UMEÅ UNIVERSITY

END-TO-END LATENCY AND THROUGHPUT EVALUATION OF TWO MESSAGE BROKERS

Martin Boman

Bachelor Thesis, 15 hp/credits
Department of Computing Science

2022

Abstract

As systems have grown bigger and bigger with time message brokers have become popular to help build better and more flexible designed systems. A message broker provides communication between services in a system while still keeping the components independent from one another. This thesis evaluates end-to-end latency and throughput of two of the most popular message broker software. The first message broker is Apache Kafka which uses a commit log to write all messages to disk and a message batching to optimize throughput. The second message broker is RabbitMQ which implements the standardized Advanced Message Queuing Protocol and, contrary to Kafka, primarily stores messages in the main memory. The throughput and message latency of both brokers were tested by integrating them both in an established system with micro-services and sending messages from one service to another through the message brokers. Results showed that RabbitMQ reached the highest recorded throughput while sending a large amount of data, and it also reached lower latency on individual messages sent without any queue build up. Kafka had more throughput and lower latency under smaller loads of data, but needs further configuration to become more effective as the default settings are limiting under heavy loads.

Contents

1 Introduction 1

1.1 Purpose 1

1.2 Research question 1

1.3 Delimitation 1

1.4 Thesis outline 1

2 Theoretical Background 2

2.1 Apache Kafka 2

2.2 RabbitMQ 3

3 Related work 4

4 Method 4

4.1 Producer and consumer clients 5

4.2 Experiments 6

5 Results 7

5.1 Single message latency results 7

5.2 Throughput results 8

6 Discussion 14

7 Conclusion 15

8 Future work 15

References 17

1 Introduction

As systems have grown bigger and bigger with time message brokers have become popular to help build better and more flexible designed systems. Message brokers provides communication between services and applications in systems, allowing these components to send messages to other parts of the system while keeping them independent from one another (IBM 2020). Today there exists several messages brokers implemented with different approaches and software to distribute messages over services and systems. Two different approaches to message brokers are a message broker based on a standardized protocol that primarily uses main memory storage, and a broker based on a distributed commit log which instead stores all messages on disk. Message broker software can have dissimilar features yet they fulfill the same purpose of creating a communication link between services which leads us to this study evaluating the performance of two popular brokers.

1.1 Purpose

Choosing a specific message broker requires understanding of the environment it will be used in, as there exist several software with different fundamentals for different use cases. Some approaches may excel at high throughput of data while other message queues are praised for their simplicity and ease of use. Apache Kafka and RabbitMQ are two popular choices when integrating a message broker into a system. Kafka is based on a distributed commit log while RabbitMQ is more traditional and implements a standardized protocol. The purpose of this thesis is to research the differences of how these two approaches perform as message queues between a service and gateway in an established system.

1.2 Research question

Does a message broker based on a distributed commit log have lower end-to-end latency and higher throughput than a message broker that implements the Advanced Message Queuing Protocol where both brokers have their most basic set-up?

1.3 Delimitation

Not only are there several other interesting message brokers to test, the differences between smaller features in each unique broker may also effect their performance depending on the experiment. Due to circumstances of time and the overall extent of this study only two of the most popular message broker software were chosen and the focus is mostly on their fundamental differences and how they affect performance.

1.4 Thesis outline

This thesis begins with a discussion of the fundamentals and background theory behind RabbitMQ and Apache Kafka as message brokers. To help with the background theory, related studies comparing the chosen brokers are also examined. After the theory an in-depth look at the method used to test the performance of the brokers is given, which is then followed by the results of those experiments. Lastly the results are discussed and earlier theory and related studies are used to further elaborate on the outcomes.

2 Theoretical Background

Distributed systems and micro-service based architectures lead to decoupling of components in a system. When different components are spread across individual devices Message-Oriented-Middleware (MOM) is often a needed tool to maintain communication between components of the system (Rheddane et al. 2014). A system with a MOM infrastructure relies on message exchange built on asynchronicity between the consumers and producers of messages. Message brokers are a key component in a MOM architecture as they handle the flow of messages inside the system (IBM 2020). These message brokers are responsible for storing and routing messages and often contain a message queue to store these messages. Messages sent from a producer are stored in the queue, ordered in the same way there were sent, until a consumer fetches them. A visualization of how a message broker can be implemented in a system built on micro-services is shown in Figure 1.

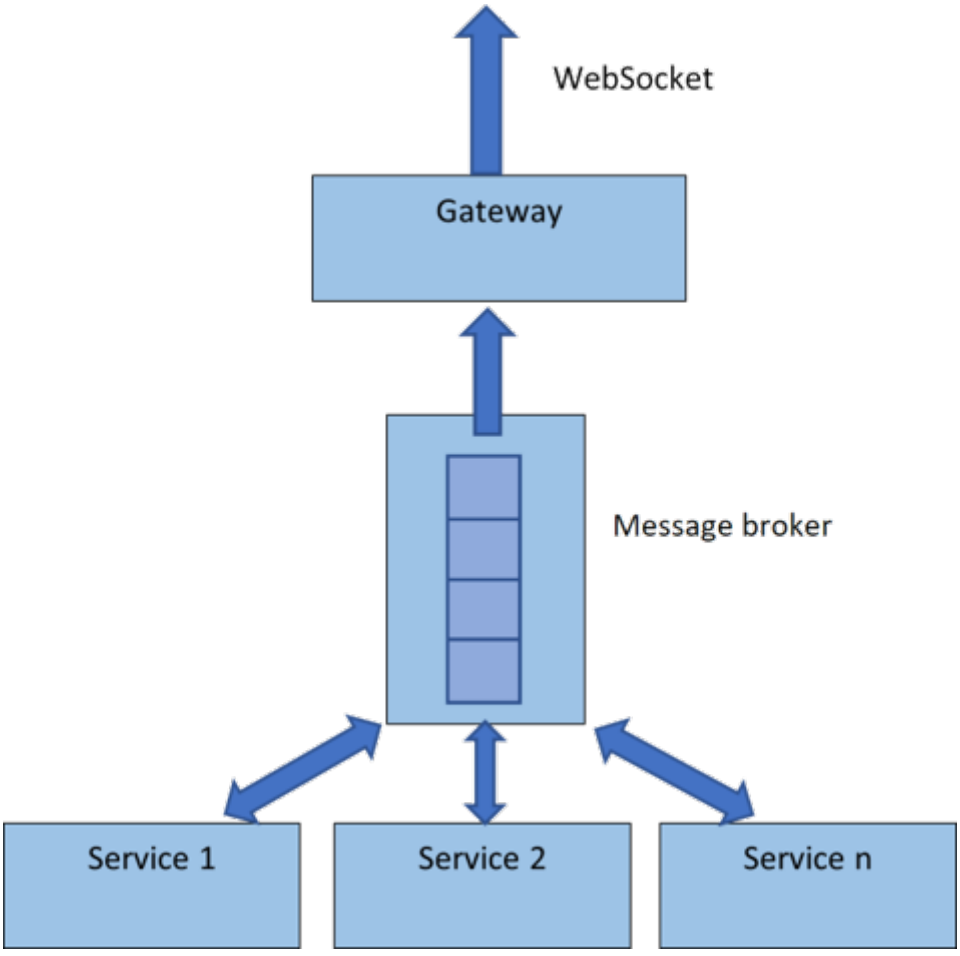

Figure 1: An example of how a message broker can provide communication between services and gateways in a micro-service built system. The gateway can communicate with a user client through a WebSocket.

2.1 Apache Kafka

Apache Kafka is an event streaming platform which can receive data from outside sources in the form of events, store and process these streams of events, and it can also send stored events to other services (Apache Software Foundation 2022). With these qualities Kafka has

many use cases, one of which is acting as a message broker. Kafka is able to store data and events with a distributed commit log which is how producers can publish data and consumers can subscribe to specific data (Confluent 2022b). The publisher and subscribers are solely connected to the commit log and therefore completely oblivious to the others existence. The commit log is an important fundamental to the platform as it both helps with scalability and durability. By writing all messages to disk, the log allows for permanent storage, similar to a database. All new publishes are written to end of the commit log and an offset is used to decide where to start reading from. When consuming messages from the queue the consumer pulls the messages from the queue rather than the queue pushing messages to the consumer (Confluent 2022a). The pull-based approach helps with removing flow control from the queue and letting the consumer recieve data at its own pace which can help increase throughput.

Kafka can easily use several brokers to create a cluster which creates better scalability and makes the system more fault tolerant as other brokers continue to serve if some of them should fail. When subscribing the consumer specifies what topics it subscribes to, which tells the broker what kind of data it wants. Topics are then divided into partitions where different partitions are placed in different brokers. Distributing partitions over brokers adds more scalability and makes it possible for several producers and consumers to write and read from the same topic simultaneously. Using several brokers and partitions are not the only configurable settings as Kafka lets user decide sizes for its end-to-end batch based messaging. Messages sent from a producer are naturally batched together to the commit log, and the consumer then fetches these batches instead of single messages which increases overall throughput. The batch sizes produced and consumed can be configured individually by the user (Apache Software Foundation 2022). The default batch size for the producer is 16.38 kilobytes while the consumer have several limiting default values, where the max amount of records pulled are 500 by default but it also has a default fetch size of 57.67 megabytes.

2.2 RabbitMQ

RabbitMQ is a message broker software which can support message queuing with its asynchronous messaging features (RabbitMQ 2022b). RabbitMQ is based on publisher and subscribers where consumers subscribe on specific topics and thus will only receive message of that kind. The broker implements the standardized Advanced Message Queuing Protocol (AMQP) which define how producers and consumers communicate with each other and what format the messages are sent in (Dossot 2014). What makes AMQP great is the interoperability, since the protocol defines the format of messages and how they are sent, the producer and consumer can always communicate despite being implemented in different ways (Ayanoglu, Aytas, and Nahum 2016). Diving deeper in AMQP shows how messaging is divided into two parts, exchanges and message queues. The queues in AMQP are quite basic as they only store and send messages to consumers. Exchanges are the first to receive a new message and its main objective is to route messages, with the help of a set of rules, to the correct message queue (Dobbelaere and Esmaili 2017).

While providing interoperability, RabbitMQ sacrifices the persistency of messages through mostly disk-less storage. A message does not need to be stored on a disk and instead RabbitMQ tries to keep messages in the computers main memory to route them to their destination. The message queue is mainly focused on delivering messages while the queue is either empty or close to being empty, which means performance may suffer if the queue starts building up (ibid.). RabbitMQ can be distributed and ran in a cluster of nodes, where each node is a unique broker (RabbitMQ 2022a). When running a cluster queues can be replicated on other nodes to increase availability, but this is only optional.

3 Related work

Dobbelaere and Esmaili (2017) conducted a similar study where they compared the performance in the form of latency and throughput between Apache Kafka and RabbitMQ. The study discusses similarities and differences regarding the features of these two message brokers and shows results from experiments testing their performance. RabbitMQ and Kafka can both be configured heavily in many ways, for example using several brokers in a cluster and using more than one produer and consumer, which the study explores. In the end Kafka and RabbitMQ both can reach relatively low latency and without tweaking any configurations RabbitMQ has higher throughput of the two. Kafka can increase its throughput significantly easily by adding partitions and nodes, meanwhile increasing the performance of RabbitMQ is more complex.

Another study has been done by Rabiee (2018) analyzing how different parameters and configurations impact latency and throughput usage on RabbitMQ and Apache Kafka. CPU and RAM usage were also measured but only as secondary metrics. In the study the author conducts his own experiments to test the chosen parameters of RabbitMQ and Kafka with a static message size of 500 bytes throughout all experiments. The results show that Kafka has higher throughput and lower latency than RabbitMQ, with Kafka reaching over 5 times as many messages sent than its counterpart. Even though being faster Kafka writes a significant more amount of data to disk than RabbitMQ, which occupied a relatively low amount of disk space.

Nikhil and Chandar (2022), two developers at Confluent, have published a study comparing 3 message brokers Apache Kafka, Apache Pulsar and RabbitMQ. Performance was measured in throughput and latency since those are the most important metrics of event streaming systems according to the study. While discussing the design and fundamentals of the different brokers, the lack of durability and availability is mentioned regarding RabbitMQ and therefore the queue is mirrored, essentially replicated on additional brokers, to improve availability at the cost of performance. After the tests of data throughput and latency the study concludes that Kafka and Pulsar provide higher throughput than RabbitMQ, though RabbitMQ can reach the lowest end-to-end latency when the overall throughput is low over the broker.

Studies researching the performance of arguably the two most popular message broker systems have been done before, but as seen in these above mentioned studies there are a lot of parameters and configurations that differ between each test. I hope to give more insight to how the traditional message broker system RabbitMQ using AMQP and the commit log based Apache Kafka perform under default configurations and with a basic setup. The benchmark tests published by Confluent may be biased as to promote their services for managing and using Kafka, and therefore I also hope to both verify and provide unbiased information of these two message broker softwares.

4 Method

To test the latency and throughput of two different message brokers experiments consisting of sending messages between services of a system were done. The already established system Ladok was used, which is a system that handles information about the education of all the former and present students in Sweden. Creating a MOM-architecture in this already established system consisted of setting up the broker server and implementing consumer and producer clients. The brokers were downloaded, RabbitMQ version 3.3.5 and Apache Kafka version 3.1.0 were used, and no configuration was done in order to preserve all default set-

tings. Apart from the default configurations none of the brokers were run in a cluster, only single nodes were used and the same topic was used for every message.

4.1 Producer and consumer clients

As Java has libraries for both Kafka and RabbitMQ clients it was the programming language chosen for implementing producer and consumer clients. Two different services within Ladok were used to act as a consumer and producer, respectively. The same algorithms were used for Kafka and RabbitMQ as both clients and libraries were very similar. Algorithm 1 shows how the producer client was implemented on a level that works for both RabbitMQ and Kafka, and Algorithm 2 describes the flow of the consumers.

Algorithm 1 Producer

```
Establish connection to broker
Create buffer with random bytes big enough for all messages
for message = 1, 2, ..., N do
    Add timestamp
    Send message
end for
```

Algorithm 2 Consumer

```
Establish connection to broker
for message = 1, 2, ..., N do
    Receive message
    if firstmessage then
        Save timestamp for first message
    end if
    Calculate time difference between message and current time
    Add time difference to file buffer
end for
Calculate time difference between first message and current time
Write buffer and total time to file
```

The producers were set to start producing messages when a simple REST API call was done. Timestamps were added to each individual message to measure both latency and throughput. Filling each message with bytes was done by using randomized strings of characters that each corresponds to one byte. Each message had a new randomized but the content of each message did not matter and this was only done for convenience sake. The timestamps were added to the messages just as they were sent and then compared to the current time of the consumer when each individual messages was reveived and processed. The messages for RabbitMQ had the timestamp as the key since the broker had no reserved slots for timestamps in the messages. In both consumer clients each message has to be processed to compare timestamps and add the time to a buffer which is then written to a file when the consumer has finished. While processing messages takes time it was done the exact same way for both clients to make sure the brokers were measured fairly. The consumer kept track of when the first message was produced and when the last message had been processed. As one service acted as a producer and another as a consumer the brokers where only routing messages one way, Figure 2 illustrates the system at work.

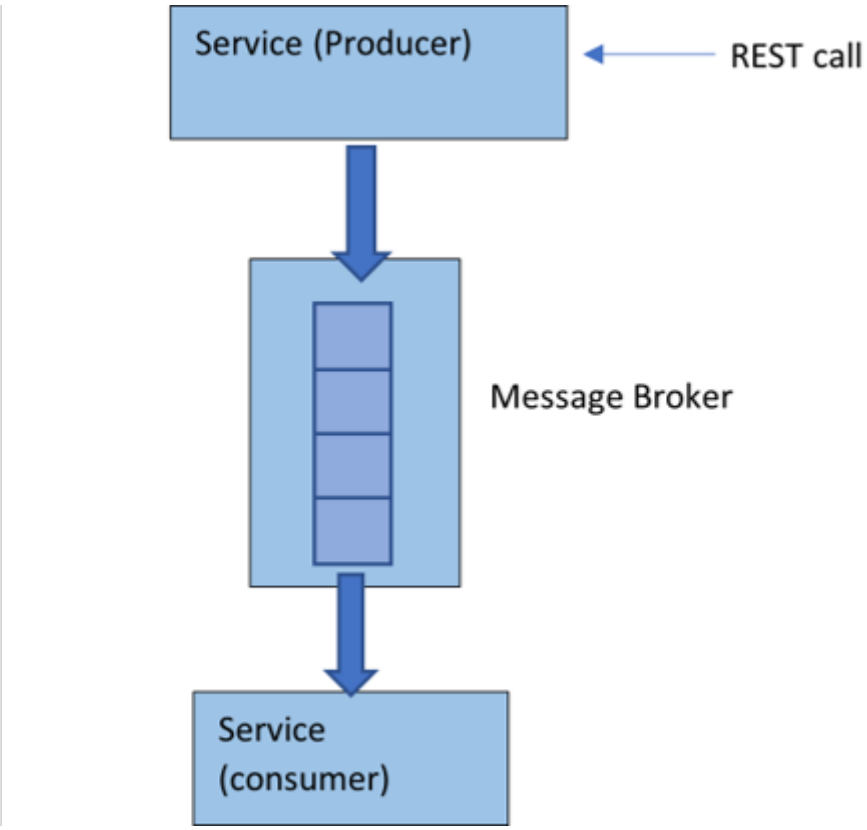

Figure 2: An illustration of how the message broker was implemented between two services that acts as producer and consumer clients.

4.2 Experiments

All components required for the tests, including the system containing the consumer and producer and the broker softwares, were running on a Linux Red Hat server with 32GB of RAM and a Intel Xeon Gold 6152 CPU at 2.10GHz clock speed.

The experiments were done with different amounts of messages and bytes per message. First off a test of only latency was done where 100 messages of 100 bytes each were sent with a 1000 millisecond delay between every message. This test was to show how the brokers performed when the queue was under no stress and kept empty after every message. The other tests were done by sending 10, 100, 1000 and 10 000 messages where for each number of message sent, four different message sizes were sent. The message sizes were 10, 100, 1000 and 10 000 bytes per message, same as the numbers of messages sent, which means there were 16 unique tests in total. The total time for all messages to be sent and then consumed was the difference of the first and last message which was used to then calculate the throughput. The throughput was measured in kilobytes per second (kB/s) which is a one-to-one conversion from bytes per millisecond which were the measurements message size and time were measured in during the tests.

When the experiments were done, the result files contained the time for each individual message and the total time for the test. The files were imported to Microsoft Excel to calculate average message latency and throughput which then could be plotted on graphs to show and compare the results between the two message brokers. As each individual message timestamp was available the growth in message latency during the throughput tests could also be plotted. Matlab was also used for producing boxplots and getting the 99th percentile for the

latency tests. When calculating the throughput, the message size was interpreted as equal to the bytes sent as the content of message. Messages contain more data such as timestamps and topics, but such data were disregarded as they are automatically included in the message and have no valuable information for the services communicating with each other.

5 Results

This section presents the results from the latency experiment and the throughput tests where the brokers got to experience more stress. The first experiment shows the end-to-end latency of messages when the queue is empty as each message were sent 1 second apart. Following the latency test are the results from the throughput tests with different size and amount of messages.

5.1 Single message latency results

Figure 3 shows the average latency of all 100 messages sent with a 1 second pause between each one. Kafka had 3 times higher latency at an average around 4.5 milliseconds than RabbitMQ, which reached an average under 1.5 milliseconds. The boxplots in Figure 4 show that every value deviating from 1 millisecond is an outlier for RabbitMQ, while Kafka has extreme data points ranging from 3 to 6 milliseconds. The 99th percentile showed that 99% of values were below 4 milliseconds for RabbitMQ and 13.5 milliseconds for Kafka.

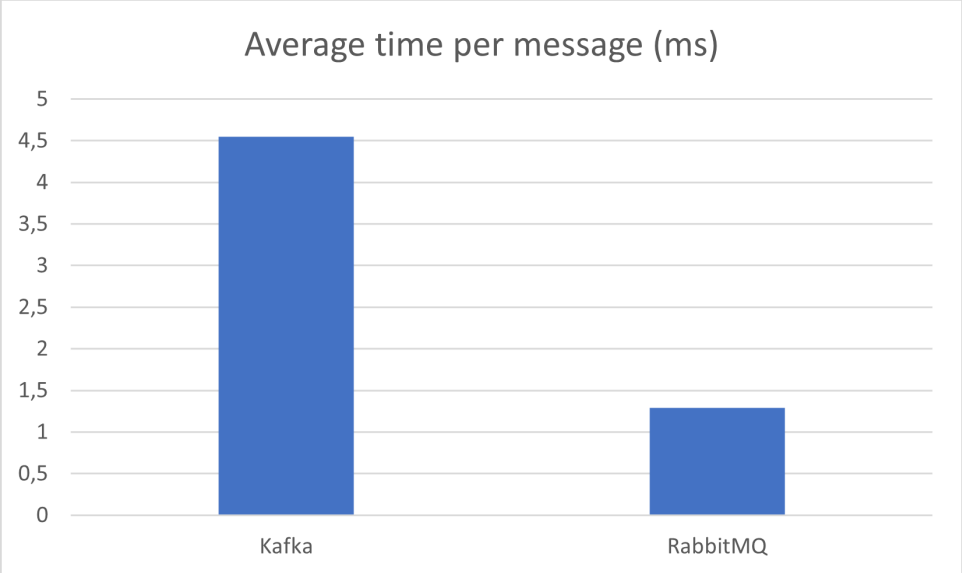

Figure 3: Average end-to-end message latency for Kafka and RabbitMQ

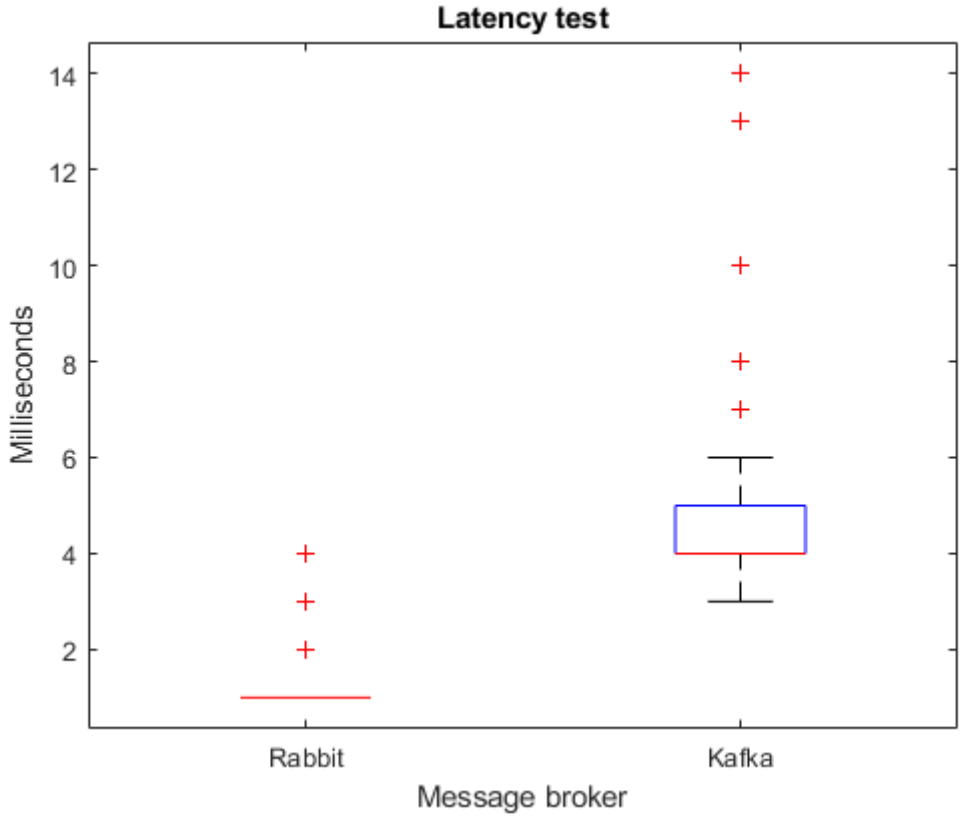

Figure 4: Boxplots for both RabbitMQ and Kafka showing median, deviation and outliers of values in the latency test

5.2 Throughput results

The throughput tests are divided into four bar charts, each with a different size on each message sent, followed by a table showing the exact throughput in each test. A graph of the growth in message latency is also presented, though only for the tests were 10 000 messages were sent.

For messages containing 10 bytes, Kafka reached higher throughput than RabbitMQ for all four amounts of messages, shown in Figure 4. The difference in throughput was close to zero at 10 messages, but the difference became bigger as the number of messages sent grew.

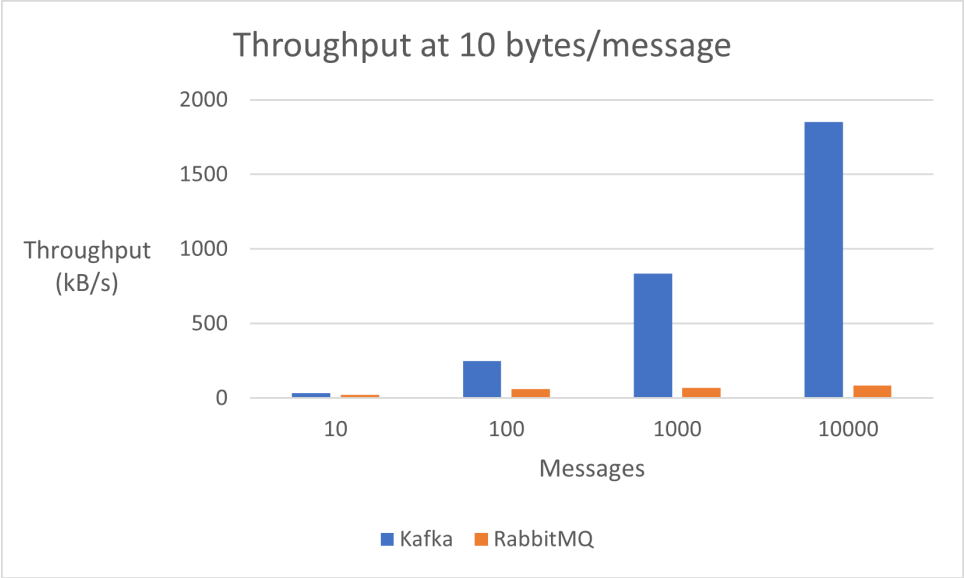

Figure 5: Figure shows the throughput at different amount of messages sent with 10 bytes each for Kafka and RabbitMQ

Table 1 shows the exact throughput at 10 byte sized messaged, where Kafka grew from 33.33 kB/s at 10 messages up to 1851.85 kB/s at 10 000 messages, atleast doubling in each step. RabbitMQ had a smaller growth of throughput going from 20.00 kB/s to 82.71 kB/s at its highest, where only between 10 and 100 messages did the throughput at least double. The growth in message latency is big for RabbitMQ where the last message nears 1200 milliseconds, while kafka has a significantly lower growth, as seen in Figure 6.

Table 1: The table shows the exact throughput in kB/s for messages of 10 bytes with different numbers of sent messages.

	10 messages	100 messages	1000 messages	10 000 messages
Kafka	33.33	250.00	833.33	1851.85
RabbitMQ	20.00	58.82	68.97	82.71

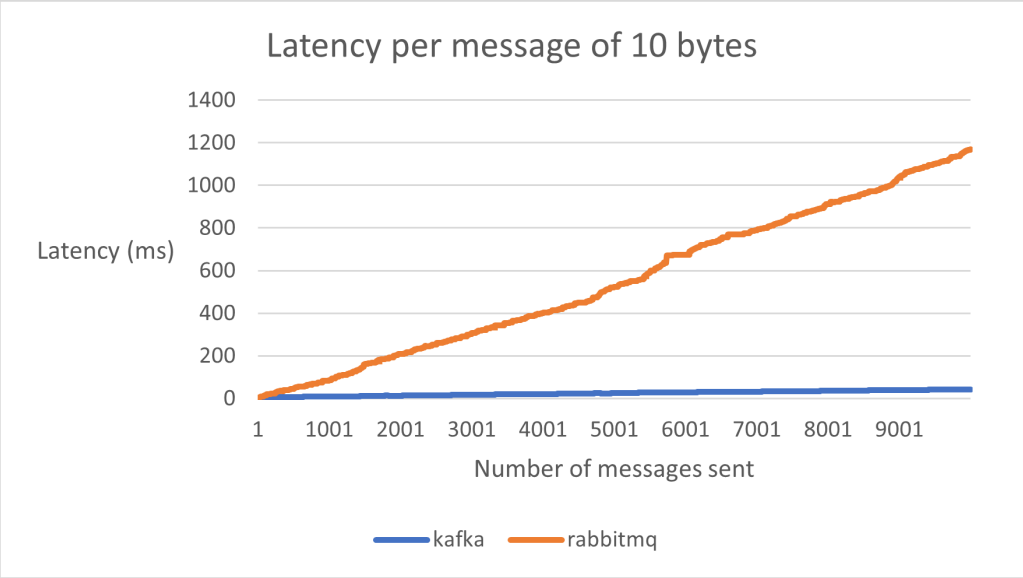

Figure 6: Figure shows the latency of each message over the span of 10 000 messages sent with a message size of 10 bytes

At bigger messages with 100 bytes Kafka also has higher throughput throughout all number of messages sent, as shown in Figure 7. The difference of throughput is significant already at 10 messages sent as seen in Table 2, where Kafka has more than double the throughput of RabbitMQ. Kafkas throughput also grows more than double between 100 messages and 1000 messages, whereas RabbitMQ has a throughput increase from 666.67 kB/s to 813.01 kB/s. Figure 8 show how RabbitMQ has a much greater growth in latency than kafka as the load on the broker gets bigger.

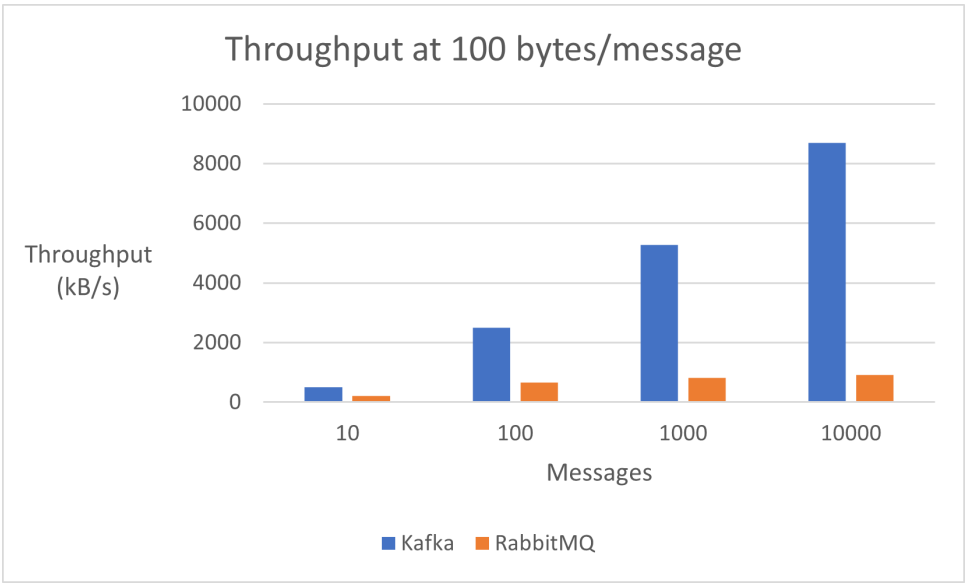

Figure 7: Figure shows the throughput at different amount of messages sent with 100 bytes each for Kafka and RabbitMQ

Table 2: The table shows the exact throughput in kB/s for messages of 100 bytes with different numbers of sent messages.

	10 messages	100 messages	1000 messages	10 000 messages
Kafka	500.00	2500.00	5263.16	8695.65
RabbitMQ	200.00	666.67	813.01	915.75

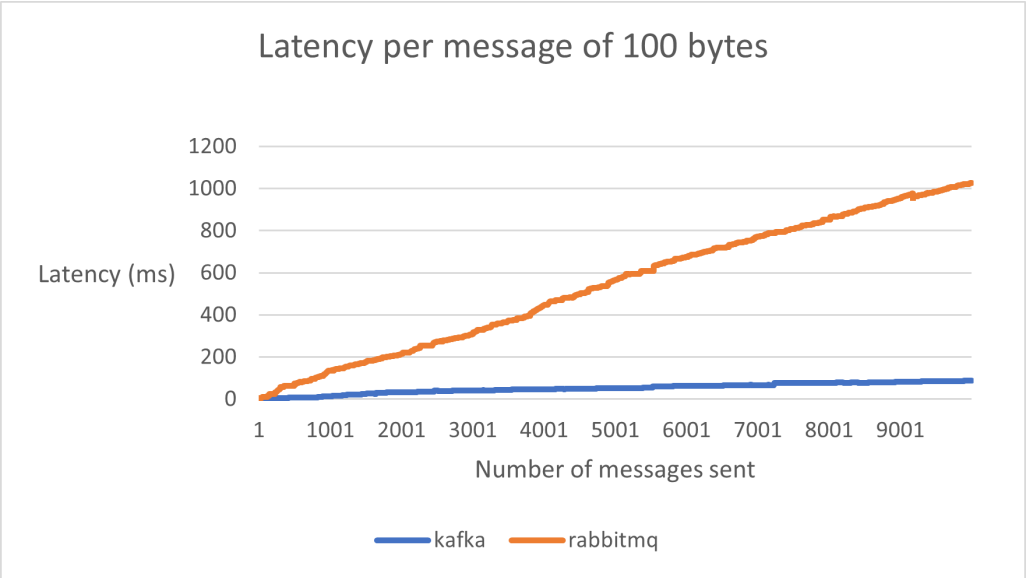

Figure 8: Figure shows the latency of each message over the span of 10 000 messages sent with a message size of 100 bytes

At 1000 bytes per message the throughput of Kafka and RabbitMQ is more alike. As Figure 9 show RabbitMQ has lower throughput than Kafka at 10 messages, but at 100 messages RabbitMQ actually has a little more throughput. Despite being so close in throughput at fewer messages, Kafka still has higher throughput when more messages are sent. Table 3 shows that RabbitMQ reaches a throughput of 8403.36 kB/s at 1000 messages which is almost doubled by Kafka at 16 129.03 kB/s. Despite Kafka still having higher throughput, the messages' latencies are actually higher for kafka than RabbitMQ's after around 3000 messages were sent, as seen in Figure 10.

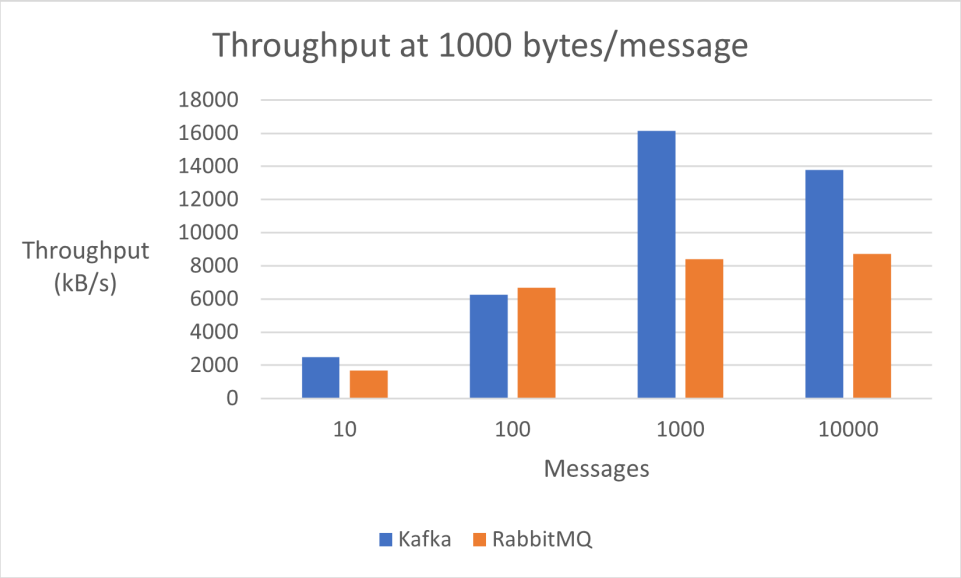

Figure 9: Figure shows the throughput at different amount of messages sent with 1000 bytes each for Kafka and RabbitMQ

Table 3: The table shows the exact throughput in kB/s for messages of 1000 bytes with different numbers of sent messages.

	10 messages	100 messages	1000 messages	10 000 messages
Kafka	2500.00	6250.00	16 129.03	13 793.10
RabbitMQ	1666.67	6666.67	8403.36	8726.00

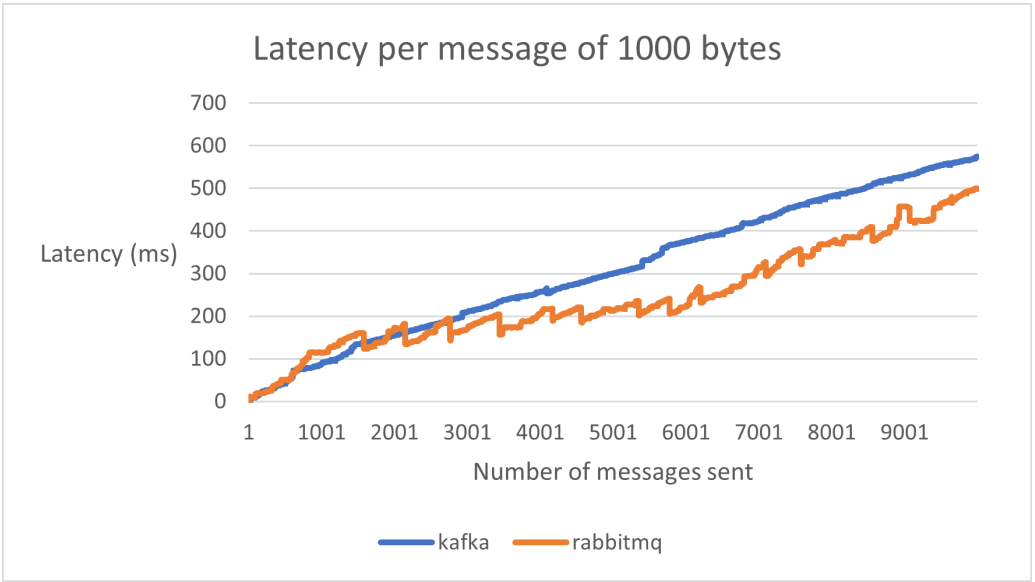

Figure 10: Figure shows the latency of each message over the span of 10 000 messages sent with a message size of 1000 bytes

Lastly at 10 000 bytes per message RabbitMQ outperformed Kafka at every message volume, which can be seen in Figure 11. Table 4 shows the exact throughput of the test which marks

RabbitMQ as the broker with the highest measured throughput out of all the tests, reaching 54 945.05 kB/s at 1000 messages sent with 10 000 bytes each. The throughput stopped increasing for Kafka after 100 messages, and for RabbitMQ 100 messages also reached higher throughput than 10 000 messages sent. Figure 12 show that Kafka had significantly higher message latency than RabbitMQ throughout whole last test. RabbitMQ never reached individual message latencies over 200 milliseconds during the test of 10 000 messages sent with 10 000 bytes each.

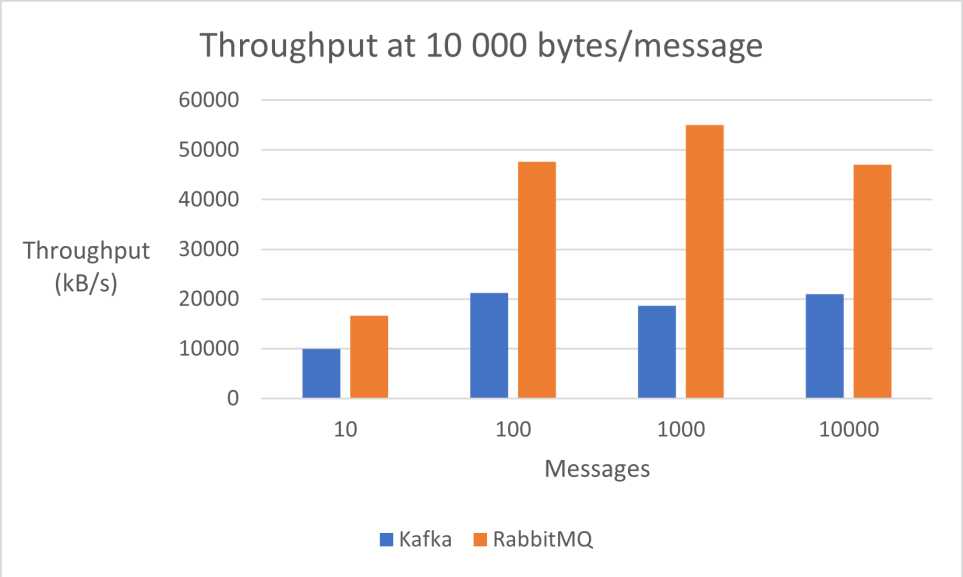

Figure 11: Figure shows the throughput at different amount of messages sent with 10 000 bytes each for Kafka and RabbitMQ

Table 4: The table shows the exact throughput in kB/s for messages of 10 000 bytes with different numbers of sent messages.

	10 messages	100 messages	1000 messages	10 000 messages
Kafka	10 000.00	21 276.60	18 587.36	20 990.76
RabbitMQ	16 666.67	47 619.05	54 945.05	46 970.41

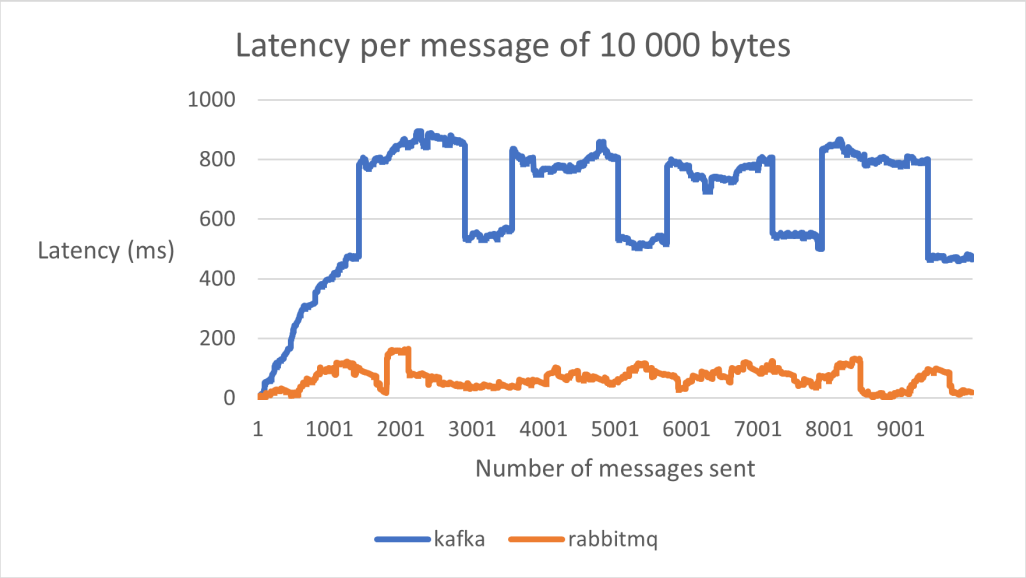

Figure 12: Figure shows the latency of each message over the span of 10 000 messages sent with a message size of 10 000 bytes

6 Discussion

The experiments were done to gain further insight on the strengths and weaknesses of a message broker with a commit log for permanent storage, Apache Kafka, and another broker which uses the standardized AMQP, RabbitMQ. Performance was measured through latency and throughput, which are the most looked for attributes when it comes to message brokers. Latency was first measured through a test with a 1 second pause between each message to get the latency of single messages while also keeping the queues empty. The second test focused on throughput, but the latency was also recorded during the tests where 10 000 messages were sent which will give insight on the message latency when the queues are busy and put under stress.

The results of the first test show that RabbitMQ reaches lower latency than Kafka when working with an empty queue where only single messages arrive with delay between each one. Having to write each message to disk and use an offset to keep track of the messages in the log is not optimal for low latency in this scenario with an empty queue. RabbitMQ, which tries to keep every message in the DRAM, reaches very low latency and proves the broker is optimized for empty queues.

More varying results were recorded from the throughput tests where Kafka shone at tests with lower message sizes until messages of 1000 bytes were sent. RabbitMQ had the highest measured throughput during the test with 10 000 bytes per message sent, which could be an answer to the research question while ignoring the significantly higher throughput of Kafka at lower message sizes. The end-to-end batching used by Kafka is limited by the producer at around 16 000 bytes per default which could be a factor to the drop off in throughput when messages became bigger. But since this study only focuses on the two brokers at their most basic set up, the commit log based message broker only reaches higher throughput when the byte size per message is low.

The message latencies recorded during the throughput tests follow the same trend as the

throughput results for the most part. At 10 and 100 bytes per message, the latencies were low for Kafka throughout both tests while RabbitMQ had a steady growth in latency as the queue was building up. The results were a little surprising for 1000 bytes per message as Kafka actually had higher latency than RabbitMQ for most messages of the test, despite having higher throughput during that particular test. Everything was back in order for the last test with messages sizes being 10 000 bytes as RabbitMQ had higher throughput and lower latency. These inconsistent results can be explained partly by Kafkas batches, but also by the fact that throughput is calculated with total time which also accounts for the time it takes for messages to be created. As messages get bigger, the process of creating the messages probably takes longer time, something that wont affect the individual messages' latencies, since they were timestamped just before they were sent. This can also explain the fact that RabbitMQ had lower message latency growth on 10 000 bytes per message compared to lower message sizes.

7 Conclusion

To answer the research question: no, a messager broker with a commit log does not reach lower latency nor higher throughput than an AMQP using broker, or atleast not in all situations. The results of this thesis is in line with the study Dobbelaere and Esmaili conducted where the commit log based Kafka needed added partitions and nodes to further increase the throughput, otherwise the AMQP using RabbitMQ had better performance without clustering or tweaking any settings. The other related studies which showed that Kafka had better throughput had tests either run in clusters or different configurations, such as the test published by the two confluent developers where RabbitMQ had a mirrored queue.

This study not only shows that the message broker based on a commit log does not have higher throughput and lower latency compared to the AMQP broker, but also the use cases of these types of brokers. At their most basic form of set up RabbitMQ can reach the highest throughput and lowest latency with an empty queue, but somewhere in between Kafka thrives at delivering a bigger amount of smaller byte sized messages.

8 Future work

For future work the batch sizes for Kafka can be configured to test whether or not the default values are limiting factors for throughput. Changing these values for the producer and consumer is easy and may affect the throughput significantly. Number of partitions could also be tweaked as well as making both brokers run in a cluster which absolutely should affect the throughput as more nodes can handle more messages together. More different fundamental functions of both brokers could be researched to further investigate which factors increases performance in throughput and latency, and which factors who instead worsen performance to provide better scalability, durability or other important attributes.

References

Apache Software Foundation (2022). *Documentation.* URL: https://kafka.apache.org/documentation (visited on 04/20/2022).

Ayanoglu, Emrah, Yusuf Aytas, and Dotan Nahum (2016). *Mastering rabbitmq.* Packt Publishing Ltd.

Confluent (2022a). *Kafka vs. Pulsar vs. RabbitMQ: Performance, Architecture, and Features Compared.* URL: https://www.confluent.io/kafka-vs-pulsar/ (visited on 04/21/2022).

– (2022b). *What is Apache Kafka?* URL: https://www.confluent.io/what-is-apache-kafka/ (visited on 04/20/2022).

Dobbelaere, Philippe and Kyumars Sheykh Esmaili (2017). "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*, pp. 227–238.

Dossot, David (2014). *RabbitMQ essentials.* Packt Publishing Ltd.

IBM (2020). *Message Brokers.* URL: ibm.com/cloud/learn/message-brokers (visited on 04/25/2022).

Nikhil, Alok and Vinoth Chandar (2022). *Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest?* URL: https://www.confluent.io/blog/kafka-fastest-messaging-system/ (visited on 04/27/2022).

RabbitMQ (2022a). *Distributed RabbitMQ.* URL: https://www.rabbitmq.com/distributed.html (visited on 04/25/2022).

– (2022b). *RabbitMQ Features.* URL: https://www.rabbitmq.com/ (visited on 04/25/2022).

Rabiee, Amir (2018). "Analyzing Parameter Sets For Apache Kafka and RabbitMQ On A Cloud Platform". MA thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

Rheddane, Ahmed El, Noel de Palma, Alain Tchana, and Daniel Hagimont (2014). "Elastic Message Queues". In: *IEEE International Conference on Cloud Computing.*

UMEÅ UNIVERSITY