FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Ermöglichung von Mobilität und Nachrichtenübermittlungs-garantien in verteilten MQTT-Netzwerken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Manuel Geier, BSc
Matrikelnummer 01126137

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Dipl.-Ing. Thomas Rausch, BSc

Wien, 6. März 2019

| Manuel Geier | Schahram Dustdar |

Enabling Mobility and Message Delivery Guarantees in Distributed MQTT Networks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Manuel Geier, BSc
Registration Number 01126137

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Univ.Ass. Dipl.-Ing. Thomas Rausch, BSc

Vienna, 6th March, 2019

Manuel Geier	Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Manuel Geier, BSc
Meidlinger Hauptstraße 32/32, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. März 2019

Manuel Geier

Acknowledgements

There are many people I want to thank, but first of all, I want to thank *myself*. This work would not have been possible without my past-self that put a lot of effort, time, thoughts and energy into it. *Thank you very much! I am very proud you and your work!* The journey of self-mastery is a never ending process and during this work I learned a lot, not only about the subject of this work, but also about myself and how to improve in many areas. Finishing this "master" piece (pun intended) left me with many great experiences and learnings that are useful for the rest of my life.

Very special thanks to my advisor *Univ.Prof. Dr. Schahram Dustdar* and his assistance *Univ.Ass. Dipl.-Ing. Thomas Rausch, BSc* who gave me the opportunity to write my diploma thesis with them in the Distributed Systems Group in the Faculty of Informatics at the Technical University of Vienna. It was a great pleasure to work with them and I really valued their input, thoughts and suggestions on my work. I also thank them for giving me the freedom and time to learn and work on my own pace and for providing me with all the resources to create this work.

It is hard to put in words how much I appreciate my dear friends and colleagues that I met at the university. I thank all of you and I especially give a big thanks to *Dipl.-Ing. Thomas Rieder*, *Dipl.-Ing. Markus Zisser*, *Dipl.-Ing. Patrick Säuerl* and *Dipl.-Ing. Lisa Leonhartsberger*. We successfully tackled various challenges through our common journey and we always supported and motivated each other to give our best. It was a lot of fun to be on this journey with you. This final work is especially for you.

Furthermore, I give many big thanks to *my family* and *all my friends* for their great support. This work is also for you, even though you might never clearly understand what I actually was working on. ;) You gave me the strength and vision that lead to this final work.

Finally, I thank all of *my colleagues and leaders at Catalysts* that supported me during this time and that gave me the time and the space that I needed to grow and to finish my career at the Technical University of Vienna.

THANK YOU VERY MUCH! :)

Manuel

Kurzfassung

Publish/Subscribe ist ein weit verbreitetes Kommunikationsmuster für Maschine-zu-Maschine-Kommunikation im Internet der Dinge (Internet of Things, IoT) und das MQTT Protokoll hat sich aufgrund seines leichtgewichtigen Designs zum De-facto-Standard etabliert. Zentralisierte Nachrichtenverteiler sind jedoch nur begrenzt in der Lage, die strengen Servicequalitätsanforderungen moderner IoT-Szenarien zu erfüllen, welche eine dezentrale und zuverlässige Kommunikation am Rand des Netzwerks erfordern. In vielen dieser Szenarien sind Endgeräte nicht mehr nur statisch an einem Ort, sondern bewegen sich in einer Umgebung. Dadurch werden zusätzliche Herausforderungen und Möglichkeiten für eine zuverlässige Kommunikation kreiert. In dieser Arbeit präsentieren wir einen Lösungsansatz für ein skalierbares, verteiltes MQTT-Netzwerk und einen Migrationsprozess, um eine zuverlässige und transaktionale Teilnehmermobilität zu ermöglichen. Wir erweitern MQTT v3 und führen ein transaktionales Migrationsprotokoll ein, das die Nachrichtenübermittlungsgarantien von MQTTs für einzelne Abonnements (höchstens einmal, mindestens einmal, genau einmal) verwendet und gewährleistet, um einen Teilnehmer zwischen (verteilten) Verteilersystemen zu migrieren. Mit Softwaretests, empirischen Experimenten und einer theoretischen Analyse evaluieren wir die Korrektheit, die Reaktionsfähigkeit und die Systembelastung unseres Lösungsansatzes. Die Evaluierung zeigt, dass unser Migrationsansatz es Teilnehmern ermöglicht, von einem Verteiler zu einem anderen zu migrieren, während die Nachrichtenübermittlungsgarantien gewährleistet werden, wobei jedoch zusätzlicher Aufwand verursacht wird. In typischen Szenarien, in denen Verteiler synchronisiert sind, beträgt die Migrationszeit etwa das Achtfache der Verbindungslatenz zwischen den Verteilern. In interregionalen Cloudszenarien, in denen die Verbindungslatenz zwischen den Verteilern etwa 80 bis 100 ms beträgt, dauert die Migration etwa 600 bis 800 ms. Wenn sich die Broker in unmittelbarer Nähe befinden, geht die Anzahl der verworfenen, duplizierten und gespeicherten Nachrichten gegen Null. Mit unserem Migrationprozess nimmt der Netzwerkverkehr mit *höchstens einmal* ab, nimmt mit *mindestens einmal* zu und bleibt bei *genau einmal* Abonnements, mit jedoch einem zusätzlichen Aufwand zur Speicherung von Nachrichten, gleich.

Abstract

Publish/subscribe is a commonly used pattern for machine-to-machine communication in the Internet of Things (IoT) and the MQTT protocol has emerged as the de-facto standard due to its lightweight design. However, centralized messages brokers are limited in their ability to satisfy the stringent quality of service requirements of modern IoT scenarios which require decentralized and reliable communication at the edge of the network. In many of these scenarios, end devices are not static at one place anymore, but rather move around in an environment which creates additional challenges and opportunities for reliable communications. In this work we present an approach for a scalable, distributed MQTT network and a migration process to enable reliable and transactional client mobility. We extend MQTT v3 and introduce a transactional migration protocol which makes use of and ensures MQTTs message delivery guarantees for individual subscriptions (at most once; at least once; exactly once) to migrate a client between (distributed) broker systems. With software testing, empirical experiments and a theoretical analysis we evaluate the correctness, the responsiveness and the system strains of our solution approach. The evaluation shows that our migration approach enables clients to migrate from one broker to another while ensuring message delivery guarantees, but incurs additional overhead. In typical scenarios, where brokers are synchronized, the migration time is roughly eight times the link latency between brokers. In interregional cloud scenarios where the link latency between brokers is roughly 80-100 ms, the migration takes around 600-800 ms. When brokers are in close proximity, the number of discarded, duplicated and stored messages are close to zero. With our migration process, the network traffic decreases with *at most once*, it increases with *at least once* and it stays the same with *exactly once* subscriptions, but comes with an overhead in storing message.

Keywords: publish/subscribe, MQTT, distributed network, internet of things, client mobility, message delivery guarantee, migration process

Contents

CHAPTER 1

Introduction

1.1 Motivation

The Internet-of-Things (IoT) is “Anytime, Anything, Anywhere” [GBMP13]. In many areas from transportation and logistics, to healthcare, in smart home environments and in our personal and social life, IoT is all around us [AIM10]. End devices like smartphones, smartwatches and smart-sensors are getting more powerful and feature-rich every year and therefore getting more and more attention, since their wide variety of applications and possibilities. These devices are mostly connected in a network through wireless connections like Wi-Fi, Bluetooth or RFID (radio frequency identification) to exchange data and therefore are flexible in its usage. Communication between such devices, with or without direct human intervention (Machine-to-Machine, M2M), is part of the IoT paradigm [GBMP13].

To facilitate communication in a (wireless) network between loosely coupled services on such devices, one common strategy is the use of the publish-subscribe message pattern. In this pattern, message producers (publishers) publish messages of interest and message consumers (subscribers) subscribe to interests to receive published messages. Message-oriented middlewares (MoM) like JoramMQ[1], Redis[2] or Mosquitto[3] are used as message brokers, to deliver messages from publishers to its subscribers in a network. One popular message protocol for publish-subscribe systems is Message Queue Telemetry Transport[4] (MQTT), since it is designed to be extremely lightweight and can be used in environments where bandwidth is precious. It has become the de-facto standard for IoT and is highly used, as shown in [Nai17]. MQTT especially provides three Quality-of-Service

[1]http://www.scalagent.com/en/jorammq-33/products/overview
[2]https://redis.io/
[3]https://mosquitto.org/
[4]http://mqtt.org/

configurations (QoS 0-2), for publications (messages to a broker) and for subscriptions (messages from a broker): QoS 0: at most once; QoS 1: at least once; QoS 2: exactly once. MQTTs QoS are the message delivery guarantees for messages, in comparison to more general QoS like message throughput, latency or loss. Application can make use of MQTTs QoS for message delivery to optimize their systems.

The dynamic aspect in IoT scenarios is a challenge, in particular for guaranteed message delivery [RD18]. Many smart devices like tv, radio, fridges are at fixed locations in space. Conversely, devices like mobile smartwatches or mobile sensors, are not location-fixed and can move through space. Hence, there is a dynamic property of the network topology within a (wireless) infrastructure, as described in [SCZ$^+$16] and [JS15]. In such a dynamic network, interrupted connections, varying bandwidths and changing as well as moving participants in the environment have to be handled. This dynamic property of devices needs to be addressed by the MoM in order to guarantee Quality-of-Service agreements, e.g., for message delivery, and to provide performing services, e.g., minimize message round trip time between clients in close proximity, as in PEER-WD [TP13], in osmotic computing [VRF$^+$16], in edge computing [SDC16], [SCZ$^+$16], [Sat15], or in edge-centric computing [LME$^+$15]. Kim and Noble state in [KN01], that the "adaptation to changing network conditions is critical to the success of mobile systems".

Many MOMs also perform load-balancing, (overlay) network reconfigurations and other optimizations to improve the performance and quality of the service [Cug02]. One particular task is to migrate (mobile) clients in a distributed network from one node to another node. Most MOMs are aware of message loss or duplication and try to prevent or ignore this [HGM04]. Rausch et al. show in [RND18], that message loss is real when mobility is involved. Nevertheless, many MOMs do not consider MQTTs QoS for message delivery for further optimizations. However, such considerations provide opportunities to improve the overall QoS of a system, when a client migration from one broker to another has to be performed, since not all messages have to be threaded the same, but within the boundaries of MQTTs defined QoS guarantees. This opportunity is what we are addressing in this work.

1.2 Problem Statement

As MQTT has by default a centralized design, considering mobility and a distributed network design creates different challenges and opportunities for MQTT. This work addresses the following challenges in particular:

Given a distributed MQTT broker network, the challenge is to migrate a mobile subscriber from one broker to another broker while ensuring and being aware of the MQTT QoS message delivery guarantees for its subscriptions. A subscriber might have multiple subscriptions with different QoS configurations that have to be migrated. Figure 1.1 gives a brief graphical representation of the problem statement.

In this work we particularly target subscriber mobility, since publisher mobility provides different challenges as shown in [MPJ05], [MPDJ05] and also in our work and therefore left for future work.

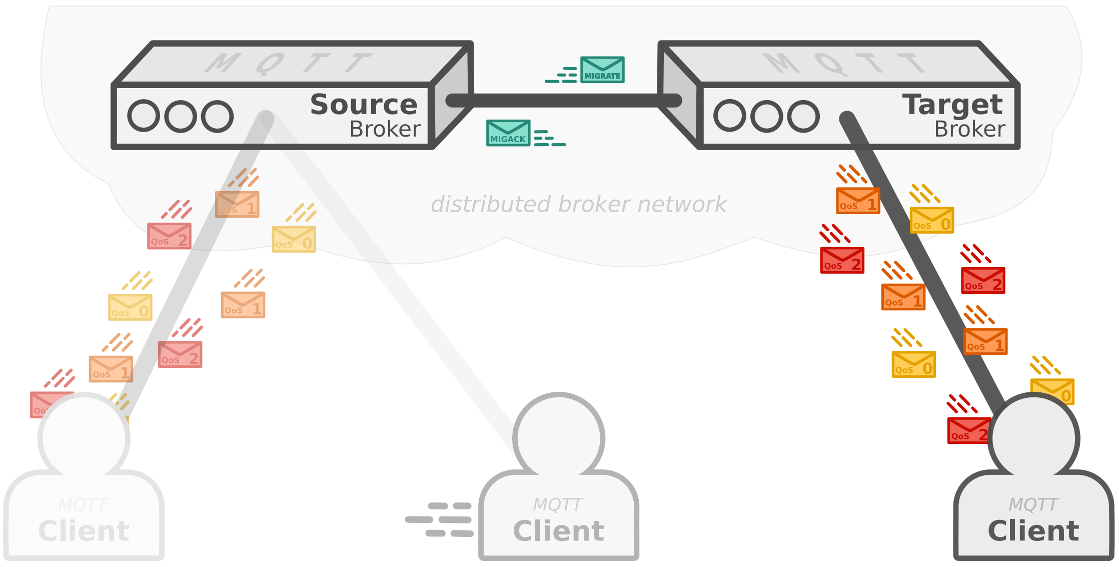


Figure 1.1: Migration of a mobile MQTT Client in a distributed MQTT broker network from the Source Broker to the Target Broker.

1.3 Aim of the Work & Expected Results

The aim of this work is to enable mobility in a distributed MQTT broker network and to design and develop a migration process and a migration protocol to perform a client migration that considers MQTTs message delivery guarantees. As made apparent by the problem statement, current approaches do consider other QoS, like message throughput, latency or loss, but do not consider the more specific QoS of the MQTT protocol for message delivery, that applies for publications and subscriptions. Compared to current approaches, we see potential to improve the other QoS by taking the message delivery guarantees of MQTT into account.

We design, model and implement a migration process and a migration protocol, that can perform a message delivery guarantee aware client migration from a broker to another broker. To that end, suitable algorithms, methods and standards are researched, chosen and combined to create our migration solution. Design concepts from the formalized transactional protocol for mobile publish/subscribe clients by [SMGJ09] are used to gain transactional properties, even though a formalization of our migration protocol is not be part of our work and left for future work.

Existing publish-subscribe solutions, e.g., JoramMQ[5], Mosquitto[6], Moquette[7], are analyzed and one of the systems is chosen to integrate the migration process implementation[8]. With the integration into chosen publish-subscribe system, we evaluate our migration solution.

We use software testing methods to verify the correctness of our solution approach, our implementation and the integration into existing systems. With empirical experiments with the actual systems running in a dockerized environment, we further verify the correctness and measure the responsiveness of our solution. Finally, in a theoretical analysis with a simplified mathematical model of our migration process we further evaluate the responsiveness and system strains.

Our results verify the correctness of our solution approach and our implementations. The responsiveness and the system strains highly depend on the network latencies, the synchronisation state of the brokers that perform the migration, the number of subscriptions and the chosen QoS of the Client's subscriptions. With QoS 0 subscriptions we decrease the network traffic, with QoS 1 subscriptions we increase network traffic and with QoS 2 subscriptions it stays the same. Even through our solution approach aims MQTT and its message delivery guarantees, the concepts we present might also be applied to other protocols like AMQP that provide message delivery guarantees as well.

1.4 Methodological Approach

To achieve the expected results the following methodological approach is used:

- Systematic review of existing publish-subscribe message broker systems and message protocols (e.g., MQTT, AMQP, CoAP), client mobility approaches and client migration strategies.
- Enhance MQTT to create a distributed broker network and design and model a migration process and a migration protocol for a client migration that is aware of message delivery guarantees.
- Implementation and integration of the migration solution into a MQTT broker and MQTT client system, by using iterative software engineering methods and software testing.
- Evaluation of correctness, responsiveness and system strains with software testing, empirical experiments and a theoretical analysis of the solution approach.

[5]http://www.scalagent.com/en/jorammq-33/products/overview
[6]https://mosquitto.org/
[7]http://moquette.io/
[8]We chose Moquette.

1.5 Terminology

Throughout this work we are consistently referring to specific subjects using the following terms:

- **QoS**: Describes the message delivery guarantee (QoS 0-2) of the MQTT protocol.
- **Client**: Describes the MQTT client, that migrates from a MQTT broker to another.
- **Source Broker**: Describes the MQTT broker system that a Client migrates from.
- **Target Broker**: Describes the MQTT broker system that a Client migrates to.
- **Coordinator**: Describes the entity that triggers the migration process.
- **Packet**: Describes a packet of our Migration Protocol or the MQTT protocol.
- **Message**: Describes a concrete message that should be delivered.

1.6 Outline

The remainder of the work is structured as follows. In Chapter 2, we provide background knowledge that lays the foundation for our work. Chapter 4 presents and describes our solution approach to enable mobility and our QoS-aware migration process. In Chapter 5, we detail about the implementation of our solution and describe the different components it involves. Our evaluation and the deriving results are presented in Chapter 6. Chapter 7 concludes this work and gives suggestions for future work. (Further details about our work can be found in the Appendix in Chapter 8.)

CHAPTER 2

Background

In this chapter, we give background information to have a common knowledge base for the remainder of this work. We start by introducing the publish/subscribe paradigm with its actors and describe different subscription languages and subscription types (Section 2.1). We discuss architectural styles for broker networks (Section 2.2) and present some publish/subscribe protocols (Section 2.3) and some broker implementations (Section 2.4). Furthermore, we discuss the message delivery guarantees for MQTT in detail (Section 2.6), since we build the foundation of our solution approach on characteristics of these guarantees in Chapter 4. Finally, we present two different migration types (Section 2.7) and communication patterns (Section 2.8) to perform migration processes.

2.1 Publish/Subscribe Paradigm

The publish/subscribe paradigm is a well-known and -used communication pattern in messaging systems. It is an example of a data-centric communication model and widely spread in enterprise networks, mainly because of its scalability and support for dynamic application topologies, as stated by Hunkeler et al. [HTSC07]. The loose coupling between publishers and subscribers is one advantage of this paradigm that provides a lot of flexibility to systems. Especially in the field of IoT and in sensory networks, this paradigm provides many advantages as described in [Bet99] and [EFGK03]. Also in mobile environments, the publish/subscribe paradigm is applied, as shown by Huang and Garcia-molina in [HGM04].

We briefly introduce the important concepts of the publish/subscribe paradigm in the following sections starting with the main three actors in a publish/subscribe system:

- **Publisher**: The publisher takes collected or generated data and publishes it, e.g. to topics, to a broker in order to provide information to interested parties.

- **Subscriber**: The subscriber registers its interests, e.g. via topics subscriptions, on a broker in order to receive information it is interested in.
- **Broker**: The broker is the middleware between publishers and subscribers and forwards messages that were published to the broker to subscribers that are interested in this information.

An individual client can be a publisher and a subscriber at the same time.

2.1.1 Subscription Languages

Publish/Subscribe systems support different subscription languages for subscribers in order to show their interest for specific messages. Common subscription languages are classified as follows:

- **topic/subject**: Topic-based or subject-based subscription languages subscribe to an exact string or keyword.
  Example: `fish`, `bird`, `mammal`

  More expressive language allow strings to be paths to generate hierarchies of topics.
  Example: `/animal/water`, `/animal/water/sweet`, `/animal/water/salty`

  These topic paths usually allow wildcards to subscribe to multiple topics or whole subhierarchies at once.
  Example: `/animal/water/*`
- **content**: Content-based subscription languages typically use a query language on attributes of messages to match interests in published messages.
  Example: `[environment == WATER && weightKg > 100]`
- **type**: Type-based subscription languages allow to subscribe to specific messages types or even subtypes [Eug07].
  Example: `type Animal`, `type WaterAnimal`, `type SweetWaterAnimal`

2.1.2 Subscription Types

Some publish/subscribe systems support different subscription types to tackle certain application needs:

- **Local subscription**: A local subscription refers to a subscription that was registered by a specific client. This client will receive all messages of its subscriptions. If multiple clients register for the same interest, every client will receive all matched messages.

- **Shared subscription**: A shared subscription is a subscription that is shared upon its subscribers. A messages just gets published to one of its subscribers. This type can be useful for messages that act like tasks that have to be processed just once by an arbitrary client that can process the message.

If a client is offline for some time and reconnects again, the broker will handle the client subscriptions while the client is offline based on its durability:

- **Durable subscriptions**: A durable subscription makes sure, that when the client reconnects to the broker system, that it will receive all messages that have been published while the client was offline. Therefore, the client does not miss any messages even if the connection gets interrupted.
- **Non-durable subscriptions**: A non-durable subscriptions does only deliver messages to client while they have an active connection, i.e. are online. If the client is offline, messages that are published in the meanwhile, are lost for the client.

This work focuses on topic-based subscriptions with exact strings and local subscriptions. We assume that the Client is online for the whole migration process. Hierarchy-support or further subscription language expressiveness are left for future work.

2.2 Broker Networks

The first publish/subscribe architectures consisted of a centralized broker that handled all requests [HGM04]. Nowadays, broker systems can consist of multiple brokers that work together to fulfill their applications need and are optimized, e.g., to increase the packet through-put or to decrease the total network traffic.

To distribute a published message to multiple subscribers, messages can be unicasted or multicasted (or have a mixed setup). A lot of research was done on multicasting in order to disseminate publications to interested subscribers such that the total network traffic is minimized as stated in [MJ10].

- **Unicast**: Sending a message from a sender to a single receiver.
- **Multicast**: Sending a message from a sender to a group of multiple receivers.

Since client mobility within broker network with multiple brokers is a challenge, i.e. clients move between brokers, and the MQTT protocol by itself does not support networks or client mobility, this features have to be extended by the specific broker implementations.

There are certain architecture styles that are common to connect multiple brokers in order to scale the whole system: bridging, clustering and distributed networks. For each architectural style, we are mentioning some examples of broker system, that support (at least) this style. These broker systems are also introduced in the upcoming Section 2.4.

- **Bridging**: Bridging refers to a method where two or more brokers are connected to exchange certain messages with each other. Brokers are linked together and define a pattern for messages that get forwarded to another broker. Within this process, message filtering, remapping of topics or other transformations can be done. A common usage is to connect edge brokers to a central or remote network [Mos18]. Bridging can be achieved, by putting the broker into the role of a publisher or a subscriber of another broker system.

  Examples: Mosquitto, VinveMQ

- **Clustering**: Clustering refers to a method to group multiple system together to form a uniform unit in order to improve the systems performance. Systems might be connected in a hierarchy form and serve in a master-slave approach. Individual systems might access shared resources like databases and use transactional approaches to create a consistent system.

  Examples: HiveMQ, EMQ

- **Distributed Networks**: Distributed networks are individual nodes that are connected via a network setup and all nodes together form the whole system. A single node itself can be a single broker system or even a cluster of broker systems. In terms of edge-computing, a single node might just serve some local space. Caused by the nature of the distributed style, the challenges and coordination of such broker networks are different compared to clustered setups, where individual broker systems are close to each other.

  Examples: WSO2 Message Broker, JoramMQ

It is important to note that certain broker systems might support multiple architectural styles depending on the configuration of the system.

Each of the presented architectural styles describes multiple broker systems connected together to form the overall broker network. Within all of these styles, clients can be connected to individual broker systems and might need to be migrated from a broker to another for various reasons. Therefore, client migrations are necessary which is a motivation of our work.

Our solution approach aims client migrations between broker systems in distributed networks. Nevertheless, it might also be applied in a clustered setup, since broker systems do not necessarily have to be distributed with our solution.

2.3 Protocols

To provide data transport for publish/subscribe broker systems different communication protocols with different characteristics exist. Most widely adopted protocols in the IoT field are MQTT and CoAP, as stated in [LCC$^+$15]. Other protocols that support the publish-subscribe paradigm include AMQP XMPP, STOMP and ZMTP.

Some of these protocols are briefly described as follows:

- **AMQP**: The Advanced Message Queuing Protocol[1] (AMQP) delivers messages through *exchanges* and *message queues*. Exchanges retrieve published messages from clients and route them into message queues, where they are provided for clients [VT06]. It is based on reliable transport layer protocols like TCP and provides a point-to-point communication pattern as well as publish/subscribe on messages queues for clients. Furthermore, it provides message delivery guarantees for message transfer, i.e. at-most-once, at-least-once and exactly-once, that are similar to MQTTs QoS as described in Section 2.6.

- **CoAP**: The Constrained Application Protocol[2] (CoAP) is a lightweight data transfer protocol that is designed to operate in constrained network environments or embedded systems, where code space is limited, processing is crucial or power consumption limited by battery life [SHB14]. It was presented by Bromann et al. in [BCS12] and has similarities to a lightweight version of HTTP. It is based on the REST (Representational State Transfer) architectural style and URIs are used to identify resources. Compared to HTTP, it uses UDP instead of TCP to reduce its complexity and defines a very simple message layer for retransmitting lost packets. Through proxies, CoAP can interact with HTTP, based on the REST architectural style. It provides some publish/subscribe functionality on top of the REST model by specifying an *Observe* option within a GET request. If the server accepts this option, the client will get the requested resource and it will be asynchronously notified with further updates, when the resource changes.

  A comparison and performance evaluation of MQTT and CoAP was done by Thangavel et al. in [TMV$^+$14].

- **MQTT**: The Message Queuing Telemetry Transport protocol[3] (MQTT) is a lightweight publish-subscribe messaging protocol, developed by OASIS [MQT15]. It gained high poularity in IoT, since it is very light and easy to use. The current version is 5.0 [MQT19]. Projects like the Facebook Messenger [Zha11] as well as many open source projects are based on it, as stated in [LKHJ13]. The data transfer is based on TCP and it uses a topic-based subscription language with

[1]`http://www.amqp.org`
[2]`http://coap.technology/`
[3]`http://mqtt.org`

wildcards filters. Furthermore, it defines three QoS for subscriptions (message delivery guarantees). More details on message delivery guarantees are discussed in Section 2.6. The MQTT protocol by itself does not support distributed networks or client mobility features.

  – **MQTT-SN**: MQTT-SN (formerly known as MQTT-S) is an extension of MQTT to provide functionalities that are more specific for Wireless Sensor Networks (WSNs) as described in [HTSC07]. It consists of clients that work within the wireless network and (transparent or aggregating) gateways that translates between MQTT and MQTT-SN.

- **STOMP**: The Simple (or Streaming) Text Oriented Message Protocol[4] (STOMP) is a simple publish-subscribe protocol that defines a text based wire-format for messages. The encoding for message bodies in STOMP is by default UTF-8, but it also supports the specification of alternative encodings. It does not support message delivery guarantees by default, but as stated in the specification [STO], STOMP servers may support additional server specific headers to customize the delivery semantics of a subscription.

- **XMPP**: The Extensible Messaging and Presence Protocol[5] (XMPP) is a set of open technologies upon XML data. The protocol messages between entities are defined in XML format and its applications include instant messaging, voice and audio calls and collaboration. Furthermore, it supports content syndication and generalized routing. A PubSub[6] extension of the protocol provides support for generic publish-subscribe functionality, adhered from the classic Observer design pattern.

- **ZMTP**: The ZeroMQ Message Transport Protocol[7] (ZMTP) is a transport layer protocol for exchanging messages between two peers over a connected transport layer such as TCP. It supports different communication patterns like Request-Reply, Publish-Subscribe, Pipeline (Push–pull) and the Exclusive Pair Pattern. It targets distributed and concurrent applications [Zer].

Since we focus on the IoT field and want to discuss message delivery guarantees, we use MQTT as our primary communication protocol and as a base for our remaining work. Many concepts are not exclusive to MQTT, but rather can be generalized and used in other protocols as well. We use MQTT version 3.1.1, since this was the standard at the time, we started working and refer it as MQTT v3. Nevertheless, we are discuss relevant new features of the current version 5.0 in this work for future work as well.

[4]http://stomp.github.io
[5]https://xmpp.org/
[6]https://xmpp.org/extensions/xep-0060.html
[7]http://zeromq.org

2.4 Broker Systems

The MQTT protocol is implemented by various broker implementations. Since the MQTT protocol itself does not support client mobility between brokers, this feature has to be provided by the specific broker implementations itself.

Some broker systems that support the MQTT protocol are briefly describe as follows:

- **EMQ**[8] (Erlang MQTT Broker) is a distributed, massively scalable, highly extensible MQTT message broker written in Erlang/OTP. It implements MQTT v3.1 and v3.1.1 and and supports MQTT-SN, CoAP, WebSocket, STOMP and SockJS. It also supports Local Subscription (every subscriber gets the a message) and shared subscription (just one of multiple subscribers gets the message). It can be deployed in a clustered setting or as a distributed node setup. In order to support client mobility, nodes can transfer the client state from one node to another, assuring all QoS 1 and QoS 2 messages are always sent to the client [EMQ17].
- **HiveMQ**[9] is a cloud-based messaging system with MQTT for the fast, efficient and reliable movement of data to and from connected IoT devices. It supports v3.x and v5 and provides a distributed and clusted architecture, as well as the concept of shared subscriptions along multiple clients with load balancing approaches[10].
- **JoramMQ**[11] is a open source messaging provider that provides an MQTT server supporting v3.1 and v3.1.1 as a plugin[12]. It has an agent-based distributed architecture and supports clustered and distributed architectures as well as message bridging to other brokers in centralized and decentralized architectures [Jor14].
- **Moquette**[13] is a lightweight MQTT broker in the JVM that it designed for IoT. It uses Netty as the NIO framework which handles TCP connections and message exchange and supports MQTTs QoS 0-2. It is written in Java and open source[14]. It is supposed as a centred broker system and does not support a broker network setup by default.
- **Mosca**[15] is a very minimal MQTT broker system as a Node.js module that supports v3.1 and 3.1.1. It is limited to QoS 0 and QoS 1 and does not support QoS 2. It further does not support broker network setup, but has an event system that gets triggered, when clients disconnect or reconnect in order to forward missed messages [Col14].

[8]http://emqtt.io/
[9]https://www.hivemq.com
[10]https://www.hivemq.com/features/
[11]http://www.scalagent.com/en/jorammq-33/products/overview
[12]http://www.scalagent.com/en/jorammq-33/technology-36/mqtt-protocol
[13]http://moquette.io/
[14]https://github.com/andsel/moquette
[15]http://www.mosca.io/

- **Mosquitto**[16] is a lightweight MQTT broker system that is suitable for usage on all devices from low power single board computers to full servers. It is open source and supports MQTT protocol v3.1 and v3.1.1 and broker bridging to connect multiple brokers. Client mobility support is not available.
- **VerneMQ**[17] is a Erlang based broker system that was build up to tolerate network failures and provides fine-grained control over the availability and consistency behaviour. It provides a dynamic cluster architecture [Ver18] and supports migrating clients on leaving nodes, by reconnecting its active clients and transferring its queues to another node.
- **WSO2 Message Broker**[18] is a lightweight, open source, distributed message-brokering server that supports AMQP and MQTT protocols. It also provides clustering of multiple broker systems with cluster coordination through a RDBMS (Rational Database Management System) or a hazelcast engine and to persist messages along multiple brokers. To do this, it is using distributed transactions that are based on Java Transaction API and organized by a transaction manager.

We could not determine that any of these brokers provides special features that enables client mobility under the concern to make use of the message delivery guarantees that the subscriber stated for its subscriptions. Therefore, our work provides new scientific research in order to enhance the client mobility process.

2.5 Quality-of-Service

The Quality-of-Service describes the overall performance of a system. This can be low latency, high throughput, memory efficiency, persistence, high concurrency, scalability, reliability, durability, etc. The actual costs to run the system can also be considered for QoS, as presented with MultiPub by Gascon-Samson et al. in [GSKK17]. Usually, a system can not just have benefits in all aspects, but has its trade-offs. If one quality property of a system gets increased, (i.e. low latency), another might suffer (i.e. high concurrency). In publish/subscribe message systems qualities might be message latency or message throughput and is an important aspects for scalability ([SCA$^+$05], [AR02], [KKY$^+$10], [YKK$^+$09]). Behnel, Ludger and Gero provide in [SGGK14] a broad overview of relevant Quality-of-Service metrics on publish/subscribe systems. Quality-of-Service in wide scale publish–subscribe systems are discussed by Bellavista, Corradi and Reale in [BMCR14].

We want to point out, that "QoS" in our work specifically refers to the message delivery guarantees of MQTT subscriptions (Section 2.6) and does not refer to other Quality-of-Service criteria.

[16]https://mosquitto.org/
[17]https://vernemq.com/
[18]https://wso2.com/products/message-broker/

2.6 Message Delivery Guarantees

Protocols like MQTT or AMQP define message delivery guarantees for messages that get sent between systems. These message delivery guarantees define the communication pattern between two parties, i.e. acknowledging messages upon arrival.

MQTT defines three quality levels of delivery guarantees for its protocol (QoS 0, QoS 1 and QoS 2) and uses different processes and packets in its delivery process to guarantee them [MQT15]:

- **QoS 0 - At most once**:
  Messages arrive at most once at the target.
  MQTT packets: PUBLISH
- **QoS 1 - At least once**:
  Messages arrive at least once at the target.
  MQTT packets: PUBLISH, PUBACK
- **QoS 2 - Exactly once**:
  Messages arrive exactly once at the target.
  MQTT packets: PUBLISH, PUBREC, PUBREL, PUBCOMP

Based on the QoS level, message loss or message duplicates are valid (see Table 2.1).

Table 2.1: Message loss and duplication validity for each QoS

	QoS 0	**QoS 1**	**QoS 2**
Message Loss	Valid	Invalid	Invalid
Message Duplication	Invalid	Valid	Invalid

The applications themselves have to choose the right QoS for their services for publications and subscriptions. A publisher *pub* or subscriber *sub* can individually decide which delivery guarantee they want to have for their deliveries. There are two deliveries where MQTT QoS are applied:

- **publish**: The publisher can publish a message and the QoS contract to the broker qos_{pub} describes how the message delivery will be guaranteed.
- **forward**: The QoS contract qos_{sub} describes the requested QoS by the subscriber from a broker. The broker will try to accomplish this QoS and calculate the actual QoS qos_{for} to forward messages for its delivery (see Section 2.6.1).

Figure 2.1 shows the delivery of a published message from the publisher to a subscriber through the broker network and the different delivery guarantees for each connection.

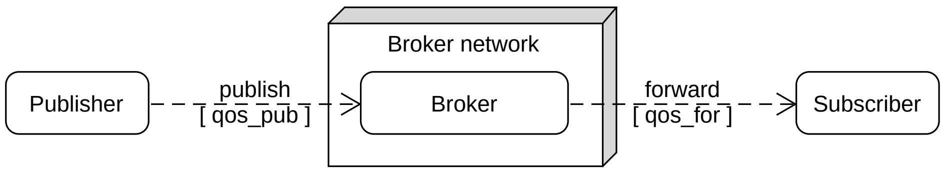


Figure 2.1: Message delivery guarantees through the broker network.

2.6.1 Actual Subscriber QoS

As mentioned, the broker will try to accomplish the requested QoS by the subscriber qos_{sub}, but it may is not possible to do so.

qos_{pub} and qos_{sub} correlate in the way, that the resulting QoS qos_{for} of the subscriber can never higher than the QoS of the publisher qos_{pub}.

$$qos_{for} \leq qos_{pub}$$

Hence, the QoS of the subscriber is just the favored (requested) QoS, but not necessarily the actual QoS for the subscriber. If the QoS of the subscriber is higher, it gets reduced to the QoS of the published message.

$$qos_{for} := min(qos_{pub}, qos_{sub})$$

It is the broker's task to forward the published message from the publisher with the correct(ed) QoS to the subscriber. Listing 2.1 shows the actual source code of Moquette broker. Table 2.2 shows all combinations of QoS for one delivery and its actual QoS for the subscriber.

```
// Source: io.moquette.spi.impl.ProtocolProcessor
static MqttQoS lowerQosToTheSubscriptionDesired(Subscription sub, MqttQoS qos
    ) {
    if (qos.value() > sub.getRequestedQos().value()) {
        qos = sub.getRequestedQos();
    }
    return qos;
}
```

Listing 2.1: Moquette broker - Subscriber QoS calculation

Table 2.2: Publish-Subscribe QoS combinations

qos_{pub}	qos_{sub}	qos_{for}
0	0	0
0	1	0
0	2	0
1	0	0
1	1	1
1	2	1
2	0	0
2	1	1
2	2	2

This has to be taken into account during the for the migration process, even if we just focus on the subscriber migration. To simplify the migration process for our work, all messages from the publisher are sent with QoS 2.

$$qos_{pub} := 2$$

Hence, the forwarding QoS for the subscriber is the same as the QoS of the requested QoS of the subscription.

$$qos_{for} = qos_{sub}$$

Considering QoS 0 and QoS 1 from the publisher is left for future work.

2.7 Client Migration Types

To migrate clients from one broker to another broker in a broker network, we can distinguish between *planned* and *unplanned* migrations:

- **planned**: We define planned migrations, such that the migration of a client is performed in order to improve the systems performance while the client is active and online during the whole migration process.
- **unplanned**: We define unplanned migrations, such that a client disconnected from the broker in the network and reconnects to a different broker, such that a migration process has to be performed in order to continue a valid message exchange.

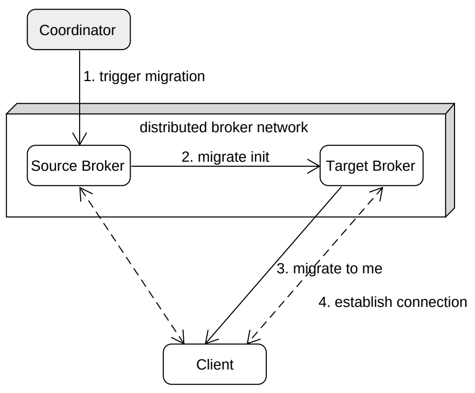
In our work, we focus on planned migrations and assume that the client is online during the migration process.

2.8 Migration Communication Patterns

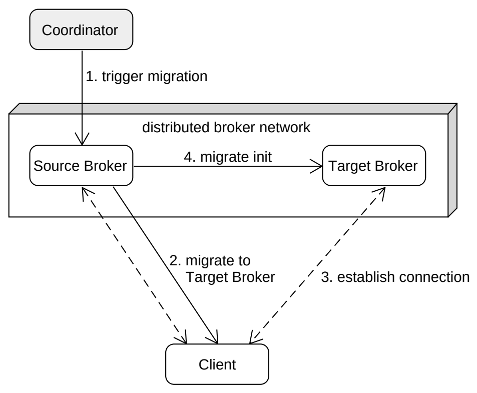
To initialize the migration process, different options are possible on how the parties interconnect. The Coordinator can trigger Source Broker, the Target Broker or the Client at first and these parties themselves can connect to one of the other parties to create a fully interconnected state for our solution approach. Different communication patterns to establish the connection between all systems are shown in Figure 2.2.

For our work, we wanted to have a very minimal impact on the Client and have the main coordination effort for the migration process on the brokers side. Therefore, we chose to implement the option in which the Coordinator tells the Source Broker to start the migration and the Target Broker connects to the Clients[19] (Figure 2.2a). In this case, the broker network itself is in control of all migration processes and therefore can coordinate the migrations.

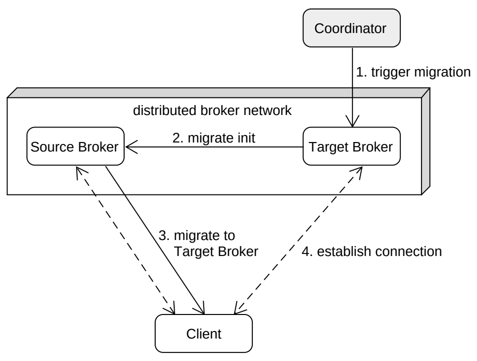
[19]By default, MQTT only allows the initiation of a connection from a client to the broker with a CONNECT packet. To initiate a connection from the broker to the client, we extended the communication process and give the broker the possibility to send a MIGTO packet to a client such that the client establishes a connection with the broker.



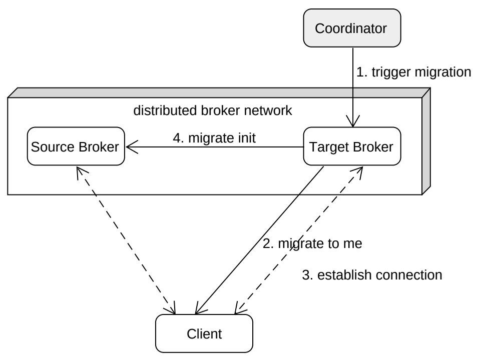
(a) Source Broker → Target Broker → Client



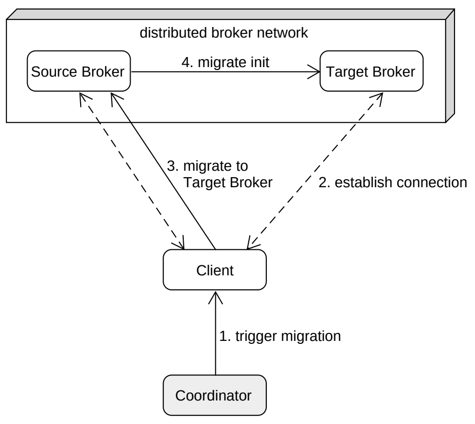
(b) Source Broker → Client → Target Broker



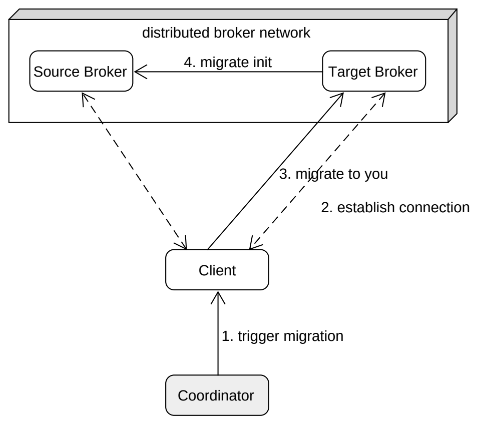
(c) Target Broker → Source Broker → Client



(d) Target Broker → Client → Source Broker



(e) Client → Source Broker → Target Broker



(f) Client → Target Broker → Source Broker

Figure 2.2: Migration Communication Patterns

CHAPTER 3

Related Work

Muthusamy et al. describe in [MJ10] some strategies for subscriber mobility that are briefly summarized in the following. Even tough these strategies assume the Client will be disconnected for some time, as in unplanned migrations, they still can be applied to planned migrations as well. Furthermore, these strategies assume, that when a client disconnects from the Source Broker, that the Source Broker starts to store messages that the client should have received locally. In the simple case where the client reconnects to the Source Broker again, these messages are simply replayed. In cases where the client reconnects to some other broker (referred to as Target Broker) more advanced steps are necessary.

- **Standard Algorithm**: This algorithm was proposed by Cugola et a. in Cugola et al. [CDF01]. On reconnect, the Client informs the Target Broker that it was previously connected to a specific Source Broker. The Target Broker communicates with Source Broker to get the associated subscriptions of the Client. The Target Broker subscribes the Client to these subscriptions and tells the Source Broker to unsubscribe from the Client. New messages on the Target Broker are stored in a local queue in the meanwhile. The Source Broker forwards its stored messages to the Target Broker and together with the locally stored messages, the Target Broker forwards the messages to the Client while removing duplicates.

- **Prefetching Algorithm**: With prefetching, the Source Broker tries to predict the Target Broker while the client is disconnected and transfers the Clients' subscriptions and stored messages to a potential Target Broker. On reconnection of the Client to a successful predicted Target Broker, the Standard algorithm is used. Since subscriptions and messages already have been exchanged, only updates to it have to be transferred from the Source Broker to the Target Broker.

- **Logging Algorithm**: Every broker keeps a local store of recently processed messages. On reconnection of a Client, the Source Broker tells the Target Broker its stored message ids, the Target Broker matches these sets and just requests missing messages from the Source Broker. This algorithm requires that messages system-wide unique message ids.
- **Home-Broker Algorithm**: Every client has a Home Broker assigned. Upon reconnection the Client physically connects to some Target Broker, but logically to its Home Broker, who know the subscriptions of the Client. The Home Broker receives messages through the regular multicast mechanism and forwards messages to the Client using the unicast mechanism.
- **Subscriptions-On-Device**: The subscriptions are stored on the Client and it can transfer these information upon reconnected to the Target Broker. Therefore, it does not need to be requested from the Source Broker. Besides this, the process from the Standard Algorithm is performed.

Our solution approach is similar to the Standard Algorithm with the following distinctions. Instead of unplanned migrations, we aim for planned migrations. We put our focus on the message delivery process for the Client and try to minimize the involvement and side-effects, like message delay through the migration process, of the Client. The associated subscriptions are sent from the Source Broker to the Target Broker and the Client is only involved to establish a connection with the Target Broker. Instead of just one broker sending messages to the Client, our solution approach involves both brokers sending messages at the same time to minimize the message delay for the Client while still ensuring the message delivery guarantees. Therefore, no message exchange between brokers is necessary. In contrast to the Standard Algorithm, not all messages are stored on the Source Broker. Just messages that are part of a QoS 2 subscription are stored and other messages are either sent (QoS 1) or get discarded (QoS 0) immediately. Methods like Prefetching and Logging as shown for the Standard Algorithm might also be applied to our solution approach to improve the migration process. However, this is not part of this work and left for future work.

Caporuscio, Carzaniga and Wolf describe in [CCW03] the basic design of mobility services and present their evaluation for multiple publish-subscribe systems that support mobility features.

An important application, where client migrations are relevant, is *load balancing*. This describes the process to balance the amount of clients between brokers within a broker network in order increase the systems performance, like in Dynamoth [GsGKK15]. Most of the time, we are interested to have a load balancing process in place, that allow clients to be migrated during runtime, in comparison to systems like Meghdoot in [TSZ11], where the load balancing process is only invoked when a new client enters the system.

King, Cheung and Jacobsen present in [KCJ06] such a load balancing process for publish/subscribe systems with content-based subscriptions. Their approach is build on

Padres Efficient Event Routing (PEER) architecture and one part of their work includes a load balancing framework that lets subscribers migrate from a offloading broker (Source Broker) to the onloading broker (Target Broker). Therefore, they introduce a mediator to coordinate the migration. They also strictly focus on subscriber migration. For their migration process they use a Mediation Protocol, that has a similar function to our Migration Protocol. Multiple load balancing sessions (migration processes) can occur at the same time, but a broker can at most be involved in one. This is different to our approach, where multiple migrations can happen at the same time on a broker. The only restriction our solution has, is that a client can only be involved in one migration process at a time. Their approach lets a batch of subscribers be migrated at once, as compared to our process, where each client is migrated individually. Their goal is to migrate a subscriber to a new broker "in the most efficient and timely manner with minimal delivery loss" and focus on a "end-user transparency and best-effort delivery". Therefore, it tries to minimize the message loss, but does not guarantee that there is not any. In fact, it only checks for message duplications and ignores possible message loss. It is further not aware of any QoS for subscriptions and treats all messages and subscriptions the same. The migration initialization and control design (see Section 2.8) also differs from ours, since the Source Broker contacts the Client that contacts the Target Broker and the control and message check is on "a thin software layer on the client side that hides the intricate details of load balancing from the end-user application", as where our work proposes it on the brokers side. Similarly to our work, the client also has to handle multiple connections to brokers. The migration process completes in their work, when the Client disconnects from the Source Broker, as it completes in our work, when the Source Broker disconnects from the Client.

Mobility of clients influences also the QoS criteria, like latency or through-put, of the whole system and therefore registered clients also have to move from one broker to another in order to increase the systems performance again.

The work of Rausch in [RND18] describes a whole system with a QoS (i.e. latency) aware mechanism to migrate clients at the broker-network-edge to increase systems performance and its QoS[1]. As in our work, it uses MQTT as a publish-subscribe communication protocol and assumes client mobility withing a broker network. It supplies clients with gateways to perform and hide the client migration. Similar to our solution, a client can hold multiple connections to brokers. For each client-broker connection, it holds two buffers for incoming and outgoing messages. If a client is requested to migrate to another broker, it creates a second connection to the new broker and disconnects from the old one. In contrast to our work, the migration process is on the client side within the gateway. Since messages are not checked, even though, multiple connections are supported, duplicates or losses of messages are possible during the migration process. As with our work, it would need a synchronization process to check for losses and duplicates on the client side in order to fulfill the QoS requirements for MQTT subscriptions.

[1]QoS of the system, e.g. latency or through-put, is different from the QoS of a MQTT subscription (QoS 0-2)

Similarly, QoS measurements and optimizations, as shown in the previous example, are the focus of research in [DDM$^+$09] and [CS05], by dynamically reconfiguring the overlay network for the brokers to improve the QoS for participants. Hermes in [PB02] also focuses on QoS-aware systems, by creating an event-based middleware that addresses scalability, interoperability, reliability, expressiveness and usability of distributed publish/subscribe systems.

The work of Songlin et al. in [SMGJ09] discusses transactional mobility in distributed content-based publish/subscribe systems and describes ideas that are similar to our work. Beside formalizing transactional properties (e.g. atomicity property: "After the transaction completes, a moving client must be either at its source or target broker, but not both."), they also developed a client movement protocol for publish/subscribe client mobility, formalized and proofed correctness of it. This formalization provides a more stable system solution and can also be applied to our migration process which is left for future work. The client movement protocol is based on the three-phase commit (3PC) distributed transaction protocol in [SS83]. Since it stated that "clients should not miss any notification while moving", the transactional and formal properties can not directly be transferred, since we allow, depending on the QoS of the subscription, losses and duplicates of messages. In this work, the initialization of the movement protocol starts with a negotiation-packet to request the migration. Upon this, the Target Broker can approve or reject the movement. If it approves, the Source Broker stops the Client and sends queued publications in a state-packet to the Target Broker. The Target Broker then dispatches them to the Client and merges it with its own state. Finally the Target Broker sends an ack-packet to the Source Broker, such that it can clean its state. Compared to our solution, the brokers perform an additional negotiation exchange, as where we have an optimistic approach and assume that the migration will be accepted by default. In our solution, when the Target Broker send the migration with a MIGACK packet and the status code of error, the movement is rejected. Additionally, we also do not send the publications from one broker to another, but let each broker send messages to a defined synchronization point in the synchronization process. Other work that includes transactional approaches are by Vargas et al. in [VPGB] or by Michlmayr and Fenham in [MF05].

Burcea et al. summarized and analyzed in [BJD$^+$04] different methods (Standard Algorithm (as in [CDF01]), Prefetching, Logging and Home-Broker) for message synchronization between brokers for mobile clients that disconnect from and reconnect to brokers in a broker network. In other words, went offline for a period of time and probably reconnected to another broker. Even though we assume no disconnection of a client throughout our migration process, the synchronization methods are still useful and have been considered within our work. Even through, none of these methods directly makes use of the QoS characteristics as we do in our work, these methods can still be applied to extend our solution approach.

Cugola et al. present in [CDF01] the JEDI event- based infrastructure. It supports some offline-features to disconnect a client with `moveOut` and to reconnect a client again with `moveIn` at a later point in time. In the meanwhile, messages get stored, such that the Client will get all messages on reconnection again. This features allow mobility for clients by using `moveOut` on the Source Broker, passing the *active objects* that represent a Client from Source Broker to the Target Broker and using `moveIn` on the Target Broker to reconnect the Client. Nevertheless, this was not fully examined in this work and QoS for subscriptions have not been mentioned.

CHAPTER 4

Solution Approach

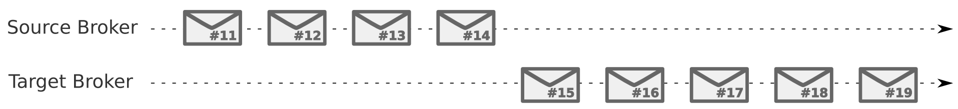
In this chapter, we present our solution approach to enable mobility and message delivery guarantees in distributed MQTT networks and describes the migration process. First, we preset the fundamental idea of our solution approach (Section 4.1) and state our assumption and what impact they have (Section 4.2). We define three migration phases and describe the tasks that need to be performed during each phase on the systems (Section 4.3). In order to coordinate the migration process, we introduce different identifiers for messages and systems (Section 4.4), present the migration protocol (Section 4.6), that is used to communicate between individual systems and visualize the migration sequence in (Section 4.7). Finally, we discuss the different synchronization states (Section 4.8) and describe in and visualize in detail the synchronization process between the Source Broker and the Target Broker (Section 4.9).

4.1 Foundation

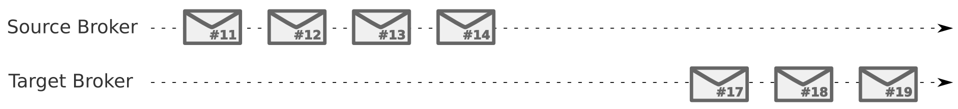
When considering the message stream for a Client through the migration phases, there are three possible scenarios the resulting message stream might look like[1]:

- **Scenario 1**: The Client receives every message exactly once (Figure 4.1a).
- **Scenario 2**: The Client does not receive some messages (Figure 4.1b).
- **Scenario 3**: The Client receives some messages multiple times (Figure 4.1c).

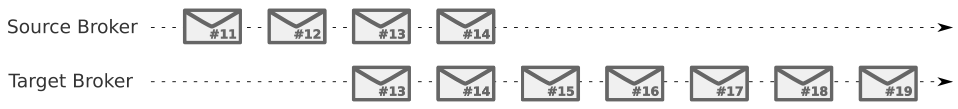
[1]The figures visualize the streams of messages of a Source Broker and a Target Broker to a Client. The numbers in the scenario represent the unique message ids of the messages that are received by the Client.



(a) Scenario 1: every message exactly once



(b) Scenario 2: missing messages



(c) Scenario 3: duplicate messages

Figure 4.1: Message Scenarios

The combined message stream of the Source Broker and the Target Broker represent the message stream that a mobile client receives during the migration process from the Source Broker to the Target Broker. This message stream must fulfill the defined message delivery guarantee (QoS 0-2) before, during and after the migration in order to be valid. Table 4.1 shows the validity of the three message scenarios regarding to the different QoS.

Table 4.1: Scenario validity for different QoS

Scenario	**QoS 0**	**QoS 1**	**QoS 2**
Scenario 1	Valid ✓	Valid ✓	Valid ✓
Scenario 2	Valid ✓	Invalid ✗ (no losses expected)	Invalid ✗ (no losses expected)
Scenario 3	Invalid ✗ (no duplicates expected)	Valid ✓	Invalid ✗ (no duplicates expected)

We can see, that Scenario 1 is valid with all QoS, Scenario 2 is invalid for QoS 1 and QoS 2, but valid for QoS 0 and therefore messages in QoS 0 can be lost during the migration process. Scenario 3 is invalid for QoS 0[2] and QoS 2, but valid for QoS 1 and therefore messages in QoS 1 can arrive multiple times during the migration process.

These specific characteristics in the message delivery guarantees for subscriptions are the foundation of our work.

[2]In a MQTT setup with a single broker no duplicates can happen, since the broker will just fire-and-forget. Nevertheless, in a broker network with multiple brokers, these is is not true anymore, since multiple brokers could send the same message. The MQTTs v3 specification is not specific enough if we consider a MQTT broker network and leaves room for interpretation if duplicates are allowed for QoS 0. In our work, we expect no duplicates even with multiple brokers. Nevertheless, our solution approach can easily be relaxed to allow duplicates for QoS 0 as well.

4.2 Assumptions

We made the following assumptions for our solution:

- The Client is online during the whole migration process, so that we are able to perform a planned migration and to sent messages from both brokers at the same time. As planned migrations can be extended with methods for unplanned migrations when the Client disconnects, this assumption has as a minor impact.
- Messages have a total ordering for a single topic, so that we can compare messages with each other and ensure the message delivery guarantees for subscriptions. In our approach, we achieve this by serving a topic by just one publisher, so that we get a message sequence ordering for each subscription of the Client (see Section 4.4.3). Usually, multiple publishers can publish messages to the same topic. Therefore, this assumption has a decent impact and solving this limitation is left for future work. Nevertheless, a publisher can publish to multiple topics, since the synchronization process can handle gaps in the message stream.
- Messages arrive at the brokers in order, so that we can ensure the message delivery guarantees during the migration process. This has a minor impact, since we can use methods like windowing to achieve this (see Section 4.4.3).
- Publishers publishes all messages with QoS 2, so that the requested QoS from the Client is actually used. This assumption has a minor impact, but needs further investigations as stated in the referred section (see Section 2.6.1).

4.3 Migration Phases

We split the whole migration process into three phases: pre-migration phase, migration phase and post-migration phase. In the pre- and post-migration phase, only one connection between the Client and the broker in the network is established and the publish-subscribe systems works normally. During the migration process, in the migration phase, the Client has multiple connections to brokers in the network. One connection to the Source Broker and one connection to the Target Broker. Illustrations of the three phases for the parties are shown in Figure 4.2.

Within the different phases, the broker systems are performing different tasks.

The Source Broker has to perform the following tasks in the pre-migration phase:

- Check if a Client with the specified client id exists
- Collect the subscriptions of the Client

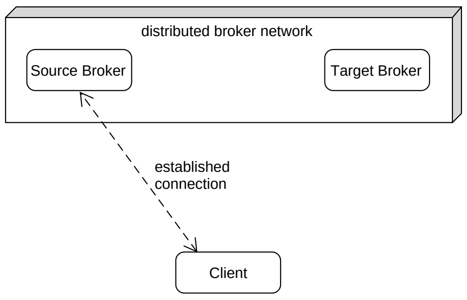
- Retrieve current state of the broker

The Target Broker has to perform the following tasks in the migration phase:

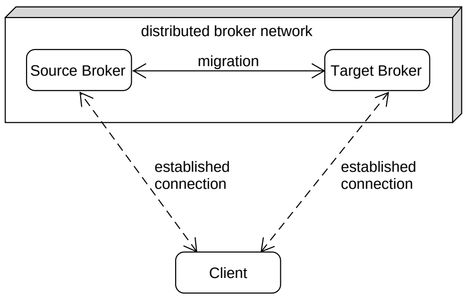
- Register Client internally repository
- Connect to the Client
- Subscribe the Client to its subscriptions
- Synchronize with Source Broker

The Source Broker has to perform the following tasks in the post-migration phase:

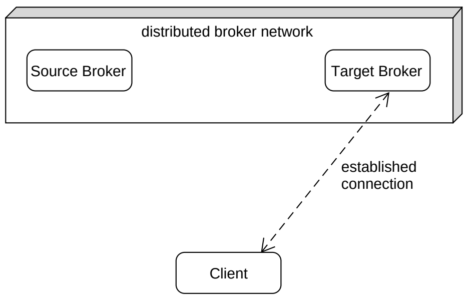
- Release stored messages for the migration
- Disconnect from the Client
- Remove the Client from internal repository



(a) Pre-migration phase



(b) Migration phase



(c) Post-migration phase

Figure 4.2: Migration Phases

4.4 Identifiers

In order to coordinate the migration process, we reuse or introduce different identifiers for messages and systems.

4.4.1 Client Id

This id identifies a client (subscriber and/or publisher) in the network. The id of a client must be unique in the whole network to identify clients. This id is used by the Coordinator to trigger a migration for a specific Client and brokers use this id to check if they already serve a specific Client.

4.4.2 Message Id

To guarantee that messages are only delivered or received once, a message must be identifiable. For MQTT QoS 1 and QoS 2, the sender of a PUBLISH packet generates a unique identify for each packet to coordinate the delivery, called *Packet Identifier* or *Message Id*. MQTT v3 uses a 16-bit long numeric value. If the delivery is complete, the message id can be reused. Therefore, only limited amount of available message ids is necessary.

As described in Section 2.6, a broker system serves two roles: *receiver* and *forwarder*. It receives published messages from publishers and forwards/publishes messages to subscribers again. It is important to note that the message is always generated by the sender, hence, the message id between the publisher to the broker and between the broker and the subscriber is usually different (by chance, it could occur that they are the same, but they are not related). Figure 4.3 shows the individual message id ranges colored in red.

Therefore, two different publishers could publish a message with the same message id. If the broker or the broker network would use the message id to forward them to subscribers, subscribers could get two messages with the same message id, and therefore not distinguish two different deliveries anymore. For QoS 0 and QoS 2, this would violate the QoS guarantee, since a message would appear as a duplicate. Therefore, the broker generates its own message id to forward messages to the subscriber.

4.4.3 Global Message Id

As stated the previous section, the message id provided by MQTT protocol is only unique per client per delivery and therefore, the message id from a received message can not be used to identify a message in a broker network. Therefore we define a *Global Message Id*. The global message id must be unique for every message in the whole broker network.

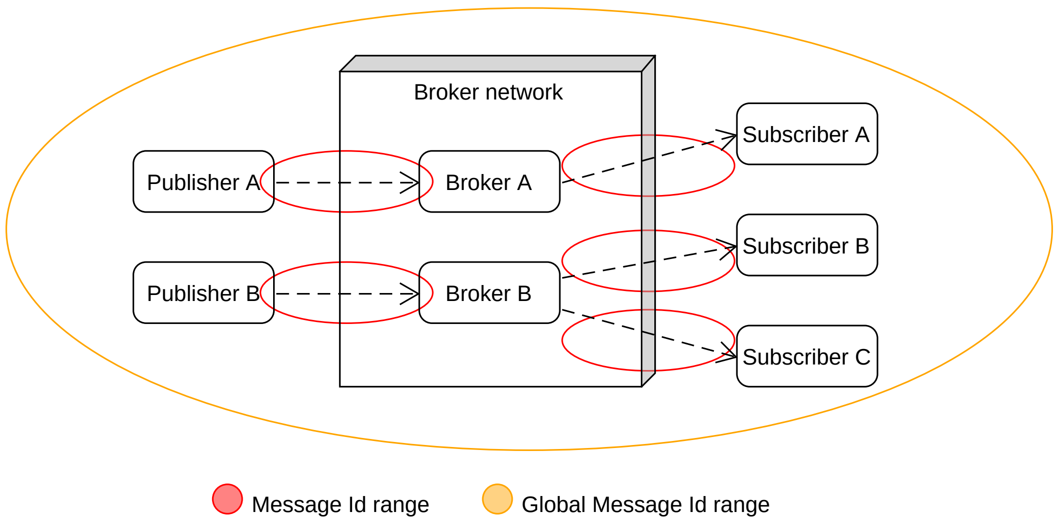


Figure 4.3: Message Id (red) and Global Message Id (orange) ranges.

If the message is routed through broker network, the message id might change, but the global message id is the same. Figure 4.3 shows the global message id range colored in orange.

Since we assume that messages are ordered, the global message id of messages must be comparable. We achieve this, by using an increasing sequence number, that can be compared. Since every client has a unique client id, all published messages on a publisher just need to concatenate a local unique sequence number to get a global unique message identifier. This approach is also used for Logging as mentioned in [MJ10].

The global message id is generated by:

$$globalMessageId = clientId + sequenceNumber$$

The global message id will be sent as part of the header information of the published message. As a separator to distinguish the client id from the comparable sequence, we use a dash “-”. For further details, how additional meta information is sent, see Section 4.5.

If we want consider client failures and restarts or that the sequence number cannot be stored, we might prefix, e.g., the startup time of the client as well to get a unique number.[3]

In order to get an ordered message stream, windowing techniques, as presented in [GÖ05], can be used, such that, within a specified time span, messages are checked, compared and reordered. All messages that arrived too late and do not fit into the ordered stream will

[3]This is not considered in our work.

automatically be discarded, such that the final outcome is always an ordered message stream. Nevertheless, windowing techniques are not part of our work. We assume that messages that arrive at the broker are already in order and that messages will get ordered on the subscriber.

4.4.4 Migration Id

Every migration process is identified by a unique identifier, the *Migration Id*. Brokers use the migration id to assign individual packets from the migration protocol to a migration process for a specific client. The migration id is part of every migration protocol packet.

In our work we use an UUID for the migration id as a unique identifier. The migration id is generated during the migration initialization on the Source Broker.

4.5 Enhanced MQTT PUBLISH Packet

To uniquely identify a message, we need to send additional header information like the Global Message Id with every MQTT PUBLISH message. Since it is not possible to send additional meta information in a “header section” of the Publish Message with MQTT v3 per se [MQT15], we split the payload into a header and body part to send additional information within the payload, as shown in Figure 4.4. Hence, we reuse MQTTs processes and its message delivery guarantees that apply to MQTT messages and can add additional meta information to it. Contrary, if we would wrap the MQTT PUBLISH packets into another packet that contains additional meta information, we would lose MQTTs message delivery guarantees and would have to implement a new communication process. With this approach, MQTT v3 can be used as it is to scale systems.

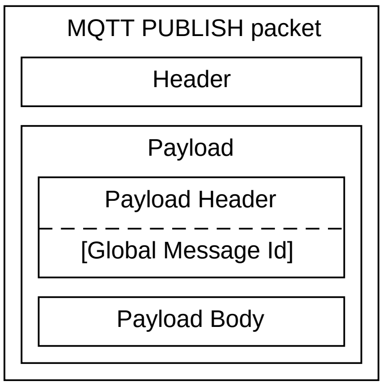


Figure 4.4: Enhanced MQTT PUBLISH packet.

MQTT v5 has enhancements for scalability and large scale systems and an extensibility mechanisms to include user properties in messages [MQT19]. The Global Message Id could be such a user property and therefore, the message payload of a MQTT PUBLISH message could just consist of the body part. Hence, an additional header part would not be necessary anymore.

With MQTT v3 we define the payload for the MQTT PUBLISH packet as follows:

`MqttPayload` class

Property name	Type	Description
header	MqttPayloadHeader	Header part of the published message.
body	MqttPayloadBody	Body part of the published message.

4.5.1 Payload Header

The header contains additional information for the message. We use it to store the unique global message id of the message.

`MqttPayloadHeader` class

Property name	Type	Description
globalMessageId	GlobalMessageId	Globally unique message id of the published message.

4.5.2 Payload Body

The body will contain the actual payload of the MQTT Publish Message.

`MqttPayloadBody` class

Property name	Type	Description
payload	String	Actual payload of the published message.

4.6 Migration Protocol

To perform the migration process communication between systems, we introduce the following migration protocol packets:

Packet name	**Description**
MIGRATE	Packet sent from the Source Broker to the Target Broker to initiate a new migration of a Client.
MIGACK	Packet sent from the Target Broker to the Source Broker to acknowledge a migration of a Client.
MIGTO	Packet sent from the Target Broker to the Client to indicate that it should connect to a new broker (Target Broker) in order to get migrated.
MIGTOACK	Packet sent from the Client to the new broker (Target Broker) to acknowledge a successful connection to the new broker and that it is ready to be migrated.
MIGSYNC	Packet sent from the Target Broker to the Source Broker to indicate that the Target Broker can not fully serve the Client with all messages and that the Source Broker needs to serve missing messages.
MIGSYNCACK	Packet sent from the Source Broker to the Target Broker to acknowledge that all missing messages were successfully send to the Client.

4.7 Migration Sequence

The sequence of the migration process between the Source Broker, the Target Broker and the Client to migrate the Client from the Source Broker to the Target Broker is illustrated in Figure 4.5. The following sections describe the migration process in more detail.

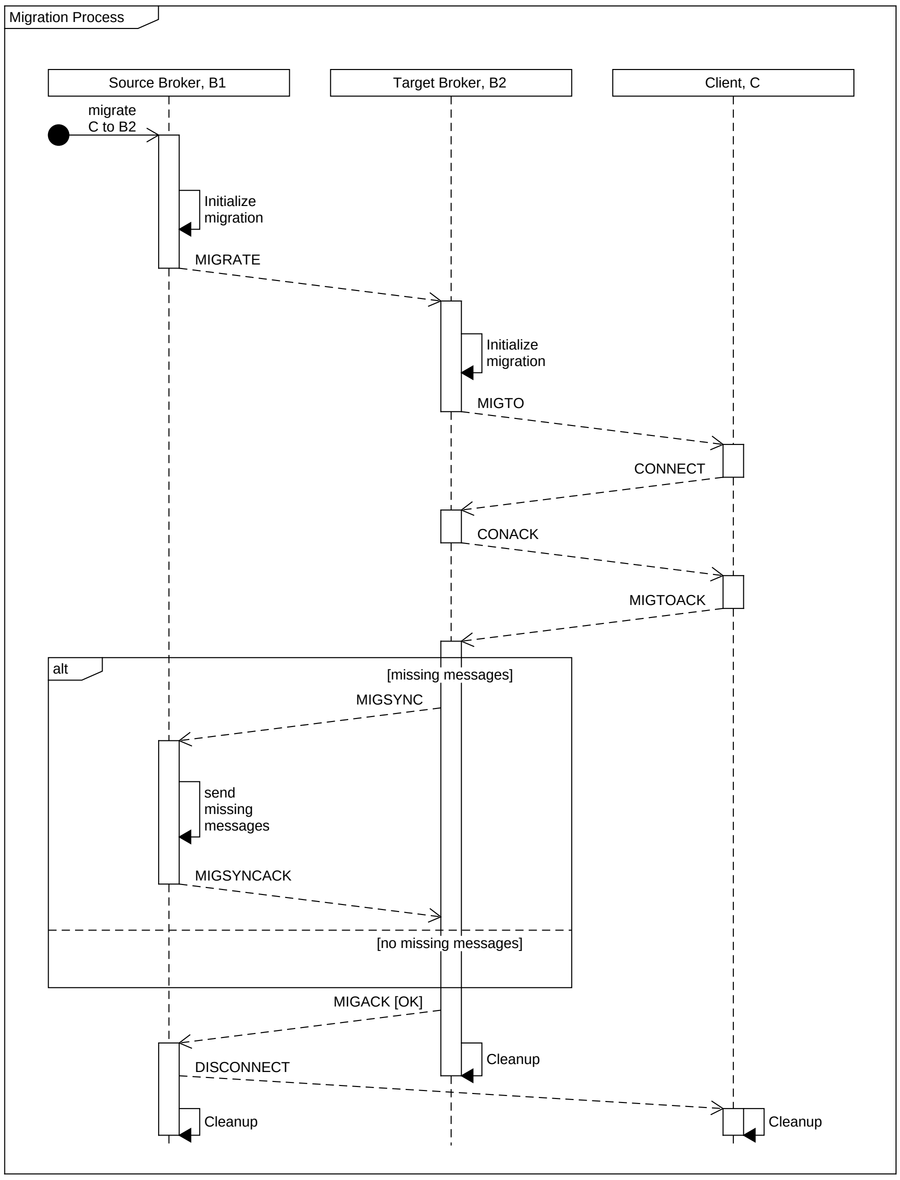


Figure 4.5: Sequence diagram of the migration process of Client C from Source Broker B1 to Target Broker B2

4.8 Synchronization State

To assure the message delivery guarantees for the Client during the migration process, the brokers have to synchronize the delivery of their messages for each subscription to the Client. Depending on the QoS level of a subscription, the Source Broker and the Target Broker have to exchange state information between each other. Details about the synchronization process for each state and QoS are described in Section 4.9.

The synchronization state of the received messages of the Target Broker compared to the Source Broker can be in one of three different states as shown in Figure 4.6 and Figure 4.7:

- **ahead**: The Target Broker is ahead of the Source Broker and already got messages, that the Source Broker did not receive yet. Since the Target Broker is ahead, some messages have already been processed by the Target Broker. Therefore, synchronization might be necessary, to request missed messages from the Source Broker, in order to fulfill the QoS for subscriptions.

- **synced**: The Target Broker received the same messages as the Source Broker and therefore is synced with the Source Broker. Therefore, no synchronization is necessary, since the Target Broker can just continue delivering messages to the Client where the Source Broker left off.

- **behind**: The Target Broker is behind the Source Broker and did not yet receive all messages. No additional synchronization between the brokers is necessary, since the Target Broker will (soon) get missing messages, that have already been processed by the Source Broker. The Target Broker might has to skip some message to fulfill the QoS for a subscription.

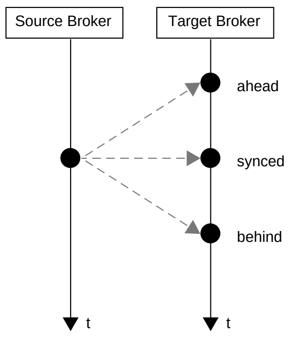


Figure 4.6: Synchronization states between brokers.

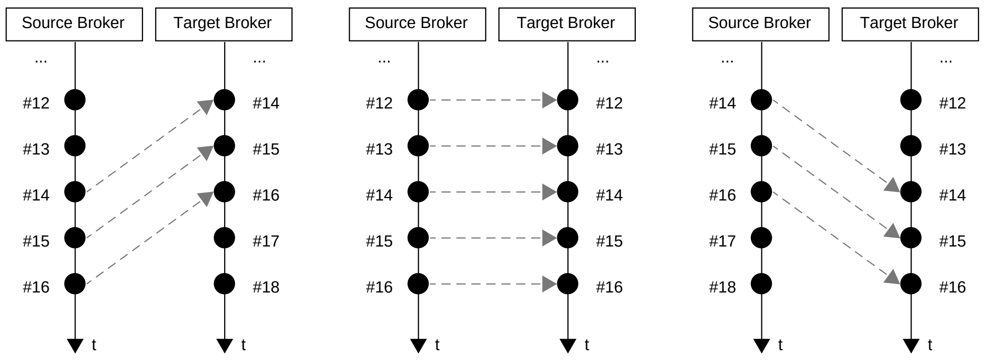


Figure 4.7: Synchronization state of Target Broker is 'ahead', 'synced' or 'behind'.

4.9 Synchronization Process

The actual synchronization process depends on the synchronization state of the broker and the QoS of the client's subscription.

$$\{\text{ahead, synced, behind}\} \times \{\text{QoS 0, QoS 1, QoS 2}\}$$

In the following sections describe the synchronization process for each synchronization state with each QoS and the interaction for the Source Broker and the Target Broker:

- Target Broker is ahead of the Source Broker with QoS 0, Qos 1 and QoS 2
- Target Broker is synced with the Source Broker with QoS 0, Qos 1 and QoS 2
- Target Broker is behind of the Source Broker with QoS 0, Qos 1 and QoS 2

The legend of the illustrations is shown in Figure 4.8. Each box defines a message. The gray "sent" box defines a message that was sent to the Client. The dashed "not sent" box defines a message that is not sent to the Client. The blue "skipped" box defines a message that was skipped, not stored and not send to the Client. The green "skipped + stored" box defines a message that was skipped, stored and not directly send to the Client. If the background is gray, it was send later within the synchronization process. The red "last processed" box defines the message that was chosen for the synchronization process, that will be used to to define the synchronization state. The number in the boxes describe the sequence number of the global message id.

The current message is referred by msg, with msg_{id} referring to the specific global message id of the message. A single migration process is referred by mig. The current state of a migration mig is defined by mig_{state}.

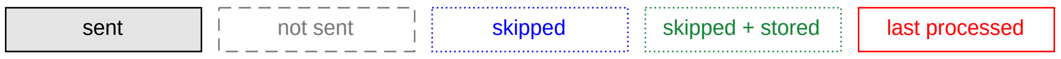


Figure 4.8: Legend for synchronization figures

Table 4.2: Migration states.

Migration state	**Description**
INITIALIZED	Migration is created.
SYNCING	Migration synchronization is in progress.
SYNCED	Migration is synchronized.
FINISHED	Migration is finished.
ERROR	Migration error occurred.

The migration state of the Source Broker is referred by mig^{source} and the migration state of the Target Broker is referred by mig_{state}^{target}. All possible states of a migration mig_{state} during the migration process are listed in Table 4.2.

A migration mig may consists of multiple subscriptions. A single subscription of a migration is referred by sub. The last processed message that was send to the Client for a subscription on the Source Target is defined by $sub_{lastProcId}^{source}$. It is sent to the Target Broker within the MIGRATE packet. The last processed message for a subscription on the Target Broker is defined by $sub_{lastProcId}^{target}$. It is sent to the Source Broker within the MIGSYNC packet, if synchronization is necessary.

A subscription with a specific QoS q, $q = \{0, 1, 2\}$, is referred by sub^q, e.g. sub^0 refers to a subscriptions with QoS 0. The current state of a subscription sub is defined by sub_{state}.

$$\begin{aligned} mig_{state} = \{&INITIALIZED,\\ &SYNCING,\\ &SYNCED,\\ &FINISHED,\\ &ERROR\} \end{aligned} \qquad \begin{aligned} sub_{state} = \{&INITIALIZED,\\ &SYNCING,\\ &SYNCING\_THIS,\\ &SYNCING\_OPPONENT,\\ &SYNCED,\\ &FINISHED,\\ &ERROR\} \end{aligned}$$

4.9.1 Target Broker is ahead of the Source Broker

In this case, for a specific subscription, the last processed message id on the Source Broker is smaller than the last processed message id on the Target Broker and therefore the Target Broker is ahead of the Source Broker:

$$sub_{lastProcId}^{source} < sub_{lastProcId}^{target}$$

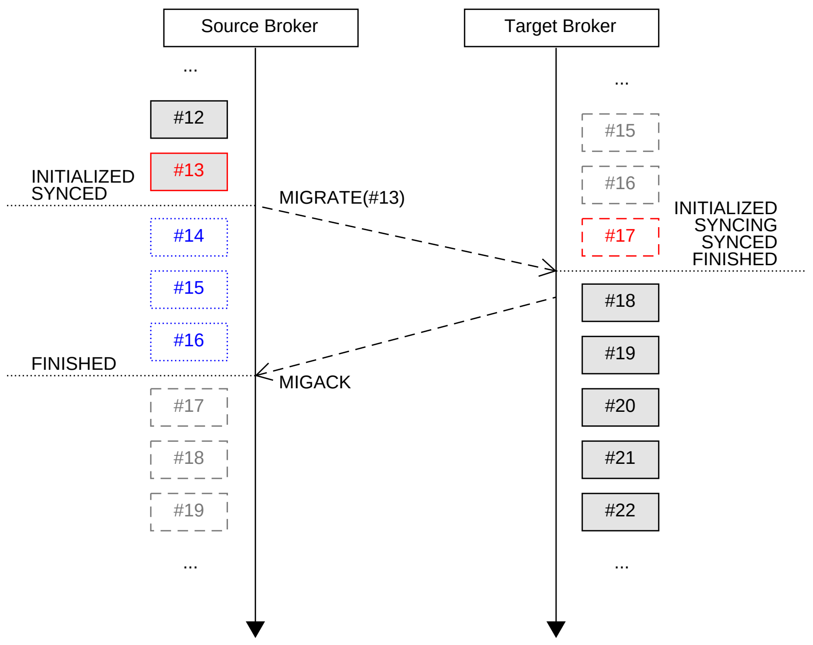
QoS 0

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and it sets its migration state immediately to SYNCED, $mig_{state}^{source} := SYNCED$, since no further synchronization will be necessary for QoS 0 on the Source Broker. Messages that arrive in the meanwhile can be skipped and do not have to be stored, since lost messages are valid for QoS 0. It sends a MIGRATE packet to the Target Broker.

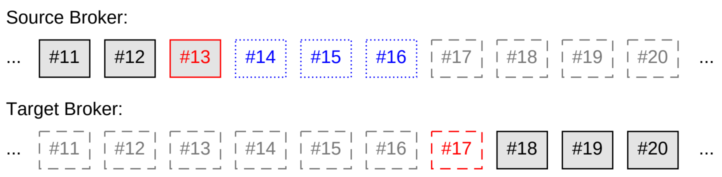
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client. It starts sending messages immediately and set its migration state to SYNCED, $mig_{state}^{target} := SYNCED$. It finishes its migration since no further actions are necessary, $mig_{state}^{target} := FINISHED$, and sends a MIGACK packet to the Source Broker. Messages that are missed will not been synchronized, since lost messages are valid for QoS 0.

Source Broker: On receive of the MIGACK packet, the migration gets finished as well, $mig_{state}^{source} := FINISHED$.

The interaction is visualized in Figure 4.9.



(a) Sequence



(b) Stream

Figure 4.9: Target Broker is ahead of the Source Broker - QoS 0

QoS 1

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and since it is valid with QoS 1 to have duplicate messages, it will continue sending messages and sends a MIGRATE packet to the Target Broker.

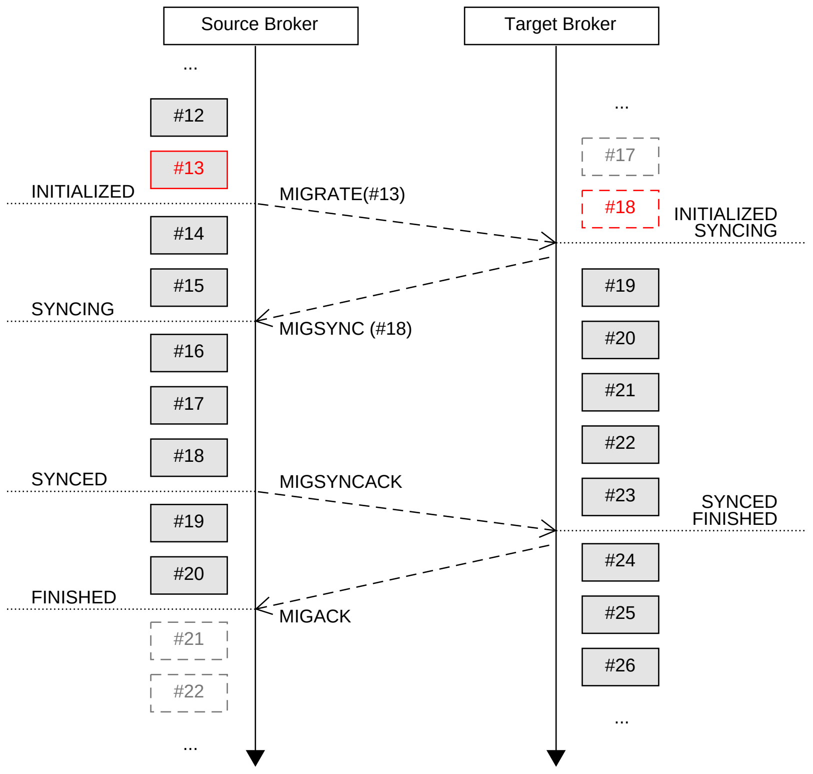
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. Since it is ahead, it sends a MIGSYNC packet with its last processed message id to the Source Broker to request the missing messages. Furthermore, it starts sending new messages immediately.

Source Broker: On receive of the MIGSYNC packet, it sets its migration state to SYNCING, $mig_{state}^{source} := SYNCING$, and it waits until the last processed message id of from the Target Broker is processed. When it is synced it sets its migration state to SYNCED, $mig_{state}^{source} := SYNCED$, and sends a MIGSYNCACK packet to the Target Broker, to acknowledge that messages up to its requested id are sent to the Client.

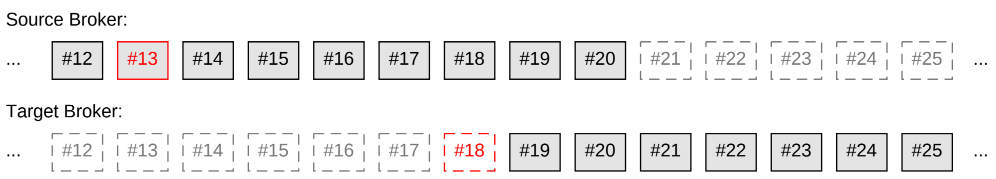
Target Broker: On receive of the MIGSYNCACK packet, it sets its migration state to SYNCED, $mig_{state}^{target} := SYNCED$, finishes its migration, $mig_{state}^{target} := FINISHED$, and the MIGACK packet is send.

Source Broker: On receive of the MIGACK packet, the migration gets finished as well, $mig_{state}^{source} := FINISHED$, and it stops sending messages to the Client.

The interaction is visualized in Figure 4.10.



(a) Sequence



(b) Stream

Figure 4.10: Target Broker is ahead of the Source Broker - QoS 1

QoS 2

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and it will stop sending further messages to the Client. The unsent messages are stored in the migration message store, if a rollback is necessary. A MIGRATE packet is sent to the Target Broker.

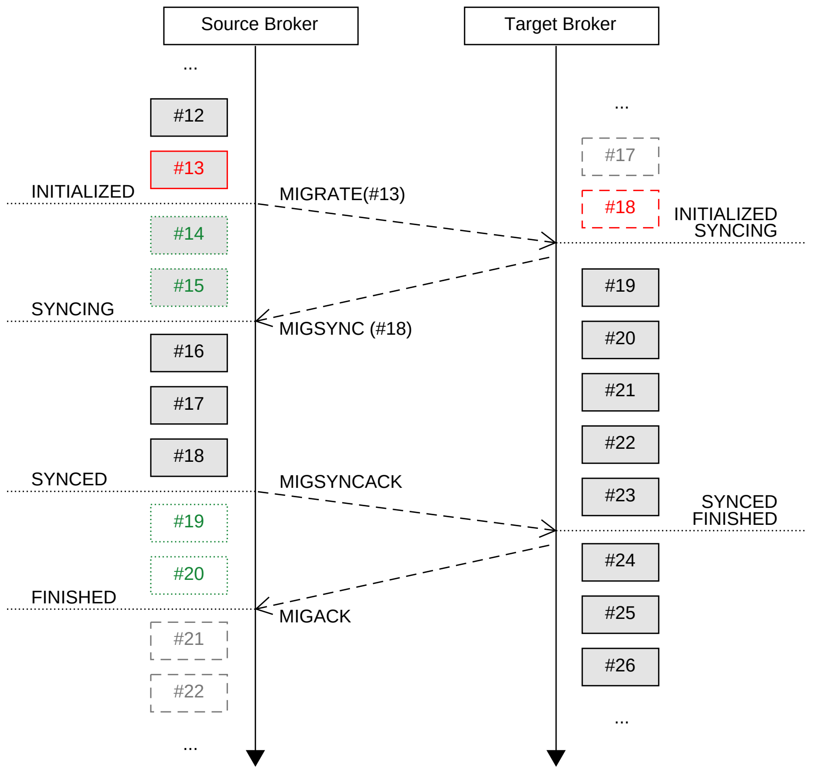
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. Since it is ahead, it sends a MIGSYNC packet with its last processed message id to the Source Broker, to request the missing messages. Furthermore, it starts sending new messages immediately.

Source Broker: On receive of the MIGSYNC packet, it sets its migration state to SYNCING, $mig_{state}^{source} := SYNCING$, it sends missed messages from the store and if the id is still ahead, it waits until the last processed message id of from the Target Broker is processed. When it is synced it sets its migration state to SYNCED, $mig_{state}^{source} := SYNCED$, and sends a MIGSYNCACK packet to the Target Broker, to acknowledge that messages up to its requested id are sent to the Client.

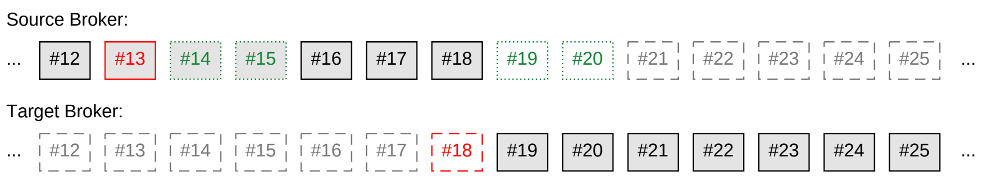
Target Broker: On receive of the MIGSYNCACK packet, it sets its migration state to SYNCED, $mig_{state}^{target} := SYNCED$. It finishes its migration, $mig_{state}^{target} := FINISHED$, and the MIGACK packet is send.

Source Broker: On receive of the MIGACK packet, the migration gets finished as well, $mig_{state}^{source} := FINISHED$, the migration message store gets cleared and no further messages are stored.

The interaction is visualized in Figure 4.11.



(a) Sequence



(b) Stream

Figure 4.11: Target Broker is ahead of the Source Broker - QoS 2

4.9.2 Target Broker is synced with the Source Broker

In this case, for a specific subscription, the last processed message id on the Source Broker is equal to the last processed message id on the Target Broker and therefore the Target Broker is synced with the Source Broker:

$$sub^{source}_{lastProcId} = sub^{target}_{lastProcId}$$

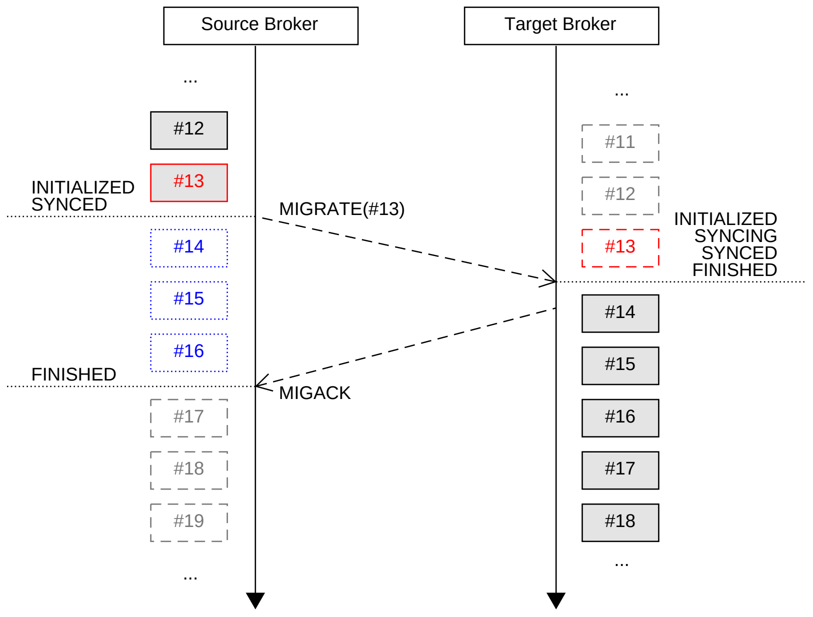
QoS 0

Source Broker: The migration process is initialized, $mig^{source}_{state} := INITIALIZED$, and it sets its migration state immediately to SYNCED, $mig^{source}_{state} := SYNCED$, since no further synchronization will be necessary for QoS 0 on the Source Broker. Messages that arrive in the meanwhile can be skipped and do not have to be stored, since lost messages are valid for QoS 0. A MIGRATE packet is sent to the Target Broker.

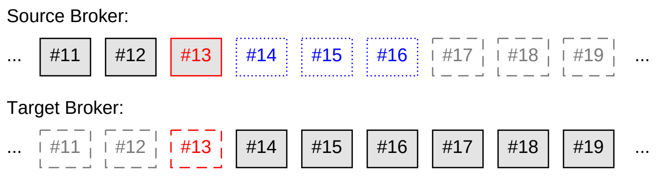
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig^{target}_{state} := INITIALIZED$, connects to the Client and starts syncing, $mig^{target}_{state} := SYNCING$. It immediately sets its migration state to SYNCED, $mig^{target}_{state} := SYNCED$, and starts sending messages immediately. It finishes its migration since no further actions are necessary, $mig^{target}_{state} := FINISHED$, and sends the MIGACK packet to the Source Broker.

Source Broker: On receive of the MIGACK packet, the migration gets finished as well, $mig^{source}_{state} := FINISHED$.

The interaction is visualized in Figure 4.12.



(a) Sequence



(b) Stream

Figure 4.12: Target Broker is synced with the Source Broker - QoS 0

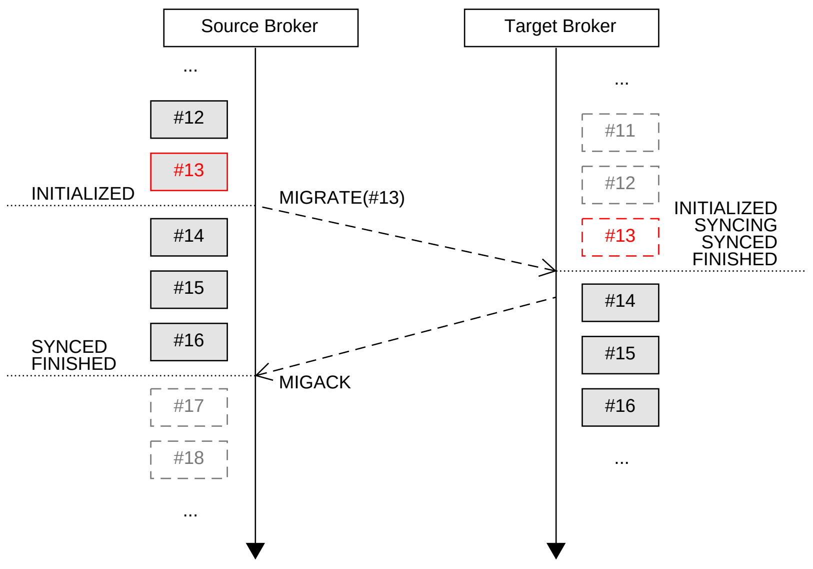
QoS 1

<u>*Source Broker:*</u> The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and since it is valid with QoS 1 to have duplicate messages, it will continue sending messages. A MIGRATE packet is sent to the Target Broker.

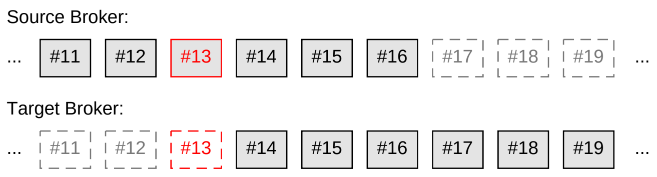
<u>*Target Broker:*</u> On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. It immediately sets its migration state to SYNCED, $mig_{state}^{target} := SYNCED$, and starts sending messages immediately. It finishes its migration since no further actions are necessary, $mig_{state}^{target} := FINISHED$, and sends the MIGACK packet to the Source Broker.

<u>*Source Broker:*</u> On receive of the MIGACK packet, the migration state is set to SYNCED, $mig_{state}^{source} := SYNCED$. It gets finished as well, $mig_{state}^{source} := FINISHED$, and it stops sending messages to the Client.

The interaction is visualized in Figure 4.13.



(a) Sequence



(b) Stream

Figure 4.13: Target Broker is synced with the Source Broker - QoS 1

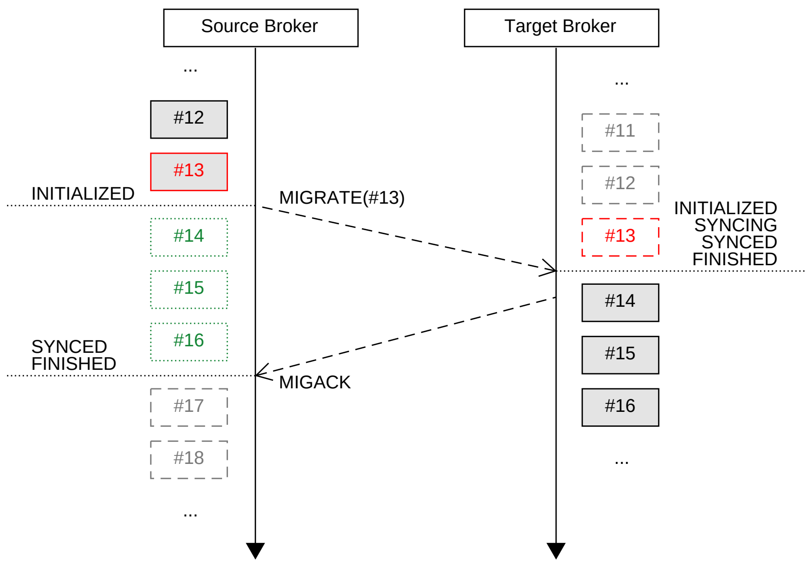
QoS 2

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and it will stop sending further messages to the Client. The unsent messages are stored in the migration message store, if a rollback is necessary. A MIGRATE packet is sent to the Target Broker.

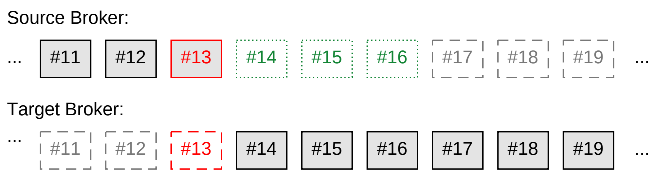
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. It immediately sets its migration state to SYNCED, $mig_{state}^{target} := SYNCED$, and starts sending messages immediately. It finishes its migration since no further actions are necessary, $mig_{state}^{target} := FINISHED$, and sends the MIGACK packet to the Source Broker.

Source Broker: On receive of the MIGACK packet, the migration is set to SYNCED, $mig_{state}^{source} := SYNCED$. It gets finished as well, $mig_{state}^{source} := FINISHED$, the migration message store gets cleared and no further messages are stored.

The interaction is visualized in Figure 4.14.



(a) Sequence



(b) Stream

Figure 4.14: Target Broker is synced with the Source Broker - QoS 2

4.9.3 Target Broker is behind of the Source Broker

In this case, for a specific subscription, the last processed message id on the Source Broker is greater than the last processed message id on the Target Broker and therefore the Target Broker is behind of the Source Broker:

$$sub_{lastProcId}^{source} > sub_{lastProcId}^{target}$$

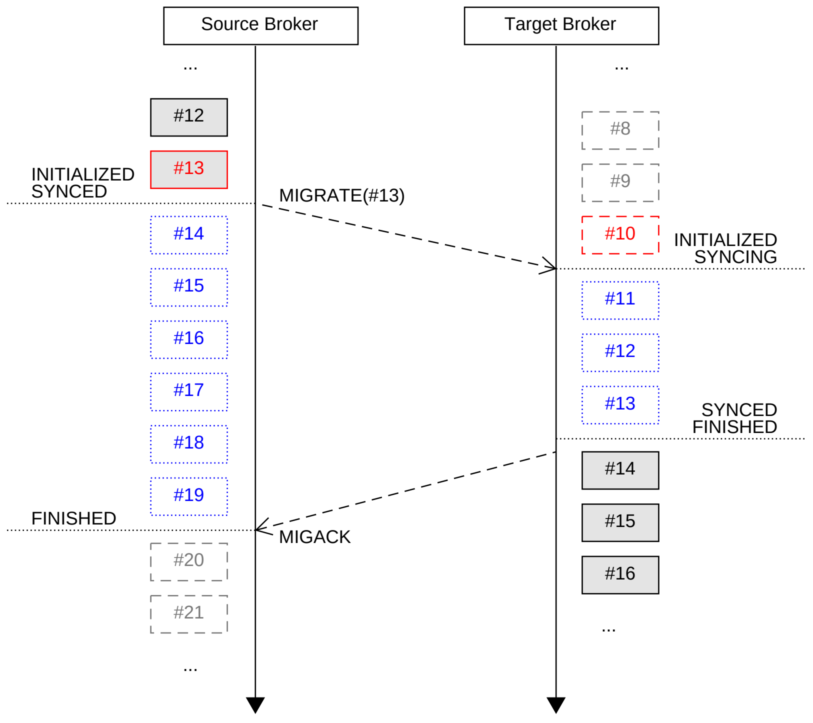
QoS 0

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$. It sets its migration state immediately to SYNCED, $mig_{state}^{source} := SYNCED$, since no further synchronization will be necessary for QoS 0 on the Source Broker. Messages that arrive in the meanwhile can be skipped and do not have to be stored, since lost messages are valid for QoS 0. A MIGRATE packet is sent to the Target Broker.

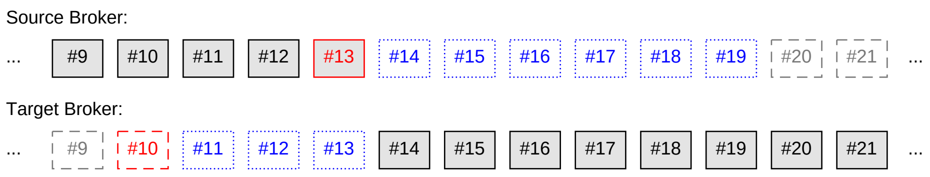
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. It waits until the last processed message id from the Source Broker has passed and skips all messages until then. After that, it starts sending messages, set its migration state to SYNCED, $mig_{state}^{target} := SYNCED$. It finishes its migration, $mig_{state}^{target} := FINISHED$, and sends the MIGACK packet to the Source Broker.

Source Broker: On receive of the MIGACK packet, the migration gets finished as well, $mig_{state}^{source} := FINISHED$.

The interaction is visualized in Figure 4.15.



(a) Sequence



(b) Stream

Figure 4.15: Target Broker is behind of the Source Broker - QoS 0

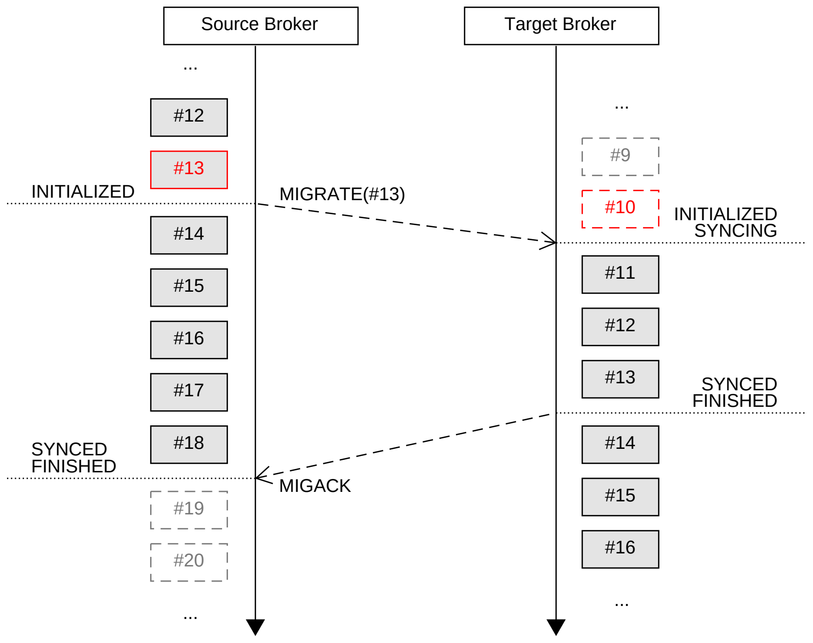
QoS 1

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and since it is valid with QoS 1 to have duplicate messages, it will continue sending messages. A MIGRATE packet is sent to the Target Broker.

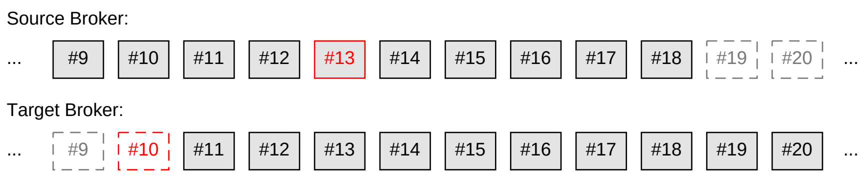
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. It starts sending new messages immediately and waits until the last processed message id from the Source Broker has passed. After that, it sets its migration state to SYNCED, $mig_{state}^{target} := SYNCED$. It finishes its migration, $mig_{state}^{target} := FINISHED$, and sends the MIGACK packet to the Source Broker.

Source Broker: On receive of the MIGACK packet, the migration state is set to SYNCED, $mig_{state}^{source} := SYNCED$. It gets finished as well, $mig_{state}^{source} := FINISHED$, and it stops sending messages to the Client.

The interaction is visualized in Figure 4.16.



(a) Sequence



(b) Stream

Figure 4.16: Target Broker is behind of the Source Broker - QoS 1

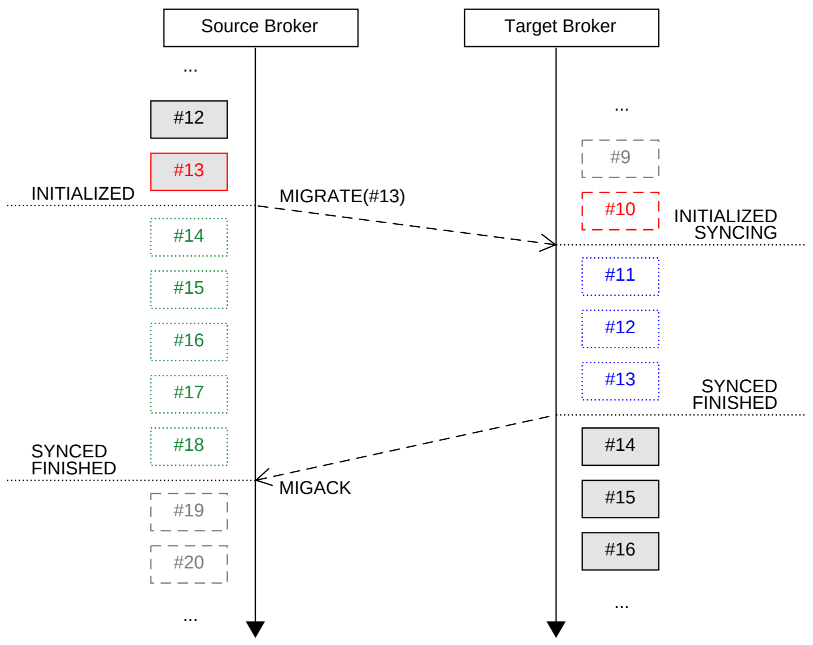
QoS 2

Source Broker: The migration process is initialized, $mig_{state}^{source} := INITIALIZED$, and it will stop sending further messages to the Client. The unsent messages are stored in the migration message store, if a rollback is necessary. A MIGRATE packet is sent to the Target Broker.

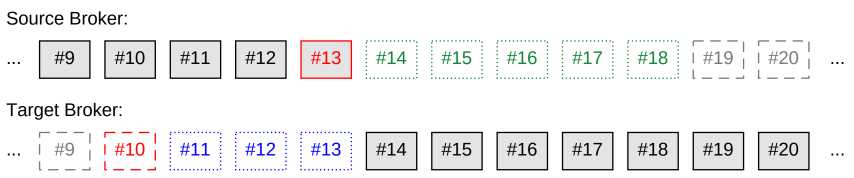
Target Broker: On receive of the MIGRATE packet, it initializes the migration, $mig_{state}^{target} := INITIALIZED$, connects to the Client and starts syncing, $mig_{state}^{target} := SYNCING$. It waits until the last processed message id from the Source Broker has passed, sets its migration state to SYNCED, $mig_{state}^{target} := SYNCED$, and starts sending messages. It finishes its migration since no further actions are necessary, $mig_{state}^{target} := FINISHED$, and sends the MIGACK packet to the Source Broker.

Source Broker: On receive of the MIGACK packet, the migration state is set to SYNCED, $mig_{state}^{source} := SYNCED$. It gets finished as well, $mig_{state}^{source} := FINISHED$, the migration message store gets cleared and no further messages are stored.

The interaction is visualized in Figure 4.17.



(a) Sequence



(b) Stream

Figure 4.17: Target Broker is behind of the Source Broker - QoS 2

CHAPTER 5

Implementation

In this chapter, we present the concrete implementation of our solution approach that was introduced in Chapter 4. We give an overview of the migration framework and all its components (Section 5.1) and describe the integration of it into a concrete MQTT broker (Section 5.2) and MQTT client implementation (Section 5.3) with implementation details of individual components and details about the migration process on the specific systems. Finally, we introduce the communication implementation between systems (Section 5.4) and describe the migration protocol packets that are used (Section 5.5).

5.1 Migration Framework

We create a migration framework for the solution approach that was introduced in Chapter 4, which can be integrated into MQTT broker and MQTT client implementations to enable mobility and message delivery guarantees while migrating clients. The migration framework was written in the Java programming language and comprises 58 classes with 3.8k lines of code. The source code is published and available in our project repository[1].

A brief overview of all components and its functionalities as shown in Figure 5.1 and Figure 5.8 as follows:

- **MQTT Broker**
  The broker system processes MQTT messages from publishers to subscribers. It is extended with the following migration framework components, that allows a reliable migration of clients from one broker to another. The broker system and its components are shown in Figure 5.1. See Section 5.2 for further details.

[1]`https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration`

  - **Broker Migration Manager**
    Processes migration requests and handles the migration process for the broker.
  - **Broker Migration Bridge**
    Communication bridge between the migration framework and a specific broker implementation.
  - **Broker Migration Integrator**
    Provides an interface to integrate the migration framework for a specific broker implementation.
  - **Migration Store**
    Stores information about migrations, such as the client id, the migration state and the subscriptions to migrate.
  - **Migration Message Store**
    Stores MQTT messages that may be needed for synchronization or rollback scenarios.
  - **Message Barrier**
    Checks if an outgoing MQTT message for a MQTT client should be sent.
  - **Broker Command Line Interface (Broker CLI)**
    Interface to send commands to the broker, e.g. to migrate a client to another broker.

- **MQTT Client**
  The client system publishes MQTT messages to a MQTT broker or subscribes its interests to receive MQTT messages from a MQTT broker. It is extended with the following migration framework components, that allows a reliable migration of clients from one broker to another. The client system and its components are shown in Figure 5.8. See Section 5.3 for further details.

  - **Client Migration Manager**
    Processes migration requests and handles the migration process for the client.
  - **Client Migration Bridge**
    Communication bridge between the migration framework and a specific client implementation.
  - **Client Command Line Interface (Client CLI)**
    Interface to send commands to the client, e.g. list all connected brokers.

- **Communication**
  The communication components are used to exchange migration protocol packets across systems. See Section 5.4 for further details.

  - **Migration Packet Server**
    Receiving and handling migration protocol packets from a Migration Packet Client component.

- **Migration Packet Client**
  Sending migration protocol packets to a Migration Packet Server component.

5.2 MQTT Broker

To enable mobility and message delivery guarantees to migrate MQTT clients we introduce the following components in our migration framework. A component overview for the broker components is shown in Figure 5.1.

Some of the features include:

- analyze incoming messages
- register/unregister clients
- subscribe/unsubscribe topics for clients
- message barrier for clients
- store messages for clients
- publish stored messages to clients
- perform the migration process

As our base MQTT broker we chose the JVM lightweight *Moquette* MQTT broker[2] and integrated our migration framework. The source code of our fork is available in our project repository[3].

5.2.1 Broker Migration Manager

The Broker Migration Manager is the central component on the broker for the migration process. It handles incoming requests and interacts with other components. Together with the Message Barrier it sets the correct migration state and subscription states for every migration. In case of an error, it also performs the rollback process.

5.2.2 Migration Broker Bridge

The Broker Migration Broker Bridge is a communication interface between a broker implementation and the migration framework. It is an interface that has to be implemented by the specific broker implementation, e.g. Moquette Migration Broker Bridge, to perform common functions, such that the migration framework does not need to know the details of a specific broker implementation.

[2]Moquette MQTT broker: `https://moquette-io.github.io/moquette/`

[3]`https://gitlab.com/manuelgeier-masterthesis/moquette/tree/migration`

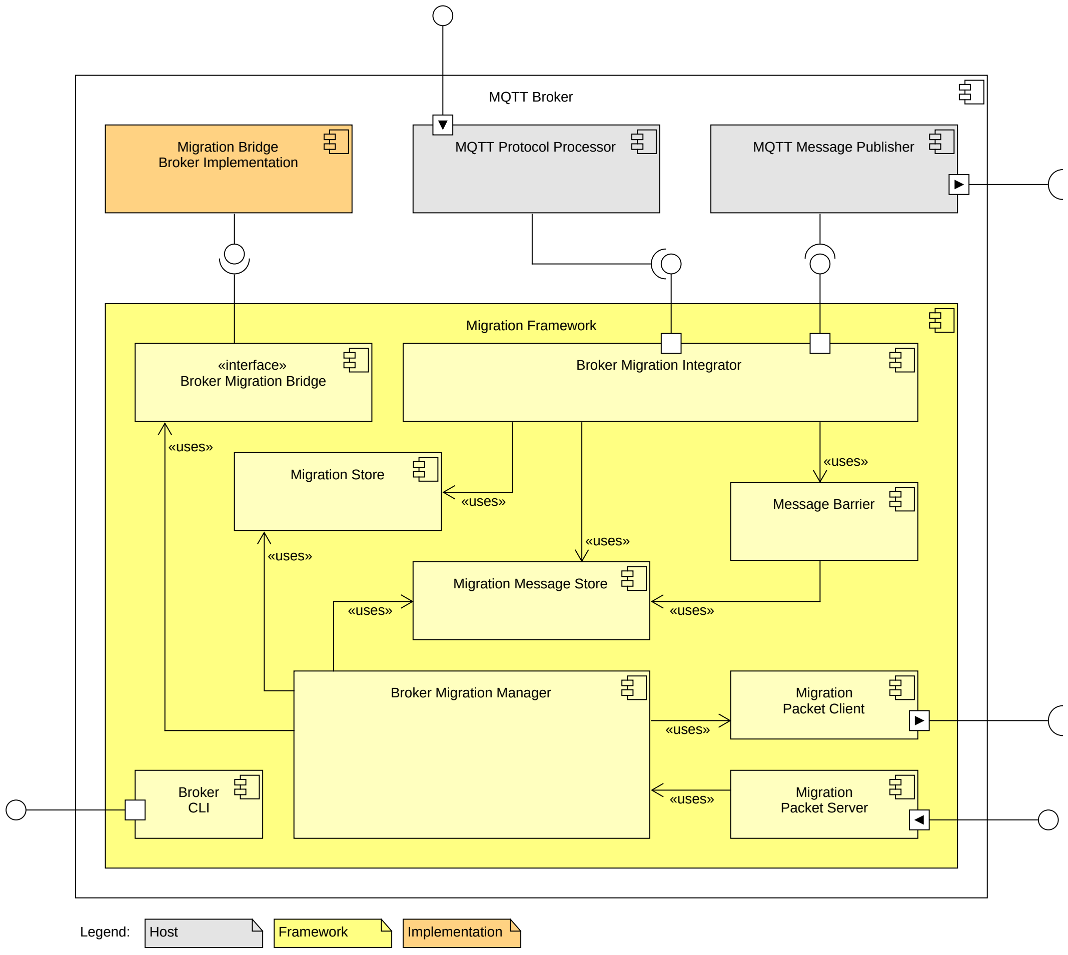


Figure 5.1: Broker components

Some functions include:

- check if a specific client exists
- retrieve client information
- subscribe/unsubscribe a client from a topic
- get current subscriptions of a client
- disconnect a client
- publish messages to a client

5.2.3 Broker Migration Integrator

The Broker Migration Integrator provides an interface to integrate the migration framework into a specific broker implementation, to analyze incoming messages or to integrate the Message Barrier.

5.2.4 Migration Store

The Migration Store holds information of all migration processes that are active. It further puts finished migration processes in an archive to retrieve information of finished migrations.

5.2.5 Migration Message Store

The Migration Message Store stores messages during the migration process to synchronize missing messages with its opponent broker or to send unsent messages in case of a rollback. When a migration process finishes, the messages related to the migration process will be cleared from the store to free up space.

5.2.6 Message Barrier

To assure the message delivery guarantees during the migration process, messages are processed through the Message Barrier. The Message Barrier will decide based on the current migration state, if the current message should be sent or discarded. If a message will be discarded it will further check if the message should to be stored in case of a synchronization or rollback process. If it should be stored, it will place the message in the Migration Message Store for further processing. Furthermore, it updates the subscription states when certain messages passed the Message Barrier.

5.2.7 Broker Command Line Interface (Broker CLI)

Command line interface to perform actions on the broker, e.g. to migrate a client from this broker to another broker. All commands, except `migrate`, are not mandatory for the migration process, but can be used, e.g. for debugging or testing.

The following commands are available for the Broker CLI:

- `clients`
  Lists all clients with its subscriptions that are currently connected to the broker.

- `migrate TARGET_BROKER_PORT CLIENT_ID [ CLIENT_HOST:localhost [ CLIENT_PORT :1702 ] ]`
  Migrates a client from this broker to another broker (Target Broker).

- `migrations`
  Lists all migrations that are currently on the broker.

- `subadd CLIENT_ID [ TOPIC:topic1 [ QOS:0 ] ]`
  Subscribes a client to a topic with a specific QoS.

- `subrem CLIENT_ID [ TOPIC:topic1 ]`
  Unsubscribes a client from a topic.

The Broker CLI acts as an API for external components and allows the Coordinator, like the Controller in EMMA [RND18], to trigger migration processes for clients in an orchestrated components composition.

5.2.8 Source Broker Process

The following section describes and visualizes the implementation of the whole migration process on the Source Broker.

Process Description

- On migration trigger, the Source Broker performs some validation (e.g., does the client with the requested id exist) and then connects to the Target Broker to send migration protocol packets. All active subscriptions for the client are collected and a new migration object with state INITIALIZED is created, $mig_{state} := INITIALIZED$, and stored in the Migration Store. For each subscription a subscription object with the state INITIALIZED, $sub_{state} := INITIALIZED$, is created (implicitly the barrier will be active) and the last processed message id (synchronization state) is retrieved and stored. A subscriptions-finished callback is registered, that will be called by the time the state of all subscriptions is FINISHED, $sub_{state} = FINISHED$, that sets the migration state to FINISHED as well, $mig_{state} := FINISHED$ (if there are no subscriptions, it will never get triggered). For each subscription the initialization event is processed differently depending on the subscription QoS, namely:
  - **QoS 0**: Through the message barrier, the broker stops sending messages. The subscription migration state is immediately set to SYNCED, $sub^0_{state} := SYNCED$, since no synchronization is necessary.
  - **QoS 1**: Through the message barrier, the broker still send all messages to the Client.

  - **QoS 2**: Through the message barrier, the broker stops sending messages. The unsent messages are stored, to be able to send them in case of a message synchronization or a migration error.

  After initialization, a MIGRATE packet is sent to the Target broker.

- On receive of a MIGSYNC packet, the migration state is set to SYNCING, $mig_{state} := SYNCING$. Each subscription that was passed within the MIGSYNC is processed (see Section 4.8 for details about the synchronization). If all requested subscriptions are synced, $sub_{state} = SYNCED$, the migration state is set to SYNCED, $mig_{state} := SYNCED$.

- On migration state is SYNCED, $mig_{state} = SYNCED$, the Source Broker sends a MIGSYNCACK packet to the Target broker, to acknowledge that the synchronization is complete.

- On receive of a MIGACK [OK] packet, no messages are stored anymore. The subscriptions are processed:

  - **QoS 0**: The subscriptions state is set to FINISHED.
  - **QoS 1**: The subscriptions state is set to FINISHED, regardless whether it was synced or not. The barrier will make sure, that no messages are sent anymore.
  - **QoS 2**: The subscriptions state is set to FINISHED, regardless whether it was synced or not.

  $$sub_{state}^{\{1,2,3\}} := FINISHED$$

  If there are subscriptions, the subscriptions-finished callback is called automatically and sets the migration state to FINISHED.

  $$mig_{state} := FINISHED$$

  If there are no subscriptions, the subscriptions-finished callback will never be fired and therefore it sets the migration state to FINISHED manually.

  $$mig_{state} := FINISHED$$

- On migration state is FINISHED, $mig_{state} = FINISHED$, the Source Broker makes sure, that all messages that must be sent are sent, it unsubscribes from all subscriptions, it disconnects from the Client and the migration is cleaned up (releasing stored messages, remove migration object from the Migration Store). The migration process is finished.

Error Handling

There are many cases in which an error could occur. On any error, a rollback is performed, to restore the state before the migration process (as if there was no migration process started at all[4]).

The following describes some error scenarios on the Source Broker:

- The client does not exist in this broker. For example when the client exists while the Coordinator triggers the migration.
- There is an active migration process of the client.
- Timeout occurs for sending packets like MIGRATE or MIGSYNCACK, e.g. due to network errors.
- In case the Target Broker is not able to migrate a Client, it can respond with MIGACK including an error status code (MIGACK [ERROR] packet).
- The migration process takes too long, idles or times out.
  Solutions for this might be (but are not part of the migration process yet):
  - Sending or requesting heartbeat packages between the brokers.

On error, the Source Broker triggers the following rollback process:

- Depending on the QoS of a subscription the following process is performed:
  - **QoS 0**: The subscriptions state is set to ERROR. It continues sending messages. Lost messages since the beginning of the migration are valid for this QoS.
  - **QoS 1**: The subscriptions state is just set to ERROR. Since it did not stop sending messages, no messages are lost. No further rollback actions are necessary.
  - **QoS 2**: The subscriptions state is set to ERROR. It stops storing message in the Message Store and it continues sending messages. Messages that have not been sent to the client since the beginning of the migration, were stored in the Message Store. These messages are sent as well.

  $$sub_{state}^{\{1,2,3\}} := ERROR$$

  Finally, the migration state is set to ERROR.

  $$mig_{state} := ERROR$$

  The migration is cleaned up (releasing stored messages, remove migration object from the Migration Store). The migration is finished.

[4]Depending on the error, the Source Broker might repeat the request some time later

Migration State

The internal state flow of a migration on the Source Broker is shown in Figure 5.2.

Subscription States

The internal synchronization state flow of the subscriptions regarding to its QoS are shown in Figure 5.3 and Figure 5.4. Al possible synchronization states of a subscription object during the migration process on the Source Broker are listed in Table 5.1. Note: Depending on the QoS of the subscription, not all subscription states are in use.

Message Barriers

The Message Barriers for the corresponding QoS of a subscriptions on the Source Broker are described in Table 5.2, Table 5.3 and Table 5.4. Note that messages with a lower or equal id as the last processed message, always have to be sent, since messages before the synchronization point should not be effected by the barrier:

$$send(msg) \mid msg_{id} \leq sub_{lastProcId}^{source}$$

Subscription state	**Description**	**QoS 0**	**QoS 1**	**QoS 2**
INITIALIZED	The subscription synchronization is initialized.	✓	✓	✓
SYNCING	The subscription is currently synchronizing.		✓	✓
SYNCED	The subscription got synchronized.	✓	✓	✓
FINISHED	The subscription synchronization finished successfully.	✓	✓	✓
ERROR	The subscription synchronization got aborted due an error.	✓	✓	✓

Table 5.1: Subscription synchronization states on the Source Broker.

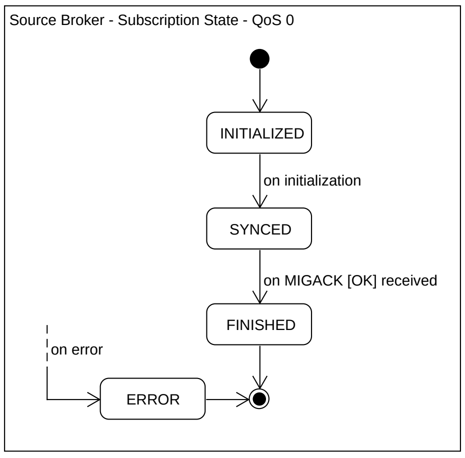


Figure 5.3: State diagram of a QoS 0 subscription on the Source Broker

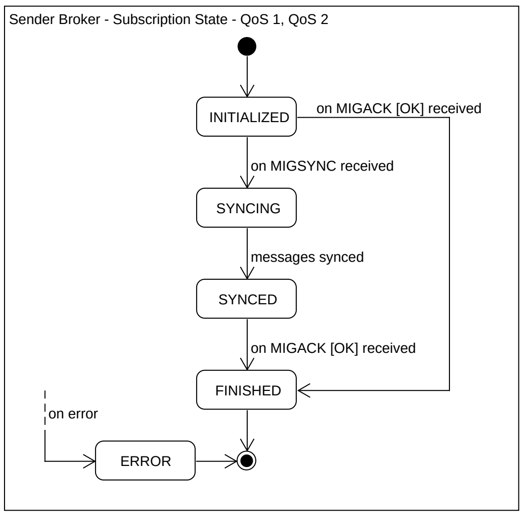


Figure 5.4: State diagram of a QoS 1 and a QoS 2 subscription on the Source Broker

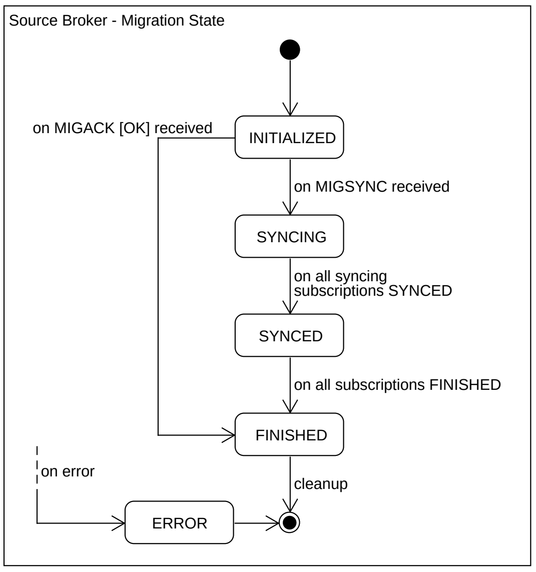

Source Broker - Migration State
on MIGACK [OK] received
INITIALIZED
on MIGSYNC received
SYNCING
on all syncing subscriptions SYNCED
SYNCED
on all subscriptions FINISHED
FINISHED
on error
cleanup
ERROR

Figure 5.2: State diagram of a migration on the Source Broker

Table 5.2: Barrier actions on the Source Broker for a QoS 0 subscription

Subscription state	**Action**
INITIALIZED	Discard message. $discard(msg)$
SYNCED	Discard message. $discard(msg)$
FINISHED	Discard message. $discard(msg)$
ERROR	Send message. $send(msg)$

Table 5.3: Barrier actions on the Source Broker for a QoS 1 subscription

Subscription state	**Action**
INITIALIZED	Send message. $send(msg)$
SYNCING	Send message. $send(msg)$ If the message id is greater or equal to the last processed message id from the Target Broker, the subscription state is set to SYNCED. $$sub_{state} := SYNCED \mid msg_{id} \geq sub^{target}_{lastProcId}$$
SYNCED	Send message. $send(msg)$
FINISHED	Discard message. $discard(msg)$
ERROR	Send message. $send(msg)$

Table 5.4: Barrier actions on the Source Broker for a QoS 2 subscription

Subscription state	**Action**
INITIALIZED	Discard Message $discard(msg)$
SYNCING	If the global message id is greater to the last processed message id from the Source Broker and less or equal to the last processed message id from the Target Broker, the message will be sent. $$send(msg) \mid sub^{source}_{lastProcId} < msg_{id} \leq sub^{target}_{lastProcId}$$ If the message id is equal to the last processed message id from the Target Broker, the subscription state is set to SYNCED. If the message id is greater than the last processed message id from the Target Broker, the message will be discarded and the state will be set to SYNCED as well. $$discard(msg) \mid msg_{id} > sub^{target}_{lastProcId}$$ $$sub_{state} := SYNCED \mid msg_{id} \geq sub^{target}_{lastProcId}$$
SYNCED	Discard message. $discard(msg)$
FINISHED	Discard message. $discard(msg)$
ERROR	Send message. $send(msg)$

5.2.9 Target Broker Process

The following section describes and visualizes the implementation of the whole migration process on the Target Broker.

Process Description

- On receive of a MIGRATE packet from another broker with information about the Client and its subscriptions, the migration process will be initialized. A new migration object with state INITIALIZED is created, $mig_{state} := INITIALIZED$, and stored in the Migration Store. For each subscription a subscription object with the state INITIALIZED, $sub_{state} := INITIALIZED$, is created (implicitly the barrier will be active). It connects to Client to be able to send Migration packets. A subscriptions-synced callback is registered, that will be called by the time the state of all subscriptions is SYNCED, $sub_{state} = SYNCED$, that sets the migration state to SYNCED as well, $mig_{state} := SYNCED$ (if there are no subscriptions, it will never get triggered). A subscriptions-finished callback is registered, that will be called by the time the state of all subscriptions is FINISHED, $sub_{state} = FINISHED$, that sets the migration state to FINISHED as well, $mig_{state} := FINISHED$ (if there are no subscriptions, it will never get triggered). After initialization, a MIGTO packet is sent to the Client, with host information (host, port) of the Target Broker and the migration id. On successful connect, the Client will respond with MIGTOACK packet including the same migration id.

- On receive of a MIGTOACK packet, the Client successfully connected to this broker (Target Broker). The Client is subscribed to its subscriptions that were received within the MIGRATE packet from the Source Broker. For each subscription, the last processed message id (synchronization state) is retrieved and stored and the subscription state is set to SYNCING, $sub_{state} := SYNCING$. The migration state is set to SYNCING as well, $mig_{state} := SYNCING$. For each subscription, the synchronization process as described in Section 4.9 is started. If there are any subscriptions that need to by synchronized with the Source Broker, a MIGSYNC packet is sent to the Source Broker, containing the subscriptions that need to be synchronized. If there are no subscriptions, the migration state is set to SYNCED, $mig_{state} := SYNCED$.

  It is important to note that after setting the subscription state for QoS 1 and QoS 2 subscriptions to SYNCING, $sub^{\{1,2\}}_{state} = SYNCING$, depending on the synchronization state the subscription states can have intermediate states:

  - **ahead**: Synchronization with the Source Broker is necessary. Therefore, the subscription state is set to SYNCING_OPPONENT,
    $sub^{\{1,2\}}_{state} := SYNCING\_OPPONENT$, meaning that the broker will wait for

  a MIGSYNCACK packet to set the subscription state to SYNCED, $sub_{state}^{\{1,2\}} := SYNCED$.
  - **synced**: No synchronization with the Source Broker is necessary and no messages have to be skipped. Therefore, the subscription state can immediately be set to SYNCED, $sub_{state}^{\{1,2\}} := SYNCED$.
  - **behind**: No synchronization with the Source Broker is necessary, but the broker has to skip some messages. Therefore, the subscription state is set to SYNCING_THIS, $sub_{state}^{\{1,2\}} := SYNCING\_THIS$, meaning that the broker will wait, until it received the right messages to be synchronized, $sub_{state}^{\{1,2\}} := SYNCED$.

- On receive of a MIGSYNCACK packet, all QoS 1 and QoS 2 subscriptions that had to be synced, are set to SYNCED, $sub_{state}^{\{1,2\}} = SYNCED$.
- On migration state is SYNCED, $mig_{state} = SYNCED$, the Target Broker sends a MIGACK packet to the Source Broker to acknowledge a successful migration.
- After the MIGACK packet got sent, it finalizes the migration process and sets the subscription states to FINISHED, $sub_{state}^{\{1,2,3\}} := FINISHED$. The subscriptions-finished callback will be triggered and the migration state is set to FINISHED as well, $mig_{state} := FINISHED$. If there are no subscriptions, the callback will not be triggered and therefore we manually set the migration state to FINISHED, $mig_{state} := FINISHED$.
- On migration state is FINISHED, $mig_{state} = FINISHED$, the migration process gets finished by closing the connection to the Source Broker and removing the migration from the migration store. The migration process is finished.

Error Handling

There are many cases in which an error could occur. On any error, the migration process is aborted and a MIGACK packet with the status code of ERROR is sent to the Source Broker.

The following describes some error scenarios on the Target Broker:

- The requested subscriptions are not available on the Target Broker, therefore it can not start the synchronization process.
  Solutions for this might be (but are not part of the migration process yet):
  - The Target Broker can also try to request the missing subscriptions during the initialization phase. Nevertheless, this will postpone the migration process until all subscriptions are available.

  - The Target Broker also might ask the Source Broker to try some time later, while it builds up the missing subscriptions.
- There is an active migration process of the client.
- The client is not authorized to connect to this broker.
- The internal load might be too high already.
- The connection with the Client cannot be established.
- It does not receive MIGTOACK or MIGSYNCACK packet after some period of time.
- Timeouts for sending packets like MIGTO or MIGSYNC occur.

On error, the Target Broker triggers the following rollback process:

- On error, the subscriptions states are set to ERROR and no messages are sent anymore.
  $$sub_{state}^{\{1,2,3\}} := ERROR$$
  Finally, the migration state is set to ERROR.
  $$mig_{state} := ERROR$$
- The Client will be unsubscribed from all subscriptions and unregistered from the Target Broker.
- It closes the connection to the Client, if there is an active connection.
- A MIGACK [ERROR] packet is sent to the Source Broker.
- The migration is cleaned up (remove migration object from the Migration Store) and the migration is finished.

Migration State

The internal state flow of a migration on the Target Broker is shown in Figure 5.5.

Subscription States

The internal synchronization state flow of the subscriptions regarding to its QoS are shown in Figure 5.6 and Figure 5.7. Al possible synchronization states of a subscription object during the migration process on the Source Broker are listed in Table 5.5. Note: Depending on the QoS of the subscription, not all subscription states are in use.

Message Barriers

The Message Barriers for the corresponding QoS of a subscriptions on the Target Broker are described in Table 5.6, Table 5.7 and Table 5.8. Note that messages with a lower or equal id as the last processed message, always have to be discarded, since messages before the synchronization point should not be effected by the barrier:

$$discard(msg) \mid msg_{id} \leq sub_{lastProcId}^{target}$$

Subscription state	**Description**	**QoS 0**	**QoS 1**	**QoS 2**
INITIALIZED	The subscription synchronization is initialized.	✓	✓	✓
SYNCING	The subscription is currently synchronizing.	✓	✓	✓
SYNCING_THIS	The Target Broker waits until itself is synced.		✓	✓
SYNCING_OPPONENT	The Target Broker waits until the Source Broker is synced missing messages.		✓	✓
SYNCED	The subscription got synchronized.	✓	✓	✓
FINISHED	The subscription synchronization finished successfully.	✓	✓	✓
ERROR	The subscription synchronization got aborted due an error.	✓	✓	✓

Table 5.5: Subscription synchronization states on the Target Broker.

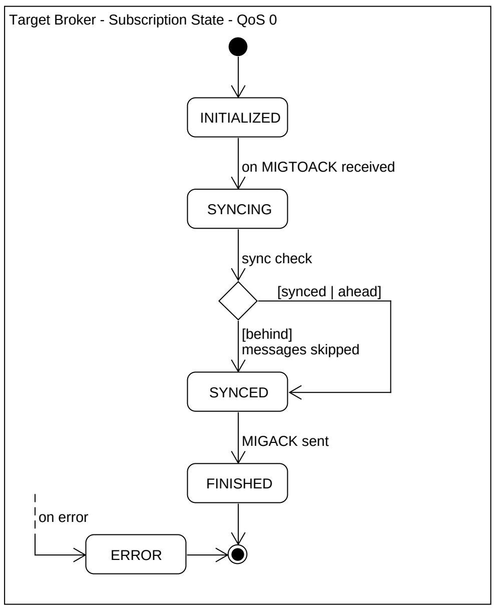


Figure 5.6: State diagram of a QoS 0 subscription on the Target Broker

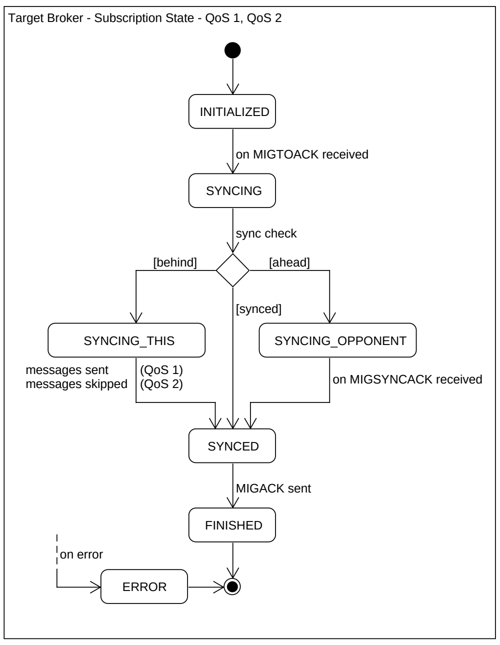


Figure 5.7: State diagram of a QoS 1 and a QoS 2 subscription on the Target Broker

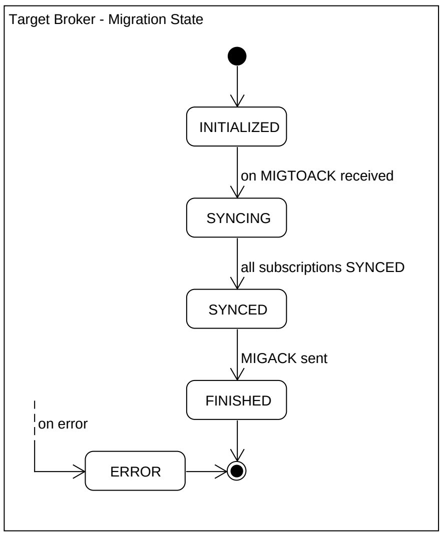


Figure 5.5: State diagram of a migration on the Target Broker

Table 5.6: Barrier actions on the Target Broker for a QoS 0 subscription

Subscription state	**Action**
INITIALIZED	Discard message. $discard(msg)$
SYNCING	If the message id is less or equal to the last processed message id from the Source Broker, the message is discarded. $$discard(msg) \mid msg_{id} \leq sub^{source}_{lastProcId}$$ If the message id is equal to the last processed message id from the Source Broker, the subscription state is set to SYNCED. If the message id is beyond the the last processed message id from the Source Broker, the message is sent and the state is set to SYNCED as well. $$send(msg) \mid msg_{id} > sub^{source}_{lastProcId}$$ $$sub_{state} := SYNCED \mid msg_{id} \geq sub^{source}_{lastProcId}$$
SYNCED	Send message. $send(msg)$
FINISHED	Send message. $send(msg)$
ERROR	Discard message. $discard(msg)$

Table 5.7: Barrier actions on the Target Broker for a QoS 1 subscription

Subscription state	**Action**
INITIALIZED	Discard message. $discard(msg)$
SYNCING	Send message. $send(msg)$ If the message id is the last processed message id from the Source Broker, the subscription state is set to SYNCED. If the message id is greater than the last processed message id from the Source Broker (in case of a missing last message, caused by a gap), then the state will be set to SYNCED as well. $$sub_{state} := SYNCED \mid msg_{id} \geq sub^{source}_{lastProcId}$$
SYNCED	Send message. $send(msg)$
FINALIZED	Send message. $send(msg)$
FINISHED	Send message. $send(msg)$
ERROR	Discard message. $discard(msg)$

Table 5.8: Barrier actions on the Target Broker for a QoS 2 subscription

Subscription state	**Action**
INITIALIZED	Discard message. $discard(msg)$
SYNCING, SYNCING_THIS	Discard the message if the message id is less or equal to the last processed message id from the Source Broker. $$discard(msg) \mid msg_{id} \leq sub^{source}_{lastProcId}$$ Otherwise, we send it. $$send(msg) \mid msg_{id} > sub^{source}_{lastProcId}$$ If the message id is equal or greater to the last processed message id of the Source Broker, we update the state. If the state is SYNCING_THIS, the state is set to SYNCED, otherwise set the state to SYNCING_OPPONENT. $$sub_{state} := SYNCED \mid msg_{id} \geq sub^{source}_{lastProcId} \text{ and } sub_{state} = SYNCING\_THIS$$ $$sub_{state} := SYNCING\_OPPONENT \mid msg_{id} \geq sub^{source}_{lastProcId} \text{ and } sub_{state} = SYNCING$$
SYNCING_OPPONENT	Send message. $send(msg)$
SYNCED	Send message. $send(msg)$
FINISHED	Send message. $send(msg)$
ERROR	Discard message. $discard(msg)$

5.3 MQTT Client

To enable mobility and message delivery guarantees to migrate MQTT clients we introduce the following components in our migration framework. A component overview for the client components is shown in Figure 5.8. To perform the migration process, the client has to be able to communicate with multiple brokers and has to integrate the migration framework.

We implemented a `MqttMultiBrokerClient` that uses multiple individual MQTT Clients and combines them into one. As the specific, individual MQTT Client we used the Java MQTT Client of Fusesource[5]. The project and its source code is available in our project repository[6].

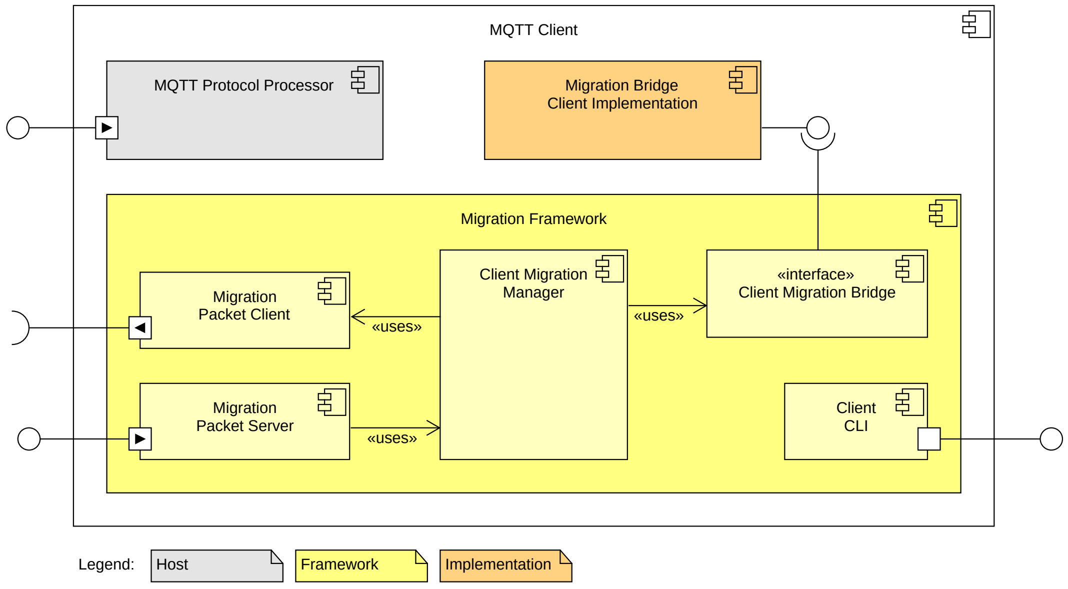


Figure 5.8: Client components

5.3.1 Client Migration Manager

The Client Migration Manager is the central component on the client for the migration process. It handles incoming requests and interacts with other components in order to perform the migration process.

[5]https://github.com/fusesource/mqtt-client
[6]https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration-clients/blob/master/src/main/java/io/geier/diplomathesis/experiment/client/MqttMultiBrokerClient.java

5.3.2 Client Migration Bridge

The Client Migration Broker Bridge is a communication interface between a client implementation and the migration framework. It is an interface that has to be implemented by the specific client implementation, to perform common functions, such that the migration framework does not need to know the details of a specific client implementation.

Some functions include:

- establish a connection with a broker (Target Broker)
- retrieve information about established broker connections

5.3.3 Command Line Interface (Client CLI)

The Client CLI provides an interface to perform commands on the client. All commands are not mandatory for the migration process, but can be used, e.g. for debugging or testing.

The following commands are available for the Client CLI:

- `brokers`
  Lists all brokers with its host information that the client is currently connected to.

5.3.4 Client Process

The following section describes the migration process on the Client. Since we aimed to create a very thin interaction layer on the client side, the client process is very short compared to the brokers processes.

Process Description

- On receive of a MIGTO packet, the Client's migration manager takes the host and port information of the new broker (Target Broker) it should connect to and it tries to establish a connection with the new broker by sending a MQTT CONNECT packet to it.
- On receive of a MQTT CONACK packet from the new broker, after a successful connection it confirms the connection by sending a MIGTOACK packet with the corresponding migration id from the MIGTO packet to the new broker. It does not forward any current subscriptions to the new broker, since these will be exchanged by the brokers itself. From this point on, it has (at least) two active connections to brokers and it might receive messages from the Source Broker and the Target Broker until the migration process is finished.

- On receive of a MQTT DISCONNECT packet from the Source Broker, the migration process is finished. From this point on, it will only receive messages from the Target Broker onwards. It cleans up its connection and broker information and continues normally. The Client must not reconnect to the disconnected broker, as long as there is another active connection to a broker, i.e. the Target Broker. Otherwise, if there would be two uncoordinated active connections, the Client would get messages from both brokers.

5.4 Communication

The communication for the migration process between the broker and client migration framework instances throughout the network is based on the Migration Protocol described in Section 4.6. Every migration framework instance has a running Migration Server (see Section 5.4.1) and can communicate with others migration framework instances by using a Migration Client (see Section 5.4.2) that connects to the Migration Server.

The packet format of the migration protocol is for the sake of simplicity a simple string and further protocol optimizations are left for future work. Within this string, the migration packet object is encoded in JSON format[7]. The Jackson model mapper[8] is used to convert migration packets from Java objects to JSON and back.

5.4.1 Migration Packet Server

The Migration Message Server handles incoming Migration Packets and communicates with Migration Clients. Incoming requests are forwarded to the Migration Manager to be processed.

It is implemented with the Netty[9] framework. Netty, as stated on the official website, is a “NIO client server framework, an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers and clients”. The channel setup of Netty on the migration server to decode and encode messages is shown in Listing 8.1.

5.4.2 Migration Packet Client

The Migration Client connects to a Migration Packet Server and communicates by sending Migration Packets. Together with the Migration Server it is the communication channel between the Source Broker, Target Broker and the Client throughout the network.

[7]https://www.json.org/
[8]http://modelmapper.org/user-manual/jackson-integration/
[9]https://netty.io/

5.5 Migration Protocol Packets

The following describes the Migration Protocol with its implementation details.

Property	Name of the property.
Type	Java Type of the property. Primitive types like String, int, etc. or custom Classes, Enumeration, etc. that are be described in more detail in this section.
Description	Description and further details of the property.

5.5.1 MIGRATE packet

Packet send from Source Broker to Target Broker to initiate a new migration of a Client.

`MigratePacket` class

Property name	Type	Description
migrationId	String	Unique Migration Id that is used to coordinate a client migration.
clientId	String	Unique id of the Client that should be migrated.
clientHost	String	Host address of the Client that should be migrated.
clientPort	int	Port of the Packet Client for the migration of the Client that should be migrated.
subscriptions	Set<MigrateSubscription>	Set of subscriptions that the Client is subscribed to and that will be migrated, with additional information for the migration process.

`MigrateSubscription` class

Property name	Type	Description
requestedQos	MqttQoS	Requested QoS for the subscription.
topoicName	String	Topic the Client has subscribed to.
lastProcessedGlobalMessageId	GlobalMessageId	Global message id of the last message that was processed by the Source Broker.

`GlobalMessageId` class

Property name	Type	Description
globalMessageId	String	Unique global id of a message.

`MqttQoS` enumeration

Property name	int Value	Description
AT_MOST_ONCE	0	QoS 0 - At most once.
AT_LEAST_ONCE	1	QoS 1 - At least once.
EXACTLY_ONCE	2	QoS 2 - Exactly once.
FAILURE	128	In case of error.

5.5.2 MIGACK packet

Packet send from Target Broker to Source Broker to acknowledge a migration of a Client.

`MigAckPacket` class

Property name	Type	Description
migrationId	String	Unique Migration Id that is used to acknowledge a migration.
statusCode	MigStatus	Status code of the migration. It indicates, if it was successful or failed.

`MigStatus` enumeration

Property name	int Value	Description
OK	0	Successful migration.
ERROR	1	Migration error occurred.

5.5.3 MIGSYNC packet

Packet send from Target Broker to Source Broker to indicate that the Target Broker can not fully serve the Client with all messages and that the Source Broker needs to serve missing messages.

`MigSyncPacket` class

Property name	Type	Description
migration id	String	Unique Migration Id that is used to coordinate a client migration.
subscriptions	List<MigSyncSubscription>	List of subscriptions that the Source Broker needs to synchronize.

`MigSyncSubscription` class

Property name	Type	Description
topicName	String	Name of the topic that the Source Broker needs to send some messages that were missed by the Target Broker.
lastProcessedGlobalMessageId	GlobalMessageId	Global message id of the last message that was processed by the Target Broker.

5.5.4 MIGSYNCACK packet

Packet send from Source Broker to Target Broker to acknowledge that all missing messages were successfully send to the Client.

`MigSyncPacket` class

Property name	Type	Description
migrationId	String	Unique Migration Id that is used to acknowledge a message synchronization.

5.5.5 MIGTO packet

Packet send from Target Broker to a Client to indicate that it should connect to a new broker (Target Broker) in order to get migrated.

`MigToPacket` class

Property name	Type	Description
migrationId	String	Unique Migration Id that is used to coordinate the migration.
host	String	Host address of the new broker.
port	int	MQTT port of the new broker.

5.5.6 MIGTOACK packet

Packet send from Client to Target Broker to acknowledge a successful connection to the new broker (Target Broker) and that it is ready to be migrated.

`MigToAck` class

Property name	Type	Description
migrationId	String	Unique Migration Id that is used to acknowledge a successful connect to the new broker.

CHAPTER 6

Evaluation

To evaluate the feasibility of our solution approach, we study for our evaluation the three aspects, *correctness*, *responsiveness* and *system strains*, with the methods for software testing, empirical experiments and a theoretical analysis. In the following section we give an introduction to our evaluation methodology and lay out the structure of this chapter. In Section 6.5, we summarize and discuss our results and findings and point out gains and limitations of our solution approach.

6.1 Methodology

The following describes our methodology for our evaluation. It describes the aspects that we chose to evaluate our solution approach and the methods that we are using to give answers to all aspects.

We chose the following aspects to evaluate our solution approach:

- **Correctness**: It describes the validity of our solution approach and tests if our migration process works as expected.
- **Responsiveness**: It describes the time and load dimensions of our solution approach. The time it takes for the migration do complete, how long a synchronization process for each QoS last and how much the Client and network is affected.
- **System Strains**: They describe the side-effects our solution approach has on the systems and the network.

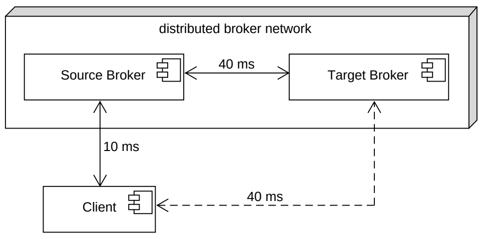
To analyse and provide answers to our evaluation aspects we use the following methods:

- **Software Testing**: To show the *correctness* and feasibility of our solution approach and to verify our implementation , we are using traditional testing methods (see Section 6.2). With unit tests we are verifying individual components and with integration tests we are checking the integration and interaction between different components. Finally, with system tests, we are checking the whole solution approach and implementation to be valid. The system tests are based on the migration process definitions from Section 4.9 and therefore also check the feasibility of our solution approach.
- **Empirical Experiments**: To further check the *correctness* and get insights of the *responsiveness* and the *system strains*, we set up a experiment environment to our solution approach as an integration into real broker and client systems (see Section 6.3). These systems run in a dockerized environment and perform a common interaction scenario (see Section 6.1.1), with different experiment configurations.
- **Theoretical Analysis**: To get additional insights of the *responsiveness* and the *system strains*, we create simplified mathematical model of our solution approach (Section 6.4). With this model we approximate the migration process duration (Section 6.4.3), how many message are lost during the migration process for QoS 0 (Section 6.4.4), how many message are duplicated for QoS 1 (Section 6.4.4) and how many messages need to be stored for QoS 2 (Section 6.4.4). Furthermore, we derive the impact on the Client for message arrival (Section 6.4.5) and the impact on the network traffic with our solution approach (Section 6.4.7, Section 6.4.8).

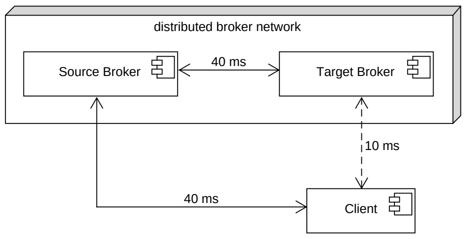
6.1.1 Evaluation Scenario

For our evaluation we use a common scenario where a client gets migrated from one broker to another broker. In the following we describe this evaluation scenario, its design and its steps.

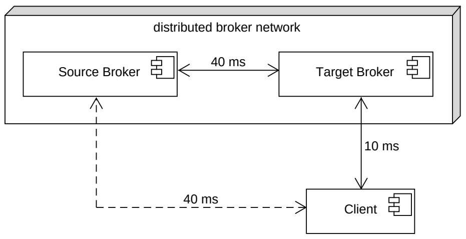
The system composition of the scenario consists of two brokers, the Source Broker and the Target Broker, two Publishers, each for every broker, and a Subscriber, that represents the Client that will be migrated from one broker to another. All systems are within a shared network environment without external traffic. The evaluation scenario is visualized in Figure 6.1. Solid links define active communication links and dashed lines define potential communication links that are not active. The label on the links states the latency for one direction for a specific link in milliseconds. Links between publishers and brokers (not shown in the figures) have no delay. As in [LCC$^+$15], we also consider a message producer that continuously generates messages with a given interval on the publishers. We chose the latencies for the links similarly but with half of the proximity to [RND18], with a link latency of 40 ms between brokers and a link latency with the client in close proximity of 10 ms and with far proximity of 40 ms.



(a) Before Client movement



(b) After Client movement/Before migration



(c) After migration

Figure 6.1: Evaluation Scenario

The sequence of the scenario is as follows:

1. The initial situation is illustrated in Figure 6.1a with the basic setup of the network and the individual systems.

2. Caused by the mobility of the client, the client moves away from the Source Broker and closer to the Target Broker. Therefore the network conditions (links latencies) changes. The network conditions after the movement is illustrated in Figure 6.1b. This is the base of our network setup, on which the migration will be applied.

3. We wait some time and keep measurements of the current traffic of the network between all systems.

4. At some point, we assume that the network change is recognized and that the Coordinator decides, to migrate the Client from the Source Broker to the Target Broker, since the link latency is smaller. The migration process is triggered at the Source Broker and starts.

5. After a while the migration process will be completed and the Target Broker will have an established communication link with the Client. The network scenario after the migration is illustrated in Figure 6.1c. The scenario is finished.

6.2 Software Testing

To verify the correctness of our implementation and moreover of our solution approach, we use softwared testing methods.

Testing of software implementations is usually approached with unit and integration tests. We used these approaches to verify that our components work as expected. 139 tests unit and integration tests were written with support of the testing framework *JUnit 4*[1] and the mocking framework *Mockito 2*[2]. These tests are separated into 40 Java classes, sum up to 7k lines of code and are available in our project repository[3].

We further created system tests with different scenarios[4], that run as either the Source Broker or the Target Broker. These tests expect the systems to handle different events from outside and to change its internal state correctly. Therefore, a system test framework was written, to challenge the migration process with different events. Our system test framework is capable to trigger the following events:

- start the migration process to move a client
- retrieve different migration packets from other simulated brokers
- retrieve different MQTT PUBLISH packets from simulated publishers

While performing the system tests the internal state was asserted and verified. Any aberrance would have lead to an unsuccessful test run. The system test framework is also based on the testing framework JUnit and the mocking framework Mockito. A simple example of a system test scenario for the Source Broker with a QoS 0 subscription, where the Source Broker and Target Broker are synchronized, is shown in Listing 8.2.

As a base for our system test scenarios, the synchronization scenarios from Section 4.8 are used, such that our solution approach is validated to be correct. Additionally, more system test scenarios, like missing messages (gaps), were performed to verify the correct behavior in abnormal cases. Altogether, 94 system test scenarios were written and successfully performed.

In total, 233 unit, integration and system tests are written and successfully performed to validate the correctness of our implementation and the feasibility of our solution approach.

[1] `https://junit.org/`

[2] `https://site.mockito.org/`

[3] `https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration/tree/master/src/test/java/io/geier/diplomathesis/migration`

[4] `https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration/tree/master/src/test/java/io/geier/diplomathesis/migration/simulation`

6.3 Empirical Experiments

To further test and empirically evaluate our implementation and the integration into the *Moquette* broker system, we created an experiment environment. The experiment environment consists of different experiment configurations with real broker, publisher and subscriber systems that were put into Docker containers, deployed on a virtual machine and orchestrated with Docker Compose. As stated in [IMA$^+$16], “Docker provides fast deployment, elasticity and good performance over VM based EC platform” and was therefore also our choice for the empirical experiments. In the following sections we are explain our experiment setup (Section 6.3.1), give an overview of our experiment configurations (Section 6.3.2), showcase some of our results in detail (Section 6.3.3) and summarize our results (Section 6.3.4). The whole empirical experiment with our results is available in our project repository[5].

6.3.1 Experiments Setup

All of our empirical experiments run in a virtual environment setup in a virtual machine. The virtual machine for our experiments uses 4 cores and 8 GB RAM and runs on a physical machine with Windows 10 Pro system, an Intel Core i7-4700MQ CPU @ 2.40GHz 2.39 GHz processor and 16 GB RAM. On the Windows machine, a Docker Machine[6] is installed, that provides the Docker environment to run our systems (two brokers, two publishers and a subscriber). We specifically use Boot2Docker[7] which provides a fully, easy-to-use, unix-based and dockerized environment on a Windows machine.

We use Docker[8] technology to manage the systems in the virtual environment. We created a docker image that contains all executables of our systems (MQTT Broker, MQTT Client, Publisher) and is reused through out our setup. During experiment runtime, each individual system runs in its own Docker container and is identified with a unique name, e.g. Broker_001, Broker_002, Publisher_001, Publisher_002, or Subscriber_001.

To setup the network conditions for our experiment within our virtual environment, we use a tool called `pumba` [9]. It is a “chaos testing and network emulation tool for Docker” and allows us to work with the containers (pause, stop and kill) and to emulate different network settings. We use it to delay egress traffic between our containers in order to simulate the evaluation scenario conditions (Section 6.1.1). We were not able to fully setup the link latencies as described in the evaluation scenario, because `pumba` is limited to just set one delay for all outgoing connections per container. Even the direct use of `tc`[10] was not successful, for the same reason. Therefore, we came up with the following

[5] `https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration-experiment`
[6] `https://docs.docker.com/machine/overview/`
[7] `https://github.com/boot2docker/boot2docker`
[8] `https://www.docker.com/`
[9] `https://github.com/alexei-led/pumba`
[10] https://wiki.linuxfoundation.org/networking/netem

latency setup for outgoing connections: Source Broker: 40 ms; Target Broker: 10 ms; Client: 10 ms. To get some variance into the network latencies, we use a jitter value of ±5 ms for each network transaction. All services are composed with Docker Compose[11] to start and stop them at once. Furthermore, all running services are within a common network environment and are able to communicate with each other.

Every experiment, strictly speaking, each experiment configuration, is located in its own folder within the `experiment/` folder. Each experiment is uniquely identified by the name of the folder (`[EXPERIMENT]`). It contains the configuration for publishers and the subscriber. After the experiment has finished, a folder named `[EXPERIMENT]-[RUNNR]-result` is created for each run, containing the initial configuration, logs from brokers, publishers and subscribers and additional measurement files. We run every experiment 10 times to verify the experiment results

To analyze the measurements and data we use R[12], a free software environment for statistical computing and graphics. Various scripts analyze different aspects, like message latency and message trips, of the migration process and visualize with plots (we present these plots in Section 6.3.3).

All experiments also measure information about lost and duplicated messages during the migration process. This gives us proof, if the process was successful or failed. Furthermore, with the support of our Data Integrity Validator (Section 6.3.1), we verify the validity for each subscription on the Client regarding its requested QoS.

Depending on the experiment configuration, a full run of an experiment takes around 2 minutes.

Mocked Distributed Broker Network

Moquette does not work in a distributed broker network setup per se. Since we want to be able to test and evaluate our migration process in a normal functional distributed broker network like it is illustrated in Figure 6.2, where a single message is automatically distributed across the broker network and to get a total ordering of messages, we took a mock approach to generate and send a message multiple times from individual publishers and distribute the generated messages to its corresponding broker, as visualized in Figure 6.3. Therefore, we get a similar functioning distributed broker network to validate our migration process.

Message Publisher

To receive messages on the client side and test our migration process, the Message Publisher generates messages for various registered topics and publishes them into the

[11]`https://docs.docker.com/compose/`

[12]`https://www.r-project.org/`

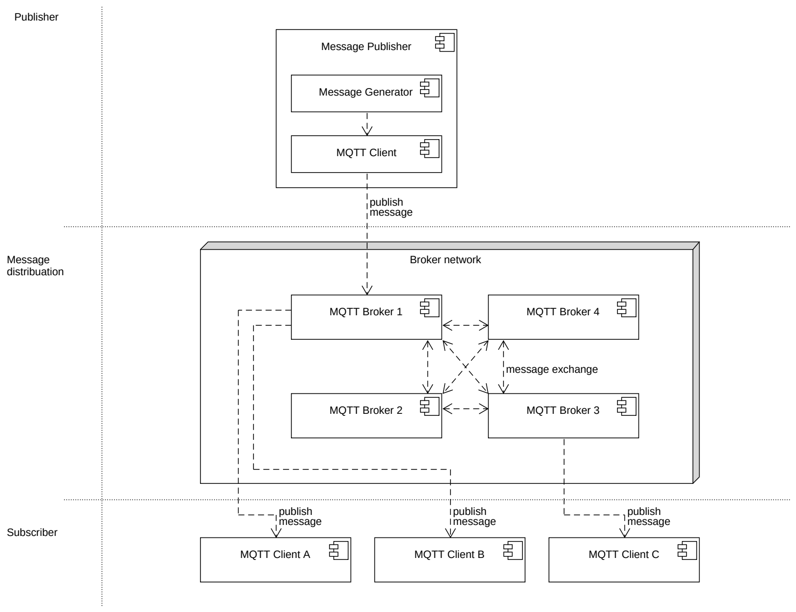


Figure 6.2: Normal distributed broker network

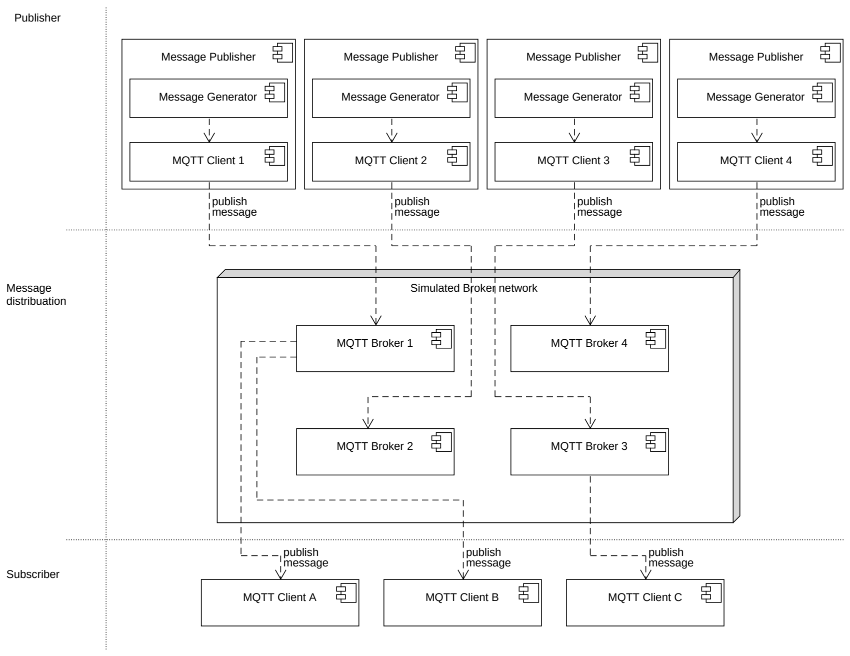


Figure 6.3: Mocked distributed broker network

mocked broker network. The component is available in our project repository[13].

Since the broker network is just mocked and no message distribution between brokers happens per se, we create one publisher for each broker, as shown in Figure 6.3, generate each message on every publisher and publish them individually to each broker. Hence, every broker gets all messages and it seems like all brokers share the same messages.

For our experiments, we configure the generation interval of messages as well as a message delay for each publisher. We use the message delay to simulate different synchronization states between broker systems. Since we assume that messages arrive in order at the brokers, the publisher makes sure that this criteria is met. Furthermore, each message will be published with QoS 2 as assumed by our work.

Data Integrity Verifier

To verify the delivery guarantees agreements, QoS 0, QoS 1 and QoS 2, on the clients for each subscription, we created a Data Integrity Verifier component. It processes all messages a client receives and based on the global message id of the message, it verifies that the integrity of the QoS for each subscription is correct. The component is available in our project repository[14].

The Data Integrity Verifier checks, if there are a) any missing messages, b) any duplicate messages, or c) both, depending on the QoS of the subscription. In case of a data integrity error, e.g. missing messages or duplicate messages, it logs error messages and our experiments report a failure. Table 6.1 shows which checks are performed for each QoS level.

Table 6.1: Data integrity checks for different QoS

QoS	**missing message check**	**duplicate message check**
QoS 0: At most once	No	Yes
QoS 1: At least once	Yes	No
QoS 2: Exactly once	Yes	Yes

For each subscribed topic, a map of all received messages is stored internally, as well as a sorted list of all received global message ids. To check missing messages, the sorted list of all received global message ids is validated and if there is a gap between ids, the missing ids will be logged. To check duplicate messages, every received message has a count how often it was received.

[13] `https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration-clients/blob/master/src/main/java/io/geier/diplomathesis/experiment/client/ClientPublisher.java`

[14] `https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration-clients/blob/master/src/main/java/io/geier/diplomathesis/experiment/verifier/DataIntegrityVerifier.java`

6.3.2 Experiments Configuration Overview

To evaluate the basic scenarios as described in Section 4.8 (behind, synced and ahead with QoS 0, QoS 1 and QoS 2), we created an experiment configuration for each of it. Furthermore, we created more complex scenarios with multiple topics and mixed message generation delays for publishers.

An overview of the configuration of the experiments is given in Table 6.2. Each experiment configuration consists of one or more subscriptions to a topic with a specific QoS for the Client. Each topic is served by the Source Publisher and Target Publisher via the corresponding brokers. The publishers generate and publish all messages with a defined rate interval and a defined message delay to simulate different synchronization states between the broker systems.

6.3.3 Experimental Result

After each experiment run, logs and measurements are processed and data analysis scripts are executed to generate different plots and statistics. We showcase, describe and analyse nine different scenarios to demonstrate and compare the migration process in different scenarios. These scenarios include the case where the Target Broker is behind and where the Target Broker is ahead of the Source Broker for each QoS. Table 6.3 gives an overview of the chosen experiment settings[15]. From each experiment, we present the first of ten runs. All detailed results of the following and other experiments are published in the repository of the experiments[16].

Table 6.3: Experiments Showcase Overview - Topics and characteristics

Experiment	**QoS**	**Synchronization state**
ahead-A0	QoS 0 - at most once	Target Broker ahead of the Source Broker
ahead-A1	QoS 1 - at least once	Target Broker ahead of the Source Broker
ahead-A2	QoS 2 - exactly once	Target Broker ahead of the Source Broker
synced-A0	QoS 0 - at most once	Target Broker is synced with the Source Broker
synced-A1	QoS 1 - at least once	Target Broker is synced with the Source Broker
synced-A2	QoS 2 - exactly once	Target Broker is synced with the Source Broker
behind-A0	QoS 0 - at most once	Target Broker behind of the Source Broker
behind-A1	QoS 1 - at least once	Target Broker behind of the Source Broker
behind-A2	QoS 2 - exactly once	Target Broker behind of the Source Broker

[15] *synced* scenarios: since network and process variances made it hard to get both brokers exactly synced all the time, we tried to come up with a configuration that is as close to the synced state as possible.

[16] `https://gitlab.com/manuelgeier-masterthesis/pubsub-message-qos-migration-experiment/tree/master/experiment`

Experiment name		*Source Publisher*		*Target Publisher*	
Topic name	**QoS**	**Interval** (ms)	**Delay** (ms)	**Interval** (ms)	**Delay** (ms)
ahead-A0					
topicA	QoS 0	100	2000	100	0
ahead-A1					
topicA	QoS 1	100	2000	100	0
ahead-A2					
topicA	QoS 2	100	2000	100	0
ahead-A0__B1__C2					
topicA	QoS 0	100	2000	100	0
topicB	QoS 1	100	2000	100	0
topicC	QoS 2	100	2000	100	0
synced-A0					
topicA	QoS 0	100	0	100	750
synced-A1					
topicA	QoS 1	100	0	100	750
synced-A2					
topicA	QoS 2	100	0	100	750
synced-A0__B1__C2					
topicA	QoS 0	100	0	100	750
topicB	QoS 1	100	0	100	750
topicC	QoS 2	100	0	100	750
behind-A0					
topicA	QoS 0	100	0	100	2000
behind-A1					
topicA	QoS 1	100	0	100	2000
behind-A2					
topicA	QoS 2	100	0	100	2000
behind-A0__B1__C2					
topicA	QoS 0	100	0	100	2000
topicB	QoS 1	100	0	100	2000
topicC	QoS 2	100	0	100	2000
mixed-A0__B1__C2__D0__E1__F2					
topicA	QoS 0	100	2000	100	0
topicB	QoS 1	100	2000	100	0
topicC	QoS 2	100	2000	100	0
topicD	QoS 0	100	0	100	2000
topicE	QoS 1	100	0	100	2000
topicF	QoS 2	100	0	100	2000

Table 6.2: Experiments Configuration Overview

For the following experiment results, publishers generate messages in a 100 ms interval (10 messages/s) and publish them in an ordered message sequence to the corresponding brokers with QoS 2. The synchronization delta for a Target Broker that is ahead or behind the Source Broker is achieved with a delay on the respective publisher.

For each experiment we show the following seven plots on two pages each. Generally, red (squares) are messages from the Source Broker and blue (diamonds) are messages from the Target Broker, or the brokers publisher respectively. All plots are clipped to ± 2 seconds around the migration process.

The first two plots on the first page give a visual representation and verification of the migration process. The top two plots on the second page show the correlation between the arrival time, departure time and sequence number. The middle two plots show the latency in by arrival time and sequence number. The bottom plot shows the empirical cumulative distribution of message latencies.

1. **Migration verification plot**: This plot visualizes all messages by its sequence number. Each symbol represents a produced message (first and fifth row) or a received message (second to fourth row). It is easy to see, if the publication process on the publishers worked well and if all necessary messages generated and sent. The second and fourth row show all messages that were received by the Client from the respective publisher through the broker. The third row combines the second and fourth row to give a complete image of all messages that were received by the Client. It is easy to see, if the QoS criteria for QoS 0, QoS 1 and QoS 2 are fulfilled, by checking duplicates (overlapping symbols) or missing messages (gaps).

2. **Migration process plot** This plot gives a visual representation and provides a good insight on the details of the migration process. It shows all messages by the time they were generated on the publisher (first and last row) to the time they were received at the Client (second and fourth row). The middle row combines all received messages from the second and fourth row and shows the actual load on the Client. The orange arrows connect the corresponding messages from start to finish. The wider the angle of the arrow, the higher is the latency of the message. The green lines indicate a migration state changes. The first and the last green line are the INITIALIZED and FINISHED states. If there are two lines in between, then the synchronization process was triggered and the SYNCING and SYNCED states of the particular migration process on the broker are shown as well.

3. **Departure time to arrival time correlation plot**: The top-left plot shows the correlation between the departure time and the arrival time for each message. This type of plot was also used in [CCWS03] to show the effects of a migration process.

The green lines mark the migration start (INITIALIZED) and finish (FINISHED) on the Source Broker. The violet line indicated the point in time, when the Client connection on the Target Broker got acknowledged and message synchronization starts (MIGTOACK)[17].

4. **Message sequence number to arrival time correlation plot**: The top-right plot shows the correlation between the message sequence number to the arrival time for each message.
5. **Latency by arrival time plot**: The bottom-left plot shows the latency of each message by the arrival time.
6. **Latency by sequence number plot**: The bottom-right plot shows the latency of each message by its sequence number.
7. **Empirical CDF plot**: This plot visualizes the empirical cumulative distribution of the message latency for the Source Broker (red) and the Target Broker (blue). Caused by the experiment design and network setup, the messages from the Target Broker have a lower latency compared to messages from the Source Broker.

[17]Messages that appear slightly after (right to) the green line, are due the fact, that these messages arrived at the Client through the network latency a bit later.

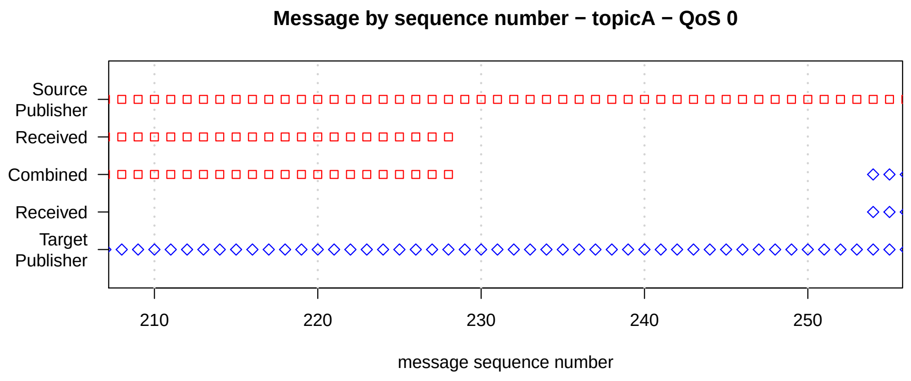
(This page is empty to show the experiment result pages next to each other.)

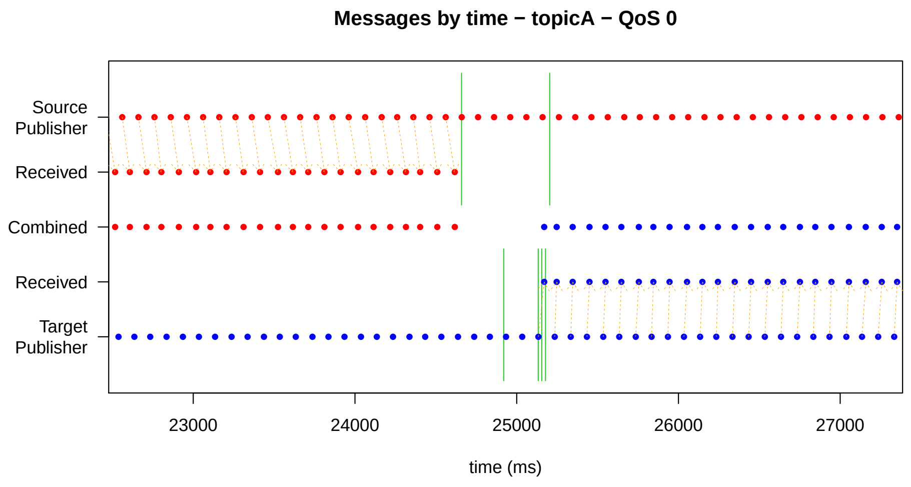
Target Broker is ahead of the Source Broker - QoS 0

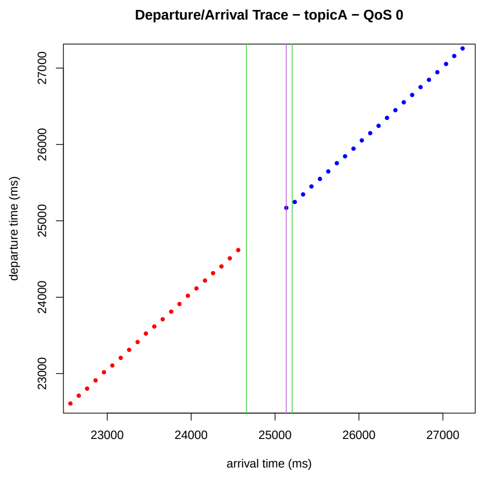
This experiment shows a subscription with QoS 0 with the Target Broker ahead of the Source Broker. The plots show that there is a message gap and 35 messages got lost, since missed messages are not requested during the migration process, and no message duplicates were detected. It is also visible that there was no synchronization necessary on the Target Broker, as the SYNCING and SYNCED marks are right after each other.

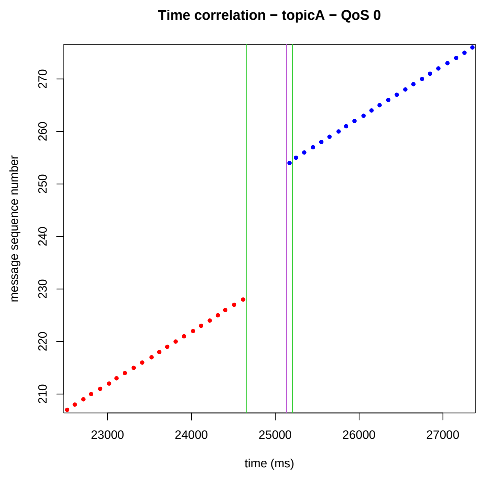
Experiment summary:

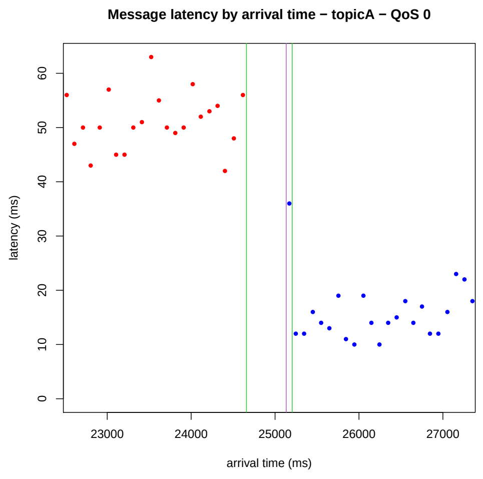
Experiment	Topic	QoS	n losses	n duplicates	Validity
ahead-A0	topicA	QoS 0	25	0	✓

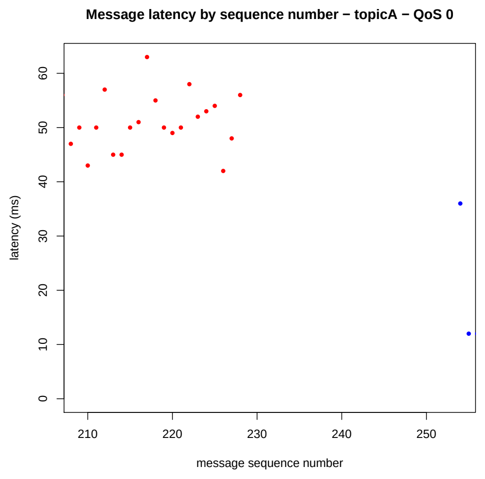


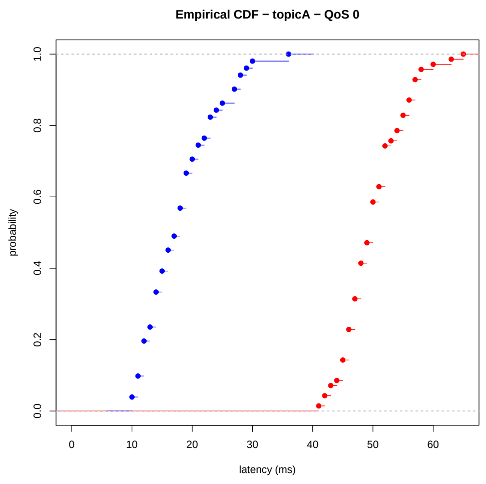



Departure/Arrival Trace − topicA − QoS 0
departure time (ms)
23000
24000
25000
26000
27000
arrival time (ms)
23000
24000
25000
26000
27000


Time correlation − topicA − QoS 0
message sequence number
210
220
230
240
250
260
270
time (ms)
23000
24000
25000
26000
27000


Message latency by arrival time − topicA − QoS 0
latency (ms)
0
10
20
30
40
50
60
arrival time (ms)
23000
24000
25000
26000
27000


Message latency by sequence number − topicA − QoS 0
latency (ms)
0
10
20
30
40
50
60
message sequence number
210
220
230
240
250

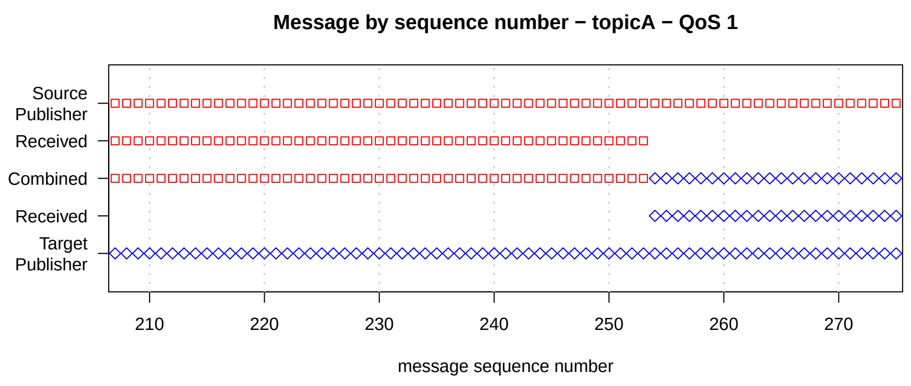

Empirical CDF − topicA − QoS 0
probability
0.0
0.2
0.4
0.6
0.8
1.0
latency (ms)
0
10
20
30
40
50
60

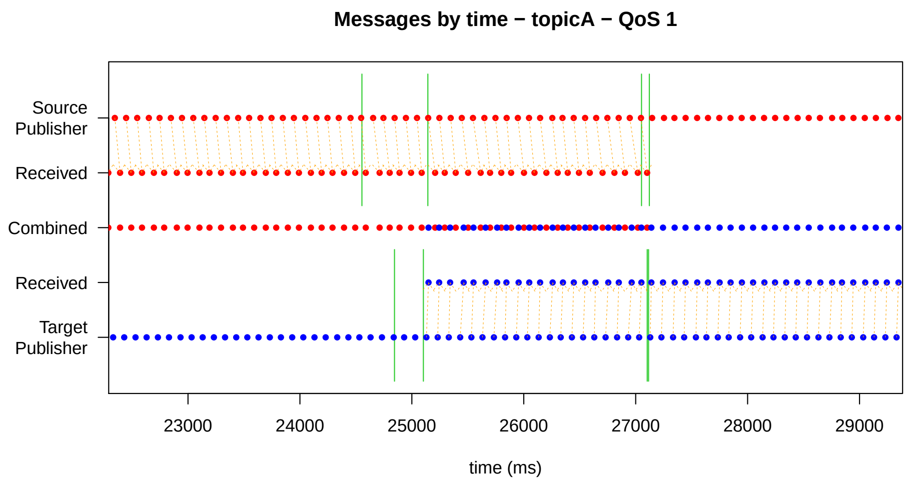
Target Broker is ahead of the Source Broker - QoS 1

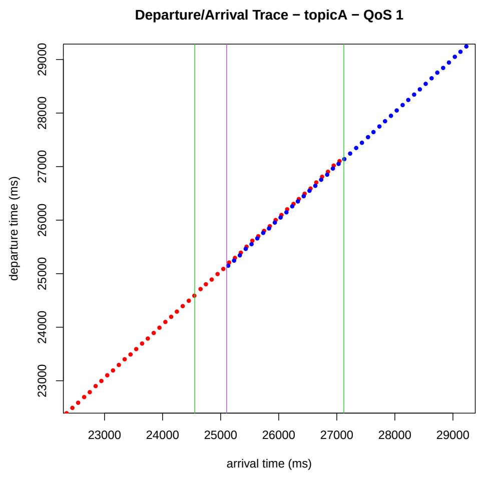
This experiment shows a subscription with QoS 1 with the Target Broker ahead of the Source Broker. Missed messages are requested from the Source Broker, resulting in not lost messages. No message got received twice. Neverthelesse, since after the synchronization process on the Source Broker, a MIGSYNCACK and then a MIGACK packet have to be sent, in which the Source Broker still sends messages, duplicate messages could occur, even if the Target Broker is ahead of the Source Broker, but this was not the case during this run. Therefore, both brokers send the same messages after the synchronization point until the migration process finishes and the Source Broker stops sending messages.

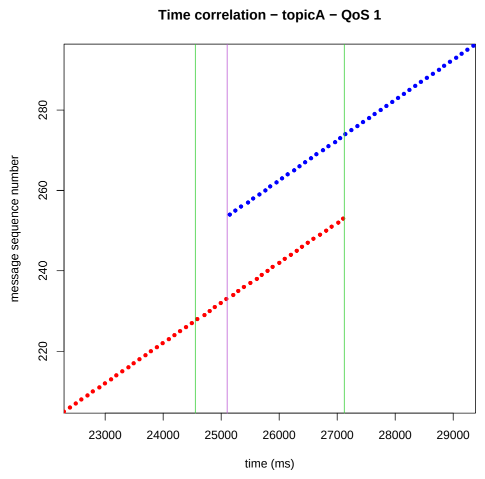
Experiment summary:

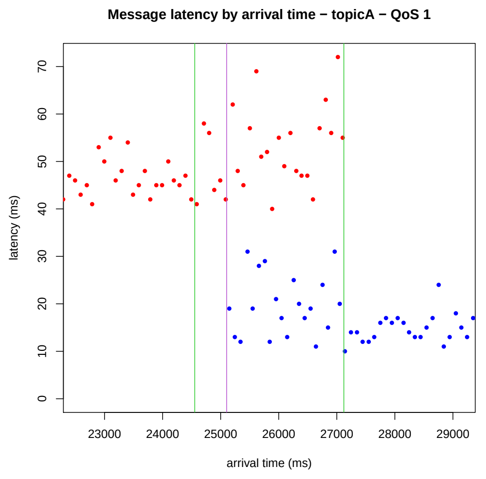
Experiment	Topic	QoS	n losses	n duplicates	Validity
ahead-A1	topicA	QoS 1	0	0	✓

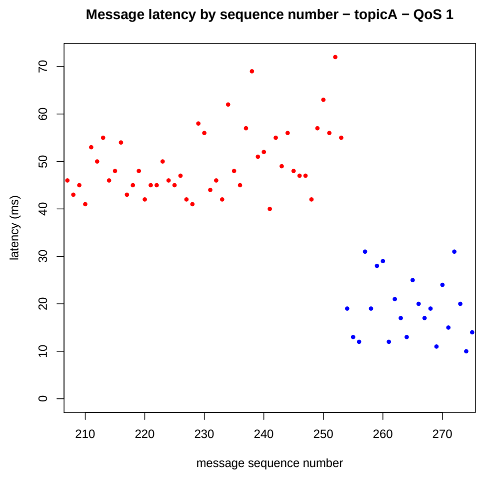


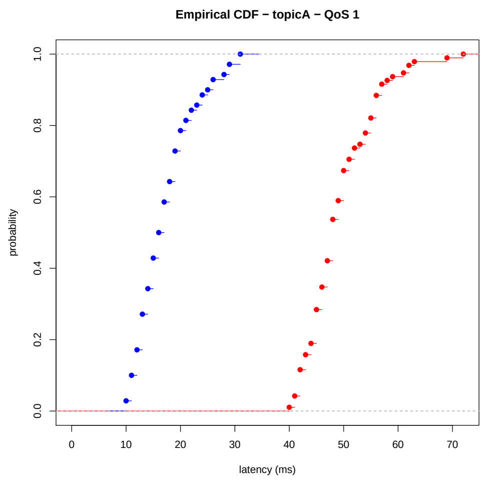



Departure/Arrival Trace − topicA − QoS 1
departure time (ms)
arrival time (ms)


Time correlation − topicA − QoS 1
message sequence number
time (ms)


Message latency by arrival time − topicA − QoS 1
latency (ms)
arrival time (ms)


Message latency by sequence number − topicA − QoS 1
latency (ms)
message sequence number

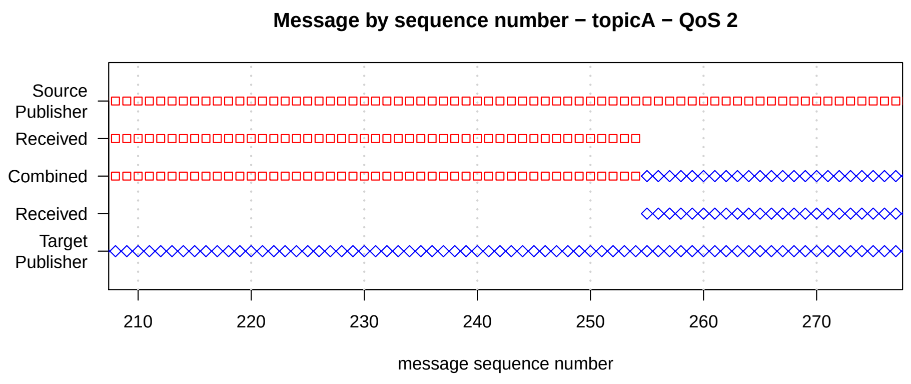

Empirical CDF − topicA − QoS 1
probability
latency (ms)

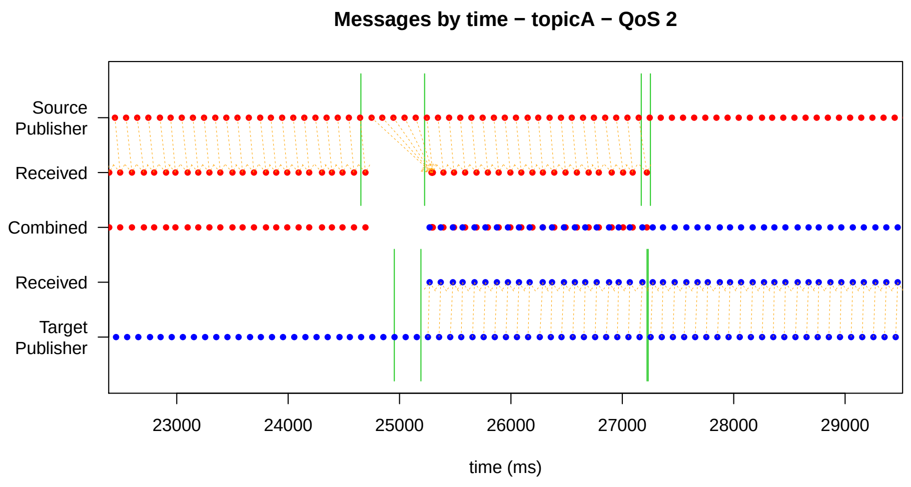
Target Broker is ahead of the Source Broker - QoS 2

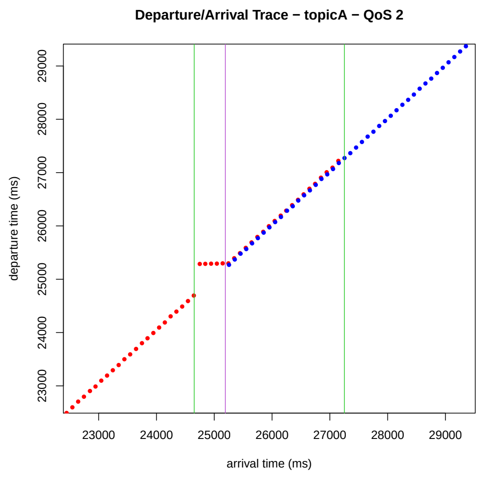
This experiment shows a subscription with QoS 2 with the Target Broker ahead of the Source Broker. Latency outliners are due the migration process, when the Target Broker has to request missing messages from the Source Broker. These messages are then send from the store of the Source Broker and therefore have a higher latency. Some messages were even ahead of the Source Broker and send on arrival, resulting in no lost messages. No message got send and received multiple times within the migration process.

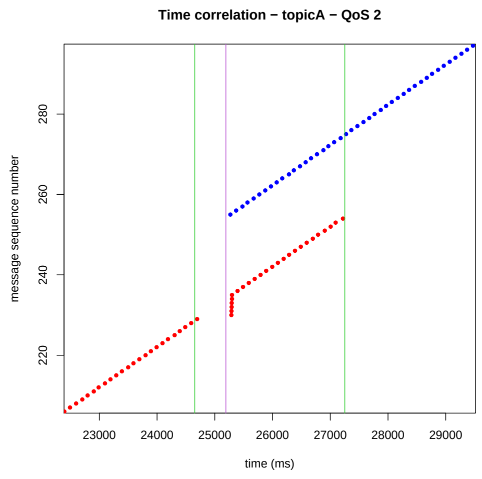
Experiment summary:

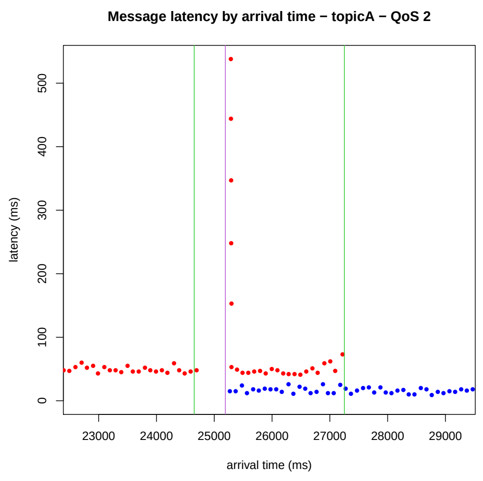
Experiment	Topic	QoS	n losses	n duplicates	Validity
ahead-A2	topicA	QoS 2	0	0	✓

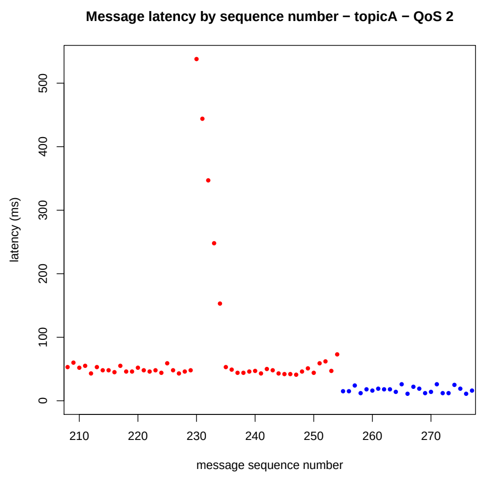


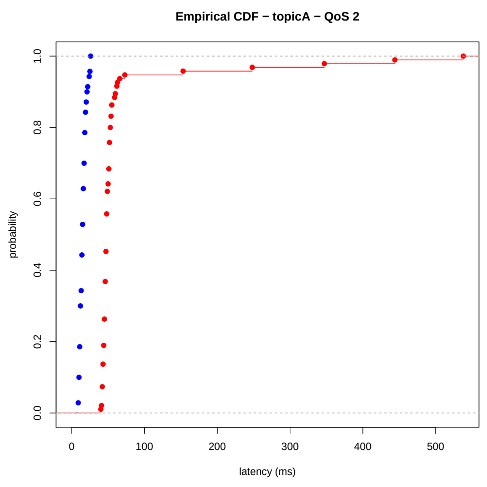



Departure/Arrival Trace − topicA − QoS 2
departure time (ms)
arrival time (ms)
23000
24000
25000
26000
27000
28000
29000


Time correlation − topicA − QoS 2
message sequence number
time (ms)
220
240
260
280
23000
24000
25000
26000
27000
28000
29000


Message latency by arrival time − topicA − QoS 2
latency (ms)
arrival time (ms)
0
100
200
300
400
500
23000
24000
25000
26000
27000
28000
29000


Message latency by sequence number − topicA − QoS 2
latency (ms)
message sequence number
0
100
200
300
400
500
210
220
230
240
250
260
270

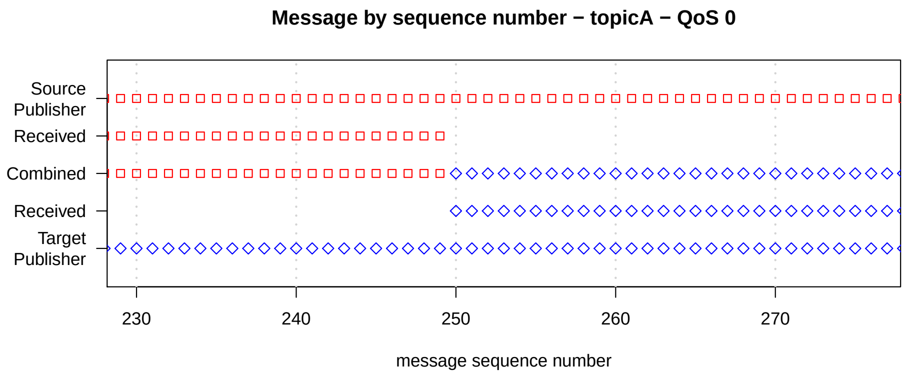

Empirical CDF − topicA − QoS 2
probability
latency (ms)
0.0
0.2
0.4
0.6
0.8
1.0
0
100
200
300
400
500

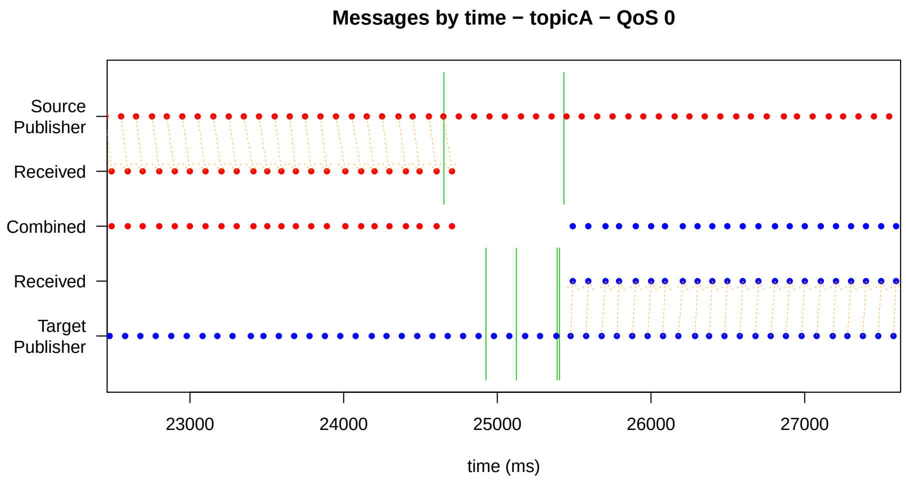
Target Broker is synced with the Source Broker - QoS 0

This experiment shows a subscription with QoS 0 with the Target Broker synced with the Source Broker. The Target Broker can immediately finish the migration process[18]. No messages got lost or arrived multiple times within the migration process.

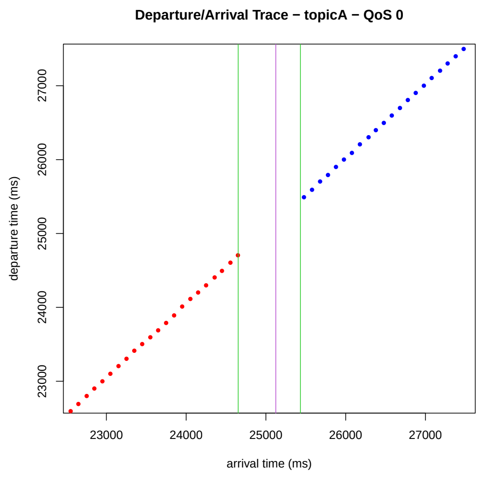
Experiment summary:

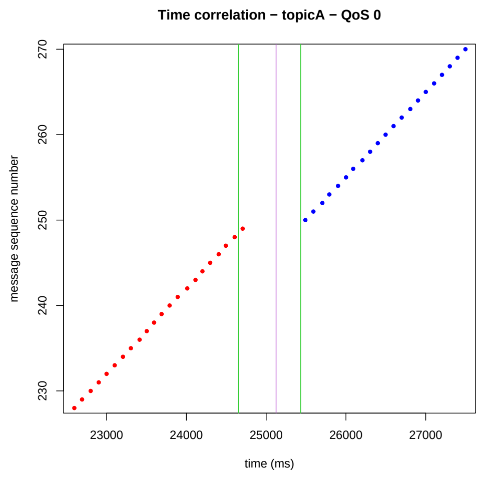
Experiment	Topic	QoS	n losses	n duplicates	Validity
synced-A0	topicA	QoS 0	0	0	✓

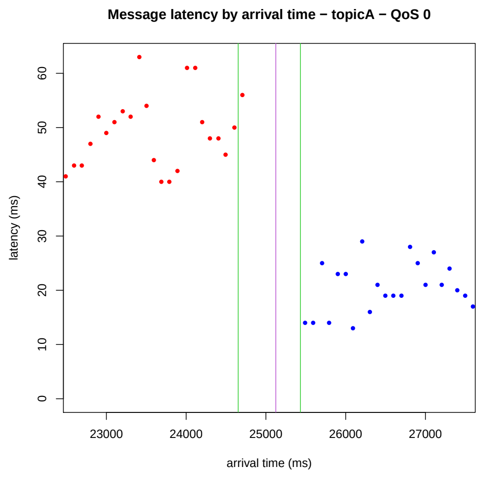


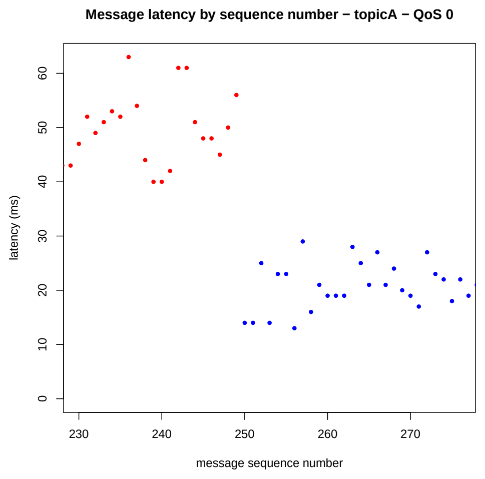


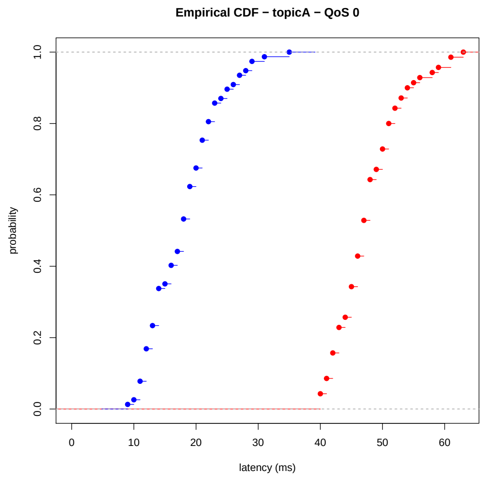
[18]Since our synced scenarios are not 100 % accurate, the Target Broker has to wait to pass three messages, since it was slightly behind.


Departure/Arrival Trace − topicA − QoS 0
departure time (ms)
arrival time (ms)
23000
24000
25000
26000
27000


Time correlation − topicA − QoS 0
message sequence number
time (ms)
230
240
250
260
270
23000
24000
25000
26000
27000


Message latency by arrival time − topicA − QoS 0
latency (ms)
arrival time (ms)
0
10
20
30
40
50
60
23000
24000
25000
26000
27000


Message latency by sequence number − topicA − QoS 0
latency (ms)
message sequence number
0
10
20
30
40
50
60
230
240
250
260
270

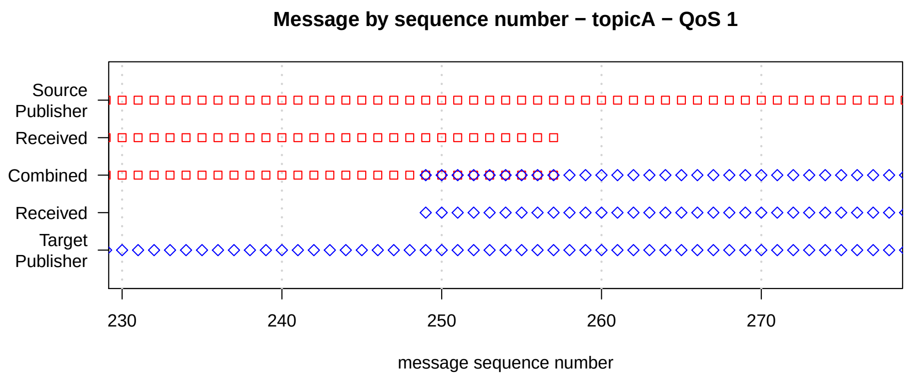

Empirical CDF − topicA − QoS 0
probability
latency (ms)
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

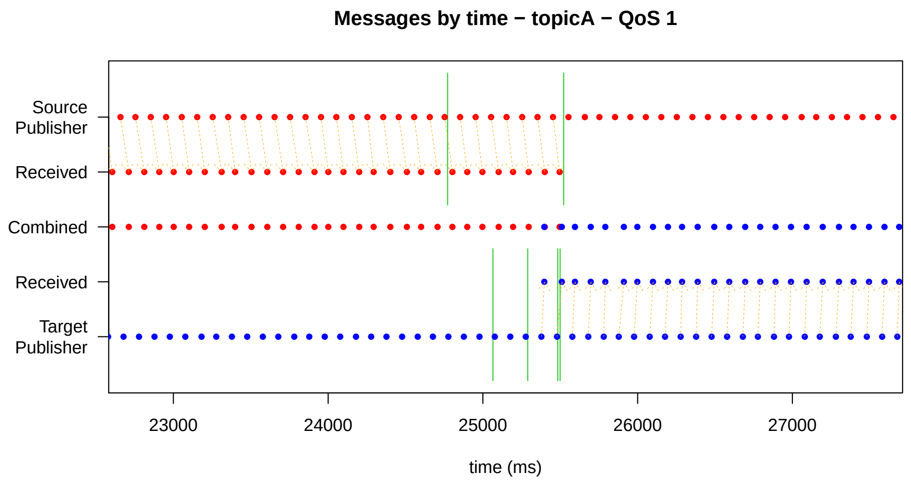
Target Broker is synced with the Source Broker - QoS 1

This experiment shows a subscription with QoS 1 with the Target Broker synced with the Source Broker. The Target Broker can immediately finish the migration process[19]. Since the Source Broker does not stop sending and the Target Broker immediately starts sending messages, both brokers are sending messages to the Client until the Target Broker is synchronized. Since the Source Broker send messages the whole time, some messages get duplicated by the Target Broker as well and sent to the Client. Nevertheless, no message got lost within the migration process.

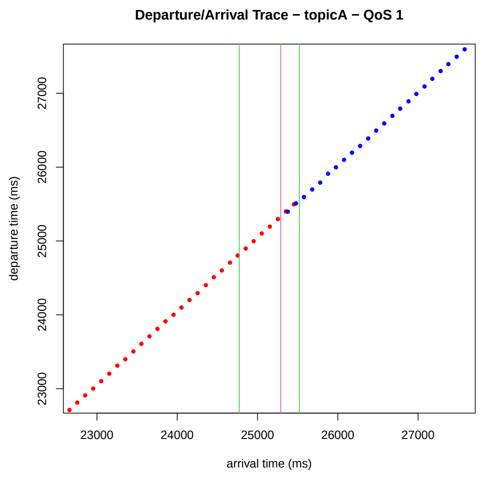
Experiment summary:

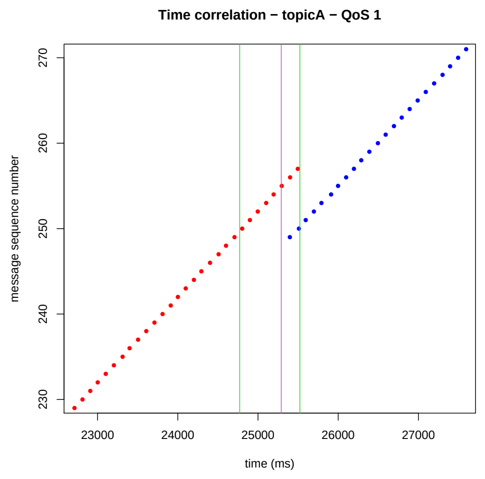
Experiment	Topic	QoS	n losses	n duplicates	Validity
synced-A1	topicA	QoS 1	0	9	✓

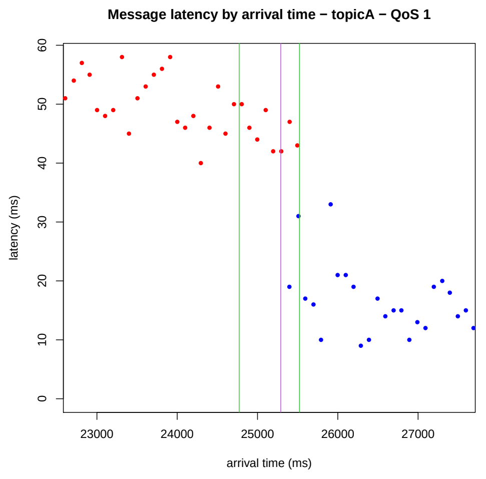


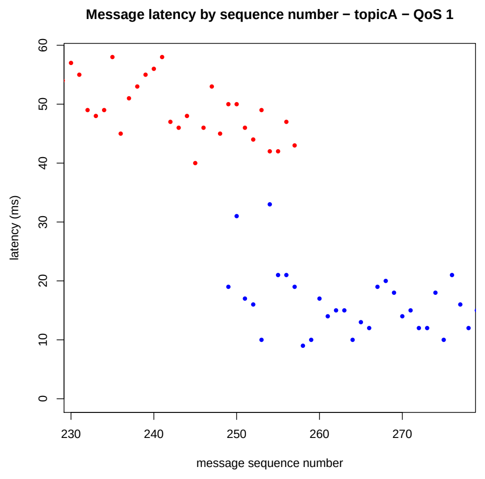


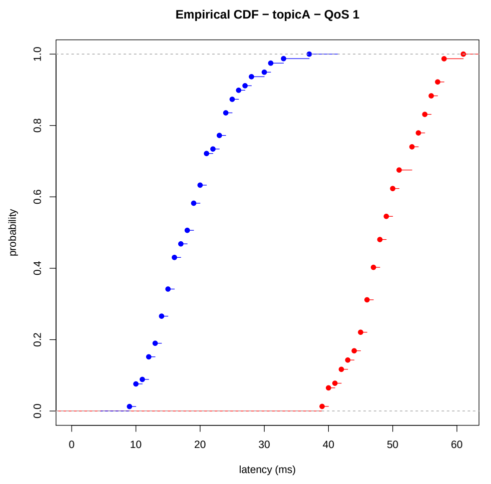
[19]Since our synced scenarios are not 100 % accurate, the Target Broker has to wait to pass two messages, since it was slightly behind.


Departure/Arrival Trace − topicA − QoS 1
departure time (ms)
arrival time (ms)
23000
24000
25000
26000
27000


Time correlation − topicA − QoS 1
message sequence number
time (ms)
230
240
250
260
270
23000
24000
25000
26000
27000


Message latency by arrival time − topicA − QoS 1
latency (ms)
arrival time (ms)
0
10
20
30
40
50
60
23000
24000
25000
26000
27000


Message latency by sequence number − topicA − QoS 1
latency (ms)
message sequence number
0
10
20
30
40
50
60
230
240
250
260
270

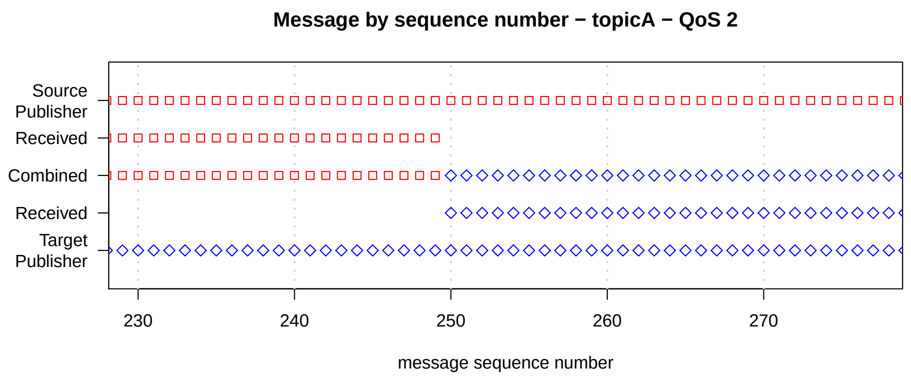

Empirical CDF − topicA − QoS 1
probability
latency (ms)
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

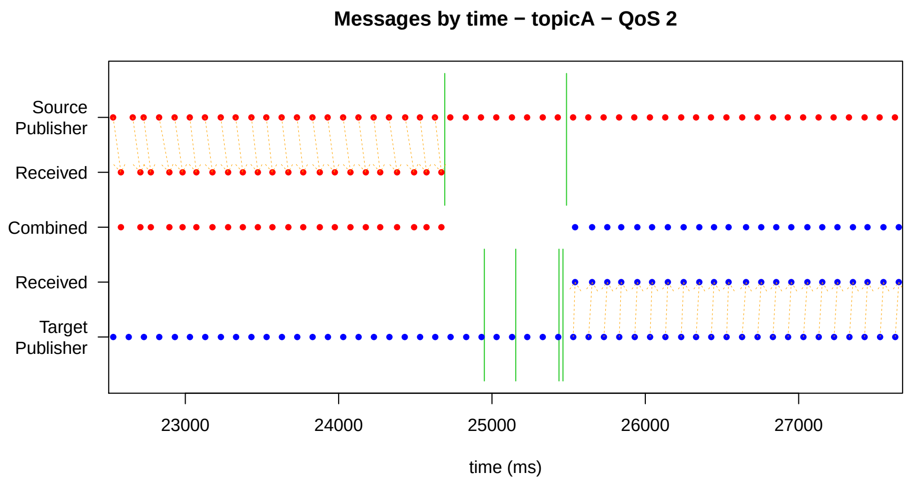
Target Broker is synced with the Source Broker - QoS 2

This experiment shows a subscription with QoS 2 with the Target Broker synced with the Source Broker. The Target Broker can immediately finish the migration process[20]. No message got send and received multiple times within the migration process.

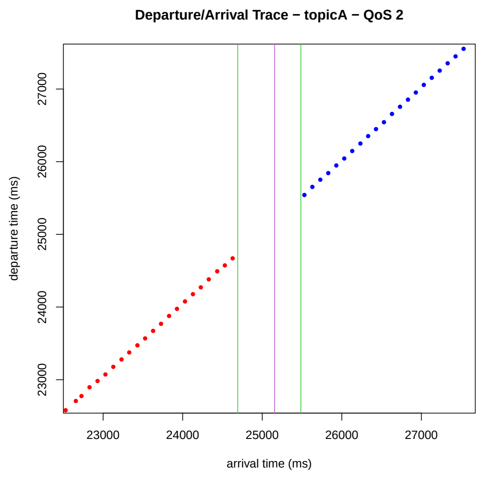
Experiment summary:

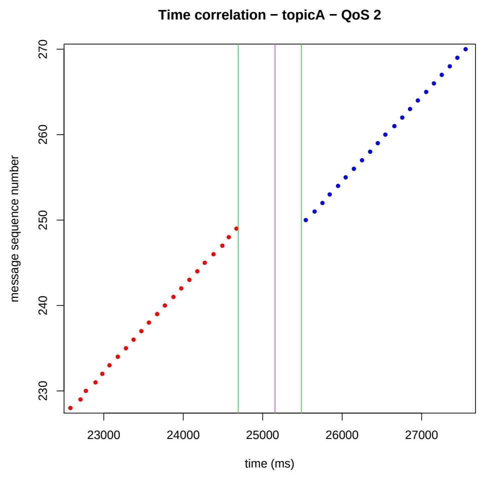
Experiment	Topic	QoS	n losses	n duplicates	Validity
synced-A2	topicA	QoS 2	0	0	✓

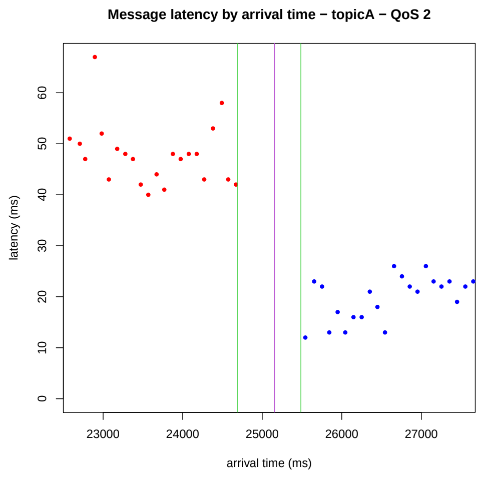


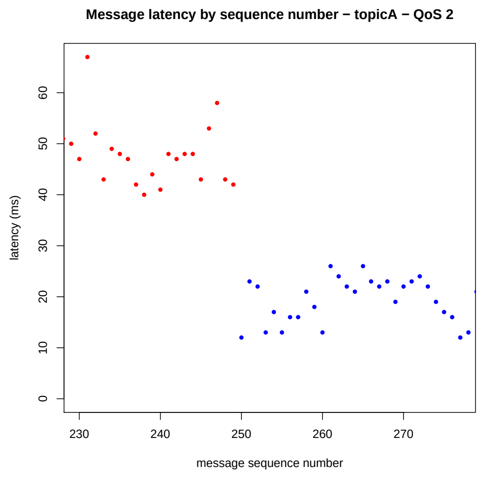


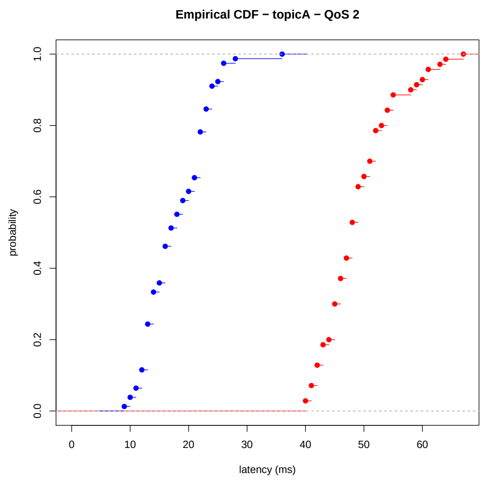
[20]Since our synced scenarios are not 100 % accurate, the Target Broker has to wait to pass three messages, since it was slightly behind.


Departure/Arrival Trace − topicA − QoS 2
departure time (ms)
arrival time (ms)
23000
24000
25000
26000
27000


Time correlation − topicA − QoS 2
message sequence number
time (ms)
230
240
250
260
270
23000
24000
25000
26000
27000


Message latency by arrival time − topicA − QoS 2
latency (ms)
arrival time (ms)
0
10
20
30
40
50
60
23000
24000
25000
26000
27000


Message latency by sequence number − topicA − QoS 2
latency (ms)
message sequence number
0
10
20
30
40
50
60
230
240
250
260
270

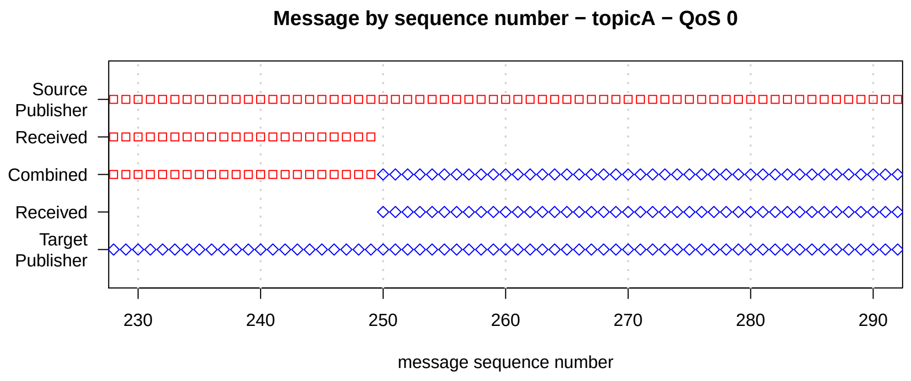

Empirical CDF − topicA − QoS 2
probability
latency (ms)
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

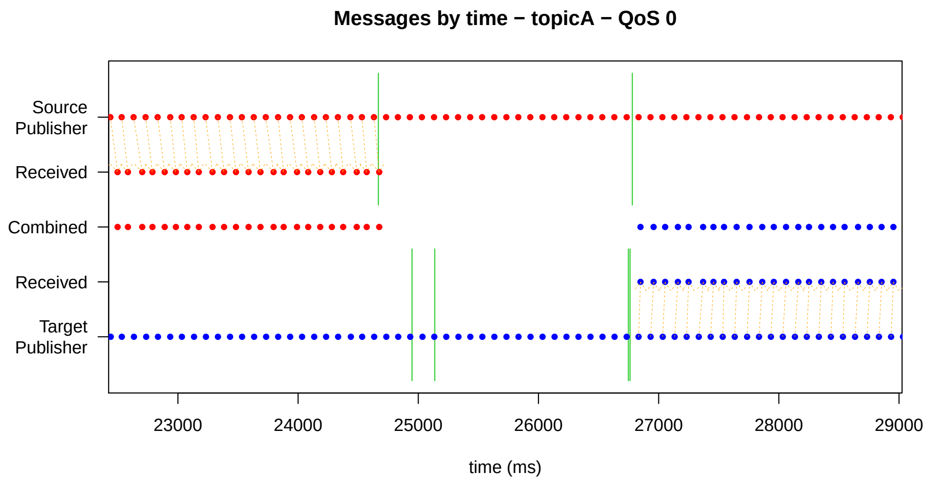
Target Broker is behind of the Source Broker - QoS 0

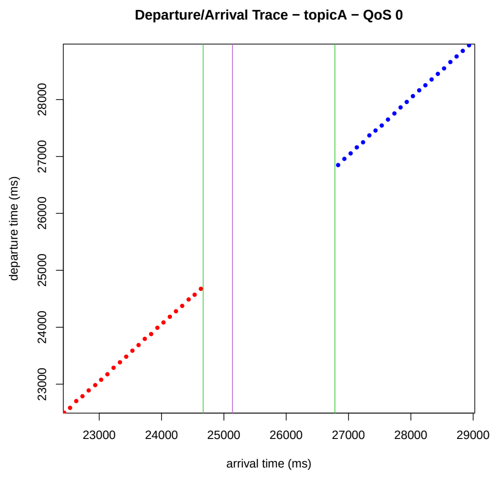
This experiment shows a subscription with QoS 0 with the Target Broker behind of the Source Broker. Since the Target Broker was behind, no messages were missed. The Target Broker had to wait sending messages during its synchronization process, resulting in a message gap for the Client. No messages arrived multiple times within the migration process.

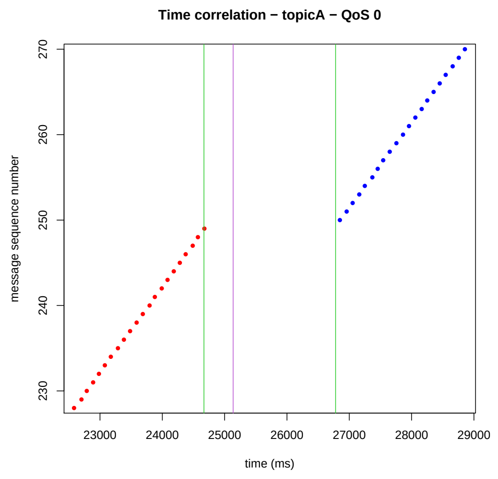
Experiment summary:

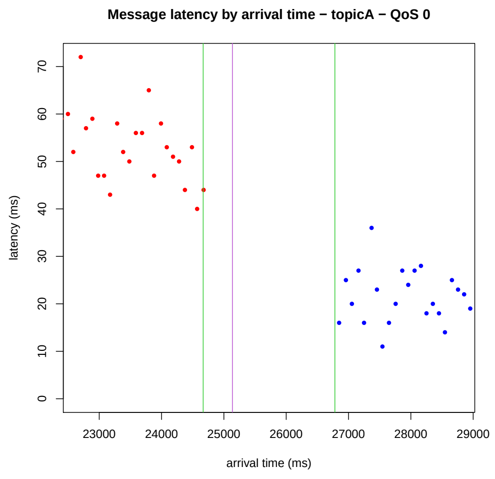
Experiment	Topic	QoS	n losses	n duplicates	Validity
behind-A0	topicA	QoS 0	0	0	✓

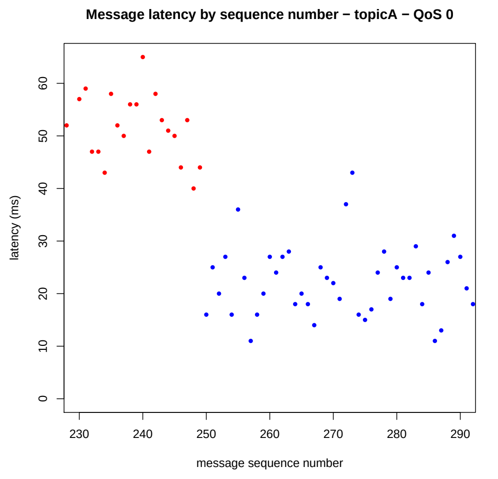


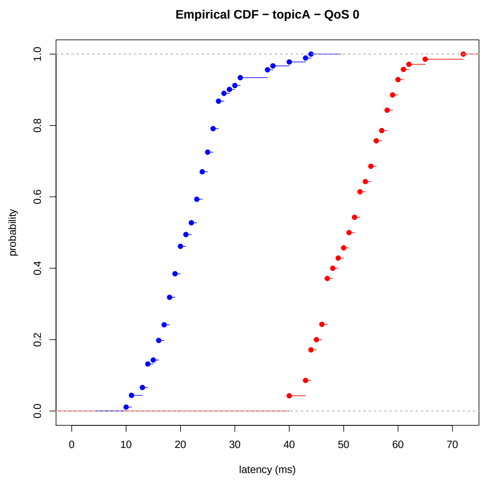



Departure/Arrival Trace − topicA − QoS 0
departure time (ms)
23000
24000
25000
26000
27000
28000
23000
24000
25000
26000
27000
28000
29000
arrival time (ms)


Time correlation − topicA − QoS 0
message sequence number
230
240
250
260
270
23000
24000
25000
26000
27000
28000
29000
time (ms)


Message latency by arrival time − topicA − QoS 0
latency (ms)
0
10
20
30
40
50
60
70
23000
24000
25000
26000
27000
28000
29000
arrival time (ms)


Message latency by sequence number − topicA − QoS 0
latency (ms)
0
10
20
30
40
50
60
230
240
250
260
270
280
290
message sequence number

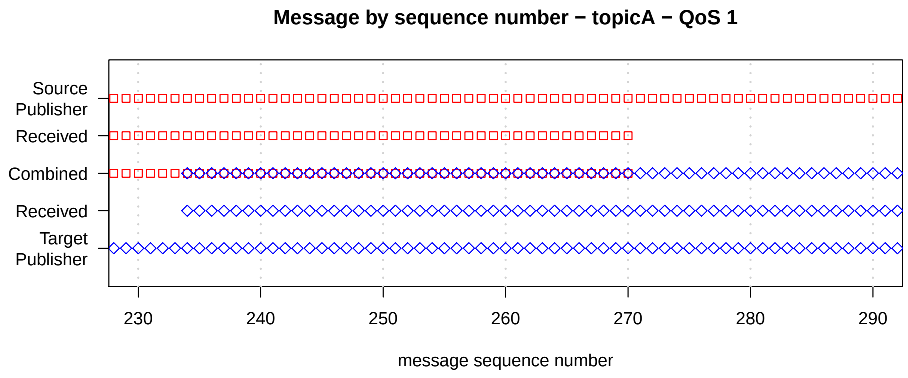

Empirical CDF − topicA − QoS 0
probability
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60
70
latency (ms)

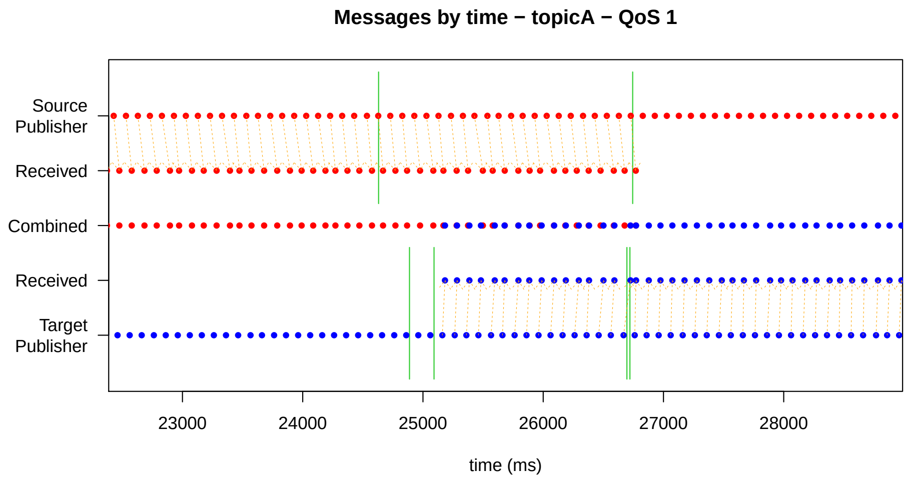
Target Broker is behind of the Source Broker - QoS 1

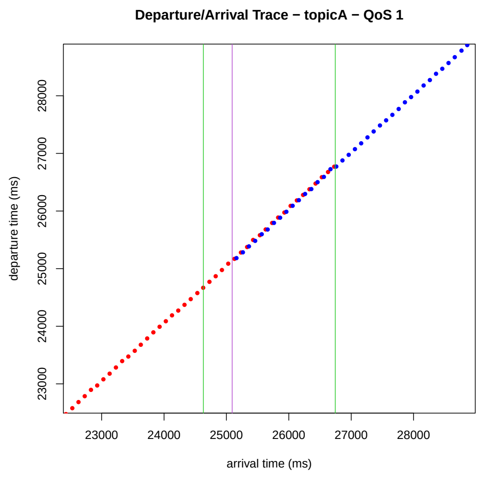
This experiment shows a subscription with QoS 1 with the Target Broker behind of the Source Broker. Since the Target Broker is behind, it has to wait until it is synchronized. Since the Source Broker does not stop sending and the Target Broker immediately starts sending messages, both brokers are sending messages to the Client until the Target Broker is synchronized. Since the Source Broker send messages the whole time, many messages get duplicated by the Target Broker as well and sent to the Client. Nevertheless, no message got lost within the migration process.

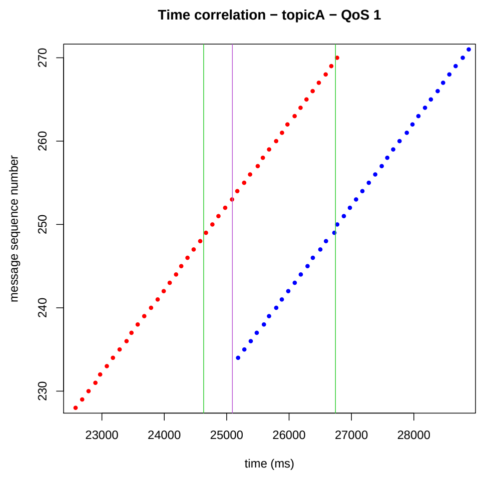
Experiment summary:

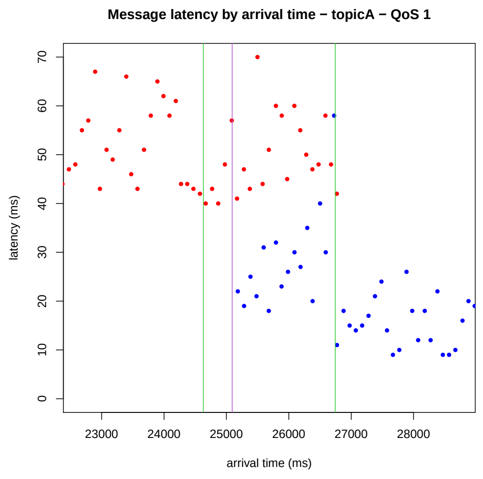
Experiment	Topic	QoS	n losses	n duplicates	Validity
behind-A1	topicA	QoS 1	0	37	✓

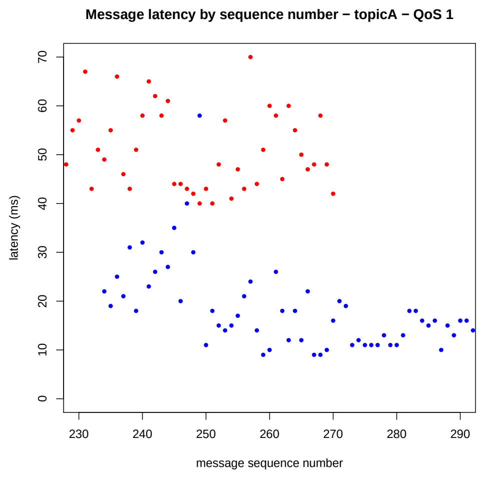


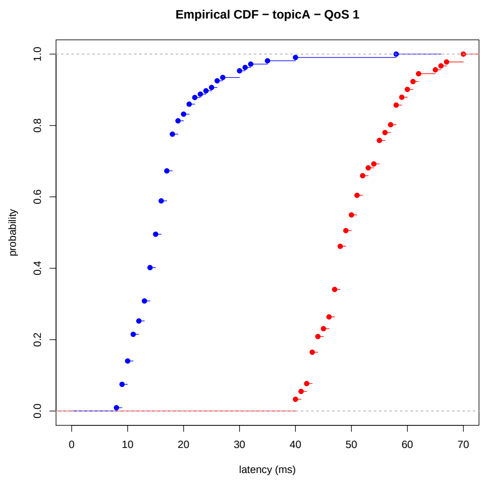



Departure/Arrival Trace − topicA − QoS 1
departure time (ms)
23000
24000
25000
26000
27000
28000
arrival time (ms)


Time correlation − topicA − QoS 1
message sequence number
230
240
250
260
270
23000
24000
25000
26000
27000
28000
time (ms)


Message latency by arrival time − topicA − QoS 1
latency (ms)
0
10
20
30
40
50
60
70
23000
24000
25000
26000
27000
28000
arrival time (ms)


Message latency by sequence number − topicA − QoS 1
latency (ms)
0
10
20
30
40
50
60
70
230
240
250
260
270
280
290
message sequence number

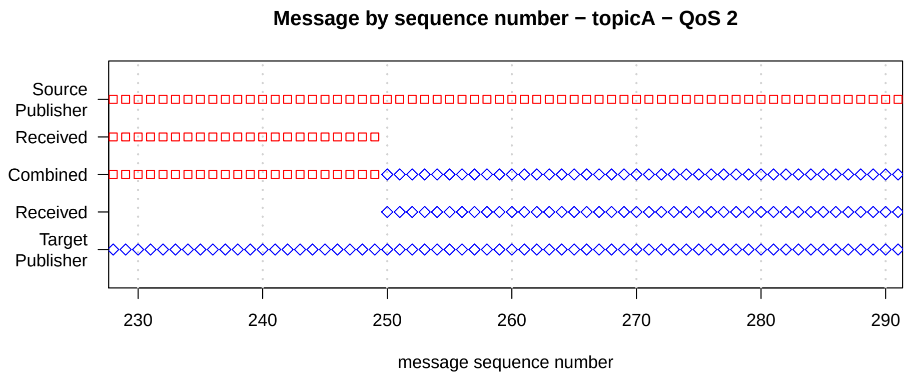

Empirical CDF − topicA − QoS 1
probability
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60
70
latency (ms)

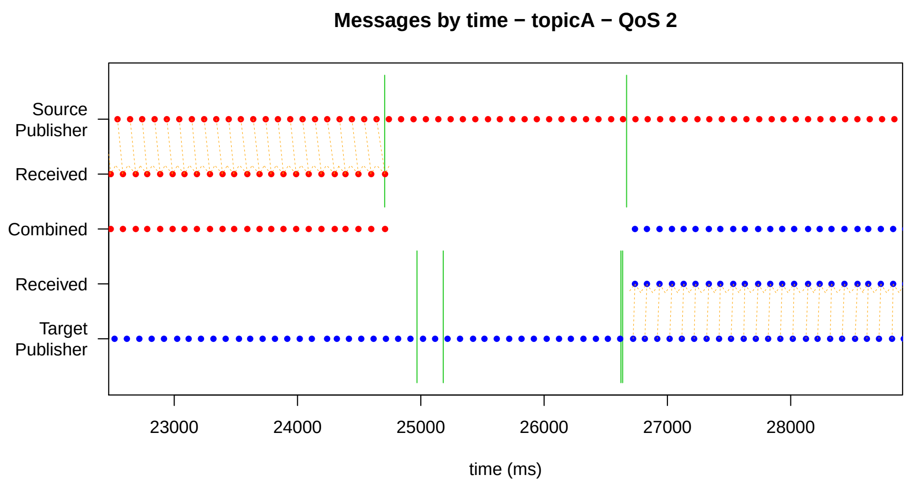
Target Broker is behind of the Source Broker - QoS 2

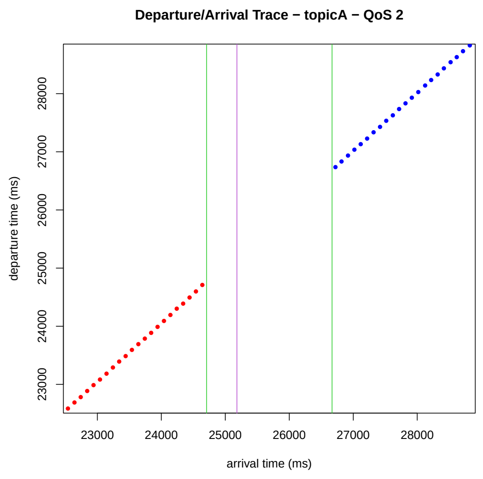
This experiment shows a subscription with QoS 2 with the Target Broker behind of the Source Broker. Similar to QoS 1, since the Target Broker is behind, it just has to wait to continue sending messages. Caused by the synchronization delta of the brokers, a gap for the Client arises, in which he does not receive any message. Nevertheless not duplicates and no losses are the result of the migration process.

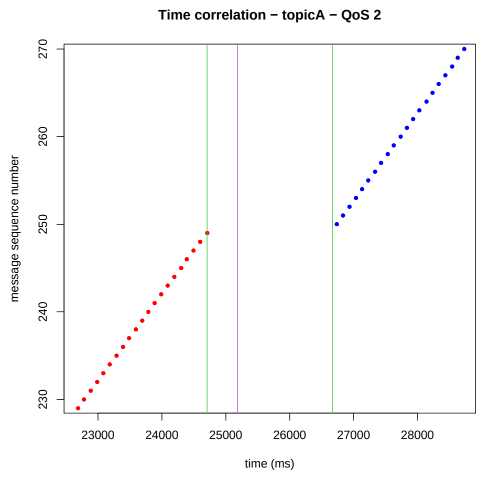
Experiment summary:

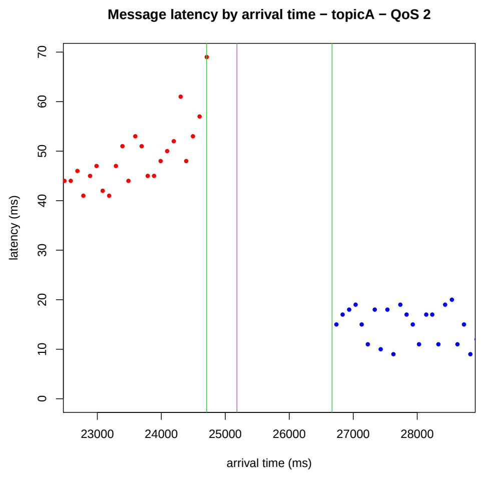
Experiment	Topic	QoS	n losses	n duplicates	Validity
behind-A2	topicA	QoS 2	0	0	✓

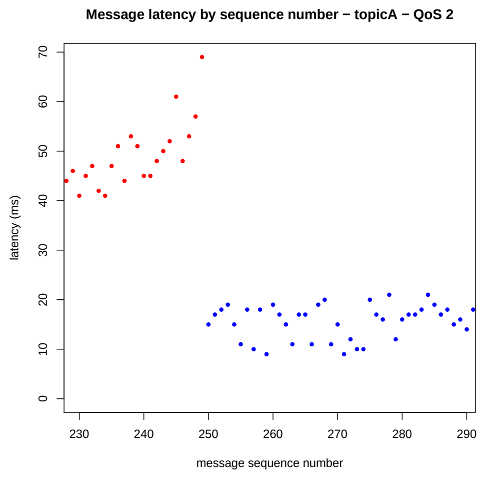


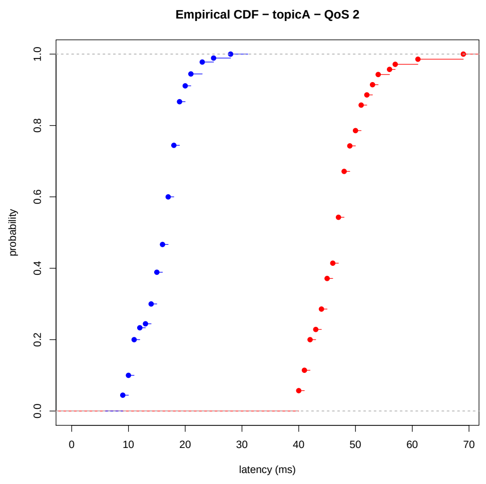



Departure/Arrival Trace − topicA − QoS 2
departure time (ms)
23000
24000
25000
26000
27000
28000
arrival time (ms)
23000
24000
25000
26000
27000
28000


Time correlation − topicA − QoS 2
message sequence number
230
240
250
260
270
time (ms)
23000
24000
25000
26000
27000
28000


Message latency by arrival time − topicA − QoS 2
latency (ms)
0
10
20
30
40
50
60
70
arrival time (ms)
23000
24000
25000
26000
27000
28000


Message latency by sequence number − topicA − QoS 2
latency (ms)
0
10
20
30
40
50
60
70
message sequence number
230
240
250
260
270
280
290


Empirical CDF − topicA − QoS 2
probability
0.0
0.2
0.4
0.6
0.8
1.0
latency (ms)
0
10
20
30
40
50
60
70

6.3.4 Experiments Summary

From the plots and the results we derive that the migration process was successfully performed and that after the Client migration, messages are delivered faster with a lower latency from the Target Broker to the Client compared to deliveries from the Source Broker, as expected by the experiment setup.

A summarized overview of our *system strains* of lost and duplicated messages during the migration process for all experiments runs is shown in Table 6.4. Furthermore, regarding to the *correctness* of our solution approach, the delivery guarantees for each subscription in every experiment through the migration process are valid.

Table 6.4: Experiment result - Total message losses and duplicates

Experiment	**Topic**	**QoS**	**losses**	**duplicates**	**Validity**
ahead-A0	topicA	QoS 0	247	0	✓
ahead-A1	topicA	QoS 1	0	1	✓
ahead-A2	topicA	QoS 2	0	0	✓
ahead-A0_B1_C2	topicA	QoS 0	254	0	✓
	topicB	QoS 1	0	7	✓
	topicC	QoS 2	0	0	✓
synced-A0	topicA	QoS 0	0	0	✓
synced-A1	topicA	QoS 1	0	101	✓
synced-A2	topicA	QoS 2	0	0	✓
synced-A0_B1_C2	topicA	QoS 0	0	0	✓
	topicB	QoS 1	0	98	✓
	topicC	QoS 2	0	0	✓
behind-A0	topicA	QoS 0	0	0	✓
behind-A1	topicA	QoS 1	0	358	✓
behind-A2	topicA	QoS 2	0	0	✓
behind-A0_B1_C2	topicA	QoS 0	0	0	✓
	topicB	QoS 1	0	353	✓
	topicC	QoS 2	0	0	✓
mixed-A0_B1_C2_D0_E1_F2	topicA	QoS 0	261	0	✓
	topicB	QoS 1	0	2	✓
	topicC	QoS 2	0	0	✓
	topicD	QoS 0	1	0	✓
	topicE	QoS 1	0	402	✓
	topicF	QoS 2	0	0	✓

The *responsiveness* of our solution approach, especially the duration of the whole migration depends on multiple parameters like:

- network delays
- amount of subscriptions of a client
- the QoS for each subscription
- the synchronization state of a subscriptions between the two brokers

The migration process durations of our experiments are visualized in Figure 6.4. The visualization shows, that the migration process time is at its minimum, when the brokers are synchronized[21], or when the Target Broker is ahead and QoS 0 is configured, since missed messages do not get synchronized and the Target Broker just continues sending messages. The minimum migration time for our scenario is around 500-800 ms.

The visualization also shows, the migration duration, when the Target Broker is behind, is roughly the same for all QoS, since the synchronization process for all QoS is the same (the Target Broker has to wait to be synced). For the cases where the Target Broker is ahead, QoS 1 and QoS 2 are roughly the same and higher compared to QoS 0, since the Target Broker has to synchronize with the Source Broker; with QoS 0, this is not necessary and therefore the migration duration is lower.

In the visualization, we can also see, that the migration time increases, if the brokers are not synced, i.e. the Target Broker is behind or ahead of the Source Broker, as expected.

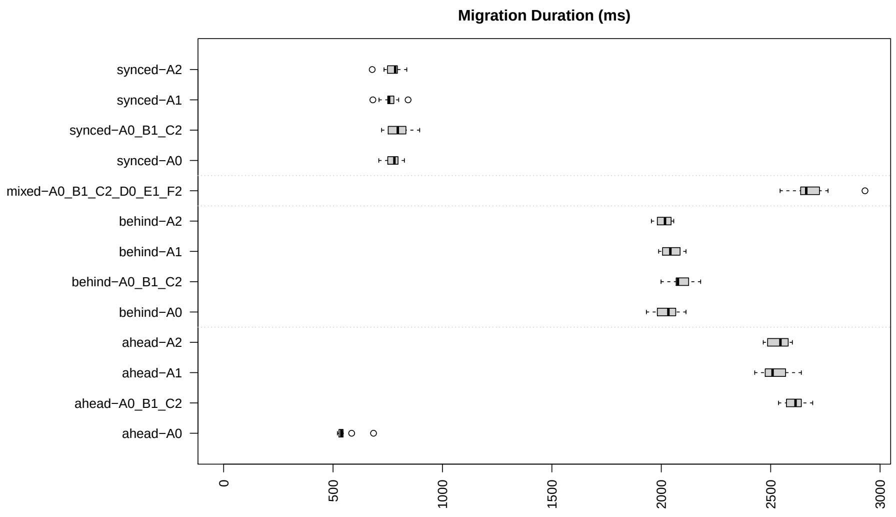
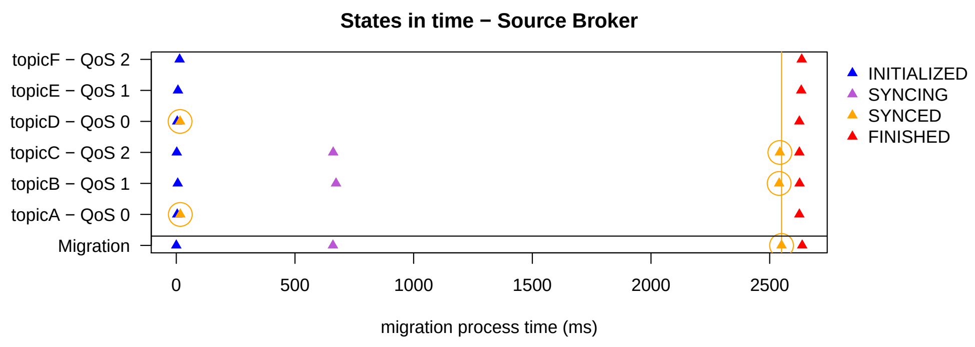


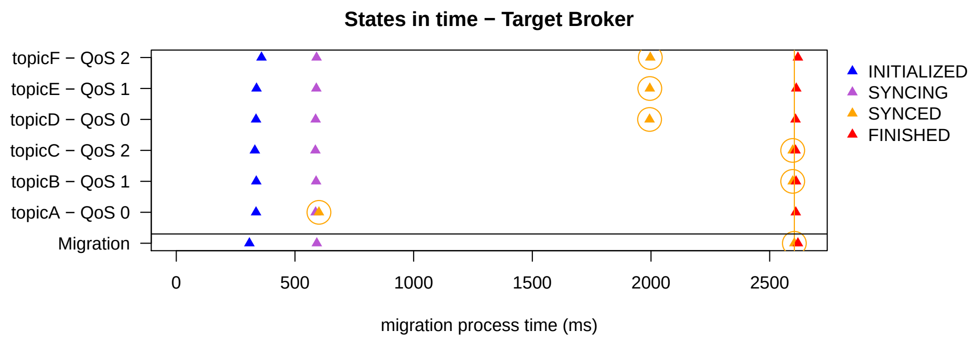
Figure 6.4: Experiment Result: Migration process durations

[21]Since our synced scenarios are not 100 % accurate, the Target Broker is slightly behind and therefore the migration time increases a little bit.

When a Client has multiple subscriptions, as with `ahead-A0\_B1\_C2`, `behind-A0\_B1\_C2` and `mixed-A0\_B1\_C2\_D0\_E1\_F2`, the migration process duration is always the duration of the longest subscription synchronization process time. For example, if there are three subscriptions and two SYNCED immediately on the Target Broker, but the third one needs to request missing messages from the Source Broker, only if these are processed, the last subscription is set to SYNCED as well, and therefore only then the whole migration is set to SYNCED and afterwards FINISHED. This leads to an increase of message duplicates or storage, since the final sending of MIGACK will be postponed.

These conclusions also are visible in the following figures by representing the synchronization states over time on the Source Broker and on the Target Broker for the experiment `mixed-A0\_B1\_C2\_D0\_E1\_F2`. The top part shows see all topics with its subscription states[22] on their time axis and the bottom part shows the same for the migration state on a particular broker system. In orange circle highlights the SYNCED state of each topic and the migration. The visualization shows, that the migration state on the Target Broker is set to SYNCED (indicated with the orange line), only if all subscriptions are SYNCED. In this example, `topicB` or `topicC` were the last subscriptions on the Target Broker that finished the synchronization and blocked sending the MIGACK packet.





In the next section, we analyse the migration time and other aspects theoretically.

[22]The subscription states SYNCING_THIS and SYNCING_OPPONENT are not shown, since they are just substates of the SYNCING state and therefore not relevant for our point.

6.4 Theoretical Analysis

In the following sections, we analyze our solution approach theoretically. We define a mathematical model of it and describe relations and outcomes in time and load to give answers to *responsiveness* and to the *systems strains* of our solution approach. We further divide our results into three buckets depending on the synchronization state and present our findings.

6.4.1 Mathematical Model

We define a simplified mathematically model to analyze *responsiveness* and *systems strains* of our solution approach. The simplification does not include the processing time of the individual systems or any variances, like network latencies. We implemented these formulas with R (Listing 8.3 shows the calculation function) to verify and plot values and to see correlations between different properties[23].

We use the following values to for visualization: $\Delta t_{N_{SB,TB}} = 40$, $\Delta t_{N_{SB,C}} = 40$, $\Delta t_{N_{TB,C}} = 10$, $t_v = 10$, $\Delta t_{P_{SB}} = 500$, $\Delta t_{P_{TB}} = -800, ..., 800$.

Figure 6.5 shows the mathematical model of the migration process with annotated variables that we use in the following.

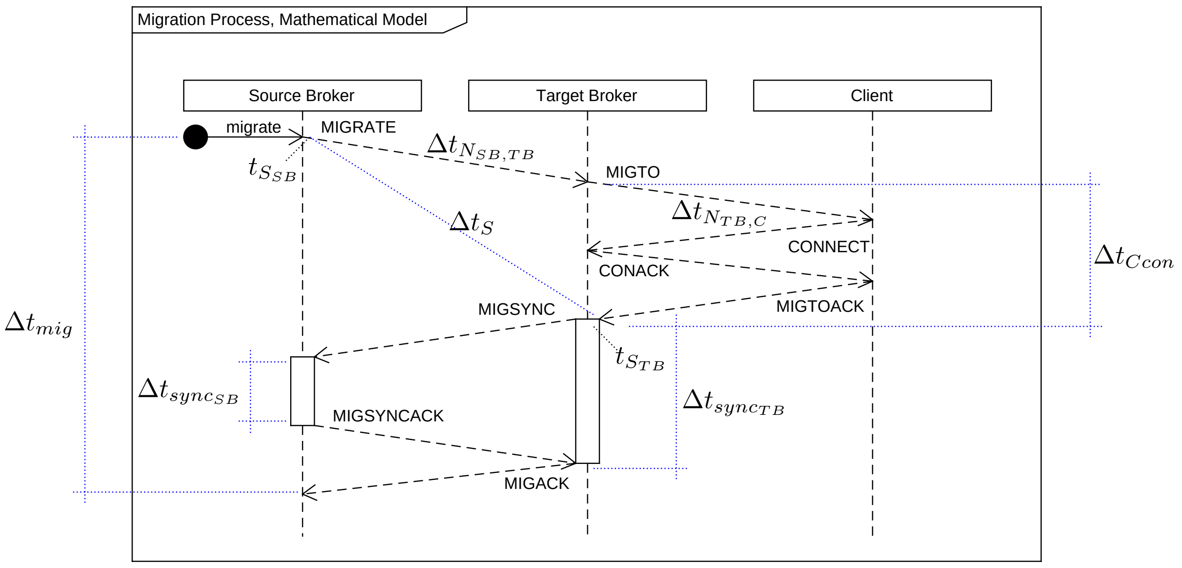


Figure 6.5: Mathematical model of the migration process with annotated variables.

[23]The following formulas analyze the migration process of a single subscription. An extensions of our mathematical model for multiple subscriptions is left for future work.

The latencies between individual systems in the network are defined as follows:
$\Delta t_{N_{SB,C}}$ is the link latency between the Source Broker and the Client.
$\Delta t_{N_{TB,C}}$ is the link latency between the Target Broker and the Client.
$\Delta t_{N_{SB,TB}}$ is link latency between the Source Broker and the Target Broker.
The link latencies are commutative, $\Delta t_{N_{x,y}} = \Delta t_{N_{y,x}}$.

The duration for a Target Broker to initialize a connection to the Client Δt_{Ccon} is four times the link latency between the Target Broker and the Client for sending the MIGTO, CONNECT, CONACK and MIGTOACK packets.

$$\Delta t_{Ccon} = 4 * \Delta t_{N_{TB,C}}$$

The network link latency difference from the brokers to the Client is defined by Δt_N.

The synchronization state of the brokers for a subscription is defined by Δt_S.

To determine the synchronization state Δt_S, we have to derive Δt_P, that defines the difference between the arrival time of the same message at a Source Broker, $\Delta t_{P_{SB}}$, and at the Target Broker, $\Delta t_{P_{TB}}$.

$$\Delta t_P = \Delta t_{P_{SB}} - \Delta t_{P_{TB}}$$

The point in time where the Source Broker defines its synchronization point is referred by $t_{S_{SB}}$ and is defined with 0 since this is the point of the initialization of the migration process. The synchronization point of the Target Broker $t_{S_{TB}}$ is ahead of the synchronization point from the Source Broker adding the time for the MIGRATE packet and the establishment of the Client connection.

$$t_{S_{SB}} = 0$$

$$t_{S_{TB}} = t_{S_{SB}} + \Delta t_{N_{SB,TB}} + \Delta t_{Ccon}$$

The synchronization state Δt_S is the difference between the current state of the Source Broker $t_{S_{SB}}$ and the Target Broker $t_{S_{TB}}$ and the delta of the time the message arrive at the brokers, Δt_P.

$$\Delta t_S = t_{S_{SB}} - t_{S_{TB}} + \Delta t_P$$

If $\Delta t_S > 0$, the Target Broker is *ahead* of the Source Broker.
If $\Delta t_S = 0$, the Target Broker is *synced* with the Source Broker.
If $\Delta t_S < 0$, the Target Broker is *behind* of the Source Broker.

6.4.2 Synchronization Process Time

Depending on the QoS and the synchronization state Δt_S, the time it takes to synchronize the Source Broker and Target broker is different.

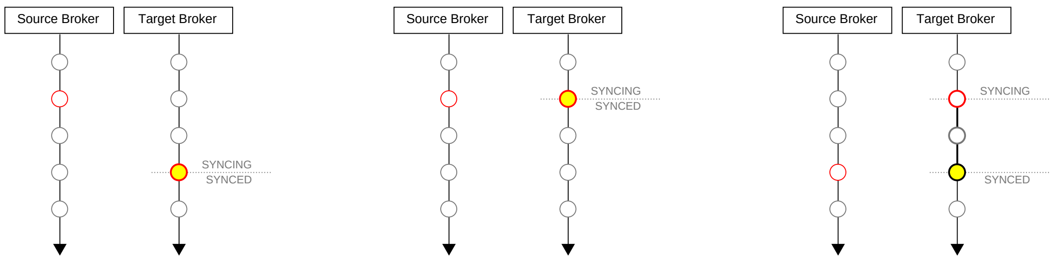
We visualized the different synchronization processes in Figure 6.6, Figure 6.7 and Figure 6.8. The circles represent the message stream for each broker. The $msg_{lastProcId}^{\{source,target\}}$ of the Source Broker and Target Broker are marked in red and are respectively $t_{S_{\{SB,TB\}}}$. The synchronization starts when the Target Broker calculated the synchronization state, starts syncing (SYNCING) and ends when both message streams are synchronized (SYNCED), marked with the yellow circle. The whole synchronization process path is marked bold.

The synchronization duration $\Delta t_{sync_{SB}}$ on the Source Broker depends on the value of the synchronization delta, Δt_S. If synchronization is necessary ($\Delta t_S > 0$), and if the synchronization delta is lower than the messages that have already passed in the meanwhile ($\Delta t_S < (t_{S_{TB}} + \Delta t_{N_{TB,SB}})$), then the synchronization duration is 0, since all messages can be served from the message store. Otherwise, if synchronization is necessary and if there are still some messages ahead, we have to wait for these messages to be sent ($\Delta t_S - (t_{S_{TB}} + \Delta t_{N_{TB,SB}})$).
If no synchronization between the brokers is necessary, the synchronization duration on the Source Broker is 0.

$$\Delta t_{sync_{SB}} = \begin{cases} max(\Delta t_S - (t_{S_{TB}} + \Delta t_{N_{TB,SB}}), 0), & \Delta t_S > 0 \text{ (ahead)} \\ 0, & \text{otherwise (synced, behind)} \end{cases}$$

The synchronization duration time on the Target Broker $\Delta t_{sync_{TB}}$ in case of requesting missing message from the Source Broker is determined by sending a MIGSYNC packet to the Source Broker, $\Delta t_{N_{TB,SB}}$, the synchronization duration of the Source Broker, $\Delta t_{sync_{SB}}$, and finally with a MIGSYNCACK packet that is sent back to the Target Broker, $\Delta t_{N_{SB,TB}}$.
Otherwise, the synchronization duration $\Delta t_{sync_{TB}}$ on the Target Broker is the absolute value of the synchronization difference Δt_S; the time it takes to synchronize locally.

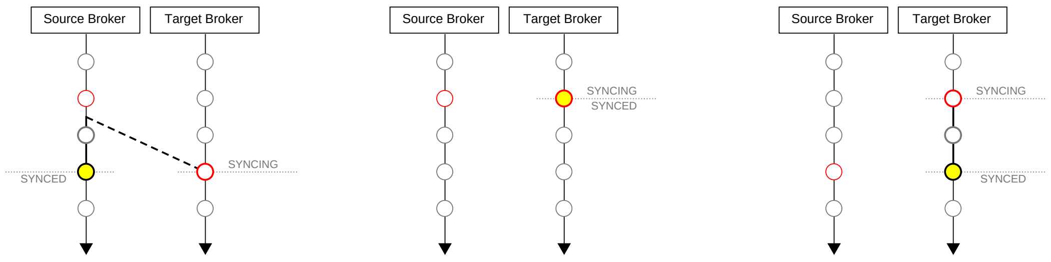
$$\Delta t_{sync_{TB}} = \begin{cases} \Delta t_{N_{TB,SB}} + \Delta t_{sync_{SB}} + \Delta t_{N_{SB,TB}}, & \Delta t_S > 0 \text{ (ahead)} \\ 0, & \Delta t_S = 0 \text{ (synced)} \\ -\Delta t_S, & \Delta t_S < 0 \text{ (behind)} \end{cases}$$



(a) ahead, $\Delta t_S > 0$ (b) synced, $\Delta t_S = 0$ (c) behind, $\Delta t_S < 0$

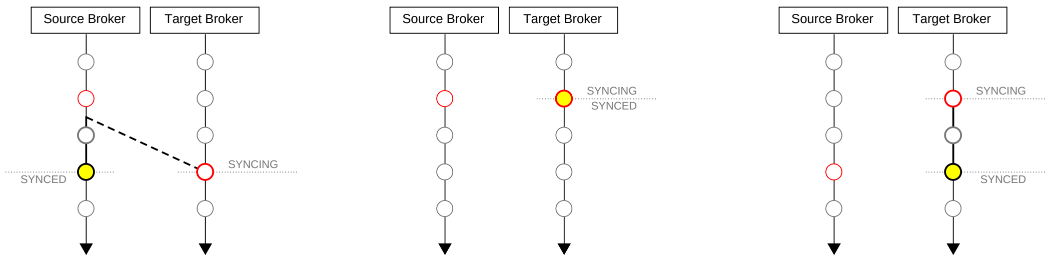
Figure 6.6: Synchronization process for QoS 0

! !



(a) ahead, $\Delta t_S > 0$ (b) synced, $\Delta t_S = 0$ (c) behind, $\Delta t_S < 0$

Figure 6.7: Synchronization process for QoS 1



(a) ahead, $\Delta t_S > 0$ (b) synced, $\Delta t_S = 0$ (c) behind, $\Delta t_S < 0$

Figure 6.8: Synchronization process for QoS 2

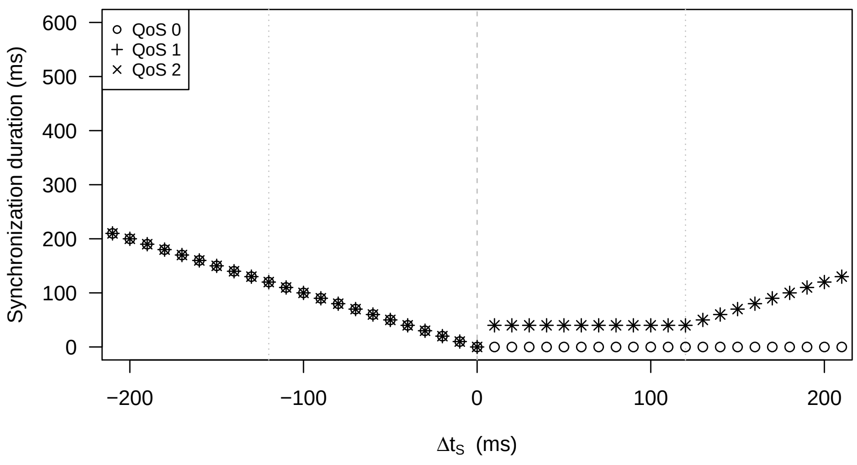


Figure 6.9: Synchronization process time for each QoS based on the synchronization delta Δt_S

From these formulas, we derive the synchronization process durations $\Delta t_{sync}^{\{qos0,qos1,qos2\}}$ for each QoS as follows:

$$\Delta t_{sync}^{qos0} = \begin{cases} 0, & \Delta t_S > 0 \text{ (ahead)} \\ 0, & \Delta t_S = 0 \text{ (synced)} \\ \Delta t_{sync_{TB}}, & \Delta t_S < 0 \text{ (behind)} \end{cases}$$

$$\Delta t_{sync}^{qos1} = \begin{cases} \Delta t_{N_{TB,SB}} + \Delta t_{sync_{SB}}, & \Delta t_S > 0 \text{ (ahead)} \\ 0, & \Delta t_S = 0 \text{ (synced)} \\ \Delta t_{sync_{TB}}, & \Delta t_S < 0 \text{ (behind)} \end{cases}$$

$$\Delta t_{sync}^{qos1} = \begin{cases} \Delta t_{N_{TB,SB}} + \Delta t_{sync_{SB}}, & \Delta t_S > 0 \text{ (ahead)} \\ 0, & \Delta t_S = 0 \text{ (synced)} \\ \Delta t_{sync_{TB}}, & \Delta t_S < 0 \text{ (behind)} \end{cases}$$

These time spans define the minimum time it takes to complete the migration process as we defined it in our work. Figure 6.9 visualizes our formulas.

Our bucket analysis shows the following results for the synchronization process time:

- *behind bucket*: $\Delta t_S < -120$
  - **QoS 0, QoS 1, QoS 2**: The synchronization process time lower bound is 120 ms and continuously increases.

- *synced bucket*: $-120 \leq \Delta t_S \leq 120$
  - **QoS 0, QoS 1, QoS 2**: The synchronization process time bound is between 0 and 120 ms.
- *ahead bucket*: $120 < \Delta t_S$
  - **QoS 0**: The synchronization process time is 0 ms.
  - **QoS 1, QoS 2**: The synchronization process time lower bound is 100 ms and continuously increases.

6.4.3 Migration Process Time

The duration of the complete migration process Δt_{mig} is defined as follows. It consists of the time to send a MIGRATE packet, to establish a connection with the Client, to perform the synchronization process and to send a MIGACK packet.

$$\Delta t_{mig} = \Delta t_{N_{SB,TB}} + \Delta t_{Ccon} + \Delta t_{sync_{TB}} + \Delta t_{N_{TB,SB}}$$

The minimum migration duration is reached, when no synchronization is necessary, $\Delta t_{sync_{TB}} = 0$.

$$\Delta t_{mig_{min}} = \Delta t_{N_{SB,TB}} + \Delta t_{Ccon} + \Delta t_{N_{TB,SB}}$$

Figure 6.10 visualizes the migration duration depending on the synchronization state Δt_S for each QoS. The migration process duration is the lowest for all QoS when the brokers are synced ($\Delta t_S = 0$) with $\Delta t_{mig_{min}} = 40 + (4 * 10) + 40 = 120$ ms. Otherwise, depending on the QoS, some time for the synchronization process is necessary. The farther the Target Broker is behind the Source Broker ($\Delta t_S < 0$), the longer the migration process will take for all QoS, since the Target Broker has to wait to be synced. If the Target Broker is ahead of the Source Broker ($\Delta t_S > 0$), the duration differentiates. For QoS 0 the migration process is at the minimum time, since lost messages are valid. For QoS 1 and QoS 2, the duration raises, since the Target Broker has to send MIGSYNC packet to the Source Broker in order to trigger sending missed messages. As long as messages can be served from the store on the Source Broker, the migration duration stays constant ($0 < \Delta t_S \leq 120$). If the Source Broker also has to wait for further messages, the migration duration raises ($\Delta t_S > 120$) for QoS 1 and QoS 2.

The migration process time bucket analysis similar to the buckets (see previous section) of the synchronization process, with a positive shift (+120 ms), since the calculation also includes the MIGRATE packet and the client connection time.

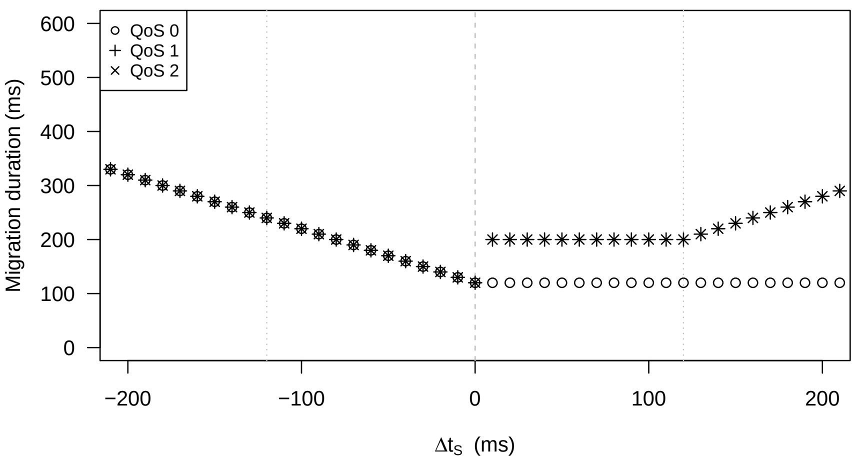


Figure 6.10: Complete migration process time for each QoS based on the synchronization delta Δt_S

Our bucket analysis shows the following results for the migration process time:

- *behind bucket*: $\Delta t_S < -120$
  - **QoS 0, QoS 1, QoS 2**: The migration process time lower bound is 240 ms and continuously increases.
- *synced bucket*: $-120 \leq \Delta t_S \leq 120$
  - **QoS 0, QoS 1, QoS 2**: The migration process time bound is between 120 and 240 ms.
- *ahead bucket*: $120 < \Delta t_S$
  - **QoS 0**: The migration process time is 120 ms.
  - **QoS 1, QoS 2**: The migration process time lower bound is 200 ms and continuously increases.

6.4.4 Loss, Duplication and Storage Time

By design, message loss is part of QoS 0, duplication of QoS 1 and storing messages is needed for QoS 2. In the following we are describing the time spans for each design characteristic. Figure 6.11 visualizes the durations for lost, duplicated and stored messages depending on the synchronization delta.

Message Loss for QoS 0

Message loss happens for QoS 0 if the Target Broker is ahead of the Source Broker ($\Delta t_S < 0$), since missed messages are not synchronized. In this case the time span of the synchronization delta related to the time span of lost messages. Otherwise, the Target Broker will continue sending, where the Source Broker left of and no messages get lost.

$$\Delta t_{qos0loss} = \begin{cases} \Delta t_S, & \Delta t_S > 0 \\ 0, & \text{otherwise} \end{cases}$$

Our bucket analysis shows the following results for the message loss time:

- *behind bucket*: $\Delta t_S < -120$
  The message loss time is 0 ms.
- *synced bucket*: $-120 \leq \Delta t_S \leq 120$
  The message loss time bound is between 0 and 100 ms.
- *ahead bucket*: $120 < \Delta t_S$
  The message loss time is lower bound is 120 ms and continuously increases.

Message Duplication for QoS 1

Message duplicates happen for QoS 1, since the Source Broker and the Target Broker are sending messages.

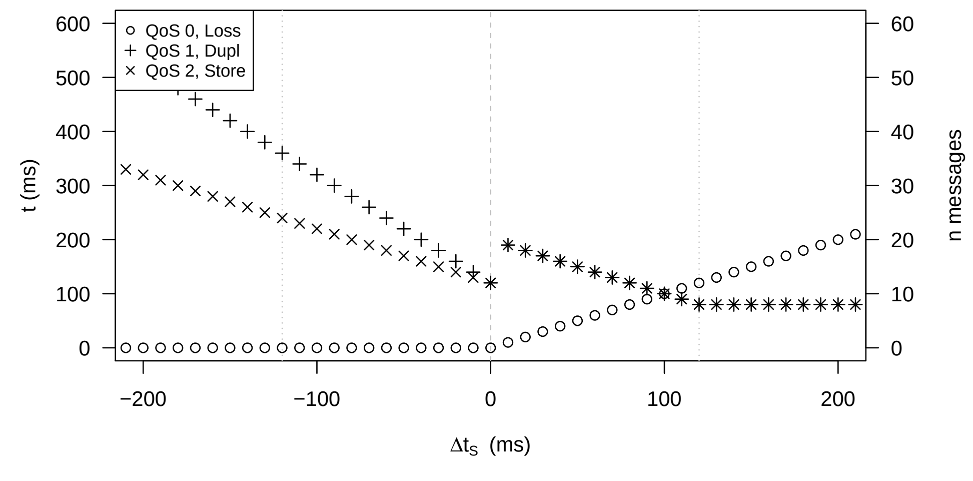


Figure 6.11: Time duration of lost, duplicate or stored messages for each QoS based on the synchronization delta Δt_S

Since the Source Broker is always sending messages until it receives the final migration acknowledgment (MIGACK), the duplication time span is at least the whole duration of the migration process Δt_{mig} if the brokers are synced. If the Target Broker is behind, it has to wait Δt_S until it reaches the synchronization point of the Source Broker. During this time span the Source Broker continues sending messages and therefore the duplication time span increases. In case the Target Broker is ahead of the Source Broker ($\Delta t_S > 0$), a MIGSYNC gets send and depending if the synchronization state is already passed or the still ahead on the Source Broker, at least in the time span for sending the MIGSYNCACK and MIGACK packets ($\Delta t_{N_{SB,TB}} + \Delta t_{N_{TB,SB}}$), duplicate messages will be sent.

$$\Delta t_{qos1dupl} = \begin{cases} max((t_{S_{TB}} + \Delta t_{N_{TB,SB}}) - \Delta t_S, 0) + \Delta t_{N_{SB,TB}} + \Delta t_{N_{TB,SB}}, & \Delta t_S > 0 \\ \Delta t_{mig} + abs(\Delta t_S) & \text{otherwise} \end{cases}$$

Our bucket analysis shows the following results for the message duplication time:

- *behind bucket*: $\Delta t_S < -120$
  The message duplication time lower bound is 360 ms and continuously increases.
- *synced bucket*: $-120 \leq \Delta t_S \leq 120$
  The message duplication time bound is between 80 and 360 ms.
- *ahead bucket*: $120 < \Delta t_S$
  The message duplication time is 80 ms.

Message Storage for QoS 2

Messages get stored for QoS 2 on the Source Broker for the whole migration process duration, Δt_{mig}. Nevertheless, if missed messages from the Target Broker need to be synced by the Source Broker in case the Target Broker is ahead of the Source Broker ($\Delta t_S > 0$), the requested messages get released or do not have to be stored, since they get send immediately. Only messages that do not have been delivered to the Client are stored. Therefore, the time span where messages actually are stored (and not released) is the migration process duration, Δt_{mig}, minus the synchronization delta Δt_S.

$$\Delta t_{qos2store} = \begin{cases} \Delta t_{mig} - \Delta t_S, & \Delta t_S > 0 \\ \Delta t_{mig}, & \text{otherwise} \end{cases}$$

Our bucket analysis shows the following results for the message storage time:

- *behind bucket*: $\Delta t_S < -120$
  The message storage time lower bound is 240 ms and continuously increases.

- *synced bucket*: $-120 \leq \Delta t_S \leq 120$
  The message storage time bound is between 80 and 240 ms.

- *ahead bucket*: $120 < \Delta t_S$
  The message storage time is 80 ms.

6.4.5 Message Arrival Shift

Δt_d is the trip time of a message from the publisher the subscriber. $\Delta t_{d_{SB}}$ is the trip time of a message through the Source Broker. $\Delta t_{d_{TB}}$ is the trip time of a message through the Target Broker.

In order to see when a specific message would arrive at the Client from a Source Broker or a Target Broker, we have to consider the arrival time of a message at the broker, $\Delta t_{P_{\{SB,TB\}}}$, and the time it takes to deliver it to the Client from each broker respectively, $\Delta t_{N_{\{SB,TB\},C}}$.

$$\Delta t_{d_{SB}} = \Delta t_{P_{SB}} + \Delta t_{N_{SB,C}}$$
$$\Delta t_{d_{TB}} = \Delta t_{P_{TB}} + \Delta t_{N_{TB,C}}$$

The delta between these durations defines the arrival delta, Δt_D, of a message for the migration, i.e. the message arrival shift in the movement from one broker to another:

$$\Delta t_D = \Delta t_{d_{TB}} - \Delta t_{d_{SB}}$$

If $\Delta t_D = 0$, messages arrive just-in-time. The client will not notice any delay.
If $\Delta t_D > 0$, messages arrive later.
If $\Delta t_D < 0$, messages arrive earlier.

6.4.6 First Message from Target Broker

The first message from the Target Broker $t_{firstMsg_{TB}}$ will be send immediately if the brokers are synced or the Target Broker is ahead of the Source Broker. If the Target Broker is behind the Source Broker, it will wait until it is synced for QoS 0 and QoS 2 and then send the first message to the Client, to avoid duplicate messages. The message will be received with $\Delta t_{N_{TB,S}}$ later at the Client.

$$t_{firstMsg_{TB}} = \begin{cases} t_{S_{TB}} + \Delta t_{sync_{TB}}, & \Delta t_S < 0 \text{ (behind) and } qos = 0,2 \\ t_{S_{TB}}, & \text{otherwise} \end{cases}$$

6.4.7 Message Load

Based on the timing analysis from the previous section we approximately can derive the amount of messages that are send, lost, duplicated or stored.

The constant production interval for messages on the producer(s), i.e. the time difference between two successive messages, is defined with t_v. The number of messages n that are processed during a time span t is calculated by dividing it through the production interval t_v:

$$n = \frac{t}{t_v}$$

With this formula we can calculate the number of lost messages with QoS 0, the number of duplicate messages with QoS 1 and the upper bound of messages that have to be stored with QoS 2:

$$\begin{aligned} n_{qos0loss} &= \frac{t_{qos0loss}}{t_v} \\ n_{qos1dupl} &= \frac{t_{qos1dupl}}{t_v} \\ n_{qos2store} &= \frac{t_{qos2store}}{t_v} \end{aligned}$$

These numbers based on the synchronization Δt_S are visualized in Figure 6.11 on the right axis.

6.4.8 Network Load

To compare the size of the packets of our migration protocol compared to the packets of MQTT, we list them in Table 6.5 and in Table 6.6.

There is potential to decrease the packet size of the Migration Protocol packets, since whole JSON objects as strings are sent (see Listing 8.4). We did not optimize the packet size, since the focus for our work was on the process itself. The number of migration packets for an individual migration process is at most six, compared to the number of PUBLISH packets varies for each application.

Since MQTT can run and our migration protocol runs over TCP/IP, additional ACK packets (66 bytes) are sent in between other packets to confirm that MQTT packets arrived.

Depending on the QoS level, MQTT is sending multiple packets in order to guarantee the delivery for a single message (see Table 6.7). This has to be considered while choosing the right QoS level for an application.

Table 6.5: MQTT messages size

Packet	**Size**	**Description**
CONNECT	96 bytes	Connect Command
CONACK	70 bytes	Connect Ack
PUBLISH	70 bytes[i]	Publish Message
PUBACK	74 bytes	Publish Ack
PUBREC	70 bytes	Publish Received
PUBREL	70 bytes	Publish Release
PUBCOMP	70 bytes	Publish Complete

[i] With our MqttPayload and content.

Table 6.6: Migration Process packets size

Packet	**Size**	**Description**
MIGRATE	935 bytes[ii]	Migrate
MIGACK	163 bytes	Migrate Ack
MIGTO	177 bytes	Migrate To
MIGTOACK	147 bytes	Migrate To Ack
MIGSYNC	336 bytes[iii]	Migration Syncronization
MIGSYNCACK	149 bytes	Migration Synchronization Ack

[ii] Size depends on the number of subscriptions (here: 6). [iii] Size depends on the number of subscriptions (here: 2).

Beside sending our migration packets, the network load during the migration process might decrease for QoS 0 subscriptions and it might increase for QoS 1 subscriptions. The load for QoS 2 subscriptions stays the same, since every message is delivered exactly once. Therefore, our solution approach has positive effects in an environment with many QoS 0 subscriptions and negative effects in an environment with many QoS 1 subscriptions.

Table 6.7: MQTT packets for each QoS

QoS	**n**	**Packets**	**Total size**
QoS 0	1	PUBLISH	70 bytes
QoS 1	2	PUBLISH, PUBACK	144 bytes
QoS 2	4	PUBLISH, PUBREC, PUBREL, PUBCOMP	280 bytes

6.5 Summary & Discussion

Our evaluation reveals the following results regarding the evaluation aspects:

- **Correctness**: The correctness was validated and verified with software testing and empirical experiments. All unit, integration and system tests that were performed are successful (Section 6.2). Furthermore, the empirical experiments result shows, that all message delivery are still valid through our migration process (Section 6.3.4). Therefore, we derive the correctness of our solution approach.
- **Responsiveness**: With empirical experiments and a theoretical analysis, we stage the responsiveness of our solution approach. Our experiment results (Section 6.3.4) reveal, that the migration process time depends on the network delays, amount of subscriptions of a client, the QoS for each subscription and the synchronization state of a subscriptions between the two brokers. When multiple subscriptions get migrated, the migration time depends synchronization process time that takes the longest. Furthermore, our theoretical analysis shows, that if the brokers within the *synched bucket*, the migration process time is minimal for all QoS (Section 6.4.3).
- **System Strains**: The system strains were analyzed with empirical experiments and an theoretical analysis. From our message load analysis (Section 6.4.7), we derive that with QoS 0 subscriptions, we can reduce the network load (discard messages), while we increase the network load with QoS 1 subscriptions (duplicate messages). With QoS 2 subscriptions the network load stays the same, since our solution approach ensures that no messages are lost or duplicated. Our bucket-analysis shows, that with QoS 0 subscriptions it is better when the Target Broker is synced or ahead, since the migration process can quickly finish (Section 6.4.4). With QoS 1 subscriptions it is better when the Target Broker is further ahead, since the number of duplicate messages will get minimized (Section 6.4.4). With QoS 2 subscriptions is is also better when the Target Broker is ahead, since the number of message that need to be stored is lower at its minimum. Therefore we derive generally, that our solution approach is better when the Target Broker is ahead.

The following sections summarize some benefits and limitations of our solution approach.

6.5.1 Benefits

We derive the following benefits from our solution approach:

- Use MQTT v3 to create a scalable (distributed) broker network.
- Enable client mobility between independent broker systems.
- During the migration process compared to common migration approaches[24]:
  - Some messages (QoS 0) are discarded to decrease the network load.
  - The Source Broker only has to store messages for QoS 2 .
  - Messages do not need to be transferred between the brokers.
  - Subscriber message arrival delay for QoS 1 messages is at its minimum since both brokers send their messages in parallel until they overlap.

6.5.2 Limitations

Applying our solution approach comes with certain limitations.

A higher network load for duplicate messages for QoS 1 subscriptions.

The following improvements can be implemented to reduce the network load for QoS 1 message duplicates:

Instead of sending QoS 1 messages on Target Broker immediately, the Target Broker can start sending them after the synchronization point from the Source Broker and to minimize duplicates.

If a QoS 1 subscriptions has to be synchronized with the Target Broker (MIGSYNC packet from the Target Broker), the Source Broker can stop sending QoS 1 messages after its synchronization process finishes. This way, the Source Broker can stop sending messages earlier compared to stop sending on MIGACK packet arrival. Nevertheless, messages that are not sent have to be stored in case of a rollback scenario.

The Source Broker could also get notified with a MIGSYNC packet from the Target Broker to stop sending messages for QoS 1 subscription, if the Target Broker is behind. Doing this, some duplicate messages could be omitted. If the Source Broker stops sending, it still has to store the messages in case of an error and a rollback.

[24]As stated in the our related work Section 3, common approaches treat all messages like QoS 2 (exactly once) subscriptions during migration and no message loss or duplication is considered or expected.

Additional migration packets on the network for the migration process itself.

Since at maximum six packets are send with meta information of the subscriptions, this limitation is minor. Nevertheless, instead of sending JSON objects, the migration protocol can be optimized into a binary protocol to reduce the packet sizes.

Processing time on brokers for the barriers and packet handling.

The integration of the migration framework needs some progressing resources. Since our work primary focused on creating a feasible solution and not a high-performing solution, there might be still potential to further improve the implementation of our migration framework and the integration into broker systems.

A topic can only be served by one publisher.

As made apparent in our assumptions (Section 4.2), our solution approach assumes that only one publisher publishes to a specific topic, to be able to derive a comparable sequence of messages. To solve this limitation is left for future work.

CHAPTER 7

Conclusion

Centralized publish/subscribe solutions with MQTT suffer from scalability issues. Since MQTT v3 was not designed with horizontal scalability and large scale systems in mind [MQT15], we extended MQTT v3 to support custom header information, e.g. for global message ids, as we need it for our solution approach to globally identify messages in a distributed broker network. As pointed out, MQTT v5 has introduced some enhancements for scalability and large scale systems in the meanwhile and solves this problem by introducing "user properties" which allows to add additional meta information to messages [MQT19].

When enabling mobility for clients in distributed broker networks, it is critical that systems operate reliably when migrating clients between brokers. Therefore, we designed a migration process that enables MQTT clients mobility within a distributed MQTT broker network. This migration process ensures the message delivery guarantees for client subscriptions while using special characteristics of MQTTs message delivery guarantees for message loss and duplication (Chapter 4). Distributed MQTT broker systems like EMMA [RND18], that only support client migrations with QoS 0, profit from our solution approach, since we consider QoS 0, QoS 1 and QoS 2 in our migration process.

In an iterative development process we implemented our solution approach as a framework and successfully integrated into the existing Moquette broker system and our MQTT client implementation (Chapter 5). We created a Broker CLI that acts as an API for external components and allows the Coordinator, like the Controller in EMMA [RND18] or in PubSubCoord [AKGH17], to trigger migration processes for clients in an orchestrated components composition. Our solution approach, can even be used to perform client migrations in inter-cloud settings, as in [CTVB12], with different broker implementations.

We evaluated our solution approach, our implementation and our integrations with software testing, empirical experiments and a theoretical analysis to show correctness, responsiveness and system strains of our work (Chapter 6).

- The correctness was successfully verified with unit, integration and system tests and with empirical experiments by validating the message delivery guarantees (message loss and duplication) for all subscriptions of the clients.

- The responsiveness evaluation of our solution approach revealed that the migration duration time as well as the total network load during the migration process highly depends on the synchronization state of the brokers that perform the migration, the network latency between systems, the amount of the client's subscriptions and its requested QoS configurations. Furthermore, our bucket-analysis showed, that the solution approach is ideal, when the brokers are synced or if the Target Broker is ahead. In a typical scenario, with a publisher close to the Target Broker, with a conservative message delay in the distributed network, with the Target Broker synced or slightly ahead of the Source Broker and with a message production of 10 messages/s, the migration takes about 500 ms, with no message loss, a 1000 ms time window with 10 duplicate messages for QoS 1 and 8 stored messages for QoS 2. In a similar scenario, with high speed networks with low latency, the number of duplicate and stored messages are close to zero.

- The system strain evaluation confirmed, that we decrease the network load, when message loss with QoS 0 is considered. Nevertheless, message duplication with QoS 1 has to be applied with care since it leads to a higher network traffic with our migration process. In total, our solution approach adds 4 to 6 migration packets (depending if additional message synchronization between the brokers is necessary or not) to the network load for each client migration and therefore has no significant effect on the network load.

Based on the assumption to have a comparable stream of messages for a topic, our solution approach has the limitation, that the distributed system has to provide a total ordering of messages for a single topic. This may be achieved with distributed locks on the topics or with other methods to linearize the message stream.

Even through our solution approach builds on MQTT and its message delivery guarantees, the concepts we presented and analysed are generalizable and could be applied to other protocols like AMQP as well, since it also provides message delivery guarantees.

7.1 Future Work

- **MQTT v5**: In the meanwhile, since we started working on this thesis, MQTT v5 got released and provides some additional features that our solution can make use of [MQT19]. One example is the feature of "user properties" as described in Section 4.5, to directly store the global message id in the header information of the MQTT PUBLISH packet without the need to split the payload into a custom header and body section.

- **Multiple Publishers for a Topic**: Since our solution approach relays on a total ordering of messages for a topic, we took the approach that just one publisher serves a topic, since the publisher generate the sequence number. Better approaches might be found, e.g. with distributed topic locks, in order to support that multiple publishers can publish to the same topic. Nevertheless, a single publisher can publish to multiple topics with our solution approach, since the synchronization process can handle gaps in the message stream.

- **Publisher Mobility**: As mentioned in the introduction, publisher mobility provides different challenges as shown in [MPJ05] and [MPDJ05] and therefore is left for future work. Intermediate buffering methods, as shown in [LCC^{+}15] and [RND18], might be used on the Client during the migration process to support publisher mobility as well.

- **Publisher QoS**: In our solutin approach, the QoS of the publisher is always sending its MQTT messages with QoS 2 (exactly once). Nevertheless, all MQTT QoS combinations as described in Section 2.6 should be considered to make the solution approach work with different publisher QoS configurations as well.

- **Optimizations**: Additional approaches, like Prefetching or Logging (Section 3), might help to further improve the responsiveness and system strains of our solution approach. Furthermore, some suggestions were presented in the discussion of the evaluation (Section 6.5), e.g. to stop sending duplicate messages for QoS 1 earlier in the process, to reduce the network load.

CHAPTER 8

Appendix

8.1 Communication Implementation - Netty Channel

```
class MigrationPacketSocketChannelChannelInitializer extends
    ChannelInitializer<SocketChannel> {

    private final MigrationPacketHandler requestHandler;

    public MigrationPacketSocketChannelChannelInitializer(
    MigrationPacketHandler requestHandler) {
        this.requestHandler = requestHandler;
    }

    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();

        // in (evaluation: down to top)
        pipeline.addLast(new StringEncoder());
        pipeline.addLast(new MigrationPacketToJsonEncoder());

        // out (evaluation: top to down)
        pipeline.addLast(new JsonObjectDecoder());
        pipeline.addLast(new StringDecoder());
        pipeline.addLast(new JsonToMigrationPacketDecoder());
        pipeline.addLast(new MigrationPacketInboundHandler(requestHandler));
    }
}
```

Listing 8.1: Migration server netty channel setup

8.2 System Test Example

```java
// Class: QoS0SourceScenarioTest
@Test
public void synced() {
    incomingAndAssertUnprocessed(12);
    incomingAndAssertUnprocessed(13);

    MigratePacket migratePacket = migrate();
    assertLastProcId(migratePacket, 13);

    assertEquals(MigrationState.SYNCED, migration.getState());
    assertEquals(MigrationSubscriptionState.SYNCED, subscription.getState());

    incomingAndAssertDiscard(14);
    incomingAndAssertDiscard(15);

    assertEquals(MigrationState.SYNCED, migration.getState());
    assertEquals(MigrationSubscriptionState.SYNCED, subscription.getState());

    incomingAndAssertDiscard(16);

    env.incomingPacket(new MigAckPacket(migration.getMigrationId(),
   MigAckPacket.MigStatus.OK));
    assertEquals(MigrationState.FINISHED, migration.getState());
    assertEquals(MigrationSubscriptionState.FINISHED, subscription.getState()
    );

    incomingAndAssertUnprocessed(17);
    incomingAndAssertUnprocessed(18);
    incomingAndAssertUnprocessed(19);
}
```

Listing 8.2: A system test scenario example: Source Broker, QoS 0, synced state

8.3 Migration Process Analysis Script

```r
calc <- function(qos,
                 # network
                 Dt_N_SBTB=50,
                 Dt_N_SBC=100,
                 Dt_N_TBC=50,
                 # message arrival delay
                 Dt_P_SB=0,
                 Dt_P_TB=0,
                 # production interval (every x ms)
                 t_v = 10
) {
  Dt_N <- Dt_N_SBC - Dt_N_TBC
  Dt_P <- Dt_P_SB - Dt_P_TB

```

```
Dt_Ccon <- 4 * Dt_N_TBC

t_S_SB <- 0
t_S_TB <- t_S_SB + Dt_N_SBTB + Dt_Ccon
Dt_S <- t_S_SB - t_S_TB + Dt_P
Dt_D <- Dt_S + Dt_N

Dt_sync_SB <- NA
if(Dt_S > 0 && qos != 0) {
  Dt_sync_SB <- max(Dt_S - (t_S_TB + Dt_N_SBTB), 0)
} else {
  Dt_sync_SB <- 0
}

Dt_sync_TB <- NA
if(Dt_S > 0) {
  if(qos == 0) {
    Dt_sync_TB <- 0
  } else {
    Dt_sync_TB <- Dt_N_SBTB + Dt_sync_SB + Dt_N_SBTB
  }
} else {
  Dt_sync_TB <- -Dt_S;
}

# synchronization process duration
Dt_Sync <- NA
if(Dt_S > 0) {
  # ahead
  if(qos == 0)
    Dt_Sync <- 0
  if(qos == 1)
    Dt_Sync <- Dt_N_SBTB + Dt_sync_SB
  if(qos == 2)
    Dt_Sync <- Dt_N_SBTB + Dt_sync_SB
} else if(Dt_S == 0) {
  # synced
  if(qos == 0)
    Dt_Sync <- 0
  if(qos == 1)
    Dt_Sync <- 0
  if(qos == 2)
    Dt_Sync <- 0
} else {
  # behind
  if(qos == 0)
    Dt_Sync <- Dt_sync_TB
  if(qos == 1)
    Dt_Sync <- Dt_sync_TB
  if(qos == 2)
    Dt_Sync <- Dt_sync_TB
}

```

```
  # migration process duration
  Dt_mig <- Dt_N_SBTB + Dt_Ccon + Dt_sync_TB + Dt_N_SBTB

  Dt_qos0_loss <- NA
  if(Dt_S > 0) {
    Dt_qos0_loss <- Dt_S
  } else {
    Dt_qos0_loss <- 0
  }
  n_qos0_loss <- Dt_qos0_loss / t_v

  Dt_qos1_dupl <- NA
  if(Dt_S > 0) {
    Dt_qos1_dupl <- max((t_S_TB + Dt_N_SBTB) - Dt_S, 0) + Dt_N_SBTB +
    Dt_N_SBTB
  } else {
    Dt_qos1_dupl <- Dt_mig + abs(Dt_S)
  }
  n_qos1_dupl <- Dt_qos1_dupl / t_v

  Dt_qos2_store <- NA
  if(Dt_S > 0) {
    Dt_qos2_store <- Dt_mig - Dt_S
  } else {
    Dt_qos2_store <- Dt_mig
  }
  n_qos2_store <- Dt_qos2_store / t_v

  return(list(
    Dt_mig,
    Dt_qos0_loss,
    n_qos0_loss,
    Dt_qos1_dupl,
    n_qos1_dupl,
    Dt_S,
    Dt_N,
    Dt_D,
    Dt_qos2_store,
    n_qos2_store,
    Dt_Sync
  ))
}
```

Listing 8.3: Migration time calculation

8.4 Migration Packets JSON

```
# MIGRATE packet
{"@class":".MigratePacket","migrationId":"bc3582fd-3018-48b0-99d4-7758cd76e960","clientId":"Subscriber_001","clientHost":"172.172.3.1","clientPort":1702,"subscriptions":[{"requestedQos":"EXACTLY_ONCE","topicName":"topicC","lastProcessedGlobalMessageId":{"globalMessageId":"topicC-86"}},{"requestedQos":"AT_MOST_ONCE","topicName":"topicD","lastProcessedGlobalMessageId":{"globalMessageId":"topicD-77"}},{"requestedQos":"AT_MOST_ONCE","topicName":"topicA","lastProcessedGlobalMessageId":{"globalMessageId":"topicA-87"}},{"requestedQos":"AT_LEAST_ONCE","topicName":"topicB","lastProcessedGlobalMessageId":{"globalMessageId":"topicB-87"}},{"requestedQos":"AT_LEAST_ONCE","topicName":"topicE","lastProcessedGlobalMessageId":{"globalMessageId":"topicE-77"}},{"requestedQos":"EXACTLY_ONCE","topicName":"topicF","lastProcessedGlobalMessageId":{"globalMessageId":"topicF-76"}}]}

# MIGTO packet
{"@class":".MigToPacket","migrationId":"bc3582fd-3018-48b0-99d4-7758cd76e960","host":"172.172.1.2","port":1883}

# MIGTOACK packet
{"@class":".MigToAckPacket","migrationId":"bc3582fd-3018-48b0-99d4-7758cd76e960"}

# MIGSYNC packet
{"@class":".MigSyncPacket","migrationId":"bc3582fd-3018-48b0-99d4-7758cd76e960","subscriptions":[{"topicName":"topicE","lastProcessedGlobalMessageId":{"globalMessageId":"topicE-90"}},{"topicName":"topicF","lastProcessedGlobalMessageId":{"globalMessageId":"topicF-90"}}]}

# MIGSYNCACK packet
{"@class":".MigSyncAckPacket","migrationId":"bc3582fd-3018-48b0-99d4-7758cd76e960"}

# MIGACK packet
{"@class":".MigAckPacket","migrationId":"bc3582fd-3018-48b0-99d4-7758cd76e960","statusCode":"OK"}
```

Listing 8.4: Migration Packets JSON

List of Figures

List of Tables

List of Listings

Bibliography

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[AKGH17] Kyoungho An, Shweta Khare, Aniruddha Gokhale, and Akram Hakiri. An Autonomous and Dynamic Coordination and Discovery Service for Wide-Area Peer-to-peer Publish/Subscribe. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17*, pages 239–248, 2017.

[AR02] Filipe Araujo and Luis Rodrigues. On QoS-aware publish-subscribe. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 511–515. IEEE, 2002.

[BCS12] Carsten Bormann, Angelo P. Castellani, and Zach Shelby. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[Bet99] S.M.A. Bettencourt. Next Century Challenges: Scalable Coordination in Sensor Networks. *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, (Section 4):263–270, 1999.

[BJD$^+$04] Ioana Burcea, Hans Arno Jacobsen, Eyal De Lara, Vinod Muthusamy, and Milenko Petrovic. Disconnected operation in publish/subscribe middleware. *Proceedings - 2004 IEEE International Conference on Mobile Data Management*, pages 39–50, 2004.

[BMCR14] Paolo Bellavista, Senior Member, Antonio Corradi, and Andrea Reale. Quality of Service in Wide Scale Publish – Subscribe Systems. *IEEE Communications Surveys & Tutorials*, 16(3):1591–1616, 2014.

[CCW03] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, 2003.

[CCWS03] Mauro Caporuscio, Antonio Carzaniga, Alexander L Wolf, and Ieee Computer Society. Design and Evaluation of a Support Service for Mobile , Wireless Publish / Subscribe Applications. 29(12):1059–1071, 2003.

[CDF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[Col14] Matteo Collina. Mosca Documentation - lib/server.js. http://www.mosca.io/docs/lib/server.js.html, 2014.

[CS05] Yuan Chen and Karsten Schwan. Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 354–374. Springer-Verlag New York, Inc., 2005.

[CTVB12] Rodrigo N. Calheiros, Adel Nadjaran Toosi, Christian Vecchiola, and Rajkumar Buyya. A coordinator for scaling elastic applications across multiple clouds. *Future Generation Computer Systems*, 28(8):1350–1362, 2012.

[Cug02] Gianpaolo Cugola. Using Publish / Subscribe Middleware for Mobile Systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[DDM$^+$09] Elisabetta Di Nitto, Daniel J. Dubois, Raffaela Mirandola, Elisabetta Nitto, Daniel J. Dubois, Raffaela Mirandola, and Politecnico Milano. Overlay self-organization for traffic reduction in multi-broker publish-subscribe systems. In *Proceedings of the 6th International Conference on Autonomic Computing*, pages 61–62. ACM, 2009.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EMQ17] EMQ broker 1.0 - Design - Architecture, Revision 7b021615. https://emqttd-docs.readthedocs.io/en/latest/design.html, 2017.

[Eug07] Patrick Eugster. Type-based publish/subscribe. *ACM Transactions on Programming Languages and Systems*, 29(1):6–es, 2007.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[GÖ05] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2005.

[GsGKK15] Julien Gascon-samson, Franz-philippe Garcia, Bettina Kemme, and Jörg Kienzle. Dynamoth : A Scalable Pub / Sub Middleware for Latency-Constrained Applications in the Cloud. 2015.

[GSKK17] Julien Gascon-Samson, Jorg Kienzle, and Bettina Kemme. MultiPub: Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe. *Proceedings - International Conference on Distributed Computing Systems*, pages 2075–2082, 2017.

[HGM04] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wireless Networks*, 10(6):643–652, 2004.

[HTSC07] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. MQTT-S - A publish/subscribe protocol for wireless sensor networks. *3rd IEEE/Create-Net International Conference on Communication System Software and Middleware, COMSWARE*, pages 791–798, 2007.

[IMA$^+$16] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of Docker as Edge computing platform. *ICOS 2015 - 2015 IEEE Conference on Open Systems*, pages 130–135, 2016.

[Jor14] JoramMQ, a distributed MQTT broker for the Internet of Things. http://www.scalagent.com/IMG/pdf/JoramMQ_MQTT_white_paper-v1-2.pdf, 2014.

[JS15] Antonio J Jara and Antonio F G Skarmeta. Mobility support for the small and smart Future Internet devices. 2015.

[KCJ06] Alex King, Yeung Cheung, and Hans-arno Jacobsen. Dynamic Load Balancing in Distributed Content-Based Publish / Subscribe. *Ifip International Federation For Information Processing*, pages 141–161, 2006.

[KKY$^+$10] Minkyong Kim, Kyriakos Karenos, Fan Ye, Johnathan Reason, Hui Lei, Konstantin Shagin, Yorktown Heights, Mt Carmel, Minkyong Kim, and Fan Ye. IBM Research Report Efficacy of Techniques for Responsiveness in a Wide-Area Publish / Subscribe System. 25058, 2010.

[KN01] Minkyong Kim and Brian Noble. Mobile network estimation. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, number October, pages 298–309. ACM, 2001.

[LCC$^+$15] Jorge E. Luzuriaga, Juan Carlos Cano, Carlos Calafate, Pietro Manzoni, Miguel Perez, and Pablo Boronat. Handling mobility in IoT applications using the MQTT protocol. *2015 Internet Technologies and Applications, ITA 2015 - Proceedings of the 6th International Conference*, pages 245–250, 2015.

[LKHJ13] Shinho Lee, Hyeonwoo Kim, Dong Kweon Hong, and Hongtaek Ju. Correlation analysis of MQTT loss and delay according to QoS level. *International Conference on Information Networking*, pages 714–717, 2013.

[LME$^+$15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, Etienne Riviere, Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing: Vision and Challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[MF05] Anton Michlmayr and P. Fenham. Integrating Distributed Object Transactions with Wide-Area Content-Based Publish/Subscribe Systems. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, 2005.

[MJ10] Vinod Muthusamy and Hans-arno Jacobsen. Mobility in Publish / Subscribe. In Laurence T. Yang, Agustinus Borgy Waluyo, Jianhua Ma Bala, Ling Tan, and Bala Srinivasan, editors, *Mobile Intelligence*, chapter Mobility i, pages 62–86. John Wiley & Sons, Inc., 2010.

[Mos18] Mosquitto mqtt bridge-usage and configuration. http://www.steves-internet-guide.com/mosquitto-bridge-configuration, 2018.

[MPDJ05] V. Muthusamy, M. Petrovic, Dapeng Gao, and H. Jacobsen. Publisher Mobility in Distributed Publish/Subscribe Systems. *Science*, pages 421–427, 2005.

[MPJ05] Vinod Muthusamy, Milenko Petrovic, and Hans-Arno Jacobsen. Effects of routing computations in content-based routing networks with mobile data sources. *Proceedings of the 11th annual international conference on Mobile computing and networking - MobiCom '05*, page 103, 2005.

[MQT15] MQTT Version 3.1.1 Specification. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, 2015.

[MQT19] MQTT Version 5.0 Specification. http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, 2019.

[Nai17] Nitin Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, 2017.

[PB02] P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. *Proceedings - International Conference on Distributed Computing Systems*, 2002-Janua:611–618, 2002.

[RD18] Thomas Rausch and Schahram Dustdar. Osmotic Message-Oriented Middleware for the Internet of Things. *IEEE Cloud Computing*, 5(2):17–25, 2018.

[RND18] Thomas Rausch, Stefan Nastic, and Schahram Dustdar. EMMA: Distributed QoS-aware MQTT middleware for edge computing applications. In *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, number i, pages 191–197, 2018.

[Sat15] Mahadev Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(June):30–39, 2015.

[SCA+05] Publish-subscribe Systems, Nuno Carvalho, Filipe Ara, Filipe Araujo, and Luis Rodrigues. Scalable QoS-Based Event Routing in. *Network Computing*, pages 101–108, 2005.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SDC16] Weisong Shi, Schahram Dustdar, and Edge Computing. The Promise of Edge Computing. *Computer*, 49(5):78–81, 2016.

[SGGK14] Priyanka Sharma, Mili Gupta, Gitanjali Goyal, and Kiranjit Kaur. On Quality-of-Service and Publish-Subscribe. *Journal of Evolution of Medical and Dental Sciences*, 3(9):2236–2244, 2014.

[SHB14] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (CoAP). Technical report, 2014.

[SMGJ09] Hu Songlin, Vinod Muthusamy, Li Guoli, and Hans Arno Jacobsen. Transactional mobility in distributed content-based publish/subscribe systems. *Proceedings - International Conference on Distributed Computing Systems*, pages 101–110, 2009.

[SS83] Dale Skeen and Michael Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, 1983.

[STO] Stomp protocol specification, version 1.2. http://stomp.github.io/stomp-specification-1.2.html.

[TMV+14] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee Xian Tan, and Colin Keng Yan Tan. Performance evaluation of MQTT and CoAP via a common middleware. *IEEE ISSNIP 2014 - 2014 IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Conference Proceedings*, (April):21–24, 2014.

[TP13] Ran Tao and Stefan Poslad. Delay sensitive distributed sensor data exchange for an IoT. *Proceedings of the International Workshop on Adaptive Security - ASPI '13*, pages 1–8, 2013.

[TSZ11] Nam Luc Tran, Sabri Skhiri, and Esteban Zimányi. EQS: An elastic and scalable message queue for the cloud. In *Proceedings - 2011 3rd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2011*, pages 391–398, 2011.

[Ver18] VerneMQ - Clustering. https://docs.vernemq.com/vernemq-clustering, 2018.

[VPGB] Luis Vargas, Lauri I W Pesonen, Ehud Gudes, and Jean Bacon. Transactions in Content-Based Publish / Subscribe Middleware.

[VRF$^+$16] Massimo Villari, Omer Rana, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[VT06] Steve Vinoski and Iona Technologies. Advanced Message Queuing Protocol. *Ieee Internet Computing*, (December):87–89, 2006.

[YKK$^+$09] Hao Yang, Minkyong Kim, Kyriakos Karenos, Fan Ye, and Hui Lei. Message-oriented middleware with QoS awareness. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5900 LNCS:331–345, 2009.

[Zer] Zeromq message transport protocol, verion 2.3. https://rfc.zeromq.org/spec:23/ZMTP/.

[Zha11] Lucy Zhang. Building Facebook Messenger. http://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920, 2011.

All URLs online; accessed on 3. March 2019.

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Diplomarbeitspräsentation

Enabling Mobility and Message Delivery Guarantees in Distributed MQTT Networks

DISTRIBUTED SYSTEMS GROUP

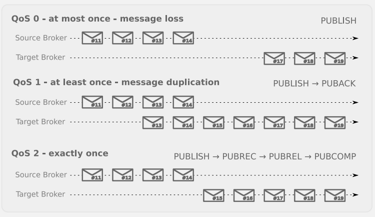
Masterstudium:
Software Engineering & Internet Computing

Manuel Geier

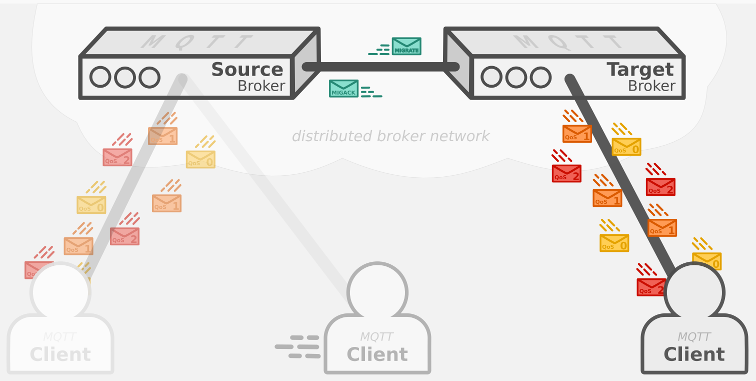
Technische Universität Wien
Institute of Information Systems Engineering
Arbeitsbereich: Distributed Systems
Betreuer: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Dipl.-Ing. Rausch Thomas, BSc

MOTIVATION

- Publish/subscribe with the MQTT protocol has become the de-facto standard for machine-to-machine communication in IoT.


QoS 0 - at most once - message loss
PUBLISH
Source Broker
Target Broker
QoS 1 - at least once - message duplication
PUBLISH → PUBACK
QoS 2 - exactly once
PUBLISH → PUBREC → PUBREL → PUBCOMP

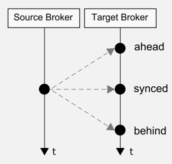
MQTTs Message Delivery Guarantees



- Distributed MQTT networks solve scalability issues, but introduce challenges for mobility of clients and messages delivery guarantees.

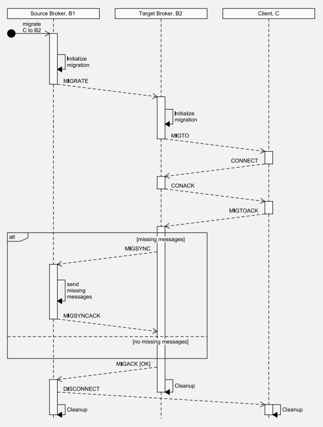
GOAL

- Design, implement and evaluate a migration process for publish/subscribe systems using MQTT while ensuring message delivery guarantees to enable mobility in distributed broker systems.



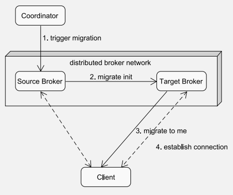
Synchronization State

SOLUTION APPROACH



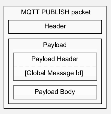
Migration Process

- Migration process that considers the characteristics of MQTTs QoS.
- Migration is triggered on Source Broker.



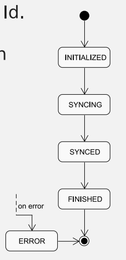
Migration Initialization

- Enhanced MQTT PUBLISH packet for meta information, e.g. Global Message Id.
- Distributed systems coordinate through the Migration Protocol.
- Migration state machines for internal migration process management.



MIGRATE
MIGSYNC
MIGTO
MIGACK
MIGSYNCACK
MIGTOACK

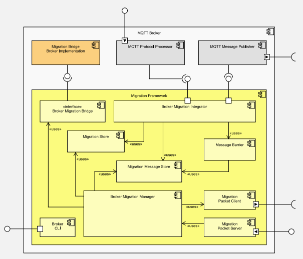
Migration Protocol Packets



Migration States

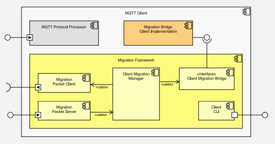
IMPLEMENTATION

- Creation of a Migration Framework in Java with an iterative software development process. It comprises of 3.8k LOC.



Broker Component Overview

- Integration of the Migration Framework into Moquette message broker[1].



Client Component Overview

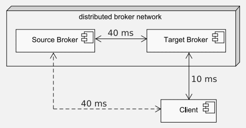
- Creation of a multi-broker MQTT client based on the Fusesource MQTT client[2].
- Integration of the Migration Framework into our client.
- API to trigger the migration.

[1]https://moquette-io.github.io/moquette
[2]https://github.com/fusesource/mqtt-client

Legend: ▭ Host ▭ Framework ▭ Implementation

EVALUATION

- Evaluation of *correctness, responsiveness* and *system strains* based on a common scenario.



Evaluation Scenario

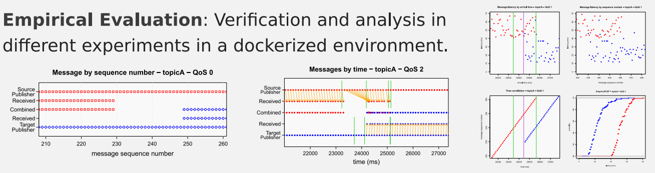
Network latency:
$\Delta t_{N_{SB,C}} = 40$
$\Delta t_{N_{SB,TB}} = 40$
$\Delta t_{N_{TB,C}} = 10$

Message arrival at brokers:
(Synchronization State)
$\Delta t_{P_{SB}} = 500$
$\Delta t_{P_{TB}} = -800, \dots, 800$

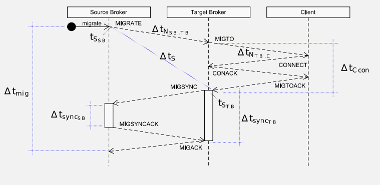
Production interval: $t_V = 25$

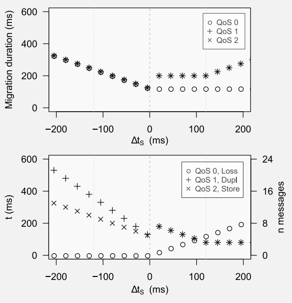
Evaluation Configuration

- **Software Testing**: 233 successful unit, integration and system tests.
- **Empirical Evaluation**: Verification and analysis in different experiments in a dockerized environment.



- **Theoretical Analysis**: Creation of a simplified mathematical model to analyize time and load responsiveness and system strains.





CONCLUSION

- Use MQTT v3 to create a scalable (distributed) broker network with client mobility.
- *Correctness* verified in our evaluation.
- *Responsiveness* and *system strains* depend on: amount of client's subscriptions, chosen QoS, network latencies, synchronization state.
- The network traffic decreases with QoS 0, it increases with QoS 1 and it is the same with QoS 2 subscriptions.
- Ideal migration process with a synced or ahead Target Broker.

Future Work
MQTT v5; Multiple Publishers for a topic; Publisher Mobility & QoS; Optimizations

https://**gitlab.com/manuelgeier-masterthesis**

Kontakt: **manuel@geier.io**