EXAMENSARBETE INOM DATATEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, SVERIGE 2020

Performance evaluation of message-oriented middleware

Utvärderdering av prestanda för meddelandeorienterade mellanprogramvaror

ERIK NILSSON

VICTOR PREGÉN

KTH
SKOLAN FÖR KEMI, BIOTEKNOLOGI OCH HÄLSA

Performance evaluation of message-oriented middleware

Utvärderdering av prestanda för meddelandeorienterade mellanprogramvaror

Erik Nilsson

Victor Pregén

Examensarbete inom
Datateknik
Grundnivå, 15 hp
Handledare på KTH: Anders Lindström
Examinator: Ibrahim Orhan
TRITA-CBH-GRU-2020:250

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

Abstract

Message-oriented middleware (MOM) is a middleware used for communication between applications. There are many different MOM technologies available today, each offering different performance (throughput and latency). The performance of MOMs depends on both message size and message guarantee settings used. The problem is that it can be difficult for users to know which MOM they should choose given their requirements. The goal was to create a performance (latency and throughput) comparison of three popular MOMs; Apache Kafka, RabbitMQ and Nats Streaming. The result shows that Kafka is the best performing MOM for smaller message sizes (under 512 bytes). RabbitMQ has the best performance for larger message sizes (over 32768 bytes). Nats Streaming only outperformed the other message system for a few combinations of message guarantee settings with the message size 4096 bytes.

Key words

Message-oriented middleware, Kafka, RabbitMQ, Nats, Nats Streaming, throughput, latency

Sammanfattning

Meddelandeorienterad mellanprogramvara (MOM) är mellanprogramvara som används för kommunikation mellan applikationer. Det finns många MOM system som erbjuder olika prestanda (genomströmning och latens). Prestandan är beroende av vilka meddelandegarantier som används samt meddelande storlek. Detta gör det svårt för användare att välja MOM utifrån sina krav. Målet är därför att jämföra tre populära MOMs; Apache Kafka, RabbitMQ och Nats Streaming. Resultaten visar att Kafka presterar bäst med små meddelandestorlekar (Under 512 bytes). RabbitMQ presterar bäst för större meddelanden (Över 32768 bytes) medans Nats Streaming enbart presterar bäst med ett begränsat antal meddelandegarantier och med en meddelandestorlek på 4096 bytes.

Nyckelord

Meddelandeorienterad mellanprogramvara, Kafka, RabbitMQ, Nats, Nats Streaming, genomströmning, latens

Table of contents

1 Introduction

Message-oriented middleware (hereafter called MOM) is a middleware used for communication between applications. The difference compared to traditional point-to-point communication is that there is a broker that lives between the communicating entities. This enables more loosely coupled communication and makes it easier to connect heterogeneous applications that use different languages or protocols.

There are many different use cases for MOMs, and it is impossible to list them all here. Some common use cases include website activity tracking, log aggregation, stream processing and event sourcing. Even if MOMs are used for a wide range of applications, a common denominator is that they are used for transporting relatively small messages.

Low latency is important for MOMs because they are often used in many real-time applications, for example real-time bidding. Throughput is also important since MOMs are often used to handle large amounts of data.

An important part of a MOM is the message guarantees it offers. The guarantees differ depending on MOM and configuration. The guarantees include:

- Delivery semantics such as: at-least-once, at-most-once and exactly-once.
- Message persistence.
- Message ordering.

1.1 Problem

There are many different MOM technologies available today, each offering different performance (throughput and latency). The MOMs also offer different message guarantees. Some MOMs let the users change the configuration for message guarantees which in turn affects the performance. Another factor which affects the performance is the message size.

The problem is that it can be difficult for users to know which MOM they should choose. Because different users use the MOM to carry messages of different sizes, and different volumes of messages. They also have different requirements for performance and message guarantees.

Several papers making this type of performance comparisons of different MOMs have already been published. However, they have one or both of the following two shortcomings: they compare MOMs for a specific use case and they do not consider how the message guarantee settings impact performance. Another problem is that the performance of the MOMs changes as developers update the systems. Therefore, the results from a paper published three years ago might not be relevant today.

Another problem is that there has not yet been a peer-reviewed comparison which includes the MOM Nats. Nats is a newer MOM that has shown to have very good performance. However it is not yet known how it will compare against the older and more widely used message systems.

1.2 Goal

The goal is to create a performance (latency and throughput) comparison of three popular MOMs; Apache Kafka, RabbitMQ and Nats. The purpose of the comparison is to help gain knowledge for users trying to find a suitable MOM for their particular use case. The overall goal can be broken down to the following set of sub-goals.

- Compare how message guarantee settings affect the performance of each MOM, e.g. at-most-once and at-least-once.
- Compare how persistence and ordering settings affect the performance of each MOM.
- Compare how message sizes between 8 bytes to 1 MB affect the performance of each MOM.

1.3 Delimitations

This thesis mainly focuses on evaluating how message size and message guarantees impact the performance of the MOMs. The thesis will not compare the functionality or make a qualitative comparison of the MOMs. For example, ease of use or the quality of documentation will not be compared.

2 Background and theory

The chapter begins by explaining the general concepts of MOMs. This includes the basic architecture, features, settings and advantages of MOMs. In the following sections, each MOM included in the thesis is described in more detail. In the last section, previous work related to the subject is presented.

2.1 Message-oriented middleware

Modern computer systems, by nature, have become more distributed. This stems from the fact most of the time a single monolithic application cannot fulfil all business requirements. A distributed system is made up from many smaller applications, each of which is responsible for a specific task. The applications are then interconnected with each other to meet the total business requirements. The applications in a distributed system tend to be very heterogeneous, which makes it difficult to integrate them. Applications are implemented in different programming languages, have different hardware, operating systems and network architecture etc [1]. Middleware aims to solve these problems by providing interfaces with one extra layer of abstraction that hides the underlying heterogeneous implementation. This enables developers to focus on the application logic instead of integration [2].

Message-oriented middleware (MOM for short) is a middleware used for communication between applications. MOMs let heterogeneous applications communicate despite differences in programming languages or operating systems etc. By queuing messages in a broker, MOMs can provide loosely coupled and asynchronous communication that is more flexible than traditional synchronous point-to-point communication. MOMs have become more popular since it simplifies communications in large distributed systems. The popularity of MOMs has also been sprung by the big data trend that requires high throughput message systems [1].

2.2 General architecture of message-oriented middleware

The general structure of a MOM, as shown in figure 2.1 includes four parts: Producers, consumers, messages, and the broker. Processes that are sending data are called producers, the processes that receive data are called consumers. The data entities that are being exchanged between the applications are called messages. Between the producers and the consumers is the broker which contains mainly routing logic and a queue (or queues). The producer generates the messages that are sent off to the broker. This is typically done asynchronously so producers can send a message and continue doing other work. When the message arrives at the broker it is placed in a queue for consumption. There are typically two methods by which a consumer can receive messages from the broker. The first method is push based; the broker sends messages to the consumer when new ones arrive. The second method is pull based where the consumer pulls the messages from the broker. Both the producer and consumer connect to the MOM using an API [1].

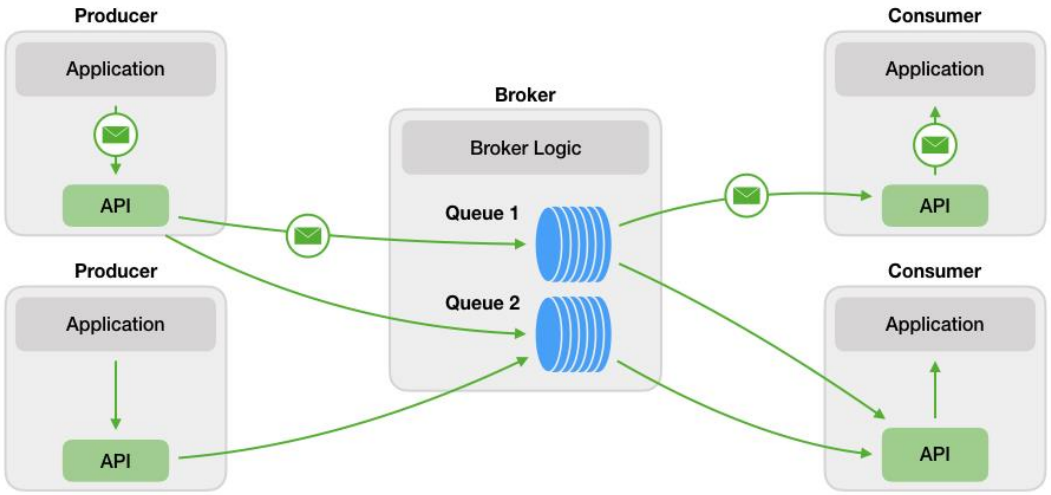


Figure 2.1: The general architecture of MOMs.

2.3 Message delivery modes

Most MOMs support two different types of main message delivery modes.

- **Point-to-point (Queue)**: Point-to-point delivery is when a producer sends a message and only one consumer receives it e.g. one-to-one communication. This means if multiple consumers listen for messages in the same queue, the broker will forward messages in a round robin fashion between the consumers [1]. Since each message is sent to only one consumer, this delivery mode is suitable for load-balancing work between the consumers.
- **Publish/Subscribe**: In this delivery mode consumers subscribe to a topic. When a producer sends a message to the topic, a copy of the message gets forwarded to all consumers that have subscribed to that topic. This is one-to-many communication and is generally used when the consumers are performing different actions on the same message. This delivery mode is often used in financial systems [3]. For example a consumer can subscribe to price updates for a certain type of stock.

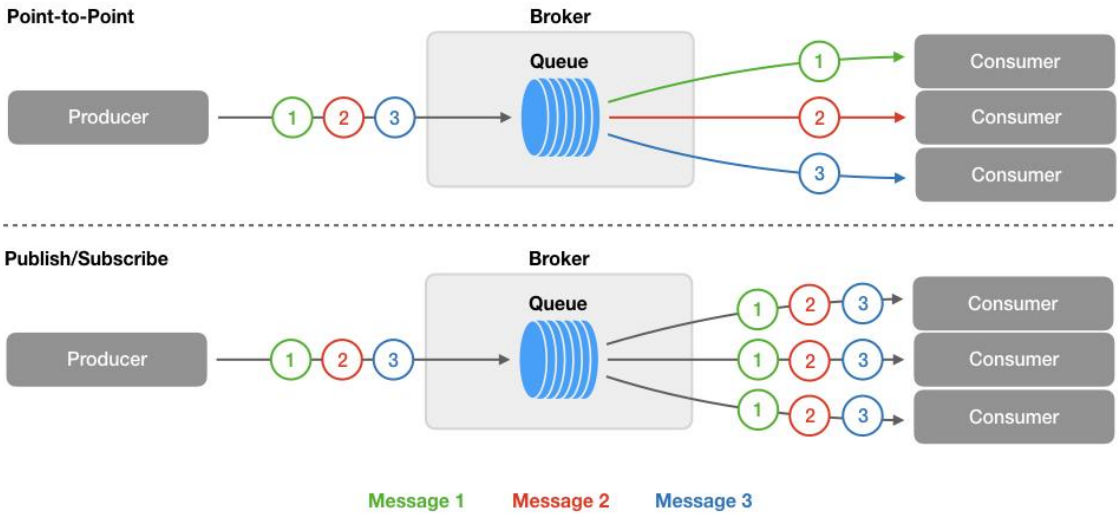


Figure 2.2: The difference between point-to-point and publish/subscribe delivery modes.

2.4 General use case and the asynchronous nature om MOMs

In Message-oriented Middleware for Scalable Data Analytics Architectures, Nicolas Nannoni summarizes the general use case for MOMs [4]. Nannoni states that MOMs are first and foremost an asynchronous "one-way" messaging technology. The producer sends a message to the message system broker and then continues with its operation. The message system is then responsible for delivering the message to the consumer. Contrasting this to a regular request-reply scheme where a client sends a request to some remote location and then waits until it gets a response. MOMs are not suited for traditional request-reply operations. This means that MOMs are better suited for writing values to a remote location rather than reading from a remote location. Traditional request-reply technologies are better suited for reading values from a remote location. Because reading a value would entail a request-reply operation where a client requests a value and gets a response. However, it is technically possible to use MOMs for asynchronous request-reply communication, even though it is not the intended primary use case. It would require additional logic to keep track of and handle the response when it arrives. It would also require logic to handle situations when there is no response.

This is reflected in the typical use cases of MOMs such as stream processing, log aggregation and website user tracking. All of these use cases involve moving values from some source to another location, they are in other words "one-way".

2.5 Advantages of using MOMs

The main advantage of using MOMs are listed below:

- **Asynchronous operations**: A producer can send a message to the MOM and continue with its operation. It does not have to block the thread that is responsible for sending the message to wait for a reply. Likewise, a consumer does not have to stop its operations to send back a mandatory

reply. According to Dobbelaere et al. [5] this leads to better resource utilization and improves performance.

- **Loose coupling**: Producers and consumers of messages do not have to know of each other's existence. Also, they do not have to exist simultaneously. According to Dobbelaere et al. this can make distributed systems more dynamic and flexible. Point-to-point communication makes systems more static and rigid.
- **Reduces connections**: With a standard client server model one unique connection is created between each endpoint of the client and the service. Sommer et al. points out how inefficient it is when the client has multiple connections [6]. Sommer et al. sees the number of connections decrease when each service can connect to a centralized broker.

2.6 Message guarantees

Different MOMs support different message guarantees and they often give users the opportunity to configure which message guarantees to use. The important guarantees are delivery schematics, persistence and ordering.

2.6.1 Delivery schematics

There are three levels of delivery schematics:

- **At-most-once**: When a MOM is using at-most-once delivery schematics it does not guarantee that all messages will arrive at the consumer, messages can get lost during failures. However, it does guarantee that a message will never be received by the consumer more than once.
- **At-least-once**: When a MOM is using at-least-once delivery schematic all the messages are guaranteed to be delivered to the consumer. This is achieved by a two-step acknowledgement scheme. The broker acknowledges all messages it receives from the producer and the consumer acknowledges all messages it receives from the broker. However, it is possible that the consumer receives messages twice when an acknowledgment gets lost during failures.
- **Exactly-once**: When a MOM is using exactly-once delivery schematics all messages are guaranteed to be delivered to the consumer without any duplicates. This requires a complicated acknowledgement and transaction scheme from the producer to the consumer. It is quite costly to provide exactly-once schematics, in this comparison the only MOM offering exactly-once is Kafka.

Requirements for delivery schematics depend on the use case. There are many use cases where at-most-once delivery schematics is enough, and a few lost messages is tolerable. For example, social media, online advertisement, and website tracking [7]. Another example where at-most-once guarantees is enough is when a MOM is used for ingestion to a machine learning application. Some lost messages do not affect the training of the model. However, some use cases require better delivery

schematics then at-most-once. For example, when MOMs are implemented in some financial systems, messages loss is not acceptable [3].

It is unusual for MOMs to offer exactly-once schematics. For many applications it is often better to use at-least-once combined with some logic that discards duplicated messages. [8].

2.6.2 Persistence

In the context of MOMs persistence refers to persistent storage of the queue and its messages. Some MOMs have the ability to store the entire queue in persistent storage, while others just keep the queue in main memory. Persistent storage can be important for some use cases. If the broker server crashes with the queue stored in memory, all the messages are lost which can be undesirable if many messages are stored in the queue. Even in applications that accept some message loss, losing the whole queue may be unacceptable. Some MOMs also support both storage types and the users can choose if they want to keep the messages in memory or store them on disk.

2.6.3 Message ordering

The concept of ordering is fairly simple, if a MOM supports ordering it just means that the messages arrive at the consumer in the same order as they were sent by the producer. Not all MOMs support message ordering and some let the user choose if they want message ordering or not. Some applications require message ordering, for example message ordering is required in some financial systems [3].

2.7 Performance

Performance (or efficiency) in MOM systems is primarily defined as latency and throughput [5].

- **Latency**: Latency is the time it takes for a message to travel from the producer to the consumer. Compared to regular point to point communication MOMs often have higher latency since there are more steps involved in transporting a message from the producer to the consumer. A message has to travel through the broker where operations related to queue management and routing has to be performed. The difference between point-to-point and MOM latency is shown in figure 2.3.
- **Throughput**: Throughput is generally defined as the amount of data that are transferred in a given time. In the context of MOMs throughput can also be viewed as the number of messages per time unit that is transferred. As described above there is queue management and routing operations that have to be performed for each message. Some operations take a fixed amount of time to perform independently of the message size [5]. In theory this would mean that larger messages result in higher throughput (bytes/s).

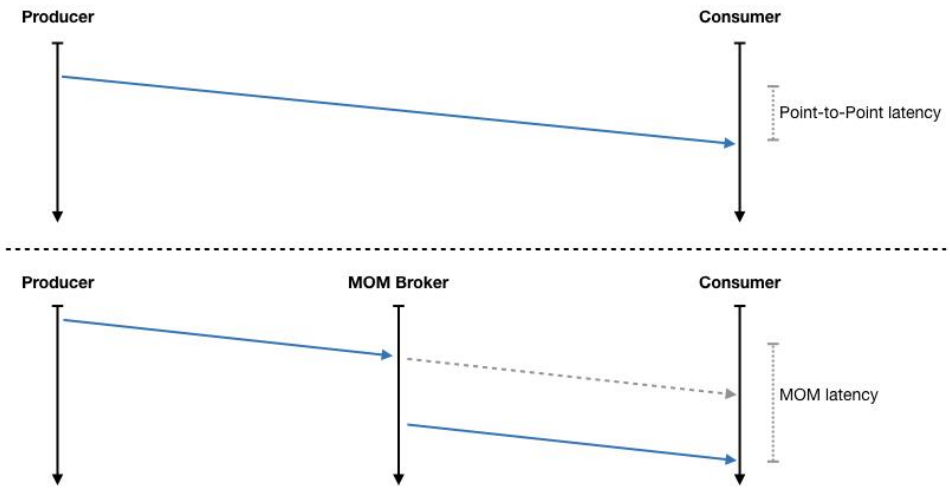


Figure 2.3: The difference between point-to-point latency and MOM latency. In the figure the MOM latency is larger because of the added latency from the MOM broker. (* The slope of the lines is intended to illustrate network latency)

MOMs are often used in real time applications, for example real-time bidding [9], therefore latency is important for MOMs. Some common use cases for MOMs include stream processing, website user tracking and big data ingestion. These use cases are synonymous with large quantities of data, this also makes throughput important for MOMs.

2.8 Performance vs message guarantees

It is important to know the relationship between message guarantees and performance. Message guarantees often come at the cost of performance because it entails additional overhead. For example, going from at-most-once to at-least-once requires additional acknowledgement messages to be exchanged, which consumes both network and processor resources [5]. This is true for many forms of communication, not just MOMs. For example, UDP has better performance compared to TCP, but it is also less reliable. Message ordering is another guarantee that consumes resources. Ordering requires additional processing steps to sort messages in the broker [5].

2.9 The message systems compared

There are a couple of key reasons behind the selection of these MOMs. Kafka is by far the most popular MOM with many new academic works each year [1]. One reason behind Kafka's popularity is its ability to offer good performance even though all messages that pass through the system are written to disk. Compared to Kafka, RabbitMQ is more of a lightweight system and can in some circumstances outperform Kafka [4]. Nats is a new and even more lightweight MOM. There have not yet been any papers published that compare the performance of Nats to more widely used MOMs like Kafka and RabbitMQ.

2.10 Apache Kafka

Apache Kafka was originally developed at LinkedIn; the initial purpose of Kafka was to handle log-data [10]. Kafka was open sourced in late 2010 and joined the Apache software foundation family in July 2011 [11]. Kafka relies heavily on Apache Zookeeper [4]. Kafka is written in Scala which is a functional version of the java programming language.

2.10.1 General architecture

The Kafka architecture follows the general architecture for MOMs described in 2.2. The main difference is that the queue in Kafka is made up of topics and partitions. This means that the broker does not contain queues, instead it contains topics which in turn is subdivided into partitions. When a producer sends a message using the Kafka broker it must specify which topic or partition it would like to append the message to. Likewise, the consumer must specify which topic or partition it wants to receive messages from. The general architecture of Kafka can be seen in figure 2.4.

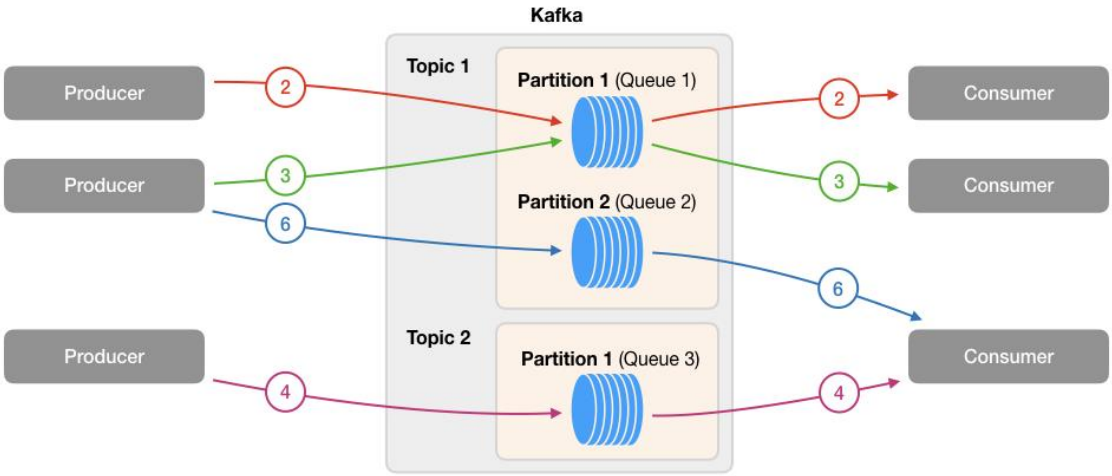


Figure 2.4: The general architecture of Apache Kafka.

In figure 2.5 the internal structure of the partition is displayed. Each message appended to the partition is assigned an offset, this message offset is used for two reasons. Firstly, it is used to keep track of message ordering within the partition. Secondly it is used by the consumer to keep track of which messages it has consumed.

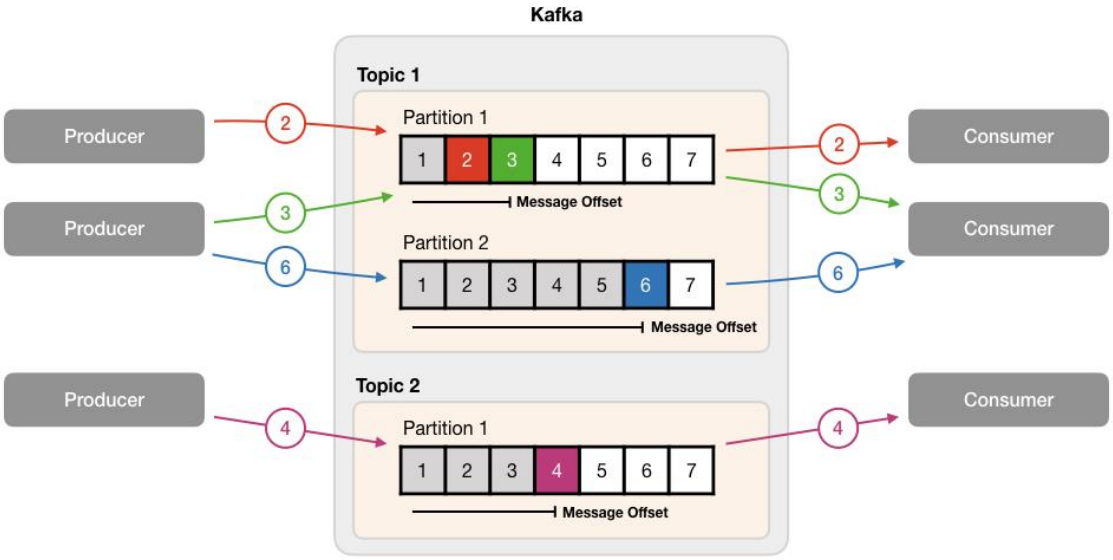


Figure 2.5: Partitions and offsets in Apache Kafka.

Topics and partitions have to be defined in advance by the administrator of the message system, they can be seen as configuration parameters. Only the administrator can add or delete topics and queues. Therefore, topics and partitions cannot be added or deleted dynamically at runtime by producers or consumers.

2.10.2 Message guarantees

The following list describes Kafka's message guarantees:

- **Delivery Schematics**: Kafka supports all three delivery schematics: at-most-once, at-least-once, and exactly-once. It is the only message system in this comparison that supports exactly-once.
- **Persistence**: All messages sent through Kafka are stored permanently on disk, it is not possible to choose to only store messages in memory. Persistence is a key part of the Kafka architecture. Note that I/O operations are traditionally quite slow, Kafka overcomes this by utilizing some clever paging techniques to increase the disk throughput [12].
- **Message Ordering**: Kafka only guarantees message ordering within a partition. This means that if a consumer pulls messages from multiple partitions Kafka cannot guarantee that they will be in the same order as they were originally sent.

2.11 RabbitMQ

RabbitMQ is based on the AMQP protocol (Advanced Message Queueing Protocol). AMQP originated at JPMorgan Chase in 2003 due to the lack of an open standard to interconnect systems at investment banks [13]. RabbitMQ is an open source project originally released in 2007 and is currently managed by Pivotal [4]. RabbitMQ is written in the Erlang functional programming language.

2.11.1 General architecture

The RabbitMQ broker seen in figure 2.6 contains exchanges and queues. Exchanges are used to route messages to queues. When a producer sends a message to RabbitMQ it specifies which exchange the message should be sent to. The exchange then routes the message to the right queue or queues based on the routing key in the message header.

The relationship between consumers and queues is important. If there are multiple consumers subscribing on the same queue, then the messages that are appended in the queue will be distributed to the consumers in a round robin fashion. This can be seen in figure 2.6 where queue 1 receives messages 1 and 2, because there are two subscribers to queue 1 the first message will go to consumer 1 while the second message will go to consumer 2. If however there is only one subscriber for each queue then the consumer will receive all the messages that are appended to that queue. This can be seen below, where queue 2 received message 3 and 4 and both messages are pushed to consumer 3.

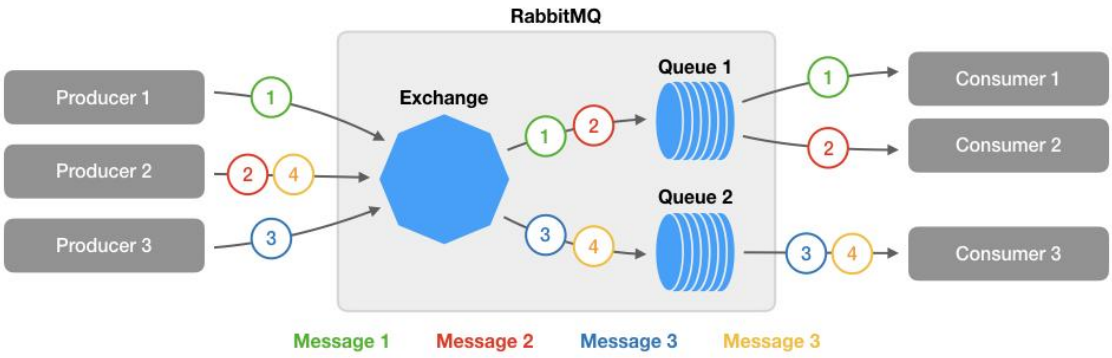


Figure 2.6: The general architecture of RabbitMQ.

RabbitMQ supports four different exchange types. The two important exchange types are direct and fanout.

- **Direct communication**: Direct communication is the default exchange type and is the equivalent of point-to-point communication. The exchange looks at the routing key of the message and sends it the corresponding queue. This exchange type is shown to the left in figure 2.7
- **Fanout**: Fanout is used for publish-subscribe communication. When sending messages to the fanout exchange, messages will be propagated to all queues bound to the exchange. This exchange type is shown to the right in the figure below.

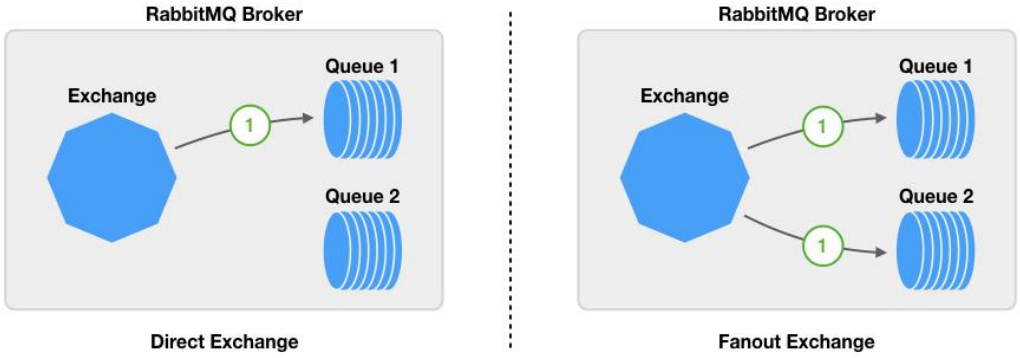


Figure 2.7: Direct exchange and fanout exchange.

The consumers can choose between a push or pull oriented API. In the push-oriented API, the messages are pushed by the broker to the consumer. In the pull-based API the consumer pulls the message from the broker.

2.11.2 Message guarantees

The following list describes RabbitMQ's message guarantees:

- **Delivery Schematics**: RabbitMQ supports both at-most-once and at-least-once delivery schematics.
- **Persistence**: RabbitMQ is mainly a memory-based message system. In the standard configuration messages are kept in main memory. However, messages can be flagged as persistent by the producer and be written to disk. When messages are dequeued by a consumer they are deleted [14]. It is not possible to keep dequeued messages on disk and replay them later which is possible with both Nats and Kafka.
- **Message Ordering**: All messages traveling in the same path are guaranteed to be delivered in the correct order. The definition of a RabbitMQ path is: producer → exchange → queue → consumer.

2.12 Nats

Nats stands for Neural Autonomic Transport System. Nats is written in GoLang, which is a functional programming language and the goal is to be simple, lightweight and fast. As with both RabbitMQ and Apache Kafka, Nats is an open source project. Nats comes in two different versions; "Nats Core" and "Nats Streaming", where Nats Streaming is a more feature rich version of Nats Core [15].

2.12.1 General architecture

Nats have a very simple structure which closely resembles the general architecture of MOMs described in section 2.2. The broker in Nats is called "Nats Server", producers and consumers are called "Nats Clients". The message system queue in Nats is called a subject. Producers send messages to subjects and consumers receive messages from the subjects which it has subscribed to. Consumers can subscribe to subjects individually which means that each subscriber receives a copy of the message added to the subject. Consumers can also subscribe as a queue group where only one of the consumers receives a message that has been added to the subject. The consumer that receives the message is chosen at random.

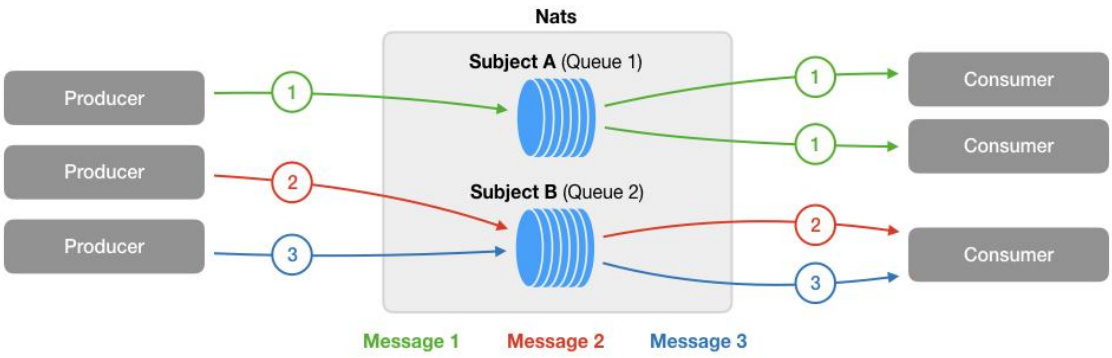


Figure 2.8: The general architecture of Nats.

2.12.2 Message guarantees

The following list describes Nats's message guarantees:

- **Delivery Schematics**: Nats Core supports only at-most-once schematics. If a subscriber is not active it will not receive the messages sent. Nats Streaming supports at-least-once schematics.
- **Persistence**: Nats Core does not support persistence. It is a fire and forget system, if there is no consumer active for a subject the message is discarded. However, Nats Streaming can store messages in both memory and on disk. Nats Streaming also supports message replay. This means that messages that have been received and acknowledged by consumers are not removed from memory. This makes it possible for consumers to go back and get older messages from the queue.
- **Message Ordering**: Both Nats Core and Nats Streaming supports the ordering of messages. However, they can only guarantee message order for each publisher.

2.13 Related work

The goal of this paper is to compare the performance of RabbitMQ and Kafka, but it will also include an additional message system - Nats, there have not been any peer reviewed papers that make performance evaluations of Nats yet.

Even if there are no papers on performance evaluation of Nats yet, there are several papers on performance benchmarking of other MOMs. Among these there are two papers published by KTH students: "Message-oriented Middleware for Scalable Data Analytics" by Nicolas Nannoni and "Comparing message-oriented middleware for financial assets trading" by John Eriksson. These papers include both RabbitMQ and Kafka, but they compare the message systems for a very specific implementation with very specific message sizes.

Two other papers on performance comparison is "The analysis of the performance of RabbitMQ and ActiveMQ" by Valeriu Manuel Ionescu and "Performance Evaluation and Comparison of Distributed Messaging Using Message Oriented Middleware" by Naveen Mupparaju. These papers do not include both RabbitMQ and Kafka which are the two of the message systems that will be compared in this paper.

Since the performance comparison will include both RabbitMQ and Kafka it is important that the related papers have included both of these MOMs. Therefore, the two papers that are most relevant for this comparison is "Kafka versus RabbitMQ" by Dobbelaere et al. and Message-oriented Middleware for Industrial Production Systems by Sommer et al.

2.13.1 Kafka versus RabbitMQ

In this paper Dobbelaere et al. makes a general performance comparison between the MOMs Apache Kafka and RabbitMQ. The paper compares how message

guarantee settings and message sizes impacts latency and throughput for both Kafka and RabbitMQ. This is important since different implementations have different requirements for message guarantee settings. The tests are performed over a single node to mitigate network layer effects on the test results. For the throughput tests Dobbelaere et al. uses message sizes between 0 and 2000 byte. Below are the summarized results from this paper.

- **Latency**: For at-most-once delivery schematics, latency is similar in both Kafka and RabbitMQ (under 10ms). For at-least-once RabbitMQ has the same latency as it has for at-most-once. When Kafka is running at-least-once, latency is doubled compared to at-most-once.
- **Throughput**: For at-most-once RabbitMQ showed a little better performance than Kafka. Both RabbitMQ and Kafka showed a significant decrease in performance for at-least-once. The throughput drops by 50% for RabbitMQ when running in at-least-once compared to at-most-once. For Kafka the throughput drops by 50% to 75% when running in at-least-once.

2.13.2 Message-oriented middleware for industrial production systems

In this paper, Sommer et al. investigates which MOMs are most suitable for transportation of messages in manufacturing applications [6]. Sommer compares a couple of message systems including Apache Kafka and RabbitMQ. The comparison consists of measuring latency and throughput for different message sizes. According to Sommer the message sizes in a manufacturing system ranges between 4 and 4096 bytes therefore these sizes are tested.

The tests show that Kafka generally has better performance in terms of throughput and latency than RabbitMQ. However, RabbitMQ has better performance for larger messages. The test also shows that RabbitMQ has more of a linear relationship between performance and message sizes.

Sommer et al introduces a clever way of measuring the maximum throughput capacity for MOMs. In short, the method uses latency measurements to determine the maximum throughput capacity. Sommer et al. calls the maximum throughput capacity for maximum sustainable throughput (MST). The method is described in more detail in chapter 3.

3 Method

Initially a literature study was carried out where previous work and platform documentation was investigated. From this literature study the method was devised, this chapter describes how the MOM comparison will be performed.

The method for comparison is based on the work by Dobbelaere at al. where the MOMs performance is compared with different message guarantees and different message sizes. The actual measurement of performance will be done with the method used by Sommer et al. The main difference between Dobbelaere et al. and Sommer et al. papers compared to this thesis is that Nats is included. Nats is a newer MOM without any peer reviewed papers yet. Another difference is that this comparison will also include a wider range of message sizes and a wider range of message guarantee settings.

The method section begins by describing the message guarantees settings and message sizes that were used in the comparison. The following sections describe how the testbed program was implemented and the computer hardware that was used.

3.1 Nats Streaming versus Nats Core

As mentioned in section 2.12 Nats comes in two versions; Nats Streaming and Nats Core. Comparing the performance of different message guarantee settings is an important part of this thesis. Since Nats Core barely offers any message guarantee settings, Nats Streaming was chosen instead. Hereafter Nats Streaming will be referred to as Nats.

3.2 Message guarantees

Below are the message guarantee settings that will be used for each MOM in the performance comparison. Section 2.6 describes all message guarantee settings in more detail.

- **Delivery schematics**: All MOMs will be tested with both at-most-once and at-least-once delivery schematics. This is the only two delivery schematics that are supported by RabbitMQ and Nats. Kafka also supports exactly-once but exactly-once will be excluded in this test because of the reasons described in section 2.6.1.
- **Persistence**: The MOMs will be tested with and without persistence, except for Kafka, which will only be tested with persistence. This is simply because it is not possible to turn off persistence in Kafka.
- **Message ordering**: This comparison will only test the message system with one producer and one consumer. Kafka is the only message system in this comparison that supports un-ordered communication when there is only one producer and one consumer. Both Nats and RabbitMQ require multiple consumers or producers in order to have un-ordered messaging.

Therefore, Kafka is the only MOM that will be tested with un-ordered messaging.

The list below shows all the permutations of message guarantee settings. Appendix A presents the configurations used to achieve each message guarantee.

Kafka (All persistent)

- At-Most-Once | Ordered
- At-Least-Once | Ordered
- At-Most-Once | Un-Ordered
- At-Least-Once | Un-Ordered

RabbitMQ (All ordered)

- At-Most-Once | Persistent
- At-Least-Once | Persistent
- At-Most-Once | Non-Persistent
- At-Least-Once | Non-Persistent

Nats (All ordered)

- At-Most-Once | Persistent
- At-Least-Once | Persistent
- At-Most-Once | Non-Persistent
- At-Least-Once | Non-Persistent

3.3 Message size

The MOMs will be tested with different message sizes, the message size range chosen is between 8 bytes and 1 MB. 8 bytes is the minimum size chosen because this is the size of a nanosecond timestamp (Long), which have to be included in each message in order for the testbed to be able to calculate the latency for each message. Both papers by Dobbelaere et al. and Sommer et al. tests for small messages e.g. under 4000 bytes. It is important to test larger sizes because some applications use MOMs to send messages larger than 4000 bytes, therefore the maximum message size is set to 1 MB. 1 MB is chosen because this is the default maximum message size for both Nats and Kafka.

The full range of message sizes chosen are: {8, 64, 512, 4096, 32768, 262144, 1048000}. Each message size added to the comparison adds 12 permutations of MOMs and delivery guarantees settings which corresponds to two additional hours of testing. Therefore, an exponential increase of the message sizes has been chosen as a compromise between keeping the test time reasonable while at the same time covering the whole range from 8 bytes to 1 MB.

3.4 Throughput and latency measurements

Section 2.7.1 describes latency as the time it takes for the message to travel from the producer to the consumer. Throughput is defined as the number of messages that can be sent through the system for a given time unit. Throughout this paper, milliseconds (ms) is used to measure latency and messages per second (mps) is used to measure throughput. The sections below describe how maximum sustainable throughput and the latency-throughput chart is determined for each MOM.

3.4.1 Maximum sustainable throughput

Sommer et al. uses the term maximum sustainable throughput (MST) to describe a message system's maximum throughput capacity. The goal for this paper is to determine the MST for each combination of MOM, message-guarantee setting and message size.

MST will be determined using a testbed, this testbed has a producer, consumer, and a MOM broker in between. The producer sends messages through the MOM broker to the consumer which measures latency and throughput. To find MST the producer starts off by sending messages to the consumer at a low rate. Then the producer slowly increases the message sending rate and the maximum throughput is recorded. When the message sending rate surpasses the MOMs maximum capacity two situations can occur:

- The first situation is when the throughput capacity from the broker to the consumer is equal or higher than the throughput capacity from the producer to the broker. In this case there will be no queue build-up in the broker since messages are leaving the broker faster than they are added to the broker.
- The second situation is when the throughput capacity from the broker to the consumer is lower than the throughput capacity from the producer to the broker. Now messages are added to the broker faster than they are leaving and there's a queue accumulating at the broker. Sommer et al. calls this saturation.

In the first situation determining MST is straightforward, MST is equal to the maximum throughput that was recorded by the consumer.

In the second situation, measurements are more complicated since the MST must be recorded right before there is a queue build-up. Sommer et al. describes a method for determining MST in this situation. When there is a queue build-up in the broker, the latency recorded by the consumer is increased, this is shown by figure 3.1. According to Sommer this can be used to identify the queue build-up. MST can be determined as the maximum throughput recorded right before there is a spike in the latency.

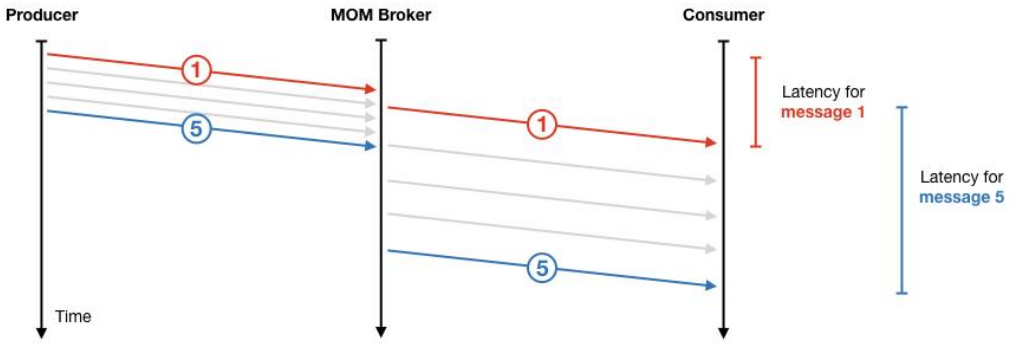


Figure 3.1: The latency of message 5 is higher than the latency for message 1 because the throughput capacity from the broker to the consumer is lower than the throughput capacity from the producer to the broker.

3.4.2 Latency

Latency will also be determined for each MOM. Compared to MST, latency cannot be determined as a single value for each MOM and respective configuration. This is because latency varies greatly depending on the throughput. Latency will instead be measured continuously and displayed in a latency-throughput chart. This also means that latency must be measured simultaneously as the throughput is measured. As with MST it is important not to include any latency measurements after the point of saturation. Mainly because latency measurements after the point of saturation will be inflated by the queue build-up.

3.5 Testbed

The test will be performed over a single computer to eliminate any network layer effects, which could interfere with the test results. Also, there is no need for synchronizing clocks between the machines. The consumer and producer in the testbed are built using java. Both the consumer and producer use the standard java client library associated with each MOM to connect to the respective broker.

3.5.1 Testbed program

There are many combinations of MOMs, message sizes and settings that will be tested in this comparison. Therefore, there must be a dedicated test software that can automate the testing of all MOMs with their respective settings and message sizes.

Figure 3.2 explains the automatic test program. The test is divided into test units. A test unit is a test for a specific combination of MOM and message guarantee setting. The four steps below explain the setup, execution, and termination of a test unit:

1. The main thread first starts the consumer thread. Before the thread is started it is configured with two parameters: MOM and message guarantee setting.
2. The main thread creates a producer object with the same settings as the consumer

3. In the next step the producer begins sending messages to the consumer. It starts with the smallest size (8 bytes) and works its way up to the maximum size of 1048000 bytes. For each message size, latency-throughput data is collected. Section 3.4.2 goes into further detail how this is implemented in the testbed.
4. The producer is done testing all message sizes and sends a “END” flag to the consumer and the thread closes.

When the unit-test is finished the main thread starts another unit-test for some other combination of MOM and settings and starts all over. It takes around 14 hours to finish all permutations of the test.

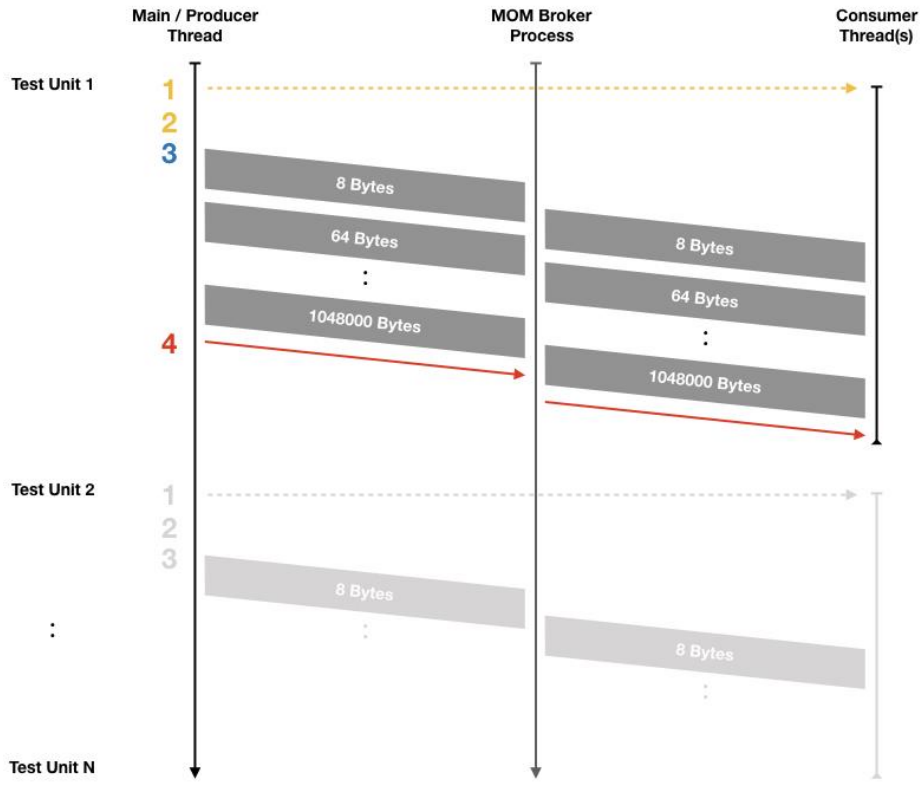


Figure 3.2: Testbed design - Step 1 and 2 starts the consumer and producer for a MOM and settings. In step 3, tests are executed for each message size. In step 4, a flag is sent indicating the test unit is finished.

3.5.2 Test for each message size

This section describes the test that is repeated for each message size.

Each of the messages contains a timestamp for when the message was sent and a payload that matches the message size. The consumer calculates the latency for each message by comparing the timestamp of the message with the current time.

Throughput is calculated by the consumer by counting how many messages is received each second.

The goal for this step is to generate data so that MST and the MOMs latency can be determined as explained in section 3.3 The producer starts sending messages at a low sending rate. It then raises the rate slowly to a sending rate that is above the maximum capacity for the MOM. For each message sending rate, the process described in the figure 3.3 is repeated:

1. Messages are sent for 10 seconds to warm up the broker - no message data is recorded by the broker.
2. The warmup step is ended with a “warm up done” flag.
3. The real messages used for data collection are sent. These messages are also sent for 10 seconds.
4. When the consumer receives the “real data done” flag it stores the mps and average latency for the messages received in step 3 to a file.

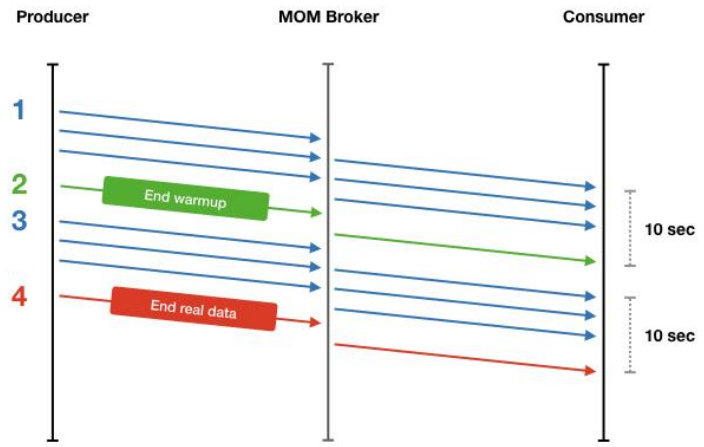


Figure 3.3: The test performed for each sending rate and message size.

3.5.3 Testbed hardware

The test will run on an Amazon Web Services instance, the specific type of instance and operation system used is described in table 3.1. Linux without any graphical user interface is used because it is important to have as much control over the process scheduler as possible. Other processes running on the system could interfere with the test results.

Table 3.1: Amazon ec2 t3.large instance specifications.

Operating system	Linux ubuntu 18.04
CPU	2 cores: 2.5 GHz Intel Scalable Processor
RAM	8 GiB
Secondary memory	80 GB, elastic block storage

4 Result

The following chapter presents the test results from benchmarking each MOM using the test architecture described in chapter 3. The chapter begins by describing how the data is aggregated and then presents the result for throughput and latency separately.

4.1 Latency and throughput measurement

In this paper, maximum throughput for a MOM is measured as MST (Maximum sustainable throughput), described in chapter 3. Figure 4.1 shows two latency-throughput charts with data collected from running RabbitMQ with the message size 4096 bytes, at-least-once schematics and persistence enabled. These charts will be used to describe how MST and latency is decided.

In figure 4.1(a) there is a significant jump in the latency where data points form a cluster in the top-right corner. This is an indication of a queue and that the MST has been reached. In this example, the queue buildup starts at 6560 messages/s and therefore the MST for this combination of MOM and settings is 6560 messages/s.

In figure 4.1(b) all the data points after MST have been excluded. These are the same data points as under the red line in figure 4.1(a). Figure 4.1(b) can be used to study the latency in detail, the latency is climbing with the increased throughput and peaks out at 5.94 ms.

The rest of the charts that have been used to decide MST and latency for each MOM and setting can be viewed in the Appendix B.

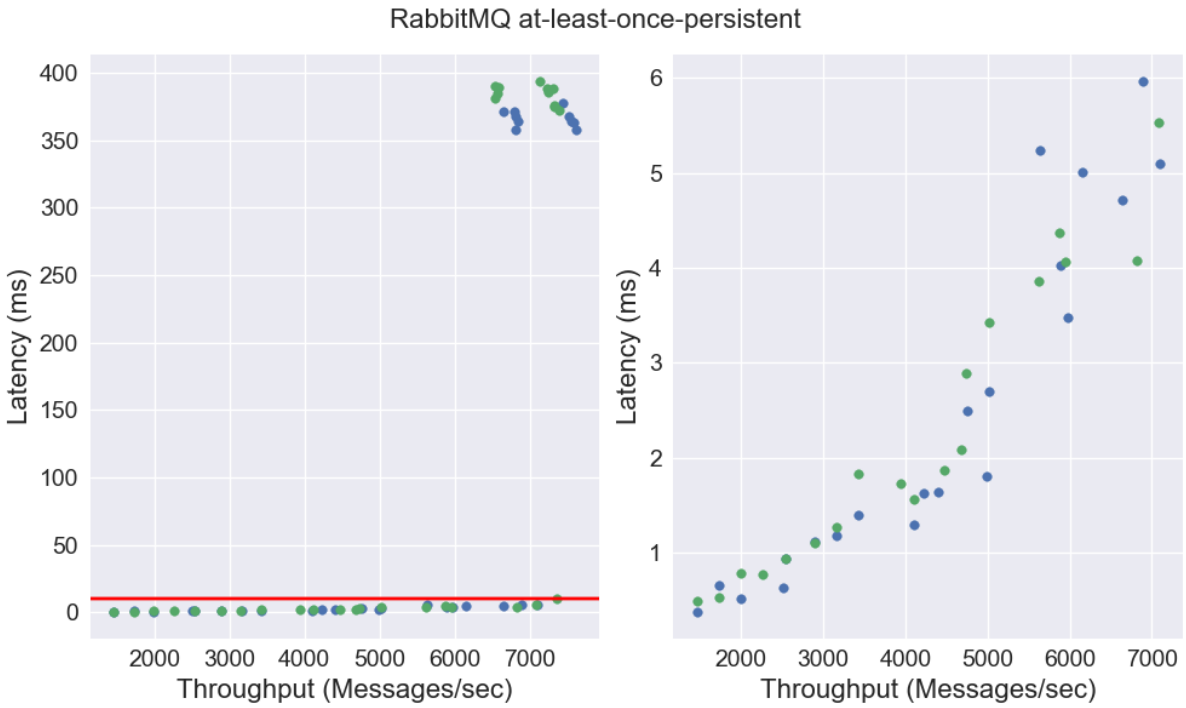
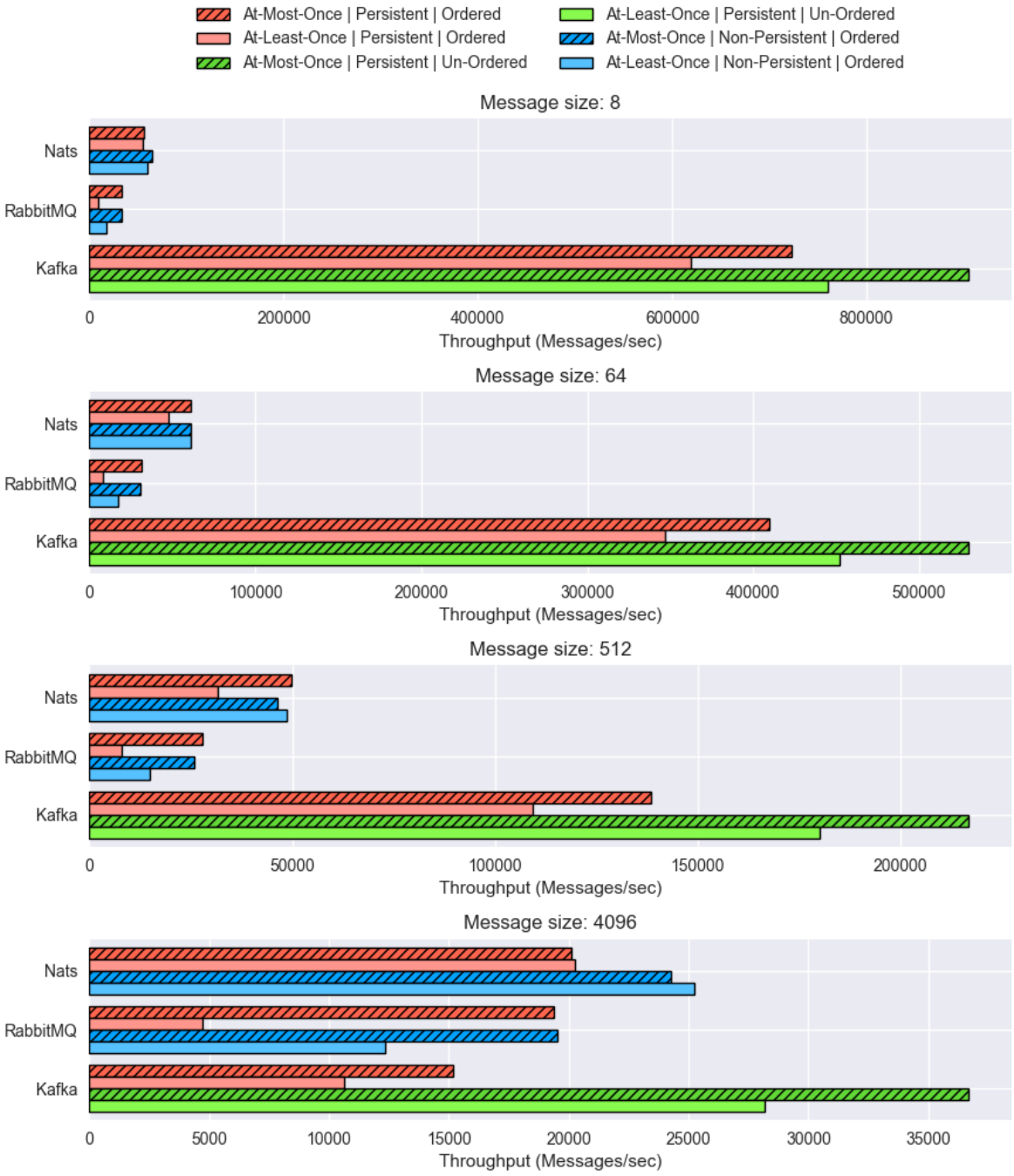


Figure 4.1: Data from running RabbitMQ with the message size 4096 bytes, at-least-once schematics and persistence enabled. (a) All data recorded, red line at 5.94 ms (b) Collected data under red line.

4.1.1 Throughput (MST)

Figure 4.2 shows the MST for each MOM and settings using vertical bar charts. If a particular MOM does not support a certain setting it is left out from the charts.


At-Most-Once | Persistent | Ordered
At-Least-Once | Persistent | Ordered
At-Most-Once | Persistent | Un-Ordered
At-Least-Once | Persistent | Un-Ordered
At-Most-Once | Non-Persistent | Ordered
At-Least-Once | Non-Persistent | Ordered
Message size: 8
Nats
RabbitMQ
Kafka
0
200000
400000
600000
800000
Throughput (Messages/sec)
Message size: 64
Nats
RabbitMQ
Kafka
0
100000
200000
300000
400000
500000
Throughput (Messages/sec)
Message size: 512
Nats
RabbitMQ
Kafka
0
50000
100000
150000
200000
Throughput (Messages/sec)
Message size: 4096
Nats
RabbitMQ
Kafka
0
5000
10000
15000
20000
25000
30000
35000
Throughput (Messages/sec)

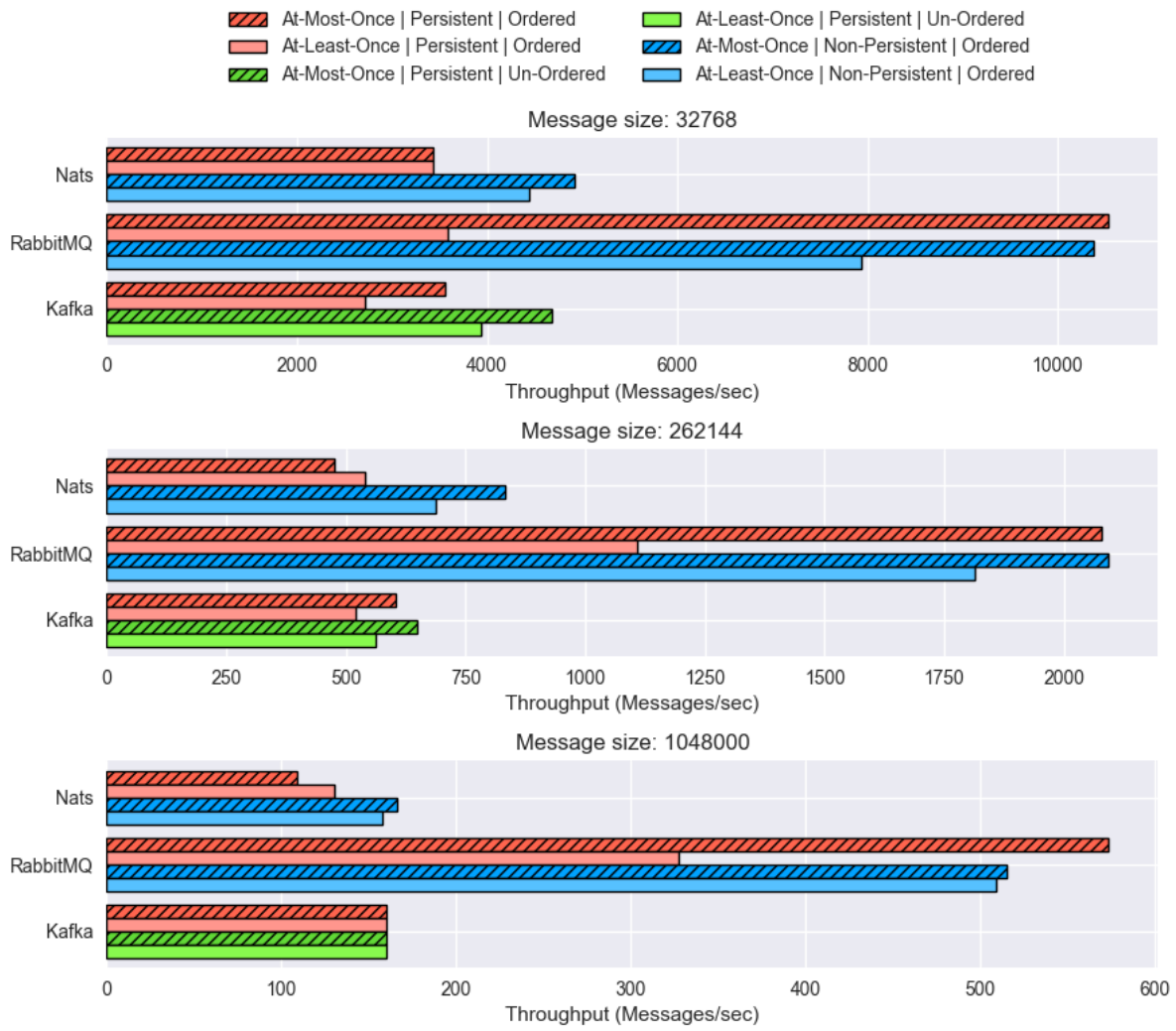


Figure 4.2: Vertical bar charts showing MST for all MOMs and settings. All message sizes are in bytes.

4.1.2 Latency

The following figures 4.3 through 4.6 shows the latency and how it changes in relation to the throughput. If a particular MOM does not support a certain setting it is left out from the figure. Datapoints after MST are excluded from the charts.

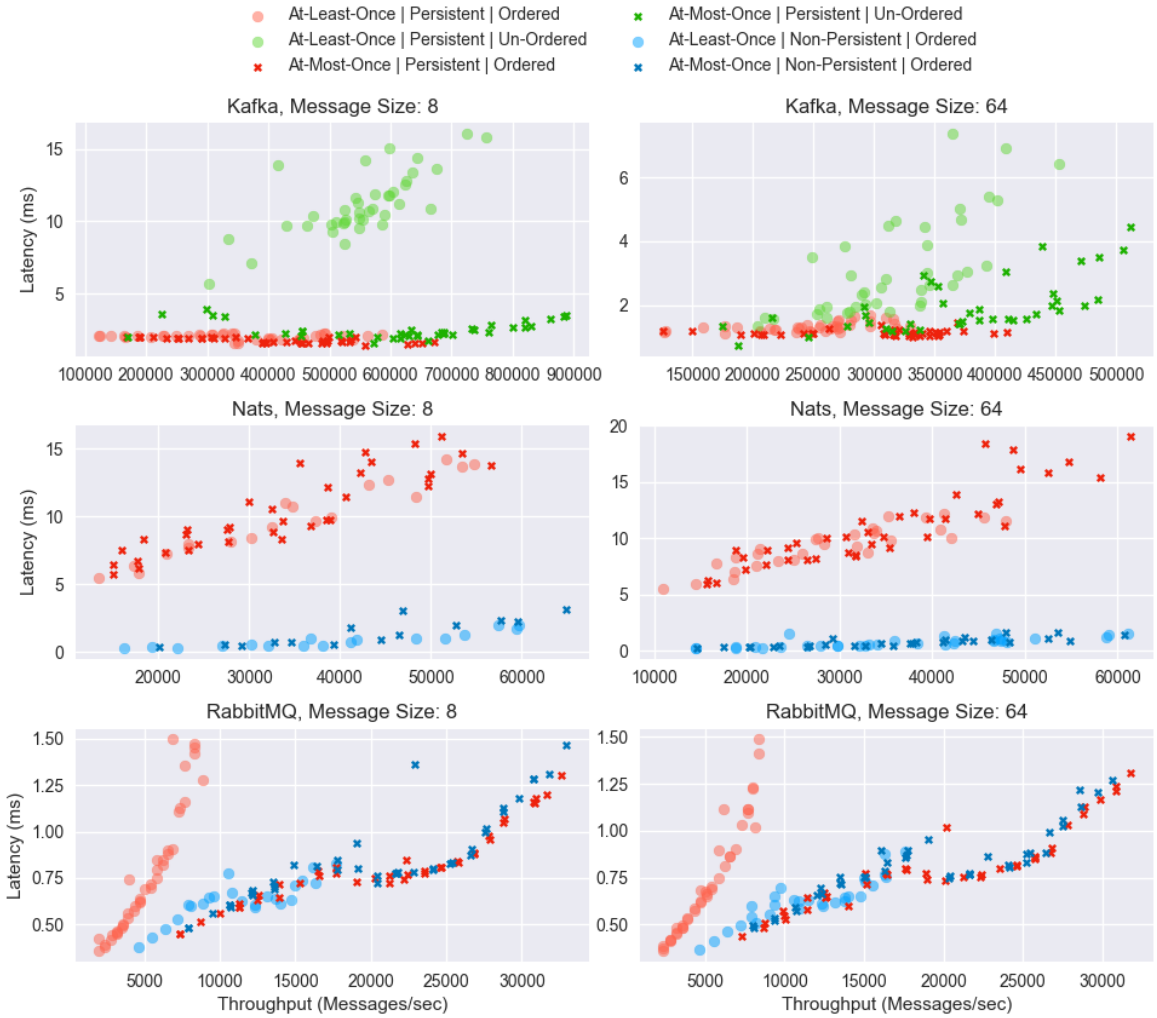


Figure 4.3: Charts showing the relationship between latency and throughput. (a) Message size 8 bytes, left side. (b) Message size 64 bytes, right side.

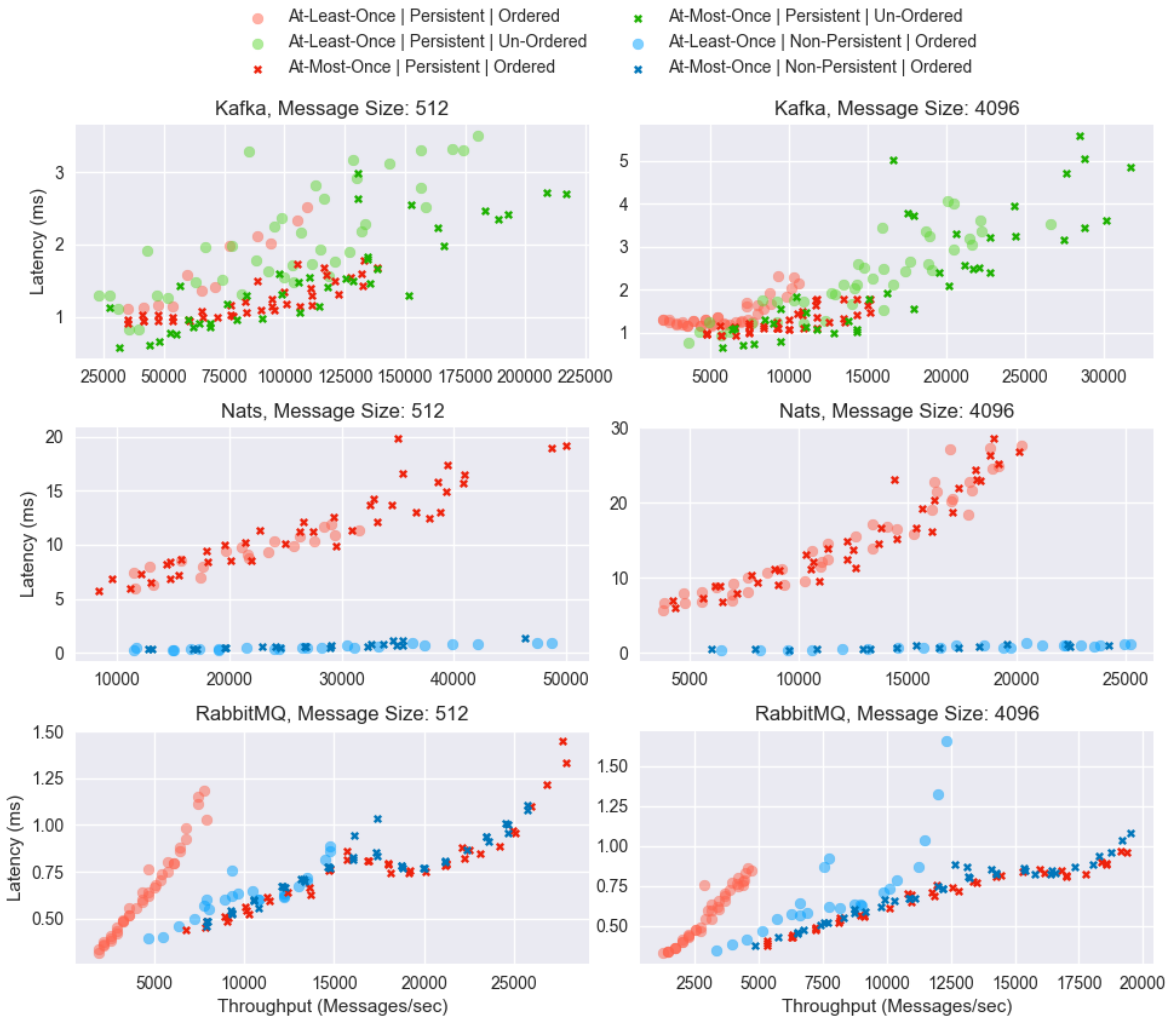


Figure 4.4: Charts showing the relationship between latency and throughput. (a) Message size 512 bytes, left side. (b) Message size 4096 bytes, right side.

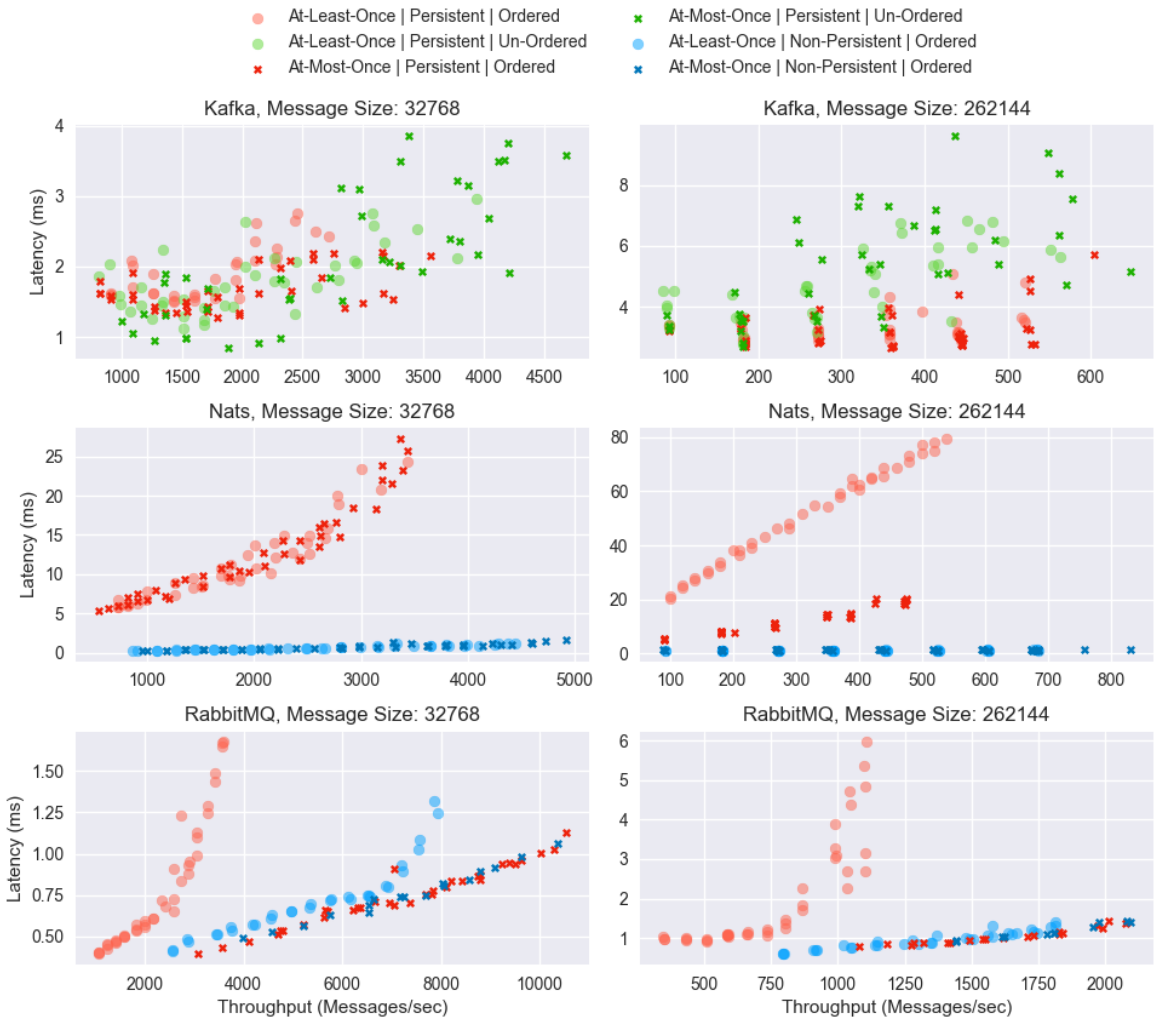


Figure 4.5: Charts showing the relationship between latency and throughput. (a) Message size 32768 bytes, left side. (b) Message size 262144 bytes, right side.

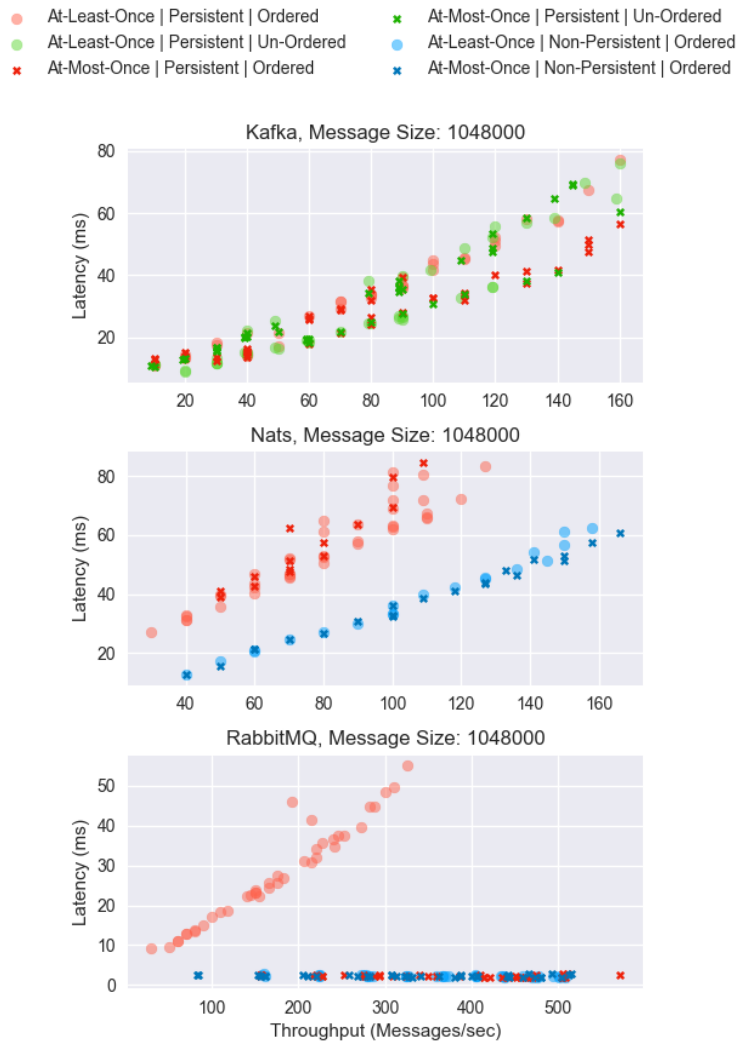


Figure 4.6: Charts showing the relationship between latency and throughput for message size 1048000 bytes.

5 Analysis and discussion

This chapter begins with analysis and discussion of the collected data presented in chapter 4. This is followed by a discussion of what other methods could have been used to perform the benchmark and potential improvements. Then different use cases for each MOM is discussed. Economic and environmental impacts of the research are discussed in the last section.

5.1 Throughput performance differences between MOMs

The discussion is based on the data provided in figure 4.2. As a rule, for all MOMs, throughput is decreased when stronger message guarantee settings are added. For example, at-least-once almost always entails lower throughput compared to at-most-once.

5.1.1 Kafka

For smaller message sizes, Kafka has a significantly higher throughput than the other two MOMs. This is probably because Kafka uses batching, smaller messages are bundled together into larger messages before they are sent to the broker. RabbitMQ and Nats do not natively support this feature. It could be argued that message batching should be turned off for a more equal comparison. However, it is a central feature of Kafka and when it is turned off, Kafka performs very poorly. In the tests the default batch size of 16384 bytes is used.

It is also noticeable that the un-ordered configuration results in a higher throughput, especially for 512 bytes and 4096 bytes messages. For these message sizes the throughput is almost doubled compared to the ordered configuration. At-least-once message schematics results in a 15-20% lower throughput compared to at-most-once.

5.1.2 RabbitMQ

RabbitMQ performs poorly for smaller messages (512 bytes and smaller) compared to both Nats and Kafka. However, RabbitMQ has the best performance in terms of throughput for larger message sizes (32769 bytes and larger). Compared to the other MOMs, the throughput for RabbitMQ is decreased significantly when using persistence in combination with at-least-once schematics. RabbitMQ at-least-once non-persistent has around 30% lower throughput compared to at-most-once non-persistent. While at-least-once persistent has around 65% lower throughput compared to at-most-once persistent.

5.1.3 Nats

When looking at the throughput bar chart for message sizes up to 4096 bytes it can be a bit misleading when comparing Nats with Kafka. Nats does not use batching, but still archives a better throughput than Kafka for the 4096 bytes message size. This is probably because Kafka starts to lose its message batching advantage since it can only pack 4 messages in one batch (16394/4096 = 4). When just comparing a

MOMs ability to route single messages up to 4096 bytes in size from A to B, Nats probably have the highest throughput of all 3 MOMs.

For Nats there is a very little difference in throughput when comparing at-least-once with at-most-once settings. At-least-once has only around 5% lower throughput compared to at-most-once.

5.2 Latency performance differences between MOMs

All MOMs in this comparison have, in some measure, a linear relationship between latency and throughput. This can be seen in the latency-throughput charts in section 4.1.2. In the benchmark results the latency starts off low, when the throughput gets higher the latency increases. This is true for all instances except Kafka with message size 8, this will be discussed further in 5.3.1.

Since the latency varies in relation to the throughput it is difficult to specify a specific latency for a certain combination of MOM, setting and message size. However, most MOMs and configurations have a latency below 5.0 ms.

5.2.1 Kafka

Kafka's latency is affected by the batching of messages. Because messages must wait for the entire batch to be filled prior to being sent off to the broker. This increases the baseline latency for Kafka which is higher than the baseline latency of the other MOMs. This can be seen in the latency-throughput graphs in Appendix B. For a message size of 8 bytes the latency starts off high then drops down and then starts to climb again. The latency starts off high because the throughput is low, and it takes longer time to fill each batch.

It is also noticeable that Kafka's latency for 1 MB messages is significantly higher than for the other message sizes, reaching 80 ms at MST.

5.2.2 RabbitMQ

RabbitMQ has an overall low latency. Persistence or at-least-once schematics does not seem to affect the latency. However, the combination with at-least-once schematics and persistence results in a slightly higher latency. The only time RabbitMQ has a high latency is for at-least-once-persistent and with a message size of 1 MB, where the latency is significantly higher than for the other message sizes.

5.2.3 Nats

As with RabbitMQ, Nats does not use batching therefore the baseline latency is almost zero. For Nats, at-least-once schematics does not seem to affect the latency of the message system. Persistence on the other hand raises the latency significantly. Similar to Kafka, the latency for 1 MB messages is significantly higher than for the other message sizes

Note that Nats will kill slow consumers when they cannot consume messages at a sufficient rate. This is the reason why there is no significant clustering in the top right corner of the latency-throughput charts in Appendix B.

5.3 MST and general behaviour of the MOMs

This thesis used a similar method to Sommer et al. (described in chapter 3) to determine the MST for each MOM. Each message system behaved as described by Sommer et al. where the latency will eventually see a sharp incline as the throughput is raised slowly in the testbed. According to Sommer et al. this latency spike was a result of queue build-up in the broker. After running the MOMs in the testbench, no queue build-up in the MOM brokers was recorded. The queue was instead accumulated in buffers either in the producer or in the consumer. It seems like the MOMs are designed to be able to push messages to the consumer faster than it lets the producers add messages. This could also be observed when the latency would not continue to accumulate but rather stay at a fixed level. This indicates that it is a queue of a fixed length rather than the “infinite” queue in the broker. For Kafka, the latency spike is caused by messages accumulated in the producer buffer. For Nats and RabbitMQ it is instead the consumer message buffer that causes the latency increase.

5.4 MST versus fixed latency

Determining MST for each MOM is a manual process and for some MOMs and settings, identifying exactly where the buffer build-up starts can be difficult. However, another method could have been used. If the buffer size is made smaller there will be no buffer build-up and hence no latency build-up when the maximum throughput is reached. This way, identifying the maximum throughput would just be a matter of recording the highest throughput that is reached in the testbed. Unfortunately, because of time limitations this was not investigated fully. It is straightforward to shrink the buffer size in Kafka. For Nats and RabbitMQ however, no solution was found. Given more time it could have been possible to figure out how to shrink the buffer size in all MOMs and thus make the throughput measurements simpler and potentially more accurate.

5.5 Single machine versus distributed machines

In this benchmark the producer, consumer and broker ran on the same machine. In a real-life implementation, all these components would most likely be running on separate machines and communicate over a network. This could have been simulated better if the testbed producer, broker and consumer was running on separate machines. Which would be the same method used by Nannoni. Another testbed implementation would be to run both the producer and consumer on the same machine and the broker on a separate machine.

If the testbed was distributed over multiple machines it could reduce the impact of resource sharing. In the development phase the effects of resource sharing could be seen when using a busy-wait subroutine to regulate the message rate. This

subroutine affected the performance of the MOMs since it wasted a lot of resources. The performance was improved by using another method for rate limitation. That being said, the resource sharing problem could be totally eliminated by running the rate limitation on a separate machine.

However, there are some drawbacks of having a distributed testbed. As stated in section 3.4 A distributed testbed can introduce network layer effects, which could interfere with the test results. Another problem with a distributed testbed is that latency measurements require synchronized clocks. Because of the drawbacks listed above a single machine testbed was used.

5.6 Use cases for each MOM

For applications where the message sizes are small (up to 512 bytes), Kafka is the best choice, since it has good latency and by far the highest throughput for all possible message guarantee settings. This makes Kafka the best choice for applications such as transportation of IOT sensor values and website user tracking. For applications where the message sizes are over 32768 bytes, RabbitMQ is the preferred MOM, since it has good latency and better throughput than the other MOMs. This makes RabbitMQ the best choice for applications with larger message sizes, such as ingestion of images for image processing.

Determining which MOM has the best performance for message sizes around 4096 bytes is not as straightforward. It depends on the message guarantee requirements of the application. As can be seen in figure 4.2, Kafka has the best performance for applications that do not require ordering. For applications that require ordering but not persistence, Nats is instead the best choice.

The result also shows that for most message sizes and settings, Nats is the second best performing MOM. Therefore, Nats could be the best choice for applications that have a wide range of message sizes. However, for applications that require low latency and persistence, Nats is not a recommended choice, since persistence configuration raises the latency significantly.

5.7 Economic and environmental impacts

The choice of a MOM could have an economic impact for companies or organisations. A better performing MOM will use less computing resources, which can save costs. The results show that Kafka has a throughput which is around 30 times higher than RabbitMQ for 8 bytes messages. For this message size, using RabbitMQ would require additional nodes to achieve the same throughput as Kafka can achieve using only one.

This thesis has not measured exact resource utilization of the MOMs. It is not possible to correlate the resource utilization with the maximum throughput of a MOM. For example, Kafka, RabbitMQ and Nats each have a MST of around 750000, 59000 and 23000 mps for the message size 8 byte. Even if Kafka has the

highest MST, it does not mean that Kafka has the lowest resource utilization for 20000 mps.

Computing resources also consumes energy, choosing a more effective MOM for a certain application could therefore have a less environmental impact. No social impacts are related to this research.

6 Conclusion

The goal was to make a performance comparison of three MOMs; Kafka, RabbitMQ and Nats (Streaming). Part of the goal was to compare the MOMs with different message sizes and different message guarantee settings. Generally across all MOMs, throughput is decreased when stronger message guarantee settings are added. Most MOMs and configurations have a latency below 5.0 ms.

- Kafka is the best performing MOM for smaller message sizes (under 512 bytes), regardless of the message guarantees chosen. This is primarily because Kafka uses batching which the other MOMs do not. Kafka is also the only message system in the comparison that has been tested with un-ordered messages. When Kafka uses un-ordering the performance is drastically improved.
- RabbitMQ has the best performance for larger message sizes (over 32768 bytes), regardless of the message guarantees chosen. RabbitMQ performance is decreased significantly when using persistence in combination with at-least-once schematics.
- Nats only outperformed the other message system for a few combinations of message guarantee settings with the message size 4096 bytes. However, Nats is the second-best performing MOM for almost all message sizes and settings. This could make Nats the best MOM for handling messages of varying sizes. However, Nats have poor latency when persistence is enabled.

6.1 Future work

In this comparison all the components of the testbed run on a single computer, this can create some resource sharing problems, which in turn can affect the test results. Therefore the same kind of test should be performed in a distributed environment to exclude potential interference of resource sharing.

The MOMs were compared using only one producer and one consumer. In many real world implementations there are often several producers and consumers. Ideally the comparison should be extended and include multiple producers and consumers.

MST for each MOM was determined manually, which might have induced some errors in the results. It might be possible to shrink the buffer size to eliminate queue build-up in the buffers. This would probably make the performance measurements more accurate and should be investigated further.

References

[1] Yongguo J, Qiang L, Changshuai Q, Jian S, Qianqian L. Message-oriented Middleware: A Review. In2019 5th International Conference on Big Data Computing and Communications (BIGCOM) 2019 Aug 9 (pp. 88-97). IEEE.

[2] Bernstein PA. Middleware: a model for distributed system services. Communications of the ACM. 1996 Feb 1;39(2):86-98.

[3] CHEN, Yuting; MAO, Ting; YU, Bo. A reliable messaging middleware for financial institutions. In: Proceedings of the 3rd International Conference on Communication and Information Processing. 2017. p. 108-112

[4] Nannoni N. Message-oriented Middleware for Scalable Data Analytics Architectures [Internet] [Dissertation]. 2015. (TRITA-ICT-EX). Available from: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-167139

[5] Dobbelaere P, Esmaili KS. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. InProceedings of the 11th ACM international conference on distributed and event-based systems 2017 Jun 8 (pp. 227-238).

[6] Sommer P, Schellroth F, Fischer M, Schlechtendahl J. “Message-oriented middleware for industrial production systems”. In 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE) 2018 Aug 20 (pp. 1217-1223). IEEE.

[7] Nag K S. A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming. arXiv. 2019 Dec:arXiv-1912.

[8] Kreps J, Narkhede N, Rao J. Kafka: A distributed messaging system for log processing. InProceedings of the NetDB 2011 Jun 12 (Vol. 11, pp. 1-7).

[9] Wiatr R, Słota R, Kitowski J. Optimising Kafka for stream processing in latency sensitive systems. Procedia Computer Science. 2018 Jan 1;136:99-108.

[10] Lugnegård L. Building a high throughput microscope simulator using the Apache Kafka streaming framework [Internet] [Dissertation]. 2018. (UPTEC F). Available from: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-340014

[11] Narkhede N, Shapira G, Palino T. Kafka: the definitive guide: real-time data and stream processing at scale. " O'Reilly Media, Inc."; 2017 Aug 31.

[12] Confluence “DOCUMENTATION” https://kafka.apache.org/documentation/#uses Retrieved 2020-05-11

[13] O'Hara J. Toward a commodity enterprise middleware. Queue. 2007 May 1;5(4):48-55.

[14] Gavin M. Roy, “RabbitMQ in Depth”, ISBN: 9781617291005, Manning Publications, 2017.

[15] Synadia, “Introduction” https://docs.nats.io/ Retrieved 2020-05-13

Appendix A – MOM configurations

This section describes how each MOM was configured to achieve each message guarantee.

Table A.1: How each MOM is configured to achieve each message guarantee.

	Kafka	**RabbitMQ**	**Nats**
At-Most-Once	**Producer settings** Acknowledgements and retries disabeled acks = 0, retries = 0 **Consumer settings** Auto commit set to true and auto commit interval lowered enable.auto.commit = true, auto.commit.interval.ms = 5000	**Producer settings** Publish confirms disabled Acks = 0 **Consumer settings** Auto acks set to true. basicConsume(..., autoAck = true, …);	**Producer settings** Create an AckHandler without any logic in it. The producer is synchronous otherwise. **Consumer** Standard configuration
At-Least-Once	**Producer settings** Acknowledgements and retries enabled acks = 1, retries = 1 **Consumer settings** Auto commit set to false and auto commit interval raised enable.auto.commit = true, auto.commit.interval.ms = 999999999 After each message fetch a commit will be performed. kafkaConsumer.commitSync()	**Producer settings** Enable acknowledgements from broker to producer Channel.confirmSelect() Confirm Callback that resends nacked massages. **Consumer settings** Set auto ack to false. basicConsume(..., autoAck = false, …);. The deliver callback acks all messages	**Producer settings** Create an AckHandler that resends messages on failure. **Consumer settings** Enable manuals acks options = SubscriptionOptions.Builder(). manualAcks().build(); messageHandler that acs all messages. StreamingConnection.subscribe(..., options, …);
Unordered	**Broker settings** The topic used was configured with eight partitions.	Not supported (Multiple producers or consumers required)	Not supported (Multiple producers or consumers required)

Ordered	**Broker settings** The topic used was configured with a single partition	Standard ordering	Standard ordering
Persistent	Standard configuration	**Producer settings** Queue durable = true queueDeclare(..., durable = true, …) Each message sent is flagged with a persistent message property. properties = MessageProperties.MINIMAL _PERSISTENT _BASIC **Consumer settings** Queue durable = true queueDeclare(..., durable = true, …)	**Broker settings** Start file store in the command line argument when launching nats-streaming server nats-streaming-server -store file -dir datastore
Non Persistent	Not Supported	**Producer setting** Queue durable = true queueDeclare(..., durable = true, …) properties = MessageProperties. BASIC **Consumer setting** Queue durable = false queueDeclare(..., durable = false, …)	Standard configuration

Appendix B – Latency-throughput charts

The section contains all latency-throughput charts used to determine MST and latency. Each file is named after MOM and message guarantee setting. The top charts contain all data points collected. While the bottom charts contains the data points under the red line in the charts above. The bottom charts has a green and a red line which divides each chart in four quadrants. The data points used to determine MST and latency are located in the 3rd quadrant.

URL B.1: All latency-throughput charts used to determine MST and latency.

https://drive.google.com/drive/folders/1QzEu7_-cUIq1dwsKXUewV4EUPQPOg_Vr

TRITA TRITA-CBH-GRU-2020:250