CODEN:LUTEDX/(TEIE-5446)/1-92(2020)

Industrial Electrical Engineering and Automation

Aggregating and utilizing sensor data

Maximillian Vilensten
Oskar Hermansson

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Aggregating and utilizing sensor data

Maximillian Vilensten
bas12mvi@student.lu.se
Oskar Hermansson
mat11ohe@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Christian Nyberg

Examiner: Mats Lilja

June 28, 2020

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The world of data is changing fast, while data always have been present in the world of computer science the importance of data has in the last couple of years grown, and it does not seem to be slowing down. The data usage is steadily increasing and the amount of data is growing, as well as the ever expanding amount of devices that produce data, especially in fields such at IoT and cloud computing. This has resulted in a need to make sure that ones data storage solution can effectively keep up without being too much of a hassle to maintain and upgrade. Data warehouses have for long been a popular choice for data storage partly due to its rigid structure. But in the world of big data, this structure can make it hard to adapt and add new types of data to the storage. A data lake can be a viable data storage solution if one seeks to achieve storage of many types of data from multiple sources since its open structure enables the user to practically store anything. This open structure however comes at a cost since the user now has to manage the raw data instead of the traditionally processed data of a data warehouse. Managing raw and unprocessed data is challenging as the purpose of the data may not be determined when the data is stored. This often leads to all data stored in a data lake to remain in the data lake without any form of structure nor purpose which quickly leads to the data lake turning into a data swamp. This thesis seeks to create a data lake solution that can store data from multiple sources and from different sensors while also making it very easy to add or remove data sources but also being resilient if the data format changes from its original form. While at the same time try to keep the infamous data swamp away. This involves sending the data from the sensors to a storage solution and then making sure that the data stored can be utilized by the data lakes owner as well as a third party.

Keywords: IoT, Data lake, Axis, Message broker, Cloud, Microsoft Azure, AWS, IBM cloud

Foreword and Acknowledgements

Firstly we would like to thank our assistant supervisor at Axis Mattias Backman and the engineering Manager at Cognimatics Mikael SW Nilsson for giving us the opportunity to have our thesis work at Axis and for being very helpful an communicative despite their hectic schedules.

A thank you should also be given to the Axis Core technology Systems for the great feedback that was given during presentations.

We would also like to thank our examiner at LTH Mats Lilja and to our supervisor Christian Nyberg for the general guidance and for their feedback during the final thesis writing process.

Finally we would like to thank all the developers at Axis that have given us guidance during the thesis and for the technical support when things didn't work.
Maximillian Vilensten and Oskar Hermansson

Table of Contents

List of Figures

Terminology

OASIS - is an abbreviation for **O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards and is a non profit organization for development of technical standards. Their focus is mostly around security and informational technology.

TLS - is an abbreviation for **T**ransport **L**ayer **S**ecurity and is a set of cryptography protocols that are used to secure communication between devices and services

WILDCARD - is a term used when having a placeholder for a string represented with a single character. It is commonly used to reduce file directories in computing.

MVP - is a abbreviation for **M**inimum **V**iable **P**roduct and is a term that describes a product with just enough features to make customers satisfied. Basically this is the bottom line of what should be allowed to deliver to a customer.

BI - is a abbreviation for **B**usiness **I**ntelligence and is referred in this thesis as business analytics using different types sales data to display this in different types of graphical representations.

SQL - is a abbreviation for **S**tructured **Q**uery **L**anguage and is a language primarily used to insert and retrieve data from tables inside databases.

AMQP - is a abbreviation for **A**dvanced **M**essage **Q**ueuing **P**rotocol and is an open protocol that is used by some message oriented middle ware. The protocol works on the application layer.

IDaaS - stands for **Id**entity **a**s **a** **s**ervice and is a phrase commonly used for cloud bases identity authorization platforms

IaaS - **I**nfrastructure **a**s **a** **S**ervice is a term most used on cloud services that have an instant computing infrastructure. An example would be data storage over the cloud or big data analysis.

SaaS - **S**oftware **a**s **a** **s**ervice is used to describe services that can include their entire services in the cloud, an example would be emailing where all work technically is done over the internet and all the user has locally is a program that can view the data that is stored online.

POS - **P**oint **o**f **S**ale is refered to data that has been obtain at a certain sales location. The data usually contains the amount of transactions,sales value,timestamps and other such data. This type of data is usually used to try to find sales trends and such for larger companies.

PaaS - **P**latform **a**s **a** **S**ervice A category of cloud computing services that provides a platform allowing customers to develop, run and manage applications.

HDFS - The **H**adoop **D**istributed **F**ile **S**ystem is a distributed file system that has been design to work well on enterprise and consumer computer hardware and was developed under the Apache foundation.

RSMB - **R**eally **S**mall **M**essage **B**roker was a message broker developed by IBM as a lightweight message broker alternative.

JMS - **J**ava **M**essage **S**ervice is an message-oriented middleware API that was developed for sending messages between multiple sources.

SDK - **S**oftware **D**evelopment **K**it is software made by a service created to ease development or usage of features provided in the original service.

Chapter 1

Introduction

1.1 Background

Axis Communications is one of the global leaders in network cameras for the physical security and video surveillance industries. The company now does not only try to maintain its expansion and growth but also focuses on staying in the forefront of providing exclusive services that will improve the customer experience and ease of use of their products, and at the same time differentiate them from their competitors.

With this in mind the company now strives to improve their current solution to collect and analyze data to be able to have many different types of input data from multiple sensors. It should also be able to handle adding of new sensors that would generate different input data for the storage without major integration issues. The collected information should be available and comprehensible to the customers. The intended customers are mostly system integrators tasked to display the data to the end users which could range from companies to individuals. Examples of this sensor data could be statistical information which is used for reports about customer flows in retail, banks, etc. This data produced by these sensors is currently stored in a traditional database, which is used by system integrators to present the end user with data for analysis. This sort of data storing poses a problem when adding a data type not currently supported by the current database, since adding a new source of data requires refactoring of a database. In the current solution data is sent from its source to its predetermined database and stored there. In the desired solution the default mode of transfer is used to transfer the data from its source to a storage medium. Axis seeks to improve the database with a more modern solution but are uncertain which solution that would be. A proposed solution to this as presented by Axis is the usage of data lakes, which solves the problem of adding data from new sources. A data lake is a data archive that stores its data in a raw format, and often stores all data from its user in a single location. This kind of repository makes managing and analysing large amounts of data more manageable, which is the reason for Axis interest in it.

1.2 Goal

The goal is to determine a solution for sending sensor data to a storage medium on either the cloud or non SQL based database that best suit the company. The data should come from multiple sources with different output formats in different locations and must therefore be able to handle this in an efficient manner. The database should also be able to identify commonality between data from different sources, such as data

coming from the same source, or sensor data that represent the same type of data, e.g. temperature data or data used to represent customer flow, while at the same time be able to sort out data that is deemed irrelevant to the customer. The database must also be able to handle losing one of the sensor inputs without creating a major error for the entire database. The end goal for the project would be to have at least one solution with a working prototype that clearly highlights the benefits and drawbacks of the solution.

1.3 Problem Specification

In order to create a feasible scope for this thesis a set of scientific questions were formulated as a basis. These scientific formulations were made by the thesis workers in consultation with their supervisors from both Axis and LTH until an accepted common ground was found. The final scientific questions were:

1. How should sensor data be stored for analysis?
    - Is there a way to analyze data prior to storing it?
    - Is there a way to determine which data should be stored?
    - Is there a way to store the same data in different formats?
2. What is needed in order to analyze data?
    - What metrics does the current Analysis Service used by Axis need?
    - What metrics would be needed in the future?
    - Is there a way to merge data in order to improve analyzing?
3. Is there a way to store multiple types of data in one database?
4. How should data be transferred? (Is there a way to efficiently transfer data from multiple sources to the database?)
5. Where should the relevance of data be determined? (Should all data be saved or is there a need to filter out incomplete or faulty data?)
    - Should data be filtered before being stored or after being stored?
    - What determines what input data should be deemed unnecessary and removed?
    - Does time affect the relevance of data?
6. Is there a way to utilize data?
    - Is there a way to visualize the analyzed data?
    - Are there tools that can utilize data?

1.4 Current solution

Axis current data aggregation solution is having all sensors, cameras and other devices that produces data to report their data to what is known as the Axis Store Data Manager. Axis Store Data Manager serves as both a database for storage with a traditional warehouse structure and also actively manages the data. The data stored in the data manager is then used for analysis by a third party. This third party uses the data to make visualizations of the data as well as offering fast queries into the data. This can be seen in figure 1.1

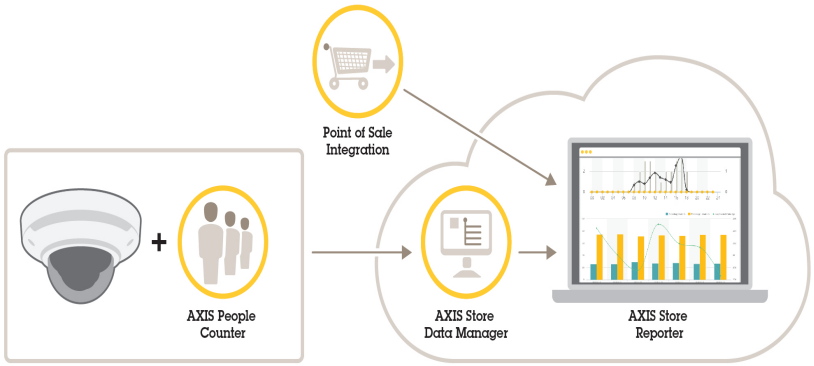

Figure 1.1: Diagram showing how the current solution at Axis works

Although the current solution works it is not without problems[64]. The most significant problem is that whenever a sensor in the system outputs data in a form not known by the data manager, the data manager has to be modified in order to be able to handle the new input and will therefore crash or have other issues if this is not properly handled beforehand. This also means that all third party applications using the data manager may have to change as well. There is also the problem of having no way of analysing the sensor produced data in real time, as of right now the sensors upload their data to the data manager once every 15 minutes. This is not a problem for all sensors as not all of the data is relevant in real time, but for some such as the Axis Queue monitor a real time analysis may be of interest. Due to a static database and a lack of real time analysis Axis is looking for ways to implement this or find solutions that can provide this for them. Axis current solution is an SQL database saving data in a fixed location. After this third parties need to access this data in order to graphically display the data for the end user. This was also considered a problem since Axis would prefer to keep third parties away from their data storage.

1.5 Purpose

The purpose of the thesis is to investigate different possibilities of solving the problem of adding new data sources to a database as well as how the data should be stored to optimize analysis of the data. This will be done by firstly researching and finding solutions within the scope given by Axis (i e. Cloud based, non SQL, adding new types of data without the need to adjust the data archive). If only one solution would be found, a prototype would be developed in more depth with a deeper analysis of why this would be the only feasible solution. If multiple solutions will be found these will then be eliminated by specifications given by Axis. Subsequent to this, a prototype will be developed for each of the remaining solutions. This will serve as a basis for the recommendation of which is most suited for Axis. There will also be a deeper investigation into whether the stored data can be automatically analyzed to some degree using a form of AI technology or machine learning. This area will not necessarily be included in the demo but a thorough analysis will be included in the thesis.

1.6 Axis

Axis communications is a Swedish hardware manufacturer that mainly specializes in security and video surveillance solutions for both companies and individuals. The company was founded in 1984 and began their journey selling print servers. This changed when Axis received a thesis worker pointing out that the systems could be used for network cameras, setting a new course for the company.

While network and security cameras are the main source of income for Axis the company has begun to widen its revenue streams by providing other services such as audio solutions and access control solutions for companies.

The company's headquarters is located in Lund Sweden where around 2000 of the company's 3600 employees work. Apart from its headquarters in Sweden Axis communications has a presence in more than 50 different countries around the world. In 2015 Canon bought a majority of the shares but regardless of that the company still remains independently run with little to few restrictions from its parent company.

1.7 Thesis scope and limitations

The thesis also contains limitations that were not stated in the scientific formulations in section 1.3. In difference to those limitations these have been set by either Axis or the department at LTH. The restrictions are as follows:

- The presented prototype should not be using a SQL database as main data storage point.
- The workflow must follow the Kanban framework used by Axis.

Resources

- Computers provided by Axis for development of software.
- Coaching from Axis staff and usage of their facilities.
- Sensor data provided by Axis.

Considerations

- Results need to have a slight consideration on financing, meaning that if the best found solutions is way more expensive than the rest, some sort of motivation on why the extra cost would be needed in the use case.
- The solution should try to be as platform independent as possible. This means that while it will be impossible to avoid cloud storage to be independent from any other company's ecosystem the proposed solution should partially or entirely be able to be implemented in other ecosystems if the client requests so. If the proposed final solution would end up completely locked to one ecosystem, a motivation on why the added productivity can be more beneficial in the intended thesis use case.

Chapter 2
Method

In this thesis a Kanban approach has been used to structure and plan the project. This chapter will go through how the work was carried out as a whole as well as how it was carried out per sprint, which can be seen in the following sections. The use of the Kanban work methodology was highly desired by Axis in order for them to be able to provide as much support in organising of workflow, and for them to be able to provide workload balance suggestions. With this in mind it was deemed highly beneficial to use this work methodology in combination with the fact that the thesis workers already were comfortable with working in Kanban. The entire thesis was split into four sprints which are documented in separate parts below. The group had daily stand-ups internally to get a clear picture what each member was doing, and also had weekly playbacks with the supervisor at Axis where the progress made by the team was presented as well as what each member should be working on for the next week. These kinds of meetings gave the company better insight into the progress of the thesis, but also gave both parties an excellent opportunity to communicate on how to prioritize the work going forward into future sprints and whether or not any extra task should be added to the backlog. This means that the work methodology included some structural reforms from Axis and is not pure Kanban.

2.1 Sprint I

The initial sprint started with a meeting with the Axis supervisor Mattias Backman, the purpose of the meeting was to determine where to start the project. Since there are a number of issues of having a static database, it was suggested that the data lake concept should be explored. As data lakes are often marketed to solve this very problem. Unfortunately Axis cameras and sensors do not currently have the capability for pushing their data to anything but Axis Data Manager, which means a message broker should probably be considered to transfer the data between Sensor and Storage. The conclusion of the meeting is presented in 2.1.

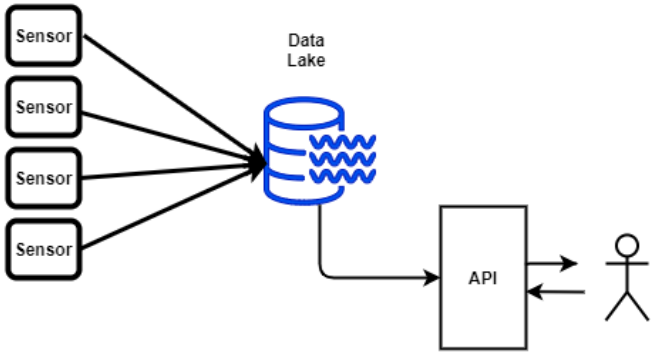

Figure 2.1: The initial suggested solution

Following the meeting a thorough research of most available data lake solutions was done followed by a broad research of possible message brokers compatible with one or more of the data lake solutions. The result generated by these researches could then be reduced based on either by being deprecated or having limited compatibilities with the data lakes. The result of this research can be found in chapter 3, listing the message brokers and data lakes considered for solving the problem. Subsequently to the research the width of the scope of the thesis needed to be evaluated as the number of message brokers resulted by the research was greater than expected and needed to be reduced. To do this a small send and receive test of all message brokers was done to determine usability and to check if the message model of the broker would make a difference on the suggested solution. Unfortunately there was no difference in this nor was there any eliminations being done based on the semantics offered by the different message brokers, since error handling is easily done by most data lakes. Since this method did not generate any reduction amongst the chosen message brokers it was suggested by the Axis supervisor that more metrics should be included. Lastly in the first sprint the most suitable message broker and data lake were selected for prototyping. The chosen broker was Apache Kafka for being the most compatible of message brokers as well as being the most well documented. The Azure data lake was selected for data storage since it was the most familiar platform to Axis Communications.

2.2 Sprint II

The Second Sprint started with the creating of a prototype utilizing Apache Kafka to transfer data from an arbitrary source to Azure Data Lake Storage. The data sent was sensor data provided by Axis to resemble actual data to be handled. This prototype was presented during a meeting to show the progress of the project and to confirm that it was going in the right direction. During this meeting the initial structure (see 2.1) of the solution was discussed, and three things was decided. The first one was that for sensors to upload individually to the lake wasn't optimal and should be avoided, the Second was the introduction of a raw data point. The third was that the message broker and data lake should be interchangeable if possible. This was due to not all of Axis current or future customers may be partial to use a specific brand of software. The conclusion of the three changes resulted in a new structure which is presented in figure 2.2

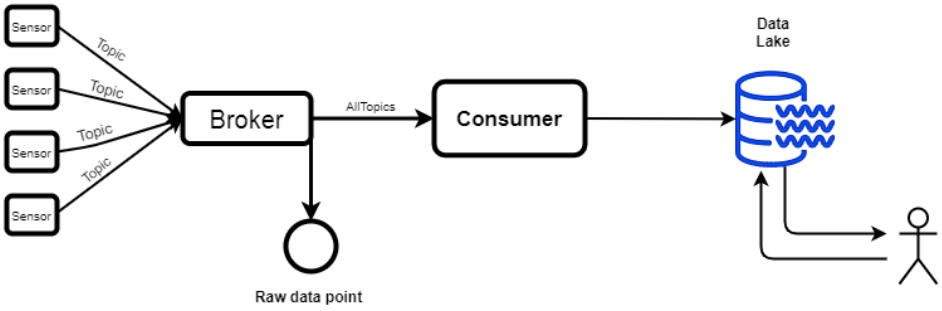

Figure 2.2: The updated solution

With the new structure came the conclusion that the broker used to implement the solution needed to support a publish/subscribe message modelThis resulted in ZeroMQ being eliminated due to it not having support for the publish/subscribe model. For the message broker to be interchangeable it needs to be directly or indirectly compatible with any of the data lakes. Whilst this is true for most of the chosen brokers it isn't true for Google pub/sub and AWS Kinesis as they both are bound to their respective distributor, which resulted in these also being eliminated. The initial research had resulted in 12 viable message brokers that now could be reduced to 9. The second half of the second sprint consisted of researching and prototyping data lake structures. This was done to find suitable metrics for a data lake comparison, as Axis had a very limited knowledge of data lakes prior to this thesis. The findings of the research as well as the data lake prototype were presented in a meeting at the end of the sprint. During this meeting the metrics for the message broker comparison was also discussed as the writers of this thesis found the current metrics to be insufficient.

2.3 Sprint III

With the new input given during the meting at the end of the second sprint a set of new aspects were taken into consideration that somewhat changed the prerequisites needed for the message broker. During the demo of the prototype Axis expressed a desire to increase the focus on security aspects and encryption while sending the messages through the message broker. Security was formerly quite low on the priority list when comparing different message brokers, and while Apache Kafka was the message broker used for demo purposes, built in security was not it's strongest suit. This new requirement lead to most of the brokers to be "phased out" due to them having either no built in security, or just authentication while the messages could still be sniffed out with software such as Wireshark. The strongest candidate for built in security at this point was RabbitMQ with it's built in TLS for securely sending data[6]. The testing showed that RabbitMQ had lower efficiency of message throughput and had a higher power consumption compared to the previously tested Apache Kafka broker. This could be due to the added encryption and it was considered that a custom made security scheme could be developed for Apache Kafka to compare how both brokers worked when secure. While researching how to add security to Kafka, a new message broker named Apache Pulsar was found with a similar structure as Apache Kafka but with higher scalability and built in security features. Therefore assessments where made to find out if the benefits of changing into Apache Pulsar would outweigh the costs in comparison to just develop some sort of encryption into Apache Kafka. It was deemed that testing Apache Pulsar would be more beneficial due to the built in security features and higher scalability. And since brokers were selected to be interchangeable the cost of changing would not be that high.

During testing between between RabittMQ and Apache Pulsar, results showed that Pulsar was indeed faster while also having a more robust built in security feature. While not leading to the elimination of RabittMQ for now, it lead to the decision of using Pulsar as the message broker intended for future demos. With the new Apache Pulsar message broker implemented with added security, the data aggregation phase was considered working and focus now shifted to the field of data utilization instead. This lead to a meeting with our representative on Axis discussing what type of data utilization would be interesting for them. During this meeting it was decided that for the MVP, three data utilization examples should be produced that could showcase the use of each layer of the finished data lake structure. The first was a way to push data from the data lake to one of Axis partners and their data warehouse. This was of high interest to Axis and was discussed before the thesis began. And was deemed a interesting way to test if data could be processed to the right format and sent within the lake in a streamlined fashion. The second decision was to transform incoming data to a format that fits users that work with BI, and export it to Microsoft's Power BI in order to create diagrams or in other ways visualize data. The third utilization required for the MVP is to visualize the data from the raw data point. More ways to utilize data were discussed but not put in MVP and can be read in the future work section and the schematic for the solutions now looked like figure 2.3.

The utilizations for the MVP where successfully developed but at the end of the sprint a question was asked by Axis if there was any way of using the solution locally. i.e without any involvement of the cloud for the small group of customers that are reluctant to use the cloud. Since this was not possible at that point, ways to implement some features to be able to run locally were researched to be presented at the next sprint meeting. While many possible solutions could be made, the most feasible one for the scope was to consider if the message broker could send the data to a local data storage as well as

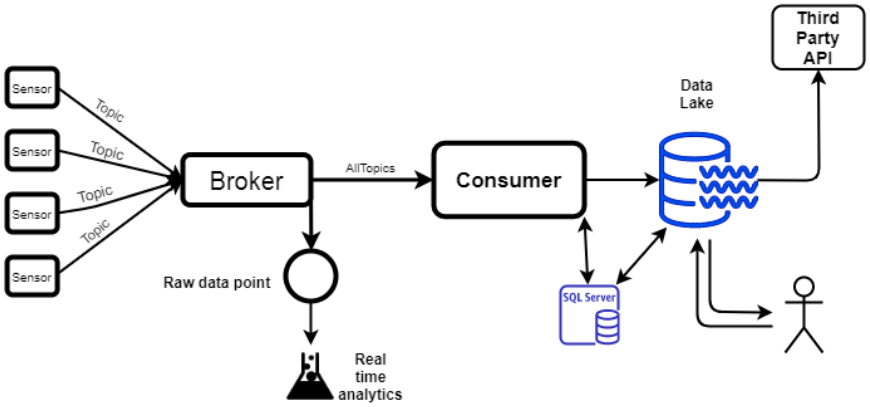

Figure 2.3: The second update of the solution

sending data up to the data lake. This would potentially solve the issue if one does not desire any sort of cloud computing or storage for their data, it could just be sent straight to a local data base instead circumventing the cloud.

2.4 Sprint IV

This sprint represented the last sprint for the project and was highly focused on finishing the demo and finalizing the analysis for both message brokers and data lakes as well as the thesis report. Since asked by Axis in a meeting to investigate possibilities to circumvent the cloud, code sending all data to an SQL data base was developed to show that data could be sent locally. While researching how to implement a data lake without the cloud, it showed that in order to get most capabilities and development freedom that could be provided by the cloud, only a Hadoop solution could be used. Due to the drastically different structure and how each feature works in Hadoop along the fact that not much time was left for the thesis, no complete Hadoop solution could be developed but a recommendation and high profile schematic was made. After presenting this to Axis the company was pleased with the results and focus completely changed to finish the analytics. To finalize what metrics where needed for the final evaluation a meeting was held with Axis. After some brainstorming alongside another extra meting with Axis Core Technology Systems, a finalized metric list was made and can be read in section 2.5 along with a priority list of the metrics that can be read in section 2.6. With this the final analysis began and it was decided to make a more theoretical approach in order to get a higher quality comparison in time. The analysis and how each test was conducted can be read about in chapter 4. After this some final updates on the demo was made to make sure that advantage was taken of the new knowledge gained from the analysis. With the analysis and demo done the only remaining tasks were to finalize the thesis report and hold a final presentation which was considered the end of the sprint.

2.5 Metrics

These were the metrics used to compare message brokers and were the result of several meetings and discussions with several departments at Axis.

- **Availability** - It is the capability of a system to maximize its uptime.
- **Interoperability** - Interoperability refers to the basic ability of different computerized products or systems to readily connect and exchange information with one another, in either implementation or access, without restriction.
- **Message delivery guarantee** - The Quality of Service offered by the broker(see 3.1).
- **Performance** - The amount of processing power needed per message sent.
- **Portability** - the usability of the same software in different environments. The prerequirement for portability is the generalized abstraction between the application logic and system interfaces. When software with the same functionality is produced for several computing platforms, portability is the key issue for development cost reduction
- **Scalability** - the adaptability of a system to cater to a growing number of tasks such as producers, consumers or messages.
- **Security** - Security in this instance refer to the measures required to develop a secure system. Where a broker with built in security features are valued higher as opposed to one that does not.
- **Throughput** - The rate of successful message delivery over a communication channel.
- **Usability** - Is the degree to which a software can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency and satisfaction in a quantified context of use.

2.6 Priority list

The aforementioned metrics for comparisons were given the following priority based on instructions and suggestions by Axis.

High

The list labeled high were the metrics that both Axis and the thesis writers found to be the most important ones and where the following:

- Scalability
- Security
- Portability

Medium

The list labeled medium were the metrics that both Axis and the thesis writers found to have importance to the work but not as much as the ones labeled high, and the metrics were the following:

- Throughput
- Message delivery guarantee

Low

The list labeled low were the metrics that both Axis and the thesis writers found to be interesting features to the message broker but were not necessary to be included if a message broker fulfilled the metrics in high or medium. The metrics were the following:

- Availability
- Usability
- Performance

2.7 Data Lake criteria

Since Axis had limited prior knowledge of data lakes, determining a list of metrics to compare them by was difficult, so instead a list of criteria Axis felt were important was established. The criteria were given any priority with the exception of data lake as a Service being more important than the others. These criteria are listed bellow.

- **Data Lake as a Service** - Axis asserted early in the process of data lake selection the preference of a data lake solution that was well established and already had a lot of the features pre built so testing the potential of data lakes would not take to long.
- **Data Lake Interopability** - Refers to the openness of the platform, i.e how well it communicates with products outside the platform.
- **Platform maturity** - The maturity of a platform does not imply that is better than its competitors but it often implies a larger number of users and more documentation.
- **Data Migration** - The ability to move data from the selected platform to a new one.

There are a number of other criteria often considered when selecting data lake services that are not mentioned here. Such as cost, performance, reliability and availability. These metrics were purposefully excluded as the they are difficult to compare to each other without having a full grasp on what data and how much data it will process. The cost of running any of the platforms is difficult to calculate without knowing the extent of the data to be processed. The purpose of the thesis was not to develop or determine the most optimal data lake, but rather suggest a solution to the specified problem by utilizing an appropriate data lake.

2.8 Source criticism

In order to make sure that sources in this thesis will hold a proper academic level and in an attempt to make sure that the reader can trust the information given, a quite rigid system was put in place to check sources for their reliability and transparency.

The first step when searching for information was to search sources trusted by the university such as LUBsearch in order to see if peer reviewed articles or publications could be found. When that was not available the Publications and dissertations section on the LTH website and Google scholar can also provide good sources that are to some degree more reliable than others. The process of doing this is quite time consuming in comparison to just googling, but usually provides with better results and one also can rest

assure that the content is of high quality and therefore spend less time on source criticism.

One can not always find the needed information by exclusively searching these mediums however and other sources have to be considered in order to get enough information to satisfy the need of this thesis. This resulted in the need to come up with a source criticism system which differs from medium and the system works as follows:

2.8.1 Books

Books are considered to be very valuable sources of information that are mostly on par with those acquired on LUBsearch. This is due to the fact that most printed works are to some extent checked for fact errors and hold an overall higher quality than most websites. Therefore in the case of a book checks were made to see if the book seems to be written in a serious manner. There was also some minor fact checking when the data presented in the book did not seem to match data acquired from other credible sources, this was a rare occurrence however.

2.8.2 Websites, E-books and PDF files

While many websites contain a lot of good and unbiased information, there is also a lot of them that do not. The internet also provides a cheaper option for individuals and companies to publish biased information that puts their favorite solution in a better light than it might have been deservant of. This is why a more rigorous system was put in place for source criticism when using Websites, E-books and PDF files. The process involved four different steps to determine how credible a source was to use for the thesis and to which extent they could be used. This means that some sources might not be credible in all aspects but were deemed credible enough in the particular use case and was used anyways but other data from the source should be taken with caution. In some cases where credible sources where not found, multiple semi credible sources where used to try and justify that this is the correct fact. This was exclusively the case for minor details in the technical background and was not deemed good enough as a source for any other types of statements in the thesis. The steps that where taken are the following:

1. **What are the editorial rights on the website?**
   There are sources on the web that have different privileges when it comes to who is allowed to edit information on the site. An example would be Wikipedia where more or less anyone is allowed to edit while sites such as ne.se only allow people knowledgeable in the subject to edit the information. This provides a more reliable data source than Wikipedia although the information might not technically be wrong it is also harder to confirm that it is correct.
2. **Are there any other incentives for the author?**
   Some web pages are written by corporations or individuals that have some sort of gain of promoting services that are not necessarily as good as they claim. This usually happens in the form of companies boasting features or performance of their solution without any proper proof. These type of exaggeration might have an affect on the end results of analytics and can therefore be unsuitable for use in a thesis.
3. **Does the website instill a serious impression to the reader?**
   There are institutions and groups that provide data and information in a detailed and more objective manner. This provides the reader with some confidence that the data can be trusted and that information might not have been made up on the spot since someone took the time and effort to properly present it.

4. **Does the site provide sources?**
   When writing any type of article or publications that includes statements one might want to see the sources as a reader to make sure that this statement is correct. If sources are not provided or some sort of credibility that the writer has knowledge of the subject one might find the information given in this publication to be not credible and might not be suited for thesis use.

2.9 Individual grading of source credibility

The sources numbered [15-22], [24], [30-33], [35], [39], [40-45], [58] and [64-74] are all documentation taken from the official source or from official partners of the service. While these might not be the most suitable sources for comparison of performance and such they do need to provide correct descriptions of their systems for developers to be able to work with them. This makes them perfectly usable in cases where technical aspects come into play but might not necessarily be good sources when quoting performance.

The sources labeled [1-14], [26], [57] and [59] are sources taken directly from LUBsearch, the university website or Google scholar and can be considered quite reliable as sources due to them already being peer reviewed and published by universities. This means that the sources are deemed most credible and require a minimal amount of fact checking before they where deemed highly trustworthy by the standards of this thesis.

The sources labeled [36] and [47-49] are books found on the subject. They fall under a similar category as the sources from LUBsearch and also have been deemed highly credible to be used in this thesis

The sources labeled [23], [25], [27-29], [34] and [60-63] are web pages that have been deemed trustworthy for use in this thesis.

The sources labeled [38], [46] and [50-56] Are websites, E-book and PDF files with less credible sources. These have been deemed to be not particularly trustworthy but have have been the most credible source found on the subject. These sources have only been used in combination with other sources to strengthen it claims or used to clarify minor details in the technical background. The information in these sources should be properly reviewed before any usage in any academical texts.

Chapter 3
Technical Background

This chapter presents Axis current solution for data aggregation as well as the concept of a data lake and a description of a message broker. The message brokers and data lake solutions evaluated in this thesis will also be presented here. This does not include all message brokers and data lakes as some were eliminated in the research process.

3.1 Message broker

The term Message Broker or occasionally integration broker is most commonly defined as "An intermediary program that translates a system's language from one internationally suitable language to another via a telecommunication medium", and its primary objective is to validate, transform and route messages. A message broker consists of three parts, a producer who sends the data, a consumer who receives the data, and a server acting as a broker in between the aforementioned. Simply put, the producers sends the message to the broker, and then the server sends the message to the consumer. How the server goes about this depends on the message model of the broker.

3.1.1 Point to point

A point-to-point communication model refers to communication connections between two endpoints, i.e a message is sent from one endpoint to another see figure 3.1. Message brokers implement this by sending a message from a producer to a previously specified queue on the server. The consumer may then fetch the messages from this queue, and when a message is consumed from the queue an acknowledgement is sent from the server to the producer. A producer can produce to any number of queues and a consumer can consume from any number of queues, but a message cannot be consumed by multiple consumers.

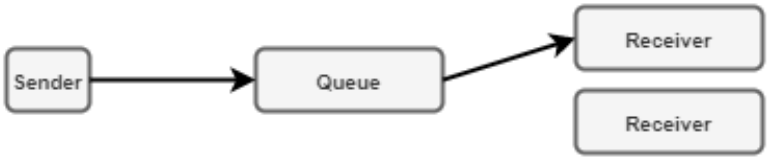

Figure 3.1: Image showing a message being sent from single source to a single receiver

3.1.2 Publish subscribe

In a publish/subscribe model (pattern) the producer published the message on a server with or without a specified topic. The consumer may subscribe to any number of messages based on one or both of the following filters:
Topic: The producer specifies a topic when it publishes a message, one or more consumers may then subscribe to the same topic to receive all messages published on that topic.
Content: A content based filter implies that the consumer specifies a number of attributes contained within the message, the server then returns all messages that satisfy these constraints.

Figure 3.2: Image showing a single publisher sending a message to two separate subscribers

Both of these filters support the so called one to many model, i.e. the same message can be read by any number of consumers as can be seen in figure 3.2. The publish/subscribe model has a lower risk of data loss as there are more components that need to fail for a message to be lost, some also offer message retention to ensure message delivery. Message retention refers to a message brokers ability to retain messages until all consumers have received the message or for a specific amount of time(retention period). In any system there is always a risk of failure, a message broker has three parties involved which is three parties that can fail independently or all together. To prevent data loss all message brokers offer a Quality of Service (QoS) level, which are classified below.

- **At least once:** To guarantee that all messages sent by the producer makes it to the consumer, the consumer sends an acknowledgment back to the producer and if the producer doesn't get the acknowledgment it sends the message again. This results in all messages arriving at least once even when the broker or consumer fails or the acknowledgement times out. This is done by simply sending the message again if no acknowledgement was received to the sender. Which means that there is a high risk of receiving duplicate messages, which need to be handled.
- **At most once:** To avoid getting duplicates the acknowledgement of having received a message is not sent, this guarantees there being no duplicates but doesn't take into account the package loss of not using acknowledgement.
- **Exactly once:** Exactly once guarantees exactly what it implies, only and exactly one message. This has no general solution and the approach on how to do it can differ from message brokers[57].

As previously mentioned some but not all messages brokers will be presented in the coming chapters. The list of messages was compiled during the initial research and does therefore include non-optimal options for the solution, since the requirements on the message brokers changed during the project. The criteria for selection and the changes made to them can be found in chapter 2.

3.2 ActiveMQ

ActiveMQ was developed by LogicBlaze in 2004 as an open source, asynchronous message broker. Written in Java, utilizing the Java Message Service (JMS), the reason for starting the ActiveMQ development was the lack of message brokers fully compliant with the Java Platform. Having originally been hosted by CodeHaus the code and trademark were donated to the Apache Foundation in 2007 where its founders together with the Apache community continues the development under an open source licence [65].

ActiveMQ supports both the point-to-point and publish/subscribe message models and the broker may be used as a queue, a topic or both simultaneously. When using the point-to-point message model the ActiveMQ broker acts as a load balancer routing messages from the queue to one of the available consumers. As opposed to the publish/subscribe model, where all messages in the queue are sent to each consumer subscribed to the topic. In both cases ActiveMQ does not guarantee that the messages are delivered immediately nor simultaneously for the consumers subscribing to the same topic, this is due to ActiveMQ sending its messages asynchronously, which is the result of having producers and consumers work independently. The producers process of composing and sending messages is separated from the consumers process of fetching it. As soon as the producer sends its message its task is done and it moves on to the next message to be sent, and the consumer on receiving a message is unaware of its origin. Having clients working independently of each other is called loose coupling and has a few benefits that include a higher throughput due to not having to wait for acknowledgement, and high flexibility by not needing to know the sender or receiver of the messages. This also allows for new clients to be added dynamically. This is also the reason for ActiveMQ offering the at least once semantic.

A consumer within ActiveMQ can be either durable or non-durable, the difference between them being that a durable consumers messages will be retained by the broker upon the broker disconnecting from the consumer, and a non-durable subscriber will lose all messages sent to it when disconnected[5].

Advantages

- The opportunity of dynamic queue creation
- Web UI for queue management
- The broker is configurable through XML
- The product is actively maintained

Disadvantages

- Messages must be sent to either queues or topics

ActiveMQ is often used for its flexibility; it has support for a large number of protocols such as STOMP, MQTT, AMQP, REST and OpenWire [3]. It is also compatible with Apache Zookeeper out of the box enabling replication, as well as offering a number of other useful features such as wildcards or Composite Destinations. Composite Destination allows the same message be sent to several queues.

3.3 Apache Kafka

Apache Kafka is a service provided by the Apache group in order to stream streams of records through a publish subscribe pattern. Apache themselves call this a distributed streaming platform which consists of three key capabilities.

- The ability to Publish and subscribe to streams of records.
- A way to durably Store streams of records.
- Process streams of records as they occur.

This means that the Kafka server is capable of having multiple producers(publishers) and consumers(subscriber) simultaneously and will still be capable of Exactly-Once data delivery.

The Kafka server is able to categorize input data with the use of topics and multiple topics can be created in a single Kafka server. Consumers are capable of consuming from not just one topic but multiple ones if desired. Kafka also has many different API's known as connectors that allow users to directly transfer data to and from the Kafka cluster with minor changes to the Kafka configuration files. This provides a fairly simple no programming required way to integrate the Kafka cluster to many different types of databases, cloud services and analytic tools given that a connector has already been developed for it. While Kafka mostly is recognized as a message broker for individual message packages Kafka also has the capability to stream data from one point to another using what Kafka calls the Streams API. In order to get a visual representation on how data can travel to and from the Kafka cluster one can view figure 3.3 which shows all the unique parts that can be added to the cluster as well in what direction the data can flow in each case.

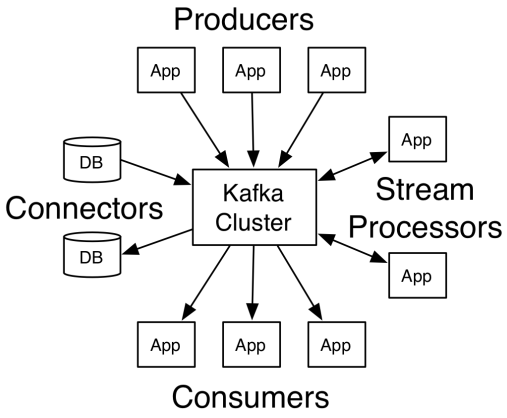

Figure 3.3: A schematic showing a Kafka Cluster working with connectors,producers,consumers and stream processors

Kafka also has the ability to save all messages within a retention period, this means that if a topic has a retention period of 24 hour all messages within these 24 hours will be saved in memory. These messages will only be deleted if instructed by a developer or if

the retention period runs out. This feature is meant to reduce loss of data if consumers somehow get bottlenecked or otherwise get disconnected to the Kafka cluster for a longer period of time. To better visualize how this works let's consider a scenario in which a topic contains one producer that writes data and two different consumers that read from it as can be seen in figure 3.4. Each numbered block can be seen as a message sent by the producer. It began at zero and has now written eleven messages and has started to seen a twelfth message. The first consumer named Consumer A has read up to message nine and the second consumer named Consumer B has managed to read eleven messages. The Kafka cluster remembers the position of all consumers which is known as individual offset. Let's say that for some reason Consumer A after reading the ninth message now loses connection to the Kafka cluster and after a short while manages to reconnect and starts to read messages again. The Kafka cluster will automatically recognize Consumer A and automatically requests it to start consuming from message ten (as long as message ten has not been deleted of course). This way no message has been lost to Consumer A and it can continue to consume without any worries[21].

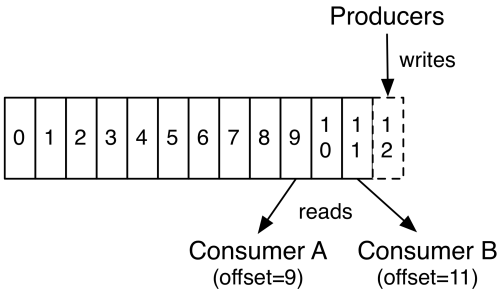

Figure 3.4: An image showing how messages are stored in a topic and how offsets for consumers work

Advantages

- Scalable
- Message retention
- Able to be clustered if desired
- Large support amongst companies
- SDK's available in many languages

Disadvantages

- Basic topic structure
- No topic wildcards selection
- Slows down with high number of message queues

Apache Kafka provides a scalable solution that is also clusterable and provides message retention. It is also open source making it an option that receives frequent updates from the community.

3.4 Apache Pulsar

Apache Pulsar is a pub/sub, massively scalable messaging system originally developed by Yahoo to address the several shortcomings of existing open source messengers at the time. Pulsar is open-source and is incubated under the Apache Software Foundation and was originally developed to support multi-tenant use cases. The Pulsar cluster is composed of a set of brokers and bookmakers and an Apache Zookeeper. The broker is a stateless component that receives, stores and delivers messages. Apache bookkeeper (Bookie) is used to store data in the cluster until they are consumed. Apache Zookeeper stores metadata about the Pulsar cluster used for configuration and coordination management. Apache Pulsar uses a generalized version of the two message models point-to-point and Publish/subscribe, to be able to offer both. The producers publish messages on a topic, these messages are then broadcast to any number of subscribers to consume the messages. Consumers under the same subscription may then choose what type of subscription should be used. Pulsar offers exclusive, failover, and shared subscriptions [66].

- **Exclusive** With an exclusive subscription only one consumer may subscribe to one topic at a time.
- **Failover** With a failover subscription any number of consumers may subscribe to the same topic, but there may only be one active consumer on a topic.
- **Shared** A shared subscription allows multiple consumers subscribe to the same topic and the messages are sent with a round-robin delivery method.

Regardless of which subscription selected Pulsar offers Message Retention, which allows the consumption of messages from a topic at a given date. To accomplish this Pulsar uses the Apache Bookkeeper who offers stream storage. This is also what enables Pulsar to guarantee that data isn't lost. Messages sent to a Pulsar cluster firstly arrive at the broker, the broker sends the message to a set of bookie nodes. When data is received in a bookie node it stores a copy in memory and writes it to a log. This log ensures that data is not lost even on machine failure. Storing on multiple bookmakers allows the pulsar cluster to survive even if multiple nodes fail and still guarantee a zero data loss, even in the presence of multiple hardware failures. The subscriptions also offers namespaces which are logical groupings of topics. The namespaces can be seen as a way to categorize topics in a more efficient manner, for example one can create a name space named public in which all topics inside can be read by all consumers and create a namespace called secret which only consumers with special permission would get access to.

Advantages

- Geo-replication
- Exactly once delivery
- Durable storage
- High throughput
- Broker and bookmakers scale independently

Disadvantages

- Many components involved
- Limited amount of documentation due to being relatively new

Apache Pulsar comes with a lot of features not mentioned in this thesis such as geo-replication, multi-tenancy and zero rebalancing time. While these and more are useful in most cases they are not relevant within the scope of the thesis. Multi-tenancy is used to run clusters with a massive amount of clients, which can be run on an enterprise scale level.

3.5 AWS Kinesis

AWS kinesis is a data streaming service that was designed to stream data to and from Amazon's cloud service AWS[15]. This means that the service requires an AWS account and membership to setup and use, regardless of where the data comes from or where the data is going[16]. The service itself is intended for streaming data and puts a high emphasis on its real time analytics capabilities and is not necessarily intended for batching data. AWS Kinesis is intended to send data and does not concern itself over messaging patterns such as pub/sub or push/pull and if messaging patterns are desired one has to use separate software on top of AWS Kinesis to achieve this[17][18][19]. This also means that AWS Kinesis itself can't decide what message delivery guarantee that can be used when sending messages, but instead that responsibility lies within the software that is used in conjunction to have a working message broker[20]. In order to reduce the risk of data loss AWS Kinesis has the ability to store data until the end of a certain retention period. This period is originally set to 24 hours but can be extended if desired[17]. This helps AWS Kinesis to guarantee exactly once message delivery and does not need to be enabled if the user does not desire this feature. The data is streamed using blobs which are truncated pieces of the original data, these blobs will then be sent through the pipeline with something that Amazon calls shards which can be seen as a data lane. These shards are capable of transporting 1Mb of data per second and each "pipeline" is capable of maintaining up to 5000 parallel shards giving the maximum theoretical transfer speed of 5Gb per second[17]. Since the intended main use for AWS Kinesis is to send data swiftly in the AWS a lot of effort has been made to make sure that sending data between AWS services goes as smooth as possible, and it does not require a lot of prior programming knowledge to begin sending data since Amazon already created specific connection services for most of the popular AWS services.

Advantages

- Swift integration with other AWS services
- Message retention
- High transfer speeds
- Capable of streaming

Disadvantages

- Locked to AWS platform
- Not as swift with non AWS software
- Not designed for batching data
- Requires separate software for use as message broker

AWS kinesis provides a fast way to transfer data between AWS services and can in this use case be considered as the most optimal choice. The requirement to use other software in order to use the service as a message broker makes the choice not as obvious when deciding sending the data outside the AWS ecosystem however.

3.6 MQTT brokers

MQTT(**M**essage **Q**ueuing **T**elemetry **T**ransport) is a lightweight publish-subscribe ISO standard protocol, developed by the OASIS organization as a message sending protocol. In this thesis three different message brokers making use of MQTT were tested in order to get a better representation on what MQTT can offer. Since they all use the same protocol they share many features, those will be described in this section in order to avoid redundancy, the following subsections numbered 3.7.1, 3.7.2 and 3.7.3 will describe what makes each of these message brokers unique in comparison to other MQTT brokers.

MQTT can use multiple transport protocols for data transport but the most common one that is used by a vast majority of MQTT solutions are using the TCP protocol. Data sent through MQTT are structured in data packets named MQTT Control Packets which consists of three parts, the fixed header, the variable header and the payload as can be seen in figure 3.5. The fixed header has to be included in each packet and contains a MQTT packet type field and a MQTT flag field and both of these fields are four bits in size each. The MQTT packet field should contain a number from zero to fifteen and each number represents a command that the packet tries to achieve. An example would be that in order to connect to the message broker a packet is initially sent where the value would be one which represents a connect request, after that another packet could be sent such as three would be a request to publish a message. The second field should contain a different set of flags that changes depending on what command the packet specified in the MQTT packet type field. This means that if a packet wanted to publish a message the MQTT packet field would be set to three as mentioned before, the second flags field would then contain information regarding if the message should be duplicated, if the message should be retained or deleted after a certain time period and also what quality of service(QOS) that should be used. This means that MQTT is capable of all the message delivery guarantees listed in section 3.1, but sadly the higher the quality of service the more acknowledgements and other measures are required to ensure this, which also means a reduced throughput of messages[22].

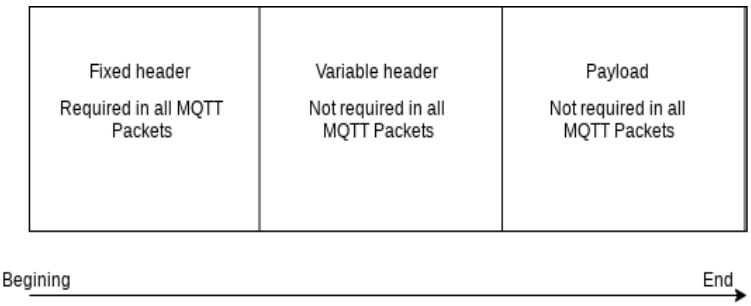

Figure 3.5: Visual representation of how MQTT data packet are structured

MQTT also also has the ability to update all clients in the server if any client has ungracefully disconnected using a feature that is called LWT(**L**ast **W**ill and **T**estament)making sure that no problems arise with clients trying to produce or consume data to a point that is no longer connected[23]. The Publish subscribe pattern also follows the simple structure as mentioned in section 3.1 and a visual representation can be seen in figure 3.6[24].

Advantages

- Multiple levels of message reliability
- Low bandwidth
- Highly popular standard

Disadvantages

- More difficult to scale than other options
- Cannot send large payloads such as images or video
- Not encrypted by standard

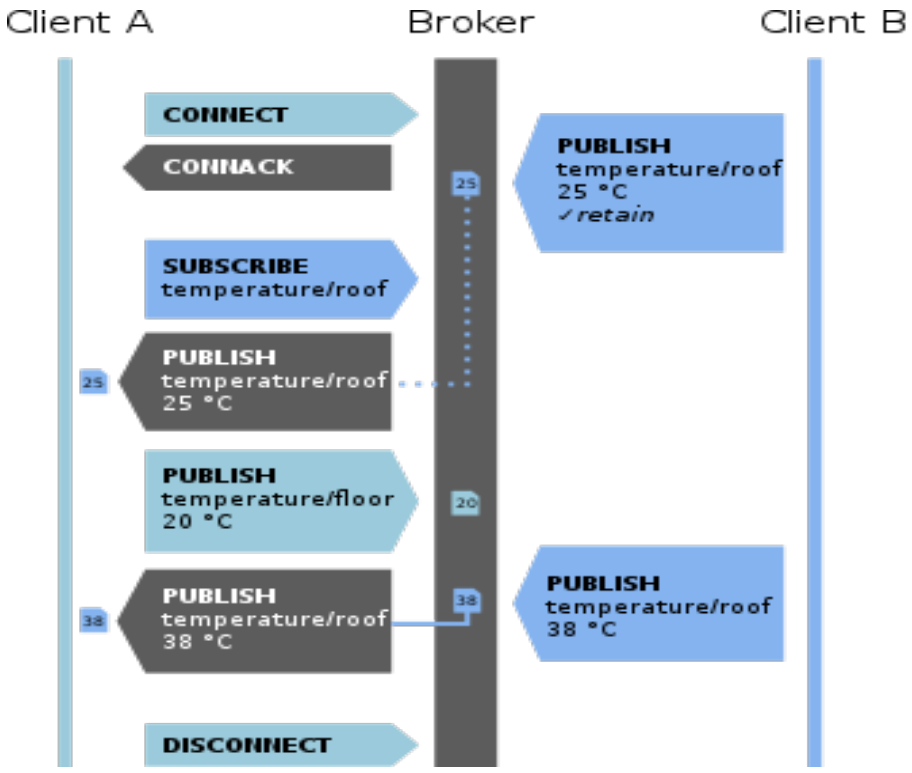

Figure 3.6: Showing a message being sent with QOS

Since MQTT is a standard, multiple different actors can use MQTT as a publish subscribe protocol and just give the message broker itself unique features, therefore the thesis will mostly explore the capabilities of MQTT itself but will slightly test out three different MQTT broker alternatives in order to see if anyone of these three brokers can add any improvement in the thesis use case. The three brokers are as follows:

3.6.1 Mosquitto

Mosquitto is a messaging broker that was specifically developed as a lightweight and low-power consuming MQTT message broker alternative[25]. It has been deemed as a

good choice for lightweight IoT devices such as Raspberry pies, due to its single thread application and the servers low ram usage have made Mosquitto quite popular in use cases where memory and bandwidth is important[26] [27]. Mosquitto was developed in C and development started in 2010 from a former IBM employee as an open source alternative to IBM's RSMB[28] and has since become part of the eclipse foundation[29].

3.6.2 VerneMQ

VerneMQ is a message broker developed in Erlang by Erlio GmbH as a scalable message broker with clustering capabilities for applications that require high scalability[30]. VerneMQ also provides multiple extra features to connect and authenticate to different databases such as PostgreSQL, MySQL, Redis and MongoDB in order to easily store data in the cloud[31]. It can also maintain message retention by storing data locally using LevelDB, this feature is also the reason why VerneMQ is unable to run on machines using Windows since LevelDB is incompatible with Windows.

3.6.3 HiveMQ

HiveMQ is a MQTT message broker that was developed by Hive themselves and places its main focus on sending messages in the world of IoT devices. The broker is developed in Java and consists of an open source edition and an enterprise edition with added features such as clustering capabilities and higher security. The broker is built for high scalability and promises ten million simultaneous clients per cluster making it capable of high volume traffic[32]. The open source edition works as a plain MQTT broker with no extra features outside what MQTT itself offers whatsoever. The enterprise edition however offers more in-depth SDK's and more ways to monitor data sent through the broker. The enterprise editions also contains Apache Kafka integration meaning that the message broker can send and receive messages from Apache Kafka brokers as well making it capable to send messages to MQTT capable clients as-well as Kafka clients[33].

3.7 NATS

This sub-chapter describes both NATS Messaging and NATS Streaming, NATS Messaging is a traditional message system and NATS Streaming is an extension of NATS Messaging. Either of these can be used as a message broker with NATS offering a lot more features. For this reason the reference made to NATS mentioned after this chapter will be referencing NATS Streaming [68].

NATS Messaging

Originally developed for cloud messaging, NATS is a lightweight, open-source messaging system. Developed by Synadia in the Go language with a core design principle of performance, scalability and ease of use. Although NATS was developed and is still maintained by Synadia, its still open-source under the Apache 2.0 Licence. Nats consists of the NATS server used as a broker, and the NATS connector framework for building senders and receivers. Nats server can be clustered. NATS supports both publish/subscribe and message queuing as well as the request reply message models. Request-reply in this case is simply that all messages sent send an ACK back to the sender to confirm delivery. This as opposed to publish/subscribe and message queue message models cannot be done asynchronously. In the case of publish/subscribe, messages are sent to the NATS server with specified topic, and the receiver may then consume from the server on a specified topic. This is as in any other pub/sub system but NATS uniquely cuts off subscriptions if there is a timeout in connection to the server. This provides high scalability at the cost of minor latency increase.

Nats Streaming

Nats does not by default support streaming of data but does so with the streaming service NATS streaming. NATS streaming uses channels instead of Topics or Queues for distributing messages, where a channel is an ordered collection of messages. Subscribing is instead on a specified channel who distributes messages according to the order specified. When subscribing in NATS streaming, a subscription is assigned one of the following types:

- **Regular:** Messages sent to channels are sent with a position to guarantee order of messages.
- **Durable:** The position of the last message consumed is stored on the streaming server. This position in the channel is maintained through disconnection to allow consumption to start where it previously stopped.
- **Queue:** Each message is consumed only once from the channel.

Regardless of subscription type any message sent that does get a ACK back is redelivered. In doing this at least once is guaranteed. By using the durable subscription type the clients can guarantee both At Most Once and At Least Once Delivery, and message persistence. Durable subscription also allows new subscribers to replay historical messages from existing channels.

Advantages (NATS Streaming)

- Easy to use for developers
- Extremely lightweight
- Client support for over 30 different programming languages

Disadvantages(NATS Streaming)

- Scalability is not well supported
- Messages order is topic independent

NATS Streaming provides subscribers messages in the order they were published by a single publisher but does not guarantee order delivery in case of multiple publishers. This means that message ordering is dependent on, and ordered by the publisher the messages were published from.

3.8 RabbitMQ

Developed by Rabbit Technologies in 2007 as an open source client, RabbitMQ is a message broker implementing the AMQP standard. RabbitMQ's support of AMQP goes beyond the standard guarantees of AMQP in a number of ways: it allows putting priority on consumers, improved support for asynchronous batch transfer, and a number of tools to handle the message lifecycle. It also has a feature called "Alternate exchange" which allows handling of messages that clients couldn't route. Messages sent by RabbitMQ are sent asynchronously between applications i.e sender and receiver does not need to be running simultaneously. These messages contain a payload and a routing key, where the routing key determines the exact queue where the message is to be delivered, it also contains a position in that queue in order to guarantee ordering of messages. When it comes to message the delivery guarantee of messages RabbitMQ provides an at least once guarantuee by default. It can however be changed in the settings to provide at most once delivery if desired [67].

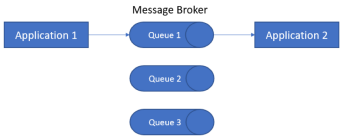

Figure 3.7: A figure showing a publisher sending data to a queue with one consumer

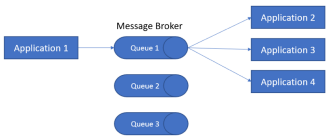

Figure 3.8: A figure showing a publisher sending data to a queue with multiple consumers

RabbitMQ supports both point-to-point and Publish/Subscribe message patterns as can be seen in figure 3.7 and 3.8, in the case of point-to-point a single queue for messages is used. And for Publish/Subscribe multiple queues are used to represent a topic. Messages in queues are deleted from the queue on consumption by default. This means that if there are multiple consumers on a topic none of them will receive all messages. This can however be changed but at a cost of throughput.

Advantages

- Ordering of messages
- Lightweight
- Highly scalable
- Multi-protocol support.

Disadvantages

- Higher latency than other options in this thesis
- Not as many message delivery guarantees as others in this thesis
- Not as fast as most other message brokers

Originally written in Erlang, RabbitMQ is language agnostic and supports languages like .NET, Python, PHP, Ruby etc. It can be deployed and used across most operating systems, and is supported by most cloud based services. This as well as having support for most messaging protocols such as STOMP and MQTT makes RabbitMQ a one size fits all of the message brokers.

3.9 Redis

Redis(**Re**mote **Di**ctionary **S**erver) is a data server solution developed by Redis Labs as an intended high speed server that is not restricted to just a string key-value store but can be able to use more advanced data structure stores, which is why Redis themselves calls it a data structures server[34]. The first release was published in 2009 and was developed in C since the developer thought that development in C would make the program smoother[35]. Since Redis is a data structure server it has many other usage patterns than message brokering such as usage as a web cache or even simply quick storage, however one of the most common use cases for Redis is the publish-subscribe message brokering feature[36]. Unlike most servers Redis saves it's data in memory and not a separate hard-drive making data access faster, however also making it less volatile[37]. As mentioned before Redis uses publish-subscribe message pattern in order to structure messages and are saved in key-value pairs where the topic is the key and the value contains the message. To get a better grasp on how Redis handles subscribers and publishers let's first take a look at figure 3.9[35].

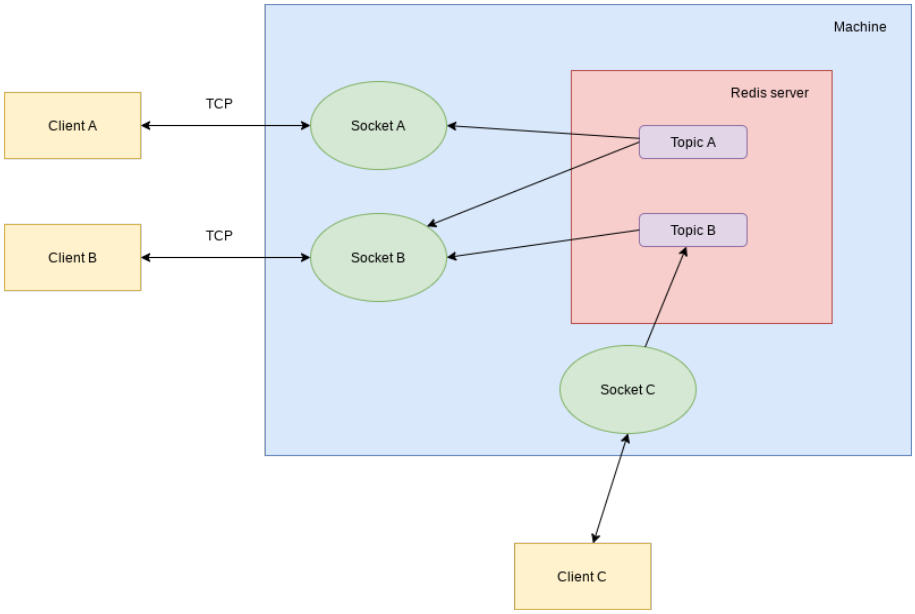

Figure 3.9: Diagram showing on a high level how Redis handles subscribers and publishers

The image features three yellow boxes that represent the clients named client A, client B and client C that can all be seen as three separate Clients that want to use a specific Redis server for either publishing or subscribing. All three clients firstly need to make a connection to the Redis server via sockets on the server machine and all data between the server and client will be sent through the socket using TCP[35]. This is where differences can be seen since client A and client B are subscribing and only has a data-flow from the topic while Client C is a publisher and therefore only sends data to the server. Since Client A and Client B only want to subscribe to data, they send a subscribe command to the Redis server along with a list of strings representing the topics that each client wants to subscribe to. As can be seen in figure 3.9 client A only subscribes to topic A

while Client B subscribes to both topic A and topic B meaning that it is possible for subscribers to subscribe to multiple topics simultaneously.

For client C the initial procedure is somewhat similar initially where it first sends a publish command to the Redis server establishing that client C wants to publish. Client C can now publish messages by sending a tuple where the first value is the topic name as a string and the second value is the message itself and can be more than just a string[38][39]. The technology behind the server gives Redis messages a theoretical maximum size of 512Mb which is quite large in comparison to many other message brokers[34]. Redis supports at most once data delivery meaning that there is no chance of redundancy but there is a possibility that subscribers might not receive the message.

Advantages

- Capable of sending large messages
- Data replication availability
- Clients are available in many programming languages
- Fast data access

Disadvantages

- Data stored in ram not always good
- Difficult to scale
- Difficult to manage keys

Due to the unique build of Redis it is able to provide users with a higher variety of storage features as well as sending large messages. Depending on use case one can argue for whether or not this can be considered an efficient message broker.

3.10 Google pub/sub(aka cloud pub/sub)

Google pub/sub is a fully-managed real-time messaging service developed by Google as a message broker solution for their cloud platform. This message broker is based inside the Google cloud platform which requires the user to have a Google cloud account in order to use it[40] . Since the solution is based in Google's cloud platform, it promises high scalability, reliability and also boasts a [41] 99.9 percent up-time for their service[42]. It can also be reached globally in a more simple manner than most publish subscribe message brokers since the service can be reached by anyone with an internet connection and the correct authentication[40]. As the name might suggest Google pub/sub exclusively uses the publish-subscribe message model in order to send data between peers and can also provide asynchronous messaging [41] in order to be more flexible and have a higher degree of availability. The publisher-subscriber relation can briefly be seen in figure 3.10.

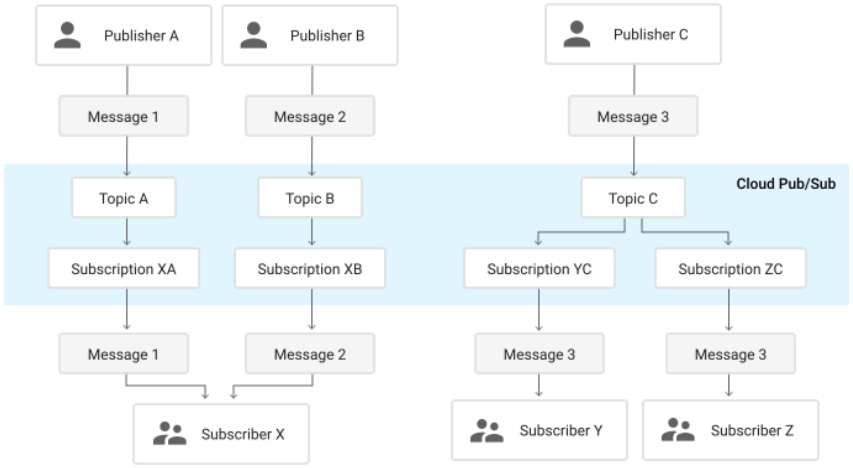

Figure 3.10: Diagram showing how Google pub/sub handles topics

While inspecting figure 3.10 one can notice that Google pub/sub uses topics to structure the data. It can also be seen that the topics are not limited to one subscriber but are capable of having multiple subscribers that are capable to "consume" simultaneously and receive the same message[41]. Google pub/sub also saves messages in the cloud in what Google calls the message storage, which is a storage unique for each topic that also contains offsets for each subscriber. This means that if a subscriber for some reason disconnects or ungracefully exits from the session they are still capable of receiving all messages that were sent during the particular subscribers offline period, a diagram showing the message storage can be seen in figure 3.11[41]. The messages that are sent through Google pub/sub are guaranteed at least once delivery, which as the name suggests guarantees that a subscriber will receive every message at least once[43].

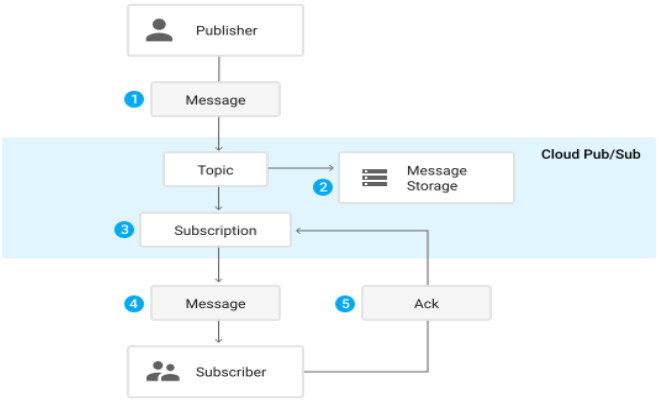

Figure 3.11: Diagram showing how Google pub/sub saves message history

Advantages

- Scalable
- Fast integration with Google cloud services
- Designed for high data loads
- Data retention

Disadvantages

- Platform locked
- Not as swift to use with non Google cloud software

Google pub/sub aims to make the switch to Google's platform easier since it provides a fast scalable solution that is tied to the Google Cloud platform. It is up to the user or the current use case to decide whether if the locked platform is a reasonable cost for the scalability and speed that google pub/sub provides.

3.11 ZeroMQ

ZeroMQ a message library which is developed by the group iMatix and is meant to be a lightweight message queue solution[44]. The code was written in C++ but has a wide array of language support from the ZeroMQ API. ZeroMQ uses the ZMTP transfer protocol for the message exchange between two peers[45]. Unlike most message queues and messaging services ZeroMQ directly connects via sockets and therefore no message broker is required in the middle[46].

This means that a socket has to be bound in order for proper communication to happen and it will also work slightly differently to many other message libraries[44]. In order to get a better understanding of how this works let's consider a simple "hello world"request-reply scenario as can be seen in figure 3.12. In order to set up a connection the server binds a port which by default would be port 5555 but can be changed. The server then awaits a request by a client and when a request is received (in this case Hello) the server sends a reply (which in this case is World). This exchange is in lockstep, meaning that the client side is expected to first send one request and later wait for a reply. If this format is not maintained for example if the client tries to send two requests in a row an error will occur[44]. When sending and receiving messages ZeroMQ can guarantee exactly once which makes it easier for the receiving end since no filtering for redundancy is required.[47]. ZeroMQ does not have active support from any of the major data lake solutions but is capable of streaming data to them using Apache spark streaming[48].

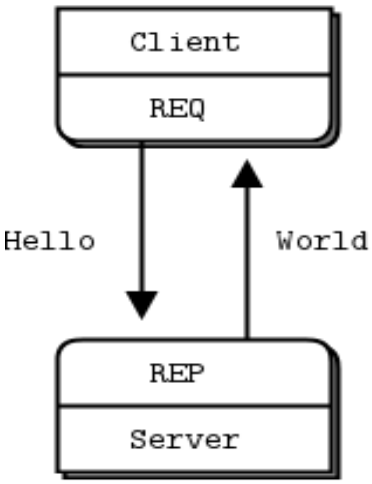

Figure 3.12: An example of a simple request reply using ZeroMQ

Besides request-reply messaging as used in the example above, ZeroMQ is also capable of Publish-subscribe messaging. This makes the publisher bind to a socket and the subscribers will try to receive messages from the bound port. If the publisher notices that none is subscribing, it will automatically drop all messages. A diagram of how this could work can be seen in figure 3.13[44]. ZeroMQ is capable of categorizing data by the use of topics[44].

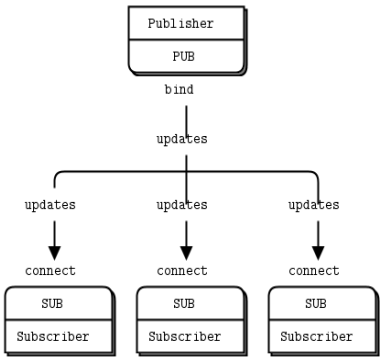

Figure 3.13: An example of a simple request reply using ZeroMQ

Advantages

- Lightweight
- Low latency
- Clients available in many languages

Disadvantages

- Brokerless
- Requires a lot of development for any advanced messaging patterns
- Message reliability

ZeroMQ markets itself to be one of the most lightweight message brokers available, to achieve this some features and design elements differ from its competitors. Depending on the use case this can be seen as either a benefit or a problem for a developer and the use case definitely should be considered before usage.

3.12 Data Lake(s)

The concept of a data lake is not clearly defined, one could think of data lakes as a data storage strategy instead of a new type of storage technology. There are some characteristics that are typical for a data lake however which can roughly be categorized into four different groups[51]. These groups would be data storage, data governance, flexible access and data ingestion. It is crucial for a data lake solution that wants to be functional for a longer period of time without turning into a data swamp to efficiently implement these cornerstones. In what way the developers chose to do so can change drastically depending on the end usage of the data lake. Therefore the thesis will mention what part of each category that will be important to the end customer. "A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale. You can store your data as-is, without having to first structure the data, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions." Simply put a data lake is storage solution that allows storage of data on any format without it being defined prior to storing it. The data lake manages and structures its data after its stored, as opposed to a traditional database that structures prior to storing it. It should also be noted that a data lake is not a technology but a storage concept, i.e there is no data lake as a Service. There are instead a number of companies that offers the tools needed to host a data lake on their platform. These are covered in the subsequent chapters.

There are a number of key components of a data lake and not all of which will be considered for this thesis, instead only the following will be taken into account since they were the most relevant to this project.

- **Data store**
  The defining feature of any data lake is that it can store data of any format and be very highly scalable.
- **Data governance**
  The Data Governance Institute defines data governance as "a set of principles and practices that ensure high quality through the complete lifecycle of your data". Data governance is more than just management of data, it is a system of rules and procedures that make sure these rules are followed. These rules focus on the availability, usability, consistency, integrity, and security of the data. Storing unstructured, structured and historical data at the same place will easily result in a so called "Data Swamp", and to prevent this a data governance system is required.
- **Flexible access** When it comes to data storage there is not only an issue with where and how to store all the data but also on whom that should have access to what part of that data. This is especially apparent in large corporations which include many types of business roles which all should have different types of access and that also are interested in their other types of data. It can become especially troublesome if the company works on a global scale in which the data also might need to be accessible globally and preferably in a secure manner. There are many ways to approach this problem and the solutions differ heavily depending on the organizations structure, size and global presence. If a data lake is properly initialized it can enable a multitude of different data access patterns for different roles and groups within one shared architecture, making it accessible without needing to create separate databases or other such solutions to determine data access. A data lake can also be easily reached globally by setting it up in the cloud making

them a highly viable option in data storage solutions at least when it comes to flexible access.

- **Data Ingestion**
  Data ingestion does in this instance refer to data being moved from multiple sources to the same destination. In the case of the data lake the data sources produces data of different formats and are often sent by a varying number of messaging protocols. Data ingested in this way needs to be processed, transformed, and stored in the right place even if it is in an previously unknown format.

A data lake offers more functionality than discussed in this thesis but the ones relevant and within the scope of this thesis are mentioned above. Not all data lake solutions available are mentioned in this chapter either, as not all data lakes offer the mentioned features. The following sections will describe different data Lake solutions and how they have chosen to implement the aforementioned features.

3.13 Azure

Microsoft Azure is a cloud based platform developed by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers. The services on Azure are quite extensive and range from basic data storage to data analysis, and amongst these services is the Azure data lake service which this chapter will focus on. Azure data lake is essentially a combination of services from the Azure Platform with the Azure Data Lake Storage as its base. Data is stored in the Azure Data Lake Storage, and managed by a number of tools and features from the Azure Platform to form a data lake solution [69]. Mentioned below are the tools and services offered by the Azure Platform to address the most crucial components of a data lake as described in section 3.12

Azure Data Lake Storage

Built on Azure Blob storage and Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2 offers storage of multiple petabytes of data of any format regardless of origin, and regardless of ingestion method. Azure Data Lake Storage Gen2 uses a directory hierarchy to organize its files and objects for efficient data access as well as making the structure intelligible. Being built for Big Data Analytics, Azure Data Lake Storage Gen2 also offers a highly scalable alternative for data storage along with a high level of input/output operations[70].

Azure Data Factory

To manage massive amounts of various forms of data is quite challenging, to achieve a stable data governance in their data lakes Microsoft offers the Azure Data Factory tool. Azure Data Factory consists of a collection of features for handling and manipulating data of any type or format. Data may be moved, copied, deleted, exported or transformed, all of which may be executed on demand, at specified time, or at a number of times within a given time frame. To achieve a stable Data Governance within a data lake there must be a continuous management of the new as well as old data in the lake. And this is what Azure Data Factory offers, a continuous managing of data through continuous applications of defined rules.

Azure Data Explorer

Data ingestion and processing the ingested data is a crucial part of any data lake and to do this Azure Data Lake uses Azure Data Explorer. Azure Data Explorer is a highly scalable data exploration service used to manage and log telemetry data, and is ideal for processing large amounts of large data. Azure Data Explorer supports several ingestion methods, including connectors to common services like Event Hub, programmatic ingestion using SDKs, such as .NET and Python and so on. Once ingested Azure Data Explorer offers a long range of service to analyze this data or store it using Azure Data Store.

Azure Active Directory

Azure Active Directory is an identity and access handling service provided by Microsoft through their cloud service Microsoft Azure for use with both Azure based applications as well as customer developed applications if desired. This service allows administrators to handle access of users, groups, catalogs and synchronizing meaning that there is potential for a quite high degree of autonomy since even single user can be allowed or restricted to data in even an individual file basis while not changing the user experience. Let's consider an example where a data scientist and a business analyst both try to get data from the lake. Both will simply log in using their own unique credentials and then have the capability to access some data but there is a difference in what data that is available to each role. And perhaps both of those roles should not be allowed to access data that is deemed classified by the company regardless if this data would have helped them speed up their work. The data scientist would probably also find different data relevant than the business analyst so for simplicity's sake one could remove access to the irrelevant information in the data lake and it would then not be accessible or visible making it easier to browse for what each role really needs.

One of the many advantages of using Azure as a platform for a data lake is the vast amount of tools and features offered on the Azure platform. Azure offers a lot more than mentioned in this thesis that includes features analysis tools such as HDInsigt and BI tools such as Azure Power BI.

3.14 AWS

Amazon Web Services is a cloud platform that provides cloud computing tools for companies and individuals. Amongst the services on the AWS platform is the AWS Lake Formation, which is a service on the AWS platform that allows customers to build, secure, and manage data lakes. AWS Lake Formation simplifies most of the steps needed to set up a data lake such as collecting, cleaning and cataloging data. The user can easily bring data from various sources into the data lake by using pre-defined templates. This data is then easily identified and analysed using one or more of the many AWS data analysis services. The services within AWS Lake formation needed to build a data lake within the scope of this thesis are mentioned below.

Amazon S3

Amazon Simple Storage Service is an object storage service on the AWS platform. Amazon S3 is designed to store data in buckets, with each object consisting of data and metadata and a unique key linking them to a bucket. Metadata is a set of name value pairs used to describe the object, these include cache control, content length and some default metadata such as date last modified. The metadata is used by S3 to organize and store data, since objects uploaded to Amazon S3 are heterogeneous which makes organizing of data difficult. Buckets are used to organize the Amazon S3 namespace at the highest level.

Data pipeline / AWS glue

Amazon Data Pipeline offers the user a way to automate the movement and transformation of data regardless of infrastructure. It may be used to build and deploy pipelines, where deployment may be done instantly or at a specified time. These pipelines may also be scheduled for one or any number of times, the service also helps the user to automate the flow of data inside of the AWS Data Lake.

Athena

Amazon Athena is a query service that enables its users to run SQL queries on data located in Amazon S3, and it may do so on both structured and unstructured data. It is also server less removing the need for infrastructure management, and the user doesn't have to worry about scaling servers or server failure.

Quicksight

Amazon Quicksight is a business intelligence and visualization tool on the AWS Platform. Quicksight offers interactive dashboards and machine learning insights on data from any AWS source.

AWS Lake Formation was developed to simplify and to remove the heavy lifting of setting up a data lake. With the Lake Formation tool it is easy to automate manual and time consuming steps, such as provisioning and configuring storage. With AWS Lake Formation customers can set up and begin using a data lake in days instead of months[71].

3.15 Google Cloud

The Google Cloud Platform was launched in 2008 and is a cloud computing service, these services vary from management tools to storage, to analytics and machine learning. The platform provides both Infrastructure as a Service and Platform as a Service. While not offering a data lake solution in its entirety, the Google cloud platform does offer most tools needed to build a data lake. With Google Cloud Storage as base and a number of tools running on the Google cloud platform a data lake can be built on the platform[73]. The tools needed to build a data lake on GCP are mentioned below.

Google Cloud Storage

Google Cloud Storage is most accurately described as an Infrastructure as a service (IaaS) and is used to access and store data on the Google Cloud Platform. Objects stored in the storage can be of any type or format but must be saved in blobs, the saved data are organised by buckets that each have a unique key. These buckets and objects are used by the ACL which is Google's platform tool for access control. The ACL is used to manage permissions for users, on both an individual and a role based level.

Google Cloud Dataflow

Google Cloud Dataflow is a fully managed streaming analytics service that minimizes latency, processing time, and cost through auto-scaling and batch processing[58]. Dataflow may also be used to prep and filter your data prior to passing it into another tool or data store, such as BigQuery and GCS. GCD also allows the building and execution of data pipelines on the Google cloud platform ecosystem, enabling scripts or applications to be run at a specified time and/or in specific intervals.

BigQuery

BigQuery is a fully-managed data warehouse on RESTful web service that enables scalable, cost-effective and fast analysis of big data located on the Google Cloud Platform. It is also a serverless Software as a service(SaaS) that utilizes Google's own original version of mapreduce algorithm. Mapreduce is simply put used to generate intermediate key-value pairs within the file system, and reduces this keyset to merge all the intermediate values associated with the same intermediate key. BigQuery also allows storing and querying massive amounts of datasets from GCS with queries in a standard SQL dialect. The service may also be used to create, delete and import data on Google storage.

Google Data Studio

Google Data Studio is a data visualization and BI tool that can also be used to combine data from multiple platforms into one place.

A data lake built on the Google Cloud platform is a more time consuming process as opposed to the out of the box data lakes offered by Microsoft or IBM. But Google does provide a long list of documentation on how to accomplish it, making it a shorter process than Hadoop.

3.16 IBM Data Lake

IBM:s platform IBM Cloud Computing offers a long list of features including infrastructure as a service (IaaS), software as a service (SaaS) and platform as a service (PaaS). There is also the ability to establish a data lake on the IBM platform, using Cloud Object Storage as storage along with tools offered on the platform[74]. Mentioned below are the tools and services on the IBM Platform considered most crucial for a data lake as mentioned in section 3.12

InfoSphere Information Governance Catalog

InfoSphere Information Governance Catalog is a collection of tools that allows handling, manipulation and identification of data. Working on both structured and unstructured data the catalog can be used to enact policies to data as well as track the lineage of data, which is the most important part of data governance.

IBM InfoSphere DataStage & Data Replication

IBM InfoSphere DataStage is a leading ETL platform that integrates data across multiple enterprise systems. It leverages a high performance parallel framework, available on-premises or in the cloud. The scalable platform provides extended metadata management and enterprise connectivity. It integrates heterogeneous data, including big data at rest (Hadoop-based) or big data in motion (stream-based), on both distributed and mainframe platforms.

Cloud Object Storage

Built to be used as a scalable centralized data repository IBM Cloud Object Storage allows storing of limitless amounts of data, in any format and from any source. Data is stored in its native format making data analytics require less conversion steps.

IBM Cloud Identity

IBM Cloud Identity is IBM's identity and access management solution for the cloud which also is known as IDaaS (**I**dentity **a**s-**a**-**s**ervice).The service offers the ability to change access privileges on a user to user basis or if desired one can also add work role access instead. The system also detects dubious activities within different accounts and can enforce multi factor authentication if deemed desirable by the system administrator. The service is not exclusive to IBM products but can be integrated to other software solutions by using what IBM calls connectors. Connectors work as secure API's for connecting to IBM Cloud Identity for authentication.

The IBM cloud platform has been around for a long time resulting in a vast amount of tools being on the platform, as well as a long list of documentation, and the process of moving to IBM cloud is quite extensive. The data lake solution is based on and compatible with Hadoop. It is also compatible with most message brokers supporting Kafka, Pulsar, and more. This and the fact that the IBM data lake solution very easily lives up to the criteria of a data lake makes it an excellent choice for a data lake.

3.17 Apache Hadoop

Apache Hadoop is a distributed computing framework platform which contains a collection of open-source software used to facilitate a network of computers to solve massive amounts of data. Written in Java Hadoop was developed for managing big data, Hadoop is built to scale up from a single server to thousands of machines and clusters. Where each of them offer local computation and storage. Hadoop's four core components are Hadoop Distributed File System, Hadoop YARN, Hadoop MapReduce and Hadoop Common[72]. Apache Hadoop is not a data lake at all but rather a collection of the tools needed to build one, the tools on Apache Hadoop used to build a data lake with required features is presented below.

Hadoop Distributed file system

HDFS is a storage system that allows a single data set to be stored across many different storage devices as if it was a single file, its also schema-less which means its able to store data of any format and from any source. This allows storing of structured and unstructured data as well as non relational data. HDFS consists of clusters and has all data stored in more than one place to ensure availability if one or more servers or clusters would fail. This gives a higher fault tolerance and makes storing large files easier. HDFS also contains management options for access to its data, users may have access to all data or a specified subset of the data.

Hadoop YARN

YARN is the cluster resource manager that allocates system resources to apps and jobs. This simplifies the process of mapping out the adequate resources necessary. It's one of the core components within the Hadoop infrastructure and schedules tasks around the nodes.

Hadoop MapReduce

Apache Hadoop MapReduce is an open-source implementation based on the Google MapReduce algorithm[59]. MapReduce is simply put used to generate intermediate key-value pairs within the file system, and reduce uses this keyset to merge all the intermediate values associated with the same intermediate key. This allows parallel execution of functions on several clusters. Apache Hadoop MapReduce also allows the execution of 'jobs' on data inside the Hadoop platform, such as mapping and applying user defined functions and combining, partitioning, sorting and merging data.

Apache Spark

Apache Spark is used to turn input data into Resilient Distributed data sets, and can also be used to transform large data clusters in a fault tolerant manner. It may also be used as an analysis engine on a HDFS based data lake to improve upon what mapreduce already does.

Sqoop

Is a tool for moving data from HDFS to a more traditional database and vice versa. Sqoop may also be used to exchange information amongst Hadoop and social database servers.

The Hadoop platform has been around for a while and is still the industry standard for building data lakes[60], and it has been for decades. But the age of the platform is not the only reason for its popularity, with age comes a lot of documentation which is always a plus, as well as most new systems developed are compatible in some way with at least one of the components on the Hadoop platform. Hadoop can be used as a base for a data lake but in using the Hadoop solution a lot more code needs to be written compared to the other options, where configuring existing tools covers most of the work. On the other side, as Hadoop is the original platform for big data analytics, and the Hadoop platform in its entirety is developed to be used for almost exactly this. With implementing a Hadoop data lake also comes a lot of freedom since there are few constraints on what operations that can be done.

Chapter 4
Analysis

This chapter will include the analysis of both the message brokers and data lakes that where mentioned in chapter 3.12. Since most of the heavy analysis was made during the later stages some brokers had already been excluded from the analysis due to differing circumstances. Which of these message brokers that were removed and why they were removed is mentioned in the first subchapter of the analysis chapter. The subsequent chapters analyze the remaining message brokers followed by the data lakes, and lastly the results will be presented in chapter 6 along with some reflections of the result.

4.1 Message brokers analysis

This section covers how the analysis stage for the message brokers was conducted and will also contain the end results of the individual message brokers. During the thesis some metrics where added, making some message brokers no longer feasible and where therefore eliminated at an earlier stage than the rest. These can be briefly read about in section 4.1.1. Section 4.1.2 will then cover the analysis and results of the rest of the remaining brokers and will be more detailed than the brief early elimination.

4.1.1 Early elimination

As mentioned in chapter 2 the initial research resulted in a large number of message brokers. A number that was reduced during the compiling of metrics for comparison. These early eliminations and the reason for their elimination are mentioned bellow:

- **ZeroMQ** For any of the suggested solutions in the project to work it requires the chosen message broker to have support for the pub/sub pattern. ZeroMQ has no support at all for this, not by default and not by plugin, which meant ZeroMQ had to be eliminated.
- **Google pub/sub** For a message broker to be bound to a specific platform isn't always a hindrance, especially when you are already using the right platform, unfortunately what platform Axis will be using is not known beforehand and may change at a later stage so choosing this would be unadvised.
- **AWS Kinesis** The need for brokers and lakes to be interchangeable makes Kinesis unsuitable.

4.1.2 Later analysis

Since most message brokers were developed with a special use case in mind one cannot easily decide on a message broker that is superior in every field when compared to others. All message brokers have their pros and cons and therefore one has to combine the results with a desired use case to truly find which is most suitable for the desired scenario. For this analysis nine key metrics where highlighted as the most important ones for Axis intended use case and a list containing these metrics can be seen in section 2.5. The metrics where then put into high, medium or low priority (as can be seen in section 2.6) in order to have an extra step of deciding the most suitable message broker. The priorities where set in conjunction with some Axis employees in order to also get a better grasp of what Axis themselves found most important. The analysis will be split up in separate smaller analyses for each metric in order to get a better understanding on which message brokers that are suitable for each metric and which that are not as suitable. This with a combination on how high of a priority each metric had will lead to which message broker this thesis found most suitable for this specific use case.

Availability

When comparing the availability between message brokers there are some external aspects that can affect the availability such as how the server containing the message broker was built or a power outage. These aspects are not something that any message broker can handle and will not be included in the analysis. The way that message brokers can affect the availability however are how they react when a broker stops working due to unforeseen circumstances. Consider a simple scenario where one publisher tries to send a message to a subscriber using a simple broker solution similar to what can be seen in figure 3.6. If the broker in that scenario brakes or halts for any reason no messages can be sent anymore and the system is as good as offline regardless of the amount of working subscribers and publishers. Message brokers that want to avoid this problem and maintain a higher level of availability therefore need be able to cluster brokers so that if one broker brakes or halts another broker can take its place. After extensive research of the message brokers on whether or not they had the capability of clustering brokers each message broker was put into one of three categories: native support, external support and no support. In the category native support the message brokers that were intended to be used as a cluster or with some degree with clustering in mind. These brokers can achieve clustering without external libraries or makeshift solutions and are often quite simple to setup. The second category named external support contains the message brokers that don't have native support but though plugin or external libraries provide clustering or that they don't provide all features when clustered[7].

Native support : Pulsar, Kafka, HiveMQ, VerneMQ, Redis, NATS, ActiveMQ

External support : RabbitMQ

No support: Mosquitto

With these results it would seem that most of the brokers included in this comparison have some sort of native support for clustering. However some of the message brokers (such as Apache Kafka and Apache Pulsar) require less configuration and coding to get up and running which could be seen as a benefit, but the other alternatives in the native support list are not difficult to set up but could be seen as somewhat more bothersome and might be taken into account if needed.

Interoperability

When implementing a new message broker it might be considered a bonus if there is a possibility to make sure that it would have some sort of similar message structure to other message brokers.It would perhaps be even better if it was able to send messages to different types of message brokers. This would especially be desirable in larger companies where different departments might have chosen different message brokers for their solutions but still need some way to communicate. Since producers and consumers are coded one can make the argument that all message brokers can be made to communicate with each other to some degree since you can write code that collects a message from one message broker that sends the acquired message of to another message brokers. Since one of the earlier metrics required that the message brokers should be publish/subscribe and contain topics, the high level structure of those message brokers that are still left are quite identical. So one could say that if one was comfortable with coding for one of the message brokers they would be very familiar to the others when it comes to the structure. The underlying strategies of sending and receiving messages are different, but this won't be apparent until huge amounts of data are sent and is somewhat of a scalability problem instead. With this in mind it was decided that the best way of measuring interoperability would be to compare which message brokers that had native support or official extensions for sending and receiving messages to other types of message brokers. The following list will contain each message broker and what other message systems that is can support. Some message brokers have other support such as directly sending data to databases or other forms of storage but this does not directly help with interoperability between message brokers and will not be included. All message brokers supports sending and receiving messages with itself and are therefore not mentioned in the list. The exception for this is when a message broker only supports sending messages to its own type of message protocol.

- ActiveMQ - AMQP, OpenWire, MQTT, STOMP, HornetQ and CORE
- Apache Pulsar - Apache Kafka (MQTT and AMQP support being developed)
- Apache Kafka - ActiveMQ, MQTT, RabbitMQ (Source), Redis (Sink), JMS , IbmMQ
- RabbitMQ - Qpid, openAMQ, AMQP, STOMP, MQTT
- HiveMQ - MQTT, Apache Kafka
- VerneMQ - MQTT
- Mosquitto - MQTT
- NATS - NATS (MQTT support being developed)
- Redis - Redis

Message delivery guarantee

There was no specified preference for Quality of Service for the message broker, however it is generally considered that exactly once for message delivery is exclusively better than at least once. Apache Kafka and Pulsar offers exactly once delivery as does any MQTT broker that supports Version 5 or higher, which in this case covers both HiveMQ and VerneMQ.

The remaining brokers support at least once as well as only once, and all of them has one way or another to support exactly once as well. But for the purpose of this thesis they will be considered to not have, since adding this feature to the message brokers takes up both time and resources.

Scalability

When it comes to scalability one should differentiate between brokers that are developed to handle big data and those who don't. Message brokers developed to handle big data such as Kafka or Pulsar scale a lot better since they need to in order to handle big data[10][11]. For a system to scale efficiently clusters needs to be used. Mosquitto is not a solid option when it comes to clustering since it has no support for it at all, making it the least scalable of the message brokers analysed. RabbitMQ needs a plugin to achieve clustering, but the queues in RabbitMQ can't be split even with clustering[7] which greatly limits its scalability[1]. For Redis to achieve clustering it needs to shard its dataset, which can be done by its built in sharding feature. This feature requires very little configuration. NATS does not scale well even though it has a clustering mode, this is due to the clustering mode is optimized to offer High Availability. To achieve anything close to high scalability with NATS Streaming scheduling mechanisms for the nodes needs to be introduced. ActiveMQ has native support of clustering that only need to a small number of configurations. This does require the user to handle the persistence of the cluster as ActiveMQ opens a file descriptor for each queue which may quickly run into limits. This can be handled by Apache Zookeeper and such, which ActiveMQ has native support for. The scalability of the brokers could be listed as:

Big Data Scalable : Pulsar, Kafka,

Highly Scalable : HiveMQ, VerneMQ, Redis, ActiveMQ,

Poorly Scalable : Mosquitto, NATS, RabbitMQ

Security

Most message brokers offers little when it comes to security as they are meant to be run on a local network. A message broker often only offers a TLS an/or a SSL encryption, and some form of authentications for producer and consumer, and this is the case for all of the brokers. All brokers offer some form of authentication for its producers and consumers, the implementation varies the result is the same. The protocols for encryptions are either TLS or SSL or either. At the time of writing of this thesis the SSL encryption is now deprecated, meaning a brokers offering only SSL encryptions should probably be avoided. The security protocols offered by the brokers are as follows.

TLS/SSL : Kafka, Pulsar, HiveMQ, VerneMQ, ActiveMQ

TLS : Redis, NATS, RabbitMQ

SSL : Mosquitto

With SSL now being deprecated this does not result in a one winner scenario but rather one loser scenario, and that loser being Mosquitto. There is however another winner in Apache Pulsar who, in addition to its TLS encryption offers end to end encryption. Pulsar uses a dynamically generated symmetric AES key to encrypt its messages. This means that Pulsar is secure over any communications channel, as opposed to the other brokers on the list, which makes it the most optimal choice if security is of high priority.

Throughput

The initial idea was for the thesis to contain an evaluation of throughput for the selected message brokers, this was scrapped since several studies were found that covers the throughput of different brokers and how they relate to each other. The first of these was a performance study [61] of the message protocol AMQP and MQTT, and it states that AMQP is far superior to MQTT when it comes to throughput. This means that ActiveMQ would be a better option than RabbitMQ(which [4] confirmed) or any of the other MQTT brokers. ActiveMQ is however outperformed by Moquitto according to [62] even though it uses the MQTT protocol, this is due to Mosquitto being a lightweight, which is beneficial for sending large amounts of messages. Mosquitto also outperforms NATS, who quite marginally outperforms RabbitMQ[14]. For the the brokers developed for big data it appears Pulsar outperforms Kafka by quite the margin, according to [13] Pulsars throughput is up to 2.5 times that of Kafka. This is similar to the findings of [9] which suggests Pulsar handles about three times as much throughput. Kafka and Pulsar outperforms the other brokers with a margin[8][14], the closest being Mosquitto[63]. Based on this the brokers could be ordered as:

High : Pulsar , Kafka , Mosquitto

Medium : ActiveMQ, NATS

Low : RabbitMQ, VerneMQ, HiveMQ, (Redis)

All of the studies and analysis mentioned above are all based on using a single broker and one producer and consumer, i.e it does not take scaling into account. This is rarely the use case in the industry but is the easiest way to compare the throughput of the brokers. This is also the reason for Mosquitto placing so high, had scalability being taken into account it would be down at the bottom since it barley scales at all.

Usability

Since usability is a highly subjective metric where opinions can change drastically depending on how experienced one is with the subject and also just what is perceived convenient on a person to person basis, one needs some set of constraints to make it somewhat quantifiable. Since this thesis initially assumes that the intended user for the message brokers have programming knowledge and some understanding of what a message broker is, one can assume that these criteria can be given for the end user as well. Therefore it would be wiser to compare how difficult it is to develop a certain feature for each message broker and find what types of struggles that where experienced along the way, and how easy these issues where to fix. It was decided that the simple publish subscribe examples for each message broker was to be used and the criteria for usability would be the following: Maturity of community, how well written the documentation is and rows of code for solution. These criteria might not cover how everyone perceives usability but they should give some indication on how easy it is to solve issues when stuck for individuals with prior coding experience. The following list rates each message broker from one to nine where one is considered most usable and later in descending order to nine which is least usable. This along with a brief motivation on how each fulfilled the three criteria in order to give a overview on why each placed as they did. The code for each message broker can be viewed in appendix A.

1. **MQTT (HiveMq,VerneMq and mosquitto)** - MQTT has been popular message system for many years and also has a very active and mature community

making, it very easy to ask for help regarding any issues with MQTT itself. In order to develop a simple publish/subscribe example one does not have to write large amounts of code and the code itself is very readable and comprehensible. Documentation for MQTT is really strong when it comes to the core features of MQTT itself but the separate brokers do sometimes offer exclusive features which can be somewhat poorly documented. Since these features where not needed however they won't affect most developers in any significant way.

2. **Apache Kafka** - Apache Kafka has become a strong contender when it comes to big data message brokers and it also has a large and mature community that is easily reached through platforms such as stack overflow or the official forum if any help is needed. The documentation is one of the best of the compared message brokers which is not only well written but also one of the most extensive ones with many code examples included for multiple programming languages. This makes it easy to understand what each and every function does and it also contains in depth schematics of how every part of the Kafka cluster works and how most key features work as well. Coding the publish/subscribe example was very easy due to the large community, and while some parts where not as straight forward initially the documentation made up for this so that with a little reading one could be right back on track.
3. **RabbitMQ** - RabbitMQ is also a quite popular choice of message broker that also has been around for quite a while. The community is quite mature and one can find help for most issues when searching around the right platforms. The documentation has some nice code examples and does contain most topics needed to be considered comprehensive but the layout of this documentation is quite confusing and it does not go into as much depth as some of the other documentations. Coding the example was not difficult but not as straight forward as some other message brokers since it has both a queue and publish subscribe pattern one has to specify a bit more before working as intended. But the end result worked fine and does not require an awful lot of code but still more than some.
4. **Apache Pulsar** - Apache Pulsar is on the rise and with that the community also grows but as of writing this thesis it is not very large and there might not be a lot of help to be found. The documentation is highly detailed and contains detailed schematics of the overall structure of a Pulsar cluster and how most feature work as well. The documentation does not update as frequently as Pulsar itself and does sometimes contain features that are deprecated or sometimes do not contain new features added. When browsing around the website one can find all of the new features and how they work and while inspecting the Javadoc for one can easily find what features are deprecated and also what to replace this with. So one can find all the information needed but its not necessarily as simple as just reading the documentation. Coding the example in Pulsar was simple due to code examples and detailed explanations and there are not a lot of rows of code either.
5. **ActiveMQ** - ActiveMQ is quite powerful as a message broker since it does support a lot of message systems and does not need a lot of code to change between them. The community around ActiveMQ however is not nearly as large as the ones higher up on this list which does mean that getting information about how to solve certain problems might take a while longer. The official documentation on their web page does not entirely work which is quite frustrating and requires the user to find other ways to solve the problem at hand. While coding the example most of the time spent on problem solving was spent searching the community instead of the documentation which is not uncommon but this included issues that most other

documentations should cover such as return types for certain functions. When coding, the community was helpful enough to make development not so frustrating and the code itself is short and quite comprehensible.

6. **NATS** - The community of NATS is dedicated but is quite small in comparison to most other message brokers on this list. The official documentation however is very detailed and contains lot of coding examples for most features that are included and would definitely be considered higher on this list if documentation was the only factor. Issues start to emerge however when coding and something not included in the documentation either went wrong or was to be added. Since the community is quite small it took a long time to find any information to solve any issues and it therefore took the longest time to finalize the coding example amongst all message brokers on the list. The finished code however wasn't too long and was comprehensible enough after one got to understand how it all works.
7. **Redis** - Redis is not exclusively a message broker but does contain the capability to act as a publish/subscribe broker. The Redis community itself is quite large but not the message broker part which is somewhat small. The official documentation does not cover a lot for messaging and it does not go into too much depth either. Features for the message broker seemingly differ greatly from library to library as well but is not mentioned making it the worst documentation on the list. There are dedicated Redis supporters however so the community can help out quite a lot regardless of its size. The finished code was also very comprehensible and worked well but it cannot make up for the lack of community and official documentation.

4.2 Data Lake comparison

As the Data Lakes as a Service was given a higher priority compared to the others, this analysis will focus on the three platforms that offer this. Hadoop and Google will not be discussed in this comparison for the simple reason that it does not offer this service, this section will focus entirely on IBM, AWS, and Azure data lake solutions and how they lives up to or don't live up to the criteria specified in section 2.7.

When reading articles online in order to get a better sense on the differences of the data lake solutions one gets the impression all data lakes that where tested for this thesis are more or less functionally identical[50][51][52].This seems to be true while setting up a basic structure for a data lake for this thesis anyways. And for the enterprise solutions (Azure, AWS, Google cloud and IBM Data Lake) all of them have big data and high performance in mind and under no circumstance under the thesis did any one preform slowly regardless of amount of data sent or analysed. When features, pricing and performance are somewhat similar one has to use other metrics to define what service to use.

Data Lake Interopability

While one can consider choosing an enterprise data lake as locking oneself up into a platform but it could be desirable to still have the ability to use third party services or even open source services to work on the stored data. It might also be nice if pushing data to other services was a simple task and did not require too much work to get done. It would also be nice to know that data could both be sent into and out of the data lake so that one does not get stuck in an undesired platform just to prevent data loss. A potential use case that would benefit from a more open data lake platform would be

if one wanted to use the data lake to filter and analyse the data but then sending that analysed data to an internal data warehouse away from the cloud. The following list contains the data lakes in order on how interoperable they where and how simple it was to send data outside the data lake. Where one is the most inoperable and three is the least inoperable.

1. **Microsoft Azure** - Azure offers a lot of different services that send data both to and from the data lake. Azure also contains the Azure marketplace that provides extra third party services that could be of interest to the user. Many features are also open source so that developers can actively contribute to add features they would want, and there's also a platform to request new features for the community to develop. The Azure feature data factory already has a lot of built in source and sink additions to easily send and receive data and during a conducted test to just send and receive data from an external data warehouse where it was perceived to be the most swift of the data lakes.
2. **IBM Data Lake** - The IBM data lake solutions use Amazons S3 API to structure data but the actual data is stored on IBM's servers. IBM has some partners which have developed third party features that can be used in conjunction with the data giving some third party support. It is also quite capable of migrating data from previous data storage solutions such as data warehouses into their data lake solution. The problems start to arise when trying to send data from the data lake to other services. There are features allowing one to send data such as support for some message brokers and IoT communication features etc. However it seems that it is their intention to keep data in their data lake and even offer to migrate former local data warehouses such as MySQL databases into their cloud to host them there instead. Trying to send data to and from a data warehouse was possible but quite tricky to pull off.
3. **AWS** - The initial stance of Amazon was to not allow open source development for the AWS platform but this has changed and Amazon does to some degree allow third party development[54]. They also have an Amazon marketplace where third parties can offer extra storage or managerial services that might but useful. Amazon has many services that integrates data from other services into their respective solutions on the cloud, and might be the best of the data lakes when it comes to that. There is however not a lot of services or information that helps you to use third party services outside their platform nor is it any intention from their side to help you get data off their platform in general.

Platform maturity

First to market their data lake solution was AWS with AWS Lake Formation, second was Microsoft with the Azure platform and last the IBM Data Lake. The amount of documentation and community adoptions follow this quite closely with IBM having very little documentation compared to the other two. IBM:s data lake is still quite new which is noticeable when setting one up with a very limited amount of documentation of common errors. The documentation on how to set up a data lake on IBM:s platform is however the most detailed and distinct making the set up quite manageable. The Azure Platform and its tools are well established and documented as is expected as the Azure platform has been around for a while. This is also true for the community around the Azure platform. The documentation for setting up an Azure data lake is very clear as is the documentation for all the tools offered on the platform. This does however only apply to the English documentation as the documentation written in the language native to the writers of this thesis, is awfully confusing and whoever wrote it clearly struggles

with the language. AWS Lake Formation does offer the largest number of tools amongst the platforms as well as the highest amount of documentation. The documentation is clear and makes setting up and managing a data lake on the AWS platform effortless. There is however a great lack of documentation for involving third party software, which is a drawback when working with data lakes.

Data migration

For companies such as Axis having a reliable way to store data is essential to their operation especially when handling a lot of different customers. Some customers do have objections to what services should be used or not used when storing or analyzing data. This could be for reasons such as the company does not want to help competitors. Another example would be the recent rise of Amazon which make many small American businesses refuse Amazon's products and services[55][56]. This requires Axis to sometimes migrate their data storage from one provider to another in order to comply with the customer's wishes. This is why it could be seen as very desirable to have the ability to quickly migrate data to or from this data lake so that they can adapt faster. All three data lake solutions have a wide offering of different services to migrate data from another service into their own platform. In this regard Amazon can be considered the best due to the wide range of migration tools. When it comes to migrating data of the data lake to another service Azure is the only data lake service that provides sending data directly from the data lake into either data warehouses outside the platform or even other third party services. Both AWS and IBM Data Lakes require either some sort of workaround or using multiple services making any potential migration of the platform quite difficult unless the new data storage solution for some reason accepts importing data from AWS. While most cloud storages have this ability many local storage's do not, making Azure the best in that regard.

Chapter 5
Prototyping

After all the analytics and comparisons had been made for both message brokers and data lakes, work could begin on the final prototype. This section will cover how the work progressed during the final demo development and how all knowledge gained during testing and analysing came into play. It will also cover some of the technical details in more depth. The section will be split into subcategories so that each aspect of the development can be read separately for those interested in specific parts. The sections are however to some degree written in a chronological order where some problems stumbled upon in the earlier sections might be resolved in later parts so in order to fully understand the process its encouraged to read the sections from top to bottom.

5.1 Message Broker

This section will cover the message broker structure set up i.e how data produced by sensors is sent from the sensors to the data lake. In the prototype Apache Pulsar is used, but the setup can be used for most message brokers that support the publish/subscribe pattern. The setup contains three main components, one broker, one consumer and any number of producers. In this setup each producer sends their message on an individual topic to the broker and the consumer consumes all message from all topics produced on the broker. These messages are then uploaded to Azure Data Lake Store.

Architecture

During the second sprint it was decided that the publish subscribe pattern should be used for the message broker architecture, this lead to what is shown in figure 5.1. The components and their role are described below.

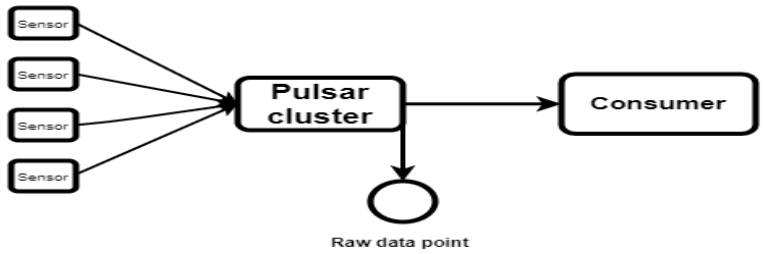

Figure 5.1: Intended message broker architecture

Broker

The broker in this case is a Pulsar cluster that consists of Apache Zookeeper and Apache bookkeeper. The broker used for the prototype is Pulsar standalone cluster which includes both the required zookeeper and bookkeeper. When running Pulsar in production or another large scale, standalone should be avoided.

Producer

Each producer publishes individual topics to not confuse the broker. The topic in this case is comprised of owner of the sensor, location of the sensor, the current date and the Serial number. Example of a topic.

```
Axis/Lund/20170113/ ACC8E20F09C
```

Ending the topic with a serial number ensures that no two sensors publishes to the same topic. While it is possible for producers to publish on the same topic it would not work for this prototype.

Consumer

The consumer subscribes to all topics from the Pulsar Broker, the consumer then uploads the data from each topic to the data lake. This means that the topic decides the structure of the folder hierarchy on the destination folder named STG. So if the previously mentioned topic is used the data from that topic would be placed in the folder path:

```
STG/Axis/Lund/2017/01/13/
```

This would create the folder hierarchy seen in figure 5.2.The data will be written to a file with the name of the serial number if such a file exists, otherwise such a file will be created first.

Figure 5.2: Folder structure in the STG folder

This solution was selected because with this solution adding a new producer requires no modifications on the broker or the consumer. When adding a new producer, all the producer needs to know is where to find the broker. The new producer may then start producing messages on its own topic which will eventually be uploaded to the data lake.

Raw data point

The raw data point is simply a consumer consuming on all messages on a broker.

There were three suggestions for the architecture solutions considered for this project, hosting all put producers on Azures platform, hosting consumer on the Azure platform. or hosting only the lake. All of which are explained below.

Host all but producers

The solution as shown in figure 5.3 is the one most similar to Axis current solution where each sensor(producer) individually sends their data, the difference being that messages sent are being sent to a broker first instead of immediately to a database. This leaves little to no hosting required for the user of the sensors, but the setting up of sensors becomes more extensive as the data sent needs to be encrypted on the sensor.

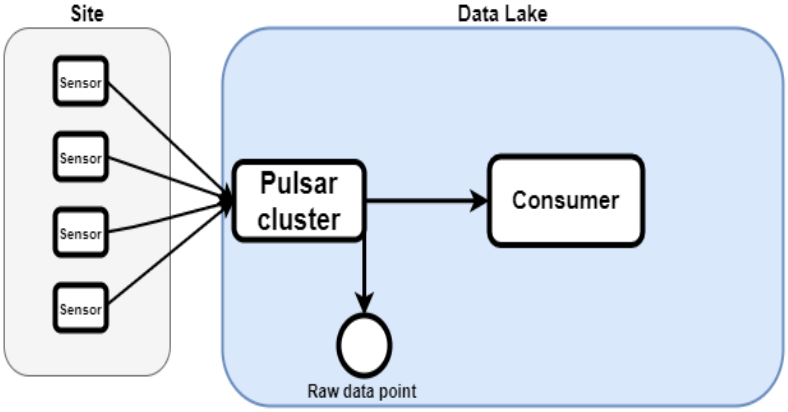

Figure 5.3: Message broker architecture for an all local solution

Host consumer and lake

By hosting the Pulsar cluster on the same site as the sensors(producers) as seen in figure 5.4 it becomes less complicated to add new sensors, as each new sensor now only needs to know the location of the Pulsar cluster. It also enables hosting the raw data point locally.

Only lake on Azure

Hosting on the cloud is generally more expensive than hosting locally, so hosting only the data lake on the cloud was considered. This solution seen in figure 5.5 requires more local set up than the others for the system integrator but in doing this makes the solution more modular. By using this set up changing either data storage solution or swapping to another message broker can be done with the least amount of effort. As opposed to the previous solutions where the data lake and some part of the message broker architecture is hosted on the same platform which creates issues when one of them needs to be replaced.

It was decided that the last solution should be the one to implement. This was due to it being the most interchangeable as the two first requires a lot of modifications if the buyer of the sensors does not want the used message broker. In the third solution switching to a new broker is a lot easier, as it requires no actual changes to the data lake. This also

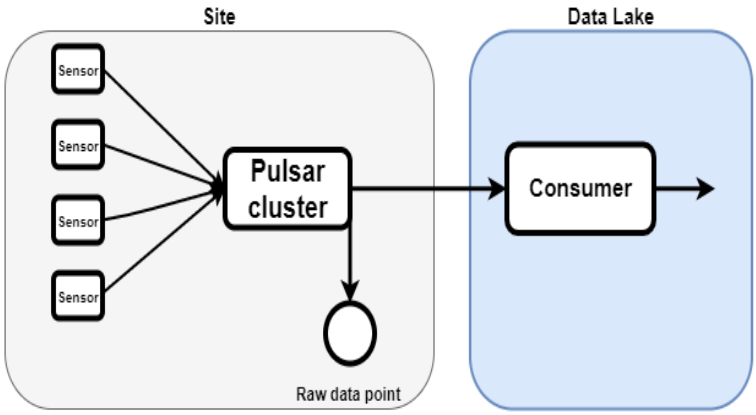

Figure 5.4: Message broker architecture for a hybrid solution

makes it easy for the user if they have a preference of data lake host. There was also the factor that this was the least costly solution as well as it having fewer security concerns since all data sent by the broker is sent locally, i.e less security needed.

5.2 Data Lake Architecture

With the ability to use the developed message broker to receive data from multiple sensors and send the data to the data lake, a lot of effort was now made to properly receive the data and to make sure that data will be available. When initializing an Azure data lake storage one receives an empty storage space without any form of structure in it. A close comparison would be to compare it to an empty hard drive since it has no structure and one technically can add any type of file in there without any restrictions. While this gives the data lakes an extreme freedom on how to store data it also requires more design from the developer to make it easily accessible and avoid what is commonly know as data swamp.

Firstly to confirm that a lack of structure would present an issue to the data lake a test was conducted where a lot of data from different sources and formats where inserted into the data lake without any structure whatsoever. Unsurprisingly the test clearly showed that after just a couple of days collection of data the system no longer was particularly easy to navigate. It was also discussed that perhaps not everyone that accesses the data lake is interested in all data or perhaps even shouldn't be allowed to access all data.

An example would be that people interested in business analytics might be more interested in sales figures than what type of sensor that calculated the amount of people that came into the store, which probably is more in tune with data scientists or electrical engineers. So with these considerations in mind, research began for a well organised data lake solution that to some degree also can keep all the data if desired and has the ability to give users a better experience by giving them data relevant for their interests.

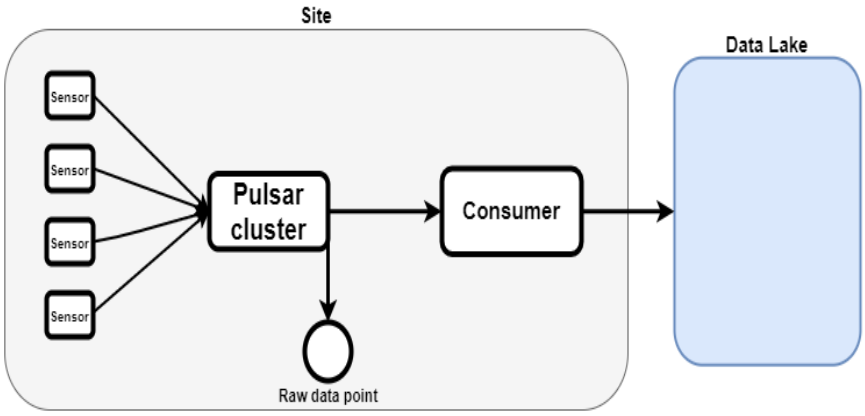

Figure 5.5: Message broker architecture for the final developed solution

Initially a design was made that would be a strict folder hierarchy where each sensor would have its own folder and save data there. This was not optimal since all data was still visible to everyone alongside that one could not find data based on specific times or dates this way. It was also discussed that if data was to be analysed and checked for errors or redundancies after being stored it would be a bit more bothersome since one then has to store all data and go through it all again to find all errors and then finally remove the errors without damaging the rest of the data. Therefore a way to make some initial data "cleaning" and redundancy removal was also needed in addition to the other points discussed previously.

Initially the data redundancy removal was implemented into the message brokers so that this aspect would not be taken into consideration when designing the data lake, but this affected the throughput to some degree and it was decided to try and fit that into the data lake design instead. While researching on data lake designs, a solution was found that would store data arriving from the sensors in a temporary middle location where redundancies, errors and other details that might be interesting to analyse or remove before storage could be done. This middle location would then send the cleaned and initially analysed data to the storage point in batches with a 15 minute interval. One can also create specific unique rules to filter data from this location more easily than if one needed to code this into the message brokers. The temporary storage was named "Stage" and the storage where all the data was stored was called "raw". These names were chosen since they were seemingly the most popular to use but other naming variants do popup. This "stage to raw" design was implemented by having a temporary "folder" inside the data lake that takes all the data from the sensors. After the allotted time which in this case was 15 minutes the data gets analysed by an Azure service called Azure data factory. In the Azure data factory code was written to check that if data was redundant and if it was it would delete this data. It also checked that the incoming Json data followed a correct template. To better understand this lets consider an example when a sensor sending data in the format mentioned in the appendix A.3.1. The message broker sent a tag along with the data so that the data lake knew what type of sensor that was sending the data. Since the data lake knew what type of sensor that sent the data it also knew how the expected format of this data should be. If the system noticed

that some tags were missing or gone it would be considered faulty and would be deleted. This first design step was implemented, tested and improved upon until it was working without any major issues and finally a schematic was drawn to illustrate the solution that can be seen in figure 5.6

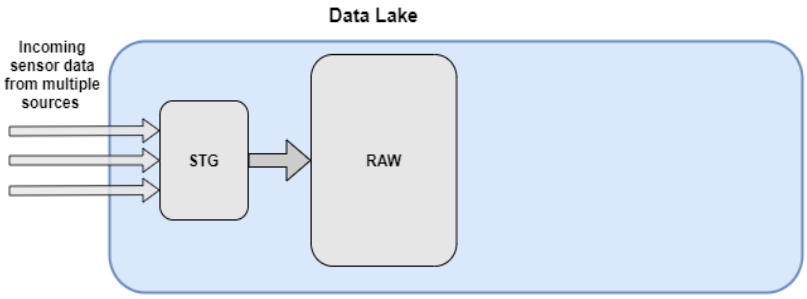

Figure 5.6: Image showing the first data lake design iteration

After implementing the design shown in figure 5.6 research began to see if it was possible to implement a way to make the data more accessible and faster to find in order to actually motivate that a data lake could be a better solution than a standard SQL database. During research an interesting approach to solve the issue of keeping as much data as possible but still retain a comprehensible structure was stumbled upon.

It suggested keeping the raw data point that saves all the data as the solution now already has but also adding extra "directories" that contains some already analysed or transformed data. An example would be if there would be a group in a company that exclusively only needed data on the amount of people that come in and out of one particular store it could be wise to create one directory that has this data in a nicely accessible format. If the customer only wants that data it will be very simple to log into the data lake and get this data, since it has already been set up the way they want it and if they would want some other data they can still access the raw data storage and get the data that way although with a little more work. This scenario might not be too common but still proves that there might be some benefit to this design if the newly added "directories" have been well thought out for the intended user base.

After further research an interesting development on that design was found where two extra "directories" named "transformed" and "processed" should be added, where processed would contain data that was most common for data scientists and transformed would consist of data more interesting to business intelligence workers. This seemed as the best design for this use case since the thesis workers knew that the solution in use now requires a bit of work from a third party that has to collect data from Axis data manager, which stores in a format that can be seen in section A.3.1 before being able to visualize this. If another directory could be made that contains precisely the data that they wanted, a lot of time and work could be saved making both parties happier. Work quickly began on implementing this "processed" directory along with testing to make sure that it worked as intended. To implement this, code was written in U-SQL that went through the data lake and found all relevant data, reformatted it to a format that was better suited for the third party and then sent that data to the "processed" directory. This code can be seen in the appendix on section A.2.1. The code also changes

the format to fit the third party so no extra time has to be spent to fix that afterwards, and this format can be seen in section A.3.3. The data lake now has an updated design that can be seen in figure 5.7.

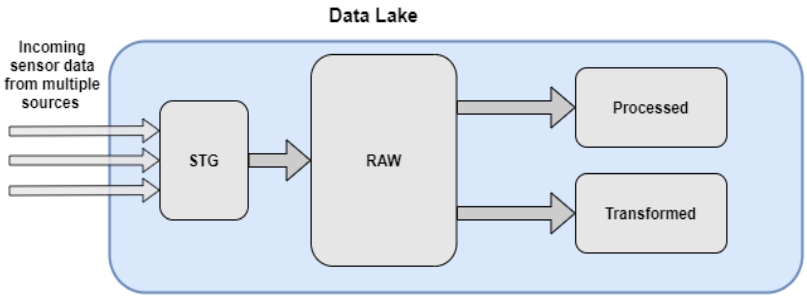

Figure 5.7: Image showing the second data lake design iteration containing a new processed section

A demo of a working proof of concept of the "processed" directory was shown to Axis in order get some feedback on the progress. Axis seemed approving and gave tips on extra features that can be read about in section 5.2.3. With a usable proof of concept and Axis feedback, work started on making a proof of concept for the "transformed" directory instead.

As mentioned previously the "transformed" directory should contain data that is more usable for business intelligence purposes. The idea being that their usual data needs to be differentiated from data scientist but in many companies their technical abilities differ as well. Since they might lack the technical abilities or the patience to search for data from different sources it might be a good idea to combine these type of data in a convenient way, with the possibility to have access to all the other data if desired as well albeit in a less convenient format. Json is also not the most common format when working with business intelligence so it would be better if one could find a way to convert the end result into a CSV format instead.

After discussions with Axis it was decided that the "transformed" directory should contain combined data from the Axis queue monitor and with POS data with the idea that it would be easier to find correlations on the amount of people in a queue and the amounts of actual sales to the store. Since both the POS data and the data coming from the Axis queue monitor was timestamped the Json files could be combined on the same timestamp using another U-SQL script that can be seen in section A.2.1. The U-SQL code takes out sales value, sales date, number of transactions, sales time and organisations id from the POS data while it takes the amount of people leaving the store from the Axis queue monitor and merges these two on the correct timestamp. The code then sorts it in order of organization number and then creates a CSV file and stores it in the "transformed" directory. This addition concluded the final design of the data lake architecture and was followed up with a meeting at Axis where a demo was shown to highlight the design aspect of the data lake that ended with some feedback on other aspects on the system

but otherwise approved of the design. A final diagram of how the system was designed can be seen in figure 5.8

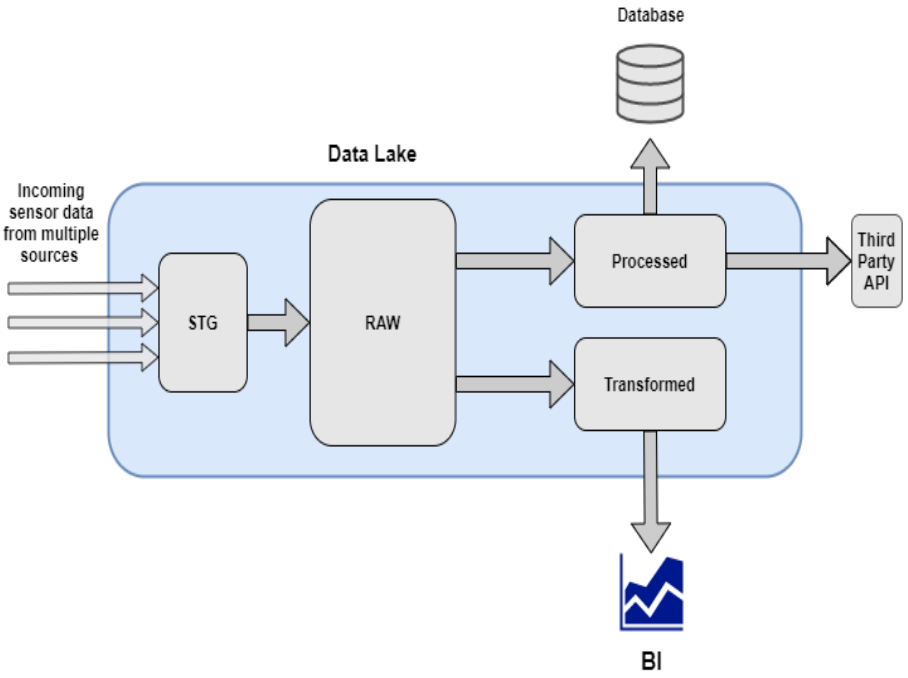

Figure 5.8: Image showing the final design of the data lake

5.2.1 Structure

While the architectural design of the data lake provided with a much improved data flow and made it more accessible in addition to comprehensible for a wider range of users it still did not entirely solve the problem of organizing the data. Because it was easier for different work groups to find data that was more relevant to their use case the data still was just dropped into the respective directory without any type of structure whatsoever.

This quickly proved to have a lot of issues in terms of finding relevant and correct data since storing all data that way solves the issue of finding where relevant data might be, but still not how to find specifically timed data or data from a specific sensor. For example if one wanted to find data for sensor A from between 11.00 and 12.00 on the first of April 2020 one could assume that finding data relevant to the intended use case was easier, there is no clear way to find what sensor that uploaded that data or during what time if one does not go into every Json file to read the sensor name and timestamp from there. This is clearly not optimal and work quickly began on implementing some kind of structure to avoid a "data swamp" in each directory of the data lake as well. Since the data lake is similar to a hard drive structure wise where each section in the architecture can be considered a folder in the root directory of the hard disk to some degree it was established that every directory should follow the same structure so that

it would not confuse the user. It should also work in a folder hierarchy so that it would be easy to navigate. This sentiment was also shared with Axis during a meeting where this issue was discussed in particular. Firstly a structure was discussed where each sensor had its own unique folder and in these folders the data was structured in order of date.

While not a bad design it required a lot navigation to get data from multiple sensors and it was not particularly easy to wildcard directories to make data retrieval easier. For example if one wanted to find data from three different sensors from one particular day the design mentioned above requires the user to firstly find the first sensor and then find the correct date, then go to the second and third sensor to do the same action. It would be a nicer option to be able to select what day that was needed and the select the sensors from there. The design did not consider multiple breaches either of the same store saved in one data lake either. So if a chain of stores had three stores in different cities but still wanted to save all data in the same data lake, all the data from the stores would have been mixed together. This problem could have been solved by naming the files differently for each store to differentiate them but this did not seem as a proper solution so this entire design was put on hold and another one was in the works instead.

After a lot of minor redesigns and meetings a final design was settled and can be seen in figure 5.9.

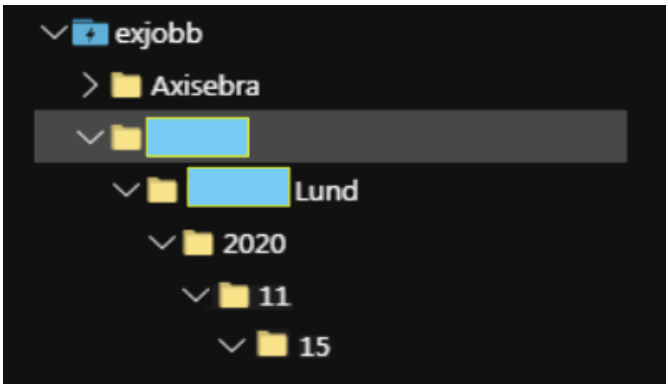

Figure 5.9: Image showing one of the folder structure iterations

As figure 5.9 shows the structure begun with the root folder which in the figure is named exjobb. This would of course be raw, stage, processed and transformed in the data lake architecture since it was decided that they all should follow the same structure. Lower in the folder hierarchy lies different franchises or store chains. This part of the hierarchy is optional and would only be used if multiple franchises were owned by one conglomerate and wanted to store their data on the same data lake. This was easily removed to the potential customers wishes. In figure 5.9 there are two franchises, one named Axisebra and the other was a name of a real store that was censored since it did not want to be shown in the image. Going further down in the hierarchy the different store locations would appear. So for example if a franchise named Fakestore has two stores where one is in city A and the other in city B there would be two folders here named Fakestore A and Fakestore B. After this its a standard hierarchy of year, month and finally day where each sensor would store one days worth of data in a file with the sensors name.

While this solution might not seem better than the previous one at first glance it becomes easier to navigate due to directory wildcards feature while using U-SQL to handle data in the Azure data lake. To use the same example as mentioned before with trying to get three different sensors data from the same day one simply makes a wildcard requesting all data from that folder and now one can easily specify what data you want from there. This feature also works on months and years which makes this solution way simpler computational wise. It also seems more graspable for a person if one wants to traverse the data on their own.

With both the structure and architecture in place one can argue that a lot has been done to prevent formation of a data swamp as well as providing easier access to different user groups while still keeping all the data in a raw format. But there are still aspects that will be taken into consideration such as if all data will be needed or not and also find ways to govern the data stored.

5.2.2 Data governance

While accessibility and ease of use are very important features for any type of storage solution one has to focus on not only how and where to store the data but also have to consider what data that should be stored.

Due to the very open and unrestricted nature of data lakes no initial data governance is in place. One can simply add whatever type of data to the lake and this data will remain in the data lake until someone manually decides to remove it. While the data redundancy and error removal feature discussed in section 5.2 cover a lot of space in regards to data governance there are other aspects to consider.
Data governance principles differ from company to company and therefore a lot of different strategies have been invented. But all of these strategies do to some degree try to solve the same issues [49]. During research for this thesis it was decided that the focus of the data governance should lie within redundancy and error removal which was achieved. It did occur, however that this would result in that some data would remain inside the data lake indefinitely. The relevance of data is to a very large degree decided by the ones that will utilize it and therefore a meeting was set up with Axis to determine if other aspects of data governance should be considered [49].

In this meeting it was determined that an additional feature should be added that could remove data after a certain time period. So if one deems that data would be unnecessary after three years this feature could be added and all data older that three years will be removed from the data lake. This was implemented with the Azure data factory feature that specifically scans the data lake and removes data satisfying a specific metric which in this case is older than a certain date.

While it is somewhat uncommon for a data lake to delete data after time since one of the key attractions of a data lake is to store raw data indefinitely. It was deemed to be convenient for the customer to have the option if desired.

5.2.3 Data utilization

Early on in the project Axis showed interest in pushing data from the data store to a third party API. So this was the obvious first choice when deciding what utilization the aggregated data should be used for.

Transform data and push to External API

By using the pipeline feature on the Azure platform one may push data saved in the data lake by using http post to third party API:s. This may then be combined with a U-SQL script that can extract data on one format and output a new.

The third API in this example requires data that is to be pushed to it to be in a specified format. The format is specified in A.3.3 and is a simple Json format. This format does not match the output of any of the sensor data provided for this thesis. Fortunately the Azure platform allows running of U-SQL scripts of files stored on the platform. With U-SQL one may query files for specified data and output the data on another format or with new or renamed objects. The data used in this case is from a people counter camera that outputs data on the format seen in A.3.1. By running the U-SQL script in A.2.1 on this file the output will be A.3.3.

By using the pipeline feature the process can be run on a schedule and by doing this there is no need to give any third party access to the storage as all data they require can be pushed to them.

Data can now also be directly visualized using power BI that can directly access the data stored in the transformed section in the data lake.

Chapter 6
Conclusion

This chapter will firstly cover the results of the analysis of message brokers and data lakes. The results are separated into two cases, the first describing the result where the metrics in section 2.5 are all given the same priority, and the second one the priority-list of section 2.6 are used to determine the best option in the use case. This is then repeated for the Data lakes using the criteria of section 2.7 instead. Secondly a discussion of the mentioned results and how they relate to the use case. Lastly a list of suggestions for further development of the prototype.

6.1 Project Result

The following subsections describes the result of the analysis for the given metrics.

6.1.1 Best message broker

Consequently reoccurring at the top spots of all the metrics is Apache Kafka, which is the main reason it was selected as the broker to be used when developing the first prototype. The closest contender being Apache Pulsar as it does outperform Kafka at Throughput, availability, and scalability. But Apache Pulsar is as of writing this thesis very inoperable as well as having limited usability. Both of these are however subject to change as both of these shortcomings are common for new software.

6.1.2 Best message broker for use-case

For the specified use-case of the thesis as can be seen in chapter 2 the metrics are given a priority(2.6). This list states that Security, Interopability and Scalability are of most importance. The most scalable of the brokers was shown to be Apache Pulsar after testing, whose Bookkeeper handled clustering which allows it to have higher scalability than Kafka. Pulsar also has a built in security feature making it the most secure broker by default. Due to its relative "newness" the interopability of the broker is more limited. The only other metric where Pulsar does not outperform the others is usability, where it scores lower for its lack of documentation and small community.

6.1.3 Best Data Lake for the use case

To make a full comparative analysis of which data lake is actually the most suitable for the use case, more criteria is required. But with the given criteria the most optimal selection is Microsoft's Azure Data Lake. Azure is by far the most interoperatable with

support for a long list of third-party apps as well having a lot of features that are open source. It is not as established as AWS but far more established than IBM:s data lake. AWS Lake formation is currently the most established of the data lake solutions but it falls short of being the best in this use case for its low interoptability and it being difficult to move the data from the platform.

6.2 Discussion

The age of NoSQL big data is clearly pushing a paradigm shift away from the familiar relational database. Solutions such as data lakes in the cloud are still quite new and definitely has its own set of flaws. So while there definitely are cases where data lakes can be seen as a massive improvement some aspects still make it seem less viable as a solution. One of these aspects are the speed that one can retrieve data from these databases. But when looking into many data lake solutions it is technically possible to achieve just as fast speeds as SQL databases if one takes the time to design it properly. So it seems that most developers just haven't been bothered to take the time to look at how to design data lakes but instead opt to stick with what they are used to. With increasing amounts of data and different types of data traditional databases seem less capable to cope however and many start to look at different alternatives to fix this either with extensions to traditional databases or solutions such as NoSQL databases or data lakes.

6.2.1 Results

When viewing the results on both message brokers and data lakes there is a lot that one can reflect on. If one looks at the message broker results its more or less impossible to crown a winner without taking a look on what's important for the use case and the developers. Most of these message brokers have widely different use cases and one cannot call them bad or worse than the other since they were not intended for the same end use. An example would be Redis which was ranked lowest on the usability ranking. Redis itself is built as a data caching system and only added message brokering as an extra feature. That the documentation is lacking and the community is small only refers to the message brokering part since Redis as a memory cache system is quite popular. The message broker itself also has some strong suites such as quite low latency and capability of sending large messages up to 512 megabytes. However this was not deemed useful for the use case and it can't therefore get a higher score.

An interesting addition one could have done to the analysis if time had permitted it would be possible to add more advantages and disadvantages in each metric to get a more nuanced view on what fields that each broker could be useful in, and not only how it relates to the use case. To mitigate this by having both a best broker and best broker for use case might to some degree show that not all message brokers are the best case for all scenarios. The issue here is that while the winner in the best message broker category was the broker that got the highest average amongst the metrics without any priority on what metric was more important than the other, it still compares with the metrics which were created for this use case. An example of other metrics that some might consider important is performance. This was initially tested and was supposed to be included as a metric but during testing most message brokers bottlenecked with their throughput before putting any major dent in processing power of the test rig.

While discussing this with Axis they mentioned that as long as no message broker showed horribly clear results of being inefficient on the testing rig one did not need to consider this too important and was therefore removed, since the thesis workers had nothing to add in the analysis except that none of the brokers where horribly inefficient. But if one would like to send data into weaker computers such as Raspberry pies one would have to consider performance and then Mosquitto would have won in that metric. But given the metrics that were defined the analysis seemed to be working well and the results where not that shocking from what was experienced during usage of the brokers.

When it comes to comparing data lakes things actually get very difficult. In difference to the message brokers where features differ from broker to broker making it easier to differentiate, it is more difficult for most enterprise data lake solutions to offer very similar features, performance and cost which forced the authors to rethink how to compare these solutions. When reading how others went about comparing data lakes most focused on how large part of the market they used. An example would be "52% of the small businesses use solution A for their data lakes therefore it must be the best" which was not an approach that the thesis workers felt actually argued for whether its better than the alternatives. Therefore a meeting was held with Axis to find some metrics that actually could be used with a better motivation than market share and those can be read in section 4. In that regard both thesis authors felt that the analysis on the data lakes was good but there are minor differences that might make one better than the other still, so an advised approach is still to look at the intended use case. Some argue that AWS is the best choice in general since it is the most mature solution and therefore had more time to iron out flaws. It also argued that Azure is the best if the intention is to use Microsoft software such as Excel or power BI. Some of these metrics where taken into account but were hardly a major basis for the analysis and it felt that they should not be part of the analysis since there was no real analysis made to come up with them but it was instead based on metrics from Axis.

Another aspect that can be considered is when to use enterprise solutions such as Azure or AWS versus using solutions such as Hadoop. Hadoop provides a lot more freedom in terms of what you can do with the data and where you want the data lake to be hosted. But since most things are not already made and is no longer running on enterprise hardware everything comes down to your skills as a developer and how much time one wants to spend on optimizing or making things work just as intended. Performance will be more of an issue as well since the hardware probably don't match up the the Amazon or Microsoft servers. But it can also be motivated that one can greatly affect the performance of a Hadoop solution by improving the code but in the enterprise solutions the best one can do is to write a ticket and hope that this will be resolved soon. There is also a question of costs, while the enterprise solutions have a direct cost in form of a bill the Hadoop solution still provides a cost with longer development time and also the building and maintenance of the server to host it on. So it would be interesting to make some sort of studies on when one solution would be more beneficial in comparison to the other from a purely financial perspective.

The discussion of whether to use an enterprise solution or a more self-developed solution such as Hadoop would have been an extremely interesting addition to the analytics if the time frame was larger for the thesis. Especially since Axis in the later stages expressed a desire to look into a data lake solution that can be keep locally in which Hadoop would have been a very interesting option to dive further into and explore.

6.2.2 General evaluation of thesis and planning

While making a general assessment on how the thesis went both authors where happy with the end result. Both thesis authors learned a lot both theoretically and practically about data lakes and message brokers. Something else that was also very rewarding was the new knowledge gained from working in large companies and the importance of teamwork. Since both thesis authors wanted more work experience this was seen as something very valuable and one will surely be able to apply a lot of the knowledge gained in future projects wherever those might be.

When it comes to the work done at Axis both thesis workers felt that a lot more work got done than what was initially estimated during the planning stage which was very satisfying to see. Those extra features came with slight quality loss for the analysis, however since focus shifted into finishing these features instead of starting the analytics in time. This is somewhat regrettable but can also be considered a good lesson in estimating the time each feature would take to develop so that one does not start a to big task when ahead of schedule, and will be had in mind in future projects. When on the subject of the planning of the project one can compare how the workflow was in comparison to the initial planning made just before starting the thesis work at Axis. In this regard the thesis workers initially performed well and got ahead of schedule during middle of sprint three. This lead to a decision to take on extra features for the demo which lead to a slight postponement of the analysis that was intended to start in the beginning of sprint four. The thesis workers also managed to hold weekly meetings with the supervisors on Axis even during the tumultuous times during the corona outbreak but then in a digital form. So in short the planning went very well and the issues mostly circle back to taking on extra features when slightly ahead of schedule without regarding how long these features would take to finish. One could argue that this might have been avoided if the sprints were cut down to shorter sprints which is a fair point and something to consider in the future. But the reality was that the sprint actually worked as phases and there was something called sub sprints which where weekly so while the terminology was different than normal Kanban the work methodology was similar.

When presenting the product to Axis they also seemed pleased with the result giving the impression that the end result could be considered a success despite having some minor bumps along the way. The thesis workers were also very pleased with the level of interaction that was received from Axis and the company also gave a lot of valuable feedback that really helped in some situations. The company also showed great interest in the progress of the thesis and many employees both from and outside the department came to ask what has been learned so far and requested meetings to follow up on how said work was going, and if they could help in any way. While both thesis members are happy with the end result there were a lot of extra features or opportunities that were found along the way that both thesis workers would love to be able to explore. Unfortunately time is a factor in every project and one cannot create everything that seems interesting or pop up along the way or the project would have never been finished. A list of all the additions that was considered along the way where written down however and the most interesting ones will be mentioned and discussed in this thesis in section 6.3.

6.3 Future Work

Both thesis workers have found that the work done at Axis has been very interesting to work with and both also find this project to be full of potential for future work, and as a

potential product that will at some point be used by Axis and their customers. During the process of research and development of the project many interesting potential additions to either research or development were uncovered but unfortunately most of them could not have been completed within the given time frame. Most of these ideas can be viewed as a bullet list in the appendix at A.5 but due to the amount it was deemed to only the most interesting ideas will be discussed in this chapter.

6.3.1 Hosting lake locally

There exists a general mistrust of storing data in the cloud for many companies, meaning that the suggested data lake solutions would be completely out of the question as all data is stored in the cloud[53]. This along with the fact that some companies might prefer the higher degree of freedom when developing solutions yourself.

The solution to this would be to offer these companies a locally hosted data lake instead of a cloud based one. This solution would require using the Hadoop HDFS for storing the data, and would require developing a data lake that operates on this storage.

6.3.2 Clean and store data

As previously mentioned there is a mistrust in storing data in the cloud, which may lead to customers not being comfortable with their data being stored in the cloud. For users with this preference the step of moving data from STG to RAW in the data lake can be changed to pushing the data to a database that the user finds more trustworthy. This removes the storing of the data in the cloud but handling the data is still done in the cloud. This solution would require no changes on the edge solution and only minor changes to the data lake.

6.3.3 Alternate Edge solutions

Discussed in chapter 5 are a number of alternate edge solutions i.e solutions to moving data between sensor and the cloud. They were all considered unsuitable for the given use case but in an alternate use case they may be a better option.

6.3.4 Linking data

This thesis focuses mainly on handling sensor data, but one of the many selling points is that they handle data of any format and the tools needed to combine such data exists within most data lake platforms. One might for example want to know how the number of customers relate to the the revenue of the store.

6.3.5 Raw data point and real time analytics in the cloud

In the current solution the raw data point that can be used for real time analytics lies locally alongside the message broker. While this is an acceptable solution it might be interesting to take a deeper look into if this is also possible to have the raw data point with real time analytics inside the data lake, in order to provide a fully cloud published solution. There might also be an interesting point to see if the real time analytics services provided by enterprise data lake solutions might make it easier for developers to get more use of their data than compared with local solutions.

6.3.6 Diving deeper into hybrid message broker solutions

While analyzing and comparing the message brokers focus was put on using just a single message broker that would try to solve all problems. This was partly in order to keep the solution more modular for Axis but also due to time constraints since the thesis workers already compared multiple brokers. One can argue however that a hybrid solution containing multiple message brokers could provide better results to Axis since one might get the benefits from multiple brokers instead of trying to find one solution that fits all.

6.3.7 Stream data

With the suggested solution data is uploaded to the data lake batches every 15 minutes, this was to emulate Axis current solution. There are many message brokers that supports streaming of data such as Kafka, NATS and Pulsar. To stream the data to the data lake was considered unnecessary for the use case as most of the sensor data isn't relevant in real-time. Streaming data is more relevant when handling big data[12][9][2], which was outside the scope of this thesis.

References

[1] Estrada, N., & Astudillo, H. (2015). Comparing scalability of message queue system: ZeroMQ vs RabbitMQ. 2015 Latin American Computing Conference (CLEI), 1.

[2] Mrozek, Tomasz Dąbek and Bozena Malysiak-Mrozek (2019) Scalable Extraction of Big Macromolecular Data in Azure Data Lake Environment, Molecules, 24(1), p. 179.

[3] Klein, A. F. et al. (2015) 'An experimental comparison of ActiveMQ and OpenMQ brokers in asynchronous cloud environment', 2015 Fifth International Conference on Digital Information Processing & Communications (ICDIPC), p. 24.

[4] Ionescu, V. M. (2015). The analysis of the performance of RabbitMQ and ActiveMQ. 2015 54th Annual Conference of the Society of Instrument & Control Engineers of Japan (SICE),p. 132.

[5] Qusay I. Sarhan, & Idrees S. Gawdan. (2017). Java Message Service Based Performance Comparison of Apache ActiveMQ and Apache Apollo Brokers. Science Journal of University of Zakho, 5(4), p. 307–312.

[6] Maatkamp, M., van Delden, M., & LeKhac, N. A. (2016). Unidirectional Secure Information Transfer via RabbitMQ.

[7] Rostanski, M., Grochla, K., & Seman, A. (2014). Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. 2014 Federated Conference on Computer Science and Information Systems, Computer Science and Information Systems (FedCSIS), 2014 Federated Conference On,p. 879–884.

[8] Dobbelaere, P., & Esmaili, K. S. (2017). Kafka versus RabbitMQ. Nokia Bell Labs

[9] Intorruk, S. and Numnonda, T. (2019) 'A Comparative Study on Performance and Resource Utilization of Real-time Distributed Messaging Systems for Big Data', 2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2019 20th IEEE/ACIS International Conference on, pp. 102–107.

[10] Le Noac'h, P., Costan, A., & Bouge, L. (2017). A performance evaluation of Apache Kafka in support of big data streaming applications. 2017 IEEE International Conference on Big Data (Big Data), Big Data (Big Data), 2017 IEEE International Conference On, 4803–4806.

[11] Hiraman, B. R., Viresh M., C., & Abhijeet C., K. (2018). A Study of Apache Kafka in Big Data Stream Processing. 2018 International Conference on Information , Communication, Engineering and Technology (ICICET), Information , Communication, Engineering and Technology (ICICET), 2018 International Conference On, 1–3.

[12] Tun, M. T., Nyaung, D. E., & Phyu, M. P. (2019). Performance Evaluation of Intrusion Detection Streaming Transactions Using Apache Kafka and Spark Streaming. 2019 International Conference on Advanced Information Technologies (ICAIT), Advanced Information Technologies (ICAIT), 2019 International Conference On, p. 25–30.

[13] Apache Pulsar Outperforms Apache Kafka by 2.5X. (2018). UNIX Update, 29(4), 2. Linux Foundation

[14] T, Sharvari. and K, Sowmya. (2019) 'A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming'. Available at: https://arxiv.org/ftp/arxiv/papers/1912/1912.03715 [Accessed 5 March 2020].

[15] What Is Amazon Kinesis Data Streams? - Amazon Kinesis Data Streams. (2020). Available at: https://docs.aws.amazon.com/streams/latest/dev/introduction [Accessed: 2 May 2020]

[16] Amazon Amazon Web Services, Inc. 2020. Amazon Kinesis - Process & Analyze Streaming Data - Amazon Web Services. [online] Available at: https://aws.amazon.com/kinesis/ [Accessed 5 March 2020].

[17] Amazon.com. 2020. Amazon Kinesis Data Streams Terminology And Concepts - Amazon Kinesis Data Streams. [online] Available at: https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html [Accessed 18 February 2020].

[18] Amazon Web Services. 2020. Building Scalable Applications And Microservices: Adding Messaging To Your Toolbox | Amazon Web Services. [online] Available at: https://aws.amazon.com/blogs/compute/building-scalable-applications-and-microservices-adding-messaging-to-your-toolbox/ [Accessed 12 March 2020].

[19] Amazon Web Services, Inc. (2020) What Is Pub/Sub Messaging?. [online] Available at: https://aws.amazon.com/pub-sub-messaging/ [Accessed 10 March 2020].

[20] Amazon Web Services, Inc. (2020) Amazon Kinesis Data Analytics Faqs - Analyze Streaming Data - Amazon Web Services (AWS). [online] Available at: https://aws.amazon.com/kinesis/data-analytics/faqs [Accessed 24 February 2020].

[21] Apache Kafka. 2020. Apache Kafka. [online] Available at: https://kafka.apache.org/documentation [Accessed 20 January 2020].

[22] Mqtt.org. (2020) Documentation | MQTT. [online] Available at: http://mqtt.org/documentation [Accessed 23 February 2020].

[23] Team, T., (2020) Last Will And Testament - MQTT Essentials: Part 9. [online] Hivemq.com. Available at: https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament [Accessed 25 February 2020].

[24] 2019. OASIS Standard. 2nd ed. [ebook] OASIS. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf [Accessed 18 March 2020].

[25] Eclipse Mosquitto. (2020) Eclipse Mosquitto. [online] Available at: https://mosquitto.org [Accessed 10 March 2020].

[26] Hwang, K. et al. (2019) 'Modification of Mosquitto Broker for Delivery of Urgent MQTT Message', 2019 IEEE Eurasia Conference on IOT, Communication and Engineering (ECICE), IOT, Communication and Engineering (ECICE), 2019 IEEE Eurasia Conference on, pp. 166–167. doi: 10.1109/ECICE47484.2019.8942800.

[27] eclipse.org. 2020. Eclipse Mosquitto. [online] Available at: https://projects.eclipse.org/projects/iot.mosquitto [Accessed 4 March 2020].

[28] Mosquitto.org. (2020) Mosquitto Library Reference. [online] Available at: https://mosquitto.org/api/files/mosquitto-h.html [Accessed 10 March 2020].

[29] Guindon, C., 2020. Mosquitto | The Eclipse Foundation. [online] Eclipse.org. Available at: https://www.eclipse.org/proposals/technology.mosquitto [Accessed 2 March 2020].

[30] Company, O., (2020) What Is Vernemq. [online] Vernemq.com. Available at: https://vernemq.com/intro/ [Accessed 17 February 2020].

[31] Vernemq.com. (2020) Vernemq Features. [online] Available at: https://vernemq.com/intro/features [Accessed 29 January 2020].

[32] Hivemq.com. (2020) See The Features Of Hivemq - The MQTT Broker. [online] Available at: https://www.hivemq.com/hivemq/features [Accessed 18 February 2020].

[33] Team, T., (2020) Discover The 3 Different Editions Of Hivemq. [online] Hivemq.com. Available at: https://www.hivemq.com/hivemq/editions [Accessed 25 February 2020].

[34] Redis.io. 2020. An Introduction To Redis Data Types And Abstractions – Redis. [online] Available at: https://redis.io/topics/data-types-intro [Accessed 1 March 2020].

[35] Redis.io. (2020) Redis. [online] Available at: https://redis.io/documentation [Accessed 14 February 2020].

[36] Nelson, J., 2016. Mastering Redis. 1st ed. Birmingham, UK: Packt Publishing.

[37] Amazon Web Services, Inc. (2020) Redis: In-Memory Data Store. How It Works And Why You Should Use It. [online] Available at: https://aws.amazon.com/redis [Accessed 6 March 2020].

[38] Fisher, J., (2020) Redis Pub/Sub Under The Hood. [online] Making Pusher. Available at: https://making.pusher.com/redis-pubsub-under-the-hood [Accessed 19 March 2020].

[39] Redis.io. (2020) Pub/Sub – Redis. [online] Available at: https://redis.io/topics/pubsub [Accessed 8 March 2020].

[40] Google Cloud. (2020) Cloud Pub/Sub | Google Cloud. [online] Available at: https://cloud.google.com/pubsub [Accessed 6 April 2020].

[41] Google Cloud. (2020) What Is Pub/Sub? | Cloud Pub/Sub Documentation | Google Cloud. [online] Available at: https://cloud.google.com/pubsub/docs/overview [Accessed 4 February 2020].

[42] google.com. (2020) G Suite Terms Of Service – G Suite. [online] Available at: https://gsuite.google.com/terms/sla.html [Accessed 8 March 2020].

[43] Google Cloud. (2020) Subscriber Overview | Cloud Pub/Sub | Google Cloud. [online] Available at: https://cloud.google.com/pubsub/docs/subscriber [Accessed 8 March 2020].

[44] Zguide.zeromq.org. (2020) ØMQ - The Guide - ØMQ - The Guide. [online] Available at: http://zguide.ZeroMQ.org/page:all [Accessed 20 February 2020].

[45] Rfc.zeromq.org. (2020) 23/ZMTP. [online] Available at: https://rfc.ZeroMQ.org/spec/23/ [Accessed 10 March 2020].

[46] 250bpm.com. (2020) Scalability Layer Hits The Internet Stack - 250Bpm. [online] Available at: http://250bpm.com/hits#toc5 [Accessed 7 March 2020].

[47] Hintjens, P. (2013). ZeroMQ: Messaging for Many Applications. 1st ed. O'Reilly Media, Inc.

[48] Tejada, Z. (2017). Mastering Azure analytics. 1st ed. O'Reilly Media inc.

[49] Smallwood, R., 2019. Information Governance: Concepts, Strategies And Best Practices. 2nd ed. New Jersey: John Wiley & Sons.

[50] Evisional.com. (2020) Comparison Of Azure Datalake Gen-2 And Aws S3. [online] Available at: http://evisional.com/comparison-of-azure-datalake-gen-2-and-aws-s3/ [Accessed 17 January 2020].

[51] Levy, E., (2020) Cloud Data Lake Vs On-Premises Data Lake: What You Need To Know. [online] Upsolver.com. Available at: https://www.upsolver.com/blog/cloud-data-lake-vs-on-premises-data-lake [Accessed 7 March 2020].

[52] Inside Out Security. (2020) AWS Vs Azure Vs Google: Cloud Services Comparison - Varonis. [online] Available at: https://www.varonis.com/blog/aws-vs-azure-vs-google/ [Accessed 10 April 2020].

[53] HALL, P. (2020) 'Is cloud computing really safe?', Fairfield County Business Journal, 56(7), p. 15. Available at: https://search-ebscohost-com.ludwig.lub.lu.se/login.aspx?direct=true&db=bwh&AN=141798411&site=eds-live&scope=site (Accessed: 8 May 2020).

[54] Amazon Web Services, Inc. (2020) Open Source At Amazon. [online] Available at: https://aws.amazon.com/opensource/ [Accessed 15 April 2020].

[55] Mandelbaum, R., 2018. Is Amazon Good Or Bad For Small Business?. [online] Forbes. Available at: https://www.forbes.com/sites/robbmandelbaum/2018/03/31/is-amazon-good-or-bad-for-small-business-yes/#450559104467 [Accessed 29 March 2020].

[56] Masters, K., 2019. These Four Companies Still Refuse To Sell On Amazon, Despite Its Market Dominance. [online] Forbes. Available at: https://www.forbes.com/sites/kirimasters/2019/09/05/these-four-companies-still-refuse-to-sell-on-amazon-despite-its-market-dominance/#258f11bc24fe [Accessed 1 May 2020].

[57] Yongqiang Huang and Garcia-Molina, H. (2001) 'Exactly-once semantics in a replicated messaging system', Proceedings 17th International Conference on Data Engineering, Data Engineering, 2001. Proceedings. 17th International Conference on, Data engineering, pp. 3–12. doi: 10.1109/ICDE.2001.914808.

[58] Google Cloud. 2020. Dataflow Google Cloud. [online] Available at: https://cloud.google.com/dataflow#section-5 [Accessed 2 May 2020].

[59] Gunarathne, T. et al. (2010) 'MapReduce in the Clouds for Science', 2010 IEEE Second International Conference on Cloud Computing Technology and Science, Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, pp. 565–572. doi: 10.1109/CloudCom.2010.107.

[60] W. W. Eckerson (2018) Data Lakes for Business Users, Arcadia Data, Available at: https://www.arcadiadata.com/lp/eckerson-report-data-lakes-for-business-users/ (Accessed: 5 May 2020).

[61] Tarigan, Iyoga & Kim, Dong-Seong. (2016). Comparative Performance AMQP and MQTT Protocol over Wired Network, Available at: https://doi.org/10.13140/RG.2.1.2089.8802 (Accessed: 5 May 2020).

[62] ScalAgent (2015) Benchmark of MQTT servers, Available at: http://www.scalagent.com/Benchmark_MQTT_servers-v1-1/ (Accessed: 7 March 2020).

[63] Renart, E. G., Balouek-Thomert, D. and Parashar, M. (2018) 'Edge Based Data-Driven Pipelines (Technical Report)'. Available at: https://arxiv.org/abs/1808.01353 (Accessed: 5 May 2020).

[64] Axis.com (2020) User Instructions | Axis Communications. [online] Available at: https://www.axis.com/sv-se/products/online-manual/37930 [Accessed 8 May 2020].

[65] ActiveMQ 5 Documentation (2020) ActiveMQ. [online] Available at: https://activemq.apache.org/components/classic/documentation [Accessed 26 May 2020]

[66] Pulsar Docs (2020) Apache Software Foundation. [online] Available at: https://pulsar.apache.org/docs/ [Accessed 26 May 2020].

[67] Documentation: RabbitMQ (2020) Rabbitmq.com. [online] Available at: https://www.rabbitmq.com/documentation.html [Accessed 26 May 2020].

[68] NATS Documentation (2020) Apache Software Foundation. [online] Available at: https://docs.nats.io/ [Accessed 26 May 2020].

[69] Azure Documentation (2020) Docs.microsoft.com [online] Available at: https://docs.microsoft.com/en-us/azure/ [Accessed 26 June 2020].

[70] Azure Data Lake Storage Gen1 Documentation (2020) Docs.microsoft.com. [online] Available at: https://docs.microsoft.com/en-us/azure/data-lake-store/ [Accessed 26 May 2020].

[71] Amazon Lake Formation Documentation (2020) Docs.aws.amazon.com. [online] Available at: https://docs.aws.amazon.com/cloudformation/ [Accessed 26 May 2020].

[72] Hadoop Documentation (2020) Hadoop.apache.org. [online] Available at: https://hadoop.apache.org/docs/r1.2.1/ [Accessed 26 May 2020].

[73] Google Cloud Platform | Documentation (2020) Google Cloud. [online] Available at: https://cloud.google.com/docs [Accessed 26 May 2020].

[74] IBM Cloud Docs (2020) Cloud.ibm.com. [online] Available at: https://cloud.ibm.com/docs? [Accessed 26 May 2020].

Appendix A

Some extra material

A.1 Message broker

A.1.1 Pulsar Producer

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Map;
import java.util.stream.IntStream;
import org.apache.pulsar.client.api.*;

public class encryptProducer {
private static final String SERVICE_URL = "pulsar://localhost:6650";
private static final String TOPIC_NAME = "test";



public static void main(String[] args) throws PulsarClientException {
// TODO Auto-generated method stub
RawFileKeyReader rawRead = new RawFileKeyReader("rsa_pubkey.pem",
privkey.pem");
PulsarClient client = PulsarClient.builder()
.serviceUrl(SERVICE_URL)
.build();


Producer<byte[]> producer = client.newProducer()
.topic(TOPIC_NAME)
.cryptoKeyReader(rawRead)
.addEncryptionKey("mykey")
.create();
producer.send("thisIssecret".getBytes());
System.out.println("sent message");
producer.close();
client.close();


}
```

```
}
```

A.1.2 Pulsar Consumer

```
import java.util.stream.IntStream;

import org.apache.pulsar.client.api.*;
public class subscriberencrypted {
private static final String SERVICE_URL = "pulsar://localhost:6650";
private static final String TOPIC_NAME = "test";
public static void main(String[] args) throws PulsarClientException {
// TODO Auto-generated method stub
PulsarClient client = PulsarClient.builder().serviceUrl(SERVICE_URL).build();

Consumer consumer = client.newConsumer()
.topic(TOPIC_NAME)
.cryptoKeyReader(new RawFileKeyReader("rsa_pubkey.pem",
"rsa_privkey.pem"))
.subscriptionName("my-subscribtion")
.subscribe();


while (true){
Message msg = consumer.receive();
System.out.println(new String(msg.getData()));
consumer.acknowledge(msg);

}
}

}
```

A.1.3 Kafka Producer

```
    import java.util.Properties;
import java.util.concurrent.TimeUnit;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
//simple producer

public class kafkaProducer {

public static void main(String[] args) throws InterruptedException {
// TODO Auto-generated method stub
Properties props=new Properties();
//add properties
props.put("bootstrap.servers", "10.6.3.98:9092");
```

```
props.put(
"key.serializer","org.apache.kafka.common.serialization.StringSerializer"
);
props.put(
"value.serializer","org.apache.kafka.common.serialization.StringSerializer"
);
//send 500 messages
KafkaProducer<String,String> producNew =
new KafkaProducer<String,String>(props);
for(int i=0; i<5000;i++){
ProducerRecord<String,String> record =
new ProducerRecord<String,String>("test","sample key",Integer.toString(i));
System.out.println(record.value());
producNew.send(record);
}

producNew.close();
}

}
```

A.1.4 Kafka Consumer

```
import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

public class kafkaConsumer {

public static void main(String[] args) {

Properties props = new Properties();
props.put("bootstrap.servers", "10.6.3.98:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms","1000");
props.put(
"key.deserializer","org.apache.kafka.common.serialization.StringDeserializer"
);
props.put(
"value.deserializer","org.apache.kafka.common.serialization.StringDeserializer"
);
KafkaConsumer<String,String > consumer =
new KafkaConsumer<String,String > (props);
consumer.listTopics();
        //consumes a list of topics
consumer.subscribe(Arrays.asList("test"));
```

```
for(int i =0;i<1000;i++){
ConsumerRecords<String,String> records = consumer.poll(Duration.ofMillis(100));

for(ConsumerRecord<String,String> record : records)
// Skriver ut hela record med samtliga tags
//System.out.println(record.toString());
// Skriver endast ut value

System.out.println(record.value().toString());
}


}

}
```

A.1.5 RabbitMQ Producer

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class produceTopic {

  private static final String EXCHANGE_NAME = "topic_logs";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    try (Connection connection = factory.newConnection();
         Channel channel = connection.createChannel()) {

        channel.exchangeDeclare(EXCHANGE_NAME, "topic");

        String routingKey = "kern.critical";
        String message = "SWIZZZZ3";

        channel.basicPublish(EXCHANGE_NAME, routingKey, null,
        message.getBytes("UTF-8"));
        System.out.println(" [x] Sent '" + routingKey + "':'" + message + "'");
    }
  }

}
```

A.1.6 RabbitMQ Consumer

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
```

```
public class consumerTopics {

  private static final String EXCHANGE_NAME = "topic_logs";

  public static void main(String[] argv) throws Exception {
    String[] argv2 = { "kern.*", "*.critical" };

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "topic");
    String queueName = channel.queueDeclare().getQueue();

    if (argv2.length < 1) {
      System.err.println("Usage: ReceiveLogsTopic [binding_key]...");
      System.exit(1);
    }

    for (String bindingKey : argv2) {
      channel.queueBind(queueName, EXCHANGE_NAME, bindingKey);
    }

    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
      String message = new String(delivery.getBody(), "UTF-8");
      System.out.println(" [x] Received '" +
      delivery.getEnvelope().getRoutingKey() + "':'" + message + "'");
    };
    channel.basicConsume(queueName, true, deliverCallback, consumerTag -> {
    });
  }
}
```

A.1.7 MQTT Producer

```
import java.nio.charset.StandardCharsets;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

public class producer {

public static void main(String [ ] args)
{
try {
```

```
MqttClient client = new MqttClient("tcp://localhost:9092",
MqttClient.generateClientId(),new MemoryPersistence());
    if(!client.isConnected()) {
     client.connect();
    }
    MqttMessage message = new MqttMessage("testade".getBytes());
    //sets qos to 2(exactly once)
    message.setQos(2);
client.publish("testExjobb", message);


} catch (MqttException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}

}


}
```

A.1.8 MQTT Consumer

```
    import java.util.Arrays;

import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

public class consumer {

public static void main(String [ ] args)
{

MqttClient client;
try {
client = new MqttClient("tcp://localhost:9092",MqttClient.generateClientId(),
new MemoryPersistence());
client.setCallback(new MqttCallback() {

    @Override
    public void connectionLost(Throwable cause) {
    //Called when the client lost the connection to the broker
    }

@Override
public void deliveryComplete(IMqttDeliveryToken arg0) {
// TODO Auto-generated method stub
```

```
}

@Override
public void messageArrived(String arg0, MqttMessage arg1) throws Exception {
// TODO Auto-generated method stub
System.out.println(arg0 + ": " + arg1.toString());
}


});

client.connect();
client.subscribe("testExjobb", 1);


} catch (MqttException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}



}



}
```

A.1.9 NATS Producer

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.time.Duration;

import io.nats.client.Connection;
import io.nats.client.Nats;

//Example code for a publisher in NATS using java
public class NatsPub {




    public static void main(String args[]) throws IOException {
        String subject;
        String message;
```

```
        String server;

// server defines what ip adress the NATS server resides
            server = "localhost:1883";
            // This is the subject so that subscribers are
            //capable to find the certain instace
            //similar to topics in Kafka or events in azure
            subject = "Tester";
            // The  text message
            BufferedReader reader =
                    new BufferedReader(new InputStreamReader(System.in));
            message = reader.readLine();


        try {
            Connection nc =
            Nats.connect(ExampleUtils.createExampleOptions(server, false));

            System.out.println();
            System.out.printf("Sending %s on %s, server is %s\n",
            message, subject, server);
            System.out.println();
            nc.publish(subject, message.getBytes(StandardCharsets.UTF_8));
            nc.flush(Duration.ofSeconds(5));
            nc.close();

        } catch (Exception exp) {
            exp.printStackTrace();
        }
    }


}
```

A.1.10 NATS Consumer

```
import java.nio.charset.StandardCharsets;
import java.time.Duration;

import io.nats.client.Connection;
import io.nats.client.Message;
import io.nats.client.Nats;
import io.nats.client.Subscription;

//Example code for a subscriber in NATS using java

public class NatsSub {



    public static void main(String args[]) {
```

```
        String subject;
        int msgCount;
        String server;
        String RecvMessage;
            server = "localhost:1883";
            subject = "Tester";
            msgCount = 1;


        try {
            System.out.println();
            System.out.printf("Trying to connect to %s, and listen to
            %s for %d messages.\n", server, subject, msgCount);
            System.out.println();

            Connection nc =
            Nats.connect(ExampleUtils.createExampleOptions(server, true));
            Subscription sub = nc.subscribe(subject);
            nc.flush(Duration.ofSeconds(5));

            for(int i=0;i<msgCount;i++) {
                Message msg = sub.nextMessage(Duration.ofHours(1));
                RecvMessage = new String(msg.getData(), StandardCharsets.UTF_8);
                System.out.printf("Received message \"%s\" on subject \"%s\"\n",
                                        RecvMessage,
                                        msg.getSubject());
            }

            nc.close();

        } catch (Exception exp) {
            exp.printStackTrace();
        }
    }


}
```

A.1.11 Redis Producer

```
    import java.io.BufferedReader;
import java.io.InputStreamReader;

import redis.clients.jedis.Jedis;

public class redisPublisher {

  public static void main(String[] args) {

    Jedis jedis = null;
    String redisPort = System.getenv().getOrDefault("REDIS_PORT", "6379");
```

```
    String redisHost = System.getenv().getOrDefault("REDIS_HOST", "localhost");
    try {
        /* Creating Jedis object for connecting with redis server */
        jedis = new Jedis();


        System.out.println("Write message you want to send \n");
        BufferedReader reader =
                new BufferedReader(new InputStreamReader(System.in));
        String message = reader.readLine();
        /* Publishing message to channel C1 */
        System.out.println("Sending "+message);
        jedis.publish("C1", message);

        /* Publishing message to channel C2 */
  //      jedis.publish("C2", "First message to channel C2");

        /* Publishing message to channel C1 */
 //       jedis.publish("C1", "Second message to channel C1");

        /* Publishing message to channel C2 */
 //       jedis.publish("C2", "Second message to channel C2");

    } catch(Exception ex) {

        System.out.println("Exception : " + ex.getMessage());
    } finally {

        if(jedis != null) {
          jedis.close();
        }
    }
  }
}
```

A.1.12 Redis Consumer

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPubSub;

public class redisSubscriber {

  public static void main(String[] args) {

    Jedis jedis = null;

    try {

    /* Creating Jedis object for connecting with redis server */
     // man kan skriva jedis("ipadress",port); för att specifiera port
    jedis = new Jedis();
```

```
    /* Creating JedisPubSub object for subscribing with channels */
    JedisPubSub jedisPubSub = new JedisPubSub() {

        @Override
        public void onMessage(String channel, String message) {
            System.out.println("Channel " + channel + " has sent a message :
            " + message );
            if(channel.equals("C1")) {
                /* Unsubscribe from channel C1 after first message is received. */
                unsubscribe(channel);
            }
        }

        @Override
        public void onSubscribe(String channel, int subscribedChannels) {
            System.out.println("Client is Subscribed to channel : "+ channel);
            System.out.println("Client is Subscribed to "+ subscribedChannels
            + " no. of channels");
        }

        @Override
        public void onUnsubscribe(String channel, int subscribedChannels) {
            System.out.println("Client is Unsubscribed from channel : "+ channel);
            System.out.println("Client is Subscribed to "+ subscribedChannels
            + " no. of channels");
        }

    };

    /* Subscribing to channel C1 and C2 */
    //jedis.subscribe(jedisPubSub, "C1", "C2");
    jedis.subscribe(jedisPubSub, "C1");
    } catch(Exception ex) {

        System.out.println("Exception : " + ex.getMessage());

    } finally {

        if(jedis != null) {
            jedis.close();
        }
    }
  }
}
```

A.1.13 ActiveMQ Producer

```
import java.net.URI;
import java.net.URISyntaxException;
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.broker.BrokerFactory;
import org.apache.activemq.broker.BrokerService;

public class ActiveMqConsumer {
    public static void main(String[] args) throws URISyntaxException, Exception {
        BrokerService broker = BrokerFactory.createBroker(new URI(
                "broker:(tcp://localhost:8081)"));
        broker.start();
        Connection connection = null;
        try {
            // Creates connection to server
            ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
                    "tcp://localhost:9092");
            connection = connectionFactory.createConnection();
            connection.setClientID("DurabilityTest");
            Session session = connection.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            Topic topic = session.createTopic("exjobbTopic");

            // Create publisher and send the message "Message!"
            String payload = "Message!";
            TextMessage msg = session.createTextMessage(payload);
            MessageProducer publisher = session.createProducer(topic);
            System.out.println("Sending text '" + payload + "'");
            publisher.send(msg, javax.jms.DeliveryMode.PERSISTENT,
            javax.jms.Message.DEFAULT_PRIORITY, Message.DEFAULT_TIME_TO_LIVE);
            connection.start();
            session.close();
        } finally {
            if (connection != null) {
                connection.close();
            }
            broker.stop();
        }
    }
}
```

A.1.14 ActiveMQ Consumer

```
import java.net.URI;
import java.net.URISyntaxException;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.broker.BrokerFactory;
import org.apache.activemq.broker.BrokerService;

public class ActiveMqConsumer {
    public static void main(String[] args) throws URISyntaxException, Exception {
        BrokerService broker = BrokerFactory.createBroker(new URI(
                "broker:(tcp://localhost:8081)"));
        broker.start();
        Connection connection = null;
        try {
            // Creates connection to server
            ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
                    "tcp://localhost:9092");
            connection = connectionFactory.createConnection();
            connection.setClientID("DurabilityTest");
            Session session = connection.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            Topic topic = session.createTopic("exjobbTopic");

            //Create a consumer that will consume everything from the topic
            MessageConsumer consumer1 =
            session.createDurableSubscriber(topic, "consumer1", "", false);

            connection.start();

            msg = (TextMessage) consumer1.receive();
            System.out.println("Consumer1 receives " + msg.getText());


            session.close();
        } finally {
            if (connection != null) {
                connection.close();
            }
            broker.stop();
        }
```

```
    }
}
```

A.2 code

A.2.1 u-Sql scripts

Format output to ZAPI

```
// A. REFERENCE ASSEMBLY: Load assemblies for compile time and execution.
REFERENCE ASSEMBLY [Newtonsoft.Json];
REFERENCE ASSEMBLY [Microsoft.Analytics.Samples.Formats];

// B. USING: Specify namespace to shorten function names
(e.g. Microsoft.Analytics.Samples.Formats.Json.JsonExtractor)
USING Microsoft.Analytics.Samples.Formats.Json;

// 1. Initialise variables for Input (e.g. JSON) and Output (e.g. CSV).
DECLARE @InputFile string = @"input.json";
DECLARE @OutputFile string = @"output.json";

// 2. Extract string content from JSON document (i.e. schema on read).
@json =
EXTRACT
    timestamp string,
serial string,
name string,
input int,
output int
FROM
    @InputFile
USING new JsonExtractor();

@datas =
SELECT
"540" AS organisation_id,
serial AS counter_id,
"in" AS visit_type,

timestamp.Substring(0, 4)+"-"+timestamp.Substring(4, 2)+
"-"+timestamp.Substring(6, 2) AS date,

timestamp.Substring(8, 2)+":"+timestamp.Substring(10, 2)+
":"+timestamp.Substring(12,2) AS time,
input AS visitors
FROM @json;


// 3. Write values to json
OUTPUT @datas
```

```
TO @OutputFile
USING new JsonOutputter();
```

Merge POS and queue data with U-SQL

```
// A. CREATE ASSEMBLY: Register assemblies (if they do not already exist).
CREATE ASSEMBLY IF NOT EXISTS [Newtonsoft.Json] FROM
@"codeLocation";
CREATE ASSEMBLY IF NOT EXISTS [Microsoft.Analytics.Samples.Formats] FROM
@"codeLocation";

// B. REFERENCE ASSEMBLY: Load assemblies for compile time and execution.
REFERENCE ASSEMBLY [Newtonsoft.Json];
REFERENCE ASSEMBLY [Microsoft.Analytics.Samples.Formats];

// C. USING: Specify namespace to shorten function names
//(e.g. Microsoft.Analytics.Samples.Formats.Json.JsonExtractor)
USING Microsoft.Analytics.Samples.Formats.Json;

// 1. Initialise variables for Input (e.g. JSON) and Output (e.g. CSV).
DECLARE @InputFile string = @"posdata.txt";
DECLARE @InputFile2 string = @"queuedata.json";
DECLARE @OutputFile string = @"outputfile.csv";

// 2. Extract string content from JSON document (i.e. schema on read).
@json =
EXTRACT
    timestamp string,
serial string,
name string,
inpep int,
outpep int
FROM
    @InputFile2
USING new JsonExtractor();

@data =
    EXTRACT
        SALES_DATE string,
SALES_TIME string,
ORGANISATION_ID string,
SALES_VALUE string,
SALES_VALUE_WO_VAT string,
NUMBER_OF_TRANSACTIONS string,
timestamp string
    FROM @InputFile
    USING Extractors.Text(delimiter: '|',skipFirstNRows: 0,quoting: false);

@new =
SELECT sales.SALES_VALUE_WO_VAT,sales.SALES_DATE,sales.NUMBER_OF_TRANSACTIONS,
sales.SALES_TIME,sales.ORGANISATION_ID, count.outpep
FROM @data AS sales
```

```
JOIN
@json AS count
ON sales.timestamp==count.timestamp;

@new2 =
SELECT * FROM @new ORDER BY ORGANISATION_ID ASC OFFSET 0 ROWS;

OUTPUT @new2
TO @OutputFile
USING Outputters.Csv(quoting: false);
```

A.2.2 Upload to Azure

```
import java.time.Duration;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.regex.Pattern;


import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import com.microsoft.azure.datalake.store.ADLException;
import com.microsoft.azure.datalake.store.ADLStoreClient;
import com.microsoft.azure.datalake.store.DirectoryEntry;
import com.microsoft.azure.datalake.store.IfExists;
import com.microsoft.azure.datalake.store.oauth2.AccessTokenProvider;
import com.microsoft.azure.datalake.store.oauth2.ClientCredsTokenProvider;

public class azureUploader {
private static String clientId = "";
private static String authTokenEndpoint = "";
private static String clientKey = "";
// full account FQDN, not just the account name
private static String accountFQDN = "";
private static final String SERVICE_URL = "pulsar://localhost:6650";
 // only needed if subscribed to specific topics
private static final String TOPIC_NAME = "test";

/**
 * @param args
 * @throws IOException
 */
public static void main(String[] args) throws IOException {
// setting up azure stuff
```

```
AccessTokenProvider provider = new ClientCredsTokenProvider(authTokenEndpoint,
clientId, clientKey);
ADLStoreClient client = ADLStoreClient.createClient(accountFQDN, provider);
OutputStream stream = null;

while (true) {

Message msg = "incomming message";
// The string vector removes the first characters
//and then splits the topic on backslashes.
String[] parts = msg.getTopicName().substring(28).split("\\\\");
// System.out.println(new String(msg.getData()));


String dire = parts[0]+"/"+parts[0]+ "-" + parts[1] +"/" +
parts[2].substring(0,4) + "/" +
            parts[2].substring(4,8) + "/" + parts[3]+".json";


if(!client.checkExists(dire)) {
client.createEmptyFile(dire);
client.setPermission(dire, "744");
System.out.println("File created");
}else {
System.out.println("File exists already");
}
stream = client.getAppendStream(dire);

stream.write(new String(msg.getData()).getBytes());
stream.close();
System.out.println("Send done!");
}


}

}
```

A.3 File format examples

A.3.1 Json file for counter

```
[
{
  "serial":"00408CAC512B",
  "name":"CounterName",
  "timestamp":"20170503112756",
  "in":12,
  "out":318
```

```
},
{
  "serial":"00408CAC512B",
  "name":"CounterName",
  "timestamp":"20170503112759",
  "in":15,
  "out":322
},
{
  "serial":"00408CAC512B",
  "name":"CounterName",
  "timestamp":"20170503112762",
  "in":20,
  "out":330
}
]
```


A.3.2 Json file for queue monitor


```
[
   {
      "serial":"ACCC8E20F09B",
      "name":"Service Counter",
      "timestamp":"20170113181132",
      "region1name":"Product Display",
      "region1people":1,
      "region2name":"Aisle Display",
      "region2people":0,
      "region3name":"Service Counter",
      "region3people":3
   },
   {
      "serial":"ACCC8E20F09C",
      "name":"Service Counter2",
      "timestamp":"20170113181142",
      "region1name":"Product Display2",
      "region1people":1,
      "region2name":"Aisle Display2",
      "region2people":0,
      "region3name":"Service Counter2",
      "region3people":3
   }
]
```


A.3.3 Json file remade for third party


```
[
   {
      "organisation_id":"540",
      "counter_id":"ACCC8E02EAB1",
      "visit_type":"in",
```

```
      "date":"2020-02-10",
      "time":"08:37:42",
      "visitors":12
   },
   {
      "organisation_id":"540",
      "counter_id":"ACCC8E02EAB1",
      "visit_type":"in",
      "date":"2020-02-10",
      "time":"08:37:44",
      "visitors":12
   }
]
```

A.4 output

A.4.1 RawFileKeyReader

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Map;

import org.apache.pulsar.client.api.CryptoKeyReader;
import org.apache.pulsar.client.api.EncryptionKeyInfo;

public class RawFileKeyReader implements CryptoKeyReader {

String publicKeyFile = "";
String privateKeyFile = "";

RawFileKeyReader(String pubKeyFile, String privKeyFile) {
publicKeyFile = pubKeyFile;
privateKeyFile = privKeyFile;
}

@Override
public EncryptionKeyInfo getPrivateKey(String keyName, Map<String, String> keyMeta)
{
EncryptionKeyInfo keyInfo = new EncryptionKeyInfo();
try {
  keyInfo.setKey(Files.readAllBytes(Paths.get(privateKeyFile)));
 } catch (IOException e) {
   System.out.println("ERROR: Failed to read private key from file " +
   privateKeyFile);
   e.printStackTrace();
 }
   return keyInfo;
}

@Override
```

```
public EncryptionKeyInfo getPublicKey(String keyName, Map<String, String> keyMeta)
{
 EncryptionKeyInfo keyInfo = new EncryptionKeyInfo();

 try {
   keyInfo.setKey(Files.readAllBytes(Paths.get(publicKeyFile)));
        } catch (IOException e) {
            System.out.println("ERROR: Failed to read public key from file " +
            publicKeyFile);
            e.printStackTrace();
        }
        return keyInfo;
}


     }
```

A.5 Future Work bullet list

This list show a quick bullet list of potential future work that could be interesting to work on. It is categorized into three different categories for each field it belongs to. There are more utilization than this but these might be non essential but may be discussed with the authors if someone finds this interesting.

Raw

- Send screen-grab when queue is full
- Share analysis from raw and data lake
- Linechart queue monitor
- move raw point to lake
- average time in queue

Lake

- Compare sales data with amount of customers
- Weather and temperature with amount of sales
- People demographics in store
- Average time spent in store
- Amount of people in store during promotions
- Time of day where queues are the longest
- Most relevant age group
- Find fastest moving queue
- Amount of people that have stood in a queue
- Better integration with data warehouses

Data that is not available at the time but could be interesting

- License plate
- Clearance or special sale data
- Air Quality

Appendix B
Licenses

Figures 3.1 and 3.2 are covered by the Apache licence 2.0 which allows use of illustrations in noncommercial work as long as a link to licence can be provided. The licence can be viewed here : `https://www.apache.org/licenses/LICENSE-2.0`

Figure 1 and 2 in section 3.11 follow the Creative commons 3.0 licence the licence can be found here: `https://creativecommons.org/licenses/by/3.0/`

All other illustrations, tables or images included in this thesis have been made by the thesis authors specifically to be included in the thesis and require no extra licence.
Figures 3.6,3.8 and 3.9 follow the Creative commons 4.0 licence the licence can be found here: `https://creativecommons.org/licenses/by/4.0/`