Collecting Information from a decentralized microservice architecture

Carl Ekbjörn
Daniel Sonesson

Tutor, Jonas Wallgren
Examiner, Lena Buffoni
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Carl Ekbjörn, Daniel Sonesson
Abstract

As a system grows in size, it is common that it is transformed into a microservice architecture. In order to be able monitor this new architecture there is a need to collect information from the microservices. The software company IDA Infront is transitioning their product iipax to a microservice architecture and is faced with this problem. In order to solve this, they propose the use of a Message-oriented Middleware (MOM). There exists many different MOMs that are suitable to execute this task. The aim of this thesis is to determine, in terms of latency, throughput and scalability, which MOM is best suitable for this. Out of four suitable MOMs Apache Kafka and RabbitMQ are chosen for further testing and benchmarking. The tests display that RabbitMQ is able to send single infrequent messages (latency) faster than Kafka. But it is also shown that Kafka is faster at sending a lot of messages rapidly and with an increased number of producers sending messages (throughput and scalability). However, the scalability test suggests that RabbitMQ possibly scales better with a larger amount of microservices, thus more testing is needed to get a definite conclusion.
6.4.2 Validity ......................................................................................................................... 37
6.5 Sources .......................................................................................................................... 37

7. Conclusion ........................................................................................................................ 39
  7.1 Future work .................................................................................................................. 39

8. References .......................................................................................................................... 41
1. Introduction

The use of software and the features of programs are rapidly increasing in the modern world. Because of this it is natural that software systems and applications grow in size and complexity. One single system can be designed to perform a large number of different tasks in order to achieve its main purpose. As a system grows, the time and effort put in to managing and developing it also increases. If the system grows too big, it could become next to impossible to manage in a viable and efficient manner.

One way to solve this is to divide a big system into smaller applications called microservices[12]. These microservices are designed to perform only one task each and can execute this task independently. A number of different microservices could then be joined and communicate with each other to form a large application. As these microservices can be updated or replaced without interfering with other components in the system, managing the system is greatly facilitated.

In a big complex architecture, it is likely that something will go wrong. In such a system with possibly hundreds of microservices it is important to have a way to monitor in real time what is happening. This aids in getting a holistic view of the system, find bugs and locate in which microservice faults are happening.

A possible way to collect information from the microservices, in order to monitor them, is to use a Message-Oriented Middleware (MOM) [5]. MOM is software that allows you to collect data from one application and stream it to another in real time, thus allowing you to stream information about all the microservices to other applications where you can for example visualize, analyze, and store the data.

Several pieces of MOM software exist today and they use different kinds of techniques to stream data. Because of this it can be difficult to know which one to use. Thus, the question at issue in this paper is:

- Which of the four chosen MOMs best suits, in terms of scalability and performance, gathering information from a decentralized microservice architecture?
2. Background

This thesis was written at and with the help of IDA Infront, a Linköping based IT-company formed in 1984. They provide digital solutions to authorities, organizations and private businesses with their product iipax. The following sections will give a brief explanation of iipax, explain why IDA Infront is interested in this thesis, and cover how the question at issue came to be.

2.1 iipax

iipax is a product family created by IDA Infront which provides functionality such as secure communication, case/information management, and digital archiving. The family is made up of three main components [15]:

- **iipax permission**: A tool that supports the information management in an organization or business. It provides document management, case management and process control.
- **iipax communication**: Provides secure communication for an organization or business. iipax communication acts as a middleman and enables applications to exchange information in a general and secure way.
- **iipax archive**: A digital archive solution. It also provides a tool that converts files to the correct format before they are archived.

All of these components are separate applications that run independently and can be configured and designed to fit the special needs of IDA Infront’s different customers. The work in this report is centered around iipax archive, therefore a more thorough explanation of that component follows.

2.2 iipax archive

The iipax archive component is made up of two parts, one that handles the pre-ingest process and one that is the actual archive (see Figure 1). The essential part for this report is the pre-ingest process. Because of this the archive part will not be discussed and explained further.

![Figure 1: A model representing the monolith archiving structure in iipax.](image)

Before a file can be archived it needs to be processed by the pre-ingest procedure. This procedure is structured as a pipeline, where each file is inserted and then processed in different steps. The steps include identification (a), conversion (b) if needed and afterwards insertion (c) into the archive. Usually this procedure is made up of a monolith architecture (see Figure 1) with one single application performing each step of the pipeline. However, in one configuration of the system the monolith architecture has been replaced by a microservice architecture (see Figure 2).
Figure 2. A model representing the microservice archiving structure in iipax.

With this configuration the identification part and conversion part of the pipeline have been implemented as independent microservices. This means that when the main application wants to either identify or convert a file it sends a request to one of these microservices and then receives a response depending on its success (see Figure 2). This is beneficial as it simplifies managing and supporting the system. Since these microservices run independently, either updating or replacing them can be done without interfering with any other components in the application.

These microservices respond to the main application with the status their operation had, either with the format of the identified file or with the result of the performed conversion. The response also includes metadata about the performed operation - information such as the time the operation took, the converter that was used, file size and more. This metadata is insignificant to the pre-ingest process and the system functions without it. However, the information has value as it can be used to monitor the state of the microservices and as a debugging or future development tool. Therefore, Ida Infront wants a way to collect this information.

The proposed way is to introduce an information bus to the system (see Figure 3). The microservices will connect to the bus and send their metadata to it instead. The information can then be extracted from the bus to either be analyzed, visualized, or stored. Thus, it is important for the bus to be able to filter different types of messages to different receivers.
As of now only these two microservices exists in IDA Infront’s systems. However, IDA Infront has a vision to transform more of their applications from a monolith architecture to a microservice one. In order to make this new architecture viable it is vital to find a suitable way to implement this information bus.
3. Theory

This section will explain the theory needed to understand the work in this thesis. It will explain what a Message-Oriented Middleware (MOM) is and how it works. Four different examples of MOM-software will be studied in more detail. We will also explain what microservices are and give a more technical view of how they are functioning in ipax.

To understand what a Message-Oriented Middleware (MOM) is, we will look at the three typical steps of a data streaming pipeline:

**Data collection:** Data is collected by a producer. A producer can either be a sensor, an event generator, or a software application.

**Data streaming:** The collected data is put through a message broker (a MOM) that puts the data in streams so it can be received by one or several applications.

**Data reception:** When the data has gone through the stream it is time to do something with the data. It is received by a consumer and it will usually be processed and/or stored.

There are different kind of MOMs that uses different kinds of techniques to perform the data streaming step. The following sections will explain the techniques used and also study four concrete MOM-software.

3.1 Messaging Protocols

The task of a MOM is to forward information from a producer to a consumer. This is done by encapsulating the information into messages which are sent to the consumers using different messaging protocols. Messaging protocols belongs to the application layer and they define the way for applications to communicate using messages. There exists a wide range of different protocols such as AMQP, MQTT, COAD, STOMP and HTTP. These are all designed to fit the need of special requirements and they all implement different techniques [1].

Since three of the chosen MOMs for this thesis uses AMQP it is the most essential messaging protocol and a more thorough explanation of it will follow. The last MOM, Apache Kafka, uses a protocol they themselves defined. Therefore, that protocol will be explained in section 3.2.4.

3.1.1 AMQP

Advanced Message Queuing Protocol (AMQP) is a message protocol that was created in 2003. It defines an open standard way for applications and organizations to exchange messages. This means that AMQP provides a way to let systems and applications, implemented on different platforms, communicate. This is done by defining an API for how messages should be handled by the involved clients, such as producers, brokers and consumers. It also provides a wire-level protocol that defines how the communication between the client’s work [2].
An AMQP message system is made up of three different types of clients: producers, brokers and consumers. The producers send their messages to a broker, which routes the messages to the consumers. A broker is made up of exchanges and queues (see Figure 4). The exchanges receive the messages from the producers and, depending on what type of exchange it is, forward the messages to one or several queues. The queue saves the message either in memory or on disk and sends it to its connected consumers (see Figure 4) [3].

In AMQP there exist four different kinds of exchanges that forward the messages differently [3]. The exchanges use routing keys and binding keys to determine where the messages should be sent. A routing key is a string that is attached to each message sent by the producers and a binding key is defined when a queue connects to an exchange. The four different exchanges that exists are:

- **Direct exchange**
  Which queue the message is sent to is based on the routing and binding keys. A message is forwarded to each queue where the binding key exactly matches the routing key.

- **Fanout exchange**
  Does not use routing or binding keys. The message is forwarded to all queues the exchange is connected to.

- **Headers Exchange**
  Uses headers instead of keys. Headers and keys work in the same way, but headers can consist of more than just strings.

- **Topic exchange**
  The binding and routing keys are made up off several different words separated by dots. Where the message should be forwarded to is then defined by patterns in the routing and binding keys. For example if a routing key looks like this “fruit.green.apple” and if the binding key for a queue is “fruit.#”, “*.green.*” or “#.apple” then it will receive the message. In the binding key ‘#’ substitutes for zero or more words and ‘*’ substitutes for exactly one word. It is only the binding key that uses this type of substitution with ‘#’ and ‘*’. 

![Figure 4. An example model representation of a AMQP messaging system.](image)
3.2 Message-Oriented Middleware

There exists many different pieces of MOM-type software to choose from. As the time for this thesis is limited we had to be narrow the scope of study. We searched on the web and in literature for software that fits three criteria: they have to be open source and have free licensing, they need to have support for Java development, and they need to have Windows support. We found a list of four different software that satisfied these criteria which were chosen for a deeper study: Apache Kafka, RabbitMQ, Apache ActiveMQ and Apache Qpid. Out of these four RabbitMQ, ActiveMQ and Qpid all use AMQP as their main messaging protocol while Kafka defines its own.

3.2.1 RabbitMQ

RabbitMQ is a message broker system originally created by Rabbit Technologies but currently managed by Pivotal Software. It uses AMQP (see figure 4) as its messaging protocol but can support other ones by using plugins. RabbitMQ is used by big companies such as 9GAG, Bitbucket and Reddit.

RabbitMQ’s focus is on how the messages are monitored, filtered and stored. By default, the data is not kept on disk but in memory so data is not persistent, that is, it is lost on a restart of the system. However, messages can be tagged to be persistent which means they will be stored on disk as soon as they reach a queue. Thus, in RabbitMQ, messages can be persistent on a message to message basis.

RabbitMQ also comes with a web panel that allows easier management and monitoring of the exchanges and queues (see Figure 5) [6].

![RabbitMQ web panel](image URL)

*Figure 5. A picture of the RabbitMQ web panel.*
3.2.2 ActiveMQ

ActiveMQ is a messaging system similar to RabbitMQ. It was created in 2004 by LogicBlaze but was donated to the Apache software foundation in 2007. It supports variety of messaging protocols such as AMQP (see figure 4), MQTT and STOMP [7].

A difference to RabbitMQ is that ActiveMQ has full JMS (Java Messaging Service) support. This means that if you have a producer and a consumer that is not Java based, you need additional broker middleware [8]. This is because a JMS based application can only connect a Java platform with another Java platform. More differences are that ActiveMQ does not support topic exchange where you can route messages based on key patterns and you also must serialize messages before sending them.

A study by Ionescu, V (2015) suggest that ActiveMQ is faster at receiving messages at the broker, while RabbitMQ is faster at transmitting messages from the broker [8].

3.2.3 Apache Qpid

Apache Qpid is a MOM that only implements AMQP (see figure 4). The goal of Qpid is to provide an easy way of integrating AMQP into applications. It has a large support for lots of different programming languages including Java, C++ and Python. Qpid provides both a messaging broker, that handles the messages as they are being sent between clients, and different API’s for implementing these clients [9].

The two main API’s provided are:

- **Qpid Proton**, allows applications to communicate using AMQP. Supports C/C++, Python, and Java development. Can also be used to also create servers, bridges and proxies, not only clients.
- **Qpid JMS**, an API that provides JMS by using AMQP in the foundation. Built using Qpid Proton. Support for Java applications.

3.2.4 Apache Kafka

Apache Kafka is a messaging system created by LinkedIn in 2011. Kafka uses a protocol specific only to Kafka, but the main idea is the same as other broker systems. Kafka uses a publish/subscribe protocol where several message producers can send messages to the broker, and several message consumers can receive messages from a broker. In the Kafka broker you can have several Topics which are categories of records that Producers publish to and Consumers subscribe to. Producers and consumers could for example be processes or applications. In this way several processes can send and receive different messages (see Figure 6).
Message broker type systems often come with some problems that can slow down the system. The broker usually has to store some metadata about the message, which does not scale very well when there is a lot of messages. The broker can counter this by deleting consumed messages but it is hard to know if a message has been consumed. The consumers can send an acknowledgement if they consumed it. However, these types of communications messages can be lost and they must be resent. There is a lot of overhead involved in the broker keeping track of messages which is why this does not scale very well.

Kafka fixes this by having the consumer keeping track of what messages it has consumed. In Kafka a Topic is a log segment with messages. The messages don’t have ID’s, they are identified by their offset in the log, which means the Topic doesn’t have to have any indexes that records which messages it has. The consumer needs to keep track of the logical offset which allows the system to know that the consumer has consumed all the messages up to that point. This makes Kafka very fast and allows it to be used to stream large amount of data [10].

Note that Kafka needs an external infrastructure called Apache Zookeeper to function. Apache Zookeeper provides a database that the consumer and producer can access. It is here the consumers store logical offsets that says how much data has been consumed, and the producers stores which brokers exist and their corresponding Topics [11].

3.3 Quality of service

Messaging publish/subscribe systems come with a set of desired guarantees, these are called quality of service guarantees (QoS). There exists a variety of different ways to define QoS for messaging systems [4]. IDA Infrafront requires the messages to be delivered at least once but no ordering is required. Thus, this section covers delivery and ordering guarantees, but also scalability and efficiency because they relate to the tests performed in the Method chapter.

3.3.1 Correctness

Correctness can be divided into the two subcategories delivery guarantees and ordering guarantees [5].

**Delivery Guarantees:**

These three guarantees define the way messages should be delivered:
1. **At most once.** A message is sent but there is no assurance that it is delivered. This is the fastest delivery method.

2. **At least once.** A message is sent and delivered at least once but there may be duplicates. This is slower than “at most once” because you must send acknowledgements and possibly resend.

3. **Exactly once.** A message is sent and is delivered only once. This is the slowest of the delivery methods. It is achieved with acknowledgments as well as the receiver saving a reference to the message received to make sure there are no duplicates.

**Ordering Guarantees:**

These three guarantees define the order that messages are consumed:

1. **No ordering.** Message ordering is not defined. This is the best case for performance.

2. **Partitioned ordering.** Can define a group or partition of messages that should be consumed in a desired order within the partition. This has worse performance than “No ordering”.

3. **Global ordering.** All messages are consumed in the order they are sent. Requires a lot of additional resources and has the largest toll on performance out of the three ordering guarantees.

### 3.3.2 Scalability

Defines the scalability capabilities of a messaging system. A system is scalable if it has support for a growing number of clients, such as producers, clients and also added brokers. Good scalability is when a system does not lose too much performance after adding new clients.

#### 3.3.3 Efficiency

The two features that defines the efficiency of a messaging system are latency and throughput.

- **Latency.** Latency is the time it takes for a message to traverse the system from a producer to a consumer.
- **Throughput.** Throughput is the number of messages that can be sent through the system on a given time unit.

### 3.4 Microservices

As a software application is being developed and grows in size, it is common that it outgrows the developers. If the application becomes to big it reaches a point where it can’t be easily maintained and making functional changes to it becomes difficult. In these situations, moving the application to a microservice architecture could be beneficial. A microservice is a small application designed to perform only one single task. It should be easy to understand and should be able to be independently deployed, scaled and tested. A microservice architecture application is built up of several different microservices. This makes the application easier to monitor, scale and modify as the developers only have to add new microservices or modify the existing ones [12].

#### 3.4.1 Microservices in iipax

In iipax there exist two microservices, one identification microservice (IDMS) and one conversion microservice (CSMS). They are built as web applications using Gserv, a Groovy library made for creating and deploying REST based services (client and server are
independent and the server is stateless). Other applications can access them using standard HTTP requests, such as GET and POST. Both microservices respond with the success of their operation, either the identified file format or the converted file, as well as with a JSON-object containing metadata about the performed operation (see Figure 7). This specific JSON-object, Figure 7, was chosen to give an idea of what type of information is gathered and sent from the Microservices.

```
{"service-name":"file-conversion",
"order":"PS-PDF",
"corr-id":"16271057e8f",
"filename":"original-format-ai_vectorimage-o2pWscStandard.ai",
"want-stats":true,
"suggested-file-extension": ".pdf",
"suggested-mime-type":"application/pdf",
"upload-size":5181644,
"converter-exit-value":"0",
"converter":"Ghostscript 9.21",
"conversion-time":4949,
"download-size":5509215,
"request-time":6216}
```

*Figure 7. Example JSON-response from the CSMS containing the metadata.*
4. Method

This chapter will present and explain the workflow of this thesis. The workflow was structured so that the question at issue could be answered:

- Which of the four chosen MOMs best suits, in terms of scalability and performance, gathering information from a decentralized microservice architecture?

The work started with studying related works to determine how to benchmark the MOMs. A closer qualitative study of the four chosen MOMs was done to determine which ones to benchmark. Lastly three different benchmarking tests were performed to determine the MOMs performance and scalability. A more detailed explanation of these steps follows in the subsequent sections.

4.1 Related work

In order to define how to benchmark the MOMs, related work was studied. The two studied papers provided us with benchmarking methods to replicate. The papers are presented here:

Patro et al. [13] conducts an experimental study to examine the performance of different MOMs. They start out by studying five different MOMs to determine which ones to do further testing on. To determine this, they looked at several attributes including the age of the MOMs, their licensing agreements and different features like support for persistence and different protocols.

They determine that Apache Qpid and YAMI4 best suit their needs and continue with benchmarking their performance. They define the performance by measuring throughput, latency, broker scalability and memory/CPU usage. As this thesis doesn’t examine CPU and memory usage, that part will not be included. Throughput is measured on both the entire system and also at the producer/consumer endpoints. They measure throughput by dividing the total amount of messages sent by the time the operation took.

Latency is measured by the time required for a message to be sent from the producer to the consumer. The average latency is then calculated by summarizing the gathered times and dividing them with the number of messages sent. They measure the brokers scalability by increasing the number of consumers from 1 up to 20, still having only one producer. How well the broker scales is then defined by measuring the throughput with the added consumers.

Ionescu [8] studies the performance times of RabbitMQ and ActiveMQ. He sends an image from a simple java client(producer) to a broker that then sends the data to another java client(consumer). He measures the time it takes to send the image from the producer by creating a timestamp with `System.nanoTime()` before sending. It is unclear how he gets the “finished sending” timestamp, he only specifies that it is logged after the sending. Receiving is done similarly, worth noting is that the “finished receiving” timestamp is taken once the receiving is done but before the data is decoded. This is done for five different image sizes. He also tests a scalability type metric by increasing the number of producers to the broker (up to 20) and then comparing sending times.
He mentions that using “nanoseconds were too small to accurately measure the performance and multiple runs and manual result filtering was necessary to eliminate erroneous data” [8].

4.2 Analysis of MOMs

To answer our research question and perform tests on different MOMs they need to be set up and implemented in the ipax environment which takes a lot of time and research. Because of the limited time to perform this study we delimited the scope of the study to only implement and evaluate two different MOMs. Thus, there was a need to briefly compare the four chosen MOMs according to different criteria to find the ones most suited for the task and environment, similar to how Patro et al. [13] conducted their study.

Some of these criteria, such as last updated, protocols and support for message persistence were also studied by Patro et al. [13] and were therefore selected for this study. The other ones were chosen because of their relevance to this studies’ requirements.

Since all four MOMs have Windows support, Java support, and free licensing, they are all viable for implementation. Some other important criteria are message persistence, broker message filtering and quality of service. Message persistence is important in case the server crashes or if the consumer somehow lost data and needs it again. Filtering the messages in the broker is important so that messages can be categorized into queues and then consumed by different consumers. In this way the consumer does not have to do any filtering of its own. In terms of QoS, IDA Infront does not need the messages to be ordered but requires a minimum of an “at least once” delivery guarantee.

As seen in Table 1, most of these criteria are fulfilled by all the MOMs. Although Kafka does not support filtering it has a library called “Streams API” which allows you to create your own filtering mechanism. And even though RabbitMQ does not support “exactly once”, resent messages have a flag that makes manually deleting potential duplicate messages more efficient.
<table>
<thead>
<tr>
<th>MOM/features</th>
<th>Apache Kafka</th>
<th>RabbitMQ</th>
<th>Apache ActiveMQ</th>
<th>Apache Qpid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last updated</td>
<td>March 2018</td>
<td>March 2018</td>
<td>February 2018</td>
<td>March 2018</td>
</tr>
<tr>
<td>License</td>
<td>Free</td>
<td>Free</td>
<td>Free</td>
<td>Free</td>
</tr>
<tr>
<td>Windows support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Java support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Message protocols</td>
<td>Product specific</td>
<td>AMQP, MQTT,</td>
<td>AMQP, MQTT,</td>
<td>AMQP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STOMP</td>
<td>STOMP, OpenWire</td>
<td></td>
</tr>
<tr>
<td>Message persistence</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Broker message</td>
<td>No (Manually with Kafka library)</td>
<td>Yes, with AMQP</td>
<td>Yes, with AMQP</td>
<td>Yes, with AMQP</td>
</tr>
<tr>
<td>filtering</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivery guarantee</td>
<td>Exactly once</td>
<td>At least once</td>
<td>Exactly once</td>
<td>Exactly once</td>
</tr>
<tr>
<td>Broker monitoring</td>
<td>No</td>
<td>Yes, via a web page.</td>
<td>Yes, via a web page.</td>
<td>No</td>
</tr>
<tr>
<td>tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>testing tools</td>
<td></td>
<td></td>
<td>with a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>downloadable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>plugin</td>
<td></td>
</tr>
<tr>
<td>Broker installation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>guide</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java client</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>quickstart</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java client API</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Showing the features of different MOMs.

Since all the MOMs are similar on the previous criteria, the defining criteria is the documentation provided on their websites. As configuring and implementing these MOMs can be difficult, good documentation with getting started guides and examples is preferred. As seen in Table 1 the ones with the best documentation is Kafka and RabbitMQ as these both come with quickstart guides and API references. Qpid does not have any quickstart guide and ActiveMQ lacks an API reference.

Based on this study we choose to implement Kafka since it has a different protocol than the other MOMs as well as good documentation. Out of the remaining three AMQP MOMs we chose RabbitMQ since it has the best documentation.
4.3 Benchmarking of MOMs

To answer our research question, we need to run some tests on Kafka and RabbitMQ to measure their performance. Performance is defined by measuring latency, throughput and scalability. Measuring latency and throughput is done like Petro et al. [13] conducted their study. While scalability is replicated from Ionescu’s study [8].

In all of the tests, the producers, consumers and the broker were run on a HP ELITEBOOK 840 G3 provided by IDA Infront. Here are the specifications of the laptop:

**Operating system:** Windows 10 Pro 64-bit  
**Processor:** Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz, 2808 Mhz  
**RAM:** 16GB  
**Graphics card:** Intel(R) HD Graphics 520

4.3.1 Latency

Latency is the time it takes from when you first send a message to the time when you can read the contents of the message at the Consumer. To benchmark latency we wanted to perform the test under conditions as similar to the real ipax environment as possible. Therefore, a program was written that traversed a directory of 623 files, supplied by IDA Infront, and sent them to be identified and then converted by the IDMS and CSMS. The queue and topic structure in the broker was also created specifically to emulate the ipax use case. IDA Infront desires three different streams of data to the consumers: one stream for IDMS data, one for CSMS data and one that streams all the data (see Figure 8).

![Figure 8. A model showing the broker structure for the latency test in RabbitMQ.](image)

When the services were finished and the metadata was to be sent to the MOM a timestamp was saved using `System.nanoTime()`, (see Figure 8). To consume the messages in the test, a consumer program was written. When the consumer had received and could open the message another timestamp was saved (see Figure 8). The two timestamps were then compared to calculate the latency of the sent message. Since the IDMS and CSMS both sent messages this resulted in 1246 messages being sent. The average latency for a message as well as the standard deviation of the times was then calculated for both Kafka and RabbitMQ.
4.3.2 Throughput

In the latency test, the conditions were set to be similar to the iipax environment, however this does not put much load on the MOM as a message is only sent when the services are done. Therefore, a throughput test was conducted that puts more load on the MOM. The broker was setup in the same way as in the latency test with three queues or topics.

Throughput was measured by the number of messages that could be sent over the MOM per millisecond. To do this a program was written that sent the same message back to back over the MOM 10,000 times. Three timings were made, one that measured the producer send time, one that measured the consumer receive time and one that measured the total time (see Figure 9).

The following is the sample message that was used in this test. The message contains metadata from the CSMS and consists of 379 bytes.

```
```

To measure the total send time from the producer a timestamp was saved before the first message was sent as well as after the last one. The total receive time at the consumer was measured in a similar way only that the timestamps were saved before the first message was consumed and after the last one. The total time, which is the time for all the messages to be sent and consumed, was then calculated using the send start time and the receive stop time (see Figure 9). Since these measurements were bigger than the ones in the latency tests, milliseconds could be used instead of nano seconds, therefore `System.currentTimeMillis()` was used to get the timestamps.

![Figure 9. A representation of how the timings were made in the throughput test.](#)

A test run of 20 back to back iterations where the 10,000 messages were sent was initially made. However, the times were getting faster each iteration. Thus, the amount of iterations was changed to 40 iterations to increase the reliability of the results. The average time and the standard deviation of the 40 iterations was then calculated. To calculate the throughput for each iteration, the number of messages sent was divided by the time each iteration took.
4.3.3 Scalability
Good scalability in this test means having a minimum throughput loss through the broker while increasing the number of producers publishing to the broker. To benchmark this a similar test to throughput was run, the only difference was the way the broker was set up.

Because IDA Infront only had 2 microservices a decision was made to not test scalability by replicating the iipax environment but to generally benchmark the MOMs scalability. The producers did not publish to three different queues/topics but only to one. To test the scalability the number of producers was increased from 1 to 2, 4, and 8. The test was run 40 times for each new number of producers.

The average send and receive time for each producer and consumer was calculated in the same way as the throughput test. But since this test used more producers, the average send time of all the producers combined had to be calculated by taking the average of all the producers individual send times.

To calculate the total time of getting all the messages sent from the producers and received by the consumer, the first “start send” time was compared with the consumer stop time. The average total time of all the 40 tests was then calculated. The average throughput for each new added producer, 1, 2, 4, and 8, was then compared.

4.4 The Kafka Configuration
In the tests that used the iipax microservices, latency and throughput, the Kafka Producer Library was used to make the microservices into Kafka producers. The consumer and the producers used in the scalability test were written from scratch. In this section we will cover how the Kafka producers and consumers were configured and why [11].

4.4.1 Kafka Producer
Here follows an explanation of the configurations for the Kafka Producers used for the tests and why they were used. First is the name of the setting followed by the value configured.

**enable.idempotence = false**
Configures Kafka to “at least once” message delivery guarantee, which is necessary for a fair comparison with RabbitMQ since it does not support “exactly once” delivery.

**acks = “all”**
Decides how many acknowledgments the producer wants from the broker. “All” is the slowest but will guarantee that the broker does not lose the message.

**linger.ms = 0**
How long to wait before sending data. Used for increasing the batch size of the sends. Set to 0 for minimum latency.

4.4.2 Kafka Consumer
The Kafka Consumer was written as a simple Java program that received the messages sent and then created the timestamps. It was configured with the following settings.
enable.auto.commit = true
The Consumer keeps track of its offset position and its committed offset. The committed offset is the position of the record that has been safely stored, so if the process crashes it will change its offset to the committed offset. The consumers offset will be committed automatically with the frequency of auto.commit.interval.ms.

auto.commit.interval.ms = 6000
The frequency in milliseconds that the consumer offset will be committed. A low value means that Kafka must process more commits but you have to reprocess more data in case of a failure and vice versa. Since we are only running during tests we do not expect failures, thus it was set to a high value (6 seconds).

In Kafka the topics are created and configured in a command window. All the topics were created with a replication factor of 1 which decides how many duplicates of the topic will be created for backups. Multiple partitions are used so multiple consumers can consume in parallel, thus since we only have one consumer they all were set to have only one partition.

4.5 The RabbitMQ Configuration
Similar to Kafka, the microservices used, CSMS and IDMS, were rewritten to become RabbitMQ producers for the latency and throughput test. The consumer and producers in the scalability test were written from scratch [6].

4.5.1 RabbitMQ Producer
Here follows and explanation of how the RabbitMQ producers were configured and an explanation of why.

AMQP.BasicProperties = MessageProperties.PERSISTENT_TEXT_PLAIN
Configures RabbitMQ to send persistent messages. This setting was chosen because Kafka keeps messages stored by default so having RabbitMQ store them as well makes the tests fairer. However more settings in the Consumer is necessary for persistent messages.

exchange_type = “topic”
The exchange was configured to be a topic exchange. This is because it minimizes the amount of sends by the producer in the latency and throughput test since the exchange will duplicate the messages instead of the producer.

4.5.2 RabbitMQ Consumer
The RabbitMQ consumer was configured in the following way.

durable = true
This sets the queues to be persistent, that is, they are saved on disk. This setting is necessary to have persistent messages.

autoack = true
Configures RabbitMQ to automatically send acknowledgements which makes RabbitMQ have the delivery guarantee “at least once” which is what Kafka was configured to as well.
5. Results

The results and data collected after performing the tests in chapter 4 are presented in the following sections. A discussion about the results follows in chapter 6.

5.1 Latency

The average latency of the 1246 messages sent was measured in nanoseconds but is presented in milliseconds since all the other tests are presented in milliseconds.

- Apache Kafka: 32,7212171 \( \approx \) 33 ms
- RabbitMQ: 4,4926620 \( \approx \) 4 ms

The standard deviation on the latency times for both MOMs are:

- Apache Kafka: 31,809645 \( \approx \) 32
- RabbitMQ: 19,769023 \( \approx \) 20

5.2 Throughput

Throughput was measured by having a producer send 10,000 messages over the MOM as fast as possible. In order to get a reliable result, the same test was performed 40 times. The different times collected are presented below.

5.2.1 Throughput Send Time

![Figure 10. A graph showing the send times of the 40 throughput tests.](image)

First, we have the send time in Figure 10. Send times in Test 1 took very long for Kafka as well as test 36 for RabbitMQ. Notice that Kafka becomes faster after 16 tests while RabbitMQ’s times oscillate between around one second and two seconds. The send time average of the 40 tests were calculated to:

- Apache Kafka: 752,950 ms
- RabbitMQ: 1214,925 ms
5.2.2 Throughput Receive Time

Figure 11. A graph showing the receive times of the 40 throughput tests.

Figure 11 presents the receive times. Here the times for test 1 and test 36 resemble the send times a lot. The following is the average of the 40 receive times:

- Apache Kafka: 750,350 ms
- RabbitMQ: 1928,750 ms

5.2.2 Throughput Total Time

Figure 12. A graph showing the total times of the 40 throughput tests.
Figure 12 presents the total time, that is, to send and receive all the 10,000 messages. This graph naturally resembles both the send times (Figure 10) and the receive times (Figure 11). The following is the average of the 40 total times:

- Apache Kafka: 755,150 ms
- RabbitMQ: 1961,250 ms

The following is the standard deviation of the 40 total times:

- Apache Kafka: 797,257
- RabbitMQ: 867,671

The throughput average was calculated by dividing 10,000 over the times for each test:

- Apache Kafka: $\frac{10,000}{20,8321587} \approx 21$ messages/ms
- RabbitMQ: $\frac{10,000}{5,796633064} \approx 6$ messages/ms

### 5.3 Scalability

To test scalability, the same test that was used to test throughput was used only that the number of producers is increasing from 1 up to 8.

#### 5.3.1 Scalability Send Time

![Graph showing send times for 1, 2, 4, and 8 producers.](image)

*Figure 13. A graph showing the average send times for 1, 2, 4 and 8 producers.*

Figure 13 shows the different send times with an increasing number of producers. Kafka clearly has lower times. However, both Kafka’s and RabbitMQ’s times change very slightly except for 8 producers, where RabbitMQ’s send time increases by a very large amount.
5.3.2 Scalability Receive Time

Figure 14. A graph showing the average receive times for 1, 2, 4 and 8 producers.

Figure 14 shows the receive times for different number of producers. It shows that with a growing number of producers the receive times is almost linearly increasing for both RabbitMQ and Kafka. Kafka has lower times for all different sets of producers.

5.3.3 Scalability Total Time

Figure 15. A graph showing the average total times for 1, 2, 4 and 8 producers.
Figure 15 shows the total time for the different sets of producers. They both increase almost linearly here except for the test with 8 producers, where Kafka’s time does not increase as much as RabbitMQ’s. Kafka has lower times here as well.

<table>
<thead>
<tr>
<th>MOM / Nr. of producers</th>
<th>1 to 2</th>
<th>2 to 4</th>
<th>4 to 8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kafka increase</strong></td>
<td>244%</td>
<td>403%</td>
<td>61%</td>
</tr>
<tr>
<td><strong>RabbitMQ increase</strong></td>
<td>172%</td>
<td>123%</td>
<td>54%</td>
</tr>
</tbody>
</table>

*Table 2. The percentage increase in time for different amounts of producers.*

However, as seen in Table 2, RabbitMQ has lower percentage increases in time as the number of producers increase.
6. Discussion

In this chapter the results previously presented will be discussed. The validity of the results and the methodology will be questioned, possible sources of error will be presented and the sources used will be critiqued, which will lead to the conclusions of this study.

6.1 Latency

In the latency test, Apache Kafka’s average time was about seven times slower than RabbitMQ’s even though in the throughput and scalability tests, Kafka overall had faster times. Kafka also had an approximately 50% larger standard deviation than RabbitMQ, which means that Kafka’s times fluctuated a lot more and were less consistent. Consistent times are preferred since in certain systems you might be dependent on the messages being consistently sent and received within a specific time period.

A theory to explain Kafka’s slow times is that Kafka is more optimized for sending many messages in batches very fast rather than one message intermittently. In the latency test 1246 infrequent messages were sent over a large time period. This meant that no batching could be done, which explains why Kafka’s result were so poor. Nonetheless, this test shows that RabbitMQ is a lot faster than Kafka at sending single infrequent messages.

The times measured in this test were so small that nanoseconds had to be used instead of milliseconds. A study that looked at the timing accuracy of computers show that a general-purpose computer that runs windows has an accuracy of -30 to +50ms. They suggest that an experiment that needs greater accuracy than this should use designated hardware specialized for more precise timing [14]. As this is not feasible for this thesis the results from this test paints a general picture of the MOMs latency rather than exact and precise figures.

6.2 Throughput

The standard deviation for both MOMs in this test are fairly similar. RabbitMQ only has an approximately 9% higher value than Kafka. This means that they are both almost equally consistent in their throughput times. But, as Figure 10, 11 and 12 all depict, Kafka generally has a higher throughput than RabbitMQ. However, it seems that Kafka requires a period of time to achieve full speed. Up to around the 15th test run it is just slightly faster than Kafka, but then drastically drops and keeps a steady time that is much faster than RabbitMQ.

A theory on why Kafka is slow at the start is because of the configuration linger.ms = 0. Setting Linger.ms to 0 means that Kafka will not wait to accumulate any batch of messages, but will try to send as fast as the messages appear. However, batching can still occur if the producer queues many messages to be sent before the send thread can actually dispatch them. Therefore, on the first few sends it is likely that there is zero batching but when Kafka has run for a while it will have more messages ready to be sent and more batching will be done.

RabbitMQ keeps a steadier pace through all the tests, fluctuating between 1000 and 3000 ms. The only upset in RabbitMQ’s tests is in test #36. As shown in Figure 10,11 and 12 RabbitMQ’s times spikes a lot during this test. As this result stands out from the rest it is likely that this is caused by something related to the computer and not caused by the MOM.
Since this is only seen in one of the tests it doesn’t affect the average result that much. The total average with the time from test #36 removed is 1807.175 ms, which is only an 8% decrease in time and is still dramatically slower than Kafka. Thus, this test shows that Kafka is overall faster than RabbitMQ when sending many messages.

6.3 Scalability

In Figure 13 it can be seen that the send times of Kafka and RabbitMQ is not affected by the number of producers, except for 8 producers in RabbitMQ where the time increases by almost 6 times. This could be because the laptop the tests were running on froze during that test which would slow the process down. However, the drastically increased send time does not seem to affect the receive time and total time. The increase from 4 to 8 producers, showed in Figure 14 and 15, is not at all equal to the increase in Figure 13. Instead the receive and total time increase linearly with the added producers. The reason for this is likely that the amount of data sent through the broker is still enough to produce a queue which keeps the consumer busy.

Something that could have affected the results is that the producer, consumer and broker were all running on the same laptop. MOMs are usually setup to be run on separate hardware and used as a link between the producer and consumer machines. It is possible that the high number of producers running slowed the computer down and therefore also the times. Thus, the increase in times in Figure 15 could be more a reflection of how bad the computer handles a large number of clients rather than the broker having bad scalability. However, since both MOMs were tested on the same computer and with similar producer programs, the conditions should be the same for both of them and the results is still comparable.

The results portrayed in Figure 15 shows that Kafka’s total times are all lower than RabbitMQ’s. This could lead one to believe that Kafka scales better than RabbitMQ, however this is not true. Since Kafka’s throughput is better than RabbitMQ’s, shown in Figure 12, it is natural that Kafka’s times are still better with added producers. Table 2, which depicts the percentage increase, shows that RabbitMQ actually scales better than Kafka does. RabbitMQ has a much better percentile increase when the producers are increased from 1 to 2 and from 2 to 4. When increasing from 4 to 8 it is still slightly better but only with a few percentages.

The scalability tests show that RabbitMQ scales a lot better with a small number of producers, up to 4 seems optimal. After that point both MOMs seem to increase with a similar amount.

6.4 Methodology

In this section the replicability and validity of the method used will be discussed and critiqued.

6.4.1 Replicability

In the throughput and scalability test the replicability is assumed to be high. This is because the program used for the test was a fairly simple program written only to send the same message over the MOM 10,000 times. However, the latency test is deemed to have low replicability since the real conversion and identification services in ipax was used for this test. This means that to replicate this, the source code for the microservices as well as the same test files is needed.
6.4.2 Validity

With the purpose of using the MOM in iipax, the validity of the latency test results is assumed to be high since it was conducted in iipax’s real use case. However, for general purpose of the MOM the validity is not very high since the normal use case for MOMs is more often to send messages with a higher frequency. In the throughput and scalability test, the validity is higher since it sends more messages with a higher frequency, which better mimics the real use case for MOMs. But, Petro et al. [13] sends 1,000,000 messages in their tests, whereas in this test only 10,000 messages were sent. The lower number of messages sent could decrease the validity as this means that the timings used were smaller, leaving more room for timing errors. Another tweak that could increase the validity in the throughput test is to implement a sleep period after each iteration of 10,000 messages sent. This is to make each iteration more independent and to make sure that no batching of messages from previous iterations occur. This is discussed in 6.2 as a theory as to why Kafka’s times decrease over time.

Changing the hardware setup could also increase the validity of all the tests. The producers, consumers and broker are usually run on separate hardware that communicate with the MOM over a network. Also, the MOM should not be setup on a general-purpose computer, but instead run on a dedicated server. To run the most optimal tests and to get the most accurate result the tests should have been conducted in a setup more similar to this. However, this was not possible in the thesis due to a limited availability of such hardware.

6.5 Sources

The sources in this paper were carefully selected to fit the subject and to be reliable. Thus, most of the sources used in this thesis are published scientific papers. The papers are all recently published with the oldest one being published in 2008.

To get technical specifications and information about the different MOMs studied, their own websites was mainly used. These websites can possibly be biased and present favoring information about their own products. However, these are also the best sources to get the desired information about them, therefore they were used.
7. Conclusion

Throughout this report we looked at the research question:

- Which of the four chosen MOMs best suits, in terms of scalability and performance, gathering information from a decentralized microservice architecture?

To answer the question, we started by doing a quantitative study of four different MOMs. Out of those four, we chose two for further studies and benchmarking. Three tests were conducted on both RabbitMQ and Apache Kafka that yielded some results that were subsequently discussed.

To determine the performance of the MOMs we tested their latency and throughput. The throughput results suggest that Kafka has higher performance with an average total time of 21 messages/ms and RabbitMQ with 6 messages/ms. Kafka had a lower average time for the send time and receive time as well. However, the latency test shows that RabbitMQ sends single infrequent messages about seven times faster than Kafka, possibly suggesting that Kafka sends batches of messages faster. Since the purpose of these MOMs is most frequently to send many messages rapidly and no single ones, and since nanoseconds had to be used, the latency test does not provide conclusive evidence to determine the highest performing MOM. Therefore, we conclude that Kafka has higher performance than RabbitMQ based on throughput.

The scalability result showed that Kafka had lower times for all different numbers of producers. However, RabbitMQ had a much lower percentage increase for 1 to 2 producers and 2 to 4 producers and a slightly lower percentage increase for 4 to 8 producers. This suggests that RabbitMQ scales better than Kafka when adding more producers. However, when increasing from 4 to 8 producers, the percentage increase is only marginally different and thus this result is not conclusive evidence to determine which MOM scales better with a higher count than 8 producers.

7.1 Future work

Our latency and throughput tests yielded wildly different results for Kafka and RabbitMQ. So, in order to achieve more conclusive and accurate results more tests need to be done. For example, tests with different batch sizes could be conducted to find out at which size Kafka and RabbitMQ breakeven.

In order to conclude which of the two MOMs actually scales best with a large number of producers more tests need to be conducted. Tests with 20, or more, producers preferably running of separate hardware from the broker would provide more conclusive and precise evidence. However, with the limited time and resources of this thesis such tests could not be performed.

Another interesting test would be to benchmark throughput and scalability on MOMs with persistence and non-persistence configured. Since Kafka’s protocol specifies persistence automatically it is likely that it is more optimized at persistence than RabbitMQ. Thus, testing RabbitMQ without persistence could yield different results than what we have seen in this paper.
8. References


