EXAMENSARBETE INOM INFORMATIONSTEKNIK,
AVANCERAD NIVÅ, 30 HP

STOCKHOLM, *SVERIGE 2018*

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

AMIR RABIEE

KTH
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

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 and small enterprises need a communication method in order to meet requirements of scalability and durability. Many communication methods exist, but the most well-used are message queues and message brokers. The problem is that it exist many different types of message queues and message brokers with their own unique design and implementation choices. These choices result in different parameter sets, which can be configured in order to meet requirements of for example high durability, throughput, and availability.

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

The results show that Kafka outshines RabbitMQ regarding throughput and latency. RabbitMQ is the most efficient in terms of quantity of data being written, while on the other hand being more CPU-heavy than Kafka. Kafka performs better than RabbitMQ in terms of the amount of messages being sent and having the shortest one-way latency.

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. Många kommunikationsmetoder existerar, men de mest använda är meddelandeköer och meddelandemäklare. Problemet är att det finns en uppsjö av olika typer av meddelandeköer och meddelandemäklare som är unika med avseende på deras design och implementering. Dessa val resulterar i olika parametersatser som kan konfigureras för att passa olika kriterier, exempelvis hög hållbarhet, genomströmning och 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 som exempelvis 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. Gällande inverkan av Kafka och RabbitMQ på mängden skriven data, är RabbitMQ den mest effektiva, medan den å andra sidan är mer CPU-tung än Kafka.

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

Contents

Chapter 1

Introduction

The data being generated and transferred from applications are constantly evolving in a fast paced manner. The amount of data generated by applications are estimated for year 2020 to hit over 40 trillion gigabytes [1].

The applications which handle these data have requirements that need to be met, such as having high reliability and high availability. The solution for meeting these requirements is to build a distributed system [2, p. 1].

A distributed system is built on relatively cheap hardware commodity, that one has been able to utilize to build a network of computers communicating together for a common purpose and goal. This setup, of having multiple machines coordinating together for a common task, have innately advantages in comparison to the client-server architecture. A distributed system is more scalable, reliable and faster when designed correctly in comparison to the client-server architecture. However, the advantages with these systems come with a cost, as designing, building and debugging distributed systems are more complicated [2, p. 2].

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 to the correct machine [4, p. 326].

1.1 Background

Distributed systems are developed to meet requirements of high availability and reliability. They are usually built upon some abstract message form being sent from one process located on one machine to another process on a different machine. The message handling is often done by using

an external architecture or system that can distribute messages.

The architecture often used, 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]. Furthermore a MOM 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.

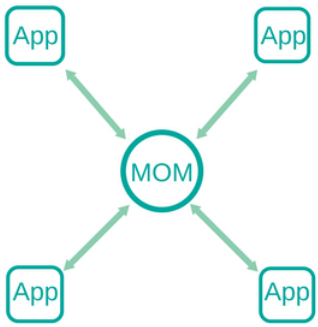

Figure 1.1: Structure of MOM. The applications communicate with the MOM to send message from one application to another.

A MOM makes use of **message queues**, which 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 includes the name, size and sorting algorithm for the queue [5, p. 7].

In order to efficiently distribute the messages to different queues one has to use 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 utilization of message queue protocols, of which there are many different types, such as: **Streaming Text Oriented Messaging Protocol** (STOMP), **Extensible Mes-**

saging and Presence Protocol (XMPP), **Message Queuing Telemetry Transport** (MQTT) and **Advanced Message Queuing Protocol** (AMQP) and more [7, p.3].

The unique capabilities of MOM come from the messaging models that it utilizes, mainly two to be found, **point-to-point** and **publish/subscribe**.

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

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

1.2 Problem

Many different message brokers exist, choosing one is a multifaceted question with many different aspects needing to be taken into 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.

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 found in Table 1.1 are widely used and can be deployed on different server architectures and platforms [8].

The message brokers Apache Kafka and RabbitMQ, need to be tested on their enqueuing performance with focus on different parameter sets to obtain a deeper understanding on the impact of latency and throughput with the parameter sets.

With this background, 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

This thesis shows and discusses an overview of the available messaging middleware solutions and aims to validate and verify the enqueuing performance for two message brokers. The enqueuing performance means the throughput aspect versus the latency, moreover it focuses on the the two brokers' utilization of resources, such as CPU and memory.

Which of the two different message brokers (RabbitMQ and Apache Kafka) to use is a debatable topic [9], and this thesis work will try to shed a light on this subject.

The thesis 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.

1.4 Goal

The goals of this project are presented below:

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

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.

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

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

With the utilization of large cloud platforms and the message brokers that is used to send millions of messages between one destination to another, one also has to think of the essential storage solutions for the data, which in this case has resulted in building large datacenters. These datacenters consume 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 fundamental choosings of a research method is based on either a *quantitative* or *qualitative* method, both of these have different objectives 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. The problem or hypothesis needs to be validified as well as verified. The 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 the experiments conducted are based on numerical data.

Another aspect of importance in the thesis is the *philosophical assumption* that can be made, with several schools of thought to be considered. Firstly, the *positivism* element relies on the independency between the observer and the system to be observed as well as from the tools for which it is measured. Another philosophical school is the *realistic*, which collects data from observed phenomenons and thereafter develops a conclusion with regards to the collected data.

Although there are several other philosophical principles, for this project the most fitting was to use a combination of both positivism and realistic.

The thesis experiments with the correlation of variable changes and their relationship between one another in order to see how the different metrics become affected, therefore an experimental research methodology was chosen, as opposed to the non-experimental approach.

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 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

Chapter 2 has an in-depth technical background of Apache Kafka and RabbitMQ and their protocols. Chapter 3 presents the prior work that was done to choose the appropriate parameters and the related work that was found. Chapter 4 presents the design of the experiments and setup and the initial process. Chapter 5 presents the results as well as the interpretations and discussion of the results. Chapter 6 presents the conclusion of the thesis and the future 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, for example defining a schema in a programming language or letting two developers agree upon a request of some sort.

With this mutual agreement, two applications do not need to know the intricacies of each others system. The programming language and the details of the two systems can change over the years but as long as the mutual interface exists between them they will always be able to communicate with each other. This results in lower coupling between the applications because the two applications are not dependent of each other.

To further attain lower coupling one can introduce a *message system*. This message system makes the communication between sender and receiver less conformative by removing the necessity to store information on how many 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 to safeguard the message between two parties [4, p. 14].

The message system helps mitigate the problem of lost messages if a receiver goes down. The message system in this case will try to resubmit the messages until it is successful.

There are three different types of communication models that a message system can use, 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 producer sends a message to a queue and can thereafter continue with its execution without having to wait for an acknowledgment from the consumer. 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. Only one of the subscribed consumers receives the message, this can be seen in Figure 2.1.

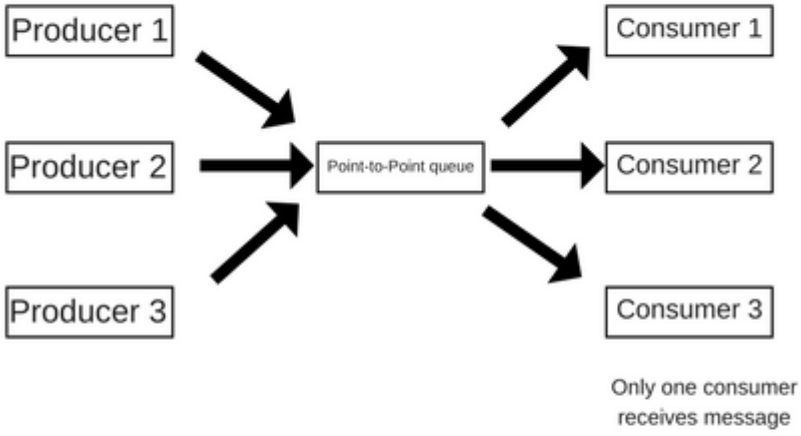

Figure 2.1: A point-to-point communication model. Only one consumer can receive the message from a producer.

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

Queues have important attributes, more specifically *persistence* and *durability*. The persistence attribute focuses on whether a failure occurs

during message processing, which is conducted by storing the message to some other form than in-memory for example a temporary folder, a database or a file [16].

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

With persistence, the reliability increases, however at the expense of performance and it all boils down to design choices for the message systems to choose how often to store a message 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 (subscribers) can retrieve messages. The sender does not need to have information on how many receivers there are and the receivers do not need to keep track on how many senders there are, the receiver can process the messages whenever preferred. This architecture is illustrated in Figure 2.2.

This type of model can be implemented with the help of *topics*. A topic can be seen as an event of which subscribers are interested, and whenever a message is sent to a topic from a publisher, all the subscribers of that topic become notified of the update.The publishers can publish messages to a topic and the subscribers can either subscribe or unsubscribe from the topic. Three important dimensionalities enhancing message passing for the publish/subscribe model are, *temporal*, *spatial* and *synchronization* [17, p. 1].

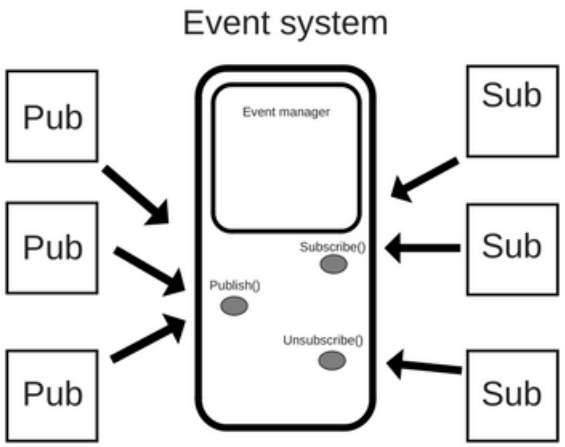

Figure 2.2: A publisher/subscriber domain. The publisher (pub) sends an event/message to the event system. The subscribers (sub) can subscribe or unsubscribe to this event.

With the *spatial* dimensionality, the requirement of two interacting parties needing to know each other is not present. Since 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 the event service [17, p. 2].

The *temporal* dimensionality does not require both the publisher and subscriber to be active at the same time. 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 refers to a publisher not needing 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 conducting 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, and NOT [17, p. 9].

2.3 Communication protocols

There are many different communication protocols that can be used when sending a message from one destination to another. These protocols have their own design goals and use cases. 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.

2.3.1 Advanced Message Queuing 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 great importance in the banking environment, due to the economical impact of a delayed, lost or incorrectly processed message [19].

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

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

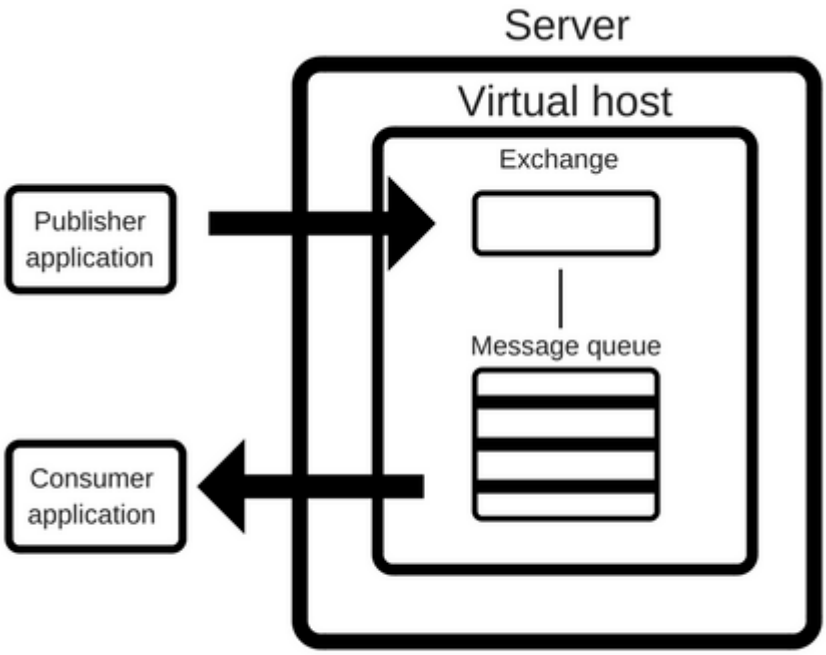

Figure 2.3: AMQP Protocol model. Publisher sends an event/message to the exchange, the exchange sends it to the appropriate queue, the consumer retrieves the message from the queue.

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 multiple readers of a queue exist, the one with the highest priority will take the message. 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 - 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 in turn determined with the help of a routing key [22, p. 50].

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

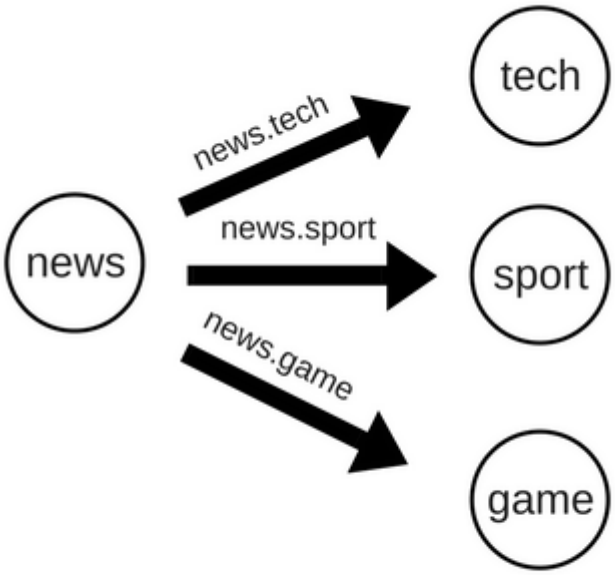

Figure 2.4: Topic exchange type. The topic is the news with tech, sport and game being queues that are a part of that topic.

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 due to the abundance of 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 focuses on the protocols and data entities that are used for real-time asynchronous communication such as instant messaging and streaming [23, p. 7].

XMPP utilizes the Extensible Markup Language (XML) to enable an exchange of data from one point to another. The technologies of XMPP

is based on a decentralized server-client architecture. This means a message sent from one destination is first sent to an XMPP server, thereafter sent to the receivers XMPP server, and finally to the receiver.

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

The <message/> stanza is used for pushing data from one destination to another, and is used for instant messaging, alerts and notifications [23, p. 18]. The <presence/> stanza is one of the key concepts of real-time communications, since this stanza enables others to know if a certain domain is online and is 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 publish-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, which the receiver must reply with and is often used to process a request. For error handling, 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 error message is that a stanza error can be recovered while other messages result in the closing of the XML stream, which 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 middle hand. 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 common used message operations [24].

STOMP is loosely modeled on HTTP and is built on frames 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 since 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, which enables the client to both be a producer by sending frame containing SEND, as well as a consumer 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 threshold 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 sends an ERROR frame back to the client.

2.3.4 Message Queue Telemetry Transport

The Message Queue Telemetry Transport (MQTT) is a lightweight messaging protocol with design goals aimed for an easy implementation standard, a high quality of service of data delivery and being lightweight and bandwidth 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, a payload of maximum 256 MB, and a Quality of Service level (QoS) [26]. There are 3 types of quality of service levels:

1. Level 0 - Employs a at-most-one 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 at least once from the intended recipient. This is conducted by sending a PUBACK message to the publishers, and until then, the publisher will store the message and attempt 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, since it guarantees that the message is received. This is carried out 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.

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

An indirect consequence of using SSL/TSL for encryption, is the augmentation of a significant overhead to the messages, which in turn contradicts the philosophy of MQTT being a lightweight protocol [27].

2.4 Apache Kafka

Apache Kafka is a distributed streaming platform used for processing streamed data, this platform scales elastically instead of having an individual message broker for each application [28]. Kafka is also a system used for storage, as it replicates and persists data in infinite time. The data is stored in-order, is durable, and 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 contain 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 of a topic, every partition can be allocated on different servers and is a key factor to enhancing scalability horizontally. This is illustrated in Figure 2.5.

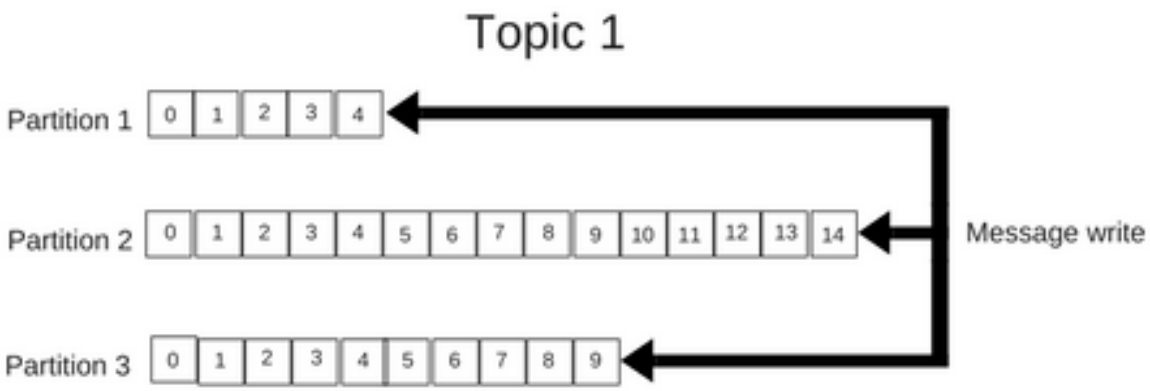

Figure 2.5: Topic partitioning. A topic consists of different partitions, the messages are written to a single partition. One consumer can read from one partition only.

The messages within a partition are 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, since 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 committing 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.

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, which can aid 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 on issues that arise in a system with multiple clusters. Examples of failures that can occur when you have distributed coordination of multiple servers, is that messages from one process to another can be delayed, the clock synchronization of different servers can lead to incorrect decisions regarding 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 utilizes the Actor-Model model. The Actor-Model is a conceptual model used for distributed computing and message passing. Every entity in the model which is an actor, receives a message and acts upon it. These actors are separated 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, i.e. all actors being considered independent, is a key feature to why RabbitMQ is scalable and robust.

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 RabbitMQ implements for AMQP is the **Exchange-to-Exchange** binding, enabling binding 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 disk. In Apache Kafka the retention mechanism of keeping messages for a set of time is usually carried out by writing to disk with

regards to the partitions of a topic. In RabbitMQ consumers will consume messages directly and rely on a ***prefetch-limit***, which can be seen as a counter for how many messages that have been unread, and is an indicator for a consumer that is starting to lag. This is a limitation to the scalability of RabbitMQ due to the fact that this prefetch limiter will cut of the consumer if it hits the threshold, 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, since 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 material from white papers to blog entries. This thesis focused on two primary metrics, throughput and latency, and two secondary metrics, CPU and memory usage, only one related paper from 2017 has done this [8], this paper is examined in section 3.3.

3.1 Metrics

Enqueuing 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 can be sent. Latency 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 to see how long a consumer has to wait on a message from a producer.

Furthermore, due to RabbitMQ and Kafka having their own unique architectures and design goals in regards to how they send and store messages, two secondary metrics were chosen; the resource usage of the CPU and memory. The CPU metric measured how much of the CPU was allocated to the system and user, the memory metric focused on how much data was written to disk. These two metrics were chosen over others, for example protocol overhead created when sending messages, firstly because it was deemed unfeasible to keep statistics of each message being sent over the network and, and secondly because RabbitMQ employs different protocols in comparison to Kafka.

3.2 Parameters

Because of how Kafka and RabbitMQ are designed, the parameter sets between them are not identical, there is no 1 to 1 relation where parameter X in Kafka is the same to parameter Y in RabbitMQ. Therefore,

an investigation was carried out in order to identify similar parameters between both architectures. Apart from the similarities identified (mentioned below, for the parameters affected), the findings also showed that Kafka has more configuration parameters than RabbitMQ. The parameters are described below.

3.2.1 Kafka parameters

- **Batch size** - The batch size is where the producer tries to collect many messages in one request when there are sent to the same partition in the broker, instead of sending each message individually.

- **Linger** - This parameter works together with the batch size, since batching occurs during loadtime when there are many messages that cannot be sent faster than they come. When this occurs one can add a configurable delay (in milliseconds) for when to send the records. By adding this delay, the batch size can fill up, which results in more messages being sent.

- **Acks** - This parameter has three different levels that can be set for the producer. Acks 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 wait for an acknowledgement from the leader. This means that a message is considered delivered after it has left the network socket.

  The second level is where the leader will have the message being written locally and thereafter responding to the producer. This can lead to messages being lost if the leader goes down before the replicas have 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**, **gzip**, and **lz4**.

- **Log flush interval messages** - This parameter decides how many messages in a partition should be kept before flushing it to disk.
- **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 sets a max-limit on how many messages that can be stored in the queue.
- **Direct exchange** - The direct exchange parameter is used to declare that the queues being used should have each message directly sent to its specific queue.
- **Fan-out 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 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 sets of parameters which meant that RabbitMQ and Kafka could not be compared on an equal basis. This made the focus of the thesis work shift to explore the performance of the different parameters in relation to throughput and latency, for both Kafka and RabbitMQ.

3.3 Related work

Testing of RabbitMQ against Kafka has not been conducted before, except for what was reported in [8], according to my findings. This report focused on two different delivery semantics 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 excluded. 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 opposite to this thesis where instead different parameters are tested and their impact on throughput and latency when running in the two different semantics. The report notes that three important factors for measuring throughput is the record size, partition and topic count, albeit true, as one can read from [33] that there are several other parameters that may affect the throughput and latency. The parameters are the *batch size*, *linger*, *compression algorithm* and *acks*, which the authors do not test.

Chapter 4

Experiment design

The work is divided in to four sections, section 4.1 presents a testing tool that was developed but scraped in favour of existing tools from RabbitMQ and Kafka. Section 4.2 is where a suitable cloud platform as well as the instances on that platform were chosen. It also presents the setup of the testing suite. Section 4.3 presents all the experiments that were conducted and designed for both Kafka and RabbitMQ. Section 4.4 shows the steps from setting up the servers, to conducting the experiments, to importing the data for data processing.

4.1 Initial process

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

The tool 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.

A more pragmatic solution was taken where tools provided by Kafka and RabbitMQ were used as they were inherently more flexible making it easier 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 was choosing a suitable cloud platform to conduct the experiments on, the chosen platform was Amazon Web Service (AWS).

AWS offers a variety of instances on their platform with focus on different use cases. These are comprised to a general purpose, compute optimization, memory optimization, accelerated computing or storage optimization [34]. These categories have their own subset of hardware, and the first choice to make was to pick which category to create an instance on. Instances in the general purpose section utilizes 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)**	**Hard-drive 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, as opposed to an object store, which stores data within a data-object model [35].

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

Four instances each were created for Kafka and RabbitMQ, in total eight instances. A cluster of three instances were to be tested against and the fourth one acted as a test runner. This was done to minimize external factors affecting the results, as opposed to testing the cluster locally, which in turn was done with the self-made tool. The setup for Kafka and

RabbitMQ can be seen in figure 4.1.

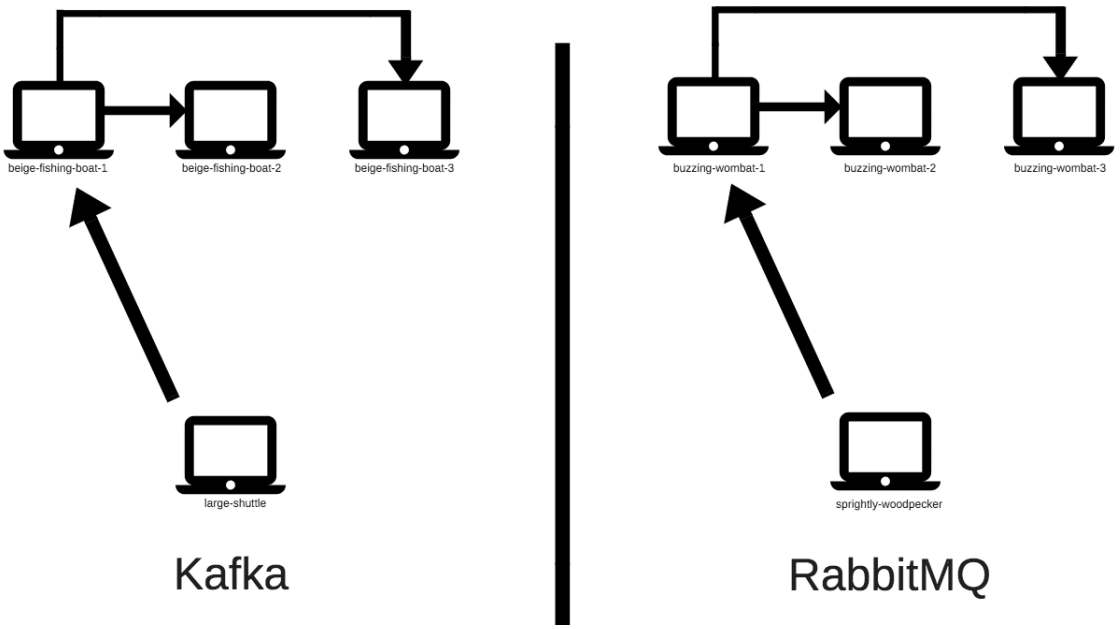

Figure 4.1: Beige-fishing-boat-1 is the main broker for the Kafka cluster, 2 and 3 are replicas, large-shuttle acts as a test runner. Buzzing-wombat-1 is the main broker for the RabbitMQ cluster, 2 and 3 are the replicas, sprightly-woodpecker is the test runner.

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 testing it 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 started 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 were 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. Firstly, a batch size interval of 10 000 to 200 000 was tested, with acknowledgement level two and snappy compression. The second experiment tested the same, however without any compression.

Experiment four

The fourth experiment tested flush interval message parameter. This experiment was conducted with two different partitions, 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 snappy compression was used.

4.3.2 RabbitMQ

Experiment one

This experiment tested the fan-out exchange, with queues from 1 to 50. This test was conducted firstly with manual acknowledgement and thereafter with 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 ran each experiment automatically and logged the findings.

This script works as such: it will connect to beige-fishing-boat or buzzing-wombat and start logging with dstat (disk usage) and mpstat (cpu usage) to two text-files. Thereafter it connects to either large-shuttle or sprightly-woodpecker depending on which message broker to be tested. These servers send messages to the main brokers and logs the results to a text-file. After this step, another connection to the beige-fishing-boat/buzzing-wombat was done in order to terminate the processes that is logging with mpstat and dstat. This was done to avoid having multiple stacks of processes running, which could effect the result of the CPU and memory logging.

With all of these tests being conducted, a large amount of text-files were being generated and it needed to be processed in an efficient way. Therefore, a parser was developed which could retrieve the data from these text-files, and generate an Excel^{TM} sheet, instead of manually filling each value from the text-file. This parser was put in front of the brokers, the whole system of the script and the servers and the parser can be seen in Figure 4.2.

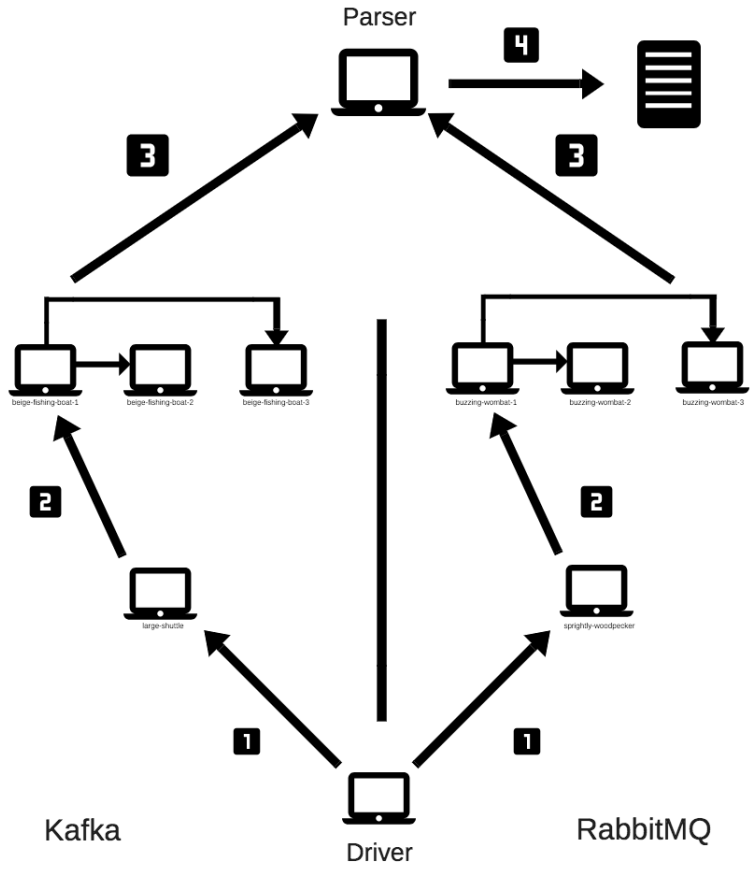

Figure 4.2: 1. The driver connects to the test runners. 2. The test runners sends messages to the main broker. 3. The results from the brokers gets sent to the parser. 4. The parser generates an Excel™ sheet.

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. The rest of the results are found in the appendix.

5.1 Kafka

5.1.1 Experiment one

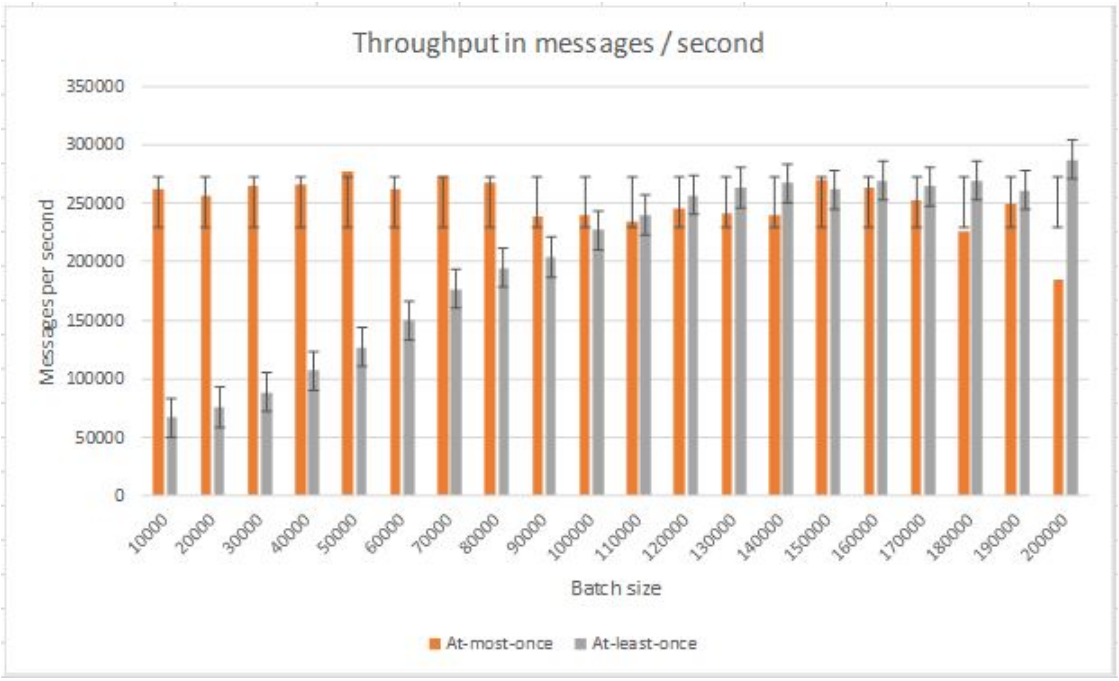

Figure 5.1: Throughput measured with 5 partitions. The batch size is set on the x-axis and the y-axis shows the throughput in messages per second. Compression algorithm used is snappy.

This first experiment shows that with delivery setting at-most-once, Kafka reaches a bit above 250 000 msg/s with a batch size of 10 000 with some test runs going down to 225 000 msg/s. With at-least-once delivery mode the throughput has a linear increase 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 around the same as the rest of the results for at-most-once so this can be dismissed. The error bars in this figure shows that for each batch the throughput is within a set of confined range which was set to 95%. This means that for each experiment run the amount of messages per second can be within the black margin for 95% of the time it is sampled. The error bars are not far from the actual data gathered except for the case of at most once with batch size 200 000 but all in all the throughput is within ± 10000 messages. The usage of confidence interval was chosen to demonstrate the accuracy of the gathered data.

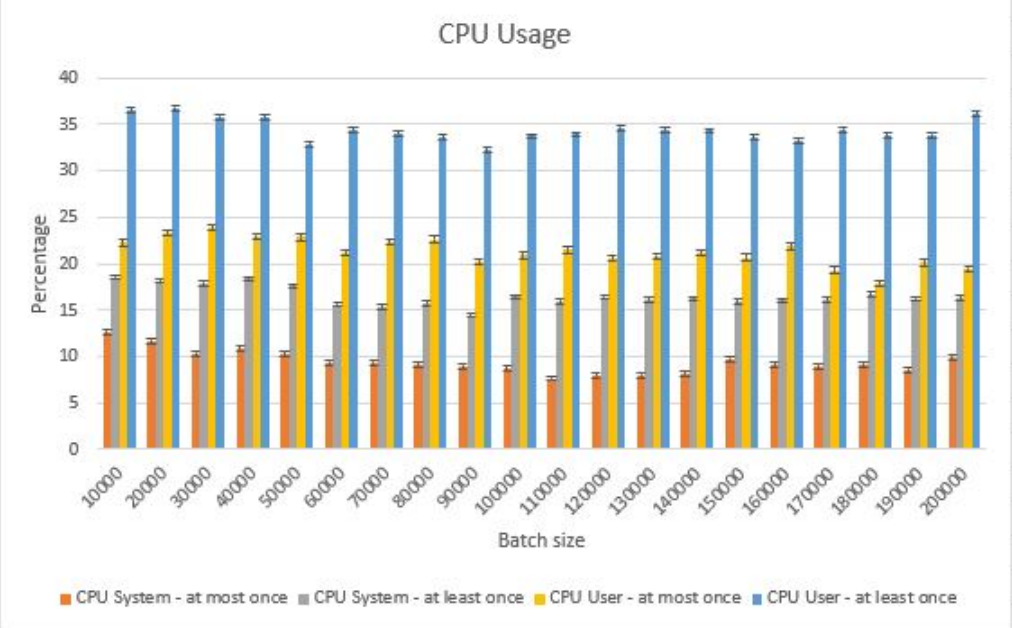

Figure 5.2: CPU Usage with 5 partitions. Batch size is on x-axis. CPU usage in percent is on the y-axis. Compression algorithm used is snappy.

Kafkas CPU usage shows that it is utilized more for user level calls than for system calls. The error margin is stable around ± 1 percent

which means that the data gathered is stable and does not fluctuate to external effects.

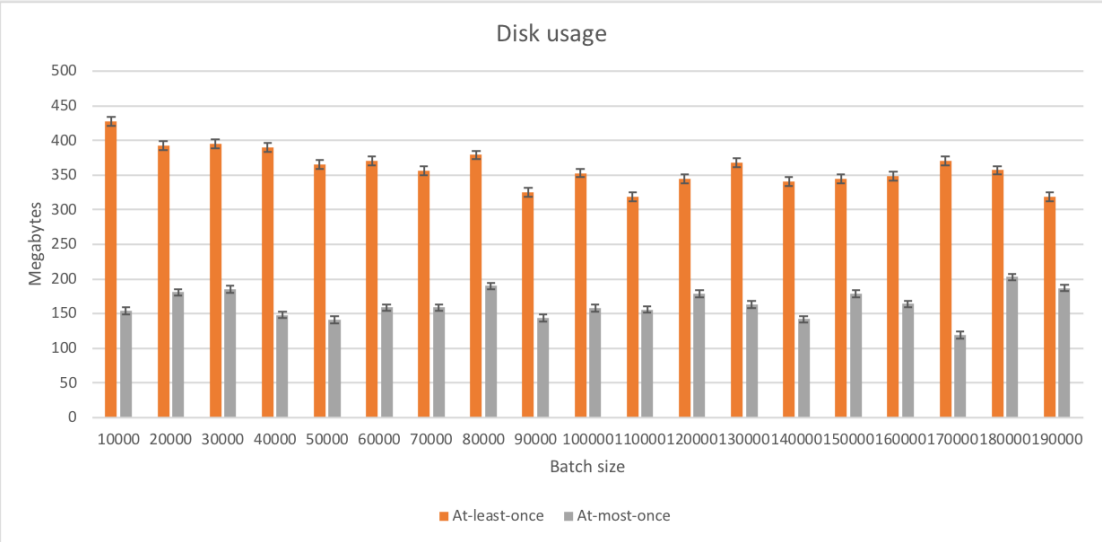

Figure 5.3: Disk usage with 5 partitions. Batch is on x-axis. Data written in megabytes is on y-axis. Compression algorithm used is snappy.

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 amount of data written can be correlated to the throughput. A higher throughput results in more data being written to disk. The level of confidence was set to 95% and the error bars for this experiment are small which means that the amount of data written does not vary that much ($\pm$ 10 megabytes) for each batch size.

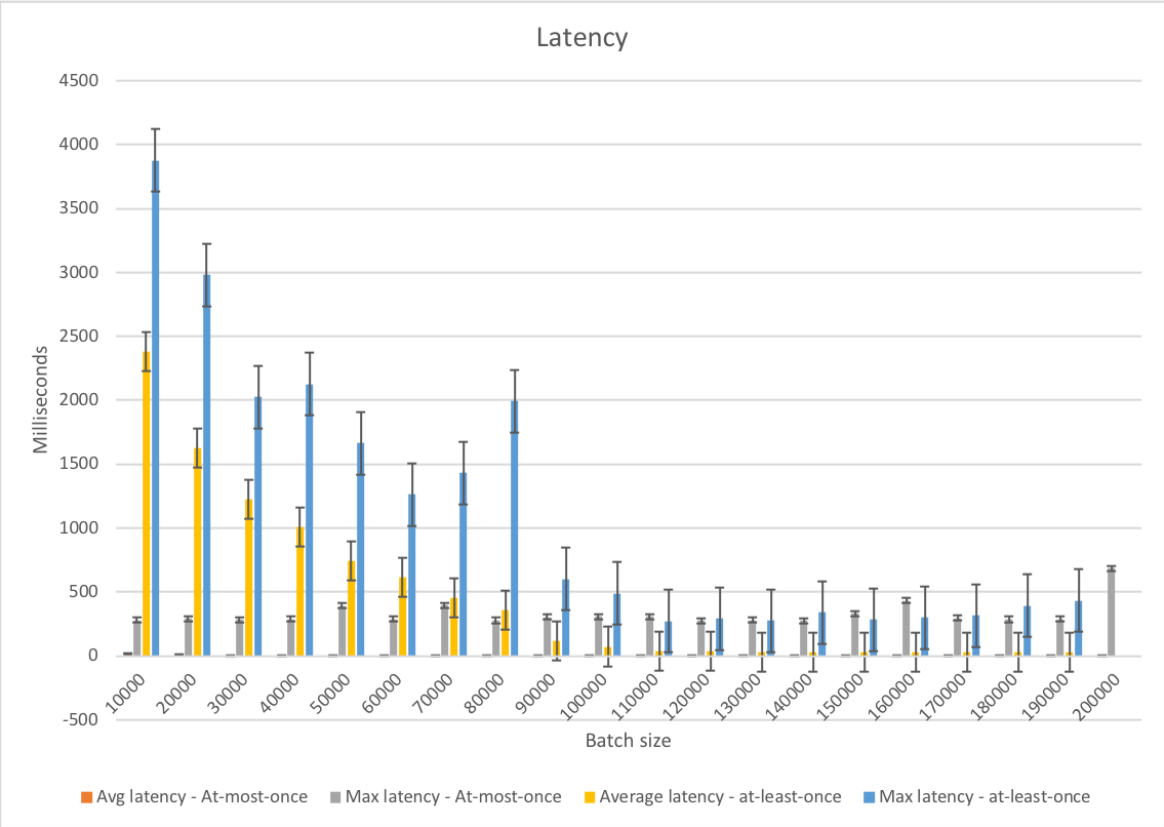

Figure 5.4: Latency measured with 5 partitions. Batch size is on x-axis. The latency in milliseconds is on the y-axis.

The latency observed for the first run 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 difference of 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.

The latency decreases for each increment of batch size which is counter intuitive because you send more messages which should result in higher latency, a plausible explanation to this is that Kafka deploys a configuration for the broker to wait for a required amount of bytes before serving a request to the consumer. This means that the broker can decrease the amount of bytes before serving a request to a consumer when the batch size is increased.

One anomaly is found for the max latency for batch size 80 000 with the delivery mode at-least-once which is higher than for smaller batch sizes, a possible explanation could be the affect of the operating system

(paging delay). Furthermore the error margin with the confidence level set to 95% shows that the latency can go below 0 for batch sizes above 100 000 which is impossible because you can not have a latency below 0 seconds. A different error metric would have been more useful for example using mean squared error in order to avoid having an accuracy below 0.

5.1.2 Experiment two

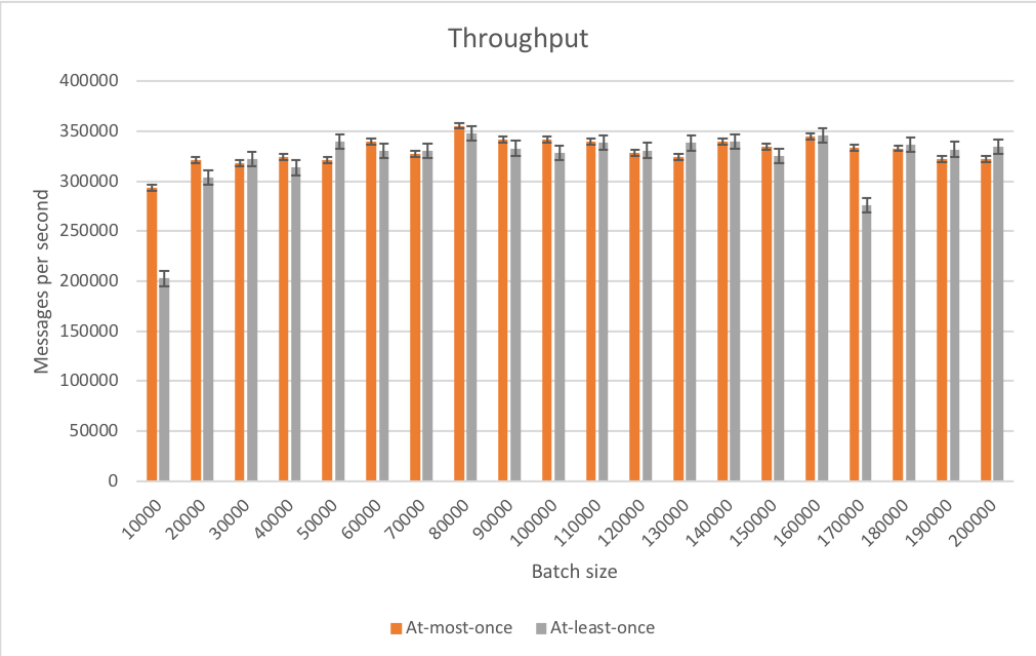

Figure 5.5: Throughput with linger set to 10. Partition set to 5. Batch size is on the x-axis. Throughput in messages per second on y-axis.

With the linger parameter, the maximum throughput increased from 250 000 to 350 000 msg/s in both delivery modes. This shows that with the linger parameter, Kafka, waits longer to fill up the batch before sending it. The throughput is not affected by the delivery mode presumably because the amount of time it takes to acknowledge the batch is as long or shorter then what the linger parameter is set to.

The error bars for this experiment shows that the gathered data only differs between $\pm$ 5000 messages with the same confidence level as the above mentioned experiments of 95%.

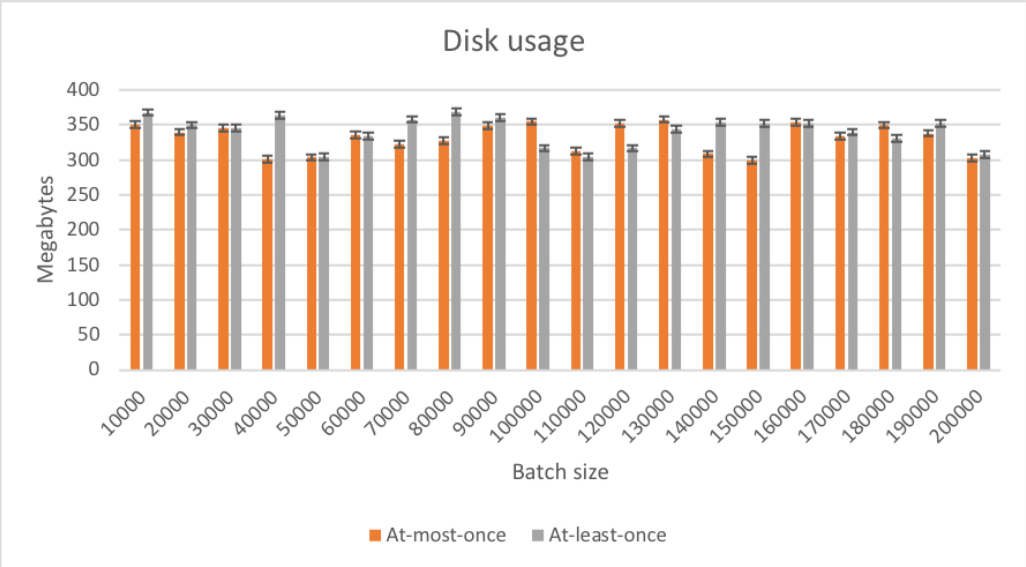

Figure 5.6: Disk usage with linger set to 10. Partition set to 5. Batch size is on the x-axis. Data written in megabytes is on the y-axis.

The disk usage is above 300 megabytes for both the delivery modes. The disk usage in this experiment is higher in comparison to Figure 5.3 because the throughput is higher hence more data needs to be written to disk. The batch size does not seem to affect the amount of data written because the linger parameter forces Kafka to wait before sending of a batch which makes the throughput roughly the same for all batches this results in the same amount of data being written regardless of batch size.

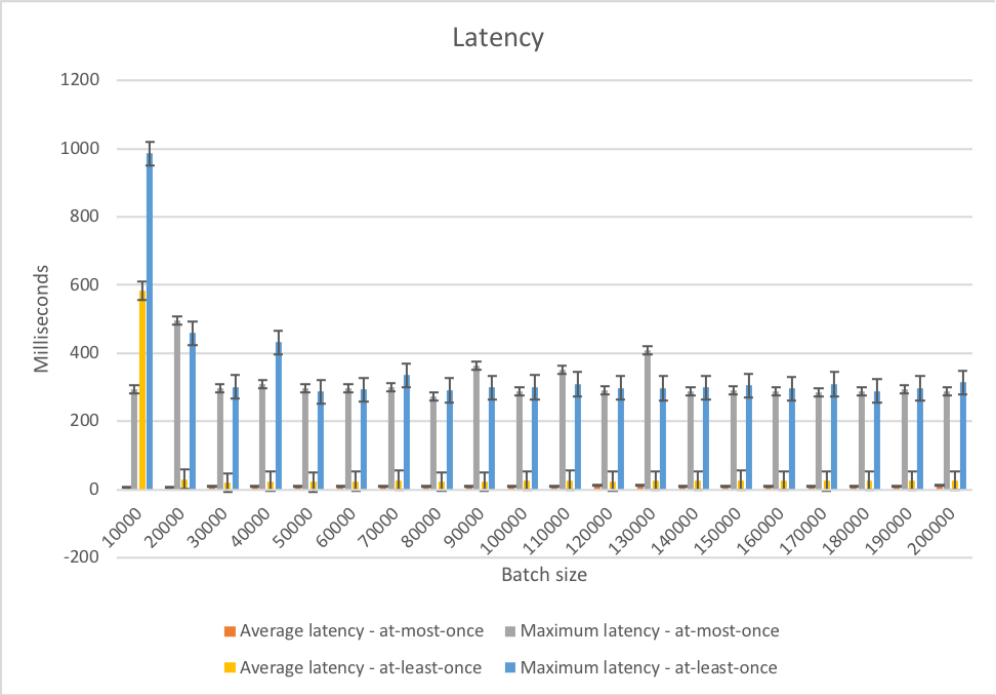

Figure 5.7: Latency with linger set to 10. Batch size is on the x-axis. Latency in milliseconds is on the y-axis.

The overall latency measurements are lower for this experiment in comparison to Figure 5.4. This is highly likely due to the linger parameter because Kafka waits a maximum of 10 milliseconds before sending a batch. One anomaly can be seen for the 10 000 batch that has a higher maximum latency for at least once delivery mode than all the others. A possible explanation to this is that the cluster on AWS does not get a bandwidth speed that is consistent through all runs which means that for this specific run the cluster got a decrease of bandwidth.

5.1.3 Experiment three

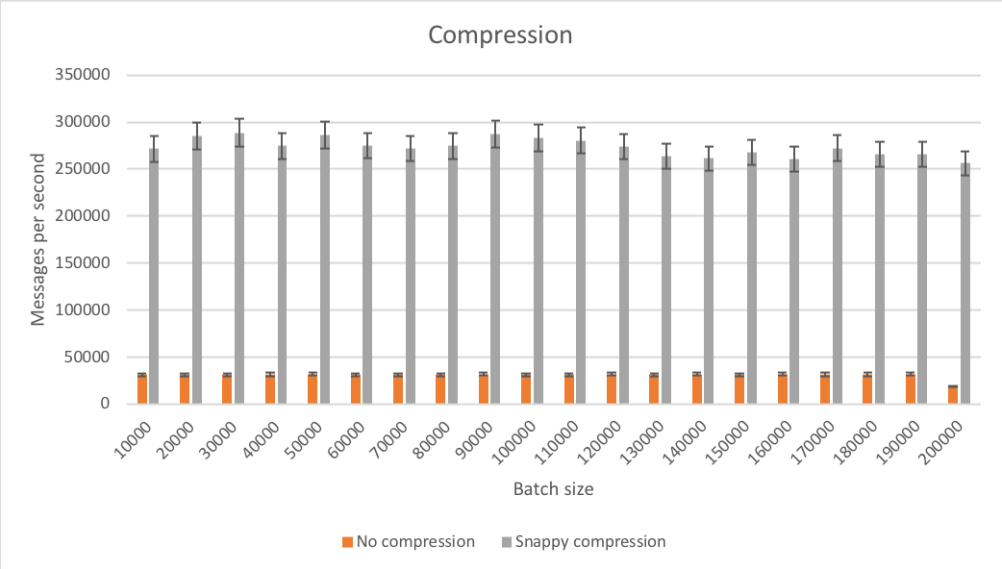

Figure 5.8: Throughput measurements with snappy compression and none compression. Batch is on the x-axis, throughput in messages per second on the y-axis.

When not using compression the expected throughput would be decreased, the throughput reaches below 50 000 msg/s. With the compression Kafka reaches above 250 000 msg/s for all runs. The impact on the throughput with the compression is huge primarily because the messages gets smaller which means that you can fit more messages into one batch. A downside with the testing tool is that the messages being sent are byte-arrays filled with the same data. This means that there is no variety of the compressed data which results in the same data and operations being fetched. This could affect the result of the runs that ran with a compression algorithm to perform better than with a normal use case of sending arbitrary messages.

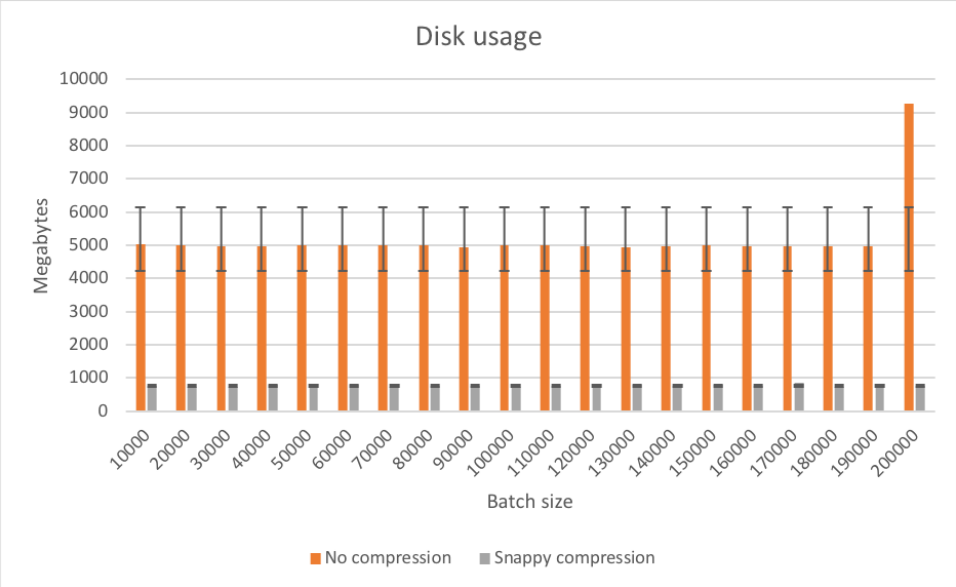

Figure 5.9: Disk usage by using compression and non-compression algorithm. Batch size on the x-axis. Data written in megabytes on the y-axis.

The disk usage does not get affected by the different batch sizes when there is no compression algorithm being used, they stay roughly the same for each batch size at 5 gigabytes except for the 200 000 batch size. A possible explanation to this is that Kafka can decide to send off a batch even if it still has not reached its batch size as the network socket gets filled with non-compressed messages.

An anomaly in this experiment was the last batch size which showed over 9 gigabytes of data being written, there is no reasonable explanation for why it should be writing more especially since the throughput for that batch is a bit lower than the rest. The error margin points to having the same amount of data being written as the rest of the batches which is quite peculiar. The only possible explanation for this behavior is that Kafka decided for that specific run to flush more messages to disk.

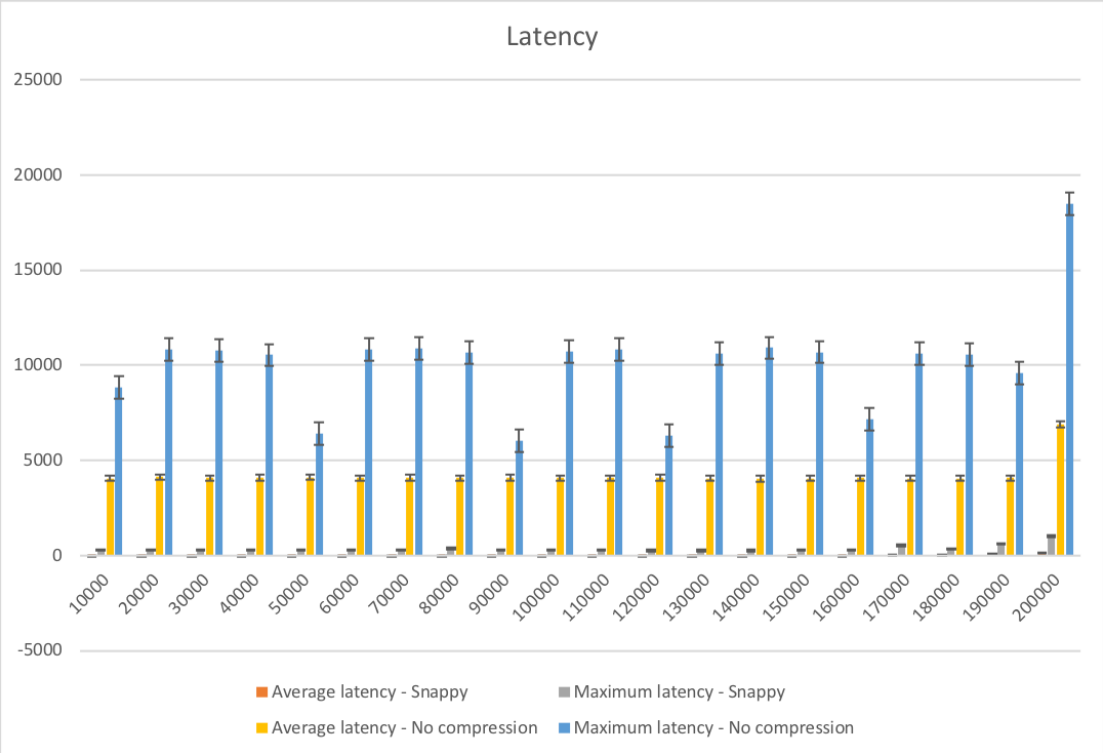

Figure 5.10: Latency measurement by using compression and non-compression. Batch size on x-axis. Latency in milliseconds on y-axis.

The impact of having no compression is also showing in the measurement of the latency. Without a compression algorithm the maximum latency reaches almost 20 seconds and having an average latency below 5 seconds. The error margin shows that the measurements are less than a second for 95% of the runs for the maximum latency measured. For average latency the error margin stretches between 0-400 milliseconds. The reason for this increased latency is linked with the size of messages being larger, these messages takes longer to write to disk. In hindsight one could have used a root mean square error method to avoid the negative error margin (which can be dismissed since the error margin is not showing) because as mentioned before, it is not possible to have a negative latency.

5.1.4 Experiment four

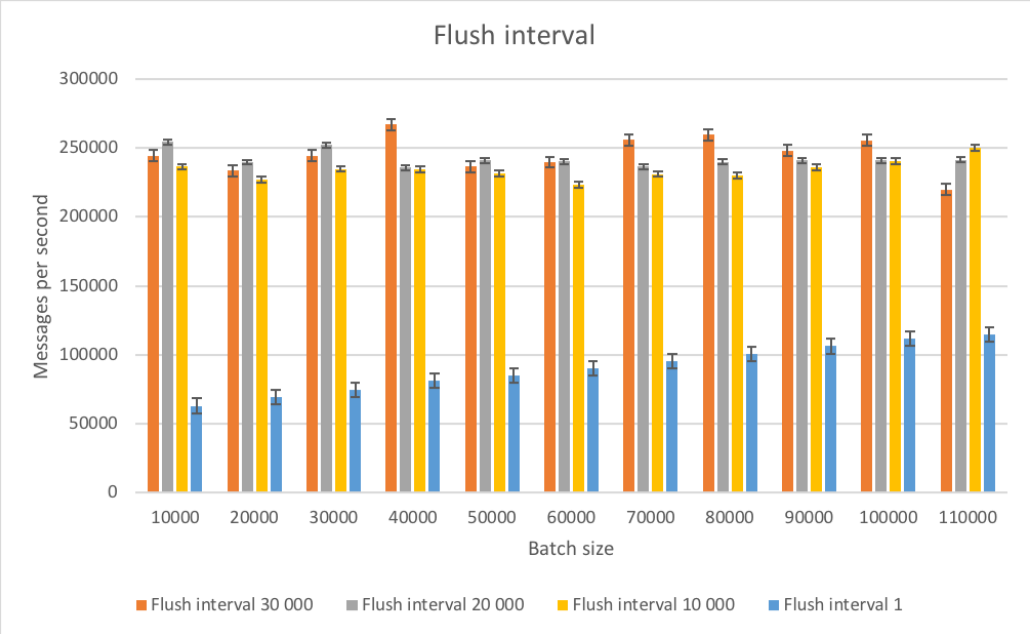

Figure 5.11: Throughput with various flush interval values. Batch size on the x-axis. Throughput in messages per second on the y-axis. Partition set to 5.

With the flush interval parameter the throughput is around the same as experiment 1, for values of 30 000, 20 000 and 10 000. With the flush interval set to 1, which means you flush each message to disk, the throughput starts at 50 000 and ends with around 110 000 msg/s depending on what batch size is used. The reason for this is that writing to disk is an expensive operation that takes time, which in turn affects the sending of messages if the producer has to wait for an acknowledgement for the whole batch. This experiment ran with delivery mode at least once because the flush interval parameter had no effect with at most once since the message is considered delivered after leaving the network socket.

The error margin for this experiment does not show anything unusual with the confidence interval set to 95%. The error margin is around $\pm$ 2000 messages for all measurements.

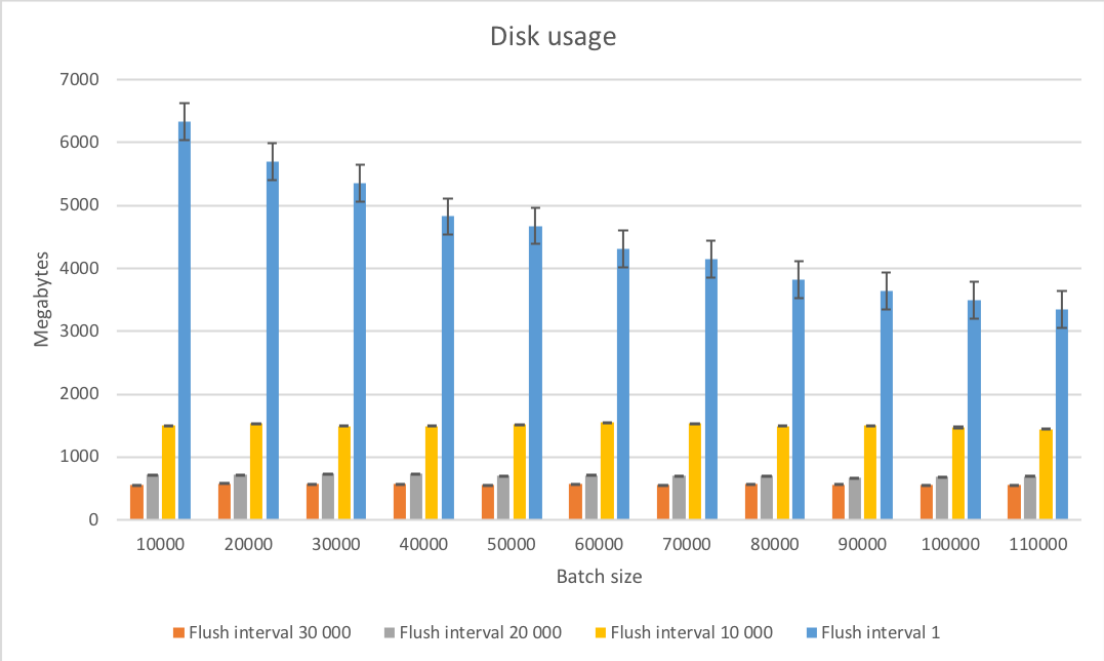

Figure 5.12: Disk usage with configured flush interval. Batch on the x-axis. Disk written in megabytes on the y-axis.

The disk usage of this experiment shows expected results, with flush interval set to 1 the amount of written data is the highest regardless of the batch size. The amount decreases for each batch size possibly because the first batch of 10 000 got filled while the other batch sizes did not.

5.2 RabbitMQ

5.2.1 Experiment one

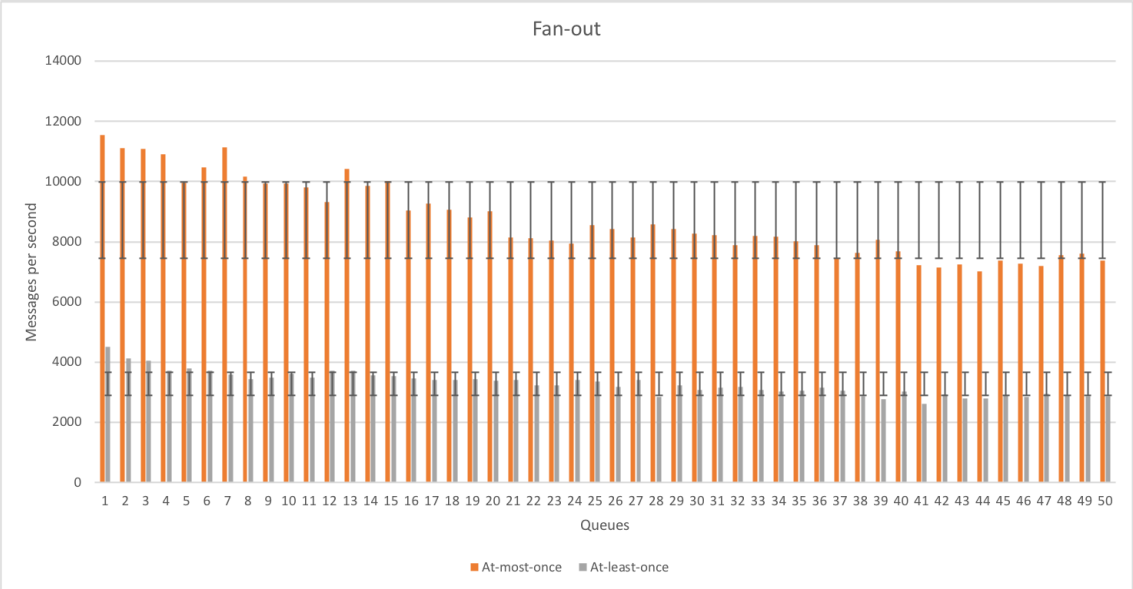

Figure 5.13: Throughput with fanout exchange. The amount of queues on x-axis. The throughput in messages per second on y-axis.

The throughput for using fan out exchange with 50 queues hits a maximum of 11 000 msg/s with delivery mode at most once, and a maximum of 4300 with at least once. The confidence interval was set to 95% and the error margins for at most once is ± 2000 messages for the first 20 queues. For 20 queues and above the error margin shows that the throughput can increase with approximately 1800 messages and below 200 messages of the measured data. For 40 queues and above the error margin shows that the throughput can be increased with 2000 messages which is based on earlier data points.

The error margins for delivery setting at most once can be deceiving as it would imply that with the increase of queues the throughput stays roughly the same which is wrong because the hardware puts a limit to the performance when the amount of queues is increased as it can not serve that many queues.

For at least once the throughput is roughly the same for all the queues. This shows that throughput is not affected in this delivery setting by the amount of queues if it has to wait for an acknowledgement from the

consumer. The error margin for at least once shows that the measured data with the confidence interval of 95% has an interval of ± 500 messages deviating from the actual result.

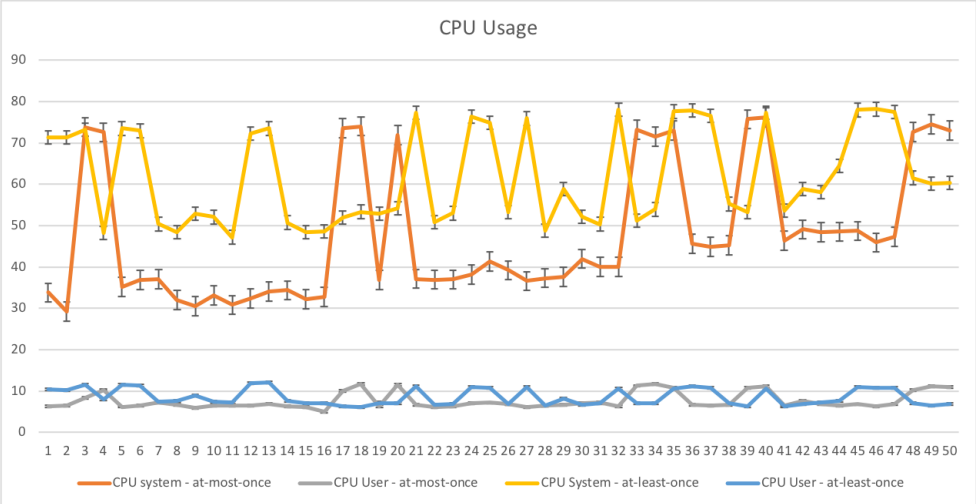

Figure 5.14: CPU usage with fanout exchange. Number of queues on the x-axis. CPU usage in percent on y-axis.

The CPU results for RabbitMQ shows that the usage of the user-level is between 0 to 10% for both delivery modes. The system usage is consistent between 30 to 40% for at most once delivery mode and spikes up to 70% several times. The system usage for at least once is around 50% and spikes up to around 75%. A possible reason for why these anomalies exists where the CPU spike up is because RabbitMQ deletes the queues randomly after the experiment has run unless it is specified to automatically delete the queues after each run.

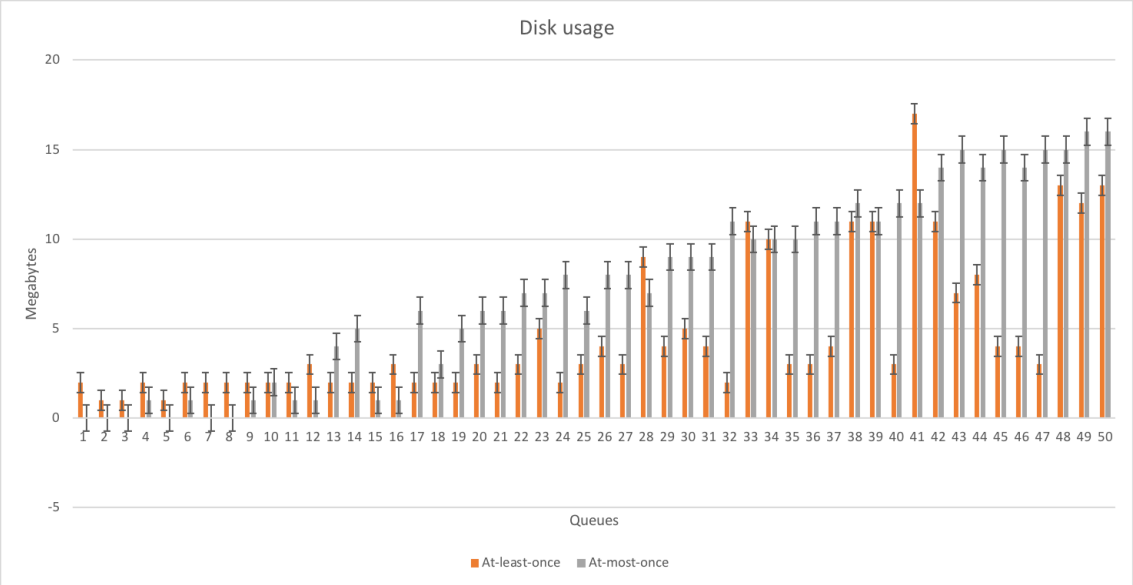

Figure 5.15: Disk usage with fanout exchange. Number of queues on x-axis. The amount of data written in y-axis.

The disk usage of RabbitMQ shows random behavior because the way RabbitMQ is optimized for keeping messages in memory. The amount of data being written is higher with the increase of queues but reaches only a maximum of 17 megabytes written. A possible explanation to the random behavior of the data written is that RabbitMQ evicts messages to the file system randomly unless the RAM is put under high pressure.

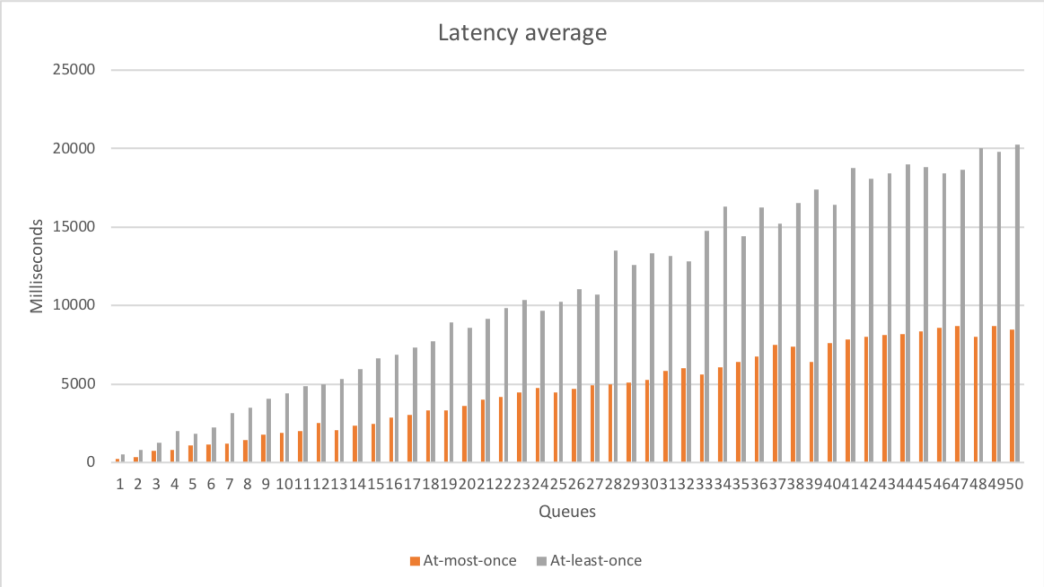

Figure 5.16: Latency measured with fanout exchange. Number of queues on the x-axis. Latency measured in milliseconds on y-axis.

The latency measurements for RabbitMQ shows that the latency increases with the addition of queues because RabbitMQ has to serve more messages for each queue that is started. The latency for delivery mode at least once increases and reaches a maximum of 20 seconds while delivery mode at most once reaches a maximum of 5 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 raw data. The reasons behind are discussed Section 5.3.

5.3 Discussion

Kafka 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 Figure 5.13 and Figure 5.1 with RabbitMQ reaching 12000 msg/s and Kafka reaching over 250 000 msg/s.

An interesting observation is the impact of applying compression with Kafka, with it, Kafka reaches over 250 000 msg/s and 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 enhanced the throughput in comparison to not having it configured, by almost 50 000 more messages, this can be seen in Figure 5.7.

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 configuration options as RabbitMQ with the potential workings of an exchange and binding messages with keys. RabbitMQ is more versatile when you want to configure delivery of specific messages to certain consumers.

The CPU utilization for Kafka and RabbitMQ are notably different which one can see from Figure 5.14 and Figure 5.2. The CPU utilization for RabbitMQ with the at-least-once mode staggers between 45-50% and spikes up to 70% for the system.

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 a maximum of around 20 megabytes in comparison to Kafka which in some experiments showed over 4 Gb of data being written.

Chapter 6

Conclusion

A distributed system needs a middleware to help with distributing messages from source to destination, this middleware persists and routes messages with the help of message brokers. There exists many different message brokers with their own design and implementation. These message brokers in turn use message queues that can be configured for different purposes. This thesis focused on testing two message brokers, RabbitMQ and Kafka, to understand the trade off between durability and throughput with regards to their configurable parameters.

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. The goals of this thesis was to design experiments for Kafka and RabbitMQ, which was conducted 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. Kafka writes more to disk than RabbitMQ, but is more CPU-heavy than Kafka.

6.1 Future work

Possible future work could be to replicate it on other hardware configurations to see how much is affected and limited by the hardware. It would also be interesting to set up a larger cluster of nodes and to simulate nodes going down in order to see how well Kafka and RabbitMQ deals with lost messages. Furthermore, this work could be extended to test

Kafka in exactly-once mode to compare the performance with the two other delivery modes.

It would be interesting to send other types of messages instead of default message types in order to see how effective the compression parameter for Kafka is and to see if the performance gets affected by sending arbitrary messages.

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

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

[35] *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, https://github.com/Esoteric925/MTReport

TRITA TRITA-EECS-EX-2018:356