Thesis Title

Thesis Subtitle

Amir Rabiee

Master of Science

Electrical Engineering and Computer Science
Royal Institute of Technology
1-1-2018

Abstract

Abstract goes here

Abstrakt

Abstrakt

Acknowledgements

I want to thank...

Contents

Chapter 1

Introduction

The world of applications and the data being generated and transfered to and from them are constantly evolving in a fast paced manner. The amount of data generated over the Internet and the applications running on it are estimated for 2020 to hit over 40 trillion gigabytes [1].

The applications handling this data has requirements that needs to be met such as having high reliability and high availability. Because of the high demands for such requirements a natural outcome of this is to build a distributed system that can support these applications whether they be a load balancing system or a game application or anything in between. [2, p. 1].

The evolving of distributed systems stems from the relative cheap hardware commodity that one has been able to utilize to build networks of computers communicating together for a specific purpose. These systems are in comparison to the *client-server* architecture constructed of different applications running on multiple separated machines. Because of the inherit nature of having multiple machines coordinating together for a common task, an innately advantage for them is that distributed systems are more scalable, reliable and faster when architectured correctly in comparison to a *client-server* model. The advantages with these systems comes with a cost, as designing, building and debugging distributed systems are more complicated than systems running on only one machine [2, p. 2].

In order to assess the scalable part of a distributed system, appropriate measures have to be taken, to easily meet the need when the communication between different machines and their applications are put to the test. To help facilitate the problems that can occur when an application on one machine tries to communicate with a different application on another machine one can use **message queues** and **message brokers** [3, p. 2].

The message brokers works in symbiosis with the message queues to help deliver messages sent from different destinations and route them according to the correct route [4, p. 326].

1.1 Background

The distributed systems that are developed to meet the requirements of having high availability and reliability are built upon some abstract message form being sent from one process located on one machine to another process on a different machine. Therefore an inherit attribute of a distributed system is that it needs an architecture or system that can distribute messages in order to achieve higher scalability and reliability.

1.1.1 Message oriented middleware

The architecture used, that strives to fulfil the requirements, is called *message-oriented middleware* (MOM), this middleware is built on the basis of an asynchronous interaction model which enables users to not be blocked after sending a message instead they continue processing with their execution [5, p. 4]. More importantly a message oriented middleware is used as a basis for distributed communication between processes without having to adapt the source system to the destination system. This architecture is illustrated in Figure 1.1 where the apps in this context refers to the various systems using the MOM.

A central concept and structure when using a message oriented middleware is the usage of **message queues**, these queues are used to store messages on the MOM platform. The systems using the MOM platform will in turn use the queues to send and receive messages through them. These queues have many configurable attributes, which include the name, size, the sorting algorithm for the queue and so forth [5, p. 7].

In order to efficiently distribute the messages to different queues one has to resort to a **message broker**, these message brokers can be seen as an architectural pattern according to [6, p. 288], the broker is responsible for translating message data formats between applications which abstracts the routing from one destination to another.

An important attribute of message brokers is their usage of message queue protocols, there are many different types of protocols that can

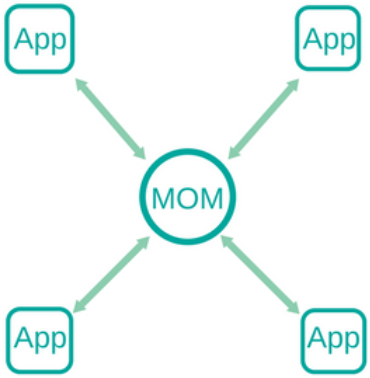

Figure 1.1: Structure of message oriented middleware.

be used such as **Streaming Text Oriented Messaging Protocol** (STOMP), **Extensible Messaging and Presence Protocol** (XMPP), **Message Queueing Telemetry Transport** (MQTT) and **OpenWire** and more [7, p.3].

The unique capabilities of the MOM-model comes from the messaging models that it uses, there are mainly two messaging models to be found, **point-to-point** and **publish/subscribe**.
The point-to-point model provides a communication link between the producer and consumer with the usage of a message queue, the consuming clients processes messages from the queue with the requirement of having only one receiver consuming the message albeit not being a strict requirement [5, p. 9], these messages are delivered **exactly once**.
In the publish/subscribe model, the producer produces a message to a specific topic, the consumers interested in these messages will subscribe to the topic, these messages are routed by a publish/subscribe engine.

An intuitive metaphor that can be used to understand the usage of the MOM-model is to see it as a post terminal where the message brokers are the persons delivering the posts to the right post code area which in this case are the message queues.

1.2 Problem

With a plethora of different message brokers available to help with implementing a message oriented middleware, choosing one is a multifaceted question that has many different aspects that need to be taken in consideration. Every message broker has their own design and implementation goals and can therefore be used for different purposes and situations. An overview of some message brokers and the protocols supported can be seen in Table 1.1.

Table 1.1: Message brokers and their supported protocols

Message broker	**Protocols supported**
Apache Kafka	Uses own protocol over TCP
RabbitMQ	AMQP, STOMP, MQTT,
ActiveMQ	AMQP,XMPP, MQTT, OpenWire

1.2.1 84codes, CloudKarafka, CloudAMQP

The message brokers found in Table 1.1 are widely used and can be deployed on different server architectures and platforms [8], the company **84codes** offers two different services called **CloudKarafka** and **CloudAMQP** [9][10]. CloudAMQP is a RabbitMQ-focused implementation and CloudKarafka is a Apache Kafka solution, these two services are hosted on different cloud platforms such as Amazon Web Services, Google Cloud, Microsoft Azure etc.

These two platforms as a service needs to be tested on their enqueueing performance in order to get a deeper understanding of the underlying mechanisms behind the enqueueing decisions when sending a message from a producer to a consumer. With this in mind the main problem adressed by this thesis work is to perform testing of both Apache Kafka and RabbitMQ as message brokers on Amazon Web Services to be able to answer the question and quantify which situations to use RabbitMQ and Kafka on a cloud platform such as AWS?

1.3 Purpose

Because of the intricacy of the different message brokers and their corresponding protocols they use it is a relative difficulty in grasping both

the fine grained differences between them as well as the coarse grained. This thesis discusses and shows an overview of the available messaging middleware solutions and aims to validate and verify the enqueueing performance for two message brokers. The enqueueing performance focuses on the throughput aspect versus the latency, moreover the thesis focuses on the resource usage of both the message brokers such as CPU and memory during load.

Furthermore the two different message brokers RabbitMQ and Apache Kafka is a debatable topic on deciding which one to use [11], this thesis work will try to shed a light on this subject, as well as focus on testing the two platform-as-a-service (PaaS) CloudKarafka and CloudAMQP on Amazon Web Services.

The thesis work will present the designed testing experiments and the results of them, in order to visualize the performance of the two message brokers running on the cloud platform Amazon Web Service.

1.4 Goal

The goals of this project is presented below:

- Design experiments for testing the enqueueing performance for Apache Kafka and RabbitMQ.
- Compare the results from each experiment and discuss the findings with regards to the the cloud platform and the respective message broker.
- Evaluate with a 95th percentile for sending and processing messages.
- Evaluate the results with a statistical linear model.
- Use the project as reference material when analyzing Apache Kafka and RabbitMQ for their enqueuing performance.

These goals presented lays the foundation for this thesis work and the results derived from the goals can be further used as a reference point for when to use RabbitMQ over Kafka and vice versa.

1.4.1 Benefits, Ethics and Sustainability

With the amount of data being generated in the present day which can be found in many enterprises one has to think of the ethical issues and aspects that can arise with processing and storing this much information and what the possibilities are with extracting valuable data from this content.

This thesis work is not focused on the data itself that is being stored, sent and processed, but rather the infrastructures which utilizes the communication passages of messages being sent from one producer to a consumer. Nonetheless the above mentioned aspects are important to discuss, because from these massive data-sets being generated one can mine and extract patterns and human behaviour [12], which in turn can be used to target more aggressive advertisements to different consumer groups.

Moreover another important aspect to be brought up is the privacy and security of peoples personal information being stored which can be exposed and used for malicious intent [13], this will be remedied to a degree with the introduction to the new changes in the General Data Protection Registration that comes into effect May 2018 [14] .

With the usage of large cloud platforms and their appropriate message brokers that is used to send millions of messages between one destination to another, one has to think of the incumbent storage solutions for the data, which in this case has resulted in the building of large datacenters. These datacenters consumes massive amount of electricity, up to an amount of several megawatts [15], in comparison to a toaster that uses about 1 kilowatt.

The company 84codes and their services holds a greater standpoint on the ethical aspect because both CloudKarafka and CloudAMQP is used by companies all over the world for sending data and therefore the ultimate responsibility of security lays on 84codes and their strategies and design decisions to keep the data intact and not exposed to third-parties or other non-authorized parties.

The main benefitters of this thesis work are those standing at a crossroads of choosing a message broker for their platform or parties interested in a quantitative analysis focused on the enqueuing performance of two message brokers, Apache Kafka and RabbitMQ on a cloud platform such

as Amazon Web Services.

1.5 Methodology

The focus of this thesis work is to analyze and test the enqueuing performance of two different message brokers, and with that in mind, one has to think of the different methodologies and research methods that are to avail and that are the most suitable to conduct such a thesis work. The fundamental choosings of a research method is based on either a *quantitative* or *qualitative* method, both of these have different goals and can be roughly divided into either a numerical or non-numerical project [16, p. 3].

The quantitative research method focuses on having a problem or hypothesis that can be measured with statistics and validified as well as verified, a qualitative research on the other hand focuses on opinions and behaviours to reach a conclusion and to form a hypothesis.

For this project the most suitable research method is of quantitative form because of the experiments and tests to be run gathers numerical data.

Another important aspect to be chosen for the thesis work is the *philosophical assumption* that can be made and there are several school of thoughts to be considered. Firstly the *positivism* element relies on the independency between the observer and the system to be observed as well from the tools to be measured with. Another philosophical school is the *realistic*, which collects data from observed phenomenons and thereafter develop knowledge. Thirdly a *criticalism* element is the one which focuses on learning how users can affect different types of computer systems. [16, p. 4]

There are several others philosophical principles but for this project the most fitting ones was to use a combination of both the positivism as well as the realistic.

Moreover to continue with the experimental testing of the enqueuing performance of the different message brokers a research method that fits the requirements of the thesis work had to be chosen. There are mainly two divisions of research methods, a *experimental* or a *non-experimental*

research method.

Because of the nature of the thesis residing in experimenting with the correlation of variable changes and their relationship between one another in order to see how the message brokers becomes affected an obvious choosing would be a experimental research methodology. An *analytical* research method based on previous testing of both RabbitMQ and Apache Kafka is also showcased for more a comprehensive conclusion. [16, p. 4]

1.6 Delimitations

Explain the delimitations. These are all the things that could affect the study if they were examined and included in the degree project. Use references!

1.7 Outline

In text, describe what is presented in Chapters 2 and forward. Exclude the first chapter and references as well as appendix.

- **Chapter 2** shows a more in-depth technical background of Apache Kafka and RabbitMQ and their protocols.
- **Chapter 3** presents the research methodologies of the thesis work with the appropriate statistical tools.
- **Chapter 4** will present the different experiments and test cases with the goal in mind.
- **Chapter 5** will present and visualize the results from the experiments.
- **Chapter 6** will discuss the results and the conclusions that can be drawn from the thesis work.

Chapter 2

Background

In order for two different applications to communicate with each other a mutual interface must be agreed upon. With this interface a communication protocol and a message template must be chosen such that an exchange can happen between them. There are many different ways of this being conducted, one can define different types of schemas in a programming language or rather let two developers agree upon a request of some sort containing identification of said developer. Provided this mutual agreement between two applications then it is of no importance for both of these said applications to know the intricacies of one anothers system. Furthermore the programming language and framework can over the years change for the application but as long as the mutual interface stays firm between these two applications they will always be able to communicate with each other which in turn results in lower coupling between systems.

To further attain a lower coupling one can introduce *messaging systems* or *message-oriented middleware* which are interchangeable. The messaging system enables the communication between sender and receiver to be less conformative in a way where it is not necessary for the sender to preserve information on how many instances of receivers there are, where they are located or whether they are active [17, p. 3]. The message system is responsible for the coordination of message passing between the sender and receiver and has therefore a primary purpose of safeguarding the communication between two parties the same way a database has the task of storing each data record safely and persistent [4, p. 14].

Because of the inherit unreliability of networks and computers the reason message systems exists is to help mitigate the problem of a message being lost after being sent where a receiver or has gone down. The message system in this case will try to resubmit the message until it is successful.

There are different types of models that a message system can use but

there are mainly three different types of communication models, a point-to-point, publish/subscribe or a hybrid of those two.

2.1 Point-to-Point

> *"Alexandra walks into the post office to send a parcel to Adam. She walks up to the counter and hands the teller the parcel. The teller places the parcel behind the counter and gives Alexandra a receipt. Adam does not need to be at home at the moment that the parcel is sent. Alexandra trusts that the parcel will be delivered to Adam at some point in the future, and is free to carry on with the rest of her day. At some point later, Adam receives the parcel."* [17, p. 3]

The above mentioned quote demonstrates the mechanism of *point-to-point* communication where it is implemented with queues that uses a first in first out schema. This leads to only one of the subscribed consumers receiving the message, this can be seen in Figure 2.1.

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

When it comes to queues one has to discuss the attributes of them, more specifically *persistence* vs *durability*. The persistence attribute focuses on if a failure happens during message processing that the message is not vanished next time it is processed. This is done by storing the message to some other form than in-memory for example a temporary folder, a database or a file etc [18].

The durability aspect is when a message is sent to a queue and the queue is offline, and there after the queue comes back online, it is of importance that the queue fetches the messages that was lost during the downtime.

With persistence the reliability increases but at the expense of performance and it all comes down to design choices for the message systems to choose how many of the messages should be persisted and in what ways.

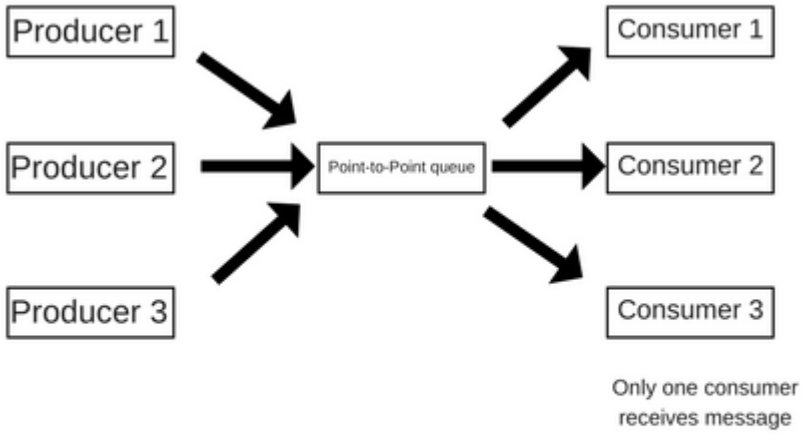

Figure 2.1: A point-to-point communication domain.

2.2 Publish/Subscribe

> *"Gabriella dials in to a conference call. While she is connected, she hears everything that the speaker is saying, along with the rest of the call participants. When she disconnects, she misses out on what is said. On reconnecting, she continues to hear what is being said."* [17, p. 4]

The quote above demonstrates the publish/subscribe interaction scheme. In this case Gabriella can connect herself up to a conference call and receive information from the speaker and the person speaking is not concerned with how many listeners there are. The system, in this case, the conference call, assures that anyone connected to the call will receive the information given. Gabriella in this scenario is the *subscriber* and the speaker the *publisher*.

This type of messaging architecture can be implemented with the help of

topics, a topic can be seen as an event from which subscribers are interested in and whenever a message is sent to a topic from a publisher all the subscribers interested in such a topic becomes notified of the update. These events being sent to a topic are sent in a asynchronous manner and one can therefore see the strengths that lie within this type of architecture because it can seperate the dimensionalities of *temporal*, *spatial* and *synchronization* from each other [19, p. 1]. This architecture is illustrated in Figure 2.2.

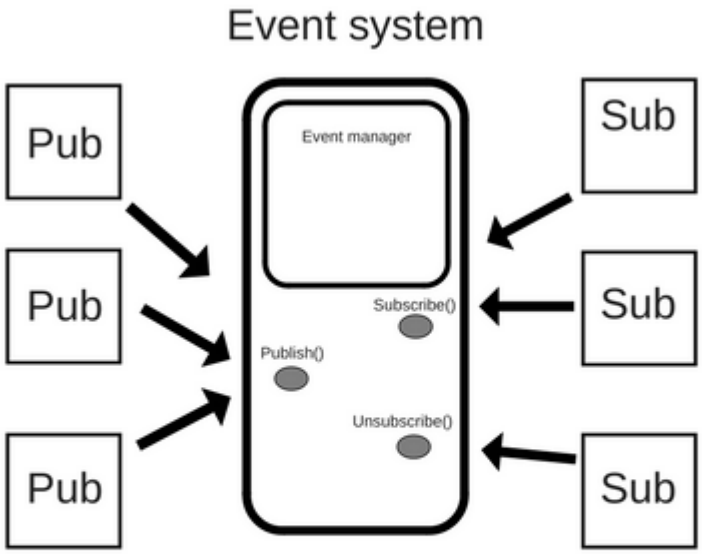

Figure 2.2: A publisher/subscriber domain.

The publishers can publish to their topics and the subscribers can either subscribe or unsubscribe from the topic. The three dimensionalities of the publish/subscriber domain helps to understand how a system with such a construct can enhance the messages passing from one producer to another consumer.

With the *spatial* dimensionality, the requirement of two interacting par-

ties needing to know each other is not present. Because the publishers needs only to publish their data/message to a topic with an event service as a middle hand and the subscribers need only to interact with event service [19, p. 2]. The publishers will also not keep any data connected to the subscribers and does not know how many of the subscribers there are, in the same way that the subscribers does not know how many of the publishers exist.

The *temporal* dimensionality is that there are no requirements of both the publisher and subscriber to be active at the same time, that is, the publisher can send events to a topic while a subscriber is offline which in turn infers that a subscriber can get an event that was sent after the publisher went offline.

The *synchronization* dimensionality means that a publisher does not need to wait for a subscriber to process the event in the topic in order to continue with the execution. This works in unity with how a subscriber can get notified asynchronously after doing some arbitrary concurrent work.

A secondary type of implementation is the *content-based* publish/subscribe model which can be seen as an extension of a topic based approach. What differs a content-based from a topic-based is that one is not bound to an already defined schema such as a topic name but rather to the attributes of the events themselves [20, p. 4]. To be able to filter out certain events from a topic one can use a subscription language based on constraints of logical operators such as or, and, not etc [19, p. 9].

2.3 Communication protocols

There are many types of communication protocols that can be used when you have to send a message from one destination to another. These protocols has their own design goals and use cases that are more fitting then others and deciding on one protocol to use is a challenge you have several use cases to think of such as, how scalable is the implementation, how many users will be sending messages, how reliable is sending one message and what happens if the messages do not get delivered etc. These are some of the aspects for the developer to dwell on and as such leaving them with protocols such as AMQP, XMPP, STOMP and MQTT.

2.3.1 Advanced Message Queueing Protocol

Advanced Message Queuing Protocol (AMQP) was initiated at the bank of JPMorgan-Chase with the goal of developing a protocol that provided high durability during intense volume messaging with a high degree of interoperability. This was of high importance in the environment of banking because there is an economic impact if a message is delayed, lost or processed incorrectly [21].

AMQP provides a rich set of features for messaging with a topic-based publish/subscribe domain messaging, flexible routing, security etc and is used by large companies that process over billion of messages a day ranging from JPMorgan Chase, NASA and Google [22].

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

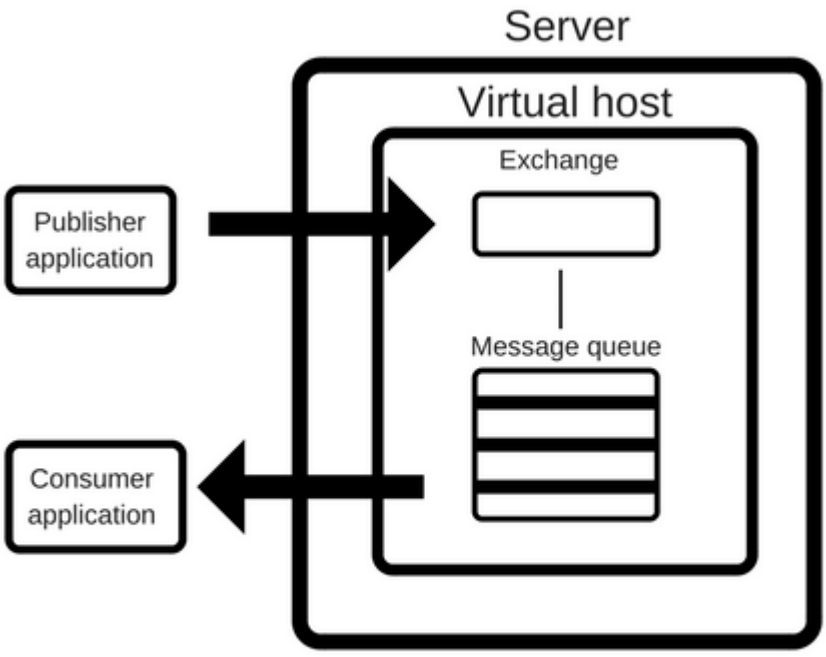

Figure 2.3: AMQP Protocol model

The exchange in Figure 2.3 accepts messages from producers and routes them to message queues, and the message queues stores the messages and forwards them to the consumer application [23].

The message queues in AMQP is defined in a **weak FIFO** way because if there exists multiple readers of a queue the one with the highest priority will take the message before the others. A message queue has the following attributes which can be configured,

- Name - Name of the queue.
- Durable - If the message queue can lose a message or not.
- Exclusive - that is if the message queue will be deleted after connection is closed.
- Auto-delete - the message queue can delete itself after the last consumer has unsubscribed.

In order to determine which queue to route a specific message from an exchange, a binding is used, this binding is determined with help of a routing key [24, p. 50].

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

Direct type

This exchange type will bind a queue to the exchange using the routing key K, if a publisher sends a message to the Exchange with the routing key R, where K = R then the message is passed to the queue.

Fan-out

This exchange type will not bind any queue to an argument and therefore all the messages sent from a publisher will be sent to every queue.

Topic

This exchange will bind a queue to an exchange using a routing pattern P, a publisher will send a message with a routing key R, if R = P then the message is passed to the queue. The match will be determined for routing keys R that contain zero or more words, where each word is delimited

with a dot. The routing pattern P works in the same way as a regular expression pattern. This can be seen in Figure 2.5

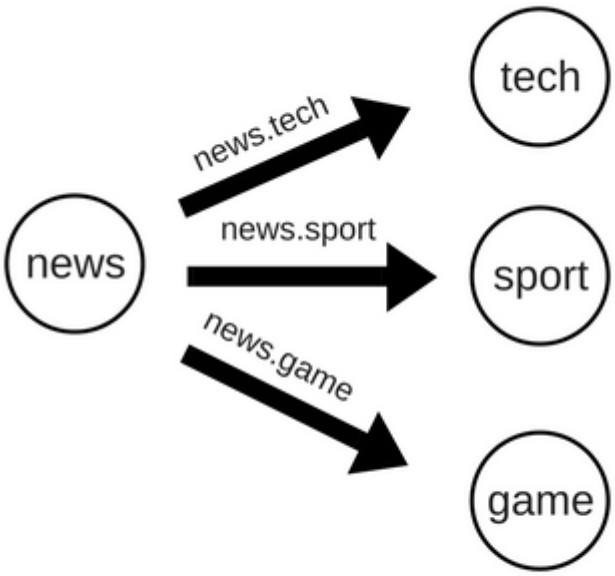

Figure 2.4: Topic exchange type

Headers

The headers exchange type will prioritize how the header of the message looks like and in turn ignores the routing key.

2.3.2 Extensible Messaging and Presence Protocol

The Extensible Messaging and Presence Protocol (XMPP) technologies were invented because of the abundance of different client applications for instant messaging services. The XMPP is an open technology in the same way as the Hypertext Transfer Protocol (HTTP), the specifications of the XMPP puts focus on the protocols and data entities that are used for real-time asynchronous communication such as instant messaging and streaming. [25, p. 7]

XMPP makes use of the Extensible Markup Language (XML) to enable an exchangement of data from one point to another. The technologies of XMPP is based on a decentralized server-client architecture much alike how the world wide web is deployed, this means, if a message is sent from one destination, the initial message is sent to an XMPP server and thereafter sent to the receivers XMPP server and finally to the receiver.

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

The *<message/>* is what is used for pushing data from one destination to another, and are used for instant messaging, alerts and notifications [25, p. 18]. The *<presence/>* is one of the key concepts of real-time communications because this stanza enables others to know if a certain domain is online and ready to be communicated with. The only way for a person to see that someone is online is with the help of a presence subscription which employs a publisch-subscribe method. The info/query stanza *<iq/>* is used for implementing a structure for two clients to send and receive requests in the same way GET, POST and PUT methods are used within the HTTP.

What differs the iq stanza from the message stanza is that the iq has only one payload that the receiver must reply with and is often used to process a request. For error handling the XMPP does not acknowledge all of the packets sent over a communication link and XMPP assumes that a message or stanza is always delivered unless an error is received [25, p. 24].

The difference between a stanza error and a regular message error is that a stanza error can be recovered while other messages results in the closing of the XML stream that was opened in the start.

2.3.3 Simple/Streaming Text Oriented Messaging Protocol

The Simple/Streaming Text Oriented Messaging Protocol (STOMP) is a simple message exchange protocol aimed for asynchronous messaging between entities with servers acting as a middlehand. This protocol is not a fully pledged protocol in the same way as other protocols such as AMQP or XMPP, instead STOMP adheres to a subset of the most common used message operations [26].

STOMP is loosely modeled on HTTP and is built on frames, these frames is made of three different components, primarily a command, a set of optional headers and body. A server that makes use of STOMP can be configured in many different ways because STOMP leaves the handling of message syntax to the servers and not in the protocol itself. This means that one can have different delivery rules for servers as well as for destination specific messages.

STOMP employs a publisher/subscriber model where the client can both be a producer by sending frame containing SEND as well as being a consumer, this is done by sending a SUBSCRIBE frame.

For error handling and to stop malicious actions such as exploiting memory weaknesses on the server STOMP allows the servers to put a threshhold on how many headers there are in a frame, the lengths of a header and size of the body in the frame. If any of these are exceeded the server has to send an ERROR frame back to the client.

2.3.4 Message Queue Telemetry Transport

The Message Queue Telemtry Transport (MQTT) is a lightweight messaging protocol with design goals aimed to an easy implementation standard, having a high quality of service data delivery and being lightweight and bandwith efficient [27, p. 6]. The protocol uses a publish/subscribe model and is aimed primarily for machine to machine communication, more specifically embedded devices with sensor data.

The messages that are sent with a MQTT protocol are lightweight because it only consists of a header of 2 bytes and a payload of maximum 256 MB and a Quality of Service level (QoS) [28]. There are 3 types quality of service levels which are listed below

1. Level 0 - Employs a at-most-onec semantic where the publisher sends a message without an acknowledgement and where the broker does not save the message, more commonly used for sending non-critical messages.

2. Level 1 - Employs an at-least-once semantic where the publisher receives an acknowledgement atleast once from the intended recipient. This is done by sending a PUBACK message to the publishers and until then the publisher will store the message and try to resend it. This type of message level could be used for shutting down nodes on different locations.

3. Level 2 - Employs an exactly-once semantic and is the most reliable level because it guarantees that the message is received, this is done by first sending a message stating that a level 2 message is inbound to the recipient, the recipient in this case replies that it is ready, the publisher relays the message and the recipient acknowledges it.

Moreover MQTT deploy something called a "Last Will and Testament" (LWT) for error handling, if a client disconnects abruptly which can be seen during power outages or unexepected network disturbances.

LWT is configured in the start for a client that connects to a broker, the broker will store it until a client disconnects abruptly, and broadcast the message to all subscribers that are connected to the topic that is published by the client. This ensures that the right precautions or actions are taken in the case of having a dead publisher.

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

An indirect consequence of using SSL/TSL for encryptions is that it augments a significant overhead to the messages which in turn goes against the philosophy of MQTT being a lightweight protocol [29]. This is something that left for the developer to think about whether it is feasible to have more data being sent over the wire which can affect performance.

2.4 Apache Kafka

Apache Kafka is a distributed streaming platform used for processing streamed data, this central platform scales elastically instead of having an individual message broker for each application [30]. Kafka is also a system used for storaging as it replicates and persists data in infinite time, the data is stored in-order and is durable can be read deterministically [30, p. 4].

The messages written in Kafka are batch-processed, and not processed individually which is a design choice that favours throughput over latency, furthermore messages in Kafka are partitioned into *topics* and these are further classified into a set of *partitions*. The partitions contains messages that are augmented in an incremental way, and is read from lowest index to highest [30, p. 5].

The time-ordering of a message is only tied with one partition and not with rest of the partitions of a topic, each of these partitions can be allocated on different servers and is a key factor to enhancing scalability horizontally. This is illustrated in Figure 2.5

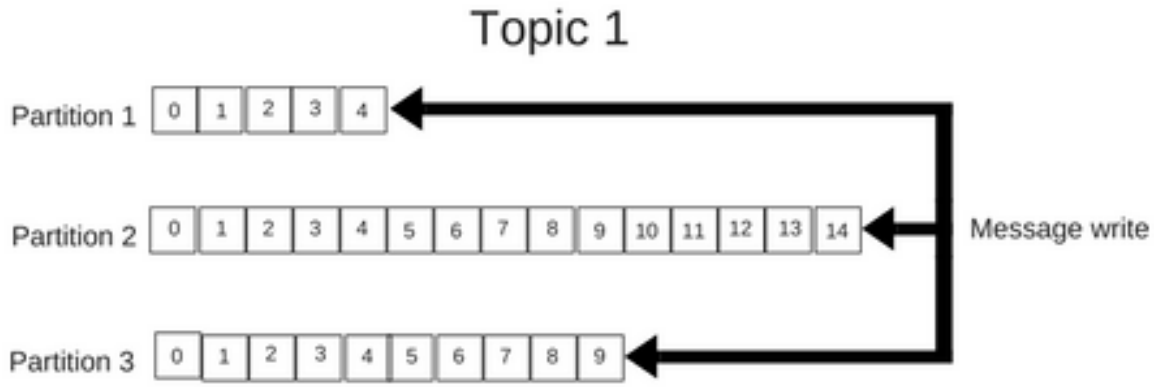

Figure 2.5: Topic partitioning

The messages within a partition is consumed by a reader and to keep track of which message has been read an *offset* is kept in the metadata in the case when a reader stops and starts reading later on.

Moreover a consumer cooperates with other consumers in a group to deplete a topic and there can only be one consumer from a consumer group for a partition. If a consumer fails the group will rebalance itself to take over the partition that was being consumed by the failed consumer [30, p. 7].

A Kafka server also called a *broker*, is responsible for commiting messages to the disk and giving offset measurements for producers, and responds to consumers requests for messages within a partition. These brokers will function with other brokers to form a *cluster*. This cluster deploys a leadership architecture where one broker will service as a leader for a partition and a so called *controller* is used for appointing leaders and to work as a failure detector. To increase the durability multiple brokers can be appointed to a partition and the partition is therefore replicated to other brokers.

A configuration metric for brokers is the duration for how long to keep messages stored in disk or how much to store. This is an important feature of Kafka because it allows flexibility for keeping messages of a specific topic much longer than others. Another configuration parameter within Kafka is to keep the latests message of a specific key.

2.4.1 Apache Zookeeper

To help maintain a cluster of Kafka servers, Apache Kafka utilizes Zookeeper to help keep track on metadata of the clusters and information regarding consumers. Zookeeper works as key-value manager that can help out with the synchronization aspects for Kafka such as leader election, crash detection, group membership management and metadata management [31, p. 11].

Zookeeper and Kafka works in symbiosis, where ZooKeeper tries to offer more control to issues that arises in a system with multiple clusters. Examples of failures that can happen when you have distributed coordination of multiple servers is that messages from one process to another process can be delayed, the clock synchronization of different servers can lead to incorrect decisions on when a certain message has arrived [31, p. 8].

2.5 RabbitMQ

RabbitMQ is a message broker that utilizes AMQP in an efficient and scalable way alongside other protocols. RabbitMQ is implemented in Erlang which uses the Actor-Model model. The Actor-Model is a conceptual model used for distributed computing and message passing, every entity in the model which are actors receives a message and acts upon them. These actors are seperated from one another and do not share memory, furthermore one actor can not change the state of another actor in a direct manner [8, p. 9][32]. The above mentioned reason is a key feature to why RabbitMQ is scalable and robust.

RabbitMQ is in comparison to Apache Kafka mainly centered around and built upon AMQP which is presented in section 2.3.1. RabbitMQ makes use of the properties from AMQP and makes further extensions to the protocol.

The extension of the routing capabilities that RabbitMQ implements for AMQP is the ***Exchange-to-Exchange*** binding which means that you can bind one exchange to another in order to create a more complex and advanced message topology. Another binding is the ***Alternate-Exchange*** that works as a similarly to a wildcard matcher where there are no defined matching bindings or queues for certain types of messages. The last routing enhancement is the ***Sender-selected*** binding which mitigates the problem that AMQP has where it can not specify a specific receiver for a message.[33]

A fundamental difference of RabbitMQ to Apache Kafka is that RabbitMQ tries to keep all messages in-memory instead of persisting them to secondary memory such as a disk. In Apache Kafka the retention mechanism of keeping messages for a set of time is usually done by writing to a disk with regards to the partitons of a topic. In RabbitMQ consumers will consume messages directly and relies on a ***prefetch-limit*** which can be seen as a counter for how many messages that has been unread and is an indicator for a consumer that is starting to lag. This is a limitation to the scalement with RabbitMQ because this prefetch limiter will cut of the consumer if it hits the threshhold resulting in stacking of messages.

The deliver semantic of RabbitMQ for the message queues is that they can offer ***at-most-once*** delivery and ***at-least-once*** delivery but never exactly once, in comparison to Kafka that offers ***exactly-once*** [34].

RabbitMQ is also focused on optimizing near-empty queues or empty queues, that is because as soon as an empty queue receives a message, the message goes directly to a consumer. In the case of non-empty queues the messages have to enqueued and dequeued which in turn results in a slower overall message processing.

2.5.1 RabbitMQ vs Apache Kafka

Chapter 3

Methodologies

Describe the engineering-related contents (preferably with models) and the research methodology and methods that are used in the degree project.

3.1 Engineering-related and scientific content

Applying engineering-related and scientific skills; modeling, analyzing, developing, and evaluating engineering-related and scientific content; correct choice of methods based on problem formulation; consciousness of aspects relating to society and ethics (if applicable).

As mentioned earlier, give a theoretical description of methodologies and methods and how these are applied in the degree project.

Chapter 4

Work

Describe the degree project.

Chapter 5

Result

Describe the results of the degree project.

Chapter 6

Conclusion

Describe the conclusions (reflect on the whole introduction given in Chapter 1).

Discuss the positive effects and the drawbacks.

Describe the evaluation of the results of the degree project.

Describe valid future work.

Bibliography

[1] EMC Digital Universe with Research Analysis by IDC. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*. 2014. URL: https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[2] Brendan Burns. *Designing Distributed System. Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly, 2017.

[3] Dotan Nahum Emrah Ayanoglu Yusuf Aytafffdfffd. *Mastering RabbitMQ. Master the art of developing message-based applications with RabbitMQ*. Packt Publishing Ltd, 2015.

[4] Gregor Hohpe. *Enterprise integration patterns : designing, building and deploying messaging solutions*. eng. The Addison-Wesley signature series. Boston: Addison-Wesley, 2004. ISBN: 0-321-20068-3.

[5] "Message-Oriented Middleware". In: *Middleware for Communications*. John Wiley Sons, Ltd, 2005, pp. 1–28. ISBN: 9780470862087. DOI: 10.1002/0470862084.ch1. URL: http://dx.doi.org/10.1002/0470862084.ch1.

[6] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[7] Vasileios Karagiannis et al. "A Survey on Application Layer Protocols for the Internet of Things". In: *Transaction on IoT and Cloud Computing* (2015). URL: https://pdfs.semanticscholar.org/ca6c/da8049b037a4a05d27d5be979767a5b802bd.pdf.

[8] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "Kafka Versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 227–238. ISBN: 978-1-4503-5065-5. DOI: 10.1145/3093742.3093908. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/3093742.3093908.

[9] *CloudAMQP - RabbitMQ as a Service*. URL: https://www.cloudamqp.com/ (visited on 03/08/2018).

[10] *CloudKarafka - Apache Kafka Message streaming as a Service*. URL: https://www.cloudkarafka.com/ (visited on 03/08/2018).

[11] Pieter Humphrey. *Understanding When to use RabbitMQ or Apache Kafka*. 2017. URL: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka (visited on 03/09/2018).

[12] Farshad Kooti et al. "Portrait of an Online Shopper: Understanding and Predicting Consumer Behavior". In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. WSDM '16. San Francisco, California, USA: ACM, 2016, pp. 205–214. ISBN: 978-1-4503-3716-8. DOI: 10.1145/2835776.2835831. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/2835776.2835831.

[13] Ryuichi Yamamoto. "Large-scale Health Information Database and Privacy Protection." In: *Japan Medical Association journal : JMAJ* 59.2-3 (Sept. 2016), pp. 91–109. ISSN: 1346-8650. URL: http://www.ncbi.nlm.nih.gov/pubmed/28299244%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5333617.

[14] *Key Changes with the General Data Protection Regulation*. URL: https://www.eugdpr.org/key-changes.html (visited on 03/19/2018).

[15] Bill Weihl et al. "Sustainable Data Centers". In: *XRDS* 17.4 (June 2011), pp. 8–12. ISSN: 1528-4972. DOI: 10.1145/1961678.1961679. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1961678.1961679.

[16] Anne Håkansson. "Portal of Research Methods and Methodologies for Research Projects and Degree Projects". In: *Computer Engineering, and Applied Computing WORLDCOMP* (2013), pp. 22–25. URL: http://www.diva-portal.org%20http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960.

[17] Jakub Korab. *Understanding Message Brokers*. ISBN: 9781491981535.

[18] Mary Cochran. *Persistence vs. Durability in Messaging. Do you know the difference? - RHD Blog*. 2016. URL: https://developers.redhat.com/blog/2016/08/10/persistence-vs-durability-in-messaging/ (visited on 03/23/2018).

[19] PTh Eugster et al. “The Many Faces of Publish/Subscribe”. In: (). URL: http://members.unine.ch/pascal.felber/publications/CS-03.pdf.

[20] Michele Albano et al. “Message-oriented middleware for smart grids”. In: (2014). DOI: 10.1016/j.csi.2014.08.002. URL: https://ac-els-cdn-com.focus.lib.kth.se/S0920548914000804/1-s2.0-S0920548914000804-main.pdf?%7B%5C_%7Dtid=76c79d76-8189-4398-8dc8-dda1e2a927e7%7B%5C&%7Dacdnat=1522229547%7B%5C_%7D.

[21] Joshua Kramer. *Advanced Message Queuing Protocol (AMQP)*. URL: http://delivery.acm.org.focus.lib.kth.se/10.1145/1660000/1653250/10379.html?ip=130.237.29.138%7B%5C&%7Did=1653250%7B%5C&%7Dacc=ACTIVE%20SERVICE%7B%5C&%7Dkey=74F7687761D7AE37.E53E9A92DC589BF3.4D4702B0C3E38B35.4D4702B0C3E38B35%7B%5C&%7D%7B%5C_%7D%7B%5C_%7Dacm%7B%5C_%7D%7B%5C_%7D=1522311187%7B%5C_%7D (visited on 03/29/2018).

[22] Piper Andy. *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP - VMware vFabric Blog - VMware Blogs*. 2013. URL: https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html (visited on 03/29/2018).

[23] Carl Trieloff et al. “AMQP Advanced Message Queuing Protocol Protocol Specification License”. In: (). URL: https://www.rabbitmq.com/resources/specs/amqp0-8.pdf.

[24] Emrah Ayanoglu, Yusuf Aytas, and Dotan Nahum. *Mastering RabbitMQ*. 2015, pp. 1–262. ISBN: 9781783981526.

[25] P Saint-Andre, K Smith, and R Tronçon. *XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies*. 2009, p. 310. ISBN: 9780596521264.

[26] *Stomp specifications*. URL: http://stomp.github.io/stomp-specification-1.1.html%7B%5C#%7DDesign%7B%5C_%7DPhilosophy (visited on 04/03/2018).

[27] “Piper,Diaz”. In: (2011). URL: https://www.ibm.com/podcasts/software/websphere/connectivity/piper%7B%5C_%7Ddiaz%7B%5C_%7Dnipper%7B%5C_%7Dmq%7B%5C_%7Dtt%7B%5C_%7D11182011.pdf.

[28] Margaret Rouse. *What is MQTT (MQ Telemetry Transport)? - Definition from WhatIs.com*. 2018. URL: https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport (visited on 04/03/2018).

[29] Todd Ouska. *Transport-level security tradeoffs using MQTT - IoT Design*. 2016. URL: http://iotdesign.embedded-computing.com/guest-blogs/transport-level-security-tradeoffs-using-mqtt/ (visited on 04/03/2018).

[30] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide : Real-time Data and Stream Processing at Scale*. O'Reilly Media, 2017. ISBN: 9781491936160. URL: https://books.google.se/books?id=qIjQjgEACAAJ.

[31] Flavio Junquera and Benjamin Reed. *Zookeeper - Distributed Process Coordination*. 2013, p. 238. URL: https://t.hao0.me/files/zookeeper.pdf.

[32] Brian Storti. *The actor model in 10 minutes*. 2015. URL: https://www.brianstorti.com/the-actor-model/ (visited on 04/08/2018).

[33] *RabbitMQ - Protocol Extensions*. URL: http://www.rabbitmq.com/extensions.html (visited on 04/08/2018).

[34] Neha Narkhede. *Exactly-once Semantics is Possible: Here's How Apache Kafka Does it*. 2017. URL: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ (visited on 04/08/2018).

Chapter A

Appendix Title