AARHUS UNIVERSITY

MASTER'S THESIS

HarakaMQ

Reliable and Lightweight Message Oriented Middleware Based on UDP

Author:
Peter Kragelund
René Rotvig Jensen

Supervisor:
Christian Fischer Pedersen

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

at the

Department of Engineering

3 January, 2018

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Preface

First the authors of this thesis would like to thank Christian Fischer Pedersen, for his guidance and supervision on this thesis, as well as for his constant encouragement.

Additional thanks to Danske Bank, MobilePay, Martin Castberg Thuesen and Rune Birkemose Jakobsen for the facility, and all the resources which has been made available.

At last thanks to family and friends for their constant support, curiosity and understanding during this work.

Peter Kragelund
Stud. no.: 201270868 - Peter Kragelund

René Rotvig Jensen
Stud. no.: 201270762 - René Rotvig Jensen

Abstract

The interest for Internet of Things has become more widespread and has expanded over the last couple of years. Internet of Things are present in ovens, light-bulbs etc. One common requirement for all these devices is the need for communication. The communication can potentially be facilitated by a message oriented middleware.

Currently, most message oriented middleware, have been based on the Transmission Control Protocol, which in theory is not very suitable for Internet of Things.

This thesis introduces HarakaMQ, a fast, reliable and light-weight message oriented middleware based on the User Datagram Protocol. With the ability to drive down protocol overhead and coordination messages, a lower resource usage should be achievable.

A solution has been designed, implemented and tested in this thesis. To quantify the quality of HarakaMQ a statistical comparison with state of the art message oriented middleware based on the Transmission Control Protocol has been performed. The solution implemented is superior in the number of coordination messages and resource usage used. The implemented solution has showed a remarkable potential, with further improvements HarakaMQ can excel and surpass existing message oriented middleware based on the Transmission Control Protocol.

Keywords: Message oriented middleware, Optimization of coordination messages and reliability, Efficient algorithms

Resumé

Interessen for Internet of Things er blevet mere udbredt og har vokset over de sidste par år. Internet of Things er tilstede i ovne, lyspære osv. Et fælles krav for alle disse enheder er kravet for kommunikation. Denne kommunikation kan potentielt set blive faciliteteret af en besked orienteret middleware.

I øjeblikket er de fleste besked orienteret middleware baseret på Transmission Control Protocol, som i teorien ikke er særlig velegnet til Internet of Things.

Denne afhandling introducere HarakaMQ, som er en hurtigt, pålideligt og letvægts besked orienteret middleware baseret på User Datagram Protocol. Med evnen til at neddrive protokol overforbruget og koordinationsbeskederne burde et mindre ressource forbrug være muligt.

En løsning er designet, implementeret og testet i denne afhandling. For at kvantificere kvaliteten af HarakaMQ er en statistisk sammenligning med en nyere besked orienteret middleware baseret på Transmission Control Protocol blevet udført. Den implementeret løsning er overlegen i forhold til antallet af koordinationsbeskeder og ressource forbrug. Den implementeret løsning har vist et nævneværdig potentiale og med yderligere forbedringer kan HarakaMQ udmærke sig og overgå nuværende besked orienteret middleware baseret på Transmission Control Protocol.

Nøgleord: Besked orirenteret middleware, Optimering af koordinations beskeder og pålidelighed, Effektive algoritmer

Contents

1 | Introduction

1.1 Background

This thesis has its origin in distributed systems, and the communication optimization for Internet of Things (IoT). The thesis aims to create a message oriented middleware based on the User Datagram Protocol (UDP), named HarakaMQ through out the thesis.

> *Haraka means fast in Swahili [1], which is the ambition for HarakaMQ, to become a fast, reliable and light-weight message oriented middleware suitable for IoT and mobile devices.*

IoT is everywhere today, and it is expanding [2]. Communication is essential in IoT, for example in the smart home where IoT devices communicate with each other, and where it is possible to communicate with IoT devices through a mobile device. This communication has to be facilitated.

Even though communication is essential in IoT, a lot of state-of-the-art message oriented middleware, such as RabbitMQ [3], uses the Transmission Control Protocol (TCP). TCP was designed to be reliable and maintain a long lived connection to deliver large bulk of data efficiently [4]. However the communication involving IoT and mobile devices typically do not required large bulk of data, and are required to go into sleep mode because of the energy constraints, thus it is not feasible to use TCP.

UDP is an unreliable light-weight transport protocol [4]. The UDP header is used together with a 20 bytes IPv4 header and also has its own 8 bytes header, which in total gives a header of 28 bytes per message. Compared to TCP which has a header of 20 bytes, giving a total header of 40 bytes per message. This is a difference of 12 bytes per message between UDP and TCP. Compared to TCP, UDP does not require a connection to the receiving side, and therefore a minimum of coordination messages are needed. An UDP message is sent without no guarantee of delivery. To guarantee delivery in a message oriented middleware based on UDP a mechanism to guarantee reliability is required.

As this thesis is created in cooperation with Danske Bank and MobilePay, it is relevant to look at some of the specific cases where a message oriented middleware can be used within the organization.

IoT and mobile devices have energy constraints, and therefore the energy consumption of operations such as sending and receiving data, should be optimized to use a minimum of energy [4], [5].

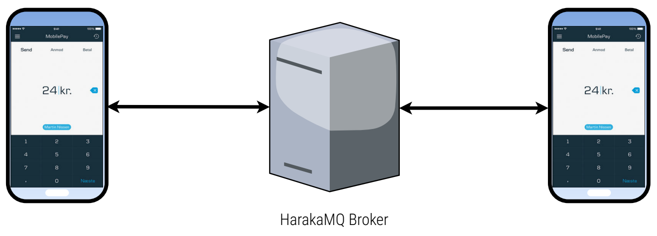


Figure 1.1: MobilePay facilitating the communication via HarakaMQ (Screenshot has been taken from MobilePay's marketing material)

Because of the limited energy in mobile devices the heavy-weight protocol TCP is not suitable for facilitating the communication. An example of a mobile device application could be the MobilePay application, where money are transfered between users, as easy as sending a text message. In MobilePay today an HTTP call is made from the mobile device to a REST API, from where the information is passed to RabbitMQ. From here RabbitMQ ensures with its reliability that messages are delivered to the rest of the system [6]. This reliability consist of replication of messages between brokers and message persistence on brokers, so in case of a system failure the messages can still be delivered at a later point in time. In theory a message oriented middleware, could facilitate the communication from the mobile device. In MobilePay's case neither the HTTP call or RabbitMQ is the optimal solution for facilitating the communication from the mobile devices with regards to energy consumption, as both solutions are based on TCP. Instead a message oriented middleware based on UDP, can be used to facilitate the communication between two mobile devices, which is shown in figure 1.1.

This thesis investigates if a message oriented middleware based on UDP can be created with fewer coordination messages and lower energy consumption, and have the same delivery guarantees as existing solutions on the market such as RabbitMQ [7].

1.2 Motivation

The term IoT is becoming more and more relevant when talking about distributed systems, and the increasingly acknowledgement of IoT becoming a real business opportunity, makes it a relevant motivation topic for this thesis [8].

RabbitMQ has with its reliability and usability become one of the leading message oriented middleware on the market [3]. Even though RabbitMQ supports the Message Queue Telemetry Transport (MQTT) protocol, which is energy efficient and build for IoT solutions, MQTT is still based on TCP, which in theory is not very suitable for IoT solutions. The motivation for this thesis is to build a message oriented middleware suitable for IoT solutions and mobile devices, which leads to the motivation to build the middleware using UDP as the underlying transport protocol.

An obvious question in this thesis is why MobilePay or other mobile application companies, should use a message oriented middleware based on UDP with reliability, to facilitate their communication between the mobile devices. In comparison to MobilePay, they can remove a layer in their technology stack, by using a message oriented middleware directly on the mobile device.

Summarizing, the motivation of this thesis is to design, develop, test and document a reliable and lightweight message oriented middleware based on UDP. The message oriented middleware should be optimized with regards to coordination messages, reliability and IoT and mobile devices, and the goal is to minimize the energy consumption, by minimizing hardware usage, and the number of coordination messages.

The hope is that the solution will seem fit for IoT solutions and mobile devices, and that the solution can start a new revolution for message oriented middleware.

1.3 Related Work

A related scientific work has been performed by Joerg Fritsch and Coral Walker [9]. Their work had a lot of resemblance with the work of this thesis. Joerg Fritsch and Coral Walker created a light weight message queue implemented in Haskell [10]. Their concept was to use UDP instead of TCP, and the motivation of doing this is that UDP is connectionless, and has a smaller protocol header than TCP. Joerg Fritsch and Coral Walker do not offer a message delivery guarantee. Which suggests that it is acceptable for Joerg Fritsch and Coral Walker that a message is lost. They use a broker less architecture, because Joerg Fritsch and Coral Walker uses multicast to broadcast the messages to all the receivers. This results in a light weight message queue which they conclude is fast.

Another related work which is used in the industry is RabbitMQ. RabbitMQ is a reliable message oriented middleware implemented in Erlang [11]. RabbitMQ ensures guaranteed delivery of messages, and fault tolerance if a broker goes down [6]. Reliability and fault tolerance depends on how the developer uses RabbitMQ. Besides this RabbitMQ also offer high availability, but this also depends on how the developer uses RabbitMQ.

The difference between Joerg Fritsch and Coral Walker's work and this thesis, is that they do not offer reliability, and the purpose of this thesis is to create an reliable light weight message oriented middleware. The result of this thesis work should be in style with RabbitMQ, but with the use of UDP as transmission protocol. It is imperative that the message oriented middleware is highly available, reliable and fault tolerant like RabbitMQ.

1.4 Problem Formulation

The focus of the present master's thesis is to design, implement, test, and document a reliable and lightweight message oriented middleware based on the User Datagram Protocol. The middleware should be able to automatically optimize the balance between coordination messages and reliability by adapting measures such as: 1. Automatic Repeat reQuest, 2. Guaranteed delivery, and 3. Replication mechanisms. The optimization should lead to a middleware that may achieve the same reliability and replication capabilities as a message oriented middleware using the Transmission Control Protocol, but with fewer coordination messages due to the ability to drive down protocol overhead.

The developed middleware must be optimized with regard to coordination messages and time and space complexities, to maximize the information per message. Moreover, the middleware must prove itself functional via concrete experiments. To quantify the quality of the middleware, statistical comparisons with state of the art message oriented middleware should be carried out via objective quality metrics.

> ***Note:*** *Any traffic between participants in the network is identified as a coordination message (e.g. protocol, acknowledgement, heart beat, packet).*

1.5 Scope

The scope of this thesis is centered around designing, developing, testing and documenting a fast, reliable and lightweight message oriented middleware, mentioned as HarakaMQ. HarakaMQ is limited to use the UDP protocol for communication. Reliability measures such as automatic repeat request and guaranteed delivery is part of this thesis. The correlation between consistency, availability and partition tolerance is considered, while different techniques to ensure these are discussed and implemented in this thesis.

The comparison of HarakaMQ is limited to RabbitMQ, which is a state of the art message oriented middleware. Furthermore RabbitMQ is one of the most used in the world [7]. As the thesis is created in cooperation with Danske Bank and MobilePay, it is imperative to compare HarakaMQ with the current solution in use at MobilePay, which is RabbitMQ.

RabbitMQ can be configured in a lot of different ways to enhance the specific usage scenario, but in this thesis a default configuration of RabbitMQ is used, as the configuration of RabbitMQ is considered being out of the scope for this thesis.

The research done by Lauri Minas and Brad Ellison at Intel Corporation [12] indicates that there is a correlation between energy consumption and CPU load and disk I/O. Therefore the CPU load and disk I/O are used to give an indication of the energy consumption in this thesis.

Security is not part of this thesis, and therefore no security measures will be taken into account.

1.6 Concepts

1.6.1 Connection Disruption

A connection disruption is when a sender is disconnected from the receiver, and thereby being unable to send or receive messages from or to the receiving part.

1.6.2 System Failure

The system a sender or receiver is running on might not always be operational, this is also known as a system failure. System failures can be either hardware or software caused, and often requires user interaction to be fixed. The consequences of a hardware or software related system failure, can result in the RAM being cleared, and it is therefore required to store vital information on the disk.

Another part to consider is system reboots (because of OS patches, updates etc.), which also requires persistent storage of vital data.

1.6.3 Desired State

The desired state of a system is when all the sent messages are received at the receiver, and the sender and the receivers state is updated accordingly. In figure 1.2 it is shown how the receiver receives three messages from the sender, and the counters are set accordingly, thereby both parties have an agreement of the system state, and the desired state has been achieved.

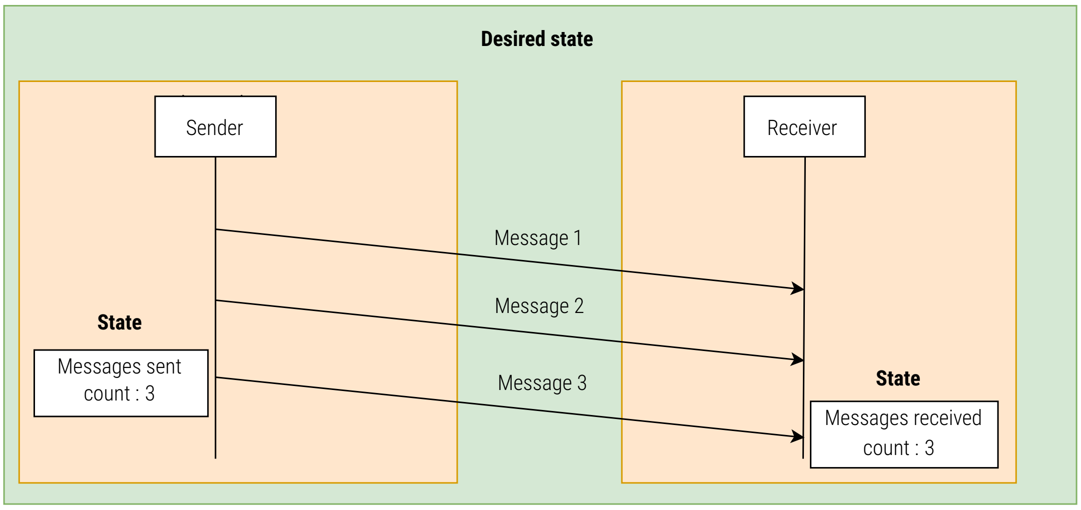


Figure 1.2: An example of the receiver receiving three messages from the sender, and counters set accordingly

2 | Collaboration with Danske Bank and MobilePay

2.1 Danske Bank and Talent Box

This thesis was developed in cooperation with Danske Bank and the Talent Box. The thesis was part of the Talent Box, but the Talent Box is an incubator for students, so the Talent Box itself does not have an interest in the thesis. However, the Talent Box established a contact with the department of MobilePay.

Danske Bank is a Nordic universal bank which is present in 16 countries and has 19.700 employees worldwide [13]. In Denmark they have a development office located in Brabrand, Denmark. This is also where the development department of MobilePay is situated. Furthermore, it is in Brabrand Denmark the thesis work was done.

Danske Bank has created the Talent Box to be disrupted from the inside, by the help of academic students, using an academic approach to solve problems. Especially with the ever-increasing pace of development within the world of IT.

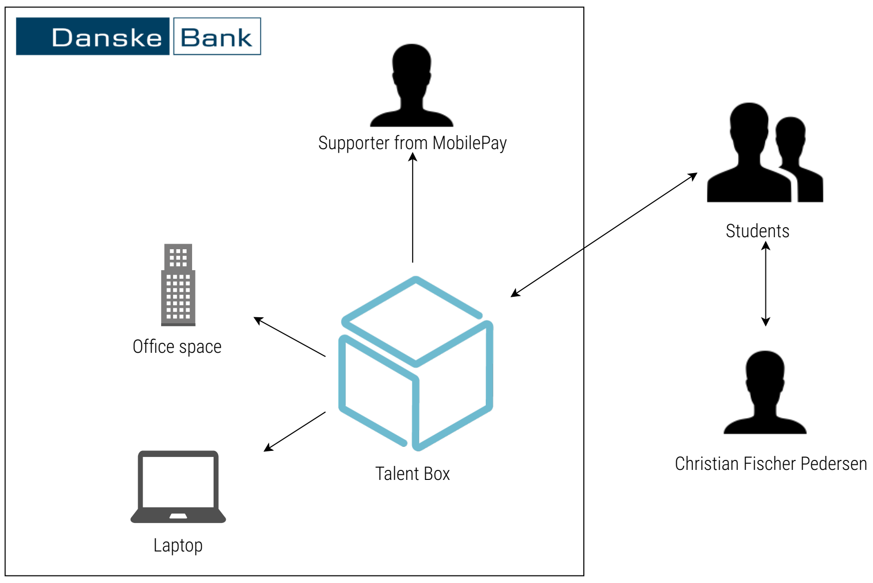


Figure 2.1: Overview of the Talent Box setup

Figure 2.1 shows the structure of the Talent Box. The advantages of working with the Talent Box, is the office space, equipment and the knowledge sharing between the students and MobilePay.

2.2 MobilePay

MobilePay is a mobile application which makes it possible to transfer money between users or companies. The essence is that the money transfer should be as easy as sending a text message. MobilePay has more than 3.4 million Danes which has downloaded the mobile application [14]. MobilePay's mobile application can been seen in figure 2.2

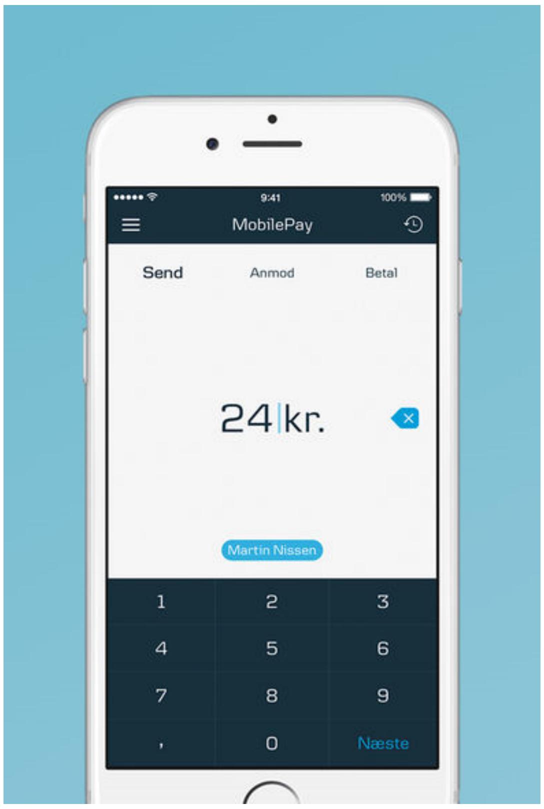


Figure 2.2: Screenshot of MobilePay's mobile application on an Apple iPhone (Screenshot has been taken from MobilePay's marketing material)

MobilePay had a supporting role in this thesis work. The role as supporter was handed to Martin Castberg Thuesen. He is an expert in RabbitMQ, and this is why he was chosen as the supporter.

MobilePay's take away from this thesis work is based on the interest to see if a message oriented middleware can be used to handle and guarantee a delivery of messages from devices with frequent disconnections, such as mobile devices. The current solution used in MobilePay today incorporate a HTTP call from the device to a REST API, and from here the message is passed to RabbitMQ, which is the message oriented middleware used in MobilePay. The goal is to replace the current solution in MobilePay, and deliver a direct connection between the mobile devices. Which results in no translation from the REST API to the message oriented middleware, hence simplifying the technology stack.

3 | Internet of Things

3.1 The Three Layers in IoT Systems

The interest in IoT has increased the last couple of years, and companies have been introducing numerous IoT-based products [8]. IoT is a network of devices that can exchange data and communicate with each other. Devices in IoT can communicate by different network protocols, such as Ethernet, WiFi and Bluetooth. For a device to communicate with another device it needs a connectivity port or an antenna. Figure 3.1 shows some of the devices that for example can be connected in IoT.

One usage example of IoT, could be that when waking up in the morning, and turning off the alarm, the alarm would tell the coffee machine to start making a cop of coffee. Another example of IoT is the smartwatch, that can connect to a mobile device, and thereby exchange information from the watch to the mobile device, or the other way around. Some IoT devices might depend heavily on reliable communication, such as the alarm and coffee machine example. If a user does not get his coffee every day, because a message was not delivered, the user of the device would not be satisfied with the product.

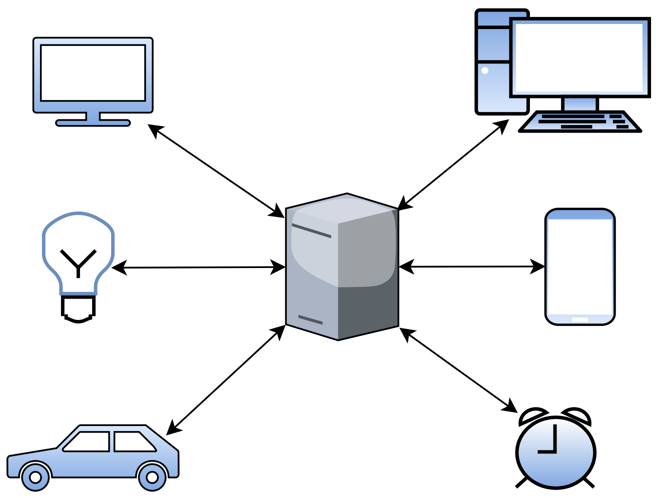


Figure 3.1: Illustration of IoT devices

Message oriented middleware that use TCP provides the desired reliability guarantees, but TCP is not suited for IoT devices, as they might go into sleep mode to ensure low energy consumption [4]. When going into a sleep mode, an IoT device cannot keep the TCP connection alive. HarakaMQ is designed to be connectionless, so a stable connection is not needed. HarakaMQ should provide the same reliability as middleware relying on TCP, even though it is based on UDP, therefore HarakaMQ should be a viable message oriented middleware to use in IoT systems.

An IoT system usually consists of three layers. The first layer is the IoT cloud layer [8]. The second layer is the Connectivity layer, and the last layer is the Thing / Device layer. The IoT cloud layer is used to coordinate the interaction between people and devices. The IoT Cloud layer also have software to handle communication with devices. The Connectivity layer enables the communication between the IoT cloud and the Thing / Device. HarakaMQ could be used in the Connectivity layer to handle the communication.

The Thing / Device layer has embedded software that is running on the device, which performs a specific task. The Thing / Device layer also consists of hardware components, and sensors or actuators that are to be used in the functionality of the device. An example of a sensor being added, could be a lamp using a sensor to detect persons, and when a break-in is happening the lamp could turn on the light, and send an alarm message to the user's mobile device.

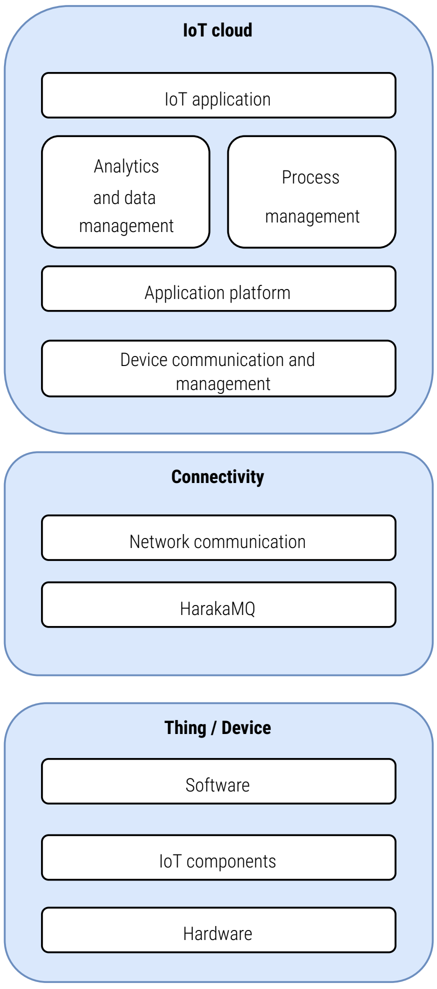


Figure 3.2: The three layers in an IoT system (based on Wortmann [8])

As IoT is becoming more and more relevant different operating systems have been developed to run on IoT devices. Windows 10 IoT Core is designed by Microsoft to run on IoT devices [15], and therefore require a minimum of resources. By using Windows 10 IoT Core, it enables the developer to use Windows development tools and built-in security and management tools.

IoT can be used in the daily life and on many devices, a challenge in IoT is energy efficiency, as a device is to be connected to the Internet, without getting maintenance [4]. Energy efficiency is also important as many IoT devices are expected to be running on resource constrained devices, and the optimization of the software is therefore important.

3.2 Resource Requirements

The following section will cover the resource requirements for IoT devices, taking into account the constraints for .NET Core 2.0 and the different operating systems supporting .NET Core 2.0.

The different operating systems that supports .NET Core 2.0 can be seen in table 3.1.

Windows [16]	**Linux [17]**	**MacOS [18]**
• Windows 7 SP1 • Windows 8.1 • Windows 10 • Windows 10 IoT Core • Windows Server 2008 R2 SP1 (Full Server or Server Core) • Windows Server 2012 SP1 (Full Server or Server Core) • Windows Server 2012 R2 (Full Server or Server Core) • Windows Server 2016 (Full Server, Server Core, or Nano Server)	• Red Hat Enterprise Linux 7 • CentOS 7 • Oracle Linux 7 • Fedora 25, Fedora 26 • Debian 8.7 or later versions • Ubuntu 17.04, Ubuntu 16.04, Ubuntu 14.04 • Linux Mint 18, Linux Mint 17 • openSUSE 42.2 or later versions • SUSE Enterprise Linux (SLES) 12 SP2 or later versions	• macOS 10.12 Sierra and later versions

Table 3.1: Operating systems supporting .NET Core 2.0

In table 3.2 the resource requirements are described for .NET Core 2.0, Windows, Linux and MacOS. Windows 10 IoT Core and Ubuntu 16.04 LTS Server are the versions of Windows and Linux that has the lowest resource requirements, and still being able to run .NET Core 2.0. As MacOS Sierra is the only version of MacOS supporting .NET Core 2.0 it has been selected as the version with lowest resource requirements.

Table 3.2 shows the resource requirements for the selected operating systems and .NET Core 2.0.

Name	CPU	RAM	Disk
.NET Core 2.0	Depends on the desired performance	36 MB	389 MB
Windows 10 IoT Core [15]	x86/x64 processor/SoC: 400 MHz	256 MB headless and 512 MB with head	2 GB
Ubuntu 16.04 LTS Server [19]	300 MHz x86 processor	256 MB	1.5 GB
MacOS Sierra [20]	-	8.8 GB	2 GB

Table 3.2: Resource requirements specification

> *Because of limited documentation on the resource requirements for .NET Core 2.0, tests have been made in this thesis to decide the resource requirements for .NET Core 2.0.*

In table 3.2 it can be seen that MacOS Sierra need more RAM compared to other operating systems, and the minimum CPU requirement has not been documented for MacOS Sierra.

An overview has been given of the required resources for IoT, and this is to be considered doing the design and implementation of HarakaMQ.

4 | Requirements of HarakaMQ

The quality properties described in this chapter are derived from the ISO/IEC 25010:2011 standard [21].

4.1 Evaluation Properties

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) develop worldwide standards through technical committees to deal with different technical fields. The ISO/IEC 25010:2011 is part of the international series of standards called SQuaRE (Systems and software Quality Requirements Evaluation). The SQuaRE series consists of different divisions, where the ISO/IEC 25010:2011 standard is present under the Quality Model Division.

The ISO/IEC 25010:2011 standard contains a product quality model, that has eight characteristics with multiple sub-characteristics, which are used to ensure the quality of HarakaMQ. The ISO/IEC 25010:2011 standard is used to define quality, and can therefore help the process of identifying requirements. The ISO/IEC 25010:2011 is used, as it is the international standard. Even though many companies might use their own variant of the ISO/IEC 25010:2011, it is still the underlying standard, and the concepts involved are well known by companies around the world [22].

The requirements for HarakaMQ are derived from the problem definition, and four characteristics with sub-characteristics, from the ISO/IEC 25010:2011 quality model, are used to describe the requirements for HarakaMQ. The four characteristics used are: *functional suitability, performance efficiency, portability and reliability*. These four characteristics with their sub-characteristics can be seen in figure 4.1.

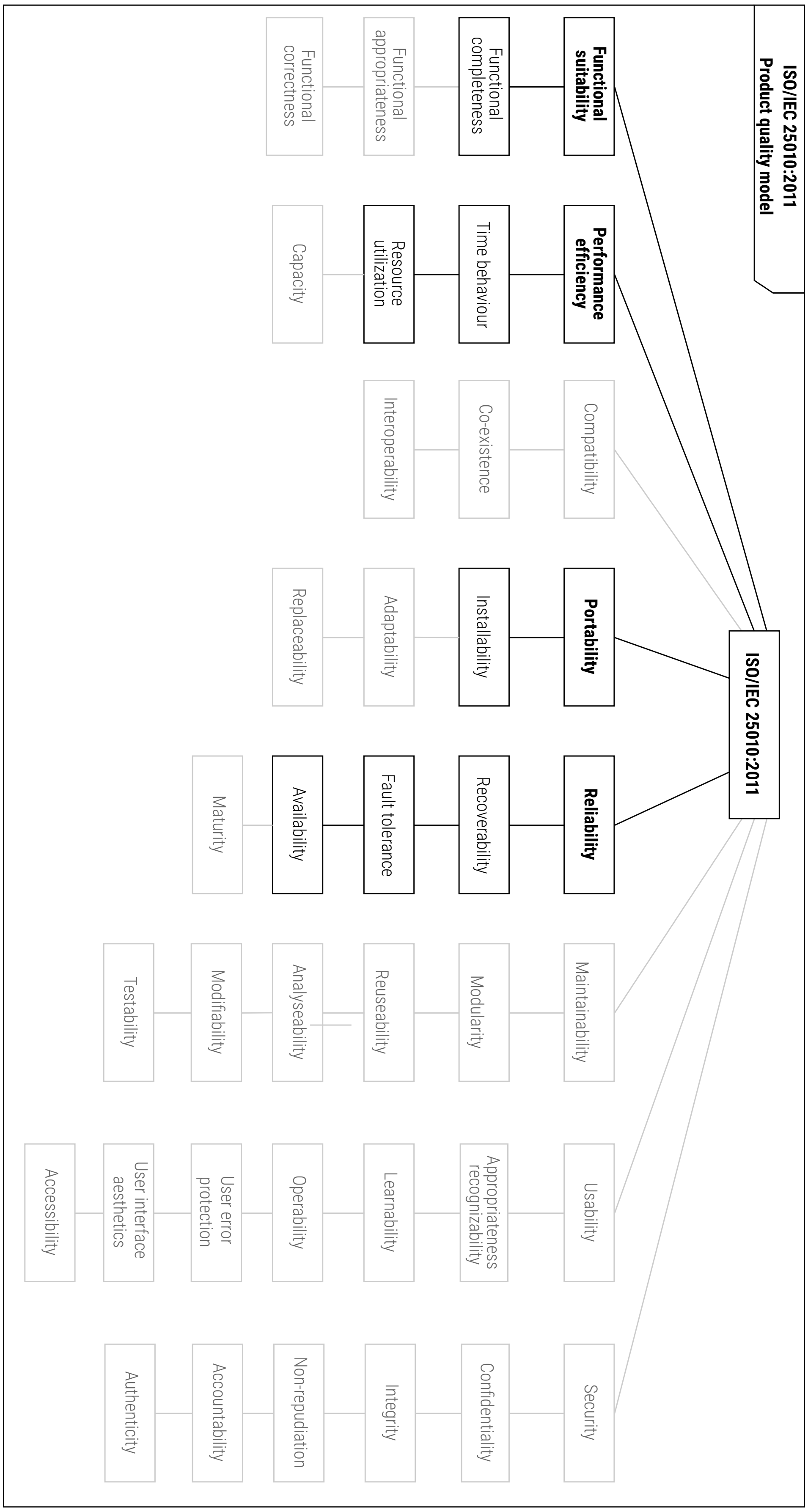


Figure 4.1: The quality properties from the product quality model of the ISO/IEC 25010:2011 standard

In figure 4.1 both the used and unused sub-characteristics are shown. The unused sub-characteristics are faded-out in the figure. The used sub-characteristics for conducting the requirements are shown for each used characteristics.

4.2 Requirements

> *Each requirement has one or more sub-requirements which needs to be fulfilled for the requirement to be approved. Sub-requirements can either return true or false.*

4.2.1 Functional Suitability

Functional suitability is whether the product provides the functions that satisfy the desired needs. The two sub-characteristics *functional completeness* and *functional appropriateness* is used to represent the requirements.

4.2.1.1 Functional Completeness

Functional completeness is the amount to which the functions assure that all specified tasks are fulfilled. Table 4.1 contains the sub-requirements for requirement 1.

1. HarakaMQ shall be able to perform interprocess communication in a distributed system

The requirement originates from the need of achieving the same reliability as a message oriented middleware using TCP.

1.1	HarakaMQ is performing interprocess communication via the User Datagram Protocol
1.2	HarakaMQ is using the publish/subscribe message pattern for interprocess communication

Table 4.1: Sub-requirements for requirement 1

4.2.2 Performance Efficiency

Performance efficiency is used to describe how efficient the product perform under stated conditions and also describes the amount of resources used. The two sub-characteristics *time behaviour* and *resource utilization* is used to represent the requirements.

4.2.2.1 Time Behaviour

Time behavior is the rate of the acceptable response, processing times and throughput rates of the product. Table 4.2 contains the sub-requirements for requirement 2.

2. HarakaMQ shall be able to perform better than RabbitMQ

The requirement is needed as HarakaMQ shall provide a better average throughput than a message oriented middleware using TCP, because of the ability to drive down protocol overhead.

2.1	HarakaMQ is able to process a better average throughput of messages than RabbitMQ, when RabbitMQ is configured to a client-server setup
2.2	HarakaMQ is able to process a better average throughput of messages than RabbitMQ, when RabbitMQ is configured to a cluster based setup

Table 4.2: Sub-requirements for requirement 2

4.2.2.2 Resource Utilization

Resource utilization is the amount of resources needed by the product, to perform its functions. Table 4.3 contains the sub-requirement for requirement 3.

3. HarakaMQ shall be able to run on IoT devices

The following requirement originates from the need for optimization regarding coordination messages and time and space complexities.

3.1	HarakaMQ is able to run on a Raspberry Pi 3 Model B [23]

Table 4.3: Sub-requirements for requirement 3

4.2.3 Portability

Portability is whether a product can be effectively transfered from one hardware, software or operational system to another. The sub-characteristics *Installability* is used to represent the requirement.

4.2.3.1 Installability

Installability is the possibility for a product to be successfully installed or uninstalled in a specified environment. Table 4.4 contains the sub-requirements for requirement 4.

4. HarakaMQ shall be able to execute on several platforms

The requirement is needed as the optimization of coordination messages and time and space complexities could lead to a usage of HarakaMQ in IoT.

4.1	HarakaMQ is installable on Windows 10, Ubuntu 17.04 and Windows 10 IoT Core

Table 4.4: Sub-requirements for requirement 4

4.2.4 Reliability

Reliability is whether a product performs reliable under specified conditions for a specified period of time. The two sub-characteristics *recoverability* and *fault tolerance* is used to represent the requirements.

4.2.4.1 Recoverability

Recoverability describes if the system is able to recover data and re-establish the desired state of the system, if a failure occur. Table 4.5 contains the sub-requirements for requirement 5.

5. HarakaMQ shall be able to re-establish the desired state of the system if a system failure or connection disruption occurs
The following requirement originates from the need for reliability and availability.

5.1	HarakaMQ is able to re-establish the desired state after a connection disruption
5.2	HarakaMQ is able to re-establish the desired state after a system failure

Table 4.5: Sub-requirements for requirement 5

4.2.4.2 Fault Tolerance

Fault tolerance is whether a product operates as intended, when experiencing a system failure. Table 4.6 contains the sub-requirements for requirement 6.

6. HarakaMQ shall be able to handle a system failure or connection disruption of of a system component which never returns to the system again
The following requirement is a result of the need for the same reliability as a state of the art message oriented middleware

6.1	HarakaMQ is able to handle if a system component experience a failure and never returns to the system
6.2	HarakaMQ is able to replicate data to multiple system components, to avoid loss of data

Table 4.6: Sub-requirements for requirement 6

4.2.4.3 Availability

Availability is whether a product is working and accessible when the product is needed for use. Table 4.7 contains the sub-requirements for requirement 8.
7. HarakaMQ shall always be available
The requirement originates from the need for availability

7.1	HarakaMQ guarantees to always be available

Table 4.7: Sub-requirements for requirement 7

5 | Principles of Distributed Computing

5.1 Interprocess Communication

In distributed systems the Internet Protocol (IP) is used to support communication between computers, by identifying their IP addresses [24, pp. 106-128]. TCP and UDP are transport protocols, and are used for process-to-process communication. An example of process-to-process communication can be seen in figure 5.1. The IP will deliver a message to a certain host, with a specific IP address, thereafter the UDP- or TCP-layer will deliver the message to a process by sending the message through a specific port on the host.

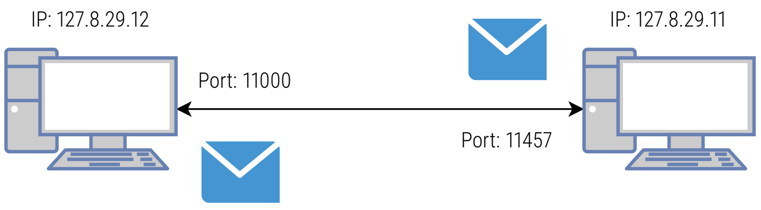


Figure 5.1: Interprocess communication between two computers

5.1.1 The Characteristics of Interprocess Communication

The aspects of interprocess communication can be handled by a message oriented middleware, with underlying TCP or UDP. Passing a message between two processes requires a send and receive operation. One process sends a message to a destination, while another process at the destination receives the message. This communication can be done either synchronous or asynchronous. Both UDP and TCP communication makes use of sockets. A socket is an endpoint for communication between processes. Interprocess communication is the transmission of a message between two sockets belonging to their own process. For a process to receive a message its socket must be bound to a local port and an IP address of the computer. Thereby a message sent to that particular IP address and port can only be received by that specific process.

5.1.2 User Datagram Protocol

UDP is a connectionless protocol [24, pp. 146-158], which means that the sender of a message do not need to ensure that the receiver is available and ready to receive. Besides this the protocol do not guarantee delivery or the order of the messages. UDP can detect errors introduced while transmitting, as it has a checksum field in the header, and thereby drops the message if the checksum fails.

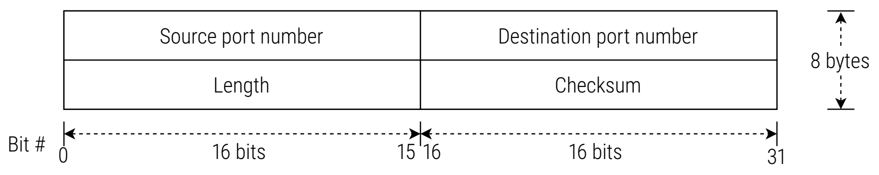


Figure 5.2: Header for UDP

In figure 5.2 the UDP header with 8 bytes can be seen [25]. The source port is optional, and can be set to zero, as UDP per default does not need a response. The length field is used to describe the length of the message, including both header and data. UDP is mainly used for applications that do not require reliable and ordered delivery, such as video streaming applications.

5.1.3 Transmission Control Protocol

TCP provides reliable delivery and ordering of messages. TCP is connection oriented, which means that a connection will have to be established between the sender and the receiver before the data transfer can commence. The connection between the sender and receiver needs to be bidirectional, as it is required to send messages both ways. A message is used to establish connections, transfer data, send acknowledgments and closing connections in TCP [25, pp. 209-247]. A message is divided into a header and a data part. The TCP header carries the identification and control information.

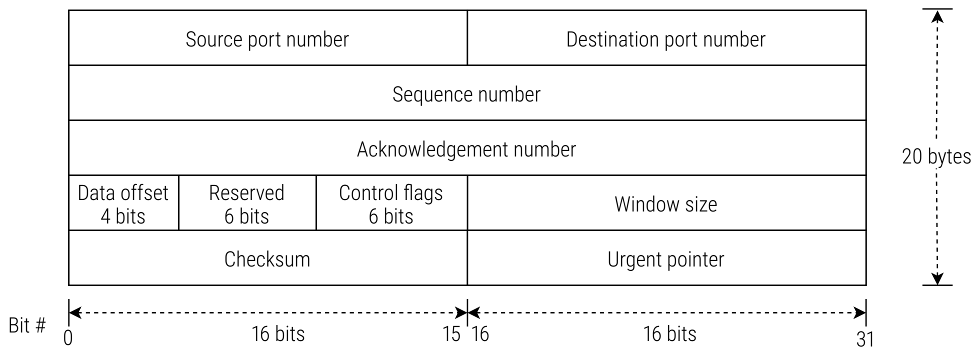


Figure 5.3: Header for TCP

In figure 5.3 the TCP header with 20 bytes can be seen. The TCP header consists of a source and destination port number, exactly like the UDP header. It also consists of a 32 bit sequence number. The sequence number is used to describe the position of the message data compared to the byte stream coming from the sender. The acknowledgment number is used to describe which message the source expects to get an acknowledgment for next, and therefore the acknowledgment number refers to the communication going from destination to source. The control flags are used to determine whether a message carries data, or if it is an acknowledgment. The urgent flag can be set, meaning that the message is urgent, and the receiving application should be notified as fast as possible. An urgent message will be pushed to the application even though it is out of order. The urgent pointer field is used to specify where urgent data ends in the message. With the urgent flag TCP is able to send urgent data within the same connection as normal data is being transmitted. An example of the usage of the urgent pointer is when connecting with a remote login session. The remote login requires to be done urgently, before the ongoing process on the remote machine is done. The checksum field is used to detect errors as in UDP. The header also consists of a window size, which is used to determine the size of the sliding window.

The sliding window allows a sender to send multiple messages, before expecting an acknowledgment for the first message. If an acknowledgment for a message is not returned before a specified timeout, the sender will retransmit the message. For a sliding window with window size five the first five messages will be sent, and thereafter the sender will wait for acknowledgements, before sending any more messages. When an acknowledgement is received for the first message, the sender will send a new message. The sliding window controls the flow in a system, by blocking messages until acknowledgements are received.

In TCP there are two approaches to acknowledge messages. The first approach is to acknowledge every message. The second approach is to use delayed acknowledgements, and there by when sending multiple messages, acknowledgements are bundled together into one packet.

TCP is used for systems with a requirement for high reliability. TCP is slower than UDP, as UDP is a best-effort protocol and do not attempt error recovery. TCP is used in the Internet protocol suite together with the Internet Protocol (IP), hence the name TCP/IP, as reliable data transfer is needed.

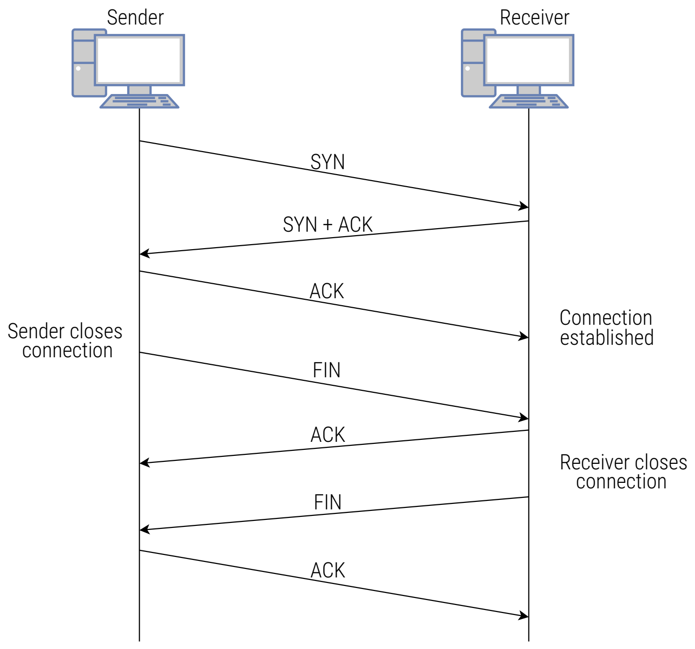


Figure 5.4: Establishing and closing a TCP connection

In figure 5.4 the connection establishment and closing of a TCP connection can be seen [25, pp. 237-240]. To establish a TCP connection a three way handshake is used. The first handshake is from the sender wanting to establish the connection, where a SYN message is sent from the sender to the receiver. The receiver answers the sender by sending back a SYN and ACK message in one message, and thereby continuing the handshake. At last the sender sends back an ACK message to the receiver, resulting in the connection being established.

When a TCP connection is to be closed the sender tells the receiver that it does not have any more data to send, by sending a FIN message to the receiver. The receiver returns an ACK message telling the sender, that the connection has been closed in the given direction. After some time the receiver sends a FIN message to the sender, to close the connection in the other direction. At last the sender sends back an ACK message back to the receiver, and thereby the TCP connection has been finalized.

The messages used to establish and close a connection are non existing in UDP, as a connection is not needed. In TCP acknowledgement messages are sent to acknowledge that a message has been received, which is not needed in UDP. The absence of the following messages in UDP, makes UDP capable of using a lower amount of coordination messages to send messages between a sender and receiver, but without any guarantees.

5.2 Automatic Repeat ReQuest

Message delivery and order are not guaranteed in UDP. To ensure the delivery of messages Automatic Repeat ReQuest (ARQ) can be used. ARQ uses acknowledgements, which are messages, sent by the receiver of the message, indicating that the message was received. When no acknowledgement is received the sender knows that the message was not received, and a retransmission of the message is necessary. There are three different kinds of ARQ protocols, Stop-and-Wait, Go-Back-N and Selective Repeat.

5.2.1 Stop-and-Wait

In the Stop-and-Wait protocol the sender sends one message at a time, while waiting for an acknowledgment [26]. The sender will first send the next message after receiving an acknowledgement for the first message. A timeout function is implemented to handle if an acknowledgment is never received. The timeout function will resend the message after a certain amount of time. Stop-and-Wait has the disadvantage that it does not utilize the maximum channel capacity, as it has to wait for an acknowledgment before sending another message. An example of the Stop-and-Wait protocol can be seen in figure 5.5.

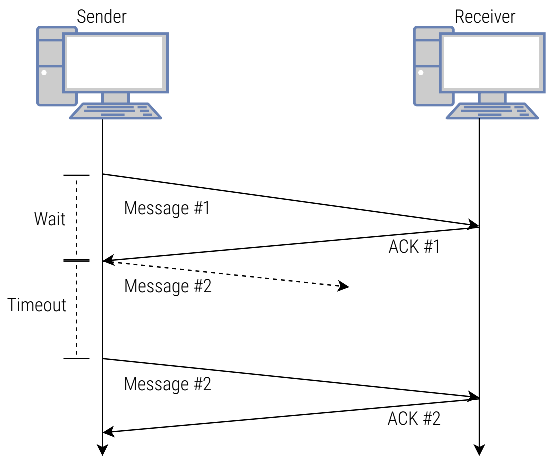


Figure 5.5: Sequence for the Stop-and-Wait protocol

5.2.2 Go-Back-N

The Go-Back-N protocol allows the sender to send multiple messages before receiving an acknowledgement. Each message consists of a sequence number, which tells the receiver if the received message is out of order. If the received message is out of order, the receiver knows that it did not receive a previous message, and that this message should be resend. The Go-Back-N protocol will go back to the non received message, and resend every message from there and forward. Go-Back-N has a better utilization of the channel, as it allows to send multiple messages. The amount of messages sent before an acknowledgment is needed, is decided by the size of the sliding window, which is used in the Go-Back-N protocol. The drawback with the Go-Back-N protocol is that it might resend messages, that the receiver already received. In figure 5.6 an example of this drawback can be seen, and it can be seen that message number four is received twice on the receiver site. With Selective Repeat this drawback can be eliminated.

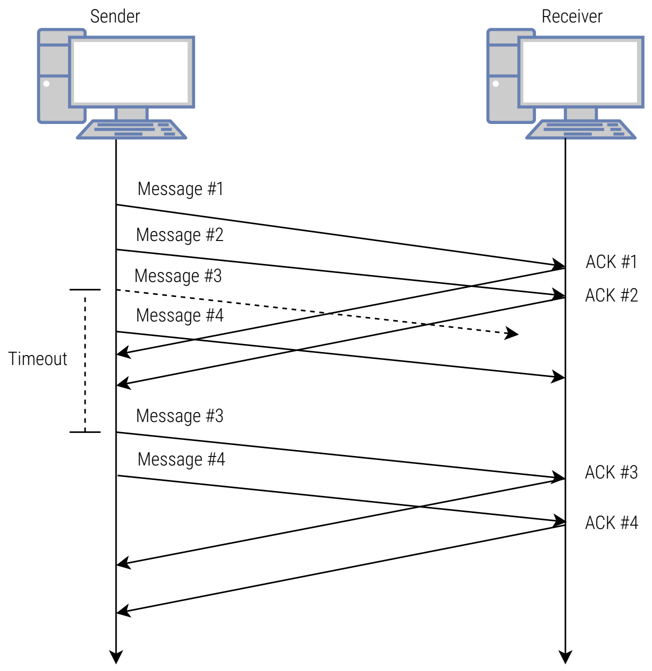


Figure 5.6: Sequence for the Go-Back-N protocol

5.2.3 Selective Repeat

The Selective Repeat protocol only resends the message that was not received at the receiver, hence a selective message is resend. Therefore the receiver needs to save the out of order messages in a buffer. When receiving the missing message the out of order messages will be handled on the receiver site. This way the channel utilization is maximized, as the sender will not have to resend already received messages. An example of the Selective Repeat protocol can be seen in figure 5.7. A drawback with Selective Repeat is that the receiver needs to save out of order messages in a buffer. The buffer on the receiver site will always be limited by the disk space on the receiver, and usually a sliding window is used to handle this drawback. With a fixed window size for the sliding window, the buffer will never exceed the size of the sliding window.

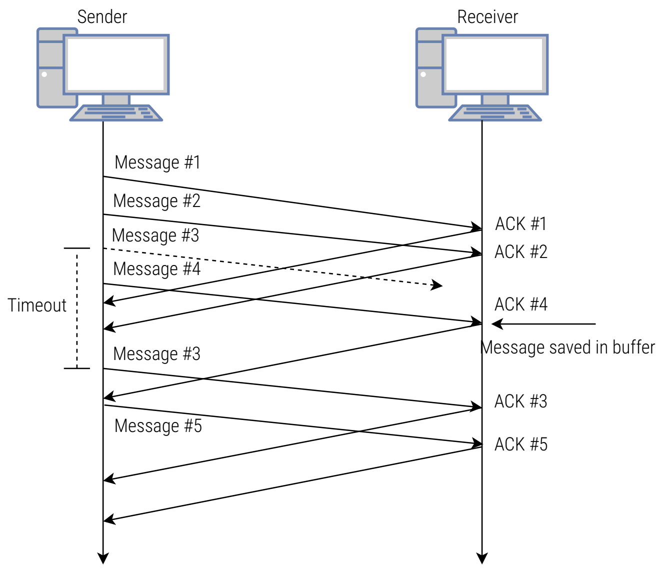


Figure 5.7: Sequence for the Selective Repeat protocol

5.3 Different Architectures in Distributed Computing

5.3.1 Client-server

The client-server setup is one of the mostly deployed architectures in the context of distributed systems [24, pp. 46-47]. The client-server setup can be seen in figure 5.8. The client-server setup is also known as a centralized architecture, as the server acts as a central computer with stored data and the ability to control who shares what [27]. In the client-server setup the client sends a request to the server, and the server replies with a response. An example of a client-server setup is a browser (client) requesting a specific web page from a web server. The web server will after the request return the requested web page. The server is also the place where data can be stored and extracted by the different clients. Processing can be moved from the clients to the server. A client-server based application often consists of a tiered architecture, which will be elaborated in Section 5.3.2.

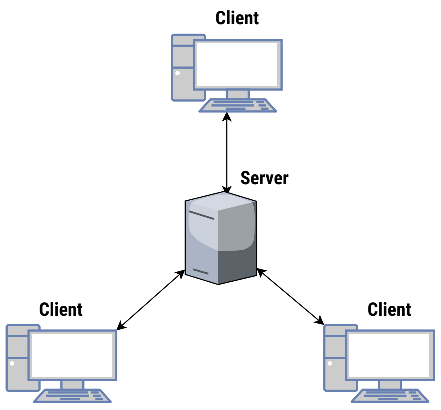


Figure 5.8: Client-server setup

The advantage of the client-server setup is that data is stored centrally on a server, as clients can access the shared data. It can be required to backup data and have security to control who accesses the data. In the client-server setup it is easier to backup the data, as it is all located on the server. Another advantage by having the data centrally stored on the server, is that security can be decided and ensured on the server side, instead of ensuring this on each client. One of the main challenges with the client-server setup is the Single-Point-Of-Failure. This is if a system failure or connection disruption occurs on the server. In this case the system stops working, and therefore a backup server is required, and can be used as a replacement of the erroneous server.

5.3.2 Peer-to-peer

In the peer-to-peer setup there is no centralized server, as each peer will act both as a client and as a server [24, pp. 47-48]. Each peer will store their own data, and the storage, processing and communication loads will be distributed between the peers, and the peer-to-peer setup is therefore a decentralized architecture. The peer-to-peer setup can be seen in figure 5.9.

One of the first applications using peer-to-peer was the file sharing program Napster, which was launched in 1999 [24, pp. 423-428]. Napster was shortly after its release used to share music between users across the world, and therefore ran into legal problems with copyrights.

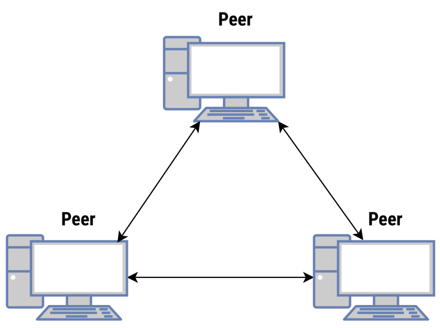


Figure 5.9: Peer-to-peer setup

The main advantage with the peer-to-peer setup is that content and resources are shared between the peers, compared to the client-server setup, where sharing of content and resources is handled by the server. The server can be restricted by the hardware, compared to the peer-to-peer setup where each peer contributes with resources to the system. A disadvantage for the peer-to-peer setup is that the accessibility to shared resources and data needs to be determined on each peer, meaning that security is required to be handled by each peer. [28]. As the data is located on each peer, there is no centralized backup of the data, and each peer therefore requires to backup the data locally. Compared to the client-server setup, a peer-to-peer setup will still be able to work, when a peer experiences a connection disruption or system failure, but the data on the specific peer will be unavailable to the rest of the peers. The difference between the peer-to-peer setup and the client-server setup is described in table 5.1.

Property	**Peer-to-peer**	**Client-server**
Security	Decentralized security. Each peer needs to set up its own security, and access rights for other peers connecting. All peers are able to reach each other on the network	Centralized security. The security and access rights for each client is defined on the server
Availability	The system will continue to work if a peer has a connection disruption or system failure, but the data on the peer will be unavailable	The system will not continue to work if the server has a connection disruption or system failure, and all data will be unavailable (Single-Point-Of-Failure). This can be fixed by using cluster computing (replica servers), which is described in section 5.3.4
Resource handling	Decentralized control (backup on each peer)	Centralized control (backup on server)
Scaleability	Resource scales with the amount of clients	Resource scales with the hardware on the server, resource scaling can be enabled by cluster computing.

Table 5.1: Difference between peer-to-peer and client-server setup

By considering the pros and cons in table 5.1 the decentralized setup in peer-to-peer gives some challenges. The peer-to-peer setup gives high availability, as a client always can access at least one client on the network, even if other clients are down. The problem is to ensure data consistency between clients.

5.3.3 Tiered Architecture

The two-tier architecture is common for client-server setups, and a two-tier and three-tier architecture can be seen in figure 5.10.

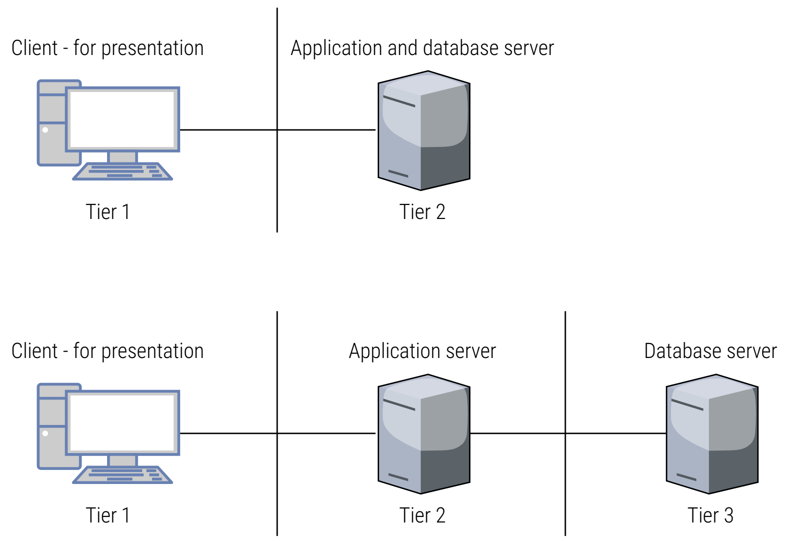

Client - for presentation
Application and database server
Tier 1
Tier 2
Client - for presentation
Application server
Database server
Tier 1
Tier 2
Tier 3

Figure 5.10: Illustration of two-tier and three-tier architectures

The two-tier architecture consists of a client and a server. The server performs application logic and stores and extracts data from the database [24, pp. 51-57]. In the three-tier architecture the application logic and the data logic has been decoupled and placed in each their tier (on each their server in the system). In the peer-to-peer setup these tiers all exist on the client, and this is known as a one-tier architecture. Generally each domain in the application can be mapped on to a separate server, and thereby generating n-tiers, this is also know as the n-tier architecture.

The client can be a thin or a thick client [29]. A thin client is a client that does not handle any application logic or data logic, as these will be handled on the server side. As the thin client does not need to process anything, it sets a low requirement for the hardware. The thick client handles all the application logic, while only letting the server handle the data logic. A web browser is a thin client, requesting a web page from a web server, and thereby letting the server handle the application and data logic.

The main advantage of a thin client, is that it does not set high requirements for the hardware, and the client can therefore be an Iot or mobile device. A disadvantage of the thin client, is that it needs to run everything through the server, and a constant communication with the server is therefor needed. The server will have high requirements for the hardware, as the server needs to perform the application logic and the data logic. The thick client has a higher requirement for the hardware, but the server will also have a lower requirement for the hardware, as some of the logic is moved to the client.

In the two-tier architecture performance decrease when adding multiple clients and therefor a three-tier architecture can be used to give a better performance, by using a separate application server [30]. In the three-tier architecture the client operates as a thin client, as the application and database server handles most of the work. The three-tier architecture is more scalable than the two-tier architecture as each tier can be scaled separately. The three-tier architecture is more secure than the two tier architecture as the client does not have direct access to the database. The security in three-tier architectures is applied on the application server, and does therefor not affect the client, like in the two-tier architectures, where security needs to be considered on the client.

5.3.4 Cluster Computing

A cluster is a system consisting of two or more computers that work together to perform tasks in a system [31]. Cluster computing is often used, as it leads to higher availability, load balancing and better performance.

To achieve higher availability in a distributed system, a High-Availability (HA) cluster is used. The HA cluster replicates data. If one server shuts down, a replicate server will take over, and thereby the system is still available, hence high availability is ensured. If a server receives data, it will eventually need to synchronize itself with the replicate servers, such that the replicate servers are up to date.

Multiple servers can also be added to achieve load balancing and this can lead to a better performance. If one server in the cluster is receiving all the requests from the clients, this server will have a high load. In the load balancing cluster, the load will be spread out between the servers, to optimize the performance in the cluster. The High-Availability and load balancing cluster can be combined to achieve better performance, load balancing and high availability. A cluster based setup can be seen in figure 5.11.

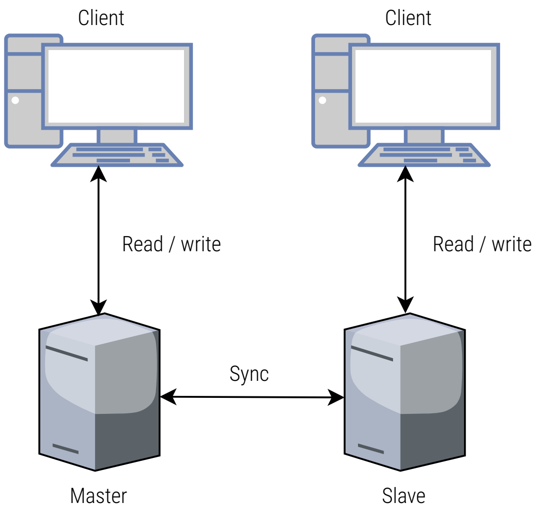


Figure 5.11: Cluster based setup

In a cluster based setup multiple servers can be used to achieve a higher processing capacity, by delegating the work unto multiple servers [32]. Maintenance can be performed without taken down the system, as the slaves can take over the load. In cluster computing it is possible to scale the system, as slaves can be added if the number of clients increase.

5.4 The Publish/Subscribe Messaging Pattern

The scale of distributed systems have changed a lot through the years, and today a distributed system can consist of thousands of components, which could be distributed all over the world [33]. With the growth in scale of distributed systems a demand for more decoupled and flexible systems arise. The publish/subscribe messaging pattern provides the decoupling required in distributed systems, and the pattern are therefor often used in message oriented middleware.

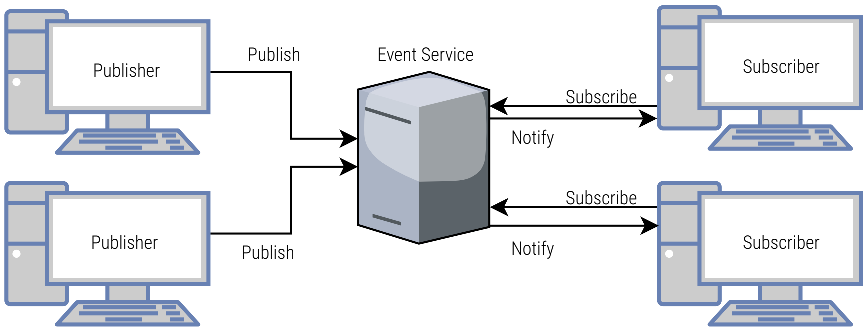


Figure 5.12: Model for a publish/subscribe system (inspired by Eugster [33])

In figure 5.12 a model for a publish/subscribe system is shown. A subscriber can subscribe to certain type of events. The Event Service (aka broker) will store and manage all subscriptions. A publisher will publish a specific event to the Event Service, and the Event Service will thereafter notify all subscribers, that subscribed to the received event. Subscribers can perform functions, while asynchronously listening for a notify event from the Event service. There are three different variations of publish/subscribe, namely topic-based, content-based and type-based publish/subscribe.

5.4.1 Topic-based Publish/Subscribe

In topic-based publish/subscribe the subscribers and publishers can subscribe and publish events to a specified topic, which is described by a keyword. A topic is represented as a string. The most recent improvement to the topic-based publish/subscribe variant is the usage of hierarchies. This improvement makes it possible for a subscriber to subscribe to a topic with sub-topics. When subscribing to the highest topic in the hierarchy the subscriber will also get notified about the subtopics for that topic. This improvement makes it able to organize topics after what they contain and their relationship with each other. Even though this improvement is introduced in modern topic-based publish/subscribe solutions, the limited expression of a topic still holds.

5.4.2 Content-based Publish/Subscribe

Due to the limited expression of a topic, the content-based publish/subscribe variant was developed. In this variation the subscribers subscribe to the content of the events that are published. The events are therefore classified accordingly to themselves and their properties. Subscribers can subscribe to an event by the usage of filters. These filters can be constructed either as a string, a template object or executable code. The string filters are constructed by using comparison operators (=, <, >). These filters are given to the Event Service when subscribing to an event. A string filter is for example a stock price being below or equal to a specific value. Another common approach is the template object filter. Here a subscriber provides a template object *t* to the Event Service. Each event that transforms to the type of *t*, is passed on to the subscriber. At last subscribers can give a predicate object to the Event Service, which makes it possible to filter events at runtime.

5.4.3 Type-based Publish/Subscribe

In the topic-based variation topics often get grouped in events that has some kind of common structure or content. This leads to the type-based publish/subscribe variation, which filters events after which type they are. A hierarchy of events can be constructed by letting classes extend a base class. By letting a subscriber subscribe to the base class, it will be able to receive the subtypes of the base class as well as the base class itself.

5.4.4 The Three Decoupling Dimensions

> *The Event Service ensures space, time and synchronization decoupling between a publisher and subscriber [33].*

5.4.4.1 Space Decoupling

In space decoupling the sending and receiving side does not need to know each other, which is ensured by the Event Service. The Event Service will handle all messages from the Publishers and will know which specific information Subscribers subscribed for. The Event Service will pass messages to any subscribers wanting the specific messages. The principles for space decoupling can be seen in figure 5.13.

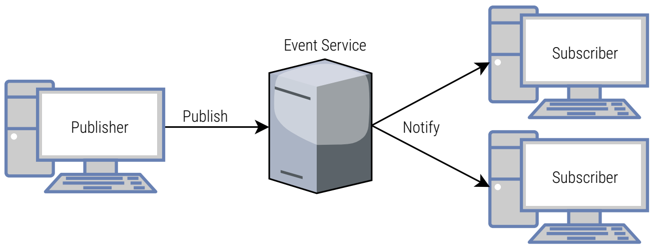


Figure 5.13: Space decoupling in publish/subscribe (inspired by Eugster [33])

5.4.4.2 Time Decoupling

If a publish/subscribe system is time decoupled the sending and receiving side does not need to be active at the same time when publishing messages. In figure 5.14 the first part show a Subscriber which has a connection disruption, but the Publisher still publishes a message to the Event Service. The second part in figure 5.14 show the Subscriber receiving a message from the Event Service, even though the Publisher that published the message is disconnected. Time decoupling is achievable as the Event Service has a storage mechanism for the messages, and is able to send the missing messages, when the Subscriber reconnects to the Event Service.

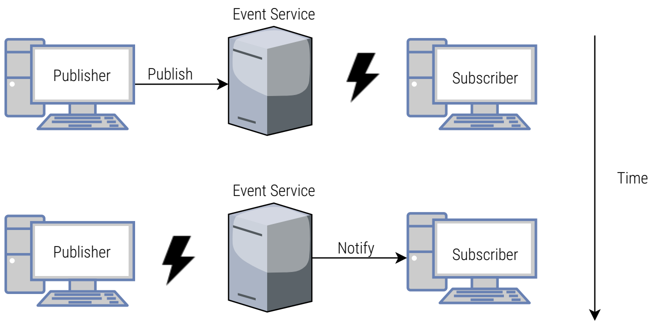


Figure 5.14: Time decoupling in publish/subscribe (inspired by Eugster [33])

5.4.4.3 Synchronization Decoupling

Synchronization decoupling ensures that Publishers publish messages asynchronously, and that Subscribers handles incoming messages from the Event Service asynchronously. This approach ensures that the flow in the Subscriber and Publisher does not get interrupted by a message. The principles for synchronization decoupling can be seen in figure 5.15.

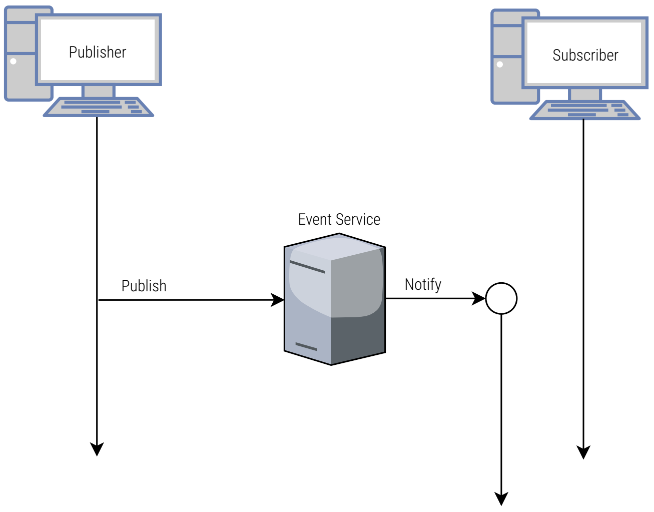


Figure 5.15: Synchronization decoupling in publish/subscribe (inspired by Eugster [33])

5.5 Overview of RabbitMQ

Pivotal, the company behind RabbitMQ, claims that RabbitMQ is the most widely deployed open source message oriented middleware, with more than 35.000 deployments across the world [3]. RabbitMQ has high scalability and availability. RabbitMQ can run on different operation systems, and can be used within a variety of different programming languages.

RabbitMQ supports the three most popular TCP based messaging protocols, which are: The Advanced Message Queuing Protocol (AMQP), the Message Queue Telemetry Transport protocol (MQTT) and the Simple/Streaming Text Oriented Messaging Protocol (STOMP) [34]. Applications using RabbitMQ are able to communicate across the three different messaging protocols, meaning that an application using AMQP can communicate with an application using MQTT, and likewise with STOMP. This is possible, as the RabbitMQ Broker can convert any incoming message to a message of any of the three different messaging protocol types. AMQP is the underlying protocol used in RabbitMQ, with RabbitMQ providing adapters for both MQTT and STOMP.

5.5.1 Advanced Message Queuing Protocol

Two of the main characteristics in AMQP [35] are reliability and interoperability, as AMQP was designed for reliable communication or data exchange between different applications inside an organization. AMQP provide a variety of messaging functions. Some of these functions are reliable queuing, topic-based publish/subscribe and security. AMQP is an approved International Standard (ISO/IEC 19464) while also being an OASIS (Organization for the Advancement of Structured Information Standards) standard.

When a client wants to communicate to a server, the client starts by establishing an AMQP connection, by sending the protocol header to the server [36]. A sequence of three messages is needed to start, tune and open the connection to the server. A two-way channel is needed for communication between the client and server. An AMQP connection can create multiple channels, which can be used for different kinds of communication. Before publishing messages an exchange and a queue needs to be declared, while also having a route / binding between them. The binding between the exchange and the queue can be seen in figure 5.16.

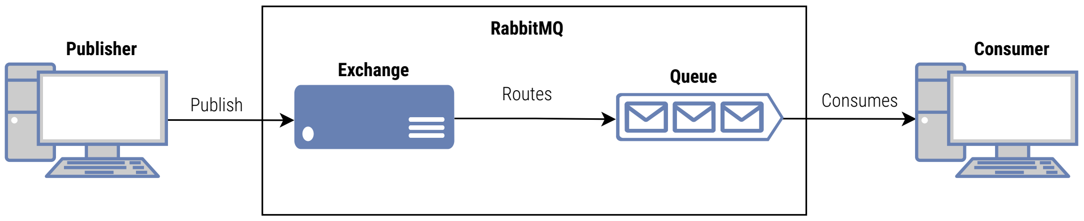


Figure 5.16: AMQP routing in RabbitMQ

The exchange and queues are a part of the RabbitMQ Broker [37]. The exchange is often referred to as a post office, that distributes message copies to queues through the bindings / routes. The publisher push messages to the exchange, while the exchange ensures that the messages gets distributed to the correct queues. The publisher will not know if the messages gets pushed to any queue at all, as the functionality for this lies in the exchange. A binding between an exchange and a queue is created to determine which messages should be pushed to the specific queue. Bindings can be done in multiple ways. One way is to create a direct exchange, where the queues will bind to the exchange, based on a routing key. A commonly used exchange for publish/subscribe systems is the topic exchange, as it is suited for multicast routing of messages.

5.5.2 Message Queue Telemetry Transport

MQTT [38] is designed for resource constrained devices and low bandwidth networks, while still trying to ensure reliability and a degree of guaranteed delivery. MQTT is more simple than AMQP, and it also implements the publish/subscribe pattern, but without the use of queues. MQTT offers three different guarantees of delivery. The first is fire-and-forget, where a sender will send a message, and thereafter forget about it. The fire-and-forget guarantee is unreliable. The next is at least once delivery, where the message is sent minimum one time. The last is the exactly once delivery, where the message will be sent exactly one time.

5.5.3 Simple/Streaming Text Oriented Messaging Protocol

STOMP [39] is a text-based message protocol, while being easy to implement, as it looks a lot like HTTP. STOMP does not contain queues and topics, but it uses a destination string, which the broker will use to map onto an internal queue or topic (destination). The subscribers will subscribe to the different destinations in the broker.

6 | Principles of CAP

6.1 CAP Theorem

The CAP theorem is an invention by Eric Brewer presented at the ACM Symposium on Wednesday July 19th 2000, at the Principles of Distributed Computing (PODC) in the context of web services [40] . CAP stands for Consistency, Availability and Partition Tolerance. All three properties are wanted in a distributed system. The theory itself is about the trade-offs between these three requirements. Eric Brewer states that you cannot have all of them at once in the same distributed system. This has been formally proved by Seth Gilbert and Nancy Lynch of MIT [41].

- Consistency

  What is meant by consistency is that a consistent service operates fully or not at all
- Availability

  Availability essential means that every request will eventually be handled
- Partition Tolerance

  Partition Tolerance refers to the tolerance of inconsistent data in a distributed system

Note: *The assumption is that the system has a cluster based setup.*

The CAP theorem is relevant for message oriented middleware such as RabbitMQ, as RabbitMQ has the option to run in a cluster based setup.

6.2 Eventual Consistency Model

The eventual consistency model was described by Werner Vogels, to point out the difficulties of consistency in large scale distributed systems [42]. In large scale distributed systems partitions are inevitably, and either availability or consistency can therefore not be accomplished at the same time.

There are two main consistency models which is the strong and the weak consistency models, with the eventual consistency model being a variation of the weak consistency model. Strong consistency ensures that every client gets the updates in the same order, and thereby the same value for variables is returned to each client. In weak consistency servers might see different values for the same variable, as a write could have been issued concurrent at one of the servers. With the weak consistency model high availability can be ensured, as a client can read from a server, even though the newest updates have not been applied to the server.

Strong consistency can be ensured on the server by the following expression:

$$W + R > N$$

The variable N is the number of servers that store the data. The variable W is the number of servers that needs to acknowledge the update of a value, before the update is complete. The variable R is the number of servers that are contacted when a read operation is called from a client. The expression formulates that a client can never receive an old value from any of the servers.

Weak consistency can be ensured by the following expression:

$$W + R <= N$$

The expression formulates that a client may risk reading from a server where a value has not yet been updated, and thereby reading an old value. The eventual consistency model is a specific form of weak consistency.

The eventual consistency model ensures that the clients eventually will see the updates made to the server, and every server will therefore eventually return the newest values. The eventual consistency model can be seen in figure 6.1.

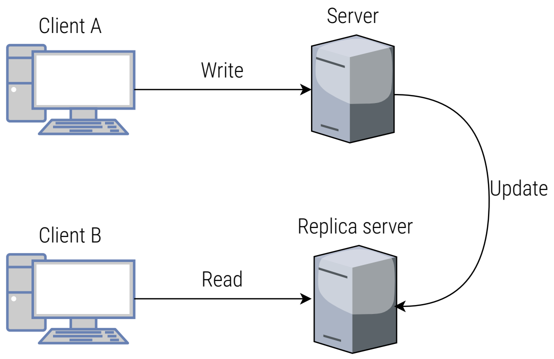


Figure 6.1: Illustration of a system implementing eventual consistency

The eventual consistency model has some variations to consider.

- **Casual consistency:** If client A is causally related to client B, casual consistency require that every client see the writes from these clients in the same order
- **Read-your-writes consistency:** Client A will after updating a value, never see an older value than the one written by itself
- **Session consistency:** Guarantees read-your-writes consistency for a session
- **Monotonic read consistency:** Client B will after reading a value always read the same value or a newer version of the value
- **Monotonic write consistency:** A write from client A on a value will always be completed before another write from client A on the same value
- **Eventual sequential consistency:** Sequential consistency guarantees that the order of writes and reads for each client is kept, but the total ordering of updates can differ [24, p. 777]. An example of this is when two different clients write concurrent, the interleaved total order of the writes on the server is not guaranteed, but the clients order of writes are guaranteed.

Even though the eventual consistency model does not guarantee strong consistency, it works well in practice, and is widely used in different products [43]. The reason for this is because the eventual consistency model often behaves like strong consistency. Different techniques and metrics can be used to predict the behavior of a system, and determine the consistency of a system. The window of inconsistency is used to determine the inconsistency of a system. The window of inconsistency is the time it takes for a write from a client, to become available for reading to other clients. The Probabilistically Bounded Staleness (PBS) tool [44] can be used to provide an answer to, how consistent the eventual consistency model is. The PBS tool showed that LinkedIn's data storage returned consistent data 99.9 percent of the time and Yammer's had an inconsistency window around 202 milliseconds [43]. Eventually consistent solutions are often faster than strongly consistent solutions, and the consistency is often adequate for most systems.

6.3 Replication

Replication is used to replicate data on multiple servers, this makes it possible to achieve high availability and fault tolerance in distributed systems [24, pp. 765-768]. Often users of the system needs the servers or the services they provide, to be available, and therefore high availability and fault tolerance is needed. By using slaves, when a server has a connection disruption or system failure, the client can access one of the slaves. The availability for a system can be calculated with the following expression:

$$1 - probability(all\ servers\ have\ a\ connection\ disruption\ or\ system\ failure) = 1 - p^n$$

The independent probability p is the probability for a server to have a connection disruption or system failure, while n is the number of servers. For example having an independent probability p of 2%, and two servers the availability is calculated to be $1 - 0.02^2 = 0.9996 = 99.96\%$.

The two replication techniques which are described in the literature are passive replication (master-slave) and active replication [24, pp. 778-782].

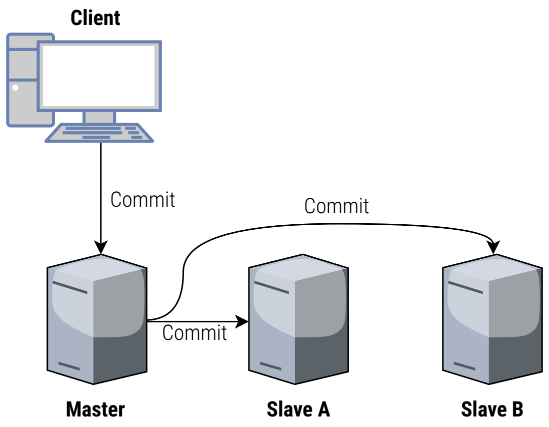


Figure 6.2: Passive replication model

Passive replication can be seen in figure 6.2. In passive replication all clients will only commit their changes to the master, and the master will thereafter commit the changes to all the slaves.

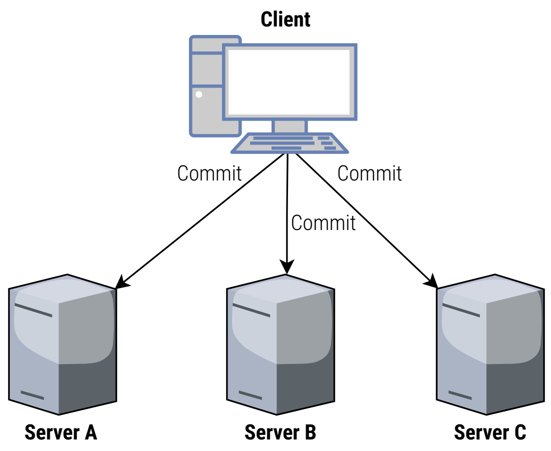


Figure 6.3: Active replication model

Active replication can be seen in figure 6.3. In active replication the client commits the changes to every server in the system. A broadcast is made from the client to the servers. If a server does not respond or acknowledges a commit, the server is most likely having a connection disruption. The process running on each server needs to be deterministic in active replication, meaning that a given input will always return the same response. As each process is deterministic, the active replication model is Byzantine fault tolerant.

A Byzantine fault can be described by generals trying to coordinate an attack, but one of the generals is a traitor, and therefore tries to corrupt the attack by telling the commander to retreat [45]. In active replication the commander is the client committing an update to the servers, and the servers in this case are the generals trying to coordinate an attack. Each general will make the same observations and thereafter tell the commander whether or not to attack. It is expected that all generals will return the same result to the commander. For the commander to get a valid answer from the generals, at least two-thirds of the generals needs to be loyal to the commander. For each traitor among the generals, at least two loyal generals are needed. This gives the following expression to how many generals are needed, given a specific amount of traitors m:

$$3m = total\ generals$$

Given a single traitor m a total of three generals are needed, for the commander to receive the correct result. In figure 6.3 a setup with one client and three servers can be seen. The setup can handle one server returning corrupted data. If the setup only consisted of two servers, and each server returned different values, the client would have no chance to decide which value is the correct one.

In fault tolerant systems using passive and active replication, updates are committed to servers as soon as the updates are available, which also is called an eager approach. In passive and active replication all servers need to be updated and reach an agreement before responding the client, which is not optimal when availability is required.

The two replication techniques described guarantees strong consistency. To support weak consistency or eventual consistency, other techniques has to be used. Bayou is a system that guarantees eventual sequential consistency.

6.3.1 Bayou

Bayou was created as a storage system for mobile devices, and was created to handle frequent connection disruptions [46] [24, pp. 792-794].

The problem with multiple servers in a distributed system is that clients can update the same value on different servers and a conflict can occur. Clients can also have a connection disruption from the server as networks are unreliable, and changes made on the client is therefore first committed to the server when reconnecting. The local changes made while having a connection disruption might give a conflict when committing, as another client could have made similar changes to the same object as the client experiencing a connection disruption.

In Bayou, clients can update data locally, which is a partition of the system. at some point later in time when the client is connected to the majority partition of the system, the client can commit the updates, and a conflict detection and resolution makes it possible to merge these updates.

Bayou uses a variety of passive replication, as there is a master which decides the order of updates, and propagate this order to the rest of the slaves. The difference from passive replication and replication in Bayou, is that clients are not forced to only communicating with the master. This is because of the tentative and committed log that enables slaves to handle updates.

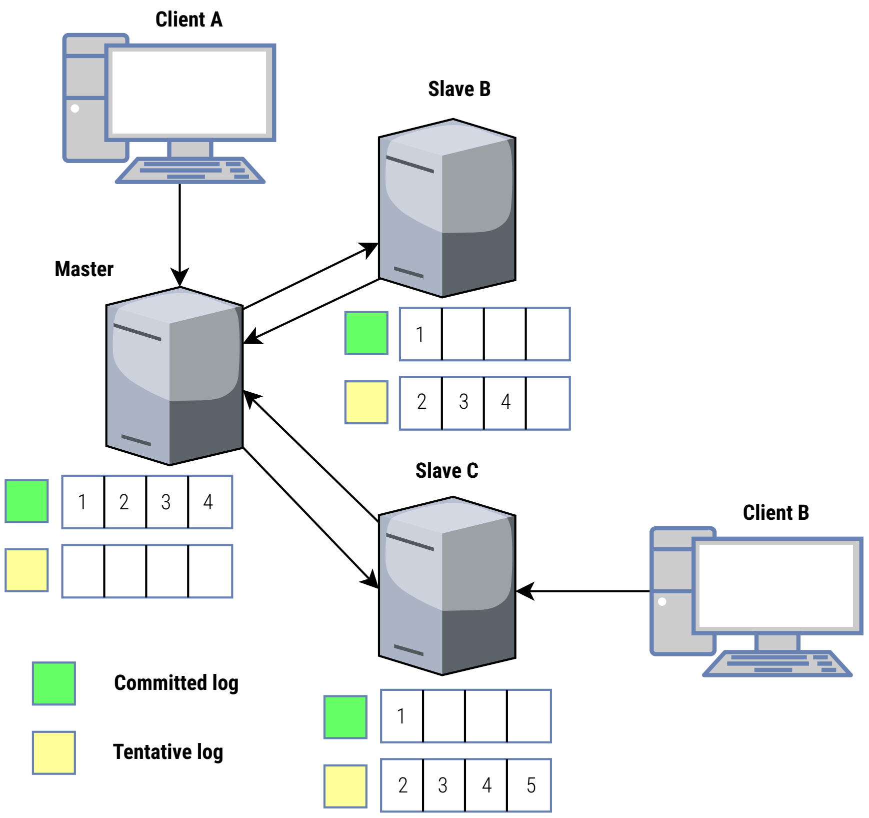


Figure 6.4: Illustration of replication in Bayou

When updates are first applied they are marked as being tentative, and these can be undone and reapplied if a conflict occurs. A master is used to decide which updates should be moved from being tentative to being committed. The master receives the tentative updates and thereafter tells the slaves, which updates are now considered as committed. When updates are committed their order can not be redone. When a server receives a new tentative update, that update might conflict with an already existing tentative update, and updates therefore also consists of a dependency check and a merge procedure. The dependency check and merge procedure is domain specific for each update. In some cases the merge procedure might not find a viable solution, and the system will indicate that an error has occurred. Bayou clients and servers with tentative and committed logs can be seen in figure 6.4. The Master has committed updates one to four, and is now able to inform the rest of the slaves of the committed order of updates. Client B has send a new tentative update to Slave C, and this tentative update is therefore added to the tentative log, and ready to be send to the Master to be committed. Bayou uses a gossip architecture to replicate data between servers.

An application for usage of Bayou could be a shared calendar, where clients for example can arrange meetings. A client experiencing a connection disruption can create a calendar entry, but meanwhile another client made a similar calendar entry at the same time. When the client reestablish the connection, and commits the calendar entry, the server needs to handle the conflict, as another client already posted a calendar entry at the same time. Multiple times can be selected as viable meeting times, and thereby give the server several opportunities when trying to merge the calendar entry.

Bayou provides high availability while also providing eventual sequentially consistency. One could think that Bayou actually beats the CAP theorem, but Bayou does only eventually ensure sequentially consistency, and therefore users have to wait to achieve sequentially consistency. The disadvantage of Bayou is that it can be complex, as the developer has to provide dependency checks and merge functions to cover every potential conflict. Users need to be made aware of which updates are tentative and committed, as their update might be changed because of the merge function. Even though Bayou is a distributed database system and HarakaMQ is a message oriented middleware, the concept of how the ordering of events works can be applied and used in HarakaMQ.

6.3.2 The Gossip Architecture

The gossip architecture is based on the essence of gossiping in real life [24, pp. 783-792]. For example in a firm John gets his coffee every half an hour, and when he goes to the coffee machine he talks with other people, and tell them some gossip within the office. The people John talks to at the coffee machine, will after meeting other people at their next coffee machine visit, also tell those people the gossip. In the end every co-worker at the firm will know about the gossip. Compared to real life gossiping, a server sends out periodically gossip messages, to spread new information or updates in the system. When the other servers get the updates, they will spread it to random neighbor servers, until all servers have the updates. This is also known as the rumor-mongering gossip architecture type.

Besides the rumor-mongoring gossip architecture there is also the anti-entropy architecture. The anti-entropy architecture is used to repair replicated updates between servers. It does this by reconciling the differences of the order of the updates on the servers. The difference between the rumor-mongering protocol and the anti-entropy protocol, is that in the rumor mongering protocol updates are spread throughout the system, but with no indication of which updates came in first [47]. In the anti-entropy protocol timestamps are used to merge new updates between replica servers, and thereby giving an indication of which order the updates should be applied in. An example of how the anti-entropy protocol can be used is Bayou.

A timestamp is required when using an anti-entropy protocol to determine the order of updates across servers [24, pp. 783-792]. The timestamp can be created by using either physical clocks. An example of a physical clock protocol is the Precision Time Protocol (PTP) [48], [49] or Network Time Protocol NTP [50]. PTP and NTP can synchronize clocks in a system from a local master. One of the disadvantages by having a physical clock is the need for clock synchronization between servers.

Another approach is to use logical clocks to provide the timestamps, an example of a logical clock is a vector clock [51]. The problems with logical clocks is that they can only provide partial ordering between concurrent updates.

In the literature two different gossip-styles are described, namely the epidemic [52] and the round-robin [53] gossip-style. An epidemic gossip-style can be used as seen in figure 6.5.

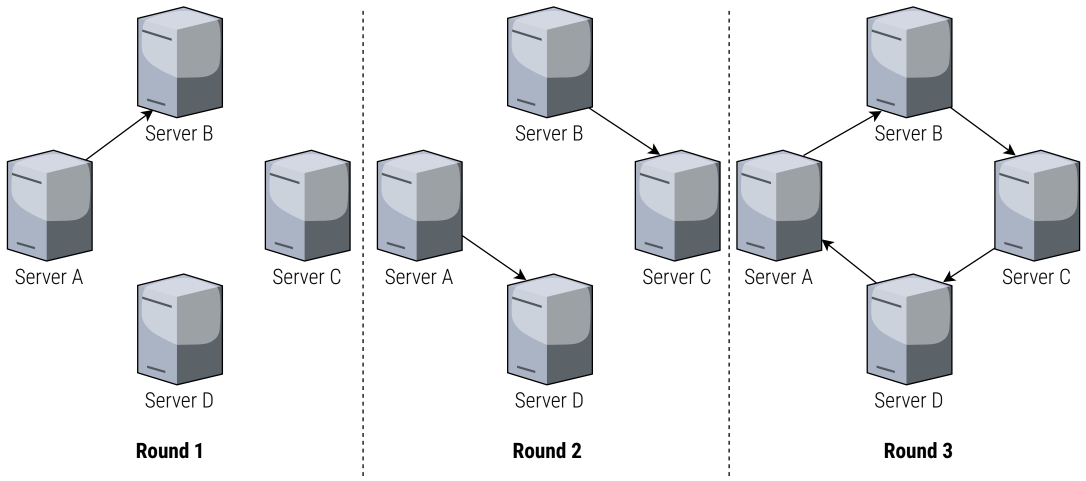


Figure 6.5: Illustration of epidemic gossip-style

The epidemic gossip-style spreads new updates to servers through out the system, as a virus spreads from human to human [52]. In round 1 a server is infected with the updates, and therefore selects a random neighbor server, and commits its updates to this server. In round 2 two servers are infected, and they will each select a random neighbor to send the updates to. In round 3 all servers have the updates, and they each send out the updates to random neighbors.

The expected amount of rounds before an update, has infected all servers, can be divided into three phases [47]. In the first phase each infected server will send a message to infect servers which is not already infected, and thereby doubling the number of servers infected. The second phase will only last a constant number of rounds. In the last and third phase there will only be few severs left uninfected, thereby making it more difficult to infect these servers, as servers are chosen randomly. The estimation of number of rounds before convergence are expressed by the three before mentioned phases in equation 6.1 [54].

$$T(n) = log_2(n) + O(1) + ln(n) = O(log(n)) \tag{6.1}$$

The expected number of rounds for convergence in a system with n number of servers is $O(log(n))$.

Another gossip-style is the round-robin gossip-style, which can be seen in figure 6.6.

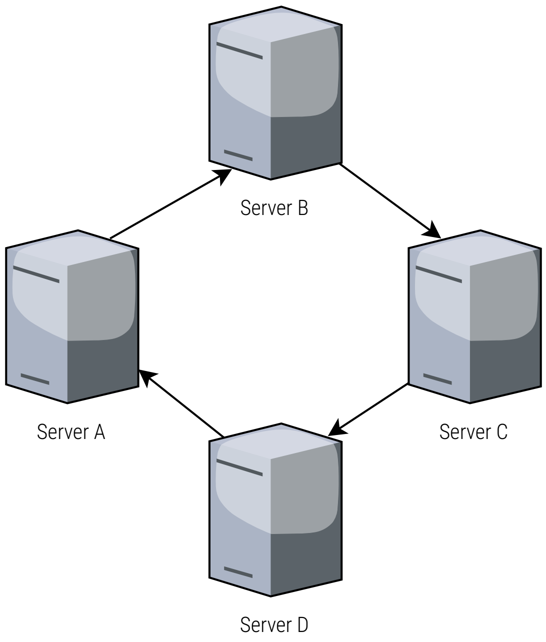


Figure 6.6: Illustration of round-robin gossip-style

In the round-robin gossip-style each server has a preset neighbor server which it sends its updates to [53]. The updates will start at one server, and will after n number of rounds return to the server that started the round, indicating that the updates have been shared with the other servers. The disadvantage of the round-robin gossip-style is that the servers have preset neighbors, and if a server disconnects, the updates will not continue to be send to the rest of the servers.

The number of rounds for an update to spread to all servers in the system, with the round-robin gossip-style, is linear dependent with the amount of servers n in the system, and is expressed as $O(n)$.

The gossip architecture is used to create highly available servers, and is intended to deliver a weaker consistency between the servers [24, pp. 783-784]. The gossip architecture guarantees that eventually all the servers will receive all the updates. The gossip architecture can be used to achieve sequential consistency if needed, but this is not the primary intended use. By using a lazy approach and sending gossip messages periodically, the gossip architecture is not suited for real-time systems. When adding more servers to the system, the amount of gossip messages before convergence will increase.

7 | Design and Implementation of HarakaMQ

7.1 Conceptual Architecture Description

State of the art message oriented middleware such as RabbitMQ can use both a client-server setup and a cluster based setup for scalability. HarakaMQ is required to handle reliability and support publish/subscribe, therefore HarakaMQ supports a client-server setup as well as a cluster based setup. The peer-to-peer setup was not implemented, as the clients would be too thick.

In the client-server setup multiple clients can connect to one server. In the cluster based setup multiple servers are added to run in a cluster. When running in a cluster setup replication are required between the HarakaMQ Brokers, to ensure that every message eventually will be delivered.

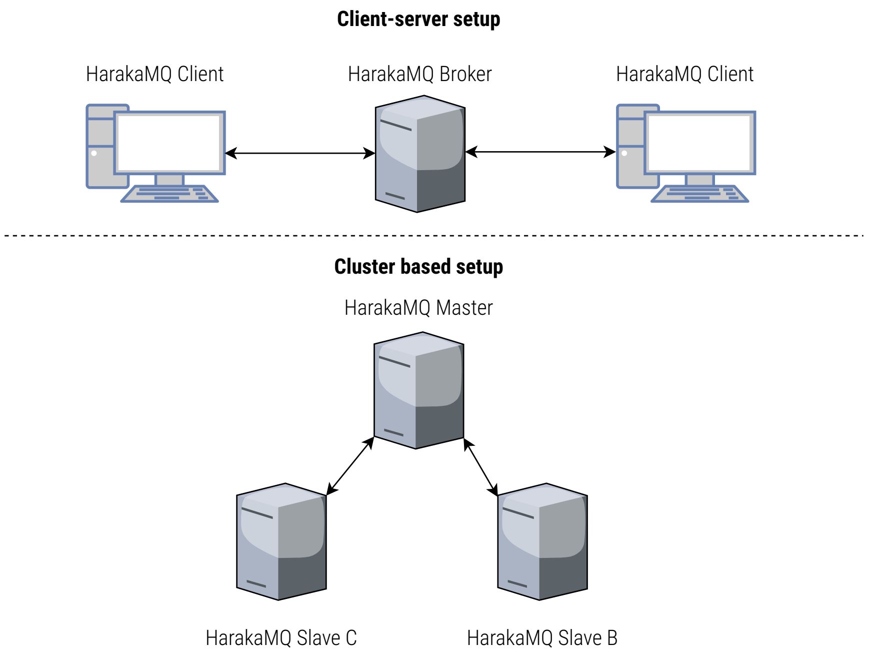


Figure 7.1: Architecture overview of HarakaMQ in either a client-server or a cluster based setup

In figure 7.1 an overview of the developed systems two different setups can be seen. The developed solution consists of a HarakaMQ Broker and a HarakaMQ Client, which communicates via UDP. A HarakaMQ Broker can either be a HarakaMQ Master or HarakaMQ Slave dependent on the configuration. HarakaMQ is implemented using the topic based publish/subscribe message pattern. This enables the solution to handle all three decoupling domains: space, time and synchronization. The time decoupling is especially important for HarakaMQ, as it is required to ensure message delivery guarantee when the subscriber is for example having a system failure, connection disruption or is in sleep mode. Each HarakaMQ Client can either publish messages to a topic, or subscribe to a topic. The HarakaMQ Client is created as a thin client. The reason for this is that the solution is required to run on resource constrained devices. The HarakaMQ Broker acts as a server, and handles all published messages, and ensures that each subscriber receives the messages.

7.2 Programming Language and Frameworks

7.2.1 .NET Framework and C#

HarakaMQ is developed with the programming language C#, and the framework .NET Core 2.0 [55]. C# is one of the most popular programming languages in the world, based on the TIOBE index 2017 [56]. Some of the advantages of the .NET Core 2.0 framework and C# is that it is: strongly typed, open-source, it has LINQ, Generics, the asynchronous programming and the possibility to run the application on Windows, Mac OS and Linux. The possibility to run the application on this number of operating systems is appealing for a system like HarakaMQ, as it gives the capability to distribute the system to a big variety of devices. This also makes it possible to have communication between different operating systems.

The performance of .NET Core 2.0 is not as good as the performance of C++ [57], which is a low-level programming language. Compared with Java version 1.8, which is a high-level programming language like C#. C# .NET Core 2.0 exceeds Java version 1.8 in 6 out of 10 categories [58]. One category that C# .NET Core 2.0 loses in, is the regex calculation, but this is okay for HarakaMQ, as it does not require to perform regex calculations.

C# .NET Core 2.0 is a high-level programming language for cross-platform applications with the asynchronous programming capabilities that is suited for a distributed system like HarakaMQ. The performance is better than alternatives like Java version 1.8, but the performance is still not as good as a low-level programming language like C++. Despite the performance difference the choice is still C#, because it is a high-level programming language providing better performance than Java 1.8.

7.2.2 Events in .NET

Events are used widespread through the HarakaMQ implementation. The events are used to raise notifications. An example of this is when a message is received the subscribers will be notified about this, and then the subscribers can consume the message. Events in the .NET framework uses the delegate model, and the delegate model uses the observer design pattern [59]. The advantage of using events which uses the observer design pattern, is that events can be used through interfaces, and thereby the software can be made more decoupled. The delegate model is a type in .NET. The type holds a reference to the method it is associated to. This reference can then later be used to call the method. A delegate is equivalent to a function pointer or a callback known from programming languages like C++ and JavaScript.

The .NET framework supports both static and dynamic event handlers. A static event handler lives as long as the application lives. The dynamic event handlers can be hooked on/off dynamically based on run-time conditions. All event handlers used in HarakaMQ are used dynamically. The advantage of only using dynamic event handlers is that it gives the possibility to have multiple instances of the HarakaMQ Client with different ports in the same application.

By using events, polling is not required. This is positive as it loads the CPU less [60]. This is especially an advantage for devices which for example have energy constraints or limited CPU-power available.

7.3 Implementation of HarakaMQ

HarakaMQ consist of modules, which furthermore consist of components. The relationship between components and modules in figure 7.2 and 7.6 are one-to-one relationships, unless otherwise stated. The faded components in these figures are not implemented in HarakaMQ.

7.3.1 HarakaMQ Broker

The HarakaMQ Broker is as an event service created to handle subscriptions of HarakaMQ Clients and the routing of publish messages. In figure 7.2 it can be seen that a Business logic module layer has been added on top of the UDP Communication module layer. All communication goes through the UDP Communication module, so the Business logic module layer is not responsible of the delivery and receiving of a message.

The components in the Business logic module layer have different responsibilities, which can be seen in table 7.1.

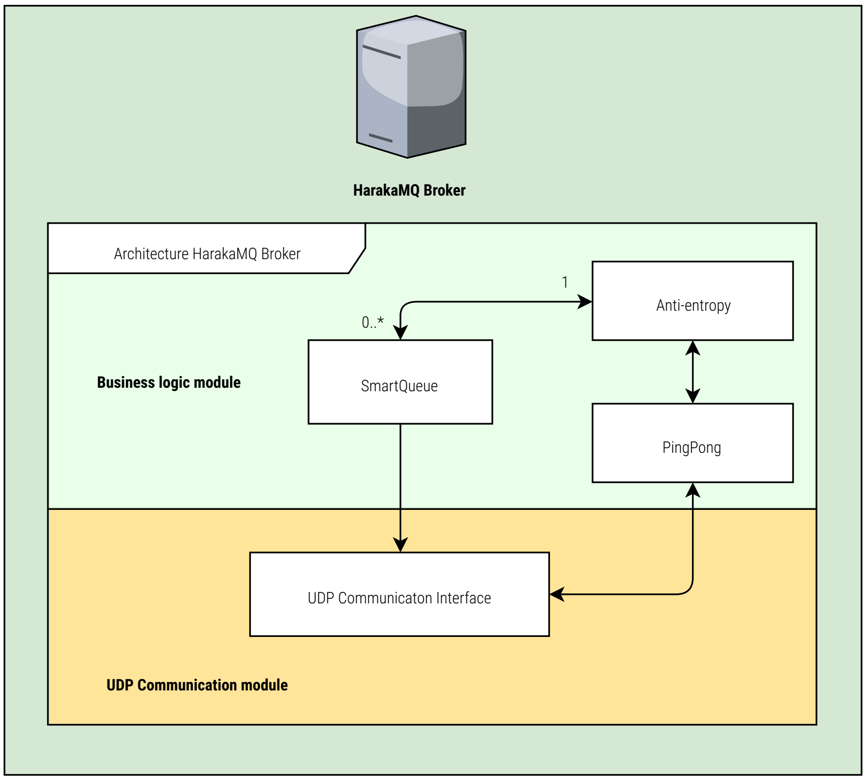


Figure 7.2: Layered architecture for the messagebroker

Component name	Description
SmartQueue	Represents a topic, and is responsible of publishing messages and handling subscribers
Anti-entropy	Handles committed and tentative logs
PingPong	Handles the gossip communication between HarakaMQ Brokers in a cluster based setup

Table 7.1: Description of the components in the Business logic module layer

7.3.1.1 SmartQueue

The SmartQueue component is self-contained, and contains a topic with events and a task [61] to consume the events. An event is a data object which contains information of how it should be consumed, and should not be confused with .NET events. A task is an abstraction on top of an OS Thread. A task can either be handled asynchronously on the thread pool [62] or be assigned to an OS Thread. A SmartQueue's responsibility is to handle the subscription of HarakaMQ Clients. When a HarakaMQ Client publishes a message to a topic, the SmartQueue will distribute the message to the subscribers of the topic. Topics are persisted on the disk, so that messages are not lost, if a system failure occurs.

The task which is associated to the SmartQueue do not run all the time. The task is created and started when an event is added to a topic. This is seen in listing 7.1.

```
private void EventAdded(object sender, EventArgs e)
{
        if (_eventConsumerTask == null ||
            _eventConsumerTask.IsCompleted)
                _eventConsumerTask =
                    Task.Factory.StartNew(ConsumeMessage,
                        TaskCreationOptions.LongRunning);
}
```

LISTING 7.1: Code snippet of the EventAddedToQueue method from the SmartQueue.cs implementation

When the task has emptied the topic of events, the task has completed its job, and the task is stopped and deleted. The SmartQueue does not do anything until a new event is added to the topic. The task creation option *TaskCreationOptions.LongRunning* hints the task scheduler that additional threads might be needed so the thread pool does not get blocked.

A consumption of an event can result in a variety of different changes. An example of an event could be an event publishing committed or tentative messages to HarakaMQ Clients, or subscription of a HarakaMQ Client. When an event has been consumed it is removed from the topic and the disk.

Events are supposed to be as small as possible because they have to be persisted on the disk.

7.3.1.2 Replication

Replication of the SmartQueues is used to guarantee, that if a HarakaMQ Broker is down, the messages still exist in the system. This is only enabled in a cluster based setup. This makes it possible to distribute messages and subscriptions to the different HarakaMQ Brokers.

The replication of the SmartQueues ensures that no matter what HarakaMQ Broker a HarakaMQ Client has subscribed to, it will eventually receive the messages published to the subscribed topic, even though that the message might have been published to a different HarakaMQ Broker.

An anti-entropy protocol is developed to perform the replication between the HarakaMQ Brokers. The developed anti-entropy protocol has been named PingPong and is inspired by the way Bayou replication is done. The theory of Bayou can be read about in section 6.3.1. The PingPong protocol applies the gossip architecture. In this thesis a broadcasting gossip-style has been proposed and compared to existing gossip-styles such as the epidemic and round-robin gossip-style.

The complexities for the epidemic, round-robin and broadcasting gossip-style is shown in figure 7.2.

Description	**Epidemic**	**Round-robin**	**Broadcasting**
Rounds before convergence	O(log(n))	O(n)	O(1)
Messages sent per broker	O(1)	O(1)	O(n) - for the master

Table 7.2: Complexities for different gossip-styles

in figure 7.3 the number of rounds before convergence for epidemic, round-robin and broadcasting gossip-style is shown.

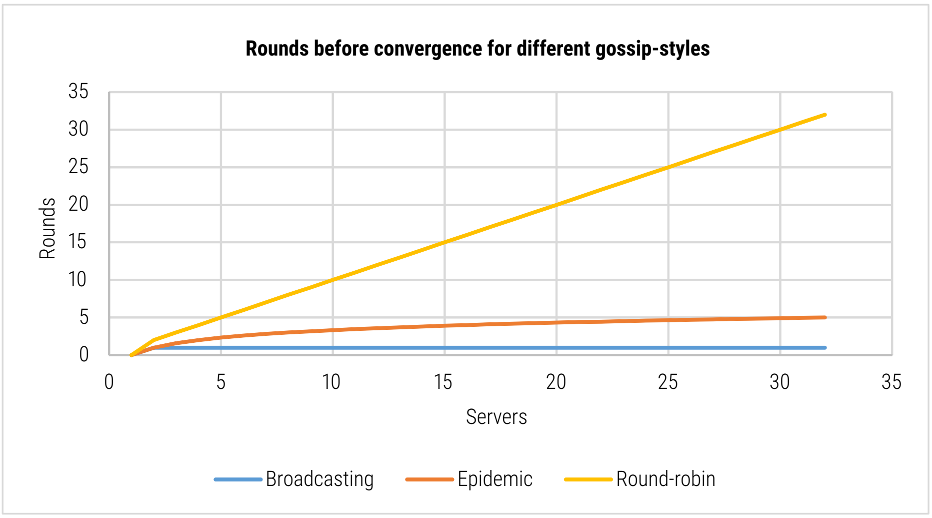


Figure 7.3: Convergence graph for the different gossip-styles described in this thesis

In figure 7.3 it can be seen that for a system with 32 servers, the round-robin gossip-style will converge after 32 rounds, and the epidemic gossip-style converging after 5 rounds. The broadcasting gossip-style is used as the number of rounds contra servers are constant, which results in the same amount of rounds for an update when adding more servers.

The reason to use a broadcasting gossip-style is that a master is required in HarakaMQ to determine the order of updates, and the fault tolerance is required. Fault tolerance is difficult to enforce in the round-robin gossip-style unless brokers are excluded when they do not respond. A complex problem in the round-robin gossip-style is to include a broker when it needs to be included again.

The problem with the epidemic gossip-style is that a master is required in HarakaMQ, so a specific broker is required to communicate with all the slaves. In the epidemic gossip-style it is random who communicates with who, it is therefore not guaranteed that the master has communicated with all slaves after convergence. This guarantee is needed to guarantee fault tolerance. The only guarantee epidemic gives is that eventually all brokers has received the update, but there is no guarantee who the update is received from.

The PingPong protocol sends out an anti-entropy message (aka a ping) at a fixed preconfigured time interval. Each anti-entropy message marks the start of a new anti-entropy round. The fixed anti-entropy round interval makes it possible to determine if a HarakaMQ Broker has a connection disruption or system failure. A HarakaMQ Broker could be deemed as having a connection disruption for example if it has not answered any of the last five anti-entropy rounds.

The PingPong protocol consists of two logs, a committed log and a tentative log. A log consists of messages from multiple topics. Both logs are sorted, by the timestamp of messages. The committed log cannot be changed when it has been set. It is the responsibility of the HarakaMQ Master to commit messages when they are stable. A message is stable after two rounds of anti-entropy, and this is because all HarakaMQ Slaves need to receive the message before the HarakaMQ Master can commit the message. An anti-entropy round has finished when all HarakaMQ Slaves have answered the ping from the HarakaMQ Master. When a message is committed it is removed from the tentative log, and when all HarakaMQ Brokers have delivered the committed messages to the HarakaMQ Clients, the committed messages are purged, so they do not clog the SmartQueues.

The tentative log consists of messages which are not yet committed. HarakaMQ Clients can receive tentative messages if they subscribe to them. The default for the HarakaMQ Clients is that they only receive messages from the committed log. The tentative messages can be rearranged by any HarakaMQ Broker. The reason for this is that if older messages, than the ones already in the tentative log, is received from another HarakaMQ Broker, the tentative log can be rearranged to fit the newly arrived messages. The tentative log is separated into two different logs at each HarakaMQ Broker, namely the *OwnTentaiveMessages* log and the *ForeignTentativeMessages* log. This is done to separate the tentative messages from other HarakaMQ Brokers, and tentative messages that are received from the HarakaMQ Brokers own HarakaMQ Clients. This approach results in less search time when searching for messages that have to be shared with other HarakaMQ Brokers.

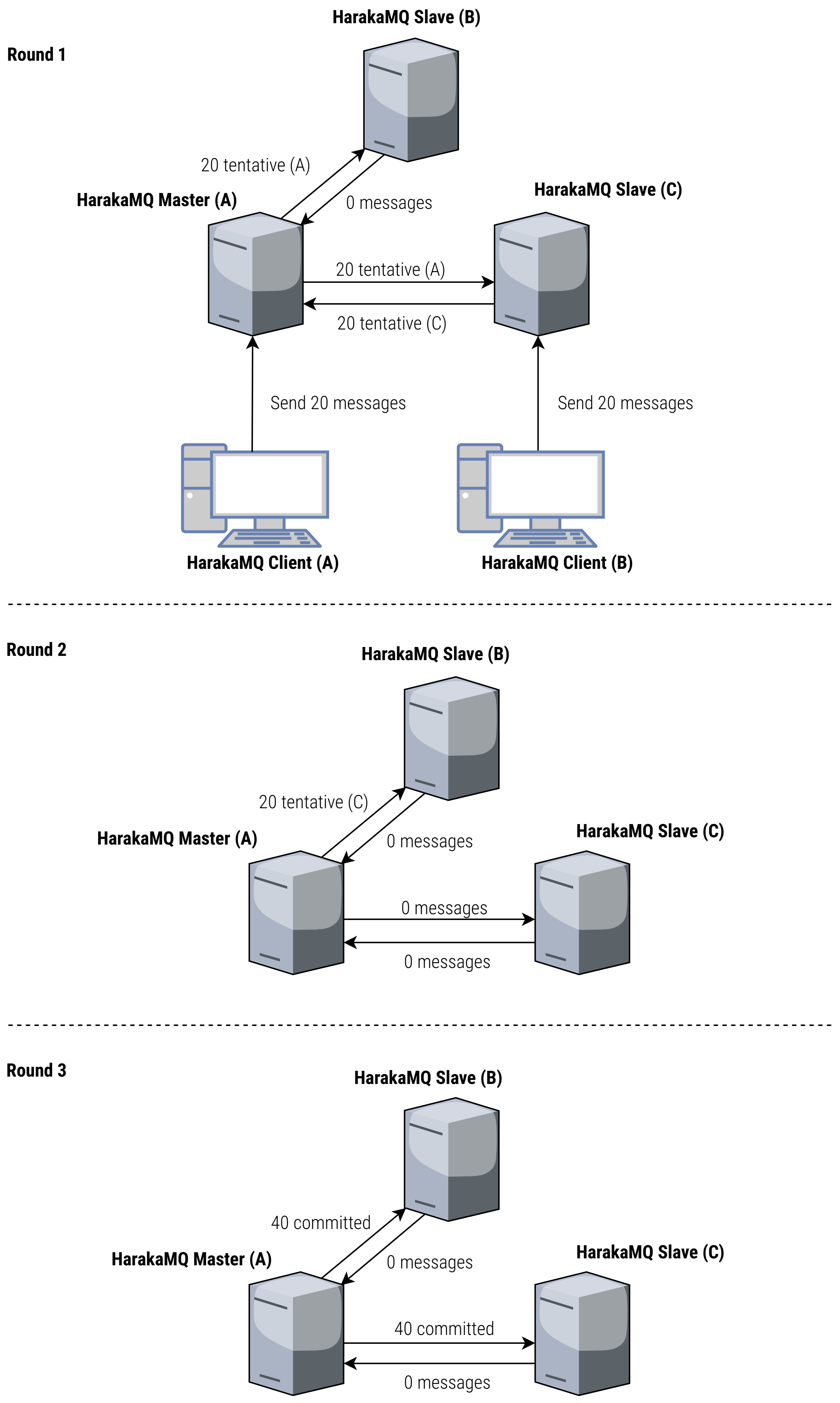


Figure 7.4: Illustration of the anti-entropy protocol PingPong over 3 rounds of anti-entropy

Figure 7.4 illustrates 3 rounds of anti-entropy in the PingPong anti-entropy protocol. At Round 1 20 messages are published from two different HarakaMQ Clients to two different HarakaMQ Brokers. The messages are then attached to the first ping from the HarakaMQ Master (A), and they are added to the tentative log. The 20 messages published to HarakaMQ Slave (C) is added to the response to the ping from the HarakaMQ Master (A).

At Round 2, HarakaMQ Slave (B) is missing the 20 messages from HarakaMQ Slave (C). These messages are attached to the ping from the HarakaMQ Master (A). There is no new information to HarakaMQ Slave (C) in Round 2.

At Round 3 the messages received in Round 1 is stable because all HarakaMQ Brokers have seen the messages. The messages are then committed and passed to the two HarakaMQ Slave (B) and (C). At this point all HarakaMQ Brokers agree on the order of the messages.

The ping from the HarakaMQ Master is sent as unicast as this is what the UDP Communication module supports in version 1.0.0 of HarakaMQ. But it could be an advantage to explore the multicast feature of UDP, and see if this yields an improvement of the throughput of the communication between the HarakaMQ Brokers.

When implementing the anti-entropy solution, a consideration has to be taken into account, which is the limit to the size of the payload a UDP message can contain. The theoretical limit is 65,535 bytes [63], but the actual limit is (65,535 - 8 (UDP-header) - 20 (IP-header) = 65,507). This limit needs to be taken into consideration when bundling committed and tentative messages in the ping messages. To handle this issue a limit was set to how many committed and tentative messages a ping message could contain.

In version 1.0.0 of HarakaMQ fault tolerance in a cluster setup is not fully developed and tested. But the foundation to handle system failures 1.6.2, and with the addition of a leader election algorithm and a Dynamic Router, a fully reliable system can be put into production. More exploration of the replication approach could improve the solution, for example an implementation of a consensus protocol could help the HarakaMQ Master to commit messages faster than the current implementation. A naive example of a consensus protocol could be that only two-thirds of the HarakaMQ Slaves needs to receive the tentative messages before a commit is performed.

7.3.1.3 Precision Time Protocol

It is imperative that the HarakaMQ Brokers in a cluster based setup are synchronized and precise. There are two approaches to synchronize the HarakaMQ Brokers. They can either be synchronized with logical clocks or physical clocks. Logical clocks will not work for HarakaMQ, as it is difficult to determine the *Happened-before* relation [64] between messages. This is because messages are published concurrent, and thereby the granularity has a correlation with the preconfigured time interval of the anti-entropy protocol. To achieve a finer granularity the preconfigured time interval should be as low as possible, but the downside is that more coordination messages are required. A solution to get a better granularity without lowering the preconfigured time interval is to use physical clocks to determine the order of messages in HarakaMQ.

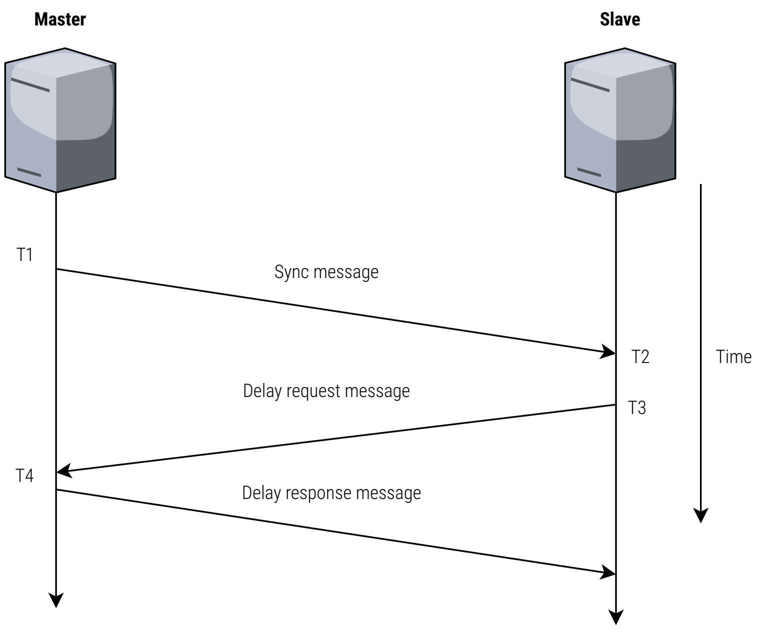


Figure 7.5: Illustration of Precision Time Protocol clock synchronization approach in HarakaMQ

To synchronize HarakaMQ Brokers a software version of the Precision Time Protocol (PTP) [48], [49] has been implemented. The reason that PTP was chosen over NTP, is that NTP is designed to pull the time from the Master, where PTP is designed to push the time from the Master. This adhere with the chosen gossip architecture in HarakaMQ, which broadcasts messages from the master to the slaves.

The implemented version does not use hardware timestamps. In figure 7.5 it is illustrated how the synchronization of a clock is performed in HarakaMQ. First the Master to Slave difference is required. This is done by sending a Sync message which contains the timestamp $T1$ of the Master. When the Sync message is received at the Slave, the Slave then takes a new timestamp of when timestamp $T1$ is received at the Slave. This timestamp is called $T2$.

Second the Slave to Master difference is required. This is the same procedure as the Master to Slave difference. The timestamp $T3$ is taken when the Delay request message is sent from the Slave, and $T4$ is when timestamp $T3$ is received at the Master. The timestamp $T4$ is then returned to the Slave to calculate the offset between the Master and the Slave. When the Slave have all the timestamps $T1$, $T2$, $T3$ and $T4$ the offset between the Master and Slave can be found by equation 7.1.

$$Offset = ((T2 - T1) - (T4 - T3))/2 \tag{7.1}$$

In version 1.0.0 all HarakaMQ Slaves are synchronized at startup by the HarakaMQ Master. With further enhancement of the solution, this can be upgraded to synchronize continually throughout the lifetime of the system.

7.3.2 HarakaMQ Client

In figure 7.6 it is shown that the HarakaMQ Client's Application layer module sends and receives messages through the UDP Communication Interface. The components in the Application layer module are derived from the application layer from RabbitMQ's .NET client [65], thereby given the ConnectionFactory, Connection and Model components. The underlying functionality differs, meaning an application using the RabbitMQ .NET client, can use the HarakaMQ Client, but the settings between the two Application layer modules can be different.

In HarakaMQ the ConnectionFactory initializes the Connection component, which thereby configures the UDP Communication module through the interface with the settings passed by the application developer. The connection then creates a Model which starts to listen for incoming messages. Besides listening for messages the Model can also pass messages to the UDP Communication Module.

The HarakaMQ client can consume messages and publish messages to a topic. Before consuming messages from the topic, the HarakaMQ Client has to declare a consumer (subscriber) to handle the ingoing messages.

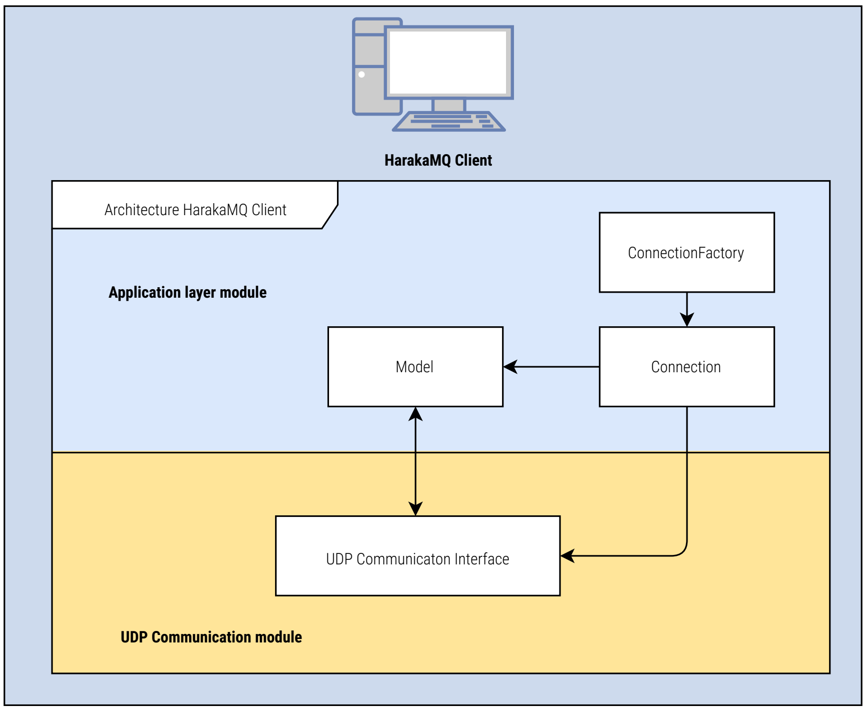


Figure 7.6: Layered architecture for the client

When a message is received on the HarakaMQ Client, a publish event is invoked from the UDP Communication module layer to the Application layer module, where the event is handled. The event consist of a message that is to be delivered and handled by the consumer. The components in the Application layer module have different responsibilities, which can be seen in table 7.3.

Component name	Description
Model	Is responsible of sending and receiving messages through the UDP Communication Interface
ConnectionFactory	Is responsible of creating a Connection
Connection	Is responsible of configuring the UDP Communication module and to create a Model

Table 7.3: Description of the components in the Application layer module

7.3.3 UDP Communication module

The UDP Communication module is created to ensure that the communication in HarakaMQ is reliable. UDP is not reliable and an order is not guaranteed, and therefore different measures such as Automatic Repeat ReQuest and Guaranteed Delivery have been applied in the UDP Communication module.

The UDP Communication module ensures that a connection disruption is handled. The UDP Communication module has the capabilities to ensure deliverance of messages despite of the network being down, hence the desired state will be reached at a later point in time. The UDP Communication module can be seen in figure 7.7.

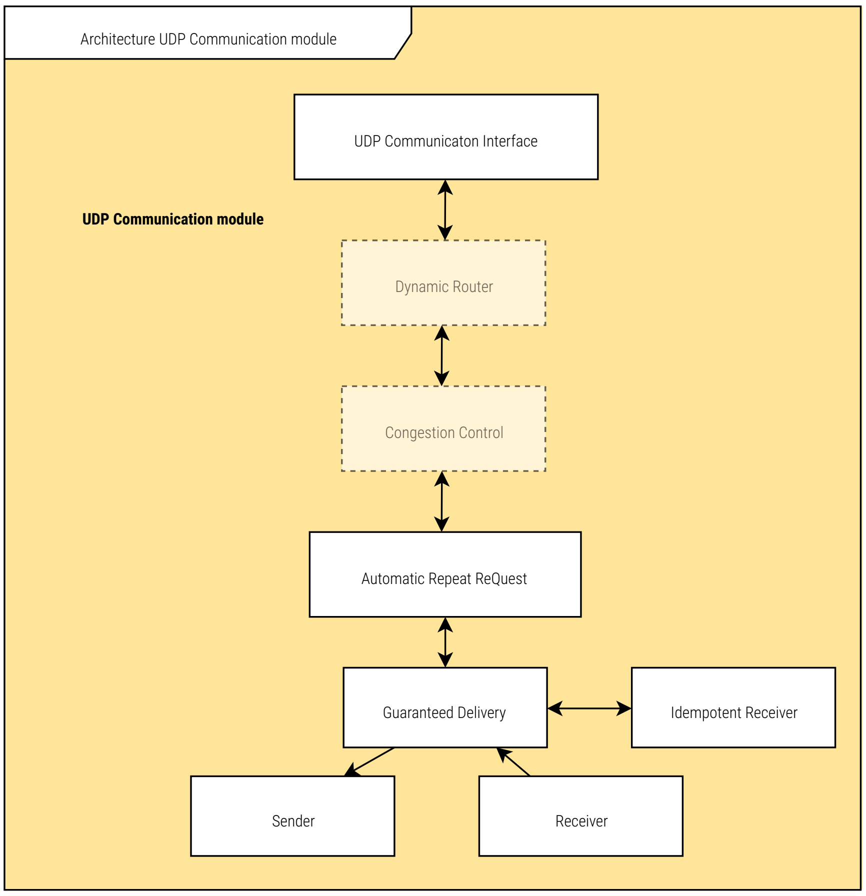


Figure 7.7: Architecture for the UDP Communication module

The UDP Communication module is used both on the HarakaMQ Client and HarakaMQ Broker. The components in the UDP Communication module have different responsibilities, which can be seen in table 7.4.

Component name	**Description**
UDP Communication Interface	Is used to communicate between the Application Layer module and the UDP Communication module
Dynamic Router	Dynamically routes the messages to one of the available HarakaMQ Brokers (not implemented in HarakaMQ)
Congestion Control	Controls the flow of messages in the network (not implemented in HarakaMQ)
Idempotent Receiver	Filters away duplicated messages
Automatic Repeat ReQuest	Ensures that messages are delivered and ordered
Guaranteed Delivery	Guarantees that messages can be delivered, by saving them to the disk
Sender	Sends UDP packets from one endpoint to another
Receiver	Receives UDP packets to be handled

Table 7.4: Description of the components in the UDP Communication module

The different message types in HarakaMQ can be seen in table 7.5. These messages are passed from the Application layer module to the UDP Communication module, where arriving at the UDP Communication module an UDP message type is assigned to each message.

Message type	**Description**
QueueDeclare	Is used by a HarakaMQ Client to declare a topic on a specific HarakaMQ Broker
Subscribe	Is used by a HarakaMQ Client to subscribe to a specific topic
Unsubscribe	Is used by a HarakaMQ Client to unsubscribe from a specific topic
Publish	Is used to publish a message from a publisher to a subscriber
AntiEntropy	Used for communication between HarakaMQ Brokers to replicate the received messages from HarakaMQ Clients
ClockSync	Used for clock synchronization

Table 7.5: Description of the different message types passed to the UDP Communication module

The different UDP message types used for communication between two UDP Communication modules can be seen in table 7.6.

UDP Message type	Description
Packet	Packet containing one or more publish messages
ResendRequest	A resend request telling the sender that a specific message is missing, and therefore should be resend
GarbageCollect	Is returned from a receiver to a sender after a preconfigured amount of messages have been received, to remove saved messages from the disk. This is necessary as IoT devices have limited space on the disk
DelayedAck	Ensures that all messages are delivered to the receiver. The DelayedAck is send to the receiver after a preconfigured delay after the last message sent from the sender
DelayedAckResponse	Ensures that the DelayedAck is received. The DelayedAck needs to be acknowledged by the receiver, to let the sender know that every message was received, and the desired state has been achieved

Table 7.6: Description of the different message types used for communication between UDP Communication modules

7.3.3.1 Idempotent Receiver

The Idempotent Receiver ensures that duplicated messages within a preconfigured amount of messages are not published to the Application layer module, provided that a system failure has not occurred. The received message is verified in the Idempotent Receiver before the message is saved on the disk.

The implementation of the Idempotent Receiver is a dictionary saving a unique message id for each message. When a message is to be verified, the Idempotent Receiver checks if the message id for the received message already has been received. The time for finding a value in the dictionary is constant, as the unique message id is hashed. When a receiver has a system failure, the dictionary saved in the RAM will be cleared, and thereby the information about which messages have been received is lost.

The reason for saving the unique ids of the received messages in the RAM, is that it is faster to access, than reading and writing the dictionary to the disk. To avoid getting an out of memory exception, the dictionary is cleared when receiving a GarbageCollect message, which happens after a preconfigured amount of published messages.

7.3.3.2 Automatic Repeat ReQuest

The Automatic Repeat ReQuest (ARQ) component ensures ordering of messages, and if a message is received out of order this message is buffered on the disk and a ResendRequest is issued. The implementation of the ARQ is based on the theory about selective repeat, which is described in section 5.2.3. HarakaMQ is a solution that aims to lower the amount of coordination messages, so a variation of selective repeat is implemented. The difference is that the receiver does not send back an acknowledgement message for each message. The receiver sends back a ResendRequest, when a received message is out of order. The receiver does not publish any messages from the specific sender, to the upper layers until the order has been restored.

A DelayedAck is required, to ensure that all messages have been delivered and the desired state has been obtained. For example a receiver receives the message with sequence number 99, and the message before that had sequence number 98. In this case the receiver sees the messages as in order. The sender thereafter sends the last message with sequence number 100, but the message is never received by the receiver, and the receiver does therefore not know that it should request a resend for that message. After a calculated delay, based on the messages sent, the DelayedAck is send to ensure the delivery of the last message. The DelayedAck will be send until a DelayedAckResponse has been received. The mentioned example is illustrated in figure 7.8.

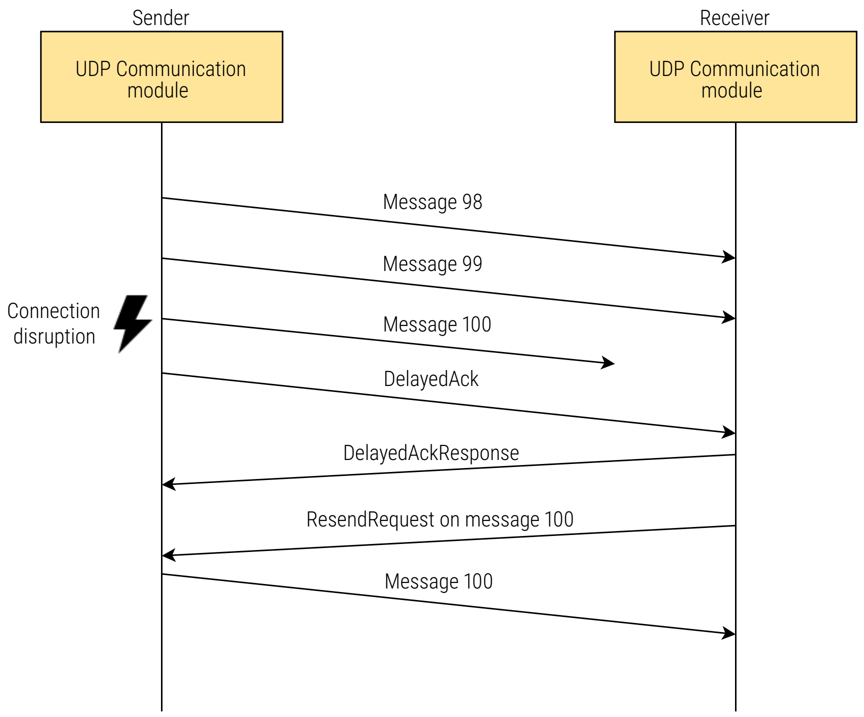


Figure 7.8: An example of how the DelayedAck is used to indicate if all the messages have been received

The sequence for sending a message can be seen in figure 7.9. The Application layer in figure 7.9 represents the components above the Automatic Repeat ReQuest component (*UDP Communication Interface, Dynamic Router, Congestion Control*). These components can be seen in figure 7.7.

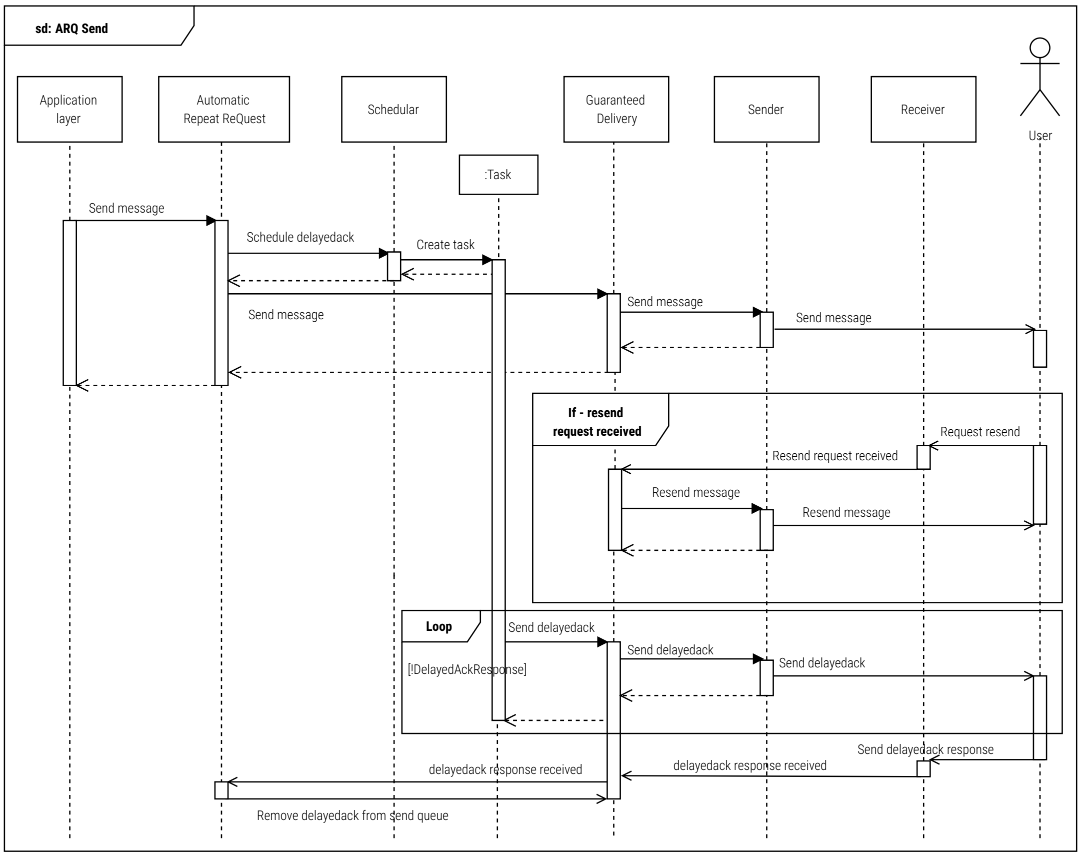


Figure 7.9: Sequence diagram describing the sequence for sending a message

The User in figure 7.9 and 7.10 is an instance of a HarakaMQ Client or HarakaMQ Broker which sends or receives messages.

When a message is sent, a receiver component will asynchronously listen for incoming messages. Instead of the consumer asking the receiver if any new messages are received, the receiver will emit new message received events as soon as they are received, this is also called an event-driven consumer [66, pp. 442-446]. The sequence for receiving a message can be seen in figure 7.10.

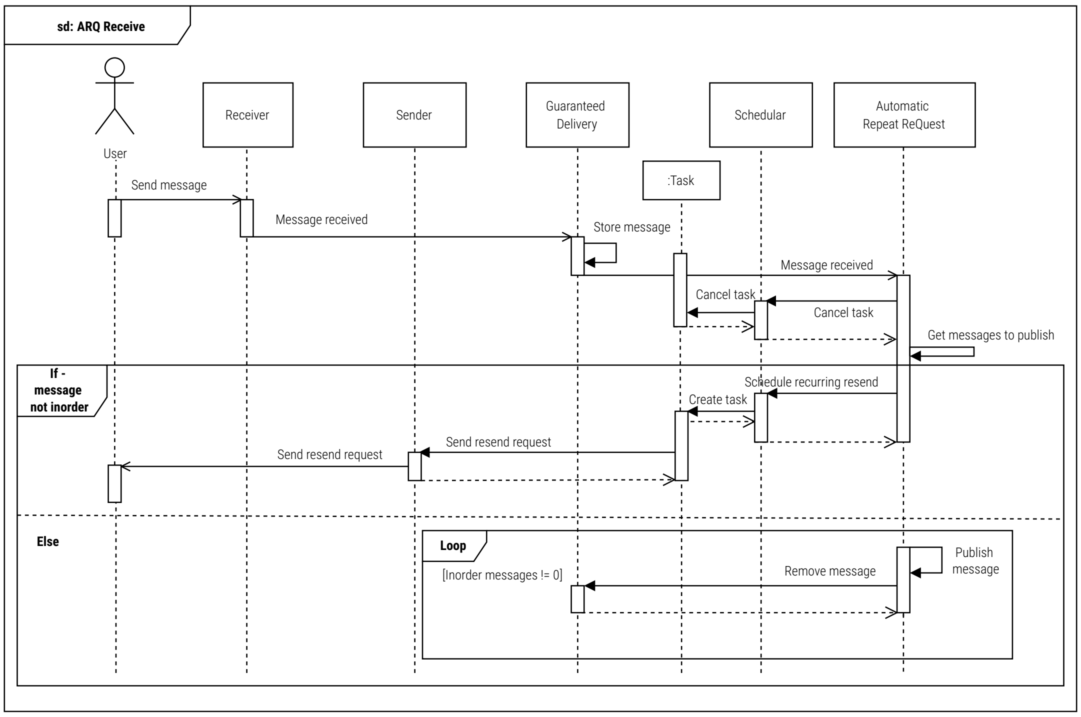


Figure 7.10: Sequence diagram describing the sequence for receiving a message

Every received message with the message type packet is saved on the disk, before it is published to the Application layer module. When a message is published it is removed from the disk.

When a message is received out of order a ResendRequest is scheduled for the missing message, in a recurring delayed task. This task will continuously send ResendRequests, until the requested message is received. To avoid sending too many ResendRequests the task is delayed, a possibility is that a message is in transport.

After a preconfigured number of messages are published to the Application layer module, a GarbageCollect message is send from the receiver to the sender. As each message with the message type packet is stored on the disk before sending, and as there is no acknowledgement message for each message, the sender does not know when it safely can remove the stored messages from the disk. This is why a GarbageCollect message is necessary.

If a received message is out of order, it is added to a sorted list. A dictionary which saves the sorted list as the value, and the client id as the key. The reason for using a dictionary to contain client ids, is that every client's sorted list of messages can be accessed with a constant search time. Furthermore the sorted list is used to guarantee that messages are sorted by the sequence number, and thereby when searching for in order messages, the searching process can be stopped as soon as the first out or order message is found. The code snippet shown in listing 7.2 shows how messages, which are in order, are retrieved.

```
for (var i = 0; i < count; i++)
        //Find The next message inorder
        if (DictionaryWithSortedPackets
                [receivedPacket.SenderClient]
                .Keys[0] == receivedPacket.Packet.SeqNo + i + 1)
        {
                packetsToPublish.Enqueue(
                DictionaryWithSortedPackets
                        [receivedPacket.SenderClient]
                        .Values[0]);

                //Remove the message from the sorted list
                DictionaryWithSortedPackets
                        [receivedPacket.SenderClient]
                        .Remove(DictionaryWithSortedPackets
                        [receivedPacket.SenderClient]
                        .Keys[0]);
        }

        //Messages are not in order anymore then stop iterating
        //Schedule a ResendRequest with 100 ms delay
        else
        {
                _schedular.ScheduleRecurringResend(100,
                    receivedPacket.Ip, receivedPacket.Port,
                    receivedPacket.Packet.SeqNo + i + 1,
                    SendResendRequest);
                break;
        }
```

LISTING 7.2: Code snippet of how to get sorted packets in order

The check to see if a received message is in order, is made by $receivedPacket.Packet.SeqNo + i + 1$. An example of the expected sequence number, with a received message sequence number being 4, would be $4 + 0 + 1$, which would result in an expected sequence number of 5.

The check is performed on the messages in the sorted list, which is also seen in listing 7.2. If the message in the sorted list has the expected sequence number, it is enqueued to the *packetsToPublish* queue and removed from the sorted list. If one of the messages in the sorted list has a different sequence number than the expected, a ResendRequest is scheduled for the message with the expected sequence number, and the search stops.

7.3.3.3 Guaranteed Delivery

The Guaranteed Delivery component resides between the Automatic Repeat ReQuest and Sender/Receiver component. This is seen in figure 7.6. The reason for this position is that the Guaranteed Delivery component stores all outgoing and ingoing messages with the message type Packet or DelayedAck [66, pp. 124-128]. In the case of a system failure the messages is stored on the sender device, and it is then possible for the messages to be routed to another receiver.

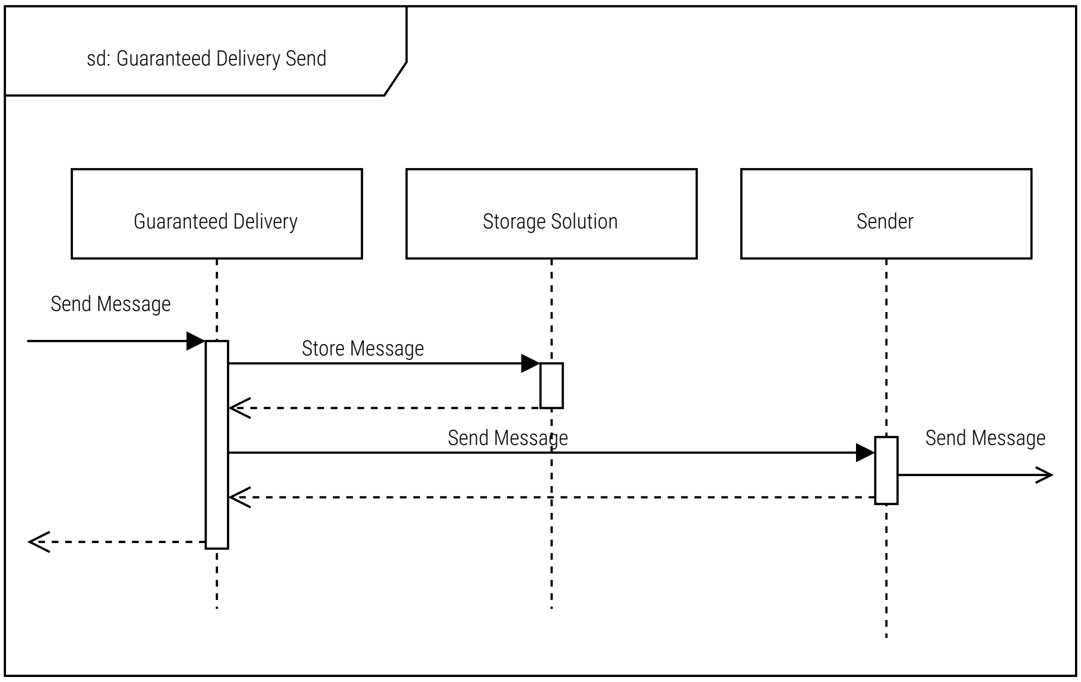


Figure 7.11: Sequence diagram depicting the send procedure for the Guaranteed Delivery Component

Figure 7.11 depicts the send procedure for the Guaranteed Delivery component. First the message is passed to the Guaranteed Delivery component from the Automatic Repeat ReQuest component. The message is then stored with the help of the storage solution, and when the message is stored, the message will be passed on to the sender component that will send the message. When resending a message, the message to be send, is already present on the disk, and can therefore be obtained from the disk. The DelayedAck is saved on the disk, as it will get resend, if a DelayedAckResponse is not received within a certain timeout.

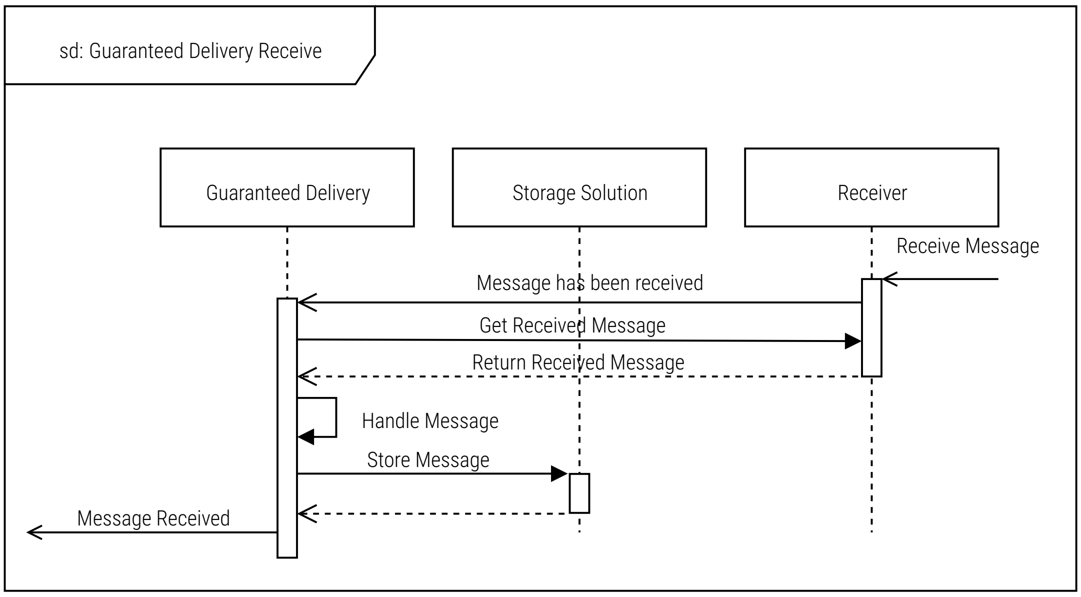


Figure 7.12: Sequence diagram depicting the receive procedure for the Guaranteed Delivery Component

Figure 7.12 depicts the receive procedure for the Guaranteed Delivery component. First a message is received and the Guaranteed Delivery component is notified about this via an event. Then the Guaranteed Delivery component fetches the message from the Sender component. Furthermore the Guaranteed Delivery component handles the message which means that the message is deserialized, and prepared for further processing.

The Guaranteed Delivery component has the responsibility of deserializing the message, because it will save processing time for the dedicated thread that the Receiver component uses. The rational of saving processing time for the dedicated thread, that the Receiver component uses, is that it needs to listen as much as possible for incoming messages.

After the message has been handled it is then send to the Storage Solution and saved to the disk. When the message has been saved to disk it is published to the Automatic Repeat ReQuest component by an event.

The Guaranteed Delivery component ensures that if a message has not been delivered to the receiver it is possible to deliver the message to another receiver at a later point in time. It is a performance heavy component as it takes time to save the messages to the disk, but this is necessary to guarantee message delivery from a sender to a receiver.

7.3.3.4 Sender

The implementation of the Sender component is based on the theory about interprocess communication, described in section 5.1, where UDP packets are sent from one endpoint to another. A socket is created in the Sender component to handle the communication between sender and receiver.

An endpoint to a receiver is required to be known by the Sender component. This endpoint consist of an IP address and a port which is provided from the Application layer module. The message is serialized to bytes at the Sender component. The bytes and the endpoint is passed to the socket, and thereby the bytes are send to the given endpoint.

7.3.3.5 Receiver

In the UDP Communication module the Receiver is listening asynchronously for new incoming messages. When a message is received it is handled in the UDP Communication module before it is published to the Application layer module. To avoid blocking the Receiver while handling the messages, the Half-Sync / Half-Async pattern is used to decouple the synchronous process from the asynchronous process [67]. The Half-Sync / Half-Async pattern enables communication between two threads via a thread safe queue. This approach ensures that received messages are stored in a queue until the other layers are ready to handle the messages.

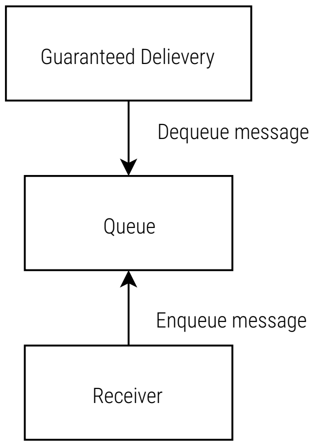


Figure 7.13: Queue added between the Receiver and the Guaranteed Delivery component

In figure 7.13 a queue is added between the Receiver and the Guaranteed Delivery component. In the Guaranteed Delivery component a task is created to handle the messages. The task created will keep dequeuing and handling messages, as long as there are messages in the queue. With this approach the Receiver can focus on a single responsibility, which is receiving messages.

7.3.3.6 Bundle Mechanism

An automatic bundle mechanism was created to boost the throughput of messages from a sender to a receiver by saving bytes on the number of headers needed per message (aka *The small-packet problem*). The algorithm developed to bundle the messages is inspired by the Nagle's algorithm [68]. The Nagle's algorithm works by bundling messages together in TCP if a response is delayed. The bundle mechanism works by adding all incoming messages to a queue. A consumer task then starts to consume the first messages from the queue after a preconfigured delay. The delay is only added for the first incoming message. If there are more messages left when the first packet is created and sent, the consumer thread continues to create packets until the queue is empty of messages. The packet is send when it is full, which corresponds to the payload size of a UDP message or as close a possible, or if there are no more messages on the queue.

Figure 7.14 is an abstraction of how 10,000 messages are bundled into packets, and sent through a HarakaMQ Broker to another HarakaMQ Client which unfolds the packets and publishes the messages to the application. The packet is not opened before it is received at the intended receiver. For example in a client-server setup the packet header contains all the information needed for the HarakaMQ Broker to pass the packet to the subscribers without opening it.

To avoid exceeding the upper limit of a payload on a UDP message (65,535 - 8 (UDP-header) - 20 (IP-header) = 65,507), a preconfigured message header size is used. An example of this could be to configure the message header size to 50 bytes. The chosen message header size requires to be configured to the maximum size of the message header.

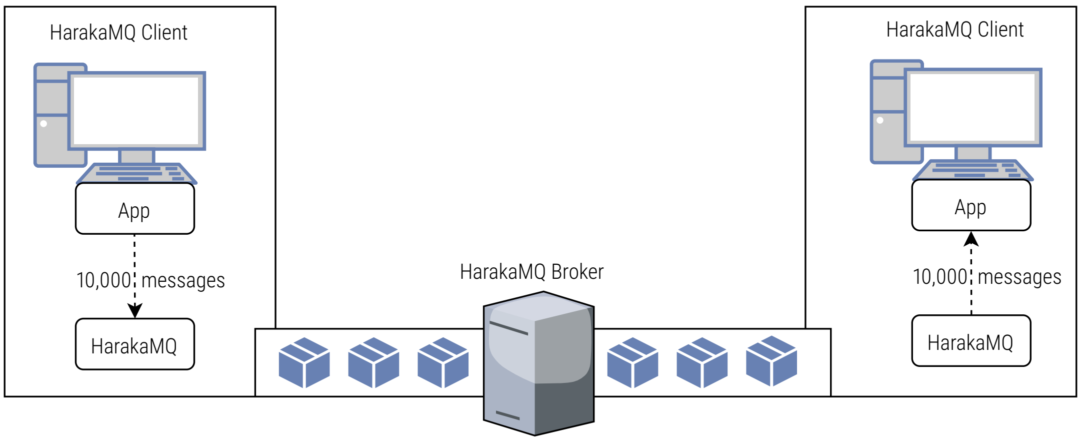


Figure 7.14: Bundling of messages into packets

In this example a HarakaMQ Client needs to publish 10,000 messages, where each message contains 30 bytes of data. The number of messages per packet can be estimated in the following expression.

$$\frac{10,000 * (50 + 30)}{65,507} = 12.21 \ packets$$

An alternative to this would be to measure the actual message header size, but this has a cost, which is processing time. It is faster to use a preconfigured message header size, for example if the message header size varies between 30-50 bytes, then there is space for more messages per packet.

In version 1.0.0 of HarakaMQ the message header is a JSON (JavaScript Object Notation) string with a max size of 50 bytes and an average size of 41 bytes, the size depends on the information added to the string. To fixate and minimize the size of the message header, the header can be converted from a string to a byte array, like the TCP header. Then the header does not change in size, and the size is smaller than the size of the JSON string, because the convention of keeping property names in the string does not need to be included in the byte array.

Even though an automatic bundling mechanism is implemented in HarakaMQ the IPv4 protocol fragments the packets when they are sent or received [63]. The fragment size for IPv4 is 1500 bytes, meaning that packets over 1500 bytes will be fragmented into packets of 1500 bytes. This is done per default in IPv4, as a packet can be formed to fit smaller MTUs. Per default the MTU size is maintained in IPv4. The fragmentation can be disabled in HarakaMQ if needed.

7.3.3.7 Durable Subscriber

A durable subscriber ensures that a subscriber receives all published messages, even the messages published when the subscriber's connection is disrupted (aka time decoupling). This guarantee is given from when the HarakaMQ Broker or HarakaMQ Client has passed a message to the UDP Communication module. The message is then persisted on the disk, and send to the intended receiver. The message can then be requested to be send again by the receiver, if the message was not delivered the first time it was send. This approach ensures subscribers are durable.

7.3.4 Storage Solution

The storage solution is created as a file reader / writer, where it is possible to create separate files to store messages in. All persistence to the disk is done by the storage solution. This is useful when two different threads are persisting data concurrently for example in an ingoing and outgoing message queue. In this situation the outgoing messages can be written to one file, and the ingoing messages can be written to another. This enables the possibility to have one thread sending messages and another thread receiving messages. This solution requires minimal thread synchronization. In the HarakaMQ UDP Communication Module the given example is used. In the HarakaMQ implementation thread synchronization cannot completely be avoided. This is because if one of the ingoing messages is a *GarbageCollect* message, the thread owning the ingoing messages file needs to make changes to the outgoing messages file.

The approach requires a *lock* [69]. A *lock* ensures that only one thread enter the critical section, which is writing and reading to a file in the case of HarakaMQ. This kind of thread synchronization should be kept to a minimum, so threads do not have to wait for the other thread to finish.

```
lock (_harakaDb.GetLock(Setup.OutgoingMessagesCS))
{
        var messages =
            _harakaDb.GetObjects<ExtendedMessageInformation>(
                Setup.OutgoingMessagesCS);
        messages.Add(msg);
        _harakaDb.StoreObject(Setup.OutgoingMessagesCS, messages);
}
```

LISTING 7.3: Code snippet of how to store a message with the storage solution

In listing 7.3 it is seen how a message is stored to the outgoing file. A problem with the solution is that the whole file needs to be read and rewritten each time a write is performed. This is a bottleneck when introducing a large number of messages for example 1,000,000 messages (each message containing 30 bytes) at once. To improve this, each message should be written into a field with a unique index. This makes it possible to change only one field, and not the whole file. The reason that this has not been implemented is because a viable solution for this thesis has not yet been discovered. Alternative approaches could be researched, such as multiple databases to support a thread per file/database policy.

If the thread per file policy is fulfilled the synchronization between threads can be kept to a minimum in HarakaMQ.

7.4 Resource Requirements for HarakaMQ

The minimum recommended resource requirements for HarakaMQ can be seen in figure 7.7.

CPU	**RAM**	**Disk**
500 MHz dual core processor	256 MB	2.2 GB

Table 7.7: Resource requirements specification

The 500 MHz dual core processor has been selected as the minimum requirement as HarakaMQ run as a multi-threaded application. As MacOS Sierra require a minimum of 8.8 GB RAM it is not well suited for IoT devices. The resource requirements for HarakaMQ does therefore not account for the resource requirements for MacOS Sierra. A disk space of 2.2 GB has been selected as the Windows IoT Core and Linux Ubuntu require a minimum of 2 GB and 1.5 GB, and HarakaMQ itself requires a small amount of storage to run, and store messages.

HarakaMQ is able to run on an Advanced RISC Machine (ARM) that fulfill the constraints mentioned in table 7.7 and also runs on one of the operating systems supporting .NET Core 2.0, which can be seen in section 3.2. HarakaMQ are not able to run on Arduino as none of the operating systems that run .NET Core 2.0 is supported by Arduino.

The size of the HarakaMQ Client in version 1.0.0 has a size of 8 KB, while the size of the HarakaMQ Broker in version 1.0.0 is 1.39 MB. These sizes are without the size of queues and database files. The size of these files depends on the number of messages currently being transfered in HarakaMQ, while also depending on how often the GarbageCollect message is issued. Therefore it is important to allocate space on the disk. In HarakaMQ version 1.0.0 10,000 messages (30 bytes of data for each message) on the disk have a size of 743 KB. The number of messages has to be taken into account when the storage requirements are to be determined.

7.5 The Mathematical Model of HarakaMQ

A mathematical model was developed to create transparency of the number of coordination messages needed to send N number of messages from a HarakaMQ Client to N_S number of HarakaMQ Client subscribers.

Disclaimer: *An important assumption for equation 7.4 is that N number of messages are ready to be send. What is meant by ready to be send is that the device the HarakaMQ Client is running on can load the messages into a queue faster than they can be bundled into a packet, and send to the receiver.*

Another important assumption is that $M_S + M_H <= P_M$.

Name	Description	Unit
N_S	Number of subscribers	Count
N	Number of messages	Count
GC	Garbage collect at message	Count
D_A	Delayed acknowledgment	Count
D_R	Delayed acknowledgment response	Count
N_B	Number of brokers	Count
M_H	Message header	Byte
M_S	Message size	Byte
P_M	Payload max size	Byte
F_S	Fragment size	Byte
L	Loss	Percentage
R_T	Run time	Milliseconds
A_{RT}	Anti-entropy round time	Milliseconds

Table 7.8: Variable names, description and units

7.5.0.1 Client-Server setup

Disclaimer: *The following equations is intended for a client-server setup, and it will not yield the correct answer when using a cluster based setup. To see the equations for the cluster based setup goto 7.5.0.2.*

$$F_M = \frac{P_M}{F_S} + 1 \tag{7.2}$$

Equation 7.2 estimates the number of fragments in a message. The $+1$ is because there is a UDP declaration message before each fragment is send.

$$P_T = \frac{(M_H + M_S) \cdot N}{P_M} \tag{7.3}$$

Equation 7.3 is an estimate of the total number of packets required for N number of messages.

$$P_N = P_T \cdot F_M \tag{7.4}$$

Equation 7.4 estimates the number of fragments the IPv4 Protocol fragments the packets into.

$$M_L = (((P_N + (N_S \cdot 2)) \cdot L) \cdot 2) + (D_A + L) + ((D_A + D_R) \cdot L + 1) \tag{7.5}$$

Equation 7.5 estimates the number of delayed acknowledgments, and if there is loss in the network the number of resends of packets and delayed acknowledgements are taken into account.

All packets P_N needs to be resend if they are lost. First a resend request is issued, and then the request is answered, that is why $((P_N + (N_S \cdot 2)) \cdot L) \cdot 2$ needs to be multiplied with 2.

A subscriber uses two messages to subscribe, first a queue declare and then a subscribe. This is represented by $(N_S \cdot 2)$.

$$M_{LGC} = M_L + P_N + \frac{M_L + P_T}{GC} \tag{7.6}$$

Equation 7.6 estimates the total number of coordination messages needed from a sender to a receiver.

$$M_T = M_{LGC} + (M_{LGC} \cdot N_S) \tag{7.7}$$

Equation 7.7 estimates the total number of coordination messages needed from a HarakaMQ Client that publishes messages to a given topic, and out to all the HarakaMQ Clients that subscribes to the given topic.

7.5.0.2 Cluster setup

> ***Disclaimer:*** *The following equations is intended for a cluster based setup, and it will not yield the correct answer when using a client-server setup. To see the equations for client-server setup goto 7.5.0.1.*

An important assumption for equation 7.8 and 7.9 is that $(M_H + M_S) \cdot N > \frac{P_M}{2}$ which implies that there is only space for one packet per anti-entropy message. Then $P_T <= \frac{R_T}{A_{RT}}$, because there needs to be enough anti-entropy messages to transmit the packets.

Common for equation 7.8 and 7.9 is that they estimates the total number of messages over a specified period of time R_T.

$$C_T = M_T + (((\frac{R_T}{A_{RT}} \cdot N_B - 1) \cdot 2) - (P_T \cdot (N_B - 1))) + ((N_B - 1) \cdot P_T \cdot F_M) \tag{7.8}$$

Equation 7.8 should be used if the HarakaMQ Client publishes messages to the HarakaMQ Master.

$$C_T = M_T + (((\frac{R_T}{A_{RT}} \cdot N_B - 1) \cdot 2) - (P_T \cdot N_B)) + (N_B \cdot P_T \cdot F_M) \tag{7.9}$$

Equation 7.9 should be used if the HarakaMQ Client publishes messages to a HarakaMQ Slave.

7.6 Remarks

A remark for this thesis is that fault tolerance was not fully developed, this is because in version 1.0.0 of HarakaMQ the Dynamic Router is not implemented. The Dynamic Router is required to change the routing dynamically. The idea is that a HarakaMQ Broker detects another HarakaMQ Broker experience a connection disruption or system failure, by looking at which HarakaMQ Brokers has and has not responded in an anti-entropy round. The HarakaMQ Broker which is still running then notifies the HarakaMQ Clients to switch the routing of the messages. This approach can also be used to distribute HarakaMQ Clients, so the load is balanced between the HarakaMQ Brokers.

Another remark for the implementation in this thesis is unit testing. It was used to check the functionality correctness of components. The intention was not to have 100 % coverage, but to open the solution for changes along the development, and still be able to prove the correctness of the functionality.

Another remark was to use dependency injection to reduce coupling between components in HarakaMQ. Dependency injection does not reduce coupling, per se, because the dependency is still there. However the responsibility of handling the dependency is collected into a single component, called a container. When the responsibility is assigned to a single component, it gives the possibility to assign conventions for the dependencies which are injected [70]. The Dependency injection container used in HarakaMQ is Simple Injector [71]: it is open-source and published under the MIT-license [72]. It was created for .NET 4+ and made to be fast [73]. Simple Injector was chosen because of the speed and it is available for .NET Core 2.0.

Another remark that should be mentioned is the use of MessagePack [74]. MessagePack is used to serialize and deserialize messages. Serialization and deserialization occurs when there is a read or write to a file, and when any kind of messages are send from a sender to a receiver. The size of the serialized messages are important for the performance of the system, as the larger the messages are, the longer it takes to serialize or deserialize. MessagePack has an ideal feature, which is to serialize everything to bytes, so it is ready to be send. It is also able to deserialize from bytes.

8 | Experiments

The experiments are inspired by Vineet John and Xia Liu and the work they have been doing [75]. The experiments are performed to compare the developed solution HarakaMQ with RabbitMQ.

An acceptance test of the requirements from section 4.2 has been performed. The results can be seen in appendix A

8.1 Test Protocol

8.1.1 Test Bench

A test bench was created to make a comparison between HarakaMQ and RabbitMQ. To perform the tests virtual machines was used through VirtualBox [76]. The reason for using a virtual machine is that it enables to perform the tests in a virtual sand box. This sand box has access to the specific resources seen in table 8.1. These resources are the same for all the virtual machines. Common for all the virtual machines is that the firewall, automatic update, and anti-virus scanning are disabled.

OS	Virtual Cores	RAM	Network setting
Windows 10 64-Bit	2 (3.5 Ghz)	2048 MB	Bridged Adapter

Table 8.1: Resources assigned to the virtual machines

The bridged adapter setting was used, as it enables the Host machine seen in table 8.2 to enable the virtual machines to communicate with each other. From the virtual machines perspective it look like they are connected to a physical network with a cable [77].

OS	CPU	RAM	Hard drive
Windows 10 64-Bit	Intel I7 4770K	16 GB DDR 3	2 TB (7200 rpm)

Table 8.2: Host machine

8.1.2 Test Procedure

It was decided to compare HarakaMQ and RabbitMQ by sending 1,000,000 messages in batches of 10,000 messages which equals 100 rounds. Between each round there was a short break, to ensure that the batch of 10,000 messages were received, before starting on a new round. The break was adjusted from two to eight seconds depending on the elapsed time for sending and receiving one batch. An average over the 100 rounds is then obtained to compare HarakaMQ with RabbitMQ. The setup for the test protocol can be seen in figure 8.1.

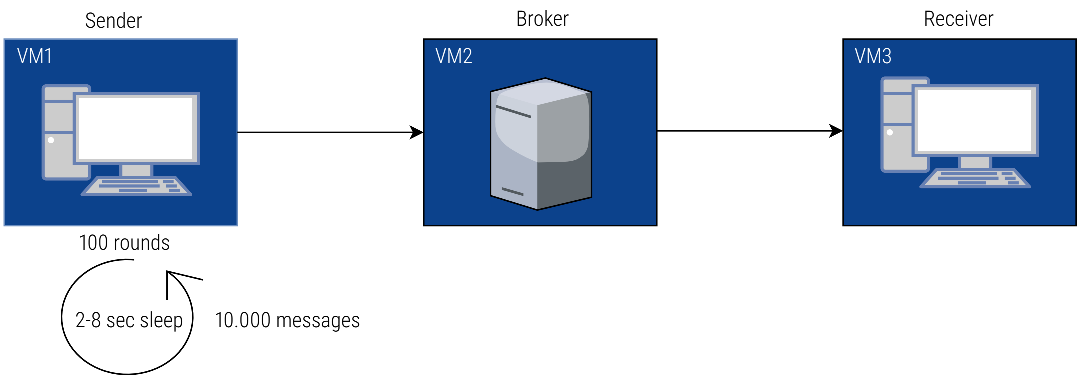
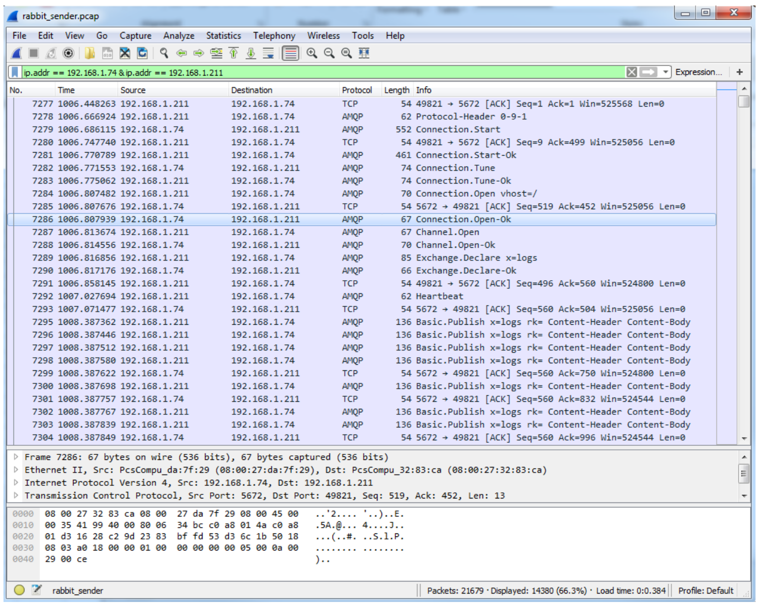


Figure 8.1: Setup for the test procedure used for the experiments

The reason for the chosen batch size of 10,000 messages is that the storage solution in HarakaMQ is a bottleneck with the current implementation. The bottleneck is explained in section 7.3.4.

8.1.3 Analyzation of Results

To analyze the number of coordination messages the ingoing and outgoing traffic from all the virtual machines are monitored by the host machine and logged into a packet capture (.pcap) file. A snippet of a packet capture file is shown in figure 8.2.


rabbit_sender.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...
rabbit_sender
Profile: Default

```
ip.addr == 192.168.1.74 & ip.addr == 192.168.1.211

No.  Time        Source          Destination     Protocol  Length Info
7277 1006.448263 192.168.1.211   192.168.1.74    TCP         54 49821 → 5672 [ACK] Seq=1 Ack=1 Win=525568 Len=0
7278 1006.666924 192.168.1.211   192.168.1.74    AMQP        62 Protocol-Header 0-9-1
7279 1006.686115 192.168.1.74    192.168.1.211   AMQP       552 Connection.Start
7280 1006.747740 192.168.1.211   192.168.1.74    TCP         54 49821 → 5672 [ACK] Seq=9 Ack=499 Win=525056 Len=0
7281 1006.770789 192.168.1.211   192.168.1.74    AMQP       461 Connection.Start-Ok
7282 1006.771553 192.168.1.74    192.168.1.211   AMQP        74 Connection.Tune
7283 1006.775062 192.168.1.211   192.168.1.74    AMQP        74 Connection.Tune-Ok
7284 1006.807482 192.168.1.211   192.168.1.74    AMQP        70 Connection.Open vhost=/
7285 1006.807676 192.168.1.74    192.168.1.211   TCP         54 5672 → 49821 [ACK] Seq=519 Ack=452 Win=525056 Len=0
7286 1006.807939 192.168.1.74    192.168.1.211   AMQP        67 Connection.Open-Ok
7287 1006.813674 192.168.1.211   192.168.1.74    AMQP        67 Channel.Open
7288 1006.814556 192.168.1.74    192.168.1.211   AMQP        70 Channel.Open-Ok
7289 1006.816856 192.168.1.211   192.168.1.74    AMQP        85 Exchange.Declare x=logs
7290 1006.817176 192.168.1.74    192.168.1.211   AMQP        66 Exchange.Declare-Ok
7291 1006.858145 192.168.1.211   192.168.1.74    TCP         54 49821 → 5672 [ACK] Seq=496 Ack=560 Win=524800 Len=0
7292 1007.027694 192.168.1.211   192.168.1.74    AMQP        62 Heartbeat
7293 1007.071477 192.168.1.74    192.168.1.211   TCP         54 5672 → 49821 [ACK] Seq=560 Ack=504 Win=525056 Len=0
7295 1008.387362 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7296 1008.387446 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7297 1008.387512 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7298 1008.387580 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7299 1008.387622 192.168.1.74    192.168.1.211   TCP         54 5672 → 49821 [ACK] Seq=560 Ack=750 Win=524800 Len=0
7300 1008.387698 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7301 1008.387757 192.168.1.74    192.168.1.211   TCP         54 5672 → 49821 [ACK] Seq=560 Ack=832 Win=524544 Len=0
7302 1008.387767 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7303 1008.387839 192.168.1.211   192.168.1.74    AMQP       136 Basic.Publish x=logs rk= Content-Header Content-Body
7304 1008.387849 192.168.1.74    192.168.1.211   TCP         54 5672 → 49821 [ACK] Seq=560 Ack=996 Win=524544 Len=0

▷ Frame 7286: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)
▷ Ethernet II, Src: PcsCompu_da:7f:29 (08:00:27:da:7f:29), Dst: PcsCompu_32:83:ca (08:00:27:32:83:ca)
▷ Internet Protocol Version 4, Src: 192.168.1.74, Dst: 192.168.1.211
▷ Transmission Control Protocol, Src Port: 5672, Dst Port: 49821, Seq: 519, Ack: 452, Len: 13

0000  08 00 27 32 83 ca 08 00  27 da 7f 29 08 00 45 00   ..'2.... '..)..E.
0010  00 35 41 99 40 00 80 06  34 bc c0 a8 01 4a c0 a8   .5A.@... 4....J..
0020  01 d3 16 28 c2 9d 23 83  bf fd 53 d3 6c 1b 50 18   ...(..#. ..S.l.P.
0030  08 03 a0 18 00 00 01 00  00 00 00 00 05 00 0a 00   ........ ........
0040  29 00 ce                                           )..

Packets: 21679 · Displayed: 14380 (66.3%) · Load time: 0:0.384
```

Figure 8.2: Example of a .pcap file containing captured packets from RabbitMQ

To measure the resource usage HWiNFO [78] was used. The logged information are stored in a comma-separated values (.CSV) file. The relevant information for this thesis was RAM usage, CPU load and Disk I/O.

All the experiments of HarakaMQ can be found on Github at `https://rotvig.github.io/HarakaMQ-Benchmark/`. All the experiments of RabbitMQ can be found on Github at `https://rotvig.github.io/RabbitMQ-Benchmark/`. All the projects are precompiled and ready to run. There are some requirements to run the experiments, which are stated in the Github repository.

8.1.4 Quality Metrics

Quality metrics are used to quantify the results obtained in the experiments, and enable a comparison of HarakaMQ with RabbitMQ.

8.1.4.1 Throughput

Throughout is a quality metric which represent the number of messages the message oriented middleware can handle over time. The throughput is calculated by equation 8.1.

$$T = \frac{Number\ of\ messages}{Elapsed\ time\ in\ seconds} \tag{8.1}$$

In equation 8.1 throughput is denoted as T which represent the amount of messages per second. The throughput is calculated by taking the total number of messages, and dividing this with the elapsed time in seconds.

The throughput is dependent on the hardware the system is running on, and the hardware used in the network, such as a router or switches.

8.1.4.2 Coordination Message

Coordination message is a quality metric which represents any traffic between participants in the network (e.g. protocol, acknowledgement, heart beat, packet).

8.1.4.3 Resource Usage

Resource usage is a quality metric which represents the usage of RAM, CPU and disk I/O. The reason for this is that there is a correlation between energy consumption and CPU and disk I/O. Furthermore RAM is a limited resource on resource constrained devices, such as IoT devices.

8.2 Client-Server Setup with Results and Discussion

An experiment is performed to examine how many coordination messages are used to transfer 10,000 messages from one publisher to one subscriber in a client-server setup for both RabbitMQ and HarakaMQ.

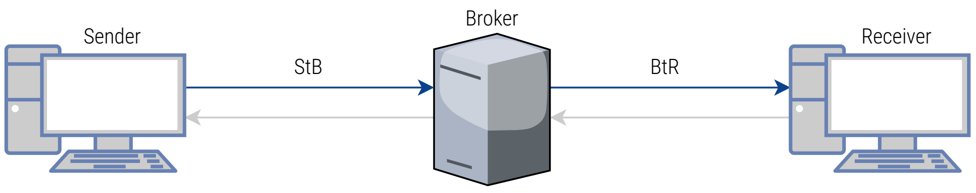


Figure 8.3: Setup for experiments running in a client-server setup

In figure 8.3 the setup for the experiment is shown. It is also shown which connections are made between the Sender, Broker and Receiver. Two connections are made in the client-server setup, namely the Sender to Broker (StB) and Broker to Receiver (BtR) connections. The blue connections in figure 8.3 represents the coordination messages sent from either the Sender or the Broker, while the light gray connections represents the received coordination messages from either the Broker or the Receiver. The two connections are described in table 8.3.

Abbreviation	Description
StB	Sender to Broker
BtR	Broker to Receiver

Table 8.3: Description of the different connections in a client-server setup

8.2.1 RabbitMQ

In figure 8.4 the coordination messages sent between the Sender, Broker and Receiver are shown.

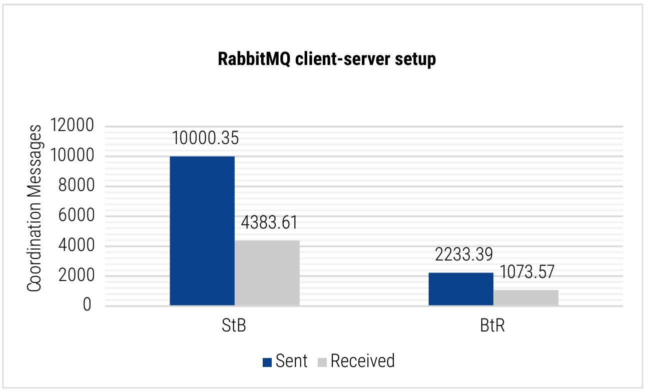


Figure 8.4: Coordination messages used in a RabbitMQ client-server setup to send 10,000 messages

In figure 8.4 it can be seen that 10,000.35 coordination messages are sent from the Sender to the Broker, and that 4383.61 coordination messages have been returned to the Sender.

The results show that not every sent message from the Sender is acknowledged by the Broker in RabbitMQ. The results show that approximately every second message is acknowledged in RabbitMQ.

From the Broker to the Receiver the messages sent from the sender is bundled into packets containing 4-5 messages, and thereby giving an average of 2233.39 coordination messages sent.

8.2.2 HarakaMQ

In figure 8.5 the coordination messages sent between the Sender, Broker and Receiver are shown for HarakaMQ.

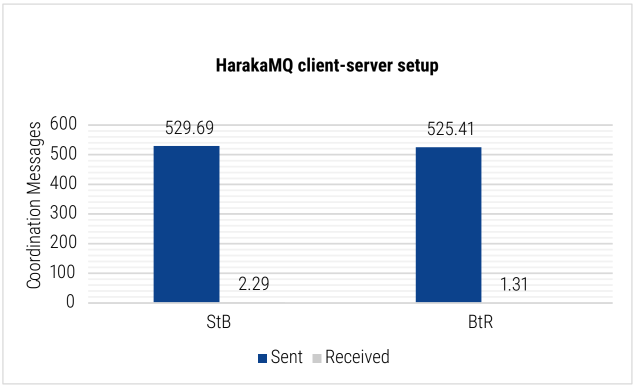


Figure 8.5: Coordination messages used in a HarakaMQ client-server setup to send 10,000 messages

With equation 7.7, the theoretical total number of expected coordination messages for sending 10,000 messages can be calculated. The results of the experiments have shown that in average a total number of 1058.7 coordination messages are used to sent 10,000 messages. In table 8.4 the preconfigured variables used in a HarakaMQ client-server setup experiment can be seen.

Name	Description	Preconfigured value
N_S	Number of subscribers	1
N	Number of messages	10,000
GC	Garbage collect at message	10
D_A	Delayed acknowledgement	1
D_R	Delayed acknowledgement response	1
M_H	Message header	50
M_S	Message size	31
P_M	Payload max size	65,507
F_S	Fragment size	(1500 - 20 IPv4 Header) 1480
L	Loss	0

Table 8.4: Variable names and preconfigured values for HarakaMQ

The values in table 8.4 are applied to the equations from section 7.5.0.1 to find the theoretical total number of coordination messages that are expected to be used for sending 10,000 messages. The calculations can be seen in equation 8.2.

$$\begin{aligned}
F_M &= \frac{65,507}{1480} + 1 = 45.26 \\
P_T &= \frac{(50+31) \cdot 10,000}{65,507} = 12.37 \\
P_N &= 12.37 \cdot 45.26 = 559.66 \\
M_L &= (((559.66 + (1 \cdot 2)) \cdot 0) \cdot 2) + (1+0) + ((1+1) \cdot 0 + 1) = 2 \\
M_{LGC} &= 2 + 559.66 + \frac{2 + 12.37}{10} = 563.1 \\
M_T &= 563.1 + (563.1 \cdot 1) = 1126.2
\end{aligned} \tag{8.2}$$

The theoretical total number of coordination messages are therefore calculated to be 1126.2, while the average number of coordination messages sent in the experiment are 1058.7 messages. The difference between the theoretical number and the number of coordination messages used in the experiment is 6.38 %

The reason for the theoretical number of coordination messages being larger than the number of coordination messages used in the experiment, is caused by the preconfigured Message Header (50 bytes). But the actual Message Header in the experiments is closer to 41 bytes. When the messages are sent and the IPv4 protocol fragments the messages, the fragmentation is based on the actual measured Message Header size, and not the preconfigured which is used in HarakaMQ, which results in fewer messages bundled together in a packet, and more packets are then required.

Adjusting the size of the message header to 41 bytes, would yield a total number of 1001.8 theoretical messages. This yields a difference of 5.37 %.

8.2.3 Comparison of Results

In figure 8.6 the average number of coordination messages used to transfer 10,000 messages for both HarakaMQ and RabbitMQ in a client-server setup are shown.

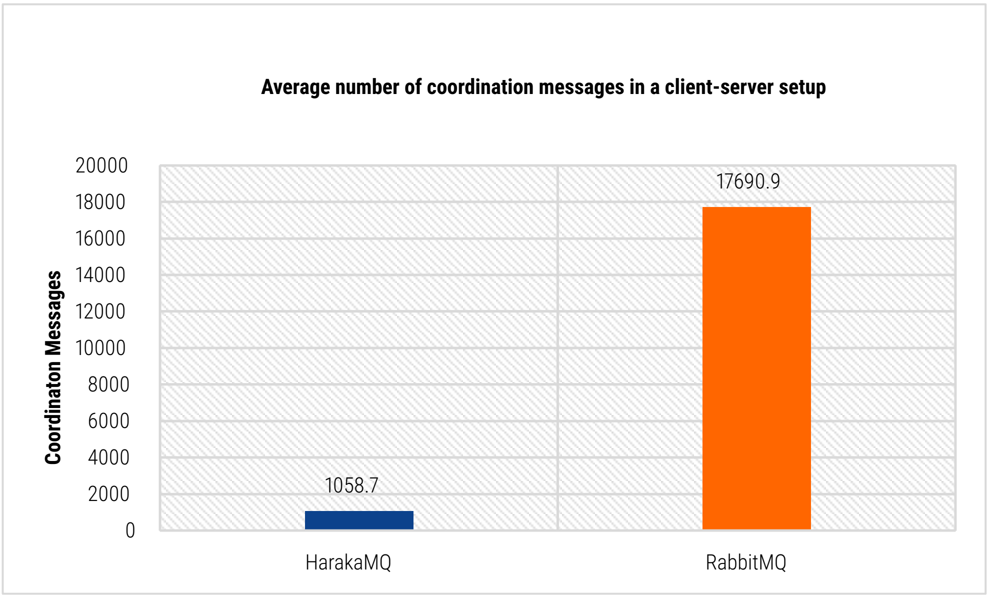


Figure 8.6: The average number of coordination messages used in RabbitMQ and HarakaMQ client-server setups to send 10,000 messages

From the results shown in figure 8.6 it can be seen that HarakaMQ sends approximately 17 times as few coordination messages as RabbitMQ, when sending 10,000 messages in a client-server setup.

The throughput for RabbitMQ and HarakaMQ in a client-server setup can be seen in table 8.5.

Middleware	**Messages sent**	**Average elapsed time (sec)**	**Throughput (msg/sec)**	**Standard deviation (sec)**
RabbitMQ	10,000	0.734	13,633	0.369
HarakaMQ	10,000	0.569	17,576	0.229

Table 8.5: Throughput for RabbitMQ and HarakaMQ in a client-server setup

The standard deviation is calculated for the results of RabbitMQ and HarakaMQ in a client-server setup. The standard deviation depicts the stability of the results. The standard deviation is measured in seconds. The standard deviation being higher for RabbitMQ, shows that the elapsed time are more inconsistent for RabbitMQ than for HarakaMQ, and thereby also being more unstable than HarakaMQ.

The throughput are visualized for both RabbitMQ and HarakaMQ in figure 8.7.

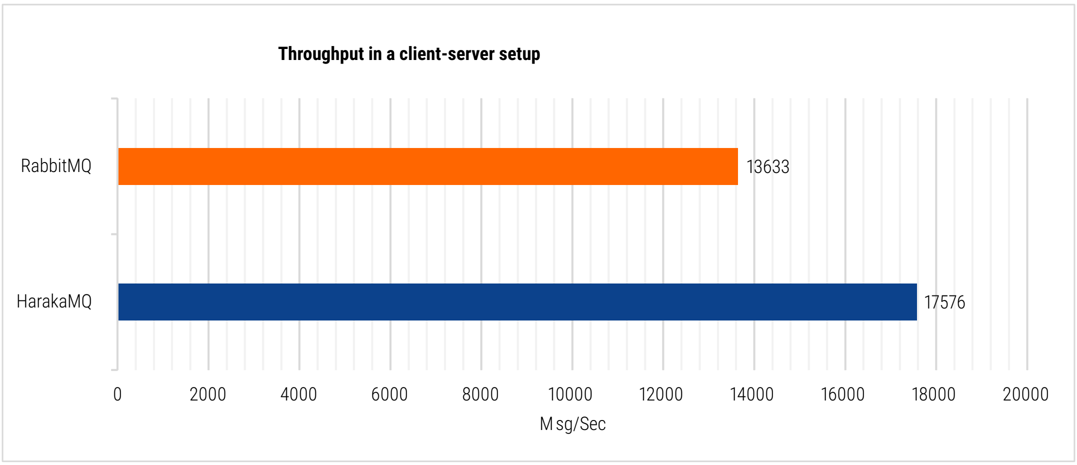


Figure 8.7: Throughput for client-server setup in RabbitMQ and HarakaMQ

In figure 8.7 it can be seen that HarakaMQ exceeds the throughput for RabbitMQ in a client-server setup with 3943 messages per second.

8.2.4 Discussion of Results

The results in throughput and coordination messages show that HarakaMQ performs better than RabbitMQ in a client-server setup. The throughput for HarakaMQ and RabbitMQ yields a difference of 3943 messages per second. Bundling of messages exist both in HarakaMQ and RabbitMQ, which presumably leads to fewer coordination messages sent overall, compared to if no bundling of messages was used. HarakaMQ bundles more messages together than RabbitMQ, which also leads to fewer coordination messages. The fact that HarakaMQ does not send acknowledgement messages, but only uses a delayed acknowledgement, have an impact on the amount of coordination messages.

The results show that the standard deviation for HarakaMQ is lower than the standard deviation for RabbitMQ. This was unexpected as HarakaMQ was designed, developed and tested in four months.

RabbitMQ acknowledged approximately half of the sent coordination messages, this indicates that RabbitMQ uses the delayed acknowledgement feature in TCP.

With the storage solution being a bottleneck in version 1.0.0 of HarakaMQ, increasing the amount of messages per batch will cause performance issues for HarakaMQ, and RabbitMQ will presumably at some point exceed the throughput for HarakaMQ. This could be found by increasing the batch size gradually, and find the crossover of the throughput between RabbitMQ and HarakaMQ.

8.3 Cluster Based Setup with Results and Discussion

An experiment is performed to examine how many coordination messages, which are used to transfer 10,000 messages from one publisher to one subscriber in a cluster based setup for both RabbitMQ and HarakaMQ.

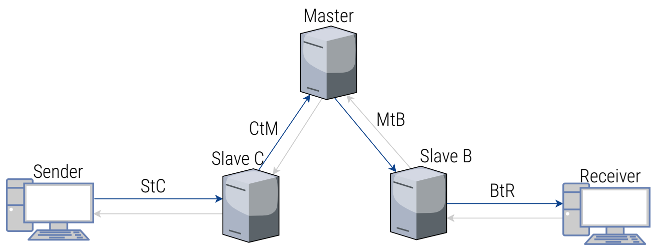


Figure 8.8: Setup for experiments running HarakaMQ in a cluster based setup

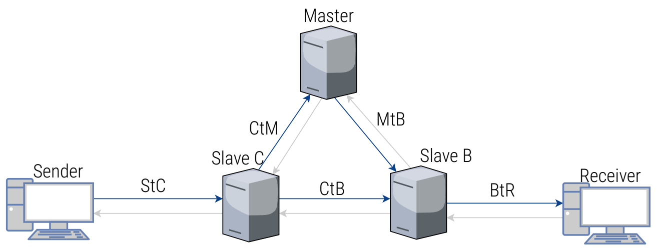


Figure 8.9: Setup for experiments running RabbitMQ in a cluster based setup

> ***Note:*** *The setup seen in figure 8.8 was the intended setup for both HarakaMQ and RabbitMQ, but after configuring RabbitMQ using the method seen in section 8.3.1. The results showed that the actual setup of RabbitMQ worked as seen in figure 8.9. The results conclude that setting up RabbitMQ during the experiments did not yield the desired setup, but the results were still used as it was a cluster based setup. However with no replication of messages between the brokers.*

The setup for HarakaMQ is shown in figure 8.8 while the setup for RabbitMQ is shown in figure 8.9. The different connections are described in table 8.6.

In RabbitMQ the messages sent from the Sender to Slave C are directly passed to Slave B, and are never sent to the Master. As the Master never receives the messages, the setup for RabbitMQ does not replicate the messages between brokers, which was the intended setup. This was discovered when analyzing the results.

In HarakaMQ the messages are passed through the Master and thereafter to Slave B, meaning that no connection between Slave C and Slave B exists. In the setup for HarakaMQ replication between brokers is used.

Abbreviation	Description
StC	Sender to Slave C
CtM	Slave C to Master
CtB	Slave C to Slave B (only used in RabbitMQ)
MtB	Master to Slave B
BtR	Slave B to Receiver

Table 8.6: Description of the different connections in a cluster based setup

8.3.1 RabbitMQ

To configure RabbitMQ to a cluster based setup, the "Clustering Guide" was used [79]. The approach used to configure a RabbitMQ cluster is done by the following steps.

1. **Unique name**

   The first step was to give all the RabbitMQ Brokers a unique name for example *rabbit@rabbitmaster*. This name has to be changed through the Operating System's environment variables [80].

2. **Users**

   Per default a RabbitMQ Broker has a user called *guest* with password *guest*. A new user needs to be created with administrator rights, because the user called *guest* is only allowed to communicate on the *localhost*. This is not gonna work because the virtual machines have to communicate with each other through the bridged adapter.

3. **Erlang cookie**

   To enable RabbitMQ Brokers to communicate with each other they need to share the same Erlang cookie. An Erlang cookie is a string of alphanumeric characters up to 255 characters.

4. **Setup Host file**

   To make it easier to configure the cluster, the Host file can be used to fix IP-addresses to a name such as *192.168.1.76 rabbit@rabbitmaster*. This binds the IP-address *192.168.1.76* to the name *rabbit@rabbitmaster*. Pivotal is encouraging people to use this approach.

5. **Cluster the RabbitMQ Brokers**

   To cluster RabbitMQ Brokers the following commands where required from Slave B, the commands for Slave C are the exact same. The prerequisite for this is that the RabbitMQ Master is running, and all the steps mentioned before this has successfully been completed.

   (a) rabbitmqctl stop_app
   (b) rabbitmqctl join_cluster rabbit@rabbitmaster
   (c) rabbitmqctl start_app

6. **Replicate Queues to all RabbitMQ Brokers**

   To configure RabbitMQ to replicate messages to other Brokers in the system, Highly Available Queues were used. A Highly Available queue based on Pivotal's guide [81] should mirror all queues on to all RabbitMQ Brokers. The following commands were used on the RabbitMQ Master.

   (a) rabbitmqctl stop_app
   (b) rabbitmqctl set_policy ha-all "\." "{""ha-mode"":""all""}"
   (c) rabbitmqctl start_app

7. **Done**

   When all these steps are complete the experiment is ready to commence.

The results for sending 10,000 messages in a cluster based setup with RabbitMQ can be seen in figure 8.10.

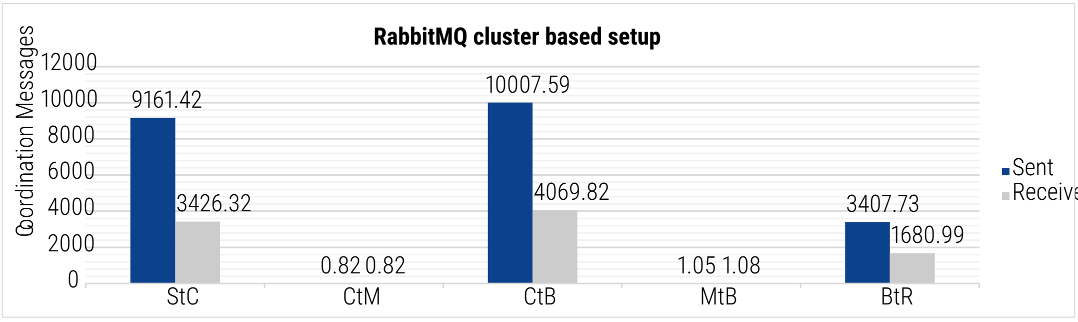


Figure 8.10: Average number of coordination messages used in a RabbitMQ cluster based setup to send 10,000 messages

In figure 8.10 it can be seen that none of the 10,000 sent messages are sent to the Master, but are directly sent from Slave C to Slave B. As the *ha-mode* in RabbitMQ was configured to *all*, the intention was that the Master should have received the messages, as the messages was to be replicated between brokers.

8.3.2 HarakaMQ

Two experiments for a cluster based setup using HarakaMQ were performed with firstly an anti-entropy round time of 500 ms and thereafter 100 ms, to see the variable's influence on the coordination messages and throughput. The experiment with an anti-entropy round time of 100 ms is only used to compare the average number of coordination messages used per 10,000 message and the throughput. The average number of coordination messages used in a cluster based setup for HarakaMQ with a anti-entropy round time of 500 ms can be seen in figure 8.11.

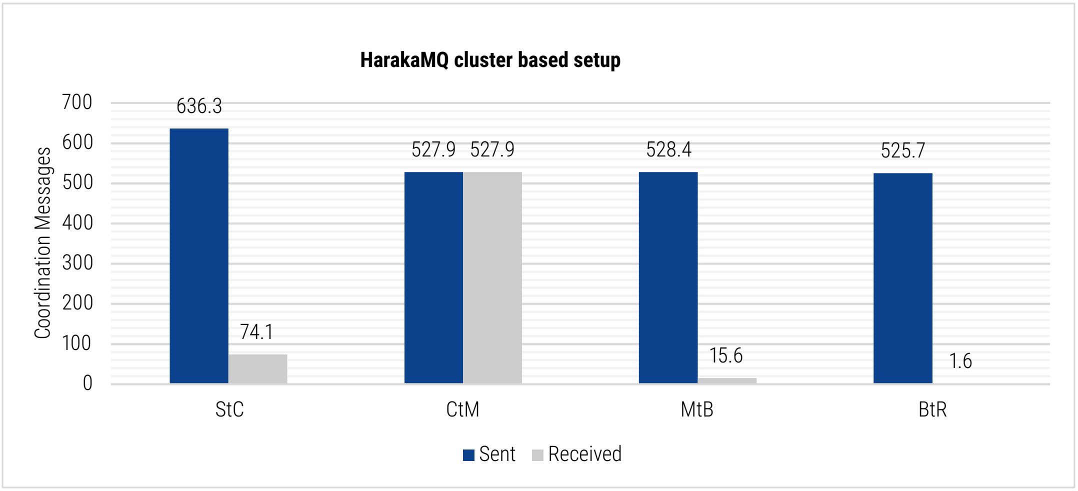


Figure 8.11: Average number of coordination messages used in a HarakaMQ cluster based setup to send 10,000 messages with an anti-entropy round time of 500 ms

The preconfigured variables used in a HarakaMQ cluster based setup, are the same as the preconfigured variables used for the client-server setup. The preconfigured variables can be seen in table 8.4. To calculate the total number of coordination messages required for a cluster based setup, information about the cluster based setup is required. This results in three extra variables which can be seen in table 8.7.

Name	Description	Preconfigured value
N_B	Number of brokers	3
R_T	Run time	7500
A_{RT}	Anti-entropy round time	500

Table 8.7: Variable names and preconfigured values for a cluster based setup

In the cluster based experiment three brokers are used, to replicate and send messages between Sender and Receiver. Run time is the elapsed time for a batch of 10,000 messages to be received in a HarakaMQ cluster based setup, while the anti-entropy round time is the interval for sending gossip messages between brokers.

The values for the variables in table 8.4 and 8.7 are applied to equation 7.9 to calculate the total number of coordination messages required when publishing messages to a slave. In equation 8.3 the variables M_T , P_T and F_M are used. The variable $M_T = 1126.2$ is the number of coordination messages required from a publisher to a subscriber. The variable $P_T = 12.37$ is the estimate of the total number of packets required to send 10,00 messages. The variable $F_M = 45.26$ is the estimation of the number of fragments per packet. These are the same for a client-server setup.

$$\begin{aligned} C_T = 1126.2 + (((\frac{7500}{500} \cdot (3-1)) \cdot 2) \\ -(12.37 \cdot 3)) + (3 \cdot 12.37 \cdot 45.26) = 2856.1 \end{aligned} \tag{8.3}$$

The theoretical number of sent coordination messages is therefore calculated to be 2856.1, while the average number of coordination messages in the experiment are 2837.65 messages, with an anti-entropy round time of 500 ms. The difference between the theoretical and the measured number of messages is 0.65 %.

8.3.3 Comparison of Results

In figure 8.12 the average number of coordination messages used in the cluster based experiments are shown.

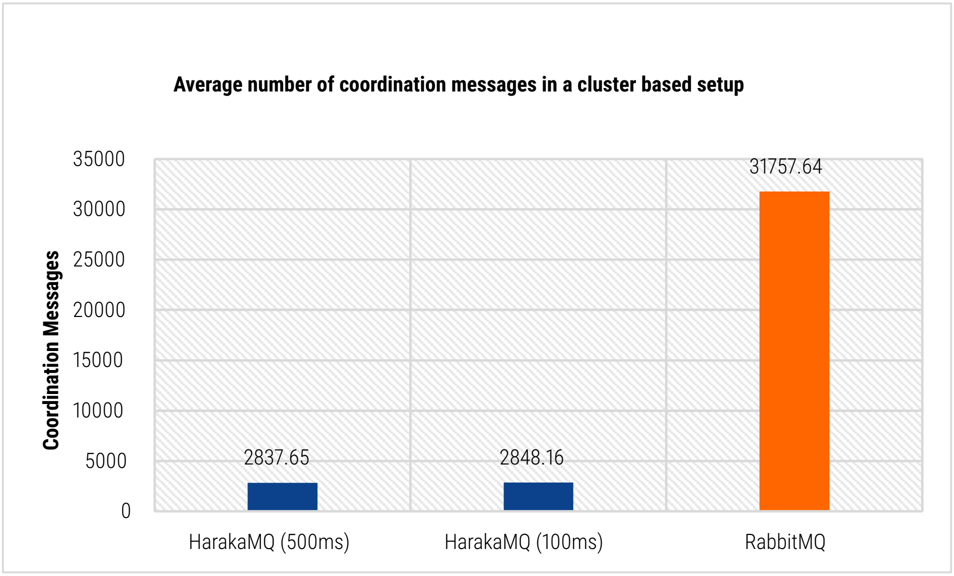


Figure 8.12: The average number of coordination messages used in RabbitMQ and HarakaMQ in a cluster based setup to send 10,000 messages

From the results in figure 8.12 it is seen that running HarakaMQ with an anti-entropy round time of 500 ms, yields a lower amount of coordination messages.

The results also show that around 11 to 12 times as few coordination messages are required in HarakaMQ compared to RabbitMQ. These are the results for RabbitMQ where replication is not enabled, which could imply that even more messages are needed, to ensure the replication of messages between brokers.

The throughput for RabbitMQ and HarakaMQ in a cluster based setup can be seen in table 8.8.

Middleware	**Messages sent**	**Average elapsed time (sec)**	**Throughput (msg/sec)**	**Standard deviation (sec)**
RabbitMQ	10,000	1.097	9118	0.286
HarakaMQ (500 ms)	10,000	7.429	1346	3.441
HarakaMQ (100 ms)	10,000	1.907	5245	0.623

Table 8.8: Throughput for RabbitMQ and HarakaMQ in a cluster setup

The standard deviation for RabbitMQ and HarakaMQ in a cluster based setup shows that the results for RabbitMQ are more stable than the results for HarakaMQ, with a standard deviation of 0.286 seconds.

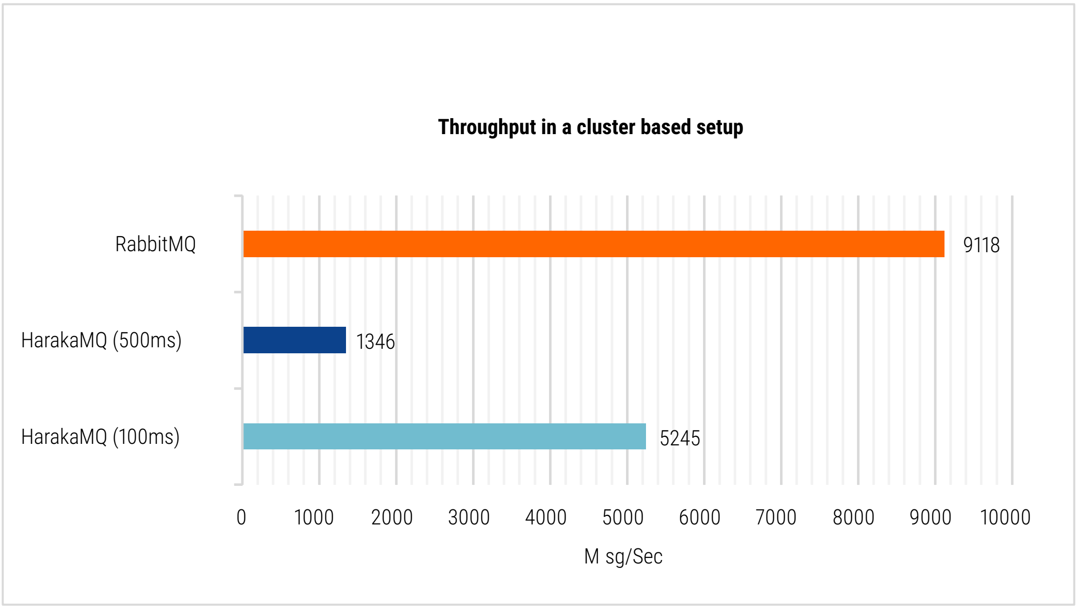


Figure 8.13: Throughput for a cluster based setup in RabbitMQ and HarakaMQ

In figure 8.13 it can be seen that the throughput for RabbitMQ is 9118 messages per second, while for HarakaMQ with an anti-entropy round time of 500 ms the throughput is 1346 messages per second. By changing the anti-entropy round time to 100 ms, the throughput is found to be 5245 messages per second, which yields a better throughput than HarakaMQ with an anti-entropy round time of 500 ms. The experiment running with an anti-entropy round time of 500 ms takes longer than the one running with a round time of 100 ms. The experiment with an anti-entropy round time of 100 ms sends 5 times as many gossip messages per second than the experiment with a anti-entropy round time of 500 ms, and this yields a higher throughput.

8.3.4 Discussion of Results

As in the client-server setup RabbitMQ uses more coordination messages than HarakaMQ, even though RabbitMQ is not replicating the messages to all the RabbitMQ Brokers. This can be seen in the results, as the number of coordination messages between the RabbitMQ Master and the RabbitMQ Slave C is only 0.82 in average per batch both ways. This is not enough coordination messages to replicate the 10,000 messages between the two. More messages would probably be used in RabbitMQ, if messages were replicated to the RabbitMQ Master. It is difficult to say if the throughput would differ for RabbitMQ with the desired setup with replication, as RabbitMQ might still transfer the messages directly between Slave C and Slave B.

For RabbitMQ in a cluster based setup it can be seen that an average of 9161.42 coordination messages have been sent by the Sender to Slave C. This could imply that the Nagle's algorithm [68], which is used in TCP, has bundled some of the 10,000 messages together. When received at Slave C the messages are unbundled and send to Slave B, which could explain why more coordination messages are sent from Slave C to Slave B than from the Sender to Slave C.

The throughput results for HarakaMQ in a cluster based setup is dependent on the anti-entropy round time. When lowering the anti-entropy round time, the number of coordination messages between the brokers increases over time. Even though the HarakaMQ cluster based experiment with an anti-entropy round time of 100 ms provides a better throughput, it results in more coordination messages used, than for the experiment with an anti-entropy round time of 500 ms. The standard deviation shows that the experiments with an anti-entropy round time of 500 ms is more unstable, which could imply a higher loss of messages, and therefore also an increase in coordination messages. Further parameter tuning on the anti-entropy round time is needed to determine the optimal relationship between throughput and coordination messages.

8.4 Connection Disruption with Results and Discussion

An experiment is performed to see how efficient RabbitMQ and HarakaMQ can reestablish a connection after a connection disruption. The setup for the experiment can be seen in figure 8.14.

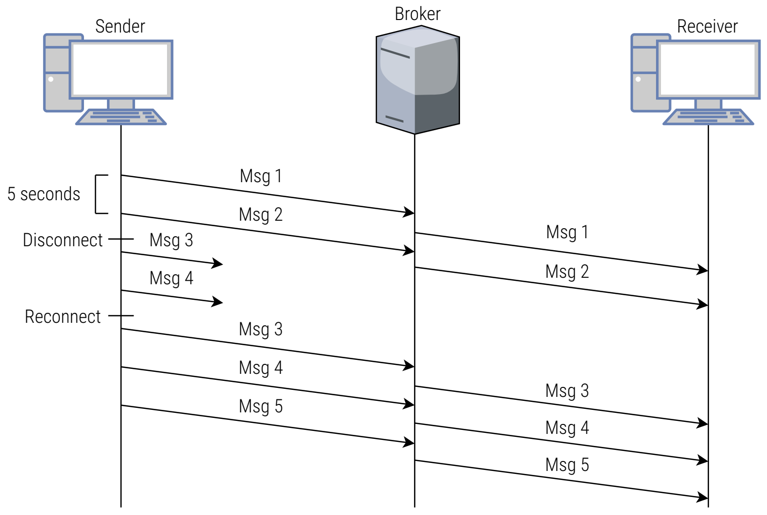


Figure 8.14: Setup for the connection disruption experiment

The connection disruption experiment sends a total of 5 messages from the Sender to the Receiver, by sending a message every 5th second. Message 1 and 2 are sent and received by the Receiver, while message 3 and 4 are sent, but without reaching the Broker, as the Sender is disconnected from the network. The expected result is that message 3 and 4 will be delivered when the connection on the Sender is reestablished.

To create a connection disruption on the Sender, the virtual machine it is running on was disconnected from the network, so no messages could leave or enter the virtual machine. This approach can be compared to removing an Ethernet cable between a computer and a router. It was then connected again later on.

In RabbitMQ message 3, 4 and 5 are never delivered, as the RabbitMQ client throws an error on the Sender when being disconnected. The failure indicates, that to deliver the missing messages a new connection needs to be established, as the old one has been closed. Thereby it is required by the application developer to send message 3,4 and 5 over the new connection.

In HarakaMQ message 3, 4 and 5 are delivered in the correct order when the Sender reconnects. No new connection is required, as HarakaMQ do not use connections.

8.4.1 Discussion of Results

The results show that RabbitMQ cannot handle a connection disruption, while HarakaMQ can. Handling of connection disruptions on IoT and mobile devices are important as they might experience bad network conditions. To ensure handling of connection disruptions within RabbitMQ, the developer using RabbitMQ needs to implement the functionality to handle this, or use a framework on top of RabbitMQ, that handles connection disruptions. Adding the functionality to handle a connection disruption in RabbitMQ adds an extra layer of complexity to the application, and might cause more resource usage when trying to handle a connection disruption, which is not the desired solution when developing IoT or mobile solutions.

8.5 Resource Usage with Results and Discussion

8.5.1 Required Disk Space Comparison

As HarakaMQ is intended to be used on IoT and mobile devices, it is important that HarakaMQ and its dependencies requires a minimum amount of disk space. All versions shown in table 8.9 and 8.10 are x64 bit versions.

.NET Core 2.0	RabbitMQ Client for C# version 5.0.1	Erlang/OTP version 20.2	RabbitMQ Server (Broker) version 3.6.14
389 MB	289KB	319.9 MB	5.44 MB

Table 8.9: Required disk space for RabbitMQ

.NET Core 2.0	HarakaMQ Client version 1.0.0	HarakaMQ Broker version 1.0.0
389 MB	8 KB	1.39 MB

Table 8.10: Required disk space for HarakaMQ

Table 8.9 and 8.10 describes the required disk space for RabbitMQ and HarakaMQ. Erlang is only required when running a RabbitMQ Server, and .NET Core 2.0 is not required for the RabbitMQ Server. RabbitMQ supports different programming languages and it is not required to use .NET Core 2.0 to run the RabbitMQ Clients. As it is the RabbitMQ C# Client which has been used for the experiments in this thesis, it is the requirement for the RabbitMQ C# Client which have been listed.

8.5.2 Comparison of Resource Usage Results

Three measurements have been chosen to compare the resource usage between RabbitMQ and HarakaMQ. All the abbreviations in figure 8.15, 8.16 and 8.17 are explained in table 8.11. All measurements are based on the sender and receiver. Brokers are not a part of the comparison as a broker is not supposed to run on an IoT or mobile device.

To get the data for Windows 10 and .NET Core 2.0 a virtual machine running nothing was used for the Windows 10 experiment, and a simple .NET Core 2.0 application was used for the .NET Core 2.0 experiment. The consumption was logged over a time window of two minutes. The time window of two minutes was chosen because the client-server experiments took approximately 2-3 minutes.

Abbreviation	Description
CS	Client-server setup
C	Cluster based setup
C-500	Anti-entropy round time preconfigured to 500 ms
C-100	Anti-entropy round time preconfigured to 100 ms
(CS-1 -> CS-3)	Average of client-server setup experiments CS-1, CS-2 and CS-3
(C-500 -> C-100)	Average of cluster based setup experiments with anti-entropy round time of 500 ms and 100 ms

Table 8.11: Description of the abbreviations in figure 8.15, 8.16 and 8.17

In figure 8.15, 8.16 and 8.17 some of the labels has the name *Rabbit(CS-1)* or *Rabbit(CS-2)*. *Rabbit(CS-1)* is the first experiment, and *Rabbit(CS-2)* is the second experiment. The same test procedure has been applied for both experiments. The reason for doing multiple of the same experiments for RabbitMQ in a client-server setup is to see if the results could be reproduced. The RabbitMQ Server version 3.6.14 and RabbitMQ Clients have been reinstalled on the virtual machines between experiments *Rabbit(CS-1)*, *Rabbit(CS-2)* and *Rabbit(CS-3)*, to ensure the independence of the experiments.

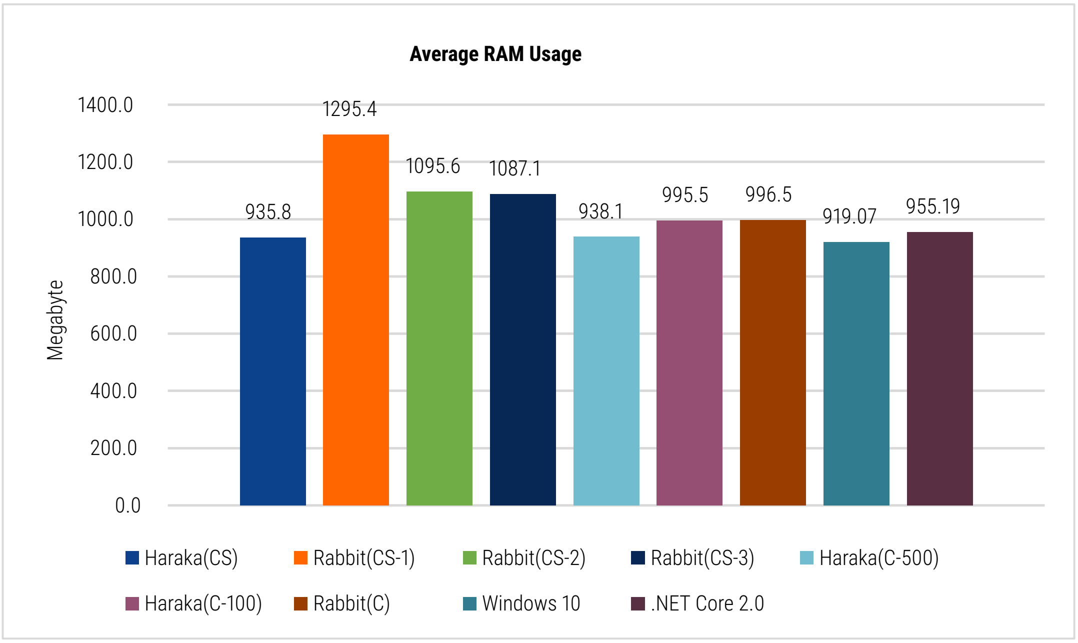


Figure 8.15: Average RAM usage in Megabytes for all client-server and cluster based experiments

Figure 8.15 shows the average RAM usage through all 100 rounds. RAM usage is represented in Megabytes (MB).

The average RAM usage of *Rabbit(CS-1 -> CS-3)* in a client-server setup, is 23.89 % MB RAM higher than *Haraka(CS)*.

The cluster based setup textitHaraka(C-500), uses 5.77 % MB less RAM in average, than *Haraka(C-100)*.

Rabbit(C) in a cluster based setup, uses 3.07 % more MB RAM than *Haraka(C-500 -> C-100)*.

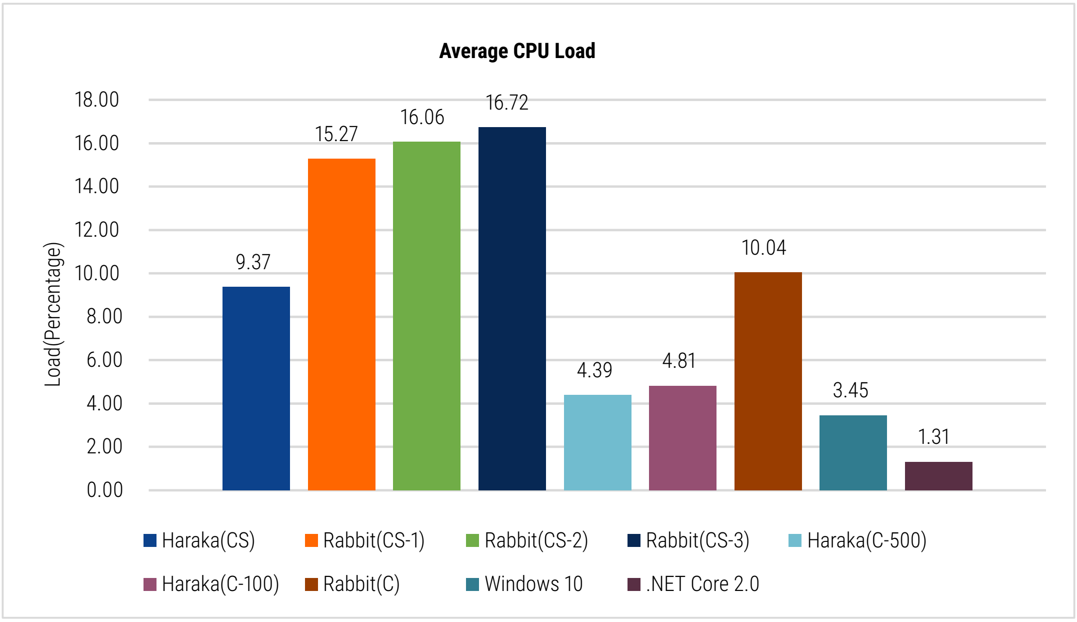


Figure 8.16: Average CPU Load in percentage for all client-server and cluster based experiments

Figure 8.16 shows the average CPU Load in percentage through all 100 rounds.

The average CPU load of *Rabbit(CS-1 -> CS-3)* in a client-server setup, reaches a load of 70.94 % more in average than *Haraka(C-100)*.

For the cluster based setup *Haraka(C-500)*, the CPU reaches a load of 8.73 % less in average, than *Haraka(C-100)*.

Rabbit(C) in a cluster based setup, loads the CPU 118.26 % more in average than *Haraka(C-500 -> C-100)*.

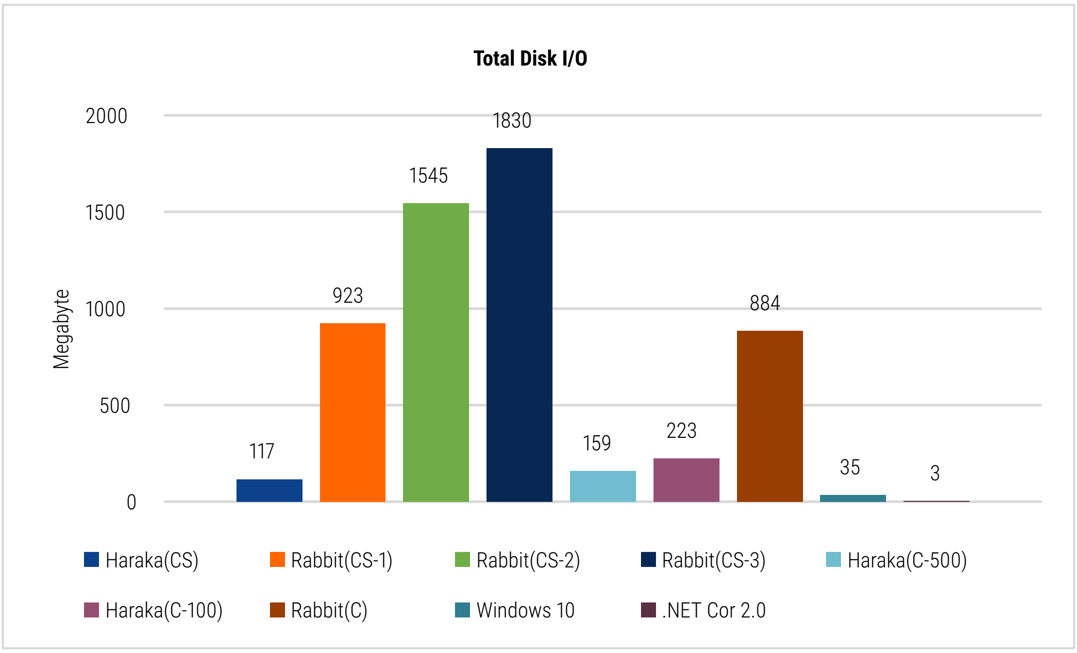


Figure 8.17: Total Disk I/O for all client-server and cluster based experiments

Figure 8.17 shows the total Read/Write (I/O) over all 100 rounds. Total Disk I/O is represented in Megabytes. What is meant by I/O, is that for example *Haraka(CS)* has written and read a total of 117 MB in the course of the whole experiment. So 117 MB has been used to process $10,000(messages) \cdot 100(rounds) = 1,000,000 messages$.

For the client-server setup *Rabbit(CS-1 -> CS-3)* has written and read 1124.50 % more MB than *Haraka(CS)*.

For the cluster based setup *Haraka(C-100)* there has been written and read 40.25 % more MB than *Haraka(C-500)*.

Rabbit(C) has written and read 362.83 % more MB than *Haraka(C-500 -> C-100)*.

8.5.3 Discussion of Results

By looking at the required disk space for HarakaMQ and RabbitMQ it is seen that the size of the clients or brokers does not require much disk space, but it is the required frameworks, which requires the most disk space.

RabbitMQ generally uses more resources than HarakaMQ. The difference is not that large when looking at RAM usage, but the CPU usage and especially the disk I/O showed a big difference. For example *Rabbit(CS-1 -> CS-3)* uses 1124.50 % more disk I/O than *Haraka(CS)*. RabbitMQ has logging capabilities which HarakaMQ does not. Per default RabbitMQ Brokers logs to a file with the default level *info* [82]. The log level *info*, logs info such as RabbitMQ Broker initialization, connection establishment and connection disruption. The logging does not apply to RabbitMQ Clients, hence the logging can be excluded as the reason for the high disk I/O. It is difficult to argue about what the disk I/O originates from in RabbitMQ, and more research and data are needed.

The RAM usage of .NET Core 2.0 is larger than Haraka(CS), this should not be possible as the HarakaMQ Client presumably uses more resources than an idle console application made in .NET Core 2.0. A reason for this could be that Windows 10 ran something in the background at the time of the experiment, which is not controllable. To reduce these kind of errors a more controllable test environment should be used. For example avoiding using Windows 10, even though a windows 10 version was installed with nothing than the necessary software, windows 10 still has a lot of uncontrollable jobs/tasks running in the background. This is also obvious when looking at the disk I/O for .NET Core 2.0 which is 3 MB, and that is lower than for Windows 10, which could indicate Windows 10 ran something in the background when the Windows 10 consumption was measured. An alternative for this would be using Windows 10 IoT Core instead. Windows 10 IoT Core is optimized for smaller devices, and has less features than Windows 10. If Windows 10 IoT Core is combined with automation to perform experiments, which consist of deployment, startup and tear down. This combination of automation and Windows 10 IoT Core would probably lower the error of the OS contaminating the results, and make the experiments less cumbersome to perform.

8.6 Remarks

It is not relevant to test and compare how many coordination messages it takes to keep a connection alive, as there is no connection to keep alive in HarakaMQ, and thereby no coordination messages are needed. In RabbitMQ a heartbeat message is used to detect a dead TCP connection, per default RabbitMQ sends a heartbeat message each 30 seconds [83]. A thing to note is that all messages are counted as a heartbeat message in RabbitMQ (e.g. protocol, publish, acknowledgment messages).

Besides RabbitMQ's heartbeat messages, TCP has a build in feature for this, which is called keep-alive messages. Per default on a Linux system when there is silence on the channel it takes 7200 seconds before the first keep-alive message is send [84]. After the first keep-alive message is send, the interval is per default 75 seconds, and the TCP protocol will probe 9 times before it deems the connection as being dead.

RabbitMQ's heartbeat messages is used to complement the default behavior of the TCP protocols keep-alive messages.

9 | Discussion

The results in chapter 8 have shown that a competitive solution has been created in this thesis, especially the client-server setup has achieved great results.

The client-server setup had a higher throughput and used fewer coordination messages than RabbitMQ. However, increasing the amount of messages per batch will eventually cause problems for the throughput of HarakaMQ. The reason for this is the bottleneck of the storage solution used in HarakaMQ.

The difference between the theoretical number of coordination messages calculated and the actual measured number of coordination messages is rather low for both the client-server setup and the cluster based setup. The difference between the numbers lies in the fragmentation part of the equation, as the fragmentation is based on the actual measured header size. In HarakaMQ version 1.0.0 the message header size is preconfigured to 50 bytes which is the maximum size of the message header. An alternative could be to measure the message header size at runtime, to get a more precise package calculation. This could have a positive impact on performance of the developed solution, because messages per packet would probably increase, hence fewer packets needs to be transmitted. The downside is that it takes processing time to measure the actual message header size.

The throughput results for HarakaMQ and RabbitMQ in a cluster based setup, indicates that HarakaMQ can over time most likely be optimized to give a better throughput than RabbitMQ. HarakaMQ is still in its early stages of development, and a lot of improvement can be done, like improving the storage solution, converting the message header to a byte array and improving the replication of messages, for example by adding a consensus protocol. With a consensus protocol it is not necessary to replicate the messages to all the HarakaMQ Brokers at once, but only a part of the HarakaMQ Brokers needs to agree on the order of the messages.

The number of coordination messages used by HarakaMQ to transfer 10,000 messages from one publisher to one subscriber, either in a client-server setup or a cluster based setup, is lower than RabbitMQ. This is because UDP is used instead of TCP. The reason for this is that UDP does not require acknowledgments and a connection establishment, and also because the bundling approach in HarakaMQ is different from the one used in RabbitMQ.

Replication for RabbitMQ was unsuccessfully configured, even though the official guide from Pivotal was used. This means that two cluster setups with replication could not be tested. If RabbitMQ was configured correctly, it would yield more comparable results between HarakaMQ and RabbitMQ, and presumably the throughput would be lower and the number of coordination messages used would be higher.

RabbitMQ failed the connection disruption experiment, while HarakaMQ passed. The reason for RabbitMQ failing is that the AMQP connection needs to be kept alive. The application developer needs to create a new connection when the old one cannot be used. Which means that a lot of coordination messages are required to re-establish the connection. This is not ideal when using an IoT device which has to go into sleep mode to preserve energy. HarakaMQ is designed to handle these kinds of issues. A problem with sending messages without knowing that the receiver is ready to receive, is that a lot of messages is sent into the network, which means wasted bandwidth, and this could be helped by creating a sliding window.

HarakaMQ Clients had a lower resource usage than RabbitMQ in all measured resources, most noticeable in disk I/O. It is difficult to reason about the high disk I/O of RabbitMQ Clients from the data obtained in these experiments, and it would require further research and data to find the cause behind the high disk I/O.

To make a comparison between the developed solution HarakaMQ and RabbitMQ a test environment was set up. Three virtual machines for each solution with the exact same characteristics were used for a client-server setup (two clients and one broker), and five virtual machines for the cluster setup (two clients and three brokers). The test environment using virtual machines should probably be optimized if further experiments were to be done. More automation of the experiments would have been preferred, for example to automate the deployment, start up and tear down of the experiments.

One could argue that it is not wise to create a packet with a size over 1500 bytes as they will get fragmented by the IPv4 protocol. Limiting the packet size to 1500 bytes instead of 65,507 bytes will create more overhead, because a packet header is needed for every packet. The fragmentation is needed because of devices, such as IoT and mobile devices, can have lower MTU's which is necessary to support. The solution was to make it possible for the application developer to disable fragmentation. A relevant experiment would be to see if there is any difference in the throughput when fragmentation is used, or when it is not used.

HarakaMQ is able to facilitate the communication between IoT and mobile devices. Furthermore HarakaMQ uses lower resources and coordination messages than the TCP based message oriented middleware RabbitMQ, and it would presumably use less resources and coordination messages than the TCP based application protocol HTTP, which is used in MobilePay. However, further experiments are required to argue about the resource consumption in the HTTP protocol.

10 | Concluding Remarks and Future Work

10.1 Conclusion

In this thesis a fast, reliable and light-weight message oriented middleware based on UDP has successfully been designed and implemented. By using a low amount of coordination messages, HarakaMQ is competitive with state of the art message oriented middleware such as RabbitMQ, with regards to reliability, throughput and replication of messages between brokers. HarakaMQ is able to handle IoT related problems, such as the IoT device going into sleep mode and connection disruptions. The results in this thesis show that further experiments should be done to tune the parameters in a cluster based setup for HarakaMQ, and thereby giving the optimal relationship between anti-entropy round time, coordination messages and throughput.

HarakaMQ in a Client-Server setup uses fewer coordination messages than RabbitMQ, while also providing a higher throughput.

HarakaMQ in a cluster based setup uses fewer coordination messages than RabbitMQ, but provides a lower throughput. The desired setup for RabbitMQ in a cluster based setup was not reached while performing the experiments. Experiments based on the desired cluster based setup for RabbitMQ might yield different results than the ones obtained in this thesis.

The mathematical model of HarakaMQ provides the total number of coordination messages used to send a specific number of messages for a client-server and a cluster based setup. The difference between the calculated and measured number of messages, indicates that the mathematical model of HarakaMQ can be used to produce a relatively close estimate of the total number of coordination messages required for sending a specific number of messages from a publisher to a subscriber.

The results show that HarakaMQ uses fewer resources and a lower number of coordination messages for the same amount of information than RabbitMQ, and thereby reaching a lower energy consumption. Furthermore the information capacity in the network has been enhanced, by lowering the number of coordination messages needed to transfer a specific number of messages. All this makes it more suitable for IoT or mobile devices than RabbitMQ, with regards to energy consumption.

To ensure the quality of HarakaMQ, requirements were created based on the ISO/IEC 25010:2011 standard. Not all requirements were fulfilled throughout this thesis, and future work could include the fulfillment of these requirements. The same fault tolerance as RabbitMQ was not achieved in this thesis, as it is not possible to handle a system failure of a HarakaMQ Client or a HarakaMQ Broker in version 1.0.0 of HarakaMQ.

The experiments performed substantiates the problem formulation proposed in this thesis, and gives reason to believe that a message oriented middleware based on UDP can replace and improve the current solution in MobilePay.

10.2 Contributions

The current message oriented middleware based on TCP are not suitable for IoT and mobile devices, therefore a new kind og message oriented middleware is needed. The developed message oriented middleware HarakaMQ is proposed as a possible solution. HarakaMQ provides the needed reliability and message delivery guarantees, and thereby making it suitable for usage on IoT and mobile devices.

The research within distributed systems, and the research of CAP are combined to provide the desired solution of HarakaMQ. HarakaMQ is a message oriented middleware with reliability build on top of UDP, which according to research done within this thesis, is the first of its kind.

This thesis opens up for a whole new research area within message oriented middleware. It thereby also encourage to further experiments and research within this area, to determine the true potential of a message oriented middleware based on UDP.

10.3 Personal Outcome

The work performed in this thesis has given us experience in using academical research papers and books to conduct research, and combining old as well a new theories to create and better understand distributed systems, especially message oriented middleware.

Before the thesis work we had a more broad understanding of distributed systems and message oriented middleware, compared to after the thesis work, where we have an expert knowledge in message oriented middleware and how to design and implement these.

The cooperation with Danske Bank and MobilePay gave us an insight in how a message oriented middleware is used on a daily basis. It also gave an insight about problems and limitations within current state of the art message oriented middleware on the market. The overall cooperation with Danske Bank and MobilePay also gave us experience in doing scientific work within the industry.

By working with HarakaMQ we recognized, that creating a fully functional middleware is complex, and requires a lot of hard and dedicated work.

The experience gained in this thesis work gave us the desire to continue working with HarakaMQ, as we feel that we have found an unexplored research area within message oriented middleware, and the developed solution has a big potential with many improvement possibilities.

10.4 Future Work - Outlook and Perspective

To enhance the user friendliness of the solution a management web application could be created to control the HarakaMQ system, and get live statistic of the performance of the HarakaMQ Brokers and HarakaMQ Clients in the system. This could also be used to check load balance, and check if the hardware needs to be upgraded to enhance the performance of the system.

Implementation of the Congestion Control and Dynamic Router components are to be done before HarakaMQ can be sent into production. The Dynamic Router component could use machine learning algorithms to predict the load of certain HarakaMQ Brokers, and thereby decide which HarakaMQ Broker to send the messages to at a given time. An implementation of the Dynamic Router is also needed, to ensure automatic selection of a HarakaMQ Broker, based on the information given from the Congestion Control component.

As the Dynamic Router is to select a specific HarakaMQ Broker, to send the messages to, it needs to know the IP address and port of the available HarakaMQ Brokers. A solution for this could be to implement a discovery protocol, and thereby making it possible for HarakaMQ Clients or HarakaMQ Brokers to discover the available HarakaMQ Brokers in a system.

In version 1.0.0 of HarakaMQ the limit size for a message is 65,507 bytes, which is the limit defined by the IPv4 and UDP protocol. Fragmentation can be implemented in the UDP Communication module in HarakaMQ, to support larger messages. The fragmentation of larger messages could give a performance boost to HarakaMQ in cluster based setups, as more data can be send in each anti-entropy round. Instead of HarakaMQ sending one 65,507 bytes sized packet per anti-entropy round, multiple packets can be sent, and thereby fewer anti-entropy rounds will be used to replicate data.

To enhance the performance of the cluster based setup in HarakaMQ a throttle mechanism could be implemented to automatically adjust the anti-entropy round time dependent on the load, hence improving the throughput and lowering the number of coordination messages used over time.

An improvement to the current solution would be to enhance the storage solution. The current storage solution has a problem when handling a larger number of data (1,000,000 messages) at the same time. This could be improved by introducing indexing. By indexing the data into fields, only a field is changed instead of a whole file. A method has not been discovered yet in this thesis to perform such an indexing directly in files. Other solutions such as a database per thread should be investigated, to see if it can improve the performance when dealing with a larger number of data, and still minimizing thread synchronization.

The current message header in HarakaMQ is a JSON string, but by converting the header to a byte header like TCP, the header size and message size could be decreased. The decrease in header and thereby also message size, could result in more messages being bundled when sending packets in HarakaMQ, and thereby also decrease the total amount of coordination messages needed.

With the achieved results in this thesis and the mentioned future work, a big potential is shown in HarakaMQ. A potential which if is realized, can lead to a revolution for message oriented middleware with focus on IoT and mobile devices.

List of Figures

List of Tables

A | Evaluation Results of Requirements

If all sub-requirements has a ✓ the requirement has passed. The evaluation was performed on the 27th December 2017.

A.1 Functional Suitability

1. HarakaMQ shall be able to perform interprocess communication in a distributed system

1.1	HarakaMQ is performing interprocess communication via the User Datagram Protocol	✓
1.2	HarakaMQ is using the publish/subscribe message pattern for interprocess communication	✓

Table A.1: Results of sub-requirements for requirement 1

A.2 Performance Efficiency

2. HarakaMQ shall be able to perform better than RabbitMQ

2.1	HarakaMQ is able to process a better average throughput of messages than RabbitMQ, when RabbitMQ is configured to a client-server setup	✓
2.2	HarakaMQ is able to process a better average throughput of messages than RabbitMQ, when RabbitMQ is configured to a cluster based setup	%

Table A.2: Results of sub-requirements for requirement 2

3. HarakaMQ shall be able to run on IoT devices

3.1	HarakaMQ is able to run on a Raspberry Pi 3 Model B [23]	✓

Table A.3: Results of sub-requirements for requirement 3

A.3 Portability

4. HarakaMQ shall be able to execute on several platforms

4.1	HarakaMQ is installable on Windows 10, Ubuntu 17.04 and Windows 10 IoT Core	✓

Table A.4: Results of sub-requirements for requirement 4

A.4 Reliability

5. HarakaMQ shall be able to re-establish the desired state of the system if a system failure or connection disruption occurs

5.1	HarakaMQ is able to re-establish the desired state after a connection disruption	✓
5.2	HarakaMQ is able to re-establish the desired state after a system failure	%

Table A.5: Results of sub-requirements for requirement 5

6. HarakaMQ shall be able to handle a system failure or connection disruption of of a system component which never returns to the system again

6.1	HarakaMQ is able to handle if a system component experience a failure and never returns to the system	%
6.2	HarakaMQ is able to replicate data to multiple system components, to avoid loss of data	✓

Table A.6: Results of sub-requirements for requirement 6

7. HarakaMQ shall always be available

7.1	HarakaMQ guarantees to always be available	✓

Table A.7: Results of sub-requirements for requirement 7

B | Attachments

A Zip has been provided with the attachments of this thesis. Each attachment in the Zip is given an ID.

1. HarakaMQ - Reliable and Lightweight Message Oriented Middleware Based in UDP
2. Source code for HarakaMQ Client, HarakaMQ Broker, UDP Communication Module and HarakaMQ storage solution (Visual Studio 2017 project)

The experiments for HarakaMQ can be found on Github at `https://rotvig.github.io/HarakaMQ-Benchmark/`.

The experiments for RabbitMQ can be found on Github at `https://rotvig.github.io/RabbitMQ-Benchmark/`.

Bibliography

[1] (2017). "haraka" in English, Oxford University Press, [Online]. Available: https://en.bab.la/dictionary/swahili-english/haraka (visited on 12/14/2017).

[2] (2016). Roundup Of Internet Of Things Forecasts And Market Estimates, Forbes, [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/#26f4fad6292d (visited on 09/15/2017).

[3] (2017). RabbitMQ, Pivotal, [Online]. Available: https://www.rabbitmq.com (visited on 09/05/2017).

[4] W. Shang, Y. Yu, R. Drom, and L. Zhang, "Challenges in IoT Networking via TCP/IP Architecture", Oct. 2016. [Online]. Available: https://named-data.net/wp-content/uploads/2016/02/ndn-0038-1-challenges-iot.pdf (visited on 09/18/2017).

[5] S. Hosio, D. Ferreira, J. Goncalves, N. van Berkel, C. Luo, M. Ahmed, H. Flores, and V. Kostakos, "Monetary assessment of battery life on smartphones", *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 1869–1880, 2016. [Online]. Available: http://www.denzilferreira.com/papers/2016/chi16b.pdf (visited on 12/13/2017).

[6] (2017). Reliability Guide, Pivotal, [Online]. Available: https://www.rabbitmq.com/reliability.html (visited on 09/25/2017).

[7] (2017). Best Message Queue Solutions, IT Central Station, [Online]. Available: https://www.itcentralstation.com/categories/message-queue (visited on 09/14/2017).

[8] F. Wortmann and K. Flüchter, "Internet of Things, Technology and Value Added", *Business & Information Systems Engineering Journal 57, 221-224*, Mar. 2015.

[9] J. Fritsch and C. Walker, "CMQ - A lightweight, asynchronous high-performance messaging queue for the cloud", *Journal of Cloud Computing*, Aug. 2012. [Online]. Available: https://doi.org/10.1186/2192-113X-1-20 (visited on 09/25/2017).

[10] (2017). Haskell, Haskell, [Online]. Available: https://www.haskell.org/ (visited on 09/25/2017).

[11] (2017). Erlang, Ericsson, [Online]. Available: https://www.erlang.org/ (visited on 09/25/2017).

[12] L. Minas and B. Ellison, "The Problem of Power Consumption in Servers", 2009.

[13] (2017). About us, Danske Bank, [Online]. Available: https://danskebank.com/about-us (visited on 01/01/2018).

[14] (2017). How it all started, MobilePay, [Online]. Available: https://www.mobilepay.dk/da-dk/Pages/The-story-in-English.aspx (visited on 10/11/2017).

[15] (2017). Learn about Windows 10 IoT Core, Microsoft, [Online]. Available: https://developer.microsoft.com/da-dk/windows/iot/Explore/IoTCore (visited on 10/11/2017).

[16] (2017). Prerequisites for .NET Core on Windows, Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/windows-prerequisites?tabs=netcore2x (visited on 01/01/2018).

[17] (2017). Prerequisites for .NET Core on Linux, Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites?tabs=netcore2x (visited on 01/01/2018).

[18] (2017). Prerequisites for .NET Core on macOS, Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites?tabs=netcore2x (visited on 01/01/2018).

[19] (2017). Recommended Minimum System Requirements, Ubuntu, [Online]. Available: https://help.ubuntu.com/community/Installation/SystemRequirements (visited on 10/11/2017).

[20] (2017). MacOS Sierra - Technical Specifications, Apple, [Online]. Available: https://support.apple.com/kb/sp742?locale=en_US (visited on 10/11/2017).

[21] ISO/IEC, *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. ISO/IEC, 2011. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (visited on 08/29/2017).

[22] S. Wagner, *Software Product Quality Control*. Springer, 2013, pp. 29–31, ISBN: 978-3-642-38570-4.

[23] R. P. Foundation. (2017). Products: RASPBERRY PI 3 MODEL B, [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (visited on 08/29/2017).

[24] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed systems, Concepts and design*. Pearson Education, 2001.

[25] D. E. Comer, *Internetworking with TCP/IP, Principles, Protocols, and Architectures Fourth Edition*. Prentice Hall, 2006, ISBN: 0-13-018380-6.

[26] M. Zhang, "Major Automatic Repeat Request Protocols Generalization and Future Develop Direction", *2013 6th International Conference on Information Management, Innovation Management and Industrial Engineering*, 2013.

[27] C. S. Guynes and J. Windsor, "Revisiting Client/Server Computing", *Journal of Business & Economics Research, Volume 9, Number 1*, Jan. 2011.

[28] R. Schollmeier, "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications", 2001.

[29] V. Beal. (2006). The Differences Between Thick & Thin Client Hardware, [Online]. Available: http://www.webopedia.com/DidYouKnow/Hardware_Software/thin_client.asp (visited on 10/01/2017).

[30] A. O. Ramirez, "Three-Tier Architecture", *Linux Journal*, vol. Issue 75, Jul. 2000. [Online]. Available: http://www.linuxjournal.com/article/3508 (visited on 10/30/2017).

[31] B. Kahanwal and T. Pal Singh, "The Distributed Computing Paradigms: P2P, Grid, Cluster, Cloud, and Jungle", *International Journal of Latest Research in Science and Technology*, vol. 1 Issue 2, Jul. 2012.

[32] (2016). IBM Cluster systems, Benefits, IBM, [Online]. Available: https://web.archive.org/web/20160429022854/http://www-03.ibm.com/systems/clusters/benefits.html (visited on 10/19/2017).

[33] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec, "The Many Faces of Publish/Subscribe", *ACM Computing Surveys, Vol. 35, No. 2*, Jun. 2003.

[34] A. Piper, "Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP", Feb. 2013. [Online]. Available: https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html (visited on 09/18/2017).

[35] (2017). AMQP, OASIS, [Online]. Available: http://www.amqp.org (visited on 09/18/2017).

[36] G. M. Roy, *RabbitMQ in Depth*. Manning Publications, 2017, pp. 18–38, ISBN: 9781617291005.

[37] (2017). AMQP 0-9-1 Model Explained, Pivotal, [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html (visited on 09/19/2017).

[38] (2017). MQTT, OASIS, [Online]. Available: http://mqtt.org (visited on 09/18/2017).

[39] (2017). STOMP, [Online]. Available: http://stomp.github.io/ (visited on 09/18/2017).

[40] E. A. Brewer, "Towards robust distributed systems", Portland, Oregon: In Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, Jul. 2000.

[41] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", *In ACM SIGACT News*, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.1495 (visited on 08/29/2017).

[42] W. Vogels, "Eventually Consistent", *Queue - Scalable Web Services*, vol. 6 Issue 6, Aug. 2008. [Online]. Available: https://dl.acm.org/citation.cfm?doid=1466443.1466448 (visited on 11/07/2017).

[43] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond", *Queue*, vol. 11 Issue 3, Mar. 2013.

[44] (2017). PBS in Brief - What is PBS and why should you care?, UC Berkeley, [Online]. Available: http://pbs.cs.berkeley.edu/ (visited on 11/20/2017).

[45] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and System*, vol. 4 No. 3, 1982. [Online]. Available: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fpeople%2Flamport%2Fpubs%2Fbyz.pdf (visited on 11/07/2017).

[46] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", *Operating Systems Review*, vol. 29 No. 5, 1995.

[47] A. Montresor, "Gossip and Epidemic Protocols", *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1997. [Online]. Available: http://disi.unitn.it/~montreso/ds/papers/montresor17.pdf (visited on 11/13/2017).

[48] E. Technologies, "Precision Time Protocol (PTP/IEEE-1588)", *White paper*, [Online]. Available: https://www.endruntechnologies.com/pdf/PTP-1588.pdf (visited on 12/06/2017).

[49] J. Eidson and K. Lee, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", Nov. 2002.

[50] D. L. Mills, "Internet time synchronization: The network time protocol", *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991, ISSN: 0090-6778. DOI: 10.1109/26.103043.

[51] C. J. Fdige, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering", 1988.

[52] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehard, and D. Terry, “Epidemic Algorithms for Replicated Database Maintenance”, *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987. [Online]. Available: https://www.cis.upenn.edu/~bcpierce/courses/dd/papers/demers-epidemic.pdf (visited on 11/10/2017).

[53] S. Ranganathan, A. D. George, R. W. Todd, and M. C. Chidester, “Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters”, *Cluster Computing*, vol. 4, 2001.

[54] B. Pittel, “On Spreading a Rumor”, *SIAM Journal on Applied Mathematics*, vol. 47, 1987.

[55] N. Foundation. (2017). Home repository for .NET Core, Microsoft, [Online]. Available: https://github.com/dotnet/core (visited on 08/29/2017).

[56] (2017). TIOBE Index for September 2017, TIOBE, [Online]. Available: https://www.tiobe.com/tiobe-index/ (visited on 09/14/2017).

[57] (2017). C# .NET Core programs versus C++ g++, The Computer Language Benchmarks Game, [Online]. Available: https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=csharpcore&lang2=gpp (visited on 09/14/2017).

[58] (2017). C# .NET Core programs versus Java, The Computer Language Benchmarks Game, [Online]. Available: https://benchmarksgame.alioth.debian.org/u64q/csharp.html (visited on 09/14/2017).

[59] (2017). Handling and Raising Events, Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/events/ (visited on 09/11/2017).

[60] R. Chen. (2006). Performance consequences of polling, Microsoft, [Online]. Available: https://blogs.msdn.microsoft.com/oldnewthing/20060124-17/?p=32553 (visited on 09/11/2017).

[61] (2017). Task Class, MSDN Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/system.threading.tasks.task(v=vs.110).aspx (visited on 08/30/2017).

[62] (2017). ThreadPool Class, MSDN Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/system.threading.threadpool(v=vs.110).aspx (visited on 08/30/2017).

[63] I. E. T. Force, *RFC: 791*, Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc791 (visited on 12/07/2017).

[64] L. Lamport, “Time, clocks and the ordering of events in a distributed system”, pp. 558–565, 1978. [Online]. Available: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/ (visited on 12/06/2017).

[65] (2017). RabbitMQ .NET client for .NET Core and .NET 4.5.1+, Pivotal, [Online]. Available: https://github.com/rabbitmq/rabbitmq-dotnet-client (visited on 08/22/2017).

[66] G. Hohpe and B. Woolf, *Enterprise Integration Patterns, Designing, Building, and Deploying Messaging Solutions*. Pearson Education, Oct. 2003, ISBN: 978-0-321-20068-6.

[67] D. C. Schmidt and C. D. Cranor, “Half-Sync/Half-Async, An Architectural Pattern for Efficient and Well-structured Concurrent I/O”, 1995. [Online]. Available: http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf (visited on 12/03/2017).

[68] J. Nagle, “Congestion Control in IP/TCP Internetworks”, 1984. [Online]. Available: https://tools.ietf.org/html/rfc896 (visited on 12/20/2017).

[69] (2017). lock Statement, Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement (visited on 12/20/2017).

[70] M. Fowler. (2004). Inversion of Control Containers and the Dependency Injection pattern, [Online]. Available: https://martinfowler.com/articles/injection.html (visited on 09/04/2017).

[71] Open-source. (2017). Simple Injector, [Online]. Available: https://simpleinjector.org/index.html (visited on 09/04/2017).

[72] (2017). The MIT License, Open Source Initiative, [Online]. Available: https://opensource.org/licenses/MIT (visited on 09/04/2017).

[73] danielpalme. (2017). Ioc Performance, [Online]. Available: https://github.com/danielpalme/IocPerformance (visited on 09/04/2017).

[74] (2017). MessagePack-CSharp, MessagePack, [Online]. Available: https://github.com/neuecc/MessagePack-CSharp (visited on 12/18/2017).

[75] V. John and X. Liu, "A Survey of Distributed Message Broker Queues", *CoRR*, vol. abs/1704.00411, 2017. [Online]. Available: http://arxiv.org/abs/1704.00411 (visited on 09/28/2017).

[76] (2017). VirtualBox, Oracle, [Online]. Available: https://www.virtualbox.org/ (visited on 12/16/2017).

[77] (2017). Virtual networking, Oracle, [Online]. Available: https://www.virtualbox.org/manual/ch06.html#network_bridged.

[78] (2017). HWiNFO, diagnostic software, [Online]. Available: https://www.hwinfo.com/ (visited on 12/20/2017).

[79] (2017). Clustering Guide, Pivotal, [Online]. Available: https://www.rabbitmq.com/clustering.html (visited on 12/16/2017).

[80] (2017). RabbitMQ Configuration, Pivotal, [Online]. Available: https://www.rabbitmq.com/configure.html (visited on 12/16/2017).

[81] (2017). Highly Available (Mirrored) Queues, Pivotal, [Online]. Available: https://www.rabbitmq.com/ha.html (visited on 12/16/2017).

[82] (2017). Logging, Pivotal, [Online]. Available: https://www.rabbitmq.com/logging.html.

[83] (2017). Detecting Dead TCP Connections with Heartbeats, Pivotal, [Online]. Available: https://www.rabbitmq.com/heartbeats.html (visited on 09/28/2017).

[84] (2017). How To Configure The TCP Keepalive Parameters For Linux ?, Progress, [Online]. Available: https://knowledgebase.progress.com/articles/Article/000044970 (visited on 09/28/2017).