Degree Programme in Computer Engineering

First Cycle 15 credits

Performance of message brokers in event-driven architecture: Amazon SNS/SQS vs Apache Kafka

Prestanda av meddelandeköer i händelsedriven arkitektur: Amazon SNS/SQS vs Apache Kafka

STOCKHOLM, SVERIGE 2023

Performance analysis of message brokers in event-driven architecture: Amazon SNS/SQS vs Apache Kafka

Prestandaanalys av meddelandeköer i händelsedriven arkitektur: Amazon SNS/SQS vs Apache Kafka

Ivan Zivkovic
Johan Edeland

Examensarbete inom datateknik
Grundnivå, 15 hp
Handledare på KTH: Lars Johan Anders Cajander
Examinator: Ibrahim Orhan
TRITA-CBH-GRU-2023:258

KTH
Skolan för kemi, bioteknologi och hälsa
141 52 Huddinge, Sverige

Sammanfattning

Mikrotjänstarkitektur, som innebär att applikationer bryts ned till mindre och löst kopplade komponenter, är något som blir allt vanligare vid utvecklingen av moderna system. Kopplingar mellan dessa komponenter kan etableras på olika sätt. Ett av dessa tillvägagångssätt är händelsedriven arkitektur, där komponenterna i systemet kommunicerar asynkront med varandra genom meddelandeköer.

AWS, Amazon Web Services, som är den största leverantören av molnbaserade tjänster, tillhandahåller en sådan meddelandekö: SQS, Amazon Simple Queue Service, som kan integreras med SNS, Amazon Simple Notification Service för att möjliggöra ”en-till-många” asynkron kommunikation.

Ett alternativt verktyg är Apache Kafka, skapat av Linkedin och senare öppen källkodspublicerad under Apache Software Foundation. Apache Kafka är en händelselogg och strömningsplattform som även kan fungera som en meddelandekö i en händelsedriven arkitektur.

Författarna av detta arbete har på uppdrag av Scania blivit ombedda att jämföra och utvärdera prestandan hos de två verktygen samt undersöka om det finns användningsfall där det ena kan vara mer lämpligt än det andra. För att uppnå detta utvecklades två prototyper, där varje prototyp består av en producent- och en konsumentmikrotjänst. Dessa prototyper användes sedan för att genomföra latens- och lasttester genom att producera meddelanden och mäta tidsintervallet till dess att de konsumerades.

Resultatet från testerna visar att Apache Kafka har lägre genomsnittlig latens än SNS/SQS och skalar mer effektivt vid ökande datamängder, vilket gör det mer lämpat för användningsfall med realtidsströmning av data. Dess potential som meddelandebuss för löst kopplade komponenter i systemet är också tydlig. I detta sammanhang är SNS/SQS lika användbart, då det fungerar som en dedikerad meddelandebuss med god latens och en användarvänlig och enkel startprocess.

Nyckelord

händelsedriven arkitektur, meddelandebuss, latenstestning, lasttestning, data strömning, moln, backend, Apache Kafka, Amazon Simple Queue Service, Amazon Simple Notification Service

Abstract

Microservice architecture, which involves breaking down applications into smaller and loosely coupled components, is becoming increasingly common in the development of modern systems. Connections between these components can be established in various ways. One of these approaches is event-driven architecture, where components in the system communicate asynchronously with each other through message queues.

AWS, Amazon Web Services, the largest provider of cloud-based services, offers such a messaging queue: SQS, Amazon Simple Queue Service, which can be integrated with SNS, Amazon Simple Notification Service, to enable "one-to-many" asynchronous communication.

An alternative tool is Apache Kafka, created by LinkedIn and later open-sourced under the Apache Software Foundation. Apache Kafka is an event logging and streaming platform that can also function as a message queue in an event-driven architecture.

The authors of this thesis have been commissioned by Scania to compare and evaluate the performance of these two tools and investigate whether there are use cases where one might be more suitable than the other. To achieve this, two prototypes were developed, each prototype consisting of a producer microservice and a consumer microservice. These prototypes were then used to conduct latency and load tests by producing messages and measuring the time interval until they were consumed.

The results of the tests show that Apache Kafka has a lower average latency than SNS/SQS and scales more efficiently with increasing data volumes, making it more suitable for use cases involving real-time data streaming. Its potential as a message bus for loosely coupled components in the system is also evident. In this context, SNS/SQS is equally valuable, as it operates as a dedicated message bus with good latency and offers a user-friendly and straightforward setup process.

Keywords

event-based architecture, message broker, latency testing, load testing, data streaming, cloud, backend, Apache Kafka, Amazon Simple Queue Service, Amazon Simple Notification Service

Table of contents

1 Introduction 1

1.1 Problem Definition 1

1.2 Goals 1

1.3 Limitations 2

2 Theory and Background 3

2.1 Cloud 3

2.1.1 Cloud Service Models 3

2.1.2 Deployment Models 4

2.1.3 Challenges with Cloud Computing 5

2.2 Amazon Web Services (AWS) 6

2.3 Inter-process communication 6

2.3.1 Synchronous communication 6

2.3.2 Asynchronous communication 7

2.4 Message Brokers 7

2.4.1 Message Queues 7

2.4.2 Publish-subscribe 8

2.4.3 Event streams 9

2.5 Simple Queue Service (SQS) 9

2.6 Simple Notification Service (SNS) 9

2.7 Combining SQS and SNS 9

2.8 Apache Kafka 9

2.9 Related Work 10

3 Method 11

3.1 Setting up the AWS Environment 11

3.2 Measuring Performance 11

3.2.1 Latency 12

3.2.2 Load 12

3.3 Setting up the Microservices 12

3.3.1 SNS/SQS Microservices 12

3.3.2 Kafka Microservices 13

3.4 Summarize the data 14

4 Results 15

4.1 Latency test case 15

4.2 Load test case 16

4.2.1 100 messages per second 17
4.2.2 200 messages per second 18
4.2.3 300 messages per second 19
4.2.4 400 messages per second 20
4.2.5 500 messages per second 21
4.2.6 600 messages per second 22
4.2.7 700 messages per second 23
4.2.8 780 messages per second 24
4.2.9 Average values 25
5 Analysis and Discussion 27
5.1 Latency test case 27
5.2 Load test case 27
5.3 Use Cases 28
5.4 Evaluation of the Research Method 28
5.5 Sustainability 29
6 Conclusions 31
6.1 Future Work 31
References 33
Appendix 37

1 Introduction

Traton Group, also known as Traton, is a subsidiary of Volkswagen that specializes in manufacturing trucks and buses. Traton is the parent company of Scania, MAN, and Navistar. The vision for the future is that these three companies will foster transparency and mutual support in terms of design and part sharing, gradually bringing the companies closer together through step-by-step integration.

Since each company have their own systems and databases, this vision for transparency gives rise to the demand for systems that ease the communication and data between the companies. A department at Scania is responsible for developing a collaboration system that helps the companies share data between each other and help synchronize their databases. To do this, every time data is shared or updated, the companies involved need to be notified that they need to update their database.

The way the companies get notified today is through a subscription to an Amazon Simple Queue Service (SQS) queue. When a company has shared new data or made updates to existing data, a message is sent to a topic using Amazon Simple Notification Service (SNS). This message then gets forwarded through SNS to the SQS queues that the other companies are subscribed to. Each company has their own SQS queue that they are subscribed to. When the message has been received the respective company can update its database according to the change.

1.1 Problem Definition

SQS and SNS have limitations, such as message size, message retention time, and message latency. (1) Larger messages may have to be split into several smaller ones to be able to send them. This does not impose a problem today for the Scania data-sharing system as message sizes are smaller than the maximum allowed, however, this may come to change in the future, so it is something to consider.

The problem at hand is that messages in SNS do not have any capability of message retention (2) and SQS can only retain messages for up to 14 days (1) which introduces problems concerning troubleshooting an application. This retention time means it is only possible to troubleshoot at most 14 days back in time if there is an error concerning the event-based system.

This thesis will investigate if switching to an alternative cloud-based message broker with a longer message retention time can be done without a sacrifice in performance.

1.2 Goals

This thesis will investigate the differences between using AWS SQS combined with SNS and using Apache Kafka hosted in an AWS environment as message brokers for an event-based system.

1.3 Limitations

This thesis will investigate the differences in the performance of the message brokers with perspective to latency, and how a heavier load will affect the latency. Additionally, it will investigate if there can be use cases where one framework is preferred over the other.

2 Theory and Background

This chapter will explore the basics of cloud computing, the different cloud computing models and services offered by cloud providers today, and go through some of the challenges with cloud computing today. It will then give a small background to AWS, what it is, and what services it provides. Then, it will go through the most important concepts that message brokers and event streaming platforms are built on, such as asynchronous/synchronous communication and publish/subscribe systems. The chapter will end by giving a background to SQS, SNS, and Apache Kafka, which are the frameworks that this thesis aims to explore in more detail.

2.1 Cloud

Cloud computing has been in increasing popularity since its introduction to the IT scene at the beginning of the 2000s. As virtualization technology (3) became more well-known and the rapid increase of internet bandwidth and processing power of computers, cloud computing began to gain a stronger foothold in the IT industry. (4)

Cloud computing gives companies or private persons access to computing resources at remote locations over the internet. Examples of these resources are servers, databases, applications, and other services. Cloud computing builds on virtualization technology that can create virtualized environments for applications and users. Virtualization makes it easy to scale a business up and down dynamically in response to the change in demand. This puts the resources to efficient use, minimizing waste, which in turn reduces the cost of the company using the services and helps them improve the agility of the organization. (5)

Cloud environments make scaling businesses easier depending on their current business demands. Being able to scale as you grow is important to both new and old companies since they can focus solely on developing their product without having to think about allocating more hardware or software resources as time goes on. This is also a cost-effective solution for companies since they will only pay for the resources, they use (6). Having a cloud resource provider also makes collaborating within bigger organizations that have many different teams and departments spread out throughout the world easier. Since the resources reside in the cloud, they are accessible from anywhere with an internet connection.

Cloud computing is as of today mainstream technology in software development, helping companies from many industries increase their scalability, agility, and efficiency all the while reducing the costs of owning and maintaining hardware resources needed.

2.1.1 Cloud Service Models

Cloud providers offer three types of service models, Infrastructure as a Service (Iaas), Platform as a Service (PaaS), and Software as a Service (Saas). Either all or some of these models are provided by most of the biggest cloud providers in the market today. (7)

2.1.1.1 Infrastructure as a Service (IaaS)

IaaS is a service that provides hardware resources such as computing, storage, server hosting, and networking. Management of these resources is not included in the service, that responsibility will be the paying customer. (7)

2.1.1.2 Software as a Service (SaaS)

SaaS provides an application in the cloud environment that is accessible to a client without the need to install anything on the client's computer. Such applications can be websites that are accessed through a web browser on the client side. (7)

2.1.1.3 Platform as a Service (PaaS)

PaaS gives the customer the complete package to develop and deploy applications with all the necessary infrastructure services and software services needed. It lets companies focus more on development and spend less time managing their deployment. (7)

2.1.2 Deployment Models

There are different ways of deploying a cloud environment. There is no best praxis and companies may use their model based on their specific needs. The most used models are public cloud, private cloud, and hybrid cloud, which is a combination of the public and private cloud. (8)

2.1.2.1 Public Cloud

A public cloud is a deployment model offered by a cloud service provider that offers different cloud computing services such as storage, hosting servers, and other applications that can be useful for software-developing organizations. (8)

The public cloud model was predicted to see a lot of growth for the 2010s. (9) There are multiple reasons for this. For one, it allows software developing organizations to get a development and deployment environment up and running in a quickly and easily since it gives instant access to all functionalities requested from the cloud service provider. The organizations will save a lot of time and effort by not having to set up their own data centers and servers, also saving money in hiring appropriate technical personnel to implement. Using a cloud service provider, companies will only have to pay for the resources they use, in turn, making companies reduce their waste of money.

This trend of using a public cloud has led to huge growth in the public cloud computing market. (10) Today there are several public cloud providers, among the most popular ones are Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, and Oracle Cloud.

2.1.2.2 Private Cloud

The private cloud is an environment that exists only in private within an organization. In a private cloud, the resources and hardware are owned and managed by the organization which gives them complete control over the resources. This is beneficial when the organization needs security and wants to protect its data since the environment is only accessible by users inside the organization. Private clouds also give more control regarding deployment and use of the environment. (8)

2.1.2.3 Hybrid Cloud

Another cloud environment model is the hybrid cloud. The hybrid cloud combines the flexibility and abstraction that come from the public cloud with the control over security and ease of management from the private cloud. In this model, the organization will link its private cloud to one or more public clouds. It provides a secure way to manage data and gives access to the computing resources offered by the public cloud. (8)

2.1.3 Challenges with Cloud Computing

It can seem as though making use of cloud computing is the definitive way to go forward when building applications and distributed systems. There are some challenges that companies need to take into consideration when deciding whether to go in that direction or not. (11)

2.1.3.1 Security and privacy

Security and privacy are important factors to consider when building applications and distributed systems. Some of the challenges regarding cloud computing are data breaches, malware infections, insider threats, Denial of Service (DoS) attacks, insecure APIs, and lack of transparency. (12)

Data breaches and data stored on remote servers can be endangered if the cloud provider hasn't taken the proper security measures to secure their customer's data. Cloud computing involves connecting multiple servers and data sources, making it easier for malware infections to spread from one infected system to another if connected. Insider threats, there will always be a risk of human error, and a cloud provider's employee may intentionally or unintentionally make an error that compromises a customer's data. Denial of Service attacks are an attempt to overload the cloud provider's servers, causing them to go down which disrupts the applications connected to that cloud provider. Managing and interacting with the cloud services are done via APIs, and if an API is not secured properly, it will present a security threat to the cloud provider, such as injection attacks. (13) Lack of transparency of the cloud provider can make it hard for customers to evaluate how safe their data is when residing in the cloud. (11)

Companies must do a proper evaluation of the different cloud providers available to ensure the security of the applications and data they are looking to deploy in the cloud.

2.1.3.2 Migrating between cloud providers

Once a company has picked its cloud services provider, it becomes dependent on that provider's infrastructure, applications, or other services. This is not a problem for a company that has done its research on which provider to choose. But for companies that want to migrate to a different cloud provider, it can be problematic. (14) For companies that want to go back to their own infrastructure and computing resources the dependency of the cloud provider can cause that migration to be problematic as well.

2.2 Amazon Web Services (AWS)

Amazon Web Services is one of the leading cloud providers in the market. It is estimated that AWS has a market share of 35 percent. (15) (16) (17)

There are several reasons for this lead in market shares. AWS was one of the very first companies to offer these cloud services when they started back in 2006. This gave them a head start over competitors such as Microsoft Azure, IBM Cloud, Google Cloud Platform, Oracle Cloud, etc. and allowed them to acquire a large customer base that has stayed loyal over the years since. AWS has made sure to use this head start to its advantage by offering a wide range of cloud services such as the usual computing, storage, and data analytics but also AI (Artificial Intelligence) and ML (Machine Learning) services. (18) AWS has also gained a reputation for its reliability with its services being fault tolerant and highly available, its scalability, (19) and its constant investment in innovation, always being in the frontline developing new services and features for their customers.

2.3 Inter-process communication

Inter-process communication (IPC) enables communication and data sharing between different processes within a computer system. Shared memory, message passing, pipes, sockets, and remote procedure calls (RPC) are all methods of inter-process communication. (20)

These methods can be divided into synchronous and asynchronous communication methods. Pipes, sockets, and RPC are all forms of synchronous communication, while shared memory and message passing belong to asynchronous communication. (20)

Pipes and sockets are used to open a byte stream of communication between processes. RPC allows one process to call procedures in others processes and get the return value of that procedure as if it were running in the same process. Shared memory is a technique that creates a partition in the memory that multiple other processes can access. The message-passing technique uses an intermediary that holds sent messages until later retrieval, often in the form of a message queue. (20)

Modern computer systems run many different applications, and oftentimes these applications run multiple processes simultaneously, examples are operating systems, databases, network servers, and multimedia applications. This makes IPC an essential component of these systems. (20)

2.3.1 Synchronous communication

Synchronous communication is communication where the sender and receiver are both available for communicating at a given time. The basis for synchronous communication is through a request-response pattern where the sender of a request waits for a response before proceeding. An example of synchronous communication is a client-server application where the client typically sends a request to the server via HTTP and waits for a response in real-time. (21) (22)

In some use cases in distributed systems, it is not appropriate with synchronous communication, since the systems can be composed of many different components and can have a complex structure that can be hard to synchronize. Those are the cases where asynchronous communication will do better. (20)

2.3.2 Asynchronous communication

Asynchronous communication is communication between sender and receiver where they are not required to be in sync to communicate with each other and may not even be aware of each other. This can be accomplished using an intermediary, where a sender can send messages to the intermediary which then forwards the messages to one or more receivers. Having an intermediary establishes a decoupling between senders and receivers. (20)

This thesis will focus on two asynchronous communication patterns, publish-subscribe and message queues.

2.4 Message Brokers

A message broker is a component in software applications that acts as an intermediary between producers and consumers of messages. Their main function is to receive messages from producers and forward them to one or more consumers. How this is done is determined by the implementation. The message broker can be implemented with a message queue or a publish-subscribe system or as a hybrid with a combination of both. (23)

There are many different options when it comes to message brokers. The two message brokers that this thesis will delve deeper into are Apache Kafka and the combination of AWS Simple Queue Service (SQS) and Simple Notification Service (SNS). (24)

2.4.1 Message Queues

Message queues are an asynchronous messaging pattern where a message producer sends a message to a message queue and later a single consumer retrieves the message from the queue and consumes the message. Message queues provide one-to-one communication between a producer and a consumer. The communication between them becomes asynchronous and decoupled, meaning that the consumer does not need to be active at the time the message is sent. (25)

The messages in a queue are most often ordered in a first-in-first-out (FIFO) way but are not limited to it. Message queues could order their messages in a priority order where a message with the highest priority gets sent first. Messages that have been sent to the queue will remain in the queue until they get consumed by the consumer. Therefore, message queues have a guaranteed message delivery. (25)

When a message queue has received a message there are different strategies as to how the consumer is made aware of the new message, depending on implementation. The consumer could wait for new messages by continuously checking the queue for new messages in a blocking manner. Alternatively, a non-blocking approach is to poll the queue from time to time and be aware of any new messages in this way. Another

way to let the consumer know there is a new message to be consumed is to notify the consumer. This approach makes the message queue responsible for notifying the consumer when there has been an update. (20)

2.4.2 Publish-subscribe

Publish-subscribe communication is an asynchronous messaging pattern where there are message producers (publishers) and message consumers (subscribers) and some sort of intermediary that is used to decouple the publishers and subscribers from each other. Participants in the same communication system can take on either one of the roles, or both. A publisher can be a subscriber and vice versa. (26)

Messages are produced by publishers and are referred to as events. The structure of the events is determined beforehand and can differ depending on the use case. When an event is produced, it is sent to an intermediary that stores the event. The intermediary is then responsible for distributing the event to the relevant subscribers. (20)

A subscriber becomes relevant for some types of events by showing interest in them which can be done by subscribing to them. What separates events from each other is different depending on the use case. As an example, events could be distinguished based on what the overall topic of the event is. A system could have two topics, one for movie-related events and one for book-related events. Subscribers could then subscribe to the movie topic and in doing so receive all the published events on that topic. (20)

When the intermediary receives an event, it matches that event to the different subscribers who are interested in that type of event and sends out an event notification to them. This is to let the subscribers know that a new event has arrived that can be read from the intermediary. With the possibility of having multiple subscribers to the type of events published by a single publisher makes the publish-subscribe communication pattern a one-to-many communications paradigm. (20)

By having separated publishers and subscribers in this way they become independent of each other and do not need to be aware of one another to communicate. The intermediary becomes the “middleman” by receiving all the events and then being responsible for distributing them further to the subscribers, resulting in the publishers and subscribers becoming completely decoupled. This makes it possible to scale both the publishers and subscribers by adding more of them. By having them scale independently the eventual bottleneck in the communication channel becomes the intermediary. (26)

One difference between the publish-subscribe pattern and the message queue pattern is that there is no guarantee that an event will get delivered to a subscriber if the subscriber is not available. Depending on the implementation, a publish-subscribe does not have to save the events that are being sent. An implementation could persist all the events going through it, but it is not a requirement. (25) (27)

2.4.3 Event streams

Event streams are events that are produced by a producer in real-time. The events in the event stream represent a specific occurrence or change and are accompanied by metadata about what type of event it is and a timestamp. They operate with a publish-subscribe pattern and allow consumers to retrieve events in real-time. Event streams also persist the incoming events in a chronological order making it ideal for data analytics. (28)

2.5 Simple Queue Service (SQS)

AWS SQS is a message queue service that lets developers create message queues. (29) It is possible to publish and consume messages to and from the created queues via a software development kit (SDK) or from other AWS services like AWS Lambda, Amazon EC2, and Amazon S3. It has built-in error handling such that it is possible to configure the queue to automatically return a message to the queue when a consumer fails to process it. (30) This way, using SQS it is guaranteed that a message will be delivered to the queue.

SQS enables companies to decouple and scale their applications which can consist of any number of microservices, distributed systems, or other serverless applications provided by AWS. (31)

Creating and using an SQS queue is straightforward. The queue is created in the AWS cloud environment, to use the created queue in an application the developer will refer to the SQS queue name to publish or listen to the queue. (29)

2.6 Simple Notification Service (SNS)

Amazon Web Services Simple Notification Service (SNS) is a notification service that can be used to send notifications via different endpoints such as SMS, email, and HTTP. It works as a publish-subscribe system with no message retention meaning that messages only get sent to subscribers that are ready to receive them. (2)

SNS can be integrated with other cloud services that AWS provides, like SQS, AWS Lambda, and more. Message encryption, message filtering, and message tagging are all features that SNS supports. These features make SNS a viable messaging solution for companies spread out over many different industries. (32)

2.7 Combining SQS and SNS

SNS can be used to publish messages to multiple SQS queues, enabling one to many communications support that SQS alone does not provide and with message retention that SNS alone does not provide. This is called a fan-out pattern and is possible via a subscription from one or more SQS queues to the SNS topic. It is possible to subscribe from SQS queues, AWS Lambda functions, SMS, email, etc. (31)

2.8 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform made for handling large volumes of data in real-time. It was initially developed by LinkedIn in 2011 to handle the large volume of data being produced by their users' activity with

the website. It was later open-sourced and became an Apache Software Foundation project in 2012. (33)

At its core, Apache Kafka is a replicated, distributed, and persistent commit log. This commit log can be subscribed to so changes can be detected, and it can be published so that new data can be persisted and shared. Producing or consuming data is done to or from clusters which are called brokers. Brokers read and write to so-called topics that consist of one or more partitions in the underlying file system which automatically replicate it to have fault tolerance and high availability. Being distributed with the possibility of having many brokers makes Apache Kafka have a relatively high throughput and latency compared to other message brokers. (34) It is designed to handle large volumes of real-time data streams for many different use cases such as payments and financial transactions, sensor data from the Internet of Things devices, customer website interactions and orders, and many more.

Apache Kafka supports batching messages, grouping them, and sending them in batches instead of one by one. This minimizes excessive I/O operations, sending larger packets with less overhead, and is one of the reasons for the framework's scalability. (35)

2.9 Related Work

Some earlier works have been done regarding this topic, but there is a lack of comparative studies between Apache Kafka and AWS SNS/SQS. There have been studies performed on RabbitMQ, another popular message broker. One work has compared RabbitMQ to Apache Kafka while another has RabbitMQ to SQS.

Dobbelaere et al. (36) compares architectural differences Apache Kafka and RabbitMQ. This work uses an experimental setup to run tests on latency and throughput. The authors concluded that both brokers deliver low latency results with a mean of around 10ms. Regarding throughput, RabbitMQ outperformed Kafka in the most basic setup, with Kafka only using one partition. An increasing number of partitions in the Kafka topic significantly improved the performance of Kafka, showing its capacity to scale well. With RabbitMQ it was only possible to improve its performance moderately by increasing the producer channel.

Fredriksson et al. (37) compare message brokers in microservice architecture. The authors performed a literature and case study on RabbitMQ and AWS SQS. This work compared the brokers about message latency, ease of use, and maintainability. It was concluded that RabbitMQ has a much lower message latency than SQS, while SQS was easier to maintain and use. In this study only SQS was used without SNS, meaning only a one-to-one communication pattern was explored. Fredriksson et al. (37) further propose that for future work, one could take a closer look at one-to-many communications with SQS coupled with SNS, which is something this thesis will explore.

3 Method

In this chapter, the method used to answer the research question will be decided.

There are three different method alternatives considered for this work.

The first alternative would be to conduct a literature study on the topic. Relevant existing literature on both frameworks will be collected and summarized to be able to give clarity to the problem statement.

The second alternative would be to conduct interviews with companies that provide the different services. They can share their views on both frameworks, some history behind them, and how they differ from each other. Also interviewing developers that use both frameworks in their daily jobs could be interesting and give insight to different use cases of the frameworks. Using this method, the views of the creators and users of the frameworks will be heard.

The third alternative is to perform different tests on the frameworks to be able to evaluate their performance in different test cases. This can be done by developing two prototype applications, one for each framework, to be able to perform these tests.

The third method will be used since it makes it possible to perform different test cases on the two frameworks within the same environment, with minimal outside factors affecting the outcome, enabling a fair comparison of the two. This is something that is missing in the first two methods mentioned. It is also the method used in the previously mentioned works. (36) (37)

3.1 Setting up the AWS Environment

An SQS queue and a SNS topic will be set up in the AWS environment that Scania uses.

A Kafka cluster will be deployed in Kubernetes pods in the same AWS environment. In this cluster, a topic will be created that will be used as a message broker for running the tests.

Being hosted in the same AWS environment enables a fair comparison and analysis between the two.

3.2 Measuring Performance

There will be two test cases for measuring performance. The first test case is for measuring the latency of a single message, the second test case will be a stress test, producing messages simultaneously to the broker to be able to analyze how well it performs under heavier load. These are the same parameters that were used by Dobbelaere et al. (36) and Fredriksson et al. (37). This motivates using the same parameters in this work as well to be able to compare results.

3.2.1 Latency

The first test case will measure end-to-end latency. The producer will produce 10,000 messages in total and send one message in each 50-millisecond interval, preventing any bottleneck that may arise in the SQS queue or Kafka topic. Each message will be given a timestamp of when it was created and contain some string data. Each message has a size of 304 bytes. The subscribing consumer will then receive messages and calculate the latency of the message by using the timestamp for when it was created and the current time. When the latency has been calculated the result will be stored in memory. When all 10,000 messages have been processed, the consumer will write the benchmark data to a CSV file that can later be opened in Excel for analysis.

3.2.2 Load

The second test case will measure the end-to-end latency of the brokers while under a heavier load. To achieve a heavier load, the producer application will start up multiple threads that all produce to the same SQS queue or Kafka topic simultaneously. Each thread will send messages in bursts of 10 at a time, with an interval of 500 milliseconds in between each burst which will be repeated 10 times. The messages in this test case also have a timestamp for when it was created and some string data making each message 301 bytes in size.

The subscribing consumer will receive the messages and calculate the latency in the same way as the previously mentioned latency test, but messages will now be produced from multiple threads simultaneously, resulting in a higher load for the queue/topic.

This test will be performed several times with different number of threads each time, starting with 5 threads and then increasing the number of threads by 5 each time the test is run. Plotting the test data in charts will visualize how the brokers are performing relative to each other when being under the same amount of load.

3.3 Setting up the Microservices

To perform these tests, microservices using the two different message brokers will have to be developed. For each message broker, one producer microservice and one consumer microservice will be developed.

3.3.1 SNS/SQS Microservices

Two Spring Boot applications will be created, one to act as a producer and the other as a consumer.

The producer application will have two different REST endpoints which when called will trigger the respective test cases and send messages to the SNS topic.

The SQS queue will be subscribed to the SNS topic mentioned above which will result in all messages in the SNS topic being forwarded to the SQS queue.

The consumer application will be subscribed to the SQS queue and consume messages as they arrive in the queue.

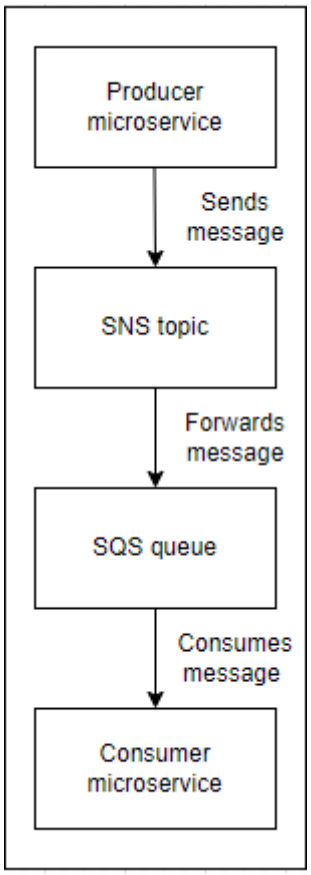

Figure 3.1: The experimental setup for testing SNS/SQS.

3.3.2 Kafka Microservices

Like with the SNS/SQS setup, two applications will be created with Spring Boot.

The producer application will have two different REST endpoints which when called will trigger the respective test cases and send messages to the Kafka topic.

The consumer application will be a subscriber to the Kafka topic, and as soon as a new message has entered the topic, it will consume the message.

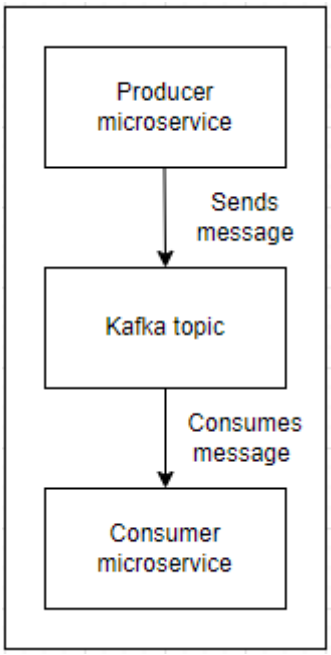

Figure 3.2: The experimental setup for testing Kafka.

3.4 Summarize the data

When the data from the different test cases have been saved, it will be opened in an Excel sheet for further analysis.

For the latency test case, the average message latency will be calculated along with a confidence interval. Graphs of the test results will also be generated.

For the load test case, the average message latency will be calculated for the different numbers of threads used and produced and a graph will be generated to showcase the result.

4 Results

This chapter will present the results generated from the different test cases. The first section will present results from the latency test case, while the following section will present the results from the load test case.

4.1 Latency test case

The results from the latency test case show that the average message latency when using SNS/SQS was 106.4 ms and when using Kafka was 20.6 ms, as shown in Table 4.1. The confidence interval (99%) for Kafka was 0.6 ms while for SNS/SQS 3.3 ms, meaning 99% of messages were within 0.6 ms and 3.3 ms of the average value for Kafka and SNS/SQS respectively as presented in Table 4.1.

Table 4.1: Average message latency and confidence interval after sending and receiving 10,000 messages using SNS/SQS and Kafka.

System	Requests	Average latency (ms)	Confidence interval (99%) (ms)
SNS/SQS	10 000	106.4	3.3
Kafka	10 000	20.6	0.6

Figure 4.1 and 4.2 shows the message latency of SNS/SQS and Kafka when sending one message every 50 ms for a total of 10,000 messages.

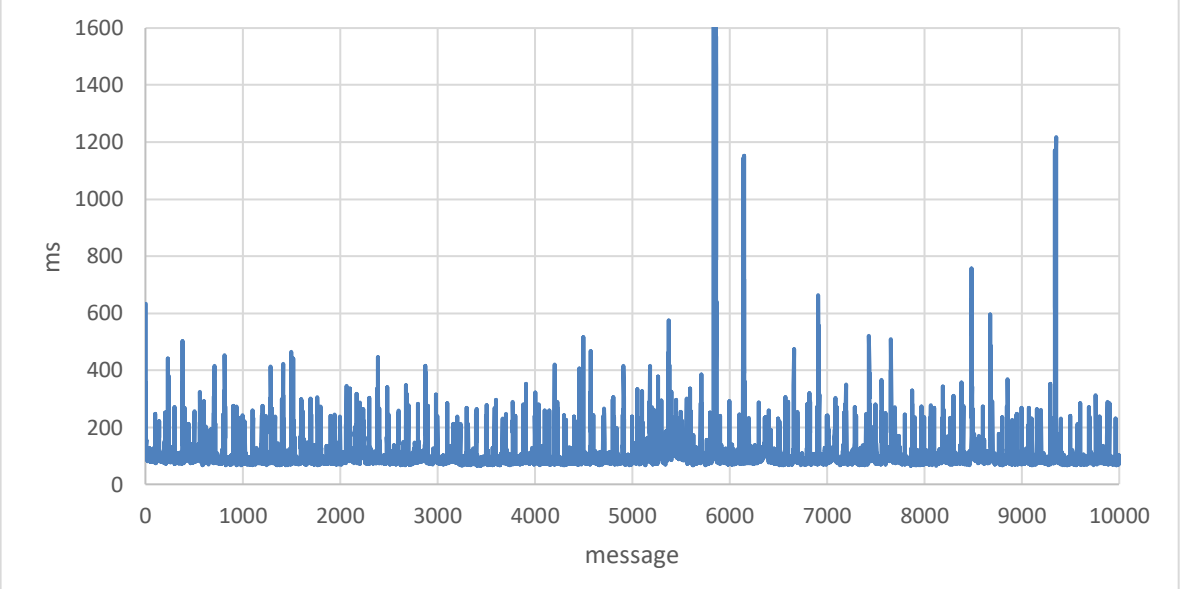

Figure 4.1: Message latency for SNS/SQS.

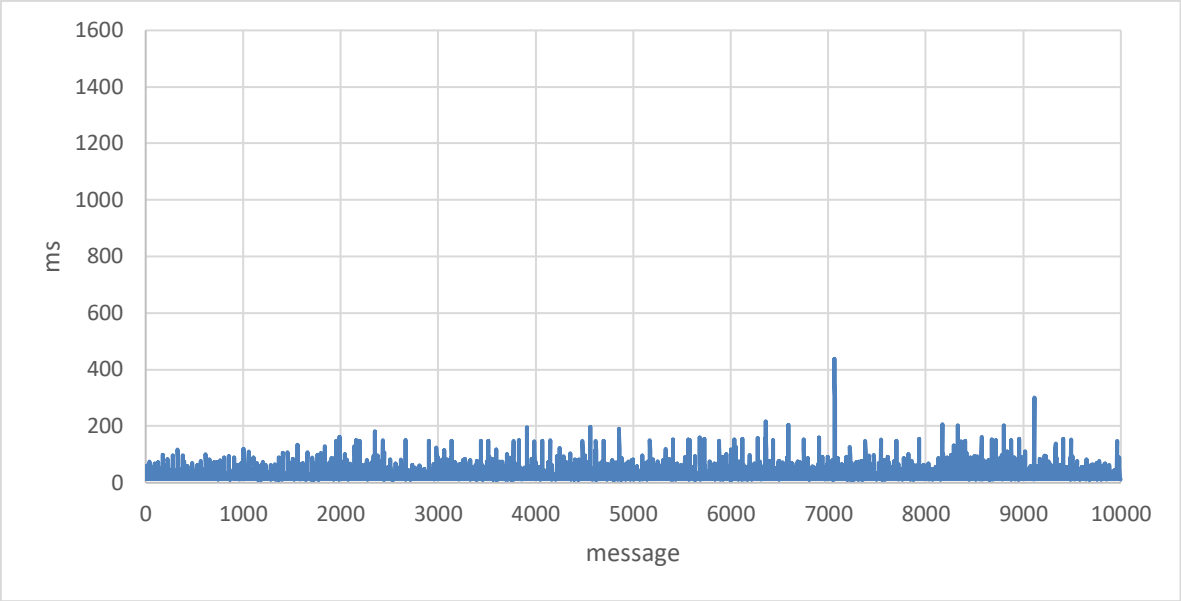

Figure 4.2: Message latency for Kafka.

4.2 Load test case

Figure 4.3 – 4.21 displays the message latency for SNS/SQS and Apache Kafka for the load test case. Two charts will be displayed for each test case, one showing the latency value for each message. The other shows the latency for each message in a logarithmic scale with base 10. It was chosen to display two graphs in different scales since the latency between SNS/SQS and Kafka differs by a great magnitude for some parameters. The logarithmic scale with base 10 complements the normal scale graph, displaying the results of both frameworks relative to each other.

While performing testing on SNS/SQS microservices, it was found that it was not possible to produce SNS/SQS from more than 39 threads simultaneously.

4.2.1 100 messages per second

Figure 4.3 and 4.4 shows the message latency for SNS/SQS and Kafka. Using 5 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 500 messages being sent in groups of 50 every 500 ms.

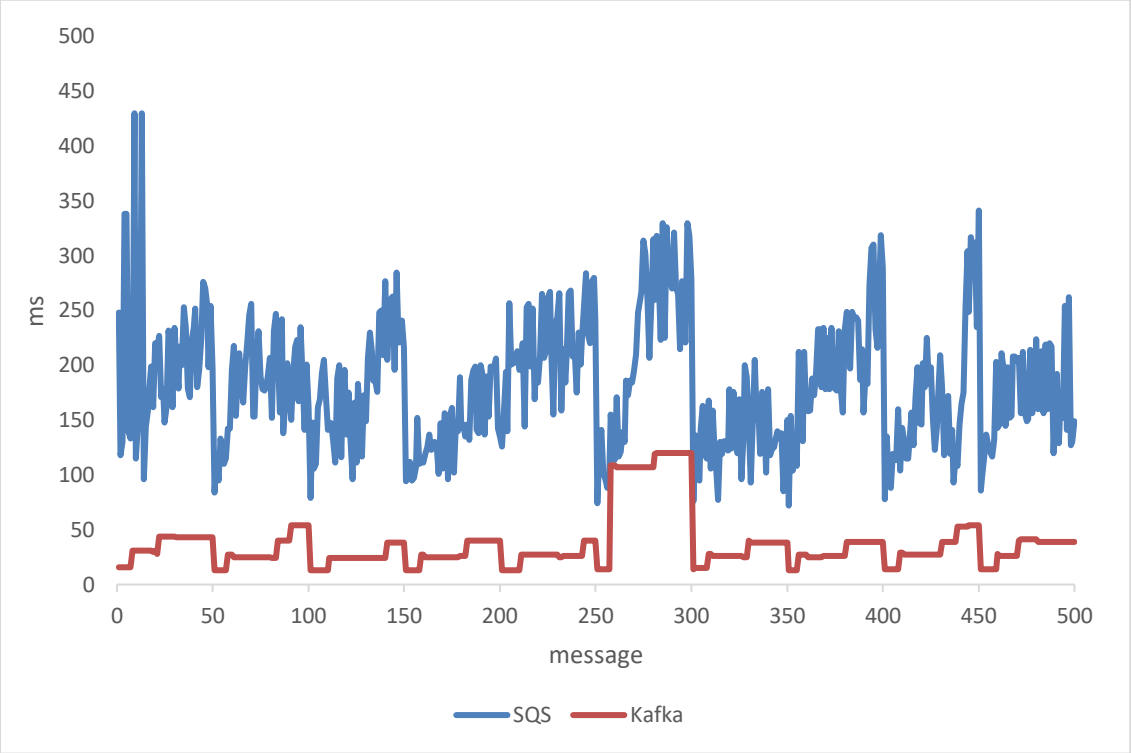

Figure 4.3: Message latency for SNS/SQS and Kafka when messages are sent in groups of 50 every 500 ms.

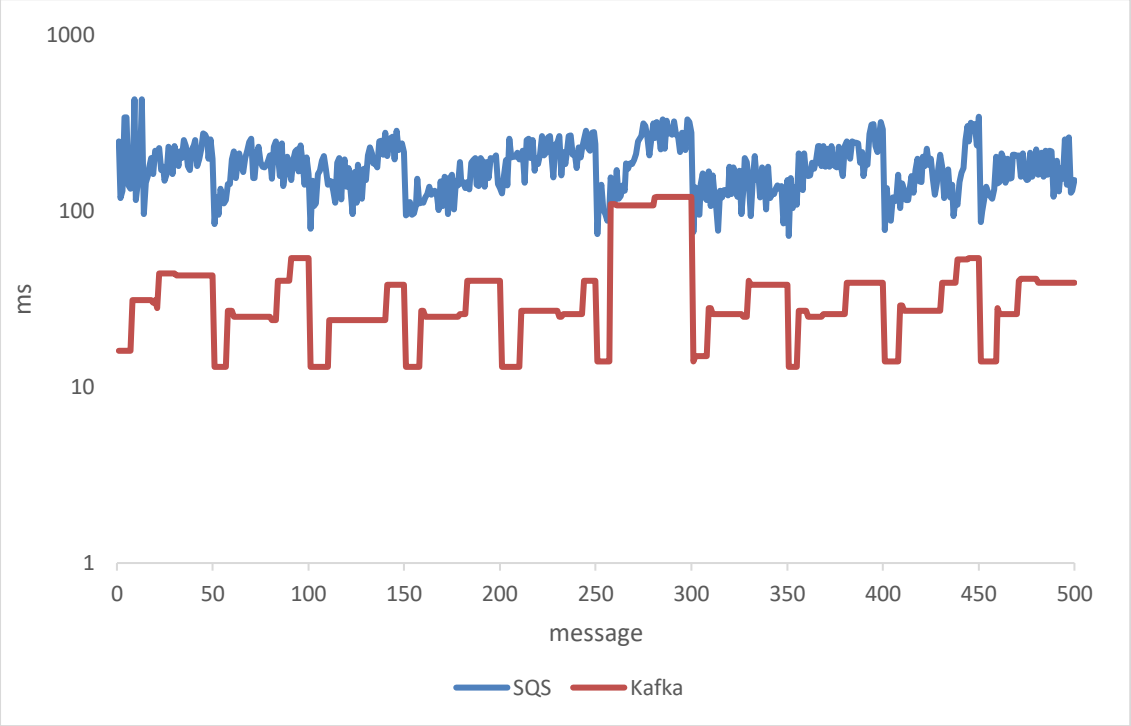

Figure 4.4: Message latency for SNS/SQS and Kafka when messages are sent in groups of 50 every 500 ms in a logarithmic scale.

4.2.2 200 messages per second

Figure 4.5 and 4.6 shows the message latency for SNS/SQS and Kafka. Using 10 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 1000 messages being sent in groups of 100 every 500 ms.

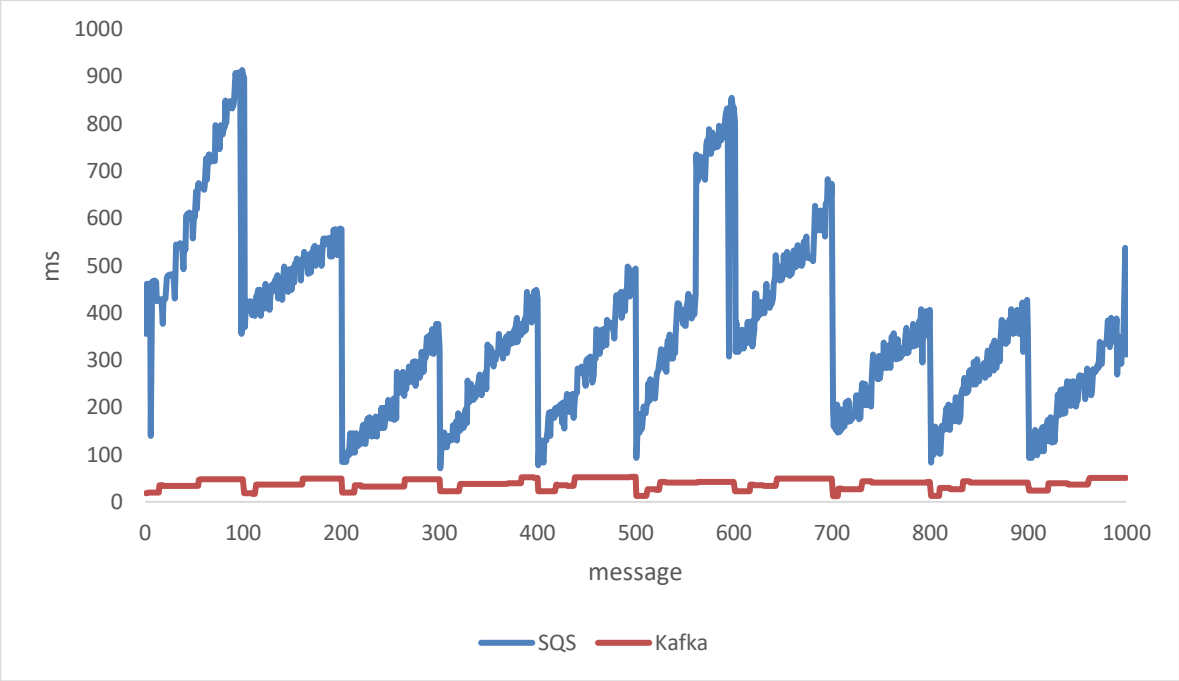

Figure 4.5: Message latency for SNS/SQS and Kafka when messages are sent in groups of 100 every 500 ms.

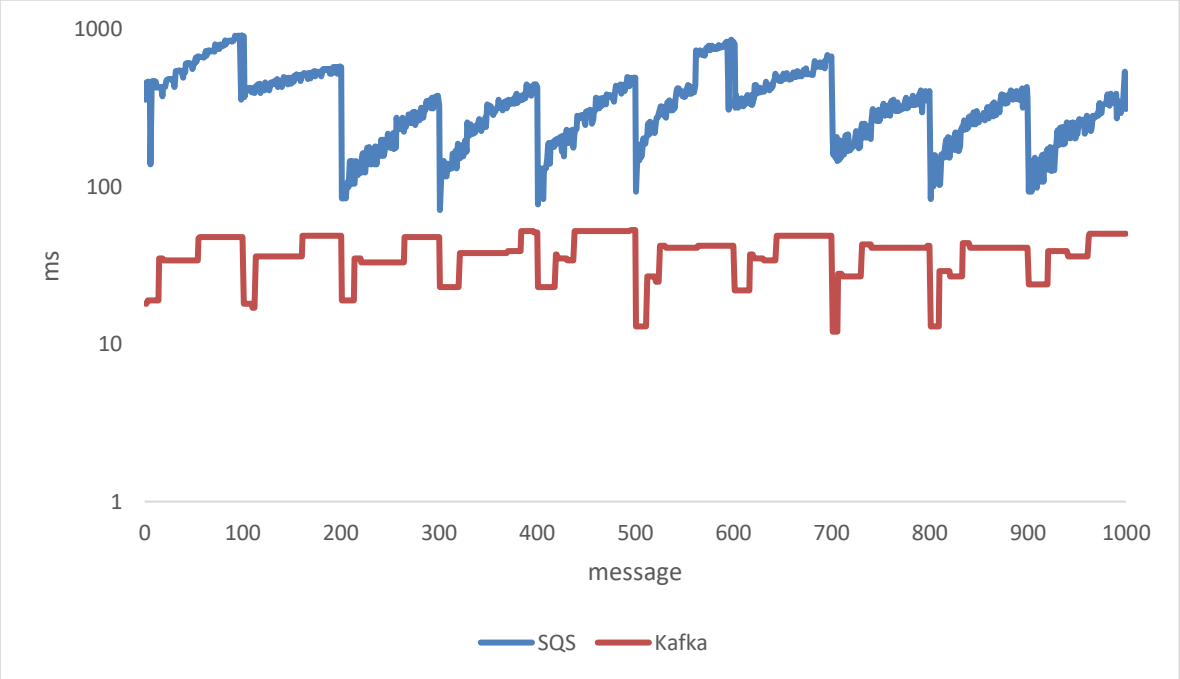

Figure 4.6: Message latency for SNS/SQS and Kafka when messages are sent in groups of 100 every 500 ms in a logarithmic scale.

4.2.3 300 messages per second

Figure 4.7 and 4.8 shows the message latency for SNS/SQS and Kafka. Using 15 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 1500 messages being sent in groups of 150 every 500 ms.

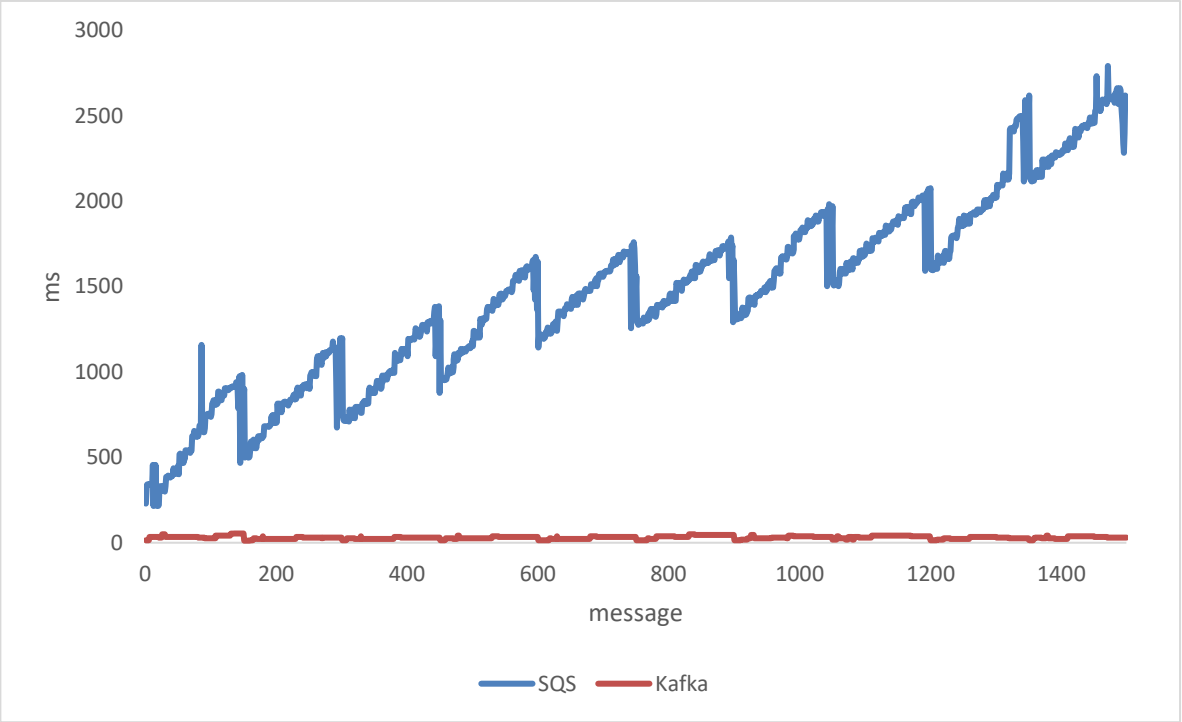

Figure 4.7: Message latency for SNS/SQS and Kafka when messages are sent in groups of 150 every 500 ms.

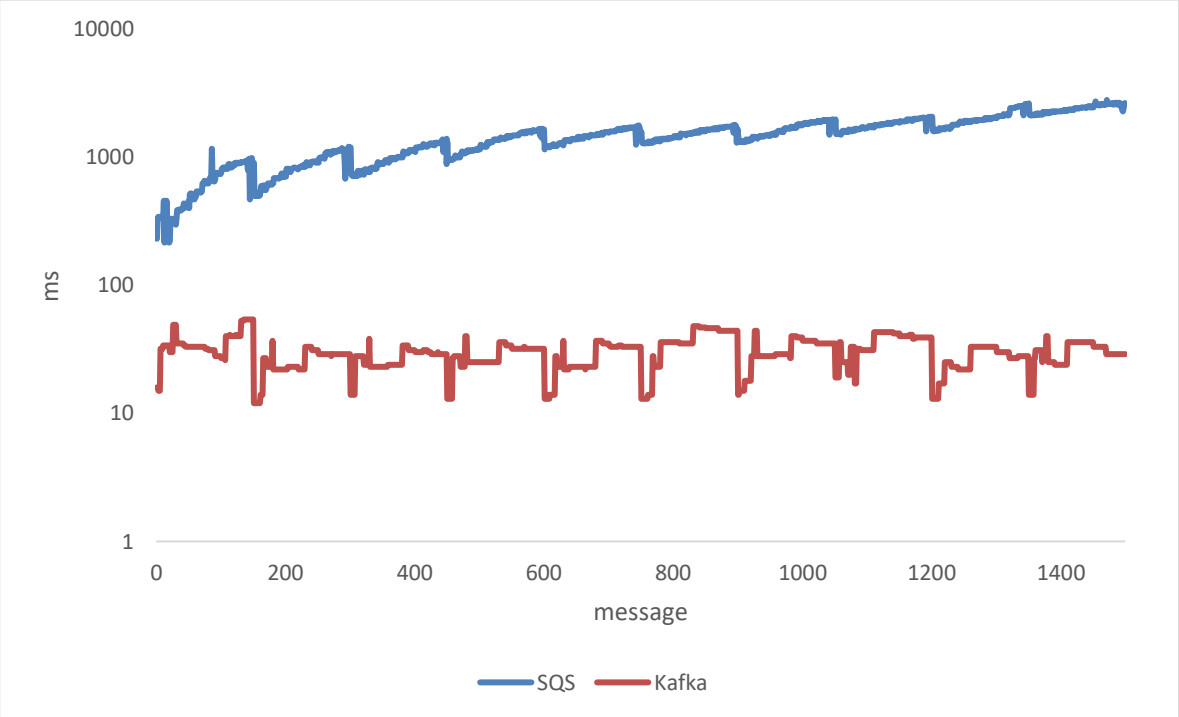

Figure 4.7: Message latency for SNS/SQS and Kafka when messages are sent in groups of 150 every 500 ms in a logarithmic scale.

4.2.4 400 messages per second

Figure 4.9 and 4.10 shows the message latency for SNS/SQS and Kafka. Using 20 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 2000 messages being sent in groups of 200 every 500 ms.

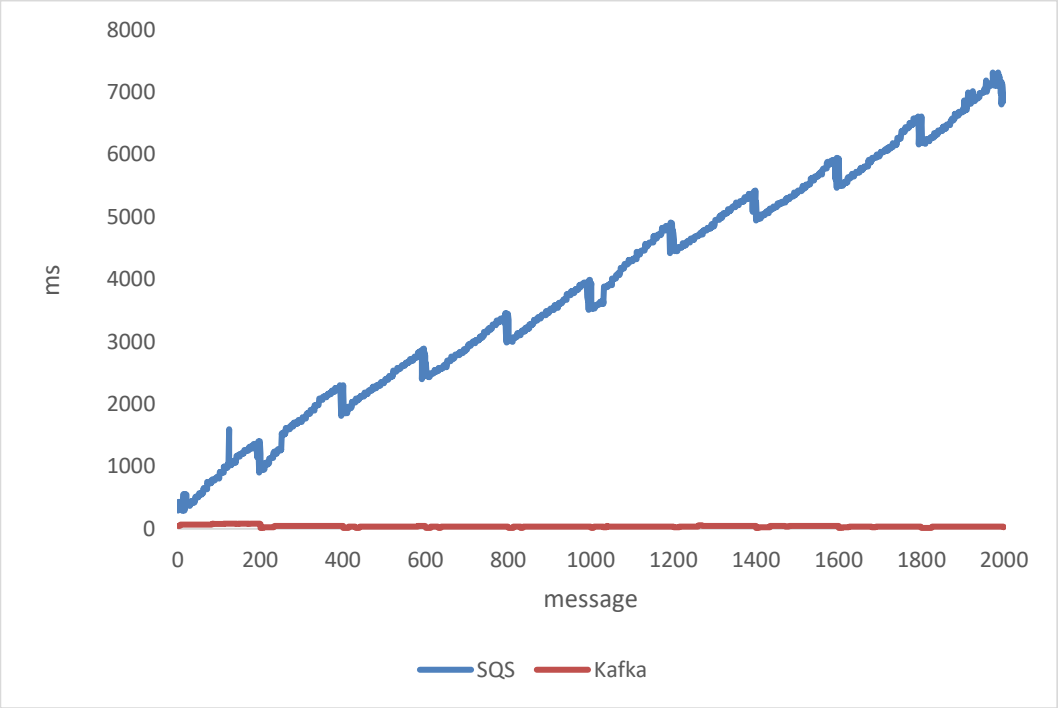

Figure 4.8: Message latency for SNS/SQS and Kafka when messages are sent in groups of 200 every 500 ms.

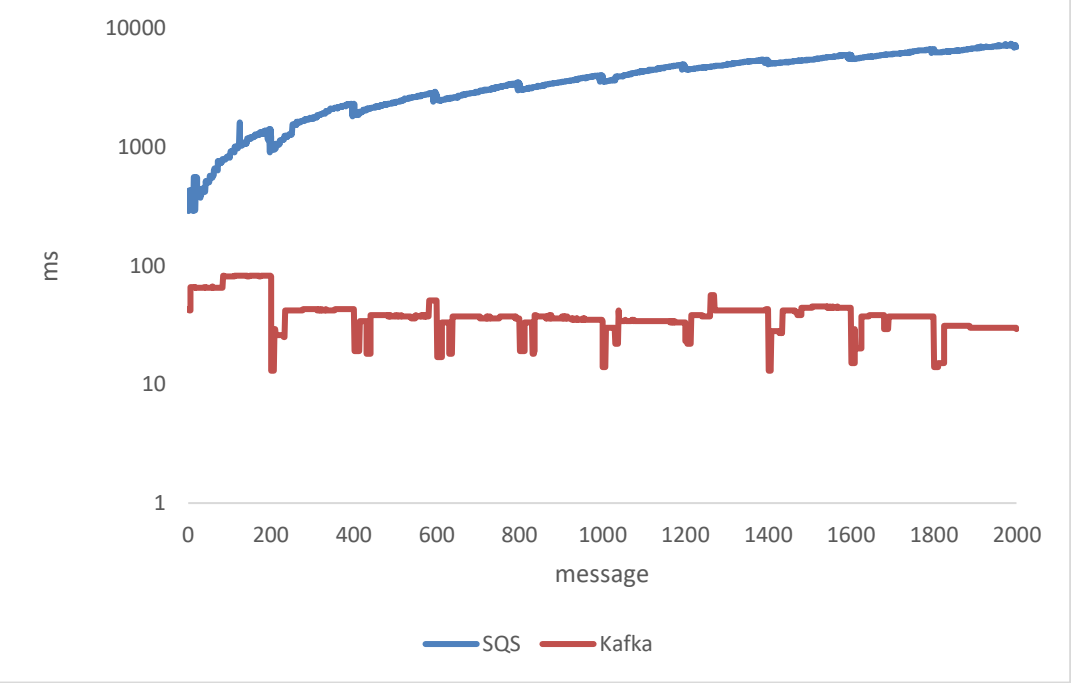

Figure 4.9: Message latency for SNS/SQS and Kafka when messages are sent in groups of 200 every 500 ms in a logarithmic scale.

4.2.5 500 messages per second

Figure 4.11 and 4.12 shows the message latency for SNS/SQS and Kafka. Using 25 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 2500 messages being sent in groups of 250 every 500 ms.

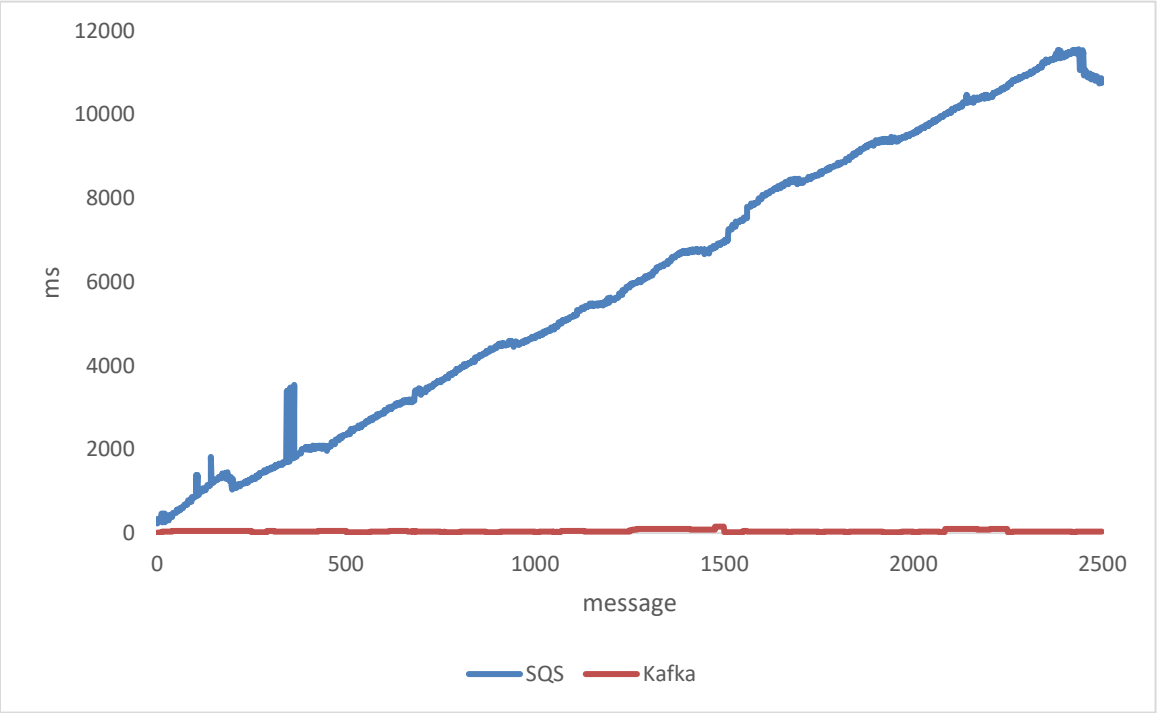

Figure 4.10: Message latency for SNS/SQS and Kafka when messages are sent in groups of 250 every 500 ms.

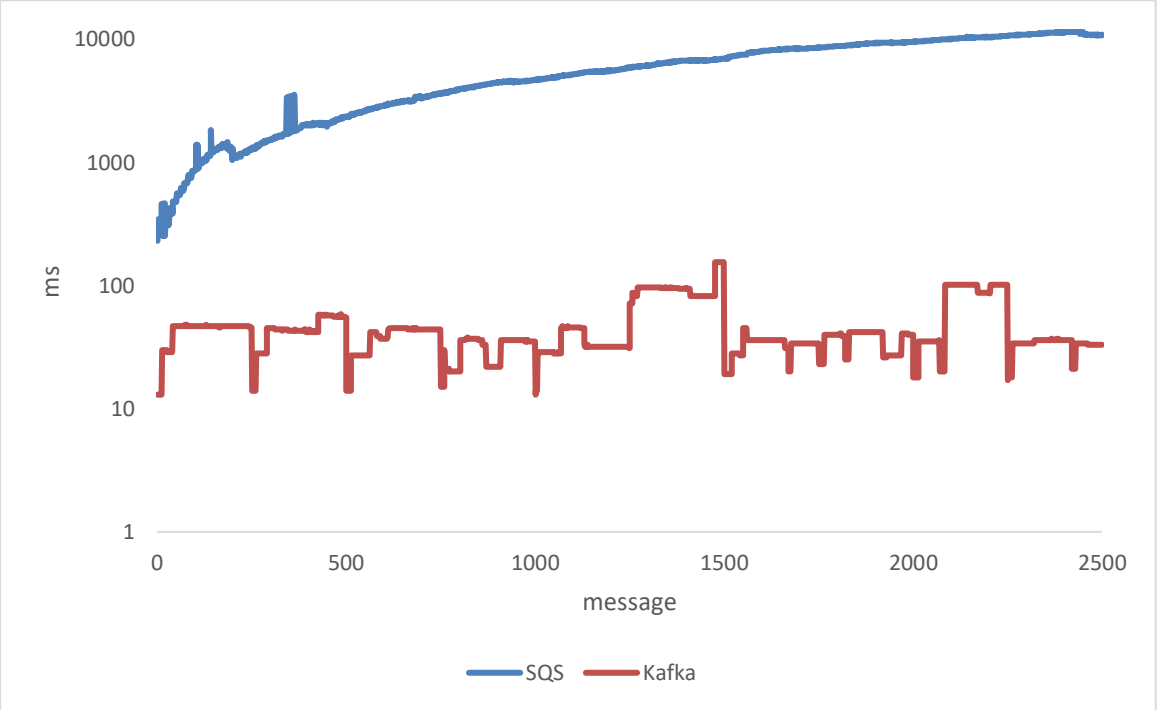

Figure 4.11: Message latency for SNS/SQS and Kafka when messages are sent in groups of 250 every 500 ms in a logarithmic scale.

4.2.6 600 messages per second

Figure 4.13 and 4.14 shows the message latency for SNS/SQS and Kafka. Using 30 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 3000 messages being sent in groups of 300 every 500 ms.

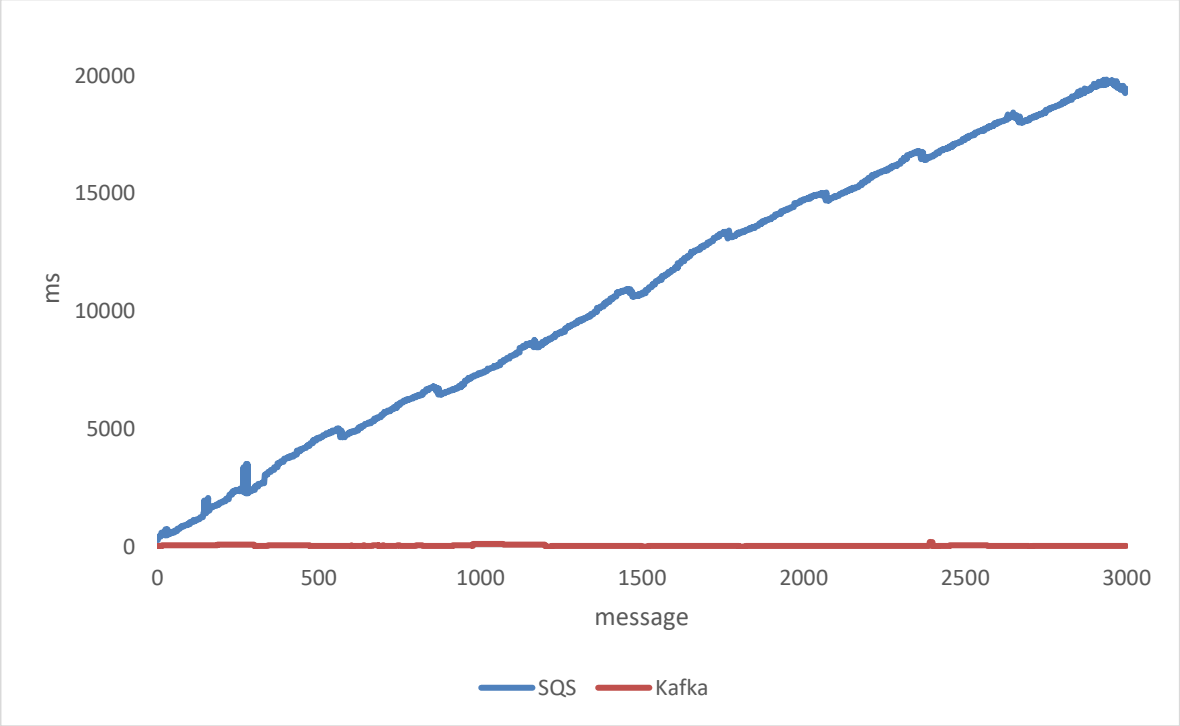

Figure 4.13: Message latency for SNS/SQS and Kafka when messages are sent in groups of 300 every 500 ms.

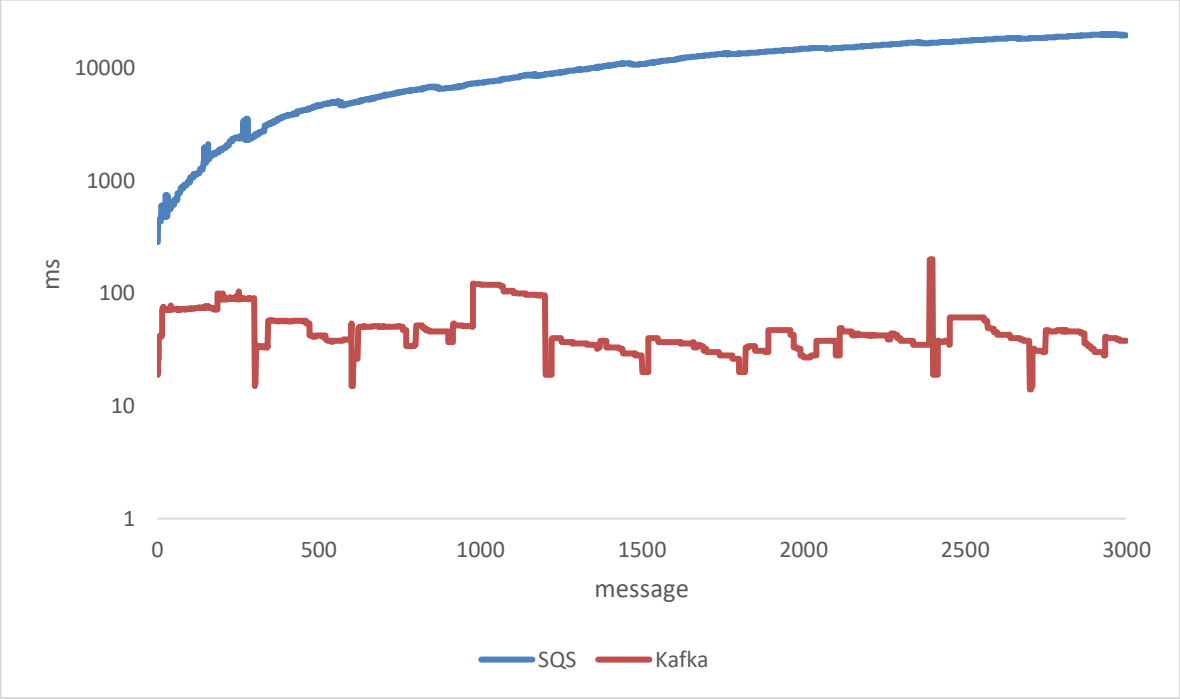

Figure 4.12: Message latency for SNS/SQS and Kafka when messages are sent in groups of 300 every 500 ms in a logarithmic scale.

4.2.7 700 messages per second

Figure 4.15 and 4.16 shows the message latency for SNS/SQS and Kafka. Using 35 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 3500 messages being in groups of 350 every 500 ms.

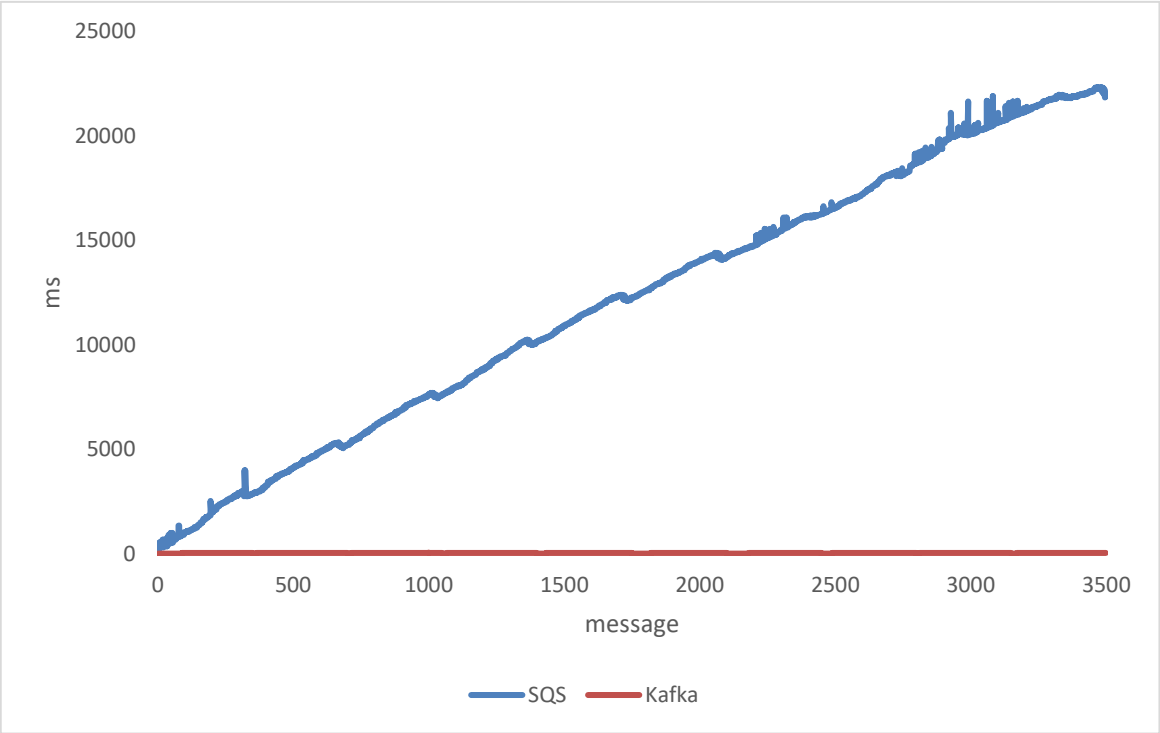

Figure 4.15: Message latency for SNS/SQS and Kafka when messages are sent in groups of 350 every 500 ms.

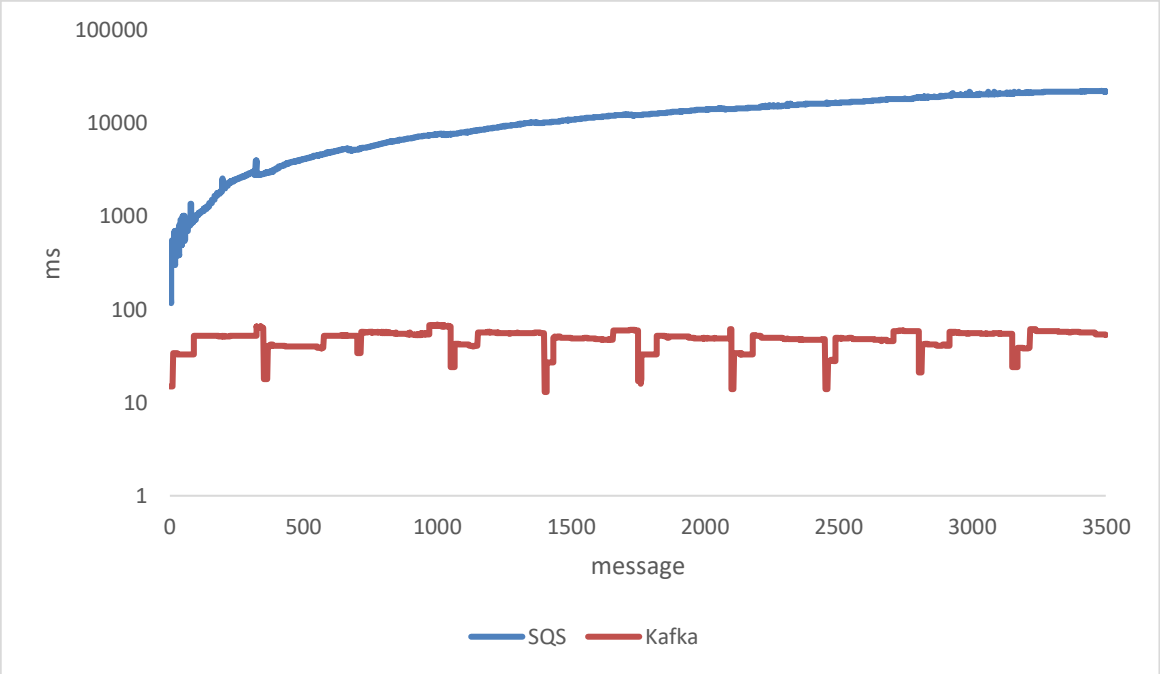

Figure 4.14: Message latency for SNS/SQS and Kafka when messages are sent in groups of 350 every 500 ms in a logarithmic scale.

4.2.8 780 messages per second

Figure 4.17 and 4.18 shows the message latency for SNS/SQS and Kafka. Using 39 threads, each producing 10 messages every 500 ms for 5 seconds. Resulting in 3900 messages being in groups of 390 every 500 ms.

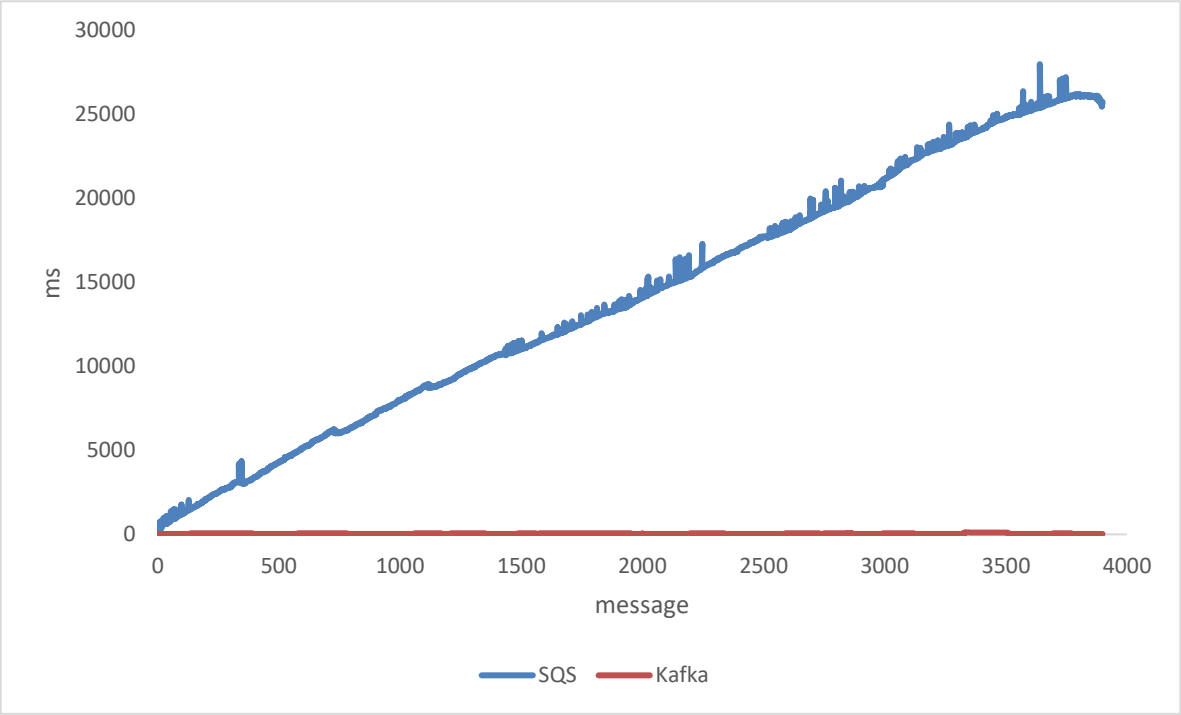

Figure 4.17: Message latency for SNS/SQS and Kafka when messages are sent in groups of 390 every 500 ms.

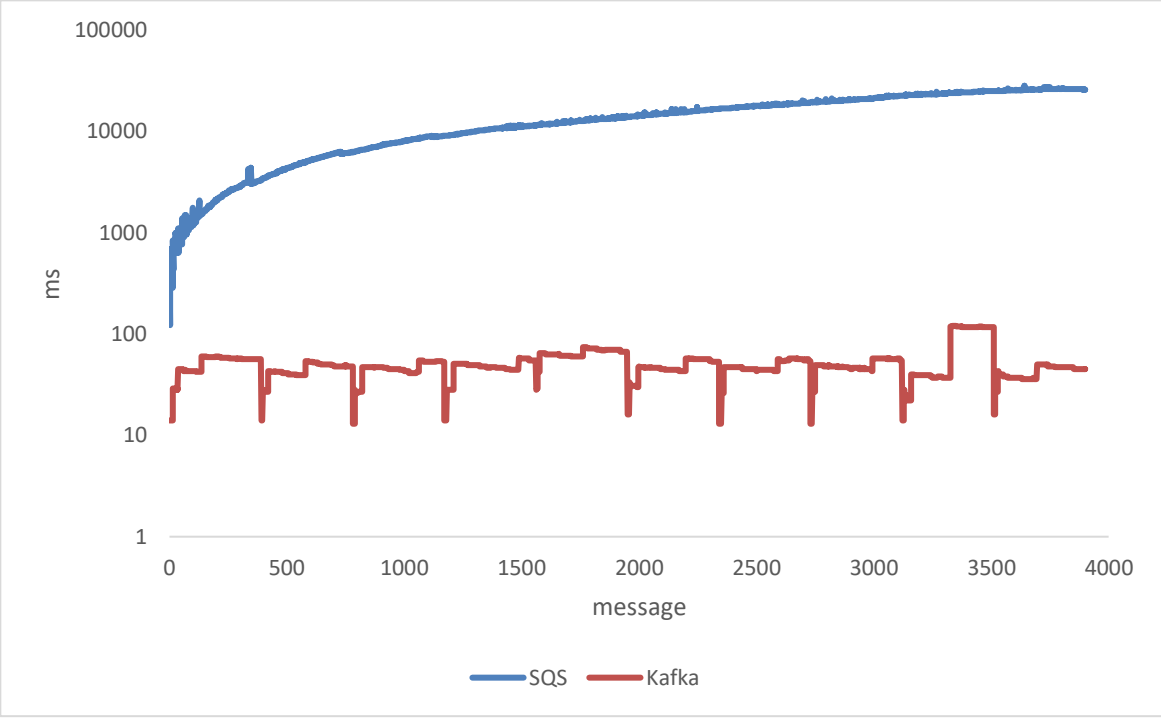

Figure 4.16: Message latency for SNS/SQS and Kafka when messages are sent in groups of 390 every 500 ms in a logarithmic scale.

4.2.9 Average values

Figure 4.19-4.20 shows the average response time for each load test case using SNS/SQS and Kafka. Figure 4.21 shows the average response for both SNS/SQS and Kafka on a logarithmic scale.

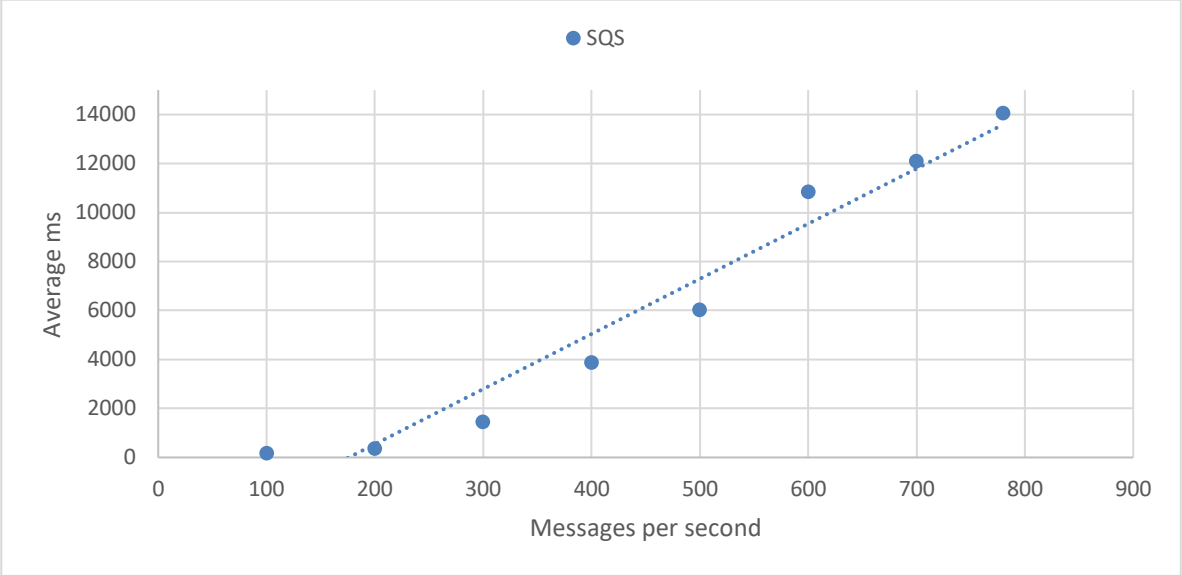

Figure 4.18: Average response time for each load test case using SNS/SQS.

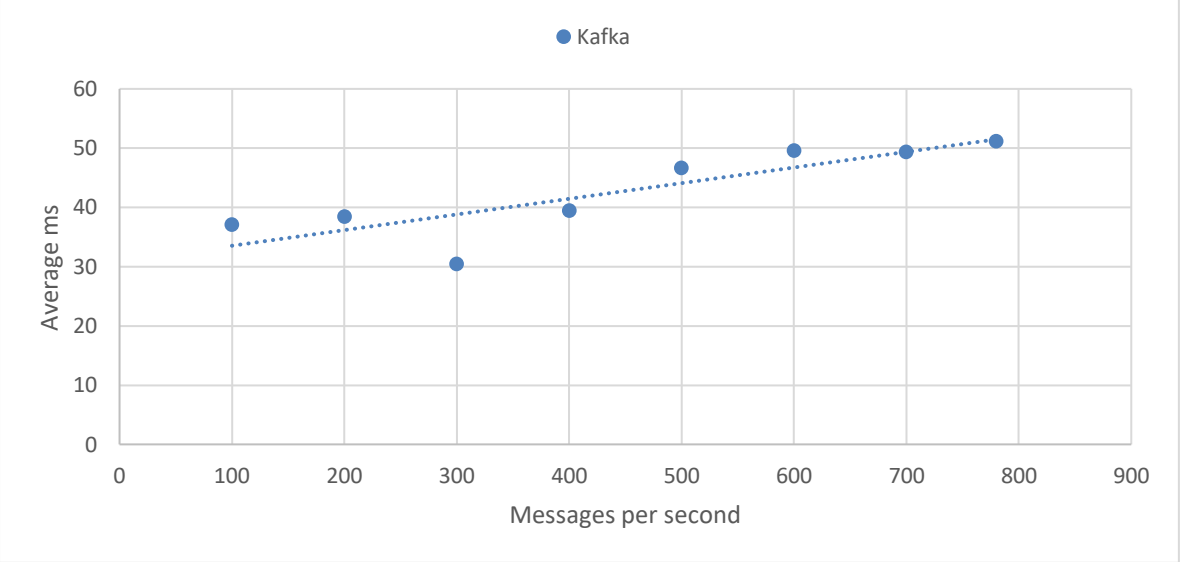

Figure 4.19: Average response time for each load test case using Kafka.

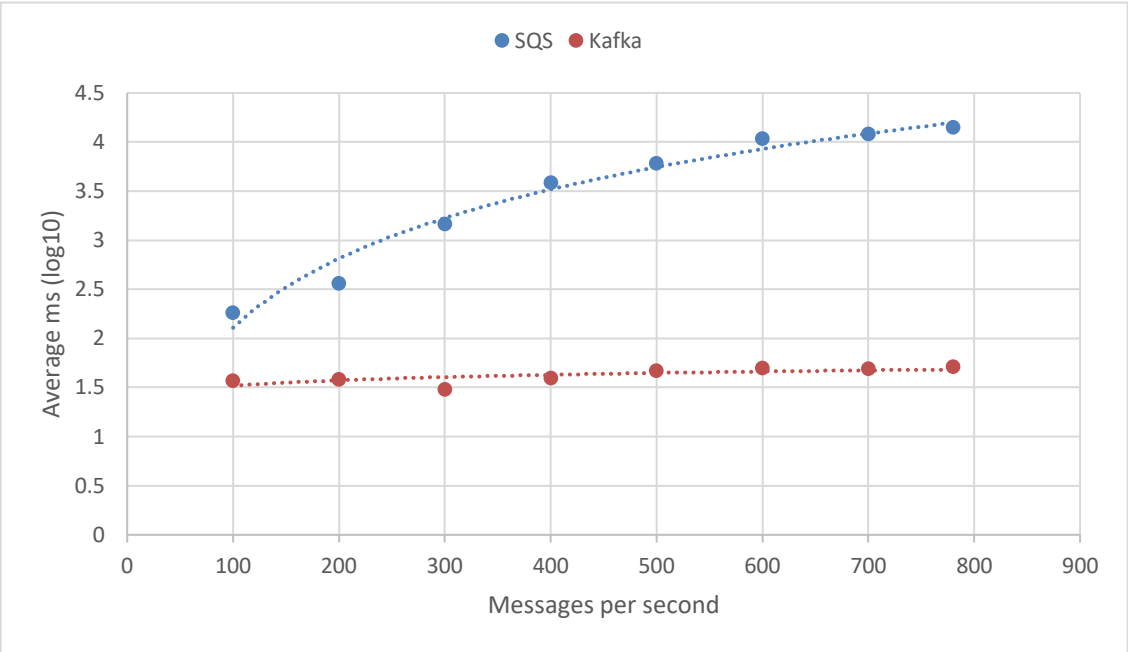

Figure 4.20: Average response time for each load test case using SNS/SQS and Kafka on a logarithmic scale.

5 Analysis and Discussion

In this chapter, the results presented in Chapter 4 will be discussed and analyzed. Firstly, the performance of the frameworks will be analyzed, and the implications of the results will be discussed, both in the case of single and multiple producers. These discussions will then be of use to discuss optimal use cases for both SNS/SQS and Kafka. Following this, the chosen research methodology will be evaluated and lastly, sustainability implications will be considered.

5.1 Latency test case

The results obtained from the latency tests of Kafka and SNS/SQS reveal a substantial performance difference between the two messaging systems, presented in section 4.1. Kafka displays an average latency of 20ms, outperforming SNS/SQS with an average latency of 106ms.

The lower latency observed in Kafka can be explained by its distributed architecture and principles of design. The explanation for the observed higher latency for SNS/SQS could be its high scalability and simplified management. This may contribute to it having higher latency due to the introduction of additional overhead that comes with being a managed service.

Fredriksson et al. (37) performed similar experiments with 100 messages, 1 message per second, the resulting average latency for SQS was 144ms, with a confidence interval of 95%, which is close to the 106ms that this work has provided. The difference in latency could be due to small differences in methodology or setup. For instance, this thesis examined SNS coupled with SQS while Fredriksson et al. (37) examined SQS exclusively.

The latency analysis shows Kafka is the superior framework when it comes to delivering low-latency messaging.

5.2 Load test case

Producing messages from multiple threads gives insights into how Kafka and SNS/SQS behave under increased load, showing the framework's capacity to handle messages while under increased load. Results from these tests, presented in section 4.2, show that there are compelling differences between the two frameworks when subjected to higher loads.

Opening multiple threads and producing messages concurrently shows that Kafka and SNS/SQS exhibit different behaviors as the load increases.

As the pace of production of messages increased, Kafka displayed a great ability to handle the increasing load without showing any notable build-up of bottlenecks. Message latency was kept on a stable level which strengthens Kafka's reputation as a robust message broker able to handle high volumes of data streams without affecting latency.

SNS/SQS however, presents a contrasting picture. As the pace of production of messages increased, a noteworthy bottleneck was built up in the queue, which caused an increase in latency. This bottleneck and increase in latency indicate limitations of the framework regarding managing heightened load.

The findings from this test case show the impact of the fundamental architectural differences between Kafka and SNS/SQS. It is also in line with Dobbelaere et al. (36), who found that increasing the number of partitions in a Kafka topic significantly improves its scalability.

The results of the load test case demonstrate that while both frameworks may perform similarly under normal conditions, their performance under increased load is significantly different.

5.3 Use Cases

The insights gathered from sections 5.1 and 5.2 act as guidance for selecting the best-fitting framework for specific use cases. Kafka displayed consistent low latency and efficient handling of high volumes of data. This gives reason to believe it is best suited for latency-sensitive applications that require real-time data processing, such as Internet of Things systems or financial trading platforms.

Dobbelaere et al. (36) come to a similar conclusion that the best-suited use cases for Kafka would be as a medium for a scalable publish/subscribe messaging system in need of high throughput or stream processing applications.

SNS/SQS, because of its simplicity of managed services, would better suit scenarios where ease of use takes priority over low latency. Applications such as order processing systems or notifications in non-real-time contexts are where SNS/SQS can find its place.

5.4 Evaluation of the Research Method

There were alternative research methods for this work, like conducting a literature study or interviewing people working in the industry and get to know their views and experiences working with the frameworks. The authors opted to use an experimental setup and run performance tests on the frameworks. As mentioned in Chapter 3, the reason performance tests were chosen, was that they, unlike the alternative methods, generate fair and valuable test data regarding latency and load, which are essential factors in comparing messaging systems. It was also the method of choice for two related works, using the same method for this work made it possible to compare results and conclusions. This method, although giving valuable benchmark data, has some limitations and could be improved, either by complementing it with a literature study and interviews, or by investigating even more technical parameters.

In this work, the same message size was used for each test performed. The reason for this was that this work was limited to studying only the latency and load of messages. However, testing with different message sizes could have provided additional depth to the study, making it possible to compare the throughput of both frameworks. Another weakness in the method was that all tests were run on the same device and

network. In a real-life scenario, these frameworks would be used on different devices with different hardware and networks, which gives reason to test on different hardware and devices as well. This would provide more realistic and reliable test data.

A combination of performance testing with interviews with developers and providers of the services could offer a comparison between Kafka and SNS/SQS that is more nuanced and applicable to real-world scenarios. Relying on performance testing alone might give an analysis that is too narrow. Holding complementary interviews may shed light on other important aspects that could make a difference in the comparison. It could also give rise to new questions and areas of investigation that weren't thought of beforehand.

5.5 Sustainability

Kafka does better than AWS SNS/SQS with handling larger amounts of data which may lead to a reduced energy consumption needed to achieve a certain level of throughput. In larger applications with more traffic, this could lead to Apache Kafka demanding less energy to run and cool down.

From an economic perspective, optimized utilization of resources could help companies reduce operational costs by managing data streams in a more efficiently. The latency results of Kafka show that the framework has the potential for this. Meanwhile, companies that are dependent on low latency message handling, and are using SNS/SQS for this case, could have their revenue negatively impacted if the quality of service provided doesn't meet the standards of the customer.

Socially, lower latency can contribute to a better user experience, which is something Kafka can provide reliably. SNS/SQS has latency challenges in some cases, and this could lead to a worse user experience, causing lagging applications that frustrate users and could negatively impact user trust.

There were no ethical aspects identified in this thesis.

6 Conclusions

This thesis set out to evaluate the differences between Kafka and SNS/SQS as a medium for asynchronous communication in event-based systems. Through experimental testing of latency and load handling of the two frameworks, a comparison could be made. With the support of the data generated from the tests run and analysis of the tests, optimal use cases for each framework could then be identified.

The results show that Kafka is the framework most capable of delivering low-latency message handling. Furthermore, it was shown that as the pace of message production was increased, the load on the frameworks was heightened, and Kafka performed significantly better than SNS/SQS. Looking at optimal use cases for each framework, it was decided Kafka was best suited for real-time data streaming and low latency use cases, while SNS/SQS suits use cases where ease of use is of higher priority than low latency.

6.1 Future Work

The research methodology that this thesis employed, was predominantly focused on performance testing and bears limitations. The inclusion of qualitative insights via interviews with industry experts could contribute to the topic.

Another area that could be investigated further is the user experience from a developer's perspective. This thesis has briefly mentioned the usability of SNS/SQS and that is something that can be investigated further both for SNS/SQS and Kafka.

There are several different vendors for Apache Kafka and a comparative study between these could be considered for future work as well. The authors would propose a comparison between the two Fully-Managed Clouds, Confluent Cloud, and Amazon MSK, with a focus on the performance and pricing of the two.

References

1. Amazon Web Services. [Online]. [cited 2023 8 10]. Available from: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/quotas-messages.html

2. Amazon Web Services. [Online]. [cited 2023 8 10]. Available from: https://docs.aws.amazon.com/sns/latest/dg/welcome-features.html

3. Nanda S. A Survey on Virtualization Technologies. New York: Stony Brook, Department of Computer Science.

4. Qian L, Luo Z, Du Y, Guo L. Cloud Computing: An Overview Berlin: Springer, Berlin, Heidelberg; 2009.

5. Liu S, Chan FTS, Yang J, Niu B. Understanding the effect of cloud computing on organizational agility: An empirical examination. International Journal of Information Management. 2018 December: p. 98-111.

6. Bibi S, Katsaros D, Bozanis P. Business Application Acquisition: On-Premise or SaaS-Based Solutions? IEEE Software. 2012 Apr; 29(3).

7. Rostami A. Cloud Service Models - IaaS, PaaS, and SaaS. University of Alberta, Department of Electrical and Computer Engineering; 2021 Oct.

8. Waschke M. How Clouds Hold IT Together Integrating Architecture with Cloud Deployment. 1st ed. Berkeley, CA: Apress; 2015.

9. Gupta A. 2010 2nd International Conference on Computer Technology and Development. In Cloud computing growing interest and related concerns; 2010. p. 462-465.

10. Li A, Yang X, Kandula S, Zhang M. Comparing Public-Cloud Providers. IEEE Internet Computing. 2011: p. 50-53.

11. Nadeem MA. Cloud Computing: Security Issues and Challenges. Journal of Wireless Communications. 2016: p. 10-15.

12. Kumar TA. Privacy and security challenges in cloud computing : a holistic approach. 1st ed. Boca Raton, Florida: CRC Press; 2022.

13. Halfond WGJ, Viegas J, Orso A. A Classification of SQL Injection Attacks and Countermeasures. Georgia Institute of Technology; 2006.

14. Amin R, Vadlamudi S, Rahaman MM. Opportunities and Challenges of Data Migration in Cloud. Engineering International. 2021: p. 41-50.

15. Shirer M. International Data Corporation. [Online].; 2023 [cited 2023 8 10]. Available from: https://www.idc.com/getdoc.jsp?containerId=prUS51009523

16. Synergy Research Group. [Online].; 2023 [cited 2023 8 10]. Available from: https://www.srgresearch.com/articles/q1-cloud-spending-grows-by-over-10-billion-from-2022-the-big-three-account-for-65-of-the-total

17. Rimol M. Gartner. [Online].; 2022 [cited 2023 8 10]. Available from: https://www.gartner.com/en/newsroom/press-releases/2022-06-02-gartner-says-worldwide-iaas-public-cloud-services-market-grew-41-percent-in-2021

18. Amazon Web Services. [Online]. [cited 2023 08 10]. Available from: https://aws.amazon.com/machine-learning/

19. Mukherjee S. Benefits of AWS in Modern Cloud. Zenodo. 2019.

20. Coulouris G, Dollimore J, Kindberg T, Blair G. Distributed Systems: Concepts and Design. 5th ed. Addison-Wesley , editor. Harlow, Essex: Pearson Education; 2012.

21. Cristian F. Synchronous and Asynchronous. Communications of the ACM. 1996 Apr: p. 88-97.

22. Budau V, Bernard G. Ninth International Conference on Parallel and Distributed Systems, 2002. Proceedings. In Synchronous/asynchronous switch for a dynamic choice of communication model in distributed systems; 2002; Taiwan. p. 97-102.

23. Korab J. Understanding Message Brokers : O'Reilly Media, Inc.; 2017.

24. Thein KMM. Apache Kafka: Next Generation Distributed Messaging System. International Journal of Scientific Engineering and Technology Research. 2014 Dec: p. 9478-9483.

25. Redkar A, Rabold K, Costall R, Boyd S, Walzer C. Pro MSMQ Microsoft Message Queue Programming. 1st ed. Berkely, CA: Apress; 2004.

26. Baldoni R, Querzoni L, Virgillito A. Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey. Rome: Universit`a di Roma “La Sapienza”, Dipartimento di Informatica e Sistemistica; 2009.

27. Richerzhagen B. Mechanism Transitions in Publish/Subscribe Systems Adaptive Event Brokering for Location-based Mobile Social Applications. 1st ed. Cham: Springer International Publishing; 2019.

28. Crettaz V, Dean A. Event Streams in Action: Manning Publications; 2019.

29. Amazon Web Services. [Online]. [cited 2023 8 10]. Available from: https://aws.amazon.com/sqs/

30. Yoon H, Gavrilovska A, Schwan K, Donahue J. 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012). In Interactive Use of Cloud Services: Amazon SQS and S3; 2012; Ottawa. p. 523-530.

31. Pogiatzis A, Samakovitis G. An Event-Driven Serverless ETL Pipeline on AWS. Applied Sciences. 2021.

32. Amazon Web Services. [Online]. [cited 2023 08 07]. Available from: https://aws.amazon.com/sns/

33. Confluent. [Online]. [cited 2023 08 10]. Available from: https://www.confluent.io/what-is-apache-kafka/

34. Confluent. [Online]. [cited 2023 08 10]. Available from: https://www.confluent.io/blog/kafka-fastest-messaging-system/

35. Confluent. [Online]. [cited 2023 08 10]. Available from: https://docs.confluent.io/kafka/design/efficient-design.html

36. Dobbelaere P, Esmaili KS. Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper. Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems. 2017 Jun;: 227–238.

37. Fredriksson B, Antonio C. Message brokers in a microservice architecture. Degree project. Stockholm: KTH Royal Institue of Technology, School of Electrical Engineering and Computer Science; 2021.

Appendix

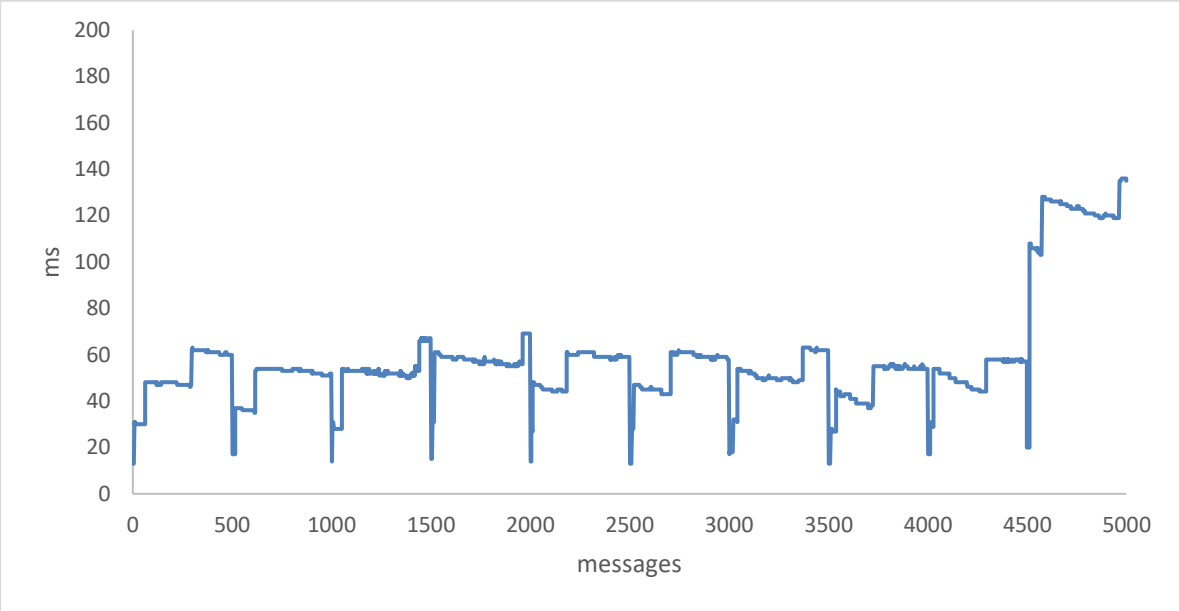

Figure 0.1 Kafka load, sending 1000 messages per second.

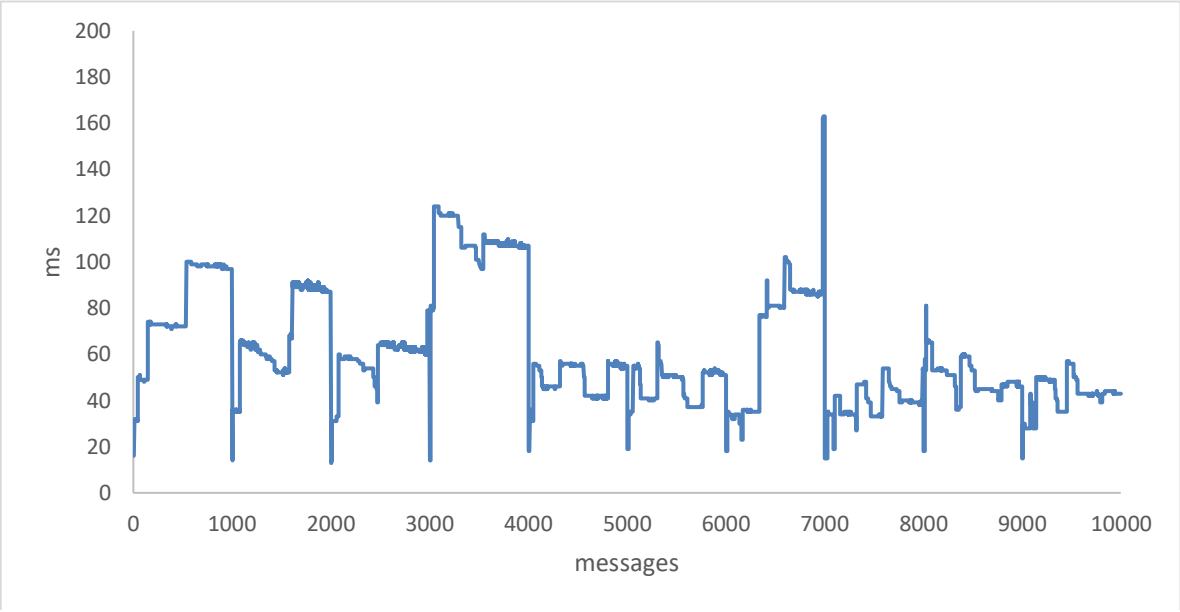

Figure 0.2 Kafka load, sending 2000 messages per second.

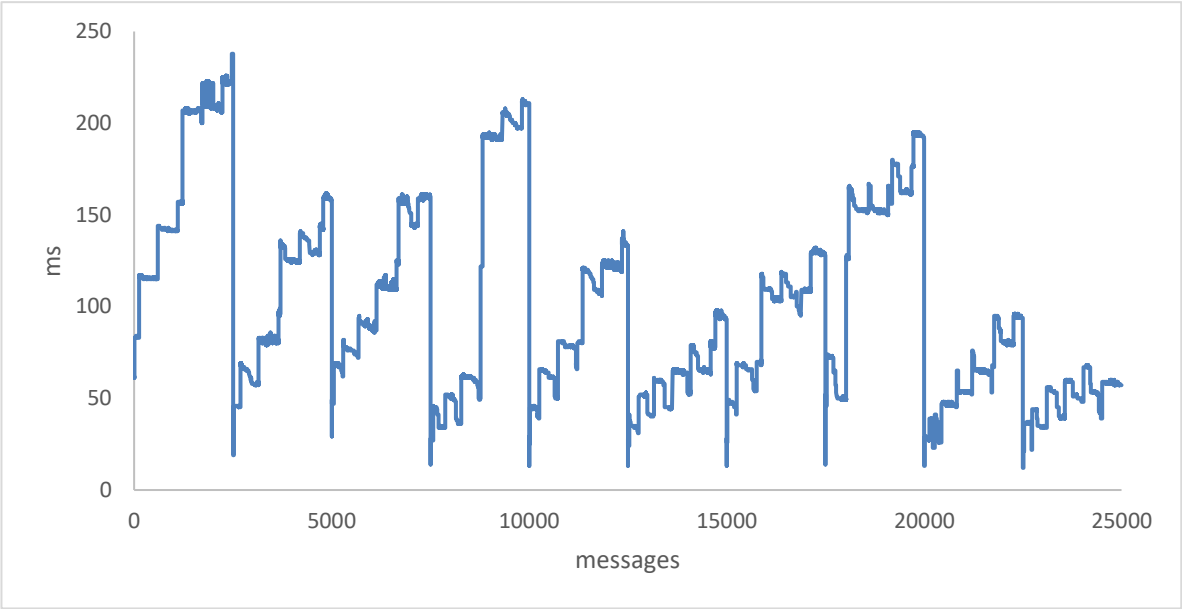

Figure 0.3 Kafka load, sending 5000 messages per second.

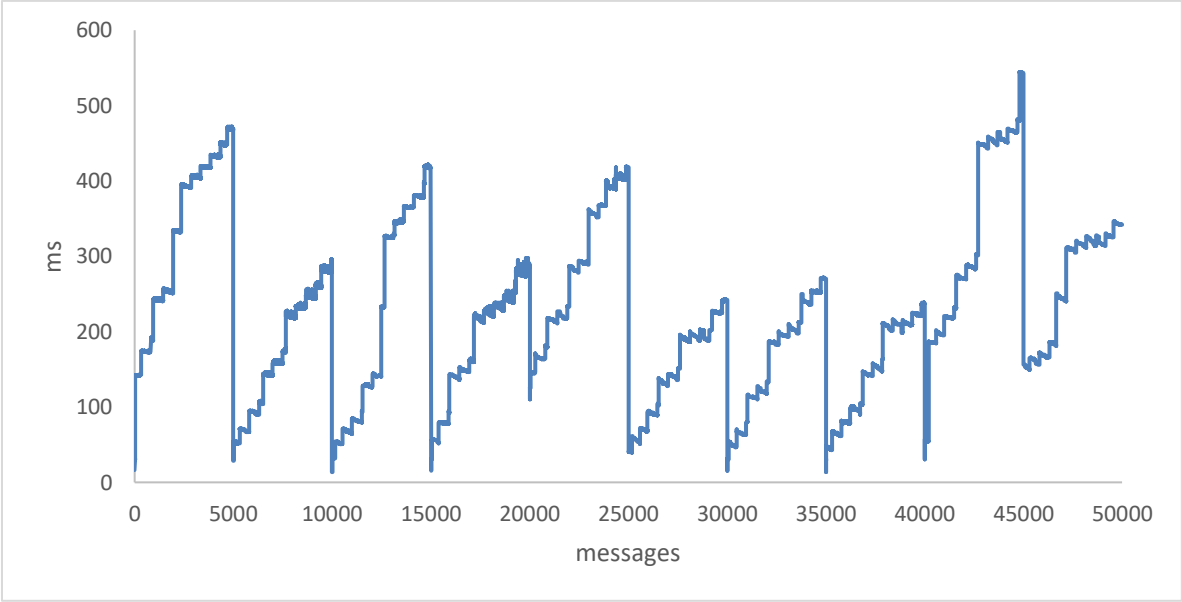

Figure 0.4 Kafka load, sending 10,000 messages per second.

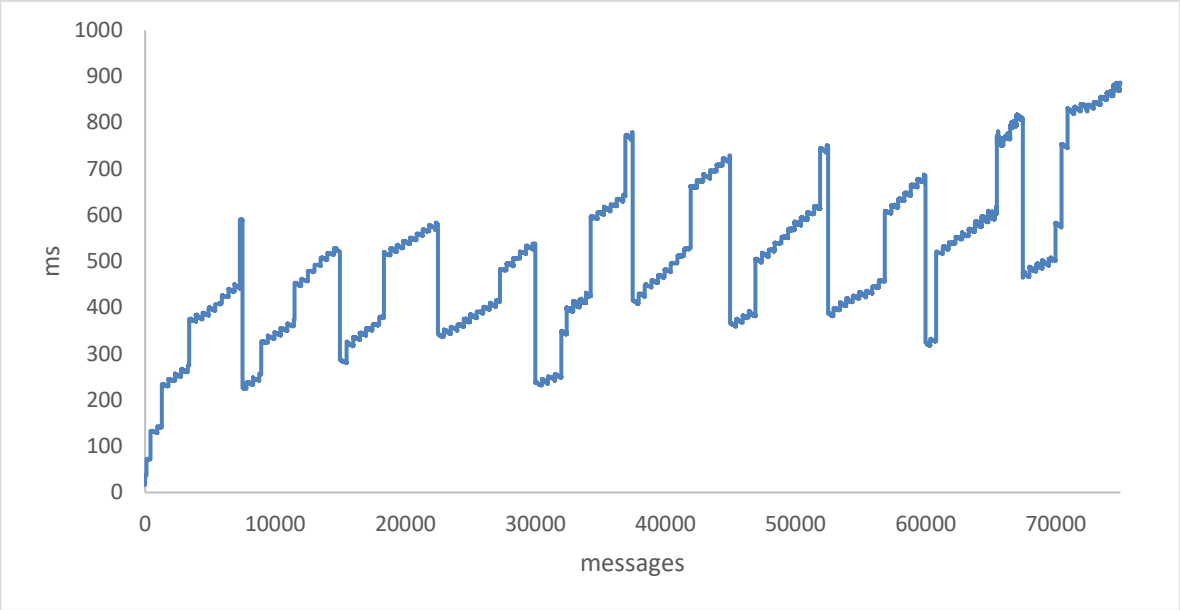

Figure 0.5 Kafka load, sending 15,000 messages per second.

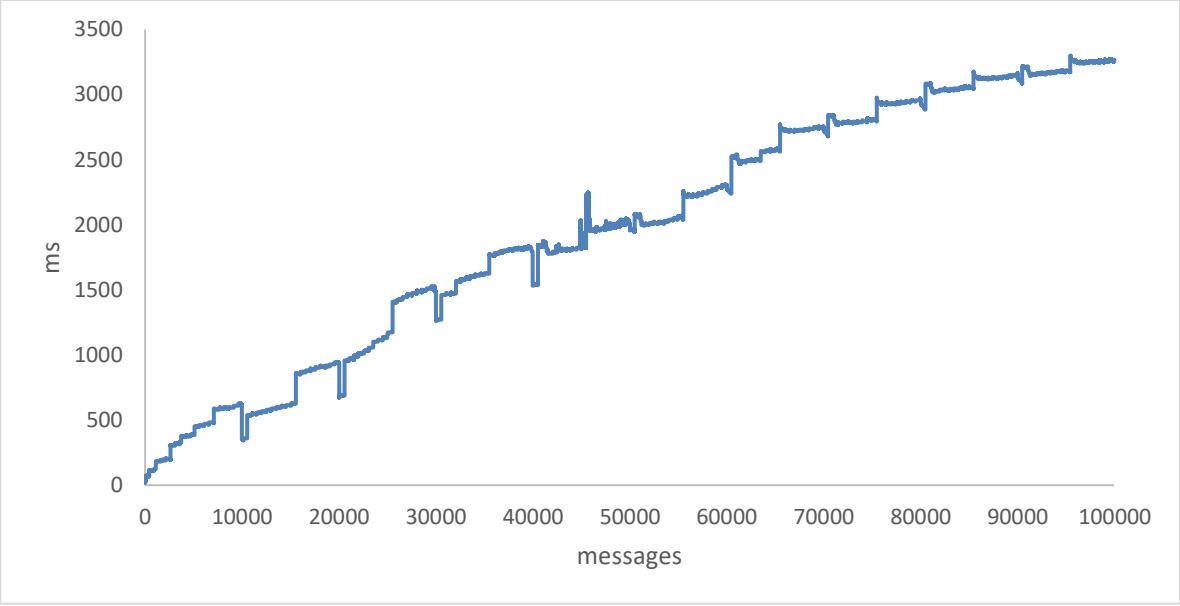

Figure 0.6 Kafka load, sending 20,000 messages per second.

TRITA – CBH-GRU-2023:258