/0470862084.ch1. URL: http://dx.doi.org/10.1002/0470862084.ch1.

[6] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[7] Vasileios Karagiannis et al. "A Survey on Application Layer Protocols for the Internet of Things". In: *Transaction on IoT and Cloud Computing* (2015). URL: https://pdfs.semanticscholar.org/ca6c/da8049b037a4a05d27d5be979767a5b802bd.pdf.

[8] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 227–238. ISBN: 978-1-4503-5065-5. DOI: 10.1145/3093742.3093908. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/3093742.3093908.

[9] Pieter Humphrey. *Understanding When to use RabbitMQ or Apache Kafka*. 2017. URL: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka (visited on 03/09/2018).

[10] Farshad Kooti et al. "Portrait of an Online Shopper: Understanding and Predicting Consumer Behavior". In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. WSDM '16. San Francisco, California, USA: ACM, 2016, pp. 205–214. ISBN: 978-1-4503-3716-8. DOI: 10.1145/2835776.2835831. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/2835776.2835831.

[11] Ryuichi Yamamoto. "Large-scale Health Information Database and Privacy Protection." In: *Japan Medical Association journal : JMAJ* 59.2-3 (Sept. 2016), pp. 91–109. ISSN: 1346-8650. URL: http://www.ncbi.nlm.nih.gov/pubmed/28299244%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5333617.

[12] *Key Changes with the General Data Protection Regulation*. URL: https://www.eugdpr.org/key-changes.html (visited on 03/19/2018).

[13] Bill Weihl et al. "Sustainable Data Centers". In: *XRDS* 17.4 (June 2011), pp. 8–12. ISSN: 1528-4972. DOI: 10.1145/1961678.1961679. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1961678.1961679.

[14] Anne Håkansson. "Portal of Research Methods and Methodologies for Research Projects and Degree Projects". In: *Computer Engineering, and Applied Computing WORLDCOMP* (2013), pp. 22–25. URL: http://www.diva-portal.org%20http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960.

[15] Jakub Korab. *Understanding Message Brokers*. ISBN: 9781491981535.

[16] Mary Cochran. *Persistence vs. Durability in Messaging. Do you know the difference? - RHD Blog*. 2016. URL: https://developers.redhat.com/blog/2016/08/10/persistence-vs-durability-in-messaging/ (visited on 03/23/2018).

[17] PTh Eugster et al. "The Many Faces of Publish/Subscribe". In: (). URL: http://members.unine.ch/pascal.felber/publications/CS-03.pdf.

[18] Michele Albano et al. "Message-oriented middleware for smart grids". In: (2014). DOI: 10.1016/j.csi.2014.08.002. URL: https://ac-els-cdn-com.focus.lib.kth.se/S0920548914000804/1-s2.0-S0920548914000804-main.pdf?%7B%5C_%7Dtid=76c79d76-8189-4398-8dc8-dda1e2a927e7%7B%5C&%7Dacdnat=1522229547%7B%5C_%7D.

[19] Joshua Kramer. *Advanced Message Queuing Protocol (AMQP)*. URL: http://delivery.acm.org.focus.lib.kth.se/10.1145/1660000/1653250/10379.html?ip=130.237.29.138%7B%5C&%7Did=1653250%7B%5C&%7Dacc=ACTIVE%20SERVICE%7B%5C&%7Dkey=74F7687761D7AE37.E53E9A92DC589BF3.4D4702B0C3E38B35.4D4702B0C3E38B35%7B%5C&%7D%7B%5C_%7D%7B%5C_%7Dacm%7B%5C_%7D%7B%5C_%7D=1522311187%7B%5C_%7D (visited on 03/29/2018).

[20] Piper Andy. *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP - VMware vFabric Blog - VMware Blogs*. 2013. URL: https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html (visited on 03/29/2018).

[21] Carl Trieloff et al. "AMQP Advanced Message Queuing Protocol Protocol Specification License". In: (). URL: https://www.rabbitmq.com/resources/specs/amqp0-8.pdf.

[22] Emrah Ayanoglu, Yusuf Aytas, and Dotan Nahum. *Mastering RabbitMQ*. 2015, pp. 1–262. ISBN: 9781783981526.

[23] P Saint-Andre, K Smith, and R Tronçon. *XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies*. 2009, p. 310. ISBN: 9780596521264.

[24] *Stomp specifications*. URL: http://stomp.github.io/stomp-specification-1.1.html%7B%5C#%7DDesign%7B%5C_%7DPhilosophy (visited on 04/03/2018).

[25] "Piper,Diaz". In: (2011). URL: https://www.ibm.com/podcasts/software/websphere/connectivity/piper%7B%5C_%7Ddiaz%7B%5C_%7Dnipper%7B%5C_%7Dmq%7B%5C_%7Dtt%7B%5C_%7D11182011.pdf.

[26] Margaret Rouse. *What is MQTT (MQ Telemetry Transport)? - Definition from WhatIs.com*. 2018. URL: https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport (visited on 04/03/2018).

[27] Todd Ouska. *Transport-level security tradeoffs using MQTT - IoT Design*. 2016. URL: http://iotdesign.embedded-computing.com/guest-blogs/transport-level-security-tradeoffs-using-mqtt/ (visited on 04/03/2018).

[28] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide : Real-time Data and Stream Processing at Scale*. O'Reilly Media, 2017. ISBN: 9781491936160. URL: https://books.google.se/books?id=qIjQjgEACAAJ.

[29] Flavio Junquera and Benjamin Reed. *Zookeeper - Distributed Process Coordination*. 2013, p. 238. URL: https://t.hao0.me/files/zookeeper.pdf.

[30] Brian Storti. *The actor model in 10 minutes*. 2015. URL: https://www.brianstorti.com/the-actor-model/ (visited on 04/08/2018).

[31] *RabbitMQ - Protocol Extensions*. URL: http://www.rabbitmq.com/extensions.html (visited on 04/08/2018).

[32] Neha Narkhede. *Exactly-once Semantics is Possible: Here's How Apache Kafka Does it*. 2017. URL: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ (visited on 04/08/2018).

[33] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "Kafka versus RabbitMQ". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17* August (2017), pp. 227–238. DOI: 10.1145/3093742.3093908. arXiv: arXiv:1709.00333v1. URL: http://dl.acm.org/citation.cfm?doid=3093742.3093908.

[34] Confluence.io. "Optimizing Your Apache Kafka TM Deployment, Levers for Throughput, Latency, Durability, and Availability". In: (), p. 2017.

[35] *Amazon EC2 Instance Types – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 06/17/2018).

[36] *Amazon EC2 Instance Types – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 06/17/2018).

Chapter A

Appendix

Results can be found here,