

A Study of the Capabilities of Message-Oriented Middleware Systems

by

Wael Al-Manasrah

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

© Wael Al-Manasrah 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This work was completed in collaboration with Zuhair AlSader and Ahmed Alquraan under the guidance of Prof. Samer Al-Kiswany and Prof. Tim Brecht. Wael established the research methodology, selected the systems we study, defined the set of features we study, and conducted an in-depth study of these features in the selected systems. Zuhair verified some of the features and findings of the study. Ahmed Alquraan helped with writing the Related Work chapter.

Abstract

We present a comprehensive characterization study of open-source [Message-Oriented Middleware \(MOM\)](#) systems. We devised a rigorous methodology to select and study 10 popular and diverse [MOM](#) systems. For each system, we examine 42 features with a total of 134 different options. We found that [MOM](#) systems have evolved to provide a framework for modern cloud applications through high flexibility and configurability and by offering core building blocks for complex applications including transaction support, active messaging, resource management, flow control, and native support for multi-tenancy. A key result of our study, is that we believe there is an opportunity for the community to consolidate its efforts on fewer open-source projects.

We have also created an annotated data set that makes it easy to verify our findings, which can also be used to help practitioners and developers determine and understand the features of different systems. For a wider impact, our data set is publicly available at [\[1\]](#).

Table of Contents

Author’s Declaration	ii
Statement of Contributions	iii
Abstract	iv
List of Figures	viii
List of Tables	ix
List of Abbreviations	xi
1 Introduction	1
2 Related Work	5
2.1 Surveys Studies	5
2.2 Benchmarking Studies	6
2.3 Summary	7
3 Methodology	8
3.1 Limitations	9
4 Message-Oriented Communication Model and Terminology	12
4.1 MOM Communication Characteristics	14

5	MOM Topologies	17
5.1	Single Broker Topology	17
5.2	Complete Mesh Topology	18
5.3	Flexible Topology	19
5.4	Brokerless Peer-to-Peer Topology	20
5.5	Summary	21
6	Reliability	22
6.1	Data Durability	22
6.2	Replication	24
6.3	Consumer Fault Tolerance	26
6.4	Subscription Recovery	27
6.5	Summary	28
7	MOM Service Semantics	29
7.1	Message Delivery Semantics	29
7.2	Handling Duplicate Messages	31
7.3	Transactions	32
7.4	Message Ordering	36
7.5	Summary	37
8	Client Interaction	38
8.1	Discovery Services	38
8.2	Access Methods	40
8.3	Dissemination Policies for the Unicast Exchange	43
8.4	Message Content	45
8.5	Filtering	46
8.6	Message Acknowledgments	47
8.7	Message Discard Policies	49
8.8	Multi-Tenancy	53
8.9	Summary	53

9	Resource Management and Flow Control	55
9.1	Resource Management	55
9.2	Flow Control	58
9.2.1	Credit-Based Flow Control	58
9.2.2	Rate-Based Flow Control	60
9.3	Summary	60
10	Implementation Details	61
10.1	Transport Protocols	61
10.2	MOM-Consumer Communication Modes	61
10.3	HTTP Interface	62
10.4	Security	63
10.5	Summary	64
11	Active Messaging	65
12	Messaging Protocols	67
13	Data Set	69
14	Discussion	72
15	Conclusions	74
	References	75
	APPENDICES	87

List of Figures

4.1	A single broker with a multicast exchange. Producers send messages to the “UW-news” topic. The broker multicasts the messages to all consumers.	13
4.2	A single broker hosting a logical topology with multiple exchanges. Arrows show the flow of messages.	13
4.3	Flexible topology. A logical topology deployed on two physical brokers located in two data centers.	14
4.4	A use case of load balancing a single multicast exchange.	16
5.1	Single broker topology example. The figure shows how Kafka replicates the partitions of a topic on multiple brokers.	19
5.2	Complete mesh topology. A topic is served by three multicast exchanges, each deployed on a separate broker. The brokers are connected in a complete mesh topology.	19
5.3	Peer-to-peer topology. Consumers connect directly to producers without an intermediate broker node.	21
13.1	An empty cell with a note explaining why Redis PubSub cannot support spilling messages to disk.	70
13.2	A screenshot showing a link to the cell showing RabbitMQ support of limiting disk space.	70
13.3	A screenshot of the web source detailing the proof of disk space limit in RabbitMQ. This web page can be accessed through the link shown in the previous figure.	71
13.4	A screenshot of a note showing the text quote of the proof that RabbitMQ supports disk space limit.	71

List of Tables

3.1	The short list of systems listed in descending order of the number of GitHub stars. E indicates that there is an enterprise version or support for the project. The Release Date is the date of the first open-source release of the project (sometimes a project starts as a propriety system and is later released as open-source). The shaded rows are the 10 systems we study in depth.	11
5.1	MOM systems' dissemination topologies.	18
6.1	Message persistence, granularity of persistence, and persistence frequency for each of the systems we study. J refers to a feature exclusively supported by NATS JetStream. Although Pulsar and Ejabberd can use different storage engines by using plugins, the default option is BookKeeper [2] for Pulsar and the Mnesia [3] database for Ejabberd. In this table, we report findings based on Pulsar and Ejabberd's default implementation.	23
6.2	Replication alternatives. Pulsar and Ejabberd allow plugging in different storage engines, the results in the table are based on the default implementation of BookKeeper in Pulsar, and the Mnesia database in Ejabberd. . .	25
6.3	The supported recovery guarantees in case of consumer failure.	26
7.1	Message delivery semantics. J refers to a feature exclusively supported by NATS JetStream.	30
7.2	Support for message deduplication. The table reports the systems that handle duplicate messages on the system side. For the consumer side, the table reports the systems that provide information to facilitate handling duplicate messages by the application.	32

7.3	Transaction properties, supported operations, and scope.	33
7.4	Supported message ordering semantics. J refers to a feature exclusively supported by NATS JetStream.	36
8.1	Supported discovery services and their consistency semantics.	39
8.2	The supported consumer access methods for message consumption. Each access method has a unique symbol that is used to correlate the access method with its granularity. J refers to a feature exclusively supported by NATS JetStream.	41
8.3	The supported dissemination policies per system. J refers to a feature exclusively supported by NATS JetStream.	43
8.4	The different types of message contents and attributes we observe in the 10 studied systems.	45
8.5	The supported filtering mechanism.	47
8.6	The supported message acknowledgment modes. NSQ does not offer acknowledgments to producers and Redis PubSub does not support consumer acknowledgments.	48
8.7	The supported retention policies, their granularity, and ways messages are discarded once a retention policy is violated. Each retention policy has a unique symbol that is used to show the granularity and the discard actions related to that policy.	50
9.1	The supported resource limits and the corresponding resource violation policies. The symbols match each resource limit to the violation policies applicable when the limit is violated. The network limit in NATS applies to both NATS Core and NATS JetStream.	56
9.2	The supported flow control mechanisms.	59
10.1	The supported transport protocols and MOM-consumer communication modes.	62
10.2	The operations supported through the HTTP interface.	63
10.3	The security measures supported in the 10 systems we studied.	64
12.1	The messaging standards supported by the systems we study. P means a standard is supported through a plug-in. J means that the standard is exclusively supported in NAT JetStream.	68

List of Abbreviations

ACID Atomicity, Consistency, Isolation, and Durability [32](#)

ACL Access Control List [64](#)

AMQP Advanced Message Queuing Protocol [62](#), [67](#), [68](#), [85](#)

API Application Programming Interface [2](#), [15](#), [24](#), [51](#), [63](#), [67](#)

CLI Command Line Interface [62](#)

FCFS First-Come-First-Served [13](#), [43–45](#)

IoT Internet of Things [1](#), [6](#)

JMS Java Message Service [28](#), [30](#), [46](#), [67](#), [68](#), [85](#)

MOM Message-Oriented Middleware [iv–vii](#), [ix](#), [x](#), [1–10](#), [12–15](#), [17](#), [18](#), [20–22](#), [24](#), [26](#), [28](#), [29](#), [31–33](#), [36](#), [37](#), [40](#), [44](#), [46](#), [47](#), [51](#), [53–55](#), [58](#), [60–62](#), [64](#), [65](#), [67](#), [72–74](#), [85](#)

MQTT Message Queuing Telemetry Transport [3](#), [16](#), [28](#), [30](#), [62](#), [67](#), [68](#), [85](#)

OOM Out-Of-Memory [56](#)

PAS Pending Acknowledgement State [35](#)

Pub/Sub Publish/Subscribe [1](#), [5](#), [6](#), [8](#)

RPC Remote Procedure Call [1](#), [14](#)

SSL Secure Sockets Layer [64](#)

STOMP Simple (or Streaming) Text Oriented Messaging Protocol [62](#), [67](#), [68](#), [86](#)

TC Transaction Coordinator [35](#)

TLS Transport Layer Security [64](#)

TTL Time-To-Live [51–53](#)

XMPP eXtensible Messaging and Presence Protocol [67](#), [68](#), [87](#)

Chapter 1

Introduction

Modern cloud applications are complex [4, 5, 6]. They integrate numerous subsystems and they are deployed on hundreds of nodes, increasingly across multiple data centers. Building cloud applications using low-level sockets or synchronous middleware systems such as [Remote Procedure Call \(RPC\)](#) is cumbersome and lacks critical features required by modern applications, such as reliability, asynchronous delivery, efficient multicast, and load balancing. Consequently, modern applications increasingly use higher-level [MOM](#) systems, also known as Messaging, Message Queuing, or [Publish/Subscribe \(Pub/Sub\)](#) systems.

[MOM](#) systems are widely used to support a broad range of modern applications including microservice-based services [7, 8, 9], streaming analytics [10, 11, 12], financial services [13, 14], [Internet of Things \(IoT\)](#) applications [15], video streaming [16, 17, 18], machine learning services [19, 20, 21], and blockchain frameworks [22].

At a minimum, [MOM](#) systems offer a simple communication abstraction. Producers produce messages to a certain topic hosted at a [MOM](#) server (i.e., broker). Brokers forward these messages to one or more consumers that have subscribed to that topic (i.e., subscribers).

Despite providing a similar communication abstraction, the community maintains tens of open-source [MOM](#) systems (Chapter 3), and new systems are frequently introduced. Many of these systems are extremely popular, such as Kafka [23], Pulsar [24], RabbitMQ [25], and ActiveMQ Artemis [26]. Additionally, many companies offer proprietary [MOM](#) systems [27, 28, 29], and major cloud providers also offer [MOM](#) services (e.g., Google Cloud [Pub/Sub](#) [30], Amazon Simple Queue Service [31], and Microsoft Azure Queues Storage [32]).

This raises a number of questions: What are the differences between the many open-source **MOM** systems? What are the techniques used to support scalability, reliability, and flow control? What are the semantics and policies for message delivery, persistence, and ordering? Do **MOM** systems offer a common subset of features? Does an opportunity exist to consolidate the community effort on a smaller subset of systems? Finally, what are the pressing open research challenges facing **MOM** systems? Answering these questions is important to guide research and development efforts, improve our understanding of the emerging capabilities of **MOM** systems and detail the trade-offs offered by different **MOM** systems, help developers choose the best system for a certain application, help practitioners understand the semantics of different systems and how to configure and maintain these systems appropriately, and help the community understand the capabilities of different systems and focus its effort on a fewer number of **MOM** systems.

To answer these questions, we conduct a comprehensive study of 10 popular **MOM** systems. For each system, we study 42 features. These features have 134 configuration and deployment options. For each system, we study its documentation, user **Application Programming Interface (API)**, and related posts on developer forums. In a few cases, we checked the source code to verify some of the characteristics, contacted the system developers, and deployed the systems.

We found that **MOM** systems are evolving and expanding the set of features far beyond simple message communication. First, **MOM** systems offer a framework for developing complex cloud applications with support for the durable storage of messages, transactions, application-specific processing, multi-tenancy, resource management, flow control, and configurable message delivery semantics. The majority of the systems we study support complex transactions that can produce messages to and consume messages from multiple topics. Recently, some of these systems have added active messaging capabilities in which applications can submit lambda-style functions to be executed on certain messages, which moves these **MOM** systems closer to a stream-processing engine or serverless-computing framework.

Second, a striking characteristic of **MOM** systems is their high flexibility and configurability. Some of the systems offer a flexible topology that allows optimization of the system deployment to match the underlying infrastructure or application access patterns. Many of the systems have multiple configuration options for most features we study allowing them to support applications with diverse semantics and performance requirements. We note that multiple systems offer the same features, with Pulsar, RabbitMQ, and ActiveMQ offering the largest number of configurable features. As a result, we believe it would be beneficial to consolidate the community efforts on fewer projects.

Reliability. Most of the systems we study support storing messages on durable storage. The majority of systems support replicating messages for higher reliability and scalability. Messages are stored with strong or eventual consistent semantics with the majority of systems offering eventually consistent storage semantics. The majority of systems also incorporate techniques to help consumers recover their state after a failure.

Service Semantics. We found that the systems we study adopt four delivery semantics: best-effort, at-most-once, at-least-once, and effectively-once. Interestingly, the majority of the systems we studied support transactions that can consume/produce messages from/to multiple topics. Nevertheless, many systems offer non-atomic transactions. We identified two message-ordering semantics: path ordering and multicast exchange ordering. Multicast exchange ordering guarantees in-order delivery of messages to consumers of a multicast exchange. Path ordering is a weaker model and guarantees that messages are delivered in order only if they take the same path in the system.

Client Interactions. All of the systems we study employ a discovery service to provide needed information for producers and consumers to access the system. The discovery service can either be strongly consistent or eventually consistent. The majority of systems use an eventually consistent discovery service. We identified five approaches to deliver messages to consumers. The majority of the systems offer multiple delivery approaches, with pushing messages to consumers being the most common because it incurs the lowest latency.

Resource Management and Flow Control. To control the use of available hardware resources, MOM systems can enforce limits on resource use, including specifying limits on processing, memory, network, and disk resources. The systems we studied also implement flow control mechanisms using two approaches: credit-based and rate-based. These techniques are the base for offering isolation for multi-tenant deployments.

Active Messaging. Interestingly, two systems have recently added support for active messaging, in which a user-defined function can consume and produce messages in the system. This capability allows MOM systems to support stream-processing applications and serverless-computing with stateful functions. This capability significantly increases system flexibility but introduces new scheduling, isolation, and resource management challenges.

Protocols. A number of standards that specify how to interact with MOM systems had already been developed. Our study (Chapter 12) shows that the majority of the systems we study build custom protocols and that the most popular standard, [Message Queuing Telemetry Transport \(MQTT\)](#), is supported by 5 of the 10 systems we study.

Implementation. We examined implementation details related to client interactions. MOM systems typically provide a client library to allow applications to access the system.

For message transfers, all [MOM](#) systems we study support TCP, whereas some systems also support UDP, Websocket, and IP Multicast. In addition, [MOM](#) systems offer standard mechanisms for authentication, access control, and encryption.

We have created a data set in which we annotate each characteristic of each system with a note and a link to the web source detailing the characteristic. As a service to the open-source [MOM](#) systems and research communities, our data set is publicly available [1]. This daunting task was performed to make it easier to verify our results and for researchers, practitioners, and application developers to find details related to each feature of each system.

The rest of this thesis is organized as follows. In Chapter 2, we discuss related work. We present our research methodology in Chapter 3. We present our abstract communication model used by [MOM](#) systems in Chapter 4 and discuss their dissemination topologies in Chapter 5. We describe our findings related to reliability in Chapter 6, service semantics in Chapter 7, client interaction in Chapter 8, resource management and flow control in Chapter 9, implementation aspects in Chapter 10, and active messaging in Chapter 11. We present an overview of standard messaging protocols in Chapter 12. We discuss our findings and open research problems in Chapter 14 and our conclusion in Chapter 15.

Chapter 2

Related Work

We identify two types of related work: surveys that study [MOM](#) systems and benchmarking studies that empirically compare [MOM](#) systems.

2.1 Surveys Studies

The work most related to our study is the 20-year-old survey conducted by Eugster et al. [33]. Their work focuses only on [Pub/Sub](#) systems and does not study any message queuing systems. In addition, some of the included systems are proprietary (e.g., IBM MQSeries [27] and TIBCO Rendezvous [34]), whereas other systems are deprecated and no longer supported (e.g., Gryphon [35]). Their work presents a high-level discussion of some features (e.g., reliability guarantees, delivery semantics, fault tolerance, and transaction support). However, they do not discuss multi-tenancy, resource management, flow control, or active messaging. Comparing our findings to those of Eugster et al., we find that [MOM](#) systems we study are more complex, provide richer sets of features, and offer many configuration options.

Several recent efforts survey applications of [MOM](#) systems. Sheltami et al. [36] conducted a survey of [Pub/Sub](#) middleware solutions for wireless sensor networks. Their paper first discusses the characteristics of wireless sensor applications that make [Pub/Sub](#) solutions the appropriate communication mechanism for these applications. Then, their paper surveys existing [Pub/Sub](#) solutions that specifically target wireless sensor applications, such as Mires [37], TinyCOPS [38], and TinyMQ [39]. For each of the surveyed systems, their paper provides a high-level discussion of various aspects of the system, including its

design and quality of service guarantees. Then, their work presents a reference model for **Pub/Sub** middleware for wireless sensor networks. The proposed model is extracted from the architecture of the surveyed systems. Finally, the authors of [36] discuss some research issues related to using **Pub/Sub** systems for wireless sensor networks such as dynamic distribution of the network load based on the remaining energy in the nodes in the network and efficient handling of the mobility of nodes in the network. Our work differs completely from their survey as we focus on general-purpose **MOM** systems that are more complex and provide more features compared to the systems studied in [36].

Razzaque et al. [40] present a detailed discussion of **IoT** characteristics and requirements and then evaluate existing middleware systems against these requirements. Their paper studies different types of middleware systems, including service-oriented middleware systems, virtual machine-based middleware systems, event-based middleware systems, and agent-based middleware systems. However, it does not focus on studying **MOM** systems. Their work surveys existing solutions for each type of middleware. Hence, their survey includes a large number of systems. However, it only provides a high-level summary of each system and does not discuss many of the features we study. Furthermore, none of the systems we study was included in any of the aforementioned surveys [36, 40]. Finally, Razzaque et al. conclude with a discussion of some research challenges of using middleware systems in **IoT**. These challenges related to dynamic resource discovery and management, scalability to the demand of **IoT**, and complex event management.

2.2 Benchmarking Studies

Ahuja et al. [41] recently conducted a performance characterization study of **MOM** systems. Their survey focuses on answering the following questions: What methodologies have been used for qualitative and quantitative evaluation of message-oriented middleware? How is benchmarking performed for message-oriented middleware, and which quantitative metrics are impactful? To answer these questions, their work surveys available benchmarks for **MOM** systems (e.g., SPECjms2007 [42], jms2009-PS [43], and OpenMessaging benchmark [44]) and it provides a comparison between these benchmarks across various factors including configurability, scalability, supported workload, and portability. Then, their work discusses various metrics that should be considered in the evaluation of a **MOM** system, including hardware utilization, throughput, and latency. Our work focuses on answering different questions related to the design of open-source **MOM** systems and their characteristics (Chapter 1). Moreover, their work [41] does not study many of the features we study in our work (e.g., transaction support, multi-tenancy, and active messaging).

A few recent efforts focus on quantitatively evaluating the performance of open-source **MOM** systems. Jain et al. [45] compare the performance of Apache Pulsar and NATS using OpenMessaging benchmark [44], whereas John et al. [46] compare the performance of Apache Kafka and RabbitMQ using Flotilla [47]. These two papers perform only empirical evaluation of these systems in terms of throughput and latency and do not provide an in-depth qualitative comparison. Specifically, Jain et al. [45] evaluate the impact of varying the payload size on the system throughput and latency. Their results show that NATS achieves higher throughput and lower latency compared to Pulsar as NATS does not store messages on disk. John et al. [46] measure the performance of Kafka and RabbitMQ when increasing the number of broker nodes and when increasing the number of clients (i.e., consumers and producers). Surprisingly, the performance of both systems suffers from degradation when increasing number of clients. The authors attribute this degradation to resource contention. Our work complements these efforts as we focus on studying the design of open-source **MOM** systems and qualitatively comparing their characteristics.

2.3 Summary

Our work differs significantly from the aforementioned papers. First, we follow a rigorous methodology (Chapter 3) to select open-source production-ready **MOM** systems to include in our study. Second, our study is comprehensive and detailed because it studies all aspects of **MOM** systems, including dissemination topologies, reliability, service semantics, client interaction, resource management, and active messaging.

Chapter 3

Methodology

We conduct an in-depth study of 10 diverse, widely popular, and general-purpose open-source **MOM** systems. The systems we study are highlighted in Table 3.1. They adopt different topologies, support a wide range of features, and are mature. We exclude proprietary **MOM** systems and cloud services such as Google Cloud Pub/Sub [30], Amazon Simple Queue service [31], IBM MQ [27], and Microsoft Azure Queue Storage [32].

We select the 10 systems we study using the following methodology. First, we use the GitHub search API [80] to search for open-source **MOM** systems. We use keywords such as “messaging,” “message queuing,” “broker,” “publish subscribe,” and “message bus.” Appendix 15 includes the full list of keywords we used. Using this approach, we found 57,768 GitHub projects in October 2021.

Second, we choose all projects that had 300 stars or more. Star counts are often used as an indicator of a project’s popularity [81]. This reduced the number of projects to 375. Third, we manually inspect the projects to identify those that build a **MOM** system. We exclude projects that build **MOM** applications or that are project specific (i.e., not general-purpose). This reduced the number of projects to 71.

Fourth, we exclude 21 projects because they are inactive. We consider a project to be inactive if the documentation states that the project is inactive or if there were no commits to the project in the prior year [82]. This reduced the number of projects to 50. Fifth, we exclude ten projects because they have limited documentation or the documentation is not in English. This left us with 40 projects.

Sixth, we inspect the remaining 40 projects and exclude 16 projects because they do not offer a readily-deployable **MOM** system but provide libraries for building custom messaging

applications (e.g., ZeroMQ [83] and Nanomsg [84]). We exclude 6 projects that implement a specialized MOM system for job scheduling. Table 3.1 lists the remaining MOM projects. Table 3.1 shows two rows for Redis because it offers two MOM systems, Redis Streams and Redis PubSub.

Seventh, we examine the systems listed in Table 3.1 to identify how they disseminate messages. We identified four common dissemination topologies (Chapter 5): peer-to-peer, single broker, mesh, and flexible topologies. For our in-depth analysis, we select the two projects that represent each topology with the largest number of stars. Although Redis Stream and Redis PubSub share a code base, we study them separately because they have different characteristics. ActiveMQ has two distributions: Classic and Artemis. ActiveMQ Classic is the older and most widely used distribution, nevertheless the community is working on phasing out Classic and replacing it with Artemis. As a result, we study the two ActiveMQ distributions.

The selected systems are widely popular and used in production by major services. For instance, Ejabberd is used by WhatsApp [85] and UbiSoft [86], Kafka by LinkedIn [87] and Lyft [88], RabbitMQ by SoundCloud [89], and Pulsar by Yahoo [90]. Although we study the open-source versions of these systems, most of the systems we examine have a proprietary enterprise (or supported) version.

For each of the 10 selected systems, we study their design documents, administrator and user manuals, tutorials, API specifications, developer blogs, and available publications. For some systems, the documentation was not complete, and we had to search the users and developers' forums, ask the system developers, and inspect the source code to determine detailed information about some characteristics.

Our process involved multiple iterations to learn about and understand the systems. In early iterations, we identify new characteristics or refine our definition of the characteristics we use to compare systems. In the end, we identified 42 distinct characteristics (described in Chapters 4 - 12).

3.1 Limitations

As with any in-depth study, the list of properties and system designs may not represent all available systems. Here, we list three potential sources of bias and describe our best efforts to address them.

- *Representativeness of the studied systems:* Although we study 10 diverse systems (highlighted in Table 3.1), our results may not generalize to systems we did not

study. To ensure the representativeness, we followed a rigorous methodology in short listing 19 popular, diverse, and general-purpose [MOM](#) systems.

- *Limiting the study to systems with 300+ stars:* Although we only considered projects with 300 stars or more, this choice does not impact the final set of selected systems. We choose the most popular system for each dissemination topology, consequently considering system with less than 300 stars would have not changed the set of projects we study.
- *Documentation accuracy and completeness:* Our team predominantly studied the publicly available sources of documentation of the selected systems. We did not study the complete source code nor deploy all the selected systems. Consequently, the accuracy of our findings inherently depends on the accuracy and completeness of the available documentation.
- *Observer error:* We study 42 characteristics in 10 systems. Some of the characteristics are not explicitly documented in the available documentation and require a careful reading to determine whether a system supports a specific characteristic. To reduce the chance of observer errors, two team members study each system independently following the same methodology, then compare their findings. If disagreement arose, the entire team discussed the characteristics in a group meeting before reaching a verdict.

Table 3.1: The short list of systems listed in descending order of the number of GitHub stars. E indicates that there is an enterprise version or support for the project. The Release Date is the date of the first open-source release of the project (sometimes a project starts as a propriety system and is later released as open-source). The shaded rows are the 10 systems we study in depth.

Project Name	License	Proprietary	Language	Version	Topologies			Release Date	
					Peer-to-Peer	Single	Mesh		Flexible
Redis Streams [48]	BSD	E [49]	C	6.2.5		✓	✓		May 2018
Redis PubSub [50]	BSD	E [49]	C	6.2.5		✓	✓		May 2010
NSQ [51]	MIT		Go	1.2.1	✓				Oct 2012
Kafka [23]	Apache	E [52]	Java	3.0		✓			Jan 2012
RocketMQ [53]	Apache	E [54]	Java	4.8.0		✓			Jan 2017
NATS ¹ [55]	Apache	E [56]	Go	2.6.1		✓	✓		Jun 2014 ²
Pulsar [24]	Apache	E [57]	Java	2.8.0		✓	✓ ³		Aug 2016
RabbitMQ [25]	MPL	E [58]	Erlang	3.8.17		✓	✓ ⁴	✓ ⁵	Aug 2007
EMQX [59]	Apache	E [60]	Erlang	4.3		✓	✓		Dec 2012
Mosquitto [61]	EPL/EDL	E [62]	C	2.0.12		✓			Dec 2009
Ejabberd (eCS) [63]	GPL	E [64]	Erlang	21.07	✓	✓	✓		Nov 2003
FAYE [65]	Apache		JavaScript	1.4.0		✓			Jun 2009
Emitter [66]	AGPL	E [67]	Go	2.8		✓	✓		Jun 2019
NCHAN [68]	MIT		C	1.2.12		✓			Nov 2009
VerneMQ [69]	Apache	E [70]	Erlang	1.12.3		✓	✓		May 2015
ActiveMQ Classic [71]	Apache	E [72]	Java	5.16.3	✓	✓	✓	✓ ⁶	Jan 2007
Aedes [73]	MIT	E [73]	JavaScript	0.46.1		✓			Mar 2015
ActiveMQ Artemis [26]	Apache	E [72]	Java	2.17.0	✓	✓	✓	✓ ⁷	May 2015
HiveMQ CE [74]	Apache	E [75]	Java	2021.2		✓			Apr 2019

- (1) NATS refers to both NATS Core and NATS JetStream.
- (2) NATS JetStream was first released in March 2021.
- (3) The mesh topology is created when using geo-replication.
- (4) A mesh topology is created when using non-exclusive queues.
- (5) Supported through the Federation [76] or the Shovel [77] plugins.
- (6) Supported through the concept of Virtual Destinations [78].
- (7) Supported by coupling Federation Addresses with Divert Bindings [79].

Chapter 4

Message-Oriented Communication Model and Terminology

One challenge for conducting an in-depth study of **MOM** systems is that the systems we study do not agree on which core features a **MOM** system should offer. Furthermore, the community does not use standard terminology for the various aspects of **MOM** systems. Worse yet, sometimes the same term can have different meanings in different systems. Please check Note 4 for further details.

We now present an abstract communication model that we use to capture the basic communication functionality offered by the majority of the systems we study. Given that no standard terminology exists for describing **MOM** systems, we also define the terminology we use in this work.

The communication model offers a communication abstraction for producers (i.e., senders or publishers) to send messages to one or more consumers (i.e., receivers or subscribers). A message carries application-specific data. **MOM** systems usually rely on a broker service to exchange messages. Producers send their messages to a broker service, which in turn forwards them to interested consumers.

MOM systems rely on a broker service to exchange messages with the exception of NSQ, which is a peer-to-peer system. Communication centers on topics (also referred to as *subjects*, *addresses*, and *channels*), which have unique IDs that can be human-readable names. Producers send messages to the broker service for a certain topic and consumers subscribe to that topic to receive messages. A typical application that uses a **MOM** system involves multiple producers and consumers that communicate through application-specific

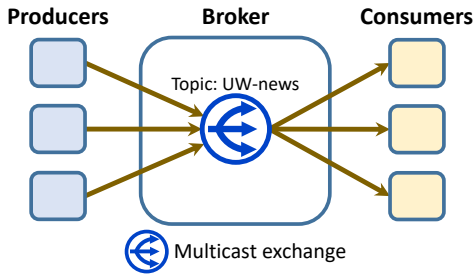


Figure 4.1: A single broker with a multicast exchange. Producers send messages to the “UW-news” topic. The broker multicasts the messages to all consumers.

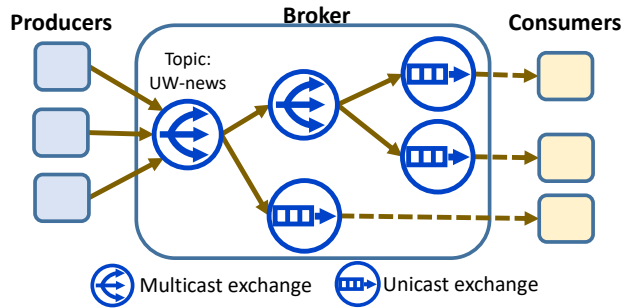


Figure 4.2: A single broker hosting a logical topology with multiple exchanges. Arrows show the flow of messages.

topics. Applications support multiple users. A user of the application may run multiple producers and consumers to produce and consume messages from different topics.

The broker hosts one or more logical *exchanges*. An exchange is either a multicast exchange, in which a message is delivered to all consumers that subscribe to that exchange, or a unicast exchange, in which a message is only delivered to one of the consumers subscribed to that exchange. Multiple consumers may subscribe to a unicast exchange. However, each message sent to a unicast exchange is delivered to only one of the subscribed consumers. MOM systems may use a range of policies to select a consumer among those that are subscribed to a unicast exchange. Examples of these policies include random, round robin, priority-based, and **First-Come-First-Served (FCFS)** policies. In Chapter 8.3, we detail the policies offered by the systems we study.

Figure 4.1 shows an example application that has multiple producers and consumers communicating through a single topic with a name “UW-news.” The broker has a single multicast exchange.

MOM systems (e.g., RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis) allow the use of multiple exchanges to form what we call a *logical dissemination graph* or *logical topology*. Figure 4.2 shows an example in which a single broker hosts a logical topology that uses two multicast exchanges and three unicast exchanges.

The logical topology can be deployed on one or more brokers. We refer to how a logical topology is placed on brokers as *physical layout*. Figure 4.3 shows the same logical topology in Figure 4.2 when it is laid out on two broker servers across two data centers.

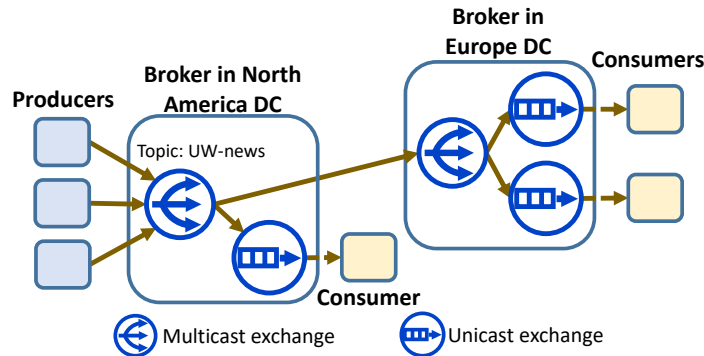


Figure 4.3: Flexible topology. A logical topology deployed on two physical brokers located in two data centers.

Note 4: Terminology in Real Systems

We note that different systems use different names to refer to the concepts presented in our model. In some cases, the same term has different meanings in two systems. For example, a multicast exchange is called a “topic” in NSQ, Kafka, RocketMQ, Pulsar, EMQX, Mosquitto, VerneMQ, ActiveMQ Classic, Aedes, and HiveMQ; “topic exchange” in RabbitMQ; “subject” in NATS; “stream” in Redis Streams; and “channel” in Redis PubSub, FAYE, Ejabberd, Emitter, and NCHAN. Interestingly, a “channel” in NSQ refers to a unicast exchange instead of a multicast exchange. A unicast exchange is called a “queue” in RabbitMQ and ActiveMQ Classic. ActiveMQ Artemis exchanges are called “addresses” that have an attribute that specifies whether an address is multicast or unicast.

4.1 MOM Communication Characteristics

The characteristics of the MOM communication model are drastically different from other popular communication middleware, such as raw TCP/IP-based communication and RPC (e.g., Linux RPC [91], Google gRPC [92], and Apache Thrift [93]). These characteristics make it attractive for modern cloud applications, including the following:

- *Space decoupling*: Producers and consumers do not need to be aware of each other. A producer does not have the addresses of the other producers or consumers, nor know how many of them are there. Similarly, consumers do not know the address or the number of other consumers and producers.

- *Time decoupling:* MOM systems often facilitate the time decoupling of producers and consumers, meaning the producers and consumers do not need to interact synchronously to communicate. Producers can send messages to a MOM system while a consumer is offline. The consumer will receive the messages when it joins the system, even if the producer is offline.
- *Flexibility:* MOM systems can be configured to provide a range of reliability, resource management, and delivery semantics. They can be deployed to match the underlying infrastructure to improve performance and scalability, such as placing brokers on central nodes or configuring a physical layout for geo-distributed setups to reduce WAN communication.
- *Separation of concerns and extensibility:* Producers and consumers communicate using a simple messaging API without being concerned with the way this API is implemented. Consequently, it is easy to extend the system design, change its implementation to add new features, or improve its performance without changing the implementation of the communicating parties.

Note 4.1: How our Communication Model is Mapped to Different Systems

To demonstrate how our model can be translated to deployments in the systems we study, consider a message replication use case in which a message should be processed by one of the database servers and one of the web servers (Figure 4.4). Producers send messages to the topic “TX” that is processed by a multicast exchange. The multicast exchange sends a copy of the message to the unicast exchanges of the database and web servers. Each unicast exchange forwards the request to one of the servers subscribed to that exchange. The systems we study use different approaches to implement this use case. We detail how this use case can be implemented in two systems: RabbitMQ and Redis Streams.

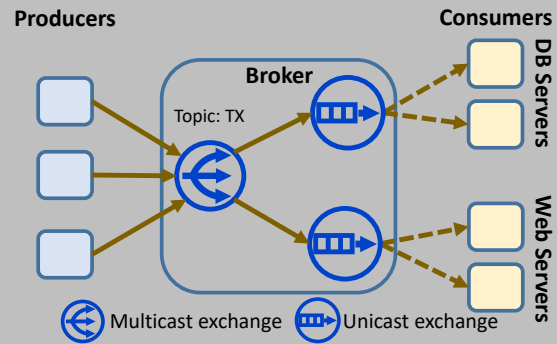


Figure 4.4: A use case of load balancing a single multicast exchange.

One can directly implement the described use case in RabbitMQ. A multicast exchange is called a *topic exchange*, while a unicast exchange is called a *queue*. Multiple consumers can subscribe to a queue. Messages sent to a queue are forwarded using one of the RabbitMQ policies listed in Table 8.3. NSQ can implement this use case in a similar way.

Redis Streams does not have explicit unicast exchanges. Each message produced to a multicast exchange (also known as a *stream*) is available to all consumers of that stream to request the message. The functionality of a unicast exchange can be achieved through a feature called *consumer groups*. Redis Streams supports grouping multiple consumers into a consumer group. A consumer group can have a single subscription to a multicast exchange, and messages sent to the group subscription are load-balanced among the group consumers. To implement the use case in Figure 4.4 using Redis Streams, one would define two consumer groups, one for database servers and one for web servers, and then subscribe each group to the multicast exchange serving the topic “TX.” Many systems implement the unicast exchange using this approach including Pulsar and NATS. Consumer groups are called *shared subscriptions* in Pulsar. We note that shared subscriptions became a standard feature in version 5.0 of the MQTT messaging protocol.

ActiveMQ Classic and ActiveMQ Artemis can implement this use case using a flexible dissemination topology (detailed in Chapter 5).

Chapter 5

MOM Topologies

MOM systems use a dissemination topology to transfer messages to consumers. We found that the systems we study support one or more of the following four dissemination typologies (Table 5.1): single broker, mesh, peer-to-peer, and flexible topology.

5.1 Single Broker Topology

A single broker is the simplest and most popular dissemination topology, in which a single node runs the MOM service (Figure 4.1). The broker hosts all topics, receives all messages, and serves all consumers. This topology is supported by all of the systems we study with the exception of NSQ, which is a brokerless peer-to-peer system (Table 5.1).

Although using a single broker server simplifies the MOM system design and implementation, it introduces a single point of failure. This is why systems adopting this single broker topology may replicate the state of the broker to backup brokers. For instance, Figure 5.1 shows how messages are replicated in Kafka, where topic partitions are replicated on multiple brokers with one of the brokers acting as the primary replica for a specific topic partition. Producers send messages to the primary broker of a given topic partition (also known as the *leader*). The primary broker can be configured to replicate every message to backup brokers before making it available to consumers to consume. The backup brokers do not serve consumers. One of the backup brokers takes over the primary role when the primary broker fails. Note that a backup broker may act as a primary broker for other topic partitions.

Table 5.1: MOM systems’ dissemination topologies.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Topology	Single	✓	✓		✓	✓	✓	✓	✓	✓	✓
	Mesh	✓	✓			✓	✓	✓	✓	✓	✓
	Flexible							✓P		✓	✓
	Peer-to-Peer			✓					✓	✓	✓

The main disadvantage of this design is that its scalability and performance are limited to a single server’s capacity, because a single broker serves all producer and consumer requests for a given topic. Furthermore, this topology is ill suited for multiple data center deployments.

5.2 Complete Mesh Topology

In a complete mesh topology, every broker is connected to all the brokers in the cluster. The same topic is served by all the brokers. The main goal of the mesh topology is to increase system scalability to improve system throughput to support a larger number of consumers. Figure 5.2 shows an example of a complete mesh topology with three brokers each hosting a multicast exchange that are serving the same topic, called “News.” As a result, a consumer can subscribe to any of the brokers. When a broker receives a new message from a producer, it forwards the message to all other brokers, which in turn forward this message to their local consumers, who are subscribed to that topic. RabbitMQ further optimizes this approach by only forwarding messages to brokers that have at least one consumer subscribed to that topic. This topology is supported by eight of the systems we study including: Redis Streams, Redis PubSub, NATS, Pulsar, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis (Table 5.1).

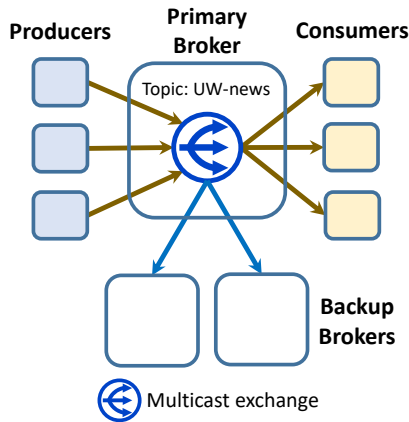


Figure 5.1: Single broker topology example. The figure shows how Kafka replicates the partitions of a topic on multiple brokers.

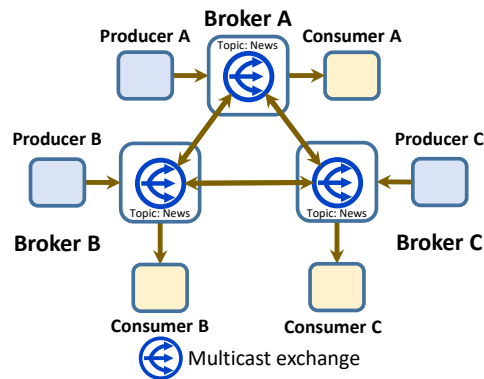


Figure 5.2: Complete mesh topology. A topic is served by three multicast exchanges, each deployed on a separate broker. The brokers are connected in a complete mesh topology.

5.3 Flexible Topology

RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis support building a logical message dissemination topology and flexibly deploying it on multiple brokers (e.g., Figure 4.3). Each broker can host a sub-graph of the logical topology. This is the most flexible design, and it can be configured to support single broker and complete mesh topologies.

Furthermore, this flexibility allows matching the dissemination topology with the underlying network within a data center or across data centers. This also helps support a wide range of application-specific topologies and allows systems to scale to support a large number of producers and consumers. This flexibility introduces additional steps for deploying these systems because it requires configuring and tuning the system for each new deployment.

Note 5.3: Flexible Topology Implementation in Real Systems

In RabbitMQ, flexibility is achieved through the *Federation* [76] or the *Shovel* [77] plugins. RabbitMQ has *topic exchanges*, which act as multicast exchanges, and *queues*, which act as unicast exchanges. The Federation plugin can forward messages between topic exchanges and queues, even if they reside on different brokers. Furthermore, multiple topic exchanges and queues can be used to create a variety of dissemination topologies. The Shovel plugin is similar to the Federation plugin, except it works on a lower level to consume messages from queues to topic exchanges.

In ActiveMQ Classic, a flexible topology can be created using *virtual destinations* [79]. A topic can be configured as a virtual destination and consume messages from other topics. This added flexibility allows the creation of flexible dissemination topologies.

In ActiveMQ Artemis, flexibility is achieved through the use of *federated addresses*. An address is a topic in our terminology. An address can be configured as a multicast or unicast address. Through the *divert bindings* feature [79], a multicast or unicast address subscribes to multiple multicast source addresses. Multiple addresses can be used to create a flexible dissemination topology.

5.4 Brokerless Peer-to-Peer Topology

This topology does not use dedicated nodes to run the broker service. Producers create topics on their local machine and consumers directly subscribe to these topics, without using an intermediate broker service (Figure 5.3). These systems use a discovery service to help consumers find the producers hosting a certain topic. If more than one producer is producing to the same topic, a consumer needs to independently connect and subscribe to each one of the producers. With this approach the consumer will receive all the messages sent to a given topic, but the order in which messages are received may differ between consumers of the same topic.

The main advantages of this topology are its high availability, low communication latency, and the elimination of dedicated broker nodes. It has high scalability for applications that are dominated by one-to-one communication. Consequently, this topology is often used in real-time communication applications such as instant messaging [85], gaming [86], VoIP calls [94], and video calls [95].

Unfortunately, this topology has serious limitations. First, this approach cannot scale to support hundreds of consumers that subscribe to the same producers of a topic. Second, this approach eliminates a number of important MOM communication characteristics

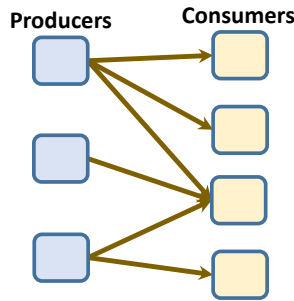


Figure 5.3: Peer-to-peer topology. Consumers connect directly to producers without an intermediate broker node.

(Chapter 4.1) including: time decoupling because producers and consumers communicate directly, space decoupling because producers and consumers know each other, and flexibility. Finally, this design limits the number of features that can be supported, such as replication and message ordering per multicast exchange.

NSQ is a purely peer-to-peer system. ActiveMQ Classic and ActiveMQ Artemis offer libraries that allow integrating the [MOM](#) service into producers and consumers to create a peer-to-peer system. Additionally, Ejabberd allows producers and consumers to establish direct peer-to-peer connections for lower latency.

5.5 Summary

In this chapter, we present the four message dissemination topologies that [MOM](#) systems use to transfer messages from producers to consumers. Among these topologies, the single broker topology is the most supported and simplest topology, but it introduces a single point of failure. Therefore, replicating the broker to replicas or using complete mesh topology instead is recommended for scalability and reliability. P2P-based [MOM](#) systems lack support of most of the [MOM](#) communication model characteristics (Chapter 4.1). Interestingly, some systems support building custom message dissemination graphs that can match with the underlying network infrastructure and support complex access patterns including combinations of single broker and complete mesh topologies. Finally, Table 5.1 shows that most systems support two or more topologies with ActiveMQ Classic and Artemis supporting all possible dissemination topologies.

Chapter 6

Reliability

An important consideration when designing, implementing, and deploying [MOM](#) systems is the service and data durability guarantees that they provide during failures. The [MOM](#) systems we study incorporate techniques to tolerate broker and consumer failures, including data durability ([Chapter 6.1](#)), replication ([Chapter 6.2](#)), handling messages of failed consumers ([Chapter 6.3](#)), and a consumer subscription recovery ([Chapter 6.4](#)).

6.1 Data Durability

The core mechanism to achieve data durability is to store messages on stable storage, through a local file system, a database, or a key-value storage system. [Table 6.1](#) lists the systems that provide data durability. We note that all systems support storing data to durable storage except Redis PubSub and NATS Core. These two systems are designed for best effort delivery where a message is removed once it is dispatched to consumers.

Kafka is the only system that stores every message to local storage and does not offer the ability to disable durable data storage. The rest of the systems include configuration options to disable storing data durably for all or a subset of messages.

The frequency of making messages durable. Storing a message to stable storage imposes an overhead. To avoid performing I/O operations on the critical path, many of the systems allow users to tune how frequently messages are synced to durable storage. Among the systems we study, we found four frequency configurations:

- *Immediate:* This configuration stores messages at the stable storage by calling `fsync()` after every publish request. An acknowledgment is sent to the producer only after

Table 6.1: Message persistence, granularity of persistence, and persistence frequency for each of the systems we study. J refers to a feature exclusively supported by NATS Jet-Stream. Although Pulsar and Ejabberd can use different storage engines by using plugins, the default option is BookKeeper [2] for Pulsar and the Mnesia [3] database for Ejabbered. In this table, we report findings based on Pulsar and Ejabberd’s default implementation.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Persistence	Persistent messages	✓		✓ ¹	✓	✓J	✓	✓	✓	✓	✓
	Non-persistent messages	✓	✓	✓		✓	✓	✓	✓	✓	✓
Persistence Granularity	System-wide	✓		✓	✓	✓J	✓	✓	✓	✓	✓
	Per Topic						✓	✓	✓ ²	✓	✓
	Per Message							✓		✓	✓
Persistence Frequency	Immediate	✓			✓					✓	✓
	Periodic	✓			✓	✓J	✓	✓	✓	✓	✓
	Batching				✓		✓	✓	✓		✓
	Operating System Persistence	✓		✓	✓	✓J	✓			✓	✓

- (1) By default, NSQ is an in-memory messaging system that persists messages to disk if the size of messages in memory exceeds a threshold. Setting this threshold to zero forces persisting messages to disk immediately.
- (2) Only when using Ejabberd PubSub module.

`fsync()` returns. This configuration offers the highest durability guarantee; however, it imposes the highest overhead because `fsync()` can incur significant overhead and it is called on the critical path.

- *Periodic*: Using this configuration, an `fsync()` is issued every preconfigured period of time. The shorter the period of time, the lower the chance of losing data and the higher the performance overhead. Using very long periods is not advised because it lowers the system reliability and introduces IO bursts.
- *Batching*: Messages are made persistent when the system accumulates a preconfigured number or total bytes of messages. Similar to the periodic configuration, the smaller the batch size the higher the system reliability and the higher the imposed

overhead.

- *OS Persistence*: This configuration leaves the decision to send messages to disk to the local file system. The [MOM](#) system writes the data to the file system without calling `fsync()` and the file system eventually sends the data to disk.

With the exception of NSQ, periodic durability is supported by all the systems that support storing messages durably. OS-based persistence is supported by all systems with the exception of RabbitMQ and Ejabberd, which use the internal Mnesia database for durability (Table 6.1). Redis Streams, Kafka, ActiveMQ Classic, and ActiveMQ Artemis are the only systems that support immediate durability. Finally, a number of systems including Kafka, Pulsar, RabbitMQ, Ejabberd, and ActiveMQ Artemis offer a hybrid approach that combines the periodic and batch-based policies. These systems persist a message if a certain period passes or the number of accumulated messages exceeds a configurable threshold.

Granularity of the Durability Configuration. The durability configuration can be set as a system-wide configuration, per topic, or per message. All the systems that support persistent messages can be configured to store every message produced (Table 6.1). Kafka is the only system that does not offer an option to disable storing messages on persistent storage; it only provides configuration options to tune the mechanism. Instead of storing all messages to disk, some of the systems offer two finer-granularity levels: topic and message.

Topic durability in Pulsar, RabbitMQ, Ejabberd PubSub, ActiveMQ Classic, and ActiveMQ Artemis allows for the configuration of a topic as either durable or transient. This granularity supports durably storing messages of a certain topic.

Message-level durability is the finest granularity offered for data durability. RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis allow one to specify the durability configuration per message. The producer [API](#) includes a parameter that indicates whether a given message should be durably stored.

In the systems that offer topic-level and message-level durability, if a transient message is sent to a durable topic, or if a message with a durability parameter set to true is produced to a non-durable topic, the message will not be durably stored.

6.2 Replication

Table 6.2 shows that all broker-based systems except Redis PubSub and NATS Core support message replication for reliability, or scalability to support larger deployments. Redis

Table 6.2: Replication alternatives. Pulsar and Ejabberd allow plugging in different storage engines, the results in the table are based on the default implementation of BookKeeper in Pulsar, and the Mnesia database in Ejabberd.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Replication	Supported	✓			✓	✓	✓ ¹	✓ ²	✓		✓
	Provided by storage engine						✓		✓	✓ ³	✓ ³
	Rack-Aware				✓		✓				

- (1) Supported through the geo-replication feature.
- (2) Exclusive to the special replicated queue which is called Quorum Queue.
- (3) Provided through a shared file system.

Streams, Kafka, NATS JetStream, Pulsar, RabbitMQ, Ejabberd, and ActiveMQ Artemis natively support message replication. Furthermore, Pulsar, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis can be configured to use external replicated storage. For higher reliability, Kafka and Pulsar can be configured to follow a rack-aware replica placement. Rack-aware placement aims to place replicas on different racks to tolerate rack failures.

Note 6.2: Replication implementation in RabbitMQ, NATS JetStream, and Kafka

RabbitMQ and NATS JetStream build replication techniques based on the Raft linearizable consensus protocol [96]. NATS JetStream creates multiple Raft instances. One global Raft instance is used to keep track of brokers in the system, one Raft group per topic is used to track brokers serving that topic, and one Raft group per topic is used for the consumers of a given topic.

Kafka builds its own replication protocol [23] and it partitions topics onto different brokers. For each partition, Kafka assigns one primary broker that serves all client requests. When the primary broker receives a new message, it replicates the message on backup brokers before serving it to consumers. Kafka can be configured to place replicas of a partition on different racks for higher reliability. Although immediate persistence to disk by calling `fsync()` after every publish request can be combined with replication on backup nodes, Kafka’s documentation discourages this configuration due to performance implications.

Table 6.3: The supported recovery guarantees in case of consumer failure.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Consumer Fault Tolerance	Reassigned by the Broker					✓	✓	✓		✓	✓
	Reassigned by a Consumer	✓			✓						
	Requeue as a new			✓							✓ ¹
Subscription Recovery for Multicast Only		✓			✓	✓	✓	✓ ²	✓ ³	✓	✓

- (1) Applies only to ring queues.
- (2) Using an exclusive queue or a durable subscription.
- (3) Supported when enabling the “mod_offline” or “mod_mam” modules.

6.3 Consumer Fault Tolerance

In **MOM** systems, consumers can be configured to send an acknowledgement when they finish processing the received message. This is the case for all the systems we study with the exception of Redis PubSub and NATS Core. If a consumer receives a message from a **MOM** system but it fails or its TCP connection drops before acknowledging it, then the **MOM** system assumes that the message has not been processed and it reassigns the message to another consumer.

We note that this fault tolerance technique is only provided for unicast exchanges, because each message sent to a unicast exchange is forwarded to one of the consumers who are subscribed to that exchange. If a consumer fails, the in-flight message(s) should be reassigned to another consumer. If no other consumer exists, the unicast exchange keeps the message(s) until a consumer connects to the exchange. This is not the case for multicast exchanges, which forward every message to all consumers subscribed to that multicast exchange. Table 6.3 lists the reassignment policies supported by the systems we study:

- *Reassigned by the broker*: In this policy, the broker reassigns the unacknowledged messages of a failed consumer to another consumer using one of the unicast exchange policies discussed in Chapter 8.3. This policy is supported by NATS JetStream, Pulsar, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis.

- *Reassigned by a consumer*: Kafka and Redis Streams allow an application to decide which consumer will claim the messages assigned to a failed consumer. In Kafka, a topic is divided into multiple partitions. A consumer claims complete partitions and processes all messages assigned to those partitions. When a consumer fails, its partitions need to be reassigned to other consumers. This reassignment can be done by the system or by the application. In the latter case, the application decides which consumer will claim the partitions of a failed consumer. In Redis Streams, consumers within the same consumer group who consume messages from a stream can claim each other’s unacknowledged messages. Consequently, if a consumer within the group fails, other consumers of that group can claim unacknowledged messages of the failed consumer. A second use of this technique in Redis Streams is to avoid slow consumers. A fast consumer can claim unacknowledged messages of a slow consumer. Unfortunately, this approach may lead to double processing of a message.

The reassignment of messages by consumers is also supported by Pulsar, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis but not for consumer fault tolerance. These systems place messages that a consumer could not process (e.g., for violating application semantics) into a “dead letter” topic. The application can use a consumer to inspect those messages.

- *Requeue as new*: Unacknowledged messages of a failed consumer are requeued as new messages without keeping track of the previous delivery attempts. The message is added to the tail of the unicast exchange and it can be forwarded to the same failed consumer if it rejoins the system. This policy is only supported by NSQ and ActiveMQ Artemis for the special ring queue.

6.4 Subscription Recovery

With the exception of Redis PubSub, NSQ, and NATS Core, all the systems we study offer a technique to recover old subscriptions if a consumer fails (Table 6.3). Recovering a subscription includes reinstating the subscription to its state before the crash.

Subscription recovery is provided only for multicast exchanges because a consumer of a multicast exchange should receive a copy of every message sent to that exchange. This recovery technique is not needed for unicast exchanges. A consumer of a unicast exchange might not receive all the messages sent to the exchange when multiple consumers exist, even when there are no failures. When a consumer of a unicast exchange fails and rejoins the cluster, it joins back as a new consumer.

Subscription recovery is achieved through one of two techniques:

- *Broker-side recovery*: The broker is tasked with storing the subscription state, such as consumer information and last consumed message ID. The broker then reinstates the subscription when a consumer rejoins the system. This approach is supported by Kafka, NATS JetStream, Pulsar, RabbitMQ, Ejabberd PubSub, ActiveMQ Classic, and ActiveMQ Artemis. We note that two popular messaging protocols, [Java Message Service \(JMS\)](#) v1.0.1+ and [MQTT](#) v3.1.1+, include specification for “Durable Subscriptions” that are reinstated when a consumer rejoins the system. We discuss messaging protocols in [Chapter 12](#).
- *Client-side recovery*: The consumer stores the subscription state in persistent storage. After rejoining the system, the consumer uses the stored state to resume consuming messages from the last message ID stored on disk. This approach is supported by Redis Streams.

6.5 Summary

Most of the [MOM](#) systems we study support storing messages on durable storage with configurable persistence frequency. Some systems offer fine message persistence granularity including message and topic level. Other than NSQ, all systems with support for data durability support replicating messages for higher reliability or scalability, with the majority relying on built-in replication mechanisms. Although some systems support strongly consistent storage, the majority of systems offers eventually consistent storage semantics. Finally, [MOM](#) systems also incorporate techniques to tolerate consumers’ failures. The majority of systems support recovering the subscription of a multicast exchange. All systems that support the load balancing of messages of a unicast exchange across multiple consumers support reassigning messages of failed consumers to another one.

Chapter 7

MOM Service Semantics

In this chapter, we discuss the semantics offered by the systems we study, including message delivery semantics, message deduplication semantics, transaction semantics, and message ordering semantics.

7.1 Message Delivery Semantics

We now describe the delivery semantics offered by the MOM systems we study. We provide details for the producer-to-broker and broker-to-consumer (or producer-to-consumer in Peer-to-Peer systems) delivery semantics. The systems we study offer four types of delivery semantics. Table 7.1 lists the semantics supported by each system.

- *Best-effort, at-most-once*: After sending a message, a sender does not wait for an acknowledgment from the receiver. If the message is lost due to a failure, then no retransmission occurs. We call this technique *best-effort, at-most-once* because a message is processed at-most-once, but will be lost in case of failures.
- *Reliable, at-most-once*: After sending a message, a sender waits for an acknowledgment from the receiver. Brokers can be configured to wait synchronously or asynchronously for an acknowledgment. Chapter 10.2 details the MOM-consumer communication mode.
- *At-least-once*: After sending a message, a sender waits for an acknowledgment from the receiver. If the sender times out before receiving an acknowledgment, the sender

Table 7.1: Message delivery semantics. J refers to a feature exclusively supported by NATS JetStream.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Producer Delivery Semantics	Best-effort at-most-once					✓ ¹		✓	✓	✓ ⁶	✓
	Reliable at-most-once	✓	✓		✓	✓J	✓ ³	✓	✓ ⁵	✓ ⁶	✓
	At-least-once				✓	✓J ¹	✓		✓ ⁵	✓ ⁶	✓
	Effectively-once	✓			✓	✓J	✓			✓	✓
Consumer Delivery Semantics	Best-effort at-most-once	✓	✓		✓	✓ ¹	✓ ⁴	✓	✓	✓	✓
	Reliable at-most-once	✓			✓	✓J ²		✓	✓ ⁵	✓	✓
	At-least-once			✓		✓J ¹	✓ ⁴		✓ ⁵		✓
	Effectively-once	✓ ⁷		✓ ⁷	✓ ⁷	✓J ⁷	✓ ⁷	✓ ⁷	✓ ⁷	✓ ⁷	✓ ⁷

- (1) In NATS JetStream with the [MQTT](#) Protocol, different delivery semantics are offered by different quality of service levels. QoS0 offers best-effort semantics and QoS1 offers at-least-once semantics.
- (2) Applies only to poll-based consumers.
- (3) In Pulsar, it depends on how the producer is implemented to handle acknowledgment time-out errors/exceptions.
- (4) The topic durability configuration (Chapter 6.1) imposes specific delivery semantics. A non-durable topic dispatches messages as best-effort at-most-once, whereas durable topics dispatch messages using at-least-once semantics.
- (5) Best-effort at-most-once is the default unless active stream management is enabled on the stream between the communicating parties.
- (6) Durable messages are always sent at-least-once and retried until successfully acknowledged. However, non-durable messages are instead sent as best-effort at-most-once. However, the client can make it reliable at-most-once by subscribing to [JMS](#) Exceptions.
- (7) The implementation of effectively-once semantics is done by the application code. We added a check for systems that support transactions or provide enough information in the message to facilitate implementing message deduplication.

retransmits the message. A message could be delivered more than once. If the broker does not receive an acknowledgment from a consumer after retrying a number of times, then the message is dropped or is added to a dead-letter exchange.

- *Effectively-once*: This is the strongest delivery guarantee. Similar to at-least-once semantics, a sender retries unacknowledged messages and may deliver a message more than once. However, message deduplication techniques or atomic transactions are used to eliminate duplicate messages. We discuss these two techniques in Chapters 7.2 and 7.3, respectively.

7.2 Handling Duplicate Messages

Messages could be retransmitted to achieve reliable delivery. This may lead to the duplicate delivery of the same message from a producer to a broker (or a consumer in case of a peer-to-peer system) or from a broker to a consumer. Here, we investigate how duplicate messages are handled both on the MOM system side and on the consumer side when the message is duplicated due to retransmissions. We do not study message duplication that is caused by the application design. For instance, an application may subscribe the same consumer twice to the same topic using two separate subscriptions, or it may use two wild-card filters that overlap (i.e., a message may match the two filters; we discuss message filtering in Chapter 8.5).

System side. Table 7.2 shows the systems that handle duplicate messages. It shows that Redis Streams, Kafka, NATS JetStream, Pulsar, ActiveMQ Classic, and ActiveMQ Artemis support the detection of duplicate messages on the broker. They identify a message as duplicate based on the message attributes, mainly message ID or sequence number. The IDs and sequence numbers of messages are generated by the producer. The pair <producer ID, message ID/sequence number> represents a unique message identifier.

Consumer side. Handling duplicate messages on the consumer side is implemented by the application and hence is outside the scope of our study. Table 7.2 shows the systems that provide enough information to facilitate the implementation of a technique to handle duplicate messages on the consumer side. We found that all of the systems we study, except Redis PubSub and NATS Core, embed enough information in messages to facilitate handling duplicate messages. Messages are uniquely identified using message ID or unique sequence numbers. Consumers can track this information to detect duplicate messages. We note that Redis PubSub and NATS Core are in-memory best-effort systems, where retransmission is not possible and hence there is no need to support deduplication.

Table 7.2: Support for message deduplication. The table reports the systems that handle duplicate messages on the system side. For the consumer side, the table reports the systems that provide information to facilitate handling duplicate messages by the application.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Handling Duplicate Messages	System side	✓ ¹			✓	✓	✓			✓ ²	✓
	Consumer side	✓		✓	✓	✓	✓	✓	✓	✓	✓

(1) If the producer creates an ID for a message, then the broker rejects messages with an ID smaller or equal to the last seen ID.

(2) By using the “Auditing” feature, the broker maintains a sliding window of messages to detect duplicates.

7.3 Transactions

Some [MOM](#) systems offer transactional support for applications. Table 7.3 shows the systems that support transactions, the transaction properties, the supported operations, and transaction scope. All systems except NSQ, NATS, and Ejabberd support transactions.

Transaction properties. We study the [Atomicity, Consistency, Isolation, and Durability \(ACID\)](#) properties [97] of the supported transactions. We note that unlike other properties we study, systems are not flexible and they provide a specific transaction support. For instance, a system can support either atomic or non-atomic transactions, but not both.

- *Atomicity:* Pulsar, ActiveMQ Classic, and ActiveMQ Artemis offer atomic transactions such that the operations carried out within a transaction are either executed entirely or not at all. Redis Streams, Redis PubSub, Kafka, and RabbitMQ offer non-atomic transactions. In these non-atomic transactions, partial results could be visible due to either system failures in RabbitMQ, lack of support for rollback of produced messages in Kafka, or failures of execution of some of the operations within a transaction in Redis Streams and Redis PubSub.
- *Consistency:* [MOM](#) systems do not check data consistency. [MOM](#) systems aim to be generic and able to support a wide range of applications. Consequently, [MOM](#)

Table 7.3: Transaction properties, supported operations, and scope.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Transaction Support		✓	✓		✓		✓	✓		✓	✓
Transaction Properties	Atomicity						✓			✓	✓
	Isolation	✓	✓				✓	✓		✓	✓
Transaction Operations	Produce	✓	✓		✓		✓	✓		✓	✓
	Consume & Acknowledge	✓			✓		✓	✓		✓	✓
	Reject							✓			
Transaction Scope	Single Topic	✓	✓		✓		✓	✓		✓	✓
	Multiple Topics	✓	✓		✓		✓	✓		✓	✓

systems do not check the schema of the message payload. Data consistency semantics are application-specific and are left to the application to handle.

- *Isolation*: **MOM** systems’ transactions offer classic transaction isolation, in which concurrent transactions are serializable. In other words, the intermediate results of a running transaction are not visible to other transactions until the transaction commits. Among systems that support transactions, Kafka is the only system that does not support isolation. In Kafka, produced messages are immediately stored in the log, messages from concurrent producers are mixed in the log, and the system does not support rolling back produced messages. Consequently, consumers outside a transaction may read messages of that transaction before it commits, or even after it aborts.
- *Durability*: Transaction durability ensures that the changes of a committed transaction remain in the system even in the case of failure. We note that systems that support transactions typically do not have a separate configuration option for transaction durability. Instead, transaction durability follows the durability configuration discussed in Chapter 6.1. For instance, if a transaction that uses durable messages is committed on a durable topic, then the transaction is durable. Furthermore, it is possible to have atomic but non-durable transactions.

Supported operations. Table 7.3 lists the operations that can be performed in a trans-

action. All systems that have transactions, except Redis PubSub, support both producing and consuming messages in a transaction. Redis PubSub is the only system that does not support message consumption in a transaction. In RabbitMQ, a consumer can receive a message and then decline to process it. RabbitMQ is the only system that allows the rejection of processing of messages and supports including the decision to reject a message as part of the transaction. Rejected messages are requeued to the topic, added to a dead-letter topic, or deleted.

Transaction scope. Interestingly, as shown in Table 7.3, all systems that support transactions allow transactions to access multiple topics. Therefore, an application can consume messages from one or more topics and produce messages to one or more topics in a single transaction.

Note 7.3: The design of Pulsar’s transactional support.

As an example, we detail the design of the transactional support in Pulsar. In Pulsar, a [Transaction Coordinator \(TC\)](#) module manages a transaction’s life cycle. The [TC](#) stores the transaction state in three data structures:

1. *Transaction log*: A transaction log is a log stored on stable storage that stores a transaction’s metadata. Pulsar implements the transaction log as a durable topic.
2. *Transaction buffers*: Each transaction has a buffer for each topic it uses to store messages produced within the transaction. Messages in a buffer are not visible to consumers until the transaction is committed.
3. *Pending Acknowledgement State (PAS)*: It is a log that stores any acknowledgments sent during a transaction. When a transaction commits, all acknowledgments stored in the [PAS](#) are sent. The [PAS](#) is stored on durable storage. The [TC](#) also uses the [PAS](#) to detect conflicts between transactions. If a transaction acknowledges a message that already has a pending acknowledgment in the [PAS](#) log, the transaction will abort.

Transaction Processing. To start a transaction, a client contacts the [TC](#). The [TC](#) logs the beginning of the transaction in the transaction log and assigns a unique ID to the transaction.

To produce a message, the client will first ask the [TC](#) to add the target topic to the transaction log. When the client produces messages, the broker stores the messages in a transaction buffer.

To acknowledge the consumption of a message, the client asks the [TC](#) to add the subscription to its log. The acknowledgments the client sends to a broker are stored in the [PAS](#) store.

To end a transaction, the client sends a request to the [TC](#) to commit or abort the transaction. If a transaction is aborted, the [TC](#) will log the decision, delete all transaction buffer(s), and delete its [PAS](#) store. If a transaction is committed, the [TC](#) logs the commit decision, moves messages from transaction buffer(s) to the actual topic(s) or topic(s) partition(s), and releases all acknowledgments stored in the [PAS](#). Finally, the [TC](#) acknowledges the commit operations to the client. After this point, an offline garbage collection mechanism truncates the transaction log.

Table 7.4: Supported message ordering semantics. J refers to a feature exclusively supported by NATS JetStream.

System	Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Message Ordering Guarantees	None	✓	✓			✓ ¹				
	Path ordering	✓		✓	✓	✓	✓	✓	✓	✓
	Multicast exchange	✓		✓	✓J	✓	✓		✓ ²	

- (1) Pulsar can be configured to allow out-of-order consumption for “Key_shared” subscriptions.
- (2) When the topology consists of a network of brokers, total ordering is not guaranteed.

7.4 Message Ordering

Each **MOM** system provides different guarantees for the order in which messages are delivered to consumers. The ordering semantics define the order in which messages are delivered to consumers differently for two cases: for messages that are produced by the same producer and for messages produced by multiple producers to the same multicast exchange. We found that systems may offer two kinds of ordering guarantees, which we call *path ordering* and *multicast exchange ordering*. Table 7.4 shows the message ordering semantics each of the systems support.

Path ordering. Path ordering means that if a producer sends two messages and these messages take the same path in the logical dissemination topology to the same consumer, then the order of these messages is preserved. Path ordering does not offer any ordering guarantees for multiple producers, even if the producers send messages to the same topic. For a unicast exchange, a consumer may not receive all the messages produced to that exchange. Path ordering guarantees that messages a consumer receives will be in the order the producer produced them.

Multicast exchange ordering. Multicast exchange ordering offers stronger ordering guarantees than path ordering. Multicast exchange ordering guarantees that consumers of the same multicast exchange will receive all messages produced to that exchange in the same order regardless of which producer produced the messages, or the path the messages took to the multicast exchange.

7.5 Summary

The **MOM** systems we study adopt four message delivery semantics: best-effort at-most-once, reliable at-most-once, at-least-once, and effectively-once. Additionally, some of the systems automatically detect and drop duplicate messages that result from message retransmission. These systems detect duplicate messages using an ID or a sequence number that the producer adds to each message upon production. The majority of the **MOM** systems support carrying out operations within a transaction that accesses multiple topics. Transaction atomicity and isolation properties are not flexible, whereas the durability property is flexible and follows the durability configuration of the system. Lastly, in terms of the message ordering semantics that **MOM** systems support, there are two possible alternatives: path ordering and multicast exchange ordering. Multicast exchange ordering guarantees that all consumers of a multicast exchange will observe messages in the exact same order. Whereas, path ordering guarantees that messages of the same producer are delivered in order only if they take the same path in the message dissemination graph.

Chapter 8

Client Interaction

In this chapter, we discuss the mechanisms through which a client interacts with the system. We detail configuration settings related to how clients discover services and produce and consume messages. These configurations specify how messages are disseminated, accessed, acknowledged, and discarded.

8.1 Discovery Services

A discovery service helps producers and consumers find information needed to access the system. At a minimum, this information is a list of brokers in the system. Some systems also help clients find which broker is serving a given topic. Additionally, the discovery service can also help with masking broker failures by pointing producers and consumers to alternative brokers.

Table 8.1 shows the discovery services of the systems we study and their consistency semantics. All of the systems we study offer a built-in discovery service. RabbitMQ and ActiveMQ Classic can also use an external discovery service. RabbitMQ can be configured to use AWS (EC2) [98] discovery service, Kubernetes [99], Consul [100], and etcd [101]. ActiveMQ Classic supports the use of a lightweight directory access protocol (LDAP) [102] server.

Discovery service consistency semantics. The consistency semantics of the discovery service specify when the service reflects a change in the system state. We found that the discovery services in the systems we study have either strong consistency guarantees or eventual consistency guarantees (Table 8.1):

Table 8.1: Supported discovery services and their consistency semantics.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Discovery Service	Built-in Service	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Add-On Service							✓ ¹		✓	
Consistency of Discovery Service	Eventually Consistent	✓	✓	✓		✓		✓ ¹	✓	✓	✓
	Strongly Consistent				✓		✓	✓ ¹			

(1) RabbitMQ can use an external system to provide a discovery service including Amazon AWS (EC2) discovery service, Kubernetes, Consul, and etcd. While all of them offer strong consistency semantics, Kubernetes semantics can be relaxed to offer eventual consistency.

- *Strongly consistent*: The system state is stored on a replicated and linearizable discovery service. This consistency guarantee is offered by Kafka, Pulsar, and RabbitMQ. Kafka and Pulsar use ZooKeeper, which offers a strongly consistent discovery service. RabbitMQ supports plugins for a discovery service. Among the supported plugins are Consul and etcd, which are based on the Raft consensus algorithm [96].
- *Eventually consistent*: In this consistency model, the discovery service might serve stale results about the state of the system, such as providing the address of a broker that is no longer a member of the system, or returning the address of an older broker for a certain topic. Table 8.1 shows that all the systems except Kafka and Pulsar offer eventually consistent discovery service.

Note 8.1: Discovery service design

Some of the systems we study document the design of their discovery services. Here, we present an overview of the design techniques used in these systems.

Strongly consistent discovery service. Kafka, Pulsar, and RabbitMQ provide strongly consistent discovery services that use a linearizable storage system such as Zookeeper or RAFT-based storage to maintain the system metadata.

Eventually consistent discovery service. MOM systems with an eventually consistent discovery service adopted one of two designs. Redis Streams, Redis PubSub, and ActiveMQ Classic can be configured to use an eventually consistent storage service to build a discovery service. Redis Stream and Redis PubSub use Redis Sentinel for their discovery service. Sentinel is a replicated and eventually consistent storage system. ActiveMQ Classic has a similar option in which it can use an external eventually consistent LDAP service [102].

NATS, ActiveMQ Classic, and ActiveMQ Artemis adopt a peer-to-peer approach to build a discovery service. The configuration file includes a list of seed brokers that clients and new brokers can contact to access the service. Nodes in the cluster periodically multicast information about newly joined or departed nodes. NATS uses a gossip-based approach to disseminate updates. In ActiveMQ Artemis, every broker creates a connection to every other broker in the system and uses these connections to send periodic updates. Alternatively, ActiveMQ Artemis brokers can multicast their updates using IP multicasting or a JGroup reliable all-to-all messaging service [103].

8.2 Access Methods

The studied systems offer the following approaches for consumers to obtain messages from the system. Table 8.2 shows the access methods supported by each of the systems we study.

- *Push*: The broker, or the producer in the case of the peer-to-peer NSQ system, pushes messages to consumers, and the consumer usually processes messages through an event listener. We found that all of the studied systems except Redis Streams and Kafka offer a push-based approach (Table 8.2). Furthermore, this is typically the default access mechanism in systems that support multiple access mechanisms. We note that this is the only access mechanism offered by Redis PubSub, NSQ, and NATS Core. This approach offers low latency and it is often used in latency-sensitive applications. Pushing messages may overwhelm consumers and lead to long delays

Table 8.2: The supported consumer access methods for message consumption. Each access method has a unique symbol that is used to correlate the access method with its granularity. J refers to a feature exclusively supported by NATS JetStream.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Consumer Access Method	Push (ρ)		ρ	ρ		ρ^1	ρ	ρ	ρ	ρ	ρ
	Pull (μ)	μ			μ					μ	
	Poll (o)	o			o	oJ		o	o^4	o	o
	Fetch (\odot)	\odot			\odot	$\odot J$	\odot^3		\odot^4	\odot^5	\odot^5
	Notify (ν)								ν		
Configuration Granularity	Request	$\mu o \odot$			$\mu o \odot$	$o^2 \odot$				$[\mu o]^6$	
	Consumer					ρo^2	\odot^3	ρo	o^4	$\rho[\mu o]^6 \odot$	$\rho o \odot$
	System		ρ	ρ		ρ^1	ρ		$[\rho \odot \nu]$		

- (1) The push access method is the only option in NATS Core.
- (2) After declaring a JetStream consumer to be poll-based, it sends poll and fetch requests.
- (3) A Pulsar consumer using the reader interface can fetch specific messages.
- (4) Fetch applies only for uploaded files and archived messages. Poll applies only if the recovered consumer chooses flexible offline message retrieval over flooding.
- (5) Can be done through creating a consumer that uses filtering to retrieve specific messages.
- (6) A consumer can be created to be push-based or poll/pull-based. Poll/pull-based consumers poll/pull on a per-request basis.

or message loss. Consequently, systems offering this approach often provide a flow control mechanism to create back pressure when the consumer is overloaded. We discuss the supported flow control mechanisms in Chapter 9.2.

- *Pull*: The consumer requests one or more messages from the broker and blocks until it receives the requested number of messages. Redis Streams, Kafka, and ActiveMQ Classic support this approach.
- *Poll*: The consumer requests one or more messages from the broker, but does not block indefinitely if no messages are available. This approach is supported by Redis Streams, Kafka, NATS JetStream, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis.
- *Fetch*: The consumer can selectively request previously processed messages. A message can be identified with a unique ID or an index in the log. Table 8.2 shows that Redis Streams, Kafka, NATS JetStream, Pulsar, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis offer this access mechanism. Pulsar offers special read-only consumers that are the only ones that can fetch older messages. In Ejabberd, fetch is the only access method for uploaded files and archived messages.
- *Notify*: A notification of a new message is pushed to the consumer, but the actual message is not forwarded until the consumer requests the message. Ejabberd is the only system that supports this approach for some modules.

Access Method Configuration Granularity. Table 8.2 shows the three granularities at which access methods are controlled:

- *Request*: A consumer specifies the access method in every request to retrieve messages. The same consumer can use different access methods for different requests. This is the only granularity available in Redis Streams and Kafka.
- *Consumer*: A consumer configuration specifies the access method, which usually happens upon the consumer's creation. NATS JetStream, Pulsar, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis support this approach. NATS JetStream offers the ability to configure consumers as push-based or poll-based consumers. NATS JetStream poll-based consumers can issue poll or fetch requests. A request-based consumer in ActiveMQ Classic can poll/pull messages depending on the request timeout parameter. In Ejabberd, recovered consumers can choose to poll missed messages to avoid being flooded if these messages were pushed by the server.

Table 8.3: The supported dissemination policies per system. J refers to a feature exclusively supported by NATS JetStream.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Multicast		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Unicast		✓		✓	✓	✓	✓	✓		✓	✓
Load Balancing Policy	Random			✓	✓	✓					
	Round Robin						✓	✓		✓	✓
	First-Come-First-Served (FCFS)	✓				✓ ^{J1}	✓ ²	✓ ¹		✓ ²	✓ ²
	Failover						✓	✓		✓	✓
	Consumer Priority						✓	✓		✓	✓
	Affinity Based				✓		✓			✓	✓

- (1) Applies only to poll-based consumers.
- (2) For push-based consumers, **FCFS** can be achieved by disabling buffering on the consumer side. This way, each consumer will consume one message at a time which achieves **FCFS** semantics.

- *System*: A system-wide configuration can control the access method used for all consumers and messages. Redis PubSub, NSQ, and NATS Core only support this configuration granularity. In Pulsar, push is the default system-wide access mechanism unless the consumer uses the reader interface to consume specific messages. In Ejabberd, push is the default system-wide access mechanism unless the system is configured to omit the inclusion of the message payload inside the notification. In addition, fetch is the only supported access mechanism if the message archiving is enabled.

8.3 Dissemination Policies for the Unicast Exchange

All of the systems we study except Redis PubSub and Ejabberd support multicasting and unicasting of messages. For systems that support unicast exchanges, if an exchange has multiple consumers, the next message will be sent to one of them. Table 8.3 shows the policies used to select a consumer in this case.

- *Random*: The broker, or the producer in the case of the NSQ peer-to-peer system, selects a destination to receive the next message, or subset of messages, using a uniform random distribution. NSQ and NATS readily support this policy. In Kafka, messages for a topic are divided into partitions and each partition can be assigned to a consumer. The default assignment of messages to partitions is done in a uniform random fashion.
- *Round Robin*: The broker selects a destination to receive the next message, or subset of messages, in a round robin fashion.
- *FCFS*: The next consumer to request or indicate that it is ready to receive message(s) will receive the message(s). This policy is often used by pull-based consumers in Redis Streams, NATS JetStream, and RabbitMQ. Alternatively, disabling buffering of messages on the consumer side for push-based consumers in Pulsar, ActiveMQ Classic, and ActiveMQ Artemis achieves this policy.
- *Failover*: The broker will pick one consumer to receive all messages sent to the unicast exchange. Should that consumer fail, the broker will pick another consumer to receive the messages. The selected consumer can be a consumer that subscribed to the same unicast exchange or a consumer sharing a multicast exchange subscription with the failed consumer.
- *Priority*: Pulsar, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis allow clients to assign priorities to consumers. Consumers buffer messages for processing. When a consumer's buffer is full, the broker stops sending messages to this consumer. When the consumer acknowledges some of the messages, the broker sends more messages to this consumer. When consumers have priority levels, the broker sends messages to the consumers with the highest priority until their buffers are full. Then, the broker selects lower priority consumers. If multiple consumers with the same priority level exist, then messages will be distributed using the default policy among the high-priority consumers. In all of the aforementioned systems, the default policy is round robin. Chapter 9.2 discusses consumer side buffering and flow control mechanism, in detail.
- *Affinity based*: A MOM system may allow messages to be grouped based on an ID, a key, or a hash of a subset of header fields. Following the affinity-based policy, the broker forwards messages that are part of the same group to the same consumer.

A use case that specially benefits from a variety of policies is scheduling. In this use case, producers send requests that are load balanced to multiple consumers (or servers

Table 8.4: The different types of message contents and attributes we observe in the 10 studied systems.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Message Content Type	Opaque		✓	✓		✓	✓	✓	✓	✓	✓
	Key-Value Pair(s)	✓			✓						
	Object									✓	✓
Message Headers	MIME-Type				✓			✓		✓	✓
	Map and List (Key-Value Sequence)						✓			✓	✓
	Message Priority							✓		✓	✓

in this scenario). Random, round robin, and **FCFS** policies try to utilize all consumers. Failover aims to use a single consumer to ensure in order processing of messages; once that consumer fails, another consumer is selected to replace it. An affinity-based policy can be used to send all messages produced by a producer, or share a grouping, to the same server.

The priority policy can be used to implement auto-scaling of cloud services. The messaging system sends all requests to a few high-priority servers. If the system receives too many requests, the system deploys additional low-priority servers and sends the additional requests to them. If the load decreases, the system shuts down low-priority servers.

8.4 Message Content

In this section, we describe our observations about the content of messages, including the payload and the metadata attributes carried in a message. Table 8.4 shows the content and metadata types supported by the systems we study.

Message Payload. Table 8.4 shows that the systems we study support three options for message payloads:

- *Opaque (Byte Stream)*: All of the systems we study, except Redis Streams and Kafka, support sending opaque messages that do not follow a specific data format.

- *Key-Value Pair*: Redis Streams and Kafka support sending the message payload as a key-value pair. A message may carry multiple key-value pairs.
- *Object*: ActiveMQ Classic and ActiveMQ Artemis support setting the message payload as an object as part of their support of the Java Message Service (JMS) standard. Nevertheless, the ActiveMQ documentation recommends against its use because it introduces coupling between producers and consumers.

Message Attribute. Table 8.4 shows that the systems we study support three options for message attributes:

- *MIME-Type Header*: Kafka, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis allow adding a media-type identifier (MIME-type) to the message payload.
- *Map and List (Key-Value Sequence)*: Pulsar, ActiveMQ Classic, and ActiveMQ Artemis allow the attribute format to be a sequence of key-value pairs. The key-value attributes can be application defined.
- *Message Priority*: RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis allow the producer to set a priority value for a message. Messages with higher priority are delivered before messages with lower priority. Consumer applications can also read the priority level of a message and use it to prioritize message processing or to filter messages based on their priority level. We noticed that in the systems that support message priorities, no limit exists on the number of messages a producer can designate as high-priority messages.

8.5 Filtering

MOM systems can filter messages and selectively forward them to a subset of the consumers. Table 8.5 lists the filtering options available in the studied systems:

- *Topic*: This is the standard filtering technique for all MOM systems. Producers produce messages to a topic and the messages are forwarded to consumers subscribed to the topic. To distribute the load among brokers, Kafka divides messages of a topic into partitions and assigns partitions to different brokers. Placing messages into partitions is based on the hashing of a key provided by the producer. This way a producer can make sure related messages are placed in the same partition. If the

Table 8.5: The supported filtering mechanism.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Filtering	Topic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Headers							✓		✓	✓
	Custom Content-Based						✓ ¹		✓ ²		

- (1) Can be implemented using Pulsar Functions.
- (2) Can be implemented using Ejabberd modules.

producer does not provide a key, messages will be randomly assigned to partitions. We note that the partitioning function can be overridden and replaced with a custom version. Pulsar can also divide messages of a topic into partitions.

- *Headers*: This strategy filters messages based on one or more fields in the message header. This filtering is supported by RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis (Table 8.5). ActiveMQ Classic and ActiveMQ Artemis offer an SQL-based selector on the header fields to select messages.
- *Custom Content-Based*: Pulsar and Ejabberd support application-specific custom filtering. Pulsar allows application developers to implement a custom filtering function to decide how messages are divided among the partitions.

8.6 Message Acknowledgments

The MOM systems we study support message acknowledgments in their interactions with producers and consumers. A broker can acknowledge messages to producers, and consumers can acknowledge messages to a broker (or a producer in the NSQ peer-to-peer system). An acknowledgment sent to the producer confirms that the broker has received the message, and in some cases has stored it on disk (e.g., ActiveMQ Classic and ActiveMQ Artemis) or replicated it on backup brokers (e.g., Kafka), but it does not typically indicate that a consumer has received the message. A consumer acknowledgment may indicate that the consumer has successfully received, processed, or stored the message.

Table 8.6: The supported message acknowledgment modes. NSQ does not offer acknowledgments to producers and Redis PubSub does not support consumer acknowledgments.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Producer Confirms	Controlled					✓ ²		✓	✓ ⁴	✓ ⁵	
Confirmation Mode	Single	✓	✓			✓ ²	✓	✓		✓	✓
	Cumulative							✓	✓ ⁴		
	Multiple										
	Batching				✓		✓ ³				
	Transactional						✓			✓	✓
Consumer Acknowledgments	Controlled	✓ ¹				✓		✓	✓ ⁴	✓	✓
Acknowledgment Mode	Single	✓ ¹		✓		✓	✓	✓		✓	✓
	Cumulative				✓	✓	✓	✓	✓ ⁴	✓	✓
	Multiple	✓ ¹								✓	
	Batching						✓ ³				
	Transactional						✓			✓	✓

- (1) Applies only to consumer groups. For regular consumers, no acknowledgment is supported.
- (2) Enabled only when the producer chooses to produce messages with the special JetStream Publish command and messages are then confirmed individually by default.
- (3) Messages that are produced as a batch must be acknowledged as a batch.
- (4) Supported only when active stream management is enabled on the stream between the communicating parties.
- (5) It depends on the durability of the produced message. Non-persistent messages are not confirmed; however, persistent messages must be confirmed.

The systems we study support a range of options for delivering these acknowledgments. Table 8.6 lists the options each system supports. The following options apply per topic (i.e., one can acknowledge multiple messages from the same topic, not across topics). Table 8.6 shows that all the systems other than NSQ support producer-side acknowledgments and all the systems except Redis PubSub and NATS Core support consumer-side acknowledgments.

- *Single*: Acknowledgments are sent per message. This is the most widely supported option. In NSQ, this is the only supported consumer acknowledgment option and it limits per-consumer throughput because the consumer has to send an acknowledgment for each message.
- *Cumulative*: Acknowledging a single message ID implicitly acknowledges all previous messages from the producer or broker with smaller message IDs.
- *Multiple*: Multiple, not-necessarily consecutive, messages are acknowledged in single acknowledgment reply. This mode is supported by Redis Streams for consumers that share a consumer group and by ActiveMQ Classic Optimized Acknowledgment Mode, where acknowledgments for a range of messages can be batched in a single operation.
- *Batch*: A group of messages is produced or consumed as a single unit and one acknowledgment is sent per batch. This is supported by Kafka and Pulsar on the producer side, and only by Pulsar on the consumer side.
- *Transaction*: In this mechanism, messages are acknowledged as part of an atomic transaction. Pulsar, ActiveMQ Classic, and ActiveMQ Artemis offer this option on the producer and consumer sides. We discuss support for transactions in Chapter 7.3.

Table 8.6 shows that most of the systems offer one or two acknowledgment modes, whereas Pulsar, ActiveMQ Classic, and ActiveMQ Artemis offer a wide range of acknowledgment options.

8.7 Message Discard Policies

Messages are generally discarded upon consumption, but they can be retained until some conditions are met. Redis Streams, NATS JetStream, and Ejabberd allow messages to

Table 8.7: The supported retention policies, their granularity, and ways messages are discarded once a retention policy is violated. Each retention policy has a unique symbol that is used to show the granularity and the discard actions related to that policy.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Discard Policy	Acknowledgment (α)			α		α	α	α	α^4	α	α
	Reject/Remove (ε)	ε				ε	ε	ε	ε		ε
	Delivery Count (μ)	μ					μ	μ^2		μ	μ
	Time-To-Live (τ)				τ	τ	τ	τ	τ	τ	τ
Policy Granularity	Per Message	ε				ε		$\varepsilon^3 \tau$	ε	τ	$\varepsilon \tau$
	Per Topic				τ	$\alpha \tau$		$\mu \tau$	τ^5	μ	μ
	Per Subscription						$\varepsilon^1 \mu$				
	Per Consumer	μ						$\alpha \varepsilon^3$	α^4	α	
	System-wide/Default			α			$\alpha \tau$		$\alpha^4 \tau^6$		α
Discard Actions	Completely Removed	ε		α	τ	$\alpha \varepsilon \tau$	$\alpha \varepsilon \tau$	$\alpha \varepsilon \mu \tau$	$\alpha \varepsilon \tau$	$\alpha \tau$	$\alpha \varepsilon \tau$
	Dead-letter	μ					μ	$\varepsilon \mu \tau$		μ	$\mu \tau$

- (1) Pulsar allows the removal of old messages in a subscription (unicast exchange, in our model) that are older than a given time.
- (2) Delivery count is only supported when using the replicated and durable “quorum queues.”
- (3) In RabbitMQ, message rejection is done per message but only by certain consumers that have the “manual acknowledgment” configuration option enabled.
- (4) Supported only when active stream management is enabled between the communicating parties.
- (5) Applies to PubSub topics. If a per topic value is not defined, then the system default applies.
- (6) Applies to the uploaded files.

be retained indefinitely. In this section, we discuss message discard policies. We do not discuss discarding messages due to resource limitations or flow control violations in this section, and we refer the reader to Chapter 9 for a detailed discussion of those issues.

We found that four message-discard policies are supported. Messages that satisfy the conditions in those policies can be deleted or moved to a special topic called “dead letter exchange.” These policies can be controlled through configuration options or [API](#) calls at different granularities. Table 8.7 shows the policies, granularities, and actions supported by the systems we study. In the table, each policy has a unique symbol. We use the same symbol to show the granularity and the discard action associated with each policy. Redis PubSub and NATS Core are the only systems that do not have a configurable message discard mechanism.

Discard Policy. The discard policy sets the conditions for discarding messages. Table 8.7 shows the discard policies supported by each of the systems we study.

- *Acknowledgment:* A message is deleted once it is acknowledged by the targeted consumers subscribed to that topic. This is the default policy in most [MOM](#) systems and it is supported by seven out of the 10 systems we study. This discard policy always results in removing the acknowledged messages. Pulsar further allows a time to be set to keep acknowledged messages in the system. For instance, an application can choose to keep processed messages for 24 hours after processing for auditing or debugging reasons.
- *Rejection and explicit removal:* A message is discarded when a consumer rejects the processing of a message, or a producer or consumer requests that a message be deleted. This policy usually leads to deleting the selected message.
- *Delivery counter:* If the broker does not receive an acknowledgment from a consumer, then the broker will try to redeliver the message. A “delivery counter” counts the number of times the broker tried to deliver the message. This policy will discard a message if the counter exceeds a configurable threshold. Redis Streams, Pulsar, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis support this policy.
- *Time to live:* A message is discarded once its [Time-To-Live \(TTL\)](#) timer expires. [TTL](#) is defined as a period (i.e., seconds or minutes). Kafka, NATS JetStream, Pulsar, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis support this policy.

Discard Actions. If a message meets the condition of one of the policies discussed above, then one of two action will be taken:

- *The message will be removed from the system:* The exact message violating the policy will be deleted from the system. This action is supported by all the systems that support discarding messages.
- *The message will be moved to a dead-letter exchange:* A message meeting a discard condition is moved to a special topic called dead-letter exchange. An application can choose to look into messages in the dead-letter exchange (e.g., for debugging purposes).

Discard Policy Granularity. The systems we study offer multiple granularities at which the discard policy is controlled.

- *Per message:* The discard policy is controlled per message. A consumer or producer can request the removal of a specific message or messages, or a producer can set per-message attributes to control the discard policy. For instance, in RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis, a producer can specify a [TTL](#) for the produced message. Once that [TTL](#) expires, the message is deleted or moved to a dead-letter exchange.
- *Per topic:* The discard policy is controlled per topic. For instance, a per-topic configuration can specify the [TTL](#) for each message produced to that topic in Kafka, NATS JetStream, and Ejabberd PubSub module. Additionally, the system configuration in RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis can specify the delivery counter for a topic. If a message in that topic exceeds this counter it is deleted or moved to a dead-letter exchange.
- *Per subscription:* Pulsar offers a configuration option to set a delivery count policy per subscription. A subscription in Pulsar is equivalent to a unicast exchange in our model (Chapter 4) in which multiple consumers can share a subscription.
- *Per consumer:* The discard policy is controlled by consumers. For instance, a message in Redis Streams contains its delivery count, and then it is up to the consumer to decide whether to process it or move it to a dead-letter exchange.
- *System-wide/default:* The discard policy is either applied as the default policy to all messages in the system, or controlled through a system-wide configuration.

Table 8.7 shows the policies, granularities, and actions supported by the systems we study. In the table, each policy has a unique symbol. We use the same symbol to show

the granularity and the discard action associated with each policy. Table 8.7 shows some interesting patterns: The **TTL** policy is typically applied per message or topic. Messages that exceed the delivery count are typically put into a dead-letter exchange. Acknowledged and explicitly removed messages are typically deleted from the system. Explicit removals are always issued per message except in the case of Pulsar in which a prefix of the messages log can be discarded in one request. Finally, we note that some systems allow the use of multiple discard policies simultaneously. If a message satisfies the conditions of one of the enabled policies, the message is discarded.

8.8 Multi-Tenancy

Some of the systems we study support multi-tenancy, which means that the system allows multiple applications to use the same **MOM** system instance. An essential aspect of supporting multi-tenancy is providing the illusion that the system is not shared with other applications. This requires isolating the applications such that one application does not affect the performance of others.

Kafka, NATS, Pulsar, RabbitMQ, and Ejabberd support multi-tenancy. These systems offer isolation between different tenants' configuration, data, and performance. Each tenant has its own topics, dissemination topology, and data. Each tenant also has its own configuration and authentication setup. Most importantly, these systems provide performance isolation through resource quotas and rate limits.

8.9 Summary

In order to interact with **MOM** systems, clients in all systems retrieve the information they need to access the system through a discovery service. Discovery services typically offer eventually consistent semantics. Then, consumers receive messages from the **MOM** system using one of five approaches, with pushing messages to consumers being the most common. When multiple consumers subscribe to a unicast exchange, the exchange load balances the messages across the consumers using one of six dissemination policies, with the majority of systems supporting multiple alternatives. The payload of the messages exchanged through a **MOM** system is mostly an opaque stream of bytes, with some systems allowing application-specific attributes to accompany the payload. Additionally, some systems allowing consumers to selectively filter messages based on the value of the attribute headers. Moreover, client interaction with **MOM** systems usually requires acknowledging

the delivery of produced and consumed messages to discard any state the producer or broker is maintaining about the sent messages. [MOM](#) systems support various options to deliver message acknowledgments. Nevertheless, [MOM](#) systems retain the produced messages until some discard condition is met, with Redis Streams, NATS JetStream, and Ejabberd allowing indefinite retention of messages. Lastly, some systems support true multi-tenancy by isolating tenants' data, configuration, and resource usage.

Chapter 9

Resource Management and Flow Control

In this chapter, we discuss the implemented mechanisms for resource management and flow control.

9.1 Resource Management

In this section, we discuss the limits [MOM](#) systems offer to control the resource usage and actions the systems take when these limits are violated. [Table 9.1](#) shows the limits and violation policies. [Table 9.1](#) uses symbols to match each resource limit to the violation policies applicable when the limit is violated.

Resource Limits. [Table 9.1](#) shows the resources that can be limited. We restrict our discussion to limits that are configurable (i.e., the user can set the resource limits). We note that the following resource limits are per [MOM](#) node (i.e., a violation of the limits by an application on one node does not affect the application on another node within the same [MOM](#) deployment).

- *Rate limit:* Only Kafka supports a rate limit ([Table 9.1](#)). The Kafka configuration can be used to set a “request rate quota,” which represents a percentage of the total processing and networking threads in the broker within a period of time. For instance, a 50% quota allows a client to use 50% of the Kafka broker threads in a window of time. This quota can be configured per client or for groups of clients and it could be configured with different values on different brokers in the cluster.

Table 9.1: The supported resource limits and the corresponding resource violation policies. The symbols match each resource limit to the violation policies applicable when the limit is violated. The network limit in NATS applies to both NATS Core and NATS JetStream.

System	Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis		
Resource Limits	Rate Limit (ε)			ε								
	Disk Space Limit (δ)				δ	δ	δ		δ	δ		
	Memory Limit (μ)	μ	μ		μ		μ	μ	μ	μ		
	Network Limit (κ)	κ	κ	κ	κ^1	κ	κ^2	κ^3	κ^3	κ	κ^3	
	Topic Size Limit (σ)	σ		σ	σ	σ	σ	σ		σ	σ	
Violation Policy	Spill to Disk			σ			$\mu\sigma$		$\mu\sigma$	$\mu\sigma$		
	Block Client(s)		μ	μ	$\varepsilon\kappa$	$\delta\mu\sigma$	δ	$\delta\mu\sigma$	μ^4	$\delta\mu\sigma$	$\delta\mu\sigma$	
	Notify		$\mu\kappa$	$\mu\kappa$	κ	$\varepsilon\kappa$	$\delta\mu\kappa\sigma$	$\delta\kappa$	$\delta\mu\kappa\sigma$	κ	$\delta\mu\kappa\sigma$	$\delta\mu\kappa\sigma$
	Refuse/Close Connection		κ	κ	κ	κ	κ	κ	κ		κ	κ
	Discard		$\mu\sigma$	μ	σ	σ	σ	σ	σ			σ

- (1) Kafka allows this limit to be set per source IP.
- (2) Configured using Websocket proxy or by setting separate limits on inbound and outbound connections.
- (3) Connection limit applies per application.
- (4) If the [Out-Of-Memory \(OOM\)](#) killer is enabled, the [OOM](#) killer will be activated once the memory limit is reached to terminate transient processes (e.g., clients connections) with high memory utilization to maintain service functionality.

- *Disk space limit*: This can be used to limit the amount of disk space used by a broker. It can be specified as a percentage of the total disk space or set as an absolute value. NATS JetStream, Pulsar, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis provide this configuration option.
- *Memory limit*: This configuration limits the total memory available for a broker. It can be specified as a percentage of the total memory or set as an absolute value. This limit is supported by Redis Streams, Redis PubSub, NATS JetStream, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis.
- *Network limit*: Table 9.1 shows that all the systems we study allow the number of active open connections to be limited.
- *Topic size limit*: This limit caps the number or total size of messages in a topic. This limit is offered by all the systems we study except Redis PubSub, NATS Core, and Ejabberd.

Violation Policies. We found five violation policies (Table 9.1) that are applicable when the aforementioned resource limits are reached.

- *Spill to disk*: When a topic size or a broker memory limit is reached, the system will spill messages to disk. This is supported by NSQ, RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis.
- *Block client(s)*: If a limit is reached, this policy blocks the offending client, a group of clients, or all clients. Table 9.1 shows that this policy is supported by nine of the systems we study.
- *Notify*: Following this policy the broker sends a notification to producers or consumers that try to utilize a resource while its limit is reached. Table 9.1 shows that this is the most widely supported violation policy.
- *Refuse/close connection*: Table 9.1 shows that all systems, except Ejabberd, refuse new connections once the connection limit is reached. In Kafka, unless the limit is set per source IP, new connections are queued to wait for a vacancy. Otherwise, new connections will be immediately dropped. Ejabberd applies this policy per application and closes the oldest connection of an application before accepting a new connection from that application.

- *Discard*: This policy discards old messages to make room for new messages. The messages are discarded by removing them from the system. This policy is supported by Redis Streams, Redis PubSub, Kafka, NATS JetStream, Pulsar, RabbitMQ, and ActiveMQ Artemis. In NSQ, ephemeral topics discard old messages once the topic size limit is reached instead of spilling to disk. RabbitMQ also provides an option to move messages to a dead-letter exchange.

9.2 Flow Control

Flow control is an essential technique to prevent brokers and consumers from being overwhelmed. Flow control mechanisms are applied between the producers and brokers, and between brokers and the consumers (or between producers and consumers for peer-to-peer systems, e.g., NSQ). The **MOM** systems we study offer two types of flow control: credit-based and rate-based. Some of the systems we study support both of these techniques and allow them to be used simultaneously. Table 9.2 shows the flow control mechanism offered by each of the systems we study.

9.2.1 Credit-Based Flow Control

Table 9.2 shows that several systems we study can apply a credit-based flow control, in which the number or size (i.e., total number of bytes) of buffered messages is limited. When the sender reaches this limit, it will stop sending messages until the receiver acknowledges some of the previous messages or grants additional credits. Noticeably, some of the systems that offer this flow control mechanism provide a configuration option to disable it.

Producer-to-broker credit-based flow control. Only ActiveMQ Classic and ActiveMQ Artemis support this type of flow control (Table 9.2). The producer requests credits from a broker. The broker responds with the number of credits allocated to that producer. The credits specify the total number of bytes of unacknowledged messages a producer is permitted. A producer can only send a message if it has available credit.

MOM-system-to-consumer credit-based flow control. To control the flow of messages from a broker, or the producer in the case of NSQ, to a consumer, **MOM** systems offer two options:

- *Consumer-specified credit*: In this approach, a consumer grants a broker credits specifying the number of messages it is ready to receive. When this credit is exhausted,

Table 9.2: The supported flow control mechanisms.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS JetStream	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Credit-Based Flow Control	Producer-Requested									✓	✓
	Consumer-Specified	✓		✓		✓	✓	✓	✓ ³	✓	
	Consumer-Preconfigured					✓ ¹	✓ ²	✓	✓ ⁴	✓	✓
Rate-Based Flow Control	Producer Rate				✓			✓	✓ ⁷	✓	✓
	Consumer Rate				✓	✓ ⁵	✓ ⁶				✓

- (1) NATS JetStream offers a flow control mechanism for consumers that share a subscription. It uses a preconfigured sliding-window flow control approach. It cannot be configured but can be disabled.
- (2) A preconfigured limit can be set on shared subscriptions.
- (3) Applies if the recovered consumer chooses to poll missed messages.
- (4) When active stream management is utilized, the client session closes once the consumer accumulates a configurable number of unacknowledged messages.
- (5) Configured upon the creation of push-based consumers.
- (6) Can be configured per subscription.

the consumer sends a message to the broker with new credits or acknowledges a minimum number of messages. Redis Streams, NSQ, NATS JetStream, Pulsar, RabbitMQ and ActiveMQ Classic all support this option. In Ejabberd, a failed consumer can choose to poll missed messages upon recovery, where it can control the flow of messages by requesting a few messages at a time.

- *System-wide consumer preconfigured credit*: In this case, a broker has a preconfigured credit value for the number of or total number of bytes for in-flight messages per consumer. NATS JetStream, Pulsar, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis support this approach.

9.2.2 Rate-Based Flow Control

Rate-based flow control limits the message rate or throughput of producers and consumers accessing a broker. Two options are available:

- *Producer rate limit*: Limits the throughput (messages or bytes per second) of a specific producer, group of producers, or every producer accessing a broker. Kafka offers a configuration option to limit data throughput per producer or group of producers. ActiveMQ Artemis limits the number of messages per producer upon the producer's request. RabbitMQ and ActiveMQ Classic automatically throttle fast producers for queues and consumers, respectively, to keep up with the pace.
- *Consumer rate limit*: Limits the throughput (messages or bytes per second) of a consumer, a group of consumers, or every consumer using the same broker. Kafka, NATS JetStream, Pulsar, and ActiveMQ Artemis support this rate limit. Kafka offers a configuration to limit data throughput per consumer or group of consumers. Pulsar offers a configurable limit per subscription. Push-based consumers in NATS JetStream set their limits upon creation. Consumers in ActiveMQ Artemis inform the broker of their limit.

9.3 Summary

In this chapter, we discuss the limits that **MOM** systems impose on the usage of available hardware resources along with the applicable violation policies. **MOM** systems can specify resource limits on processing, memory, network, and disk resources. To keep resource usage within these limits, **MOM** systems support five applicable violation policies, with notifying the clients that try to utilize an exhausted resource being the most supported policy. Additionally, point-to-point credit-based and rate-based flow control mechanisms are applicable to avoid overwhelming brokers and consumers, with some systems supporting both mechanisms. Finally, the resource management and flow control techniques are the base for offering isolation for multi-tenant deployments.

Chapter 10

Implementation Details

MOM systems often offer a client library to allow applications to access the **MOM** system. In this chapter, we discuss implementation details related to the client interactions.

10.1 Transport Protocols

Table 10.1 shows the transport layer protocol and communication mode (blocking/non-blocking) of the systems we study. All **MOM** systems support the use of TCP to transfer messages. Noticeably, Pulsar, Ejabberd, and ActiveMQ Classic offer the option to use UDP for to transfer messages. Six of the studied systems offer Websocket-based communication. Finally, ActiveMQ Classic is the only system that can leverage IP multicasting to deliver messages to consumers.

10.2 **MOM**-Consumer Communication Modes

We now examine the mode of communication (blocking or non-blocking) that is initiated from the **MOM** service to the consumer. The mode of communication from producers to the **MOM** systems depends on the producers' design, so it is out of the scope of the **MOM** system implementation.

Table 10.1 shows the modes of communication offered by the systems we study. We note that all the systems support a non-blocking mode of communication in their implementation. In this mode, a broker dispatches messages to a consumer without blocking to wait

Table 10.1: The supported transport protocols and MOM-consumer communication modes.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Transport Protocols	TCP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	UDP						✓		✓	✓	
	Websocket					✓	✓	✓ ¹	✓	✓	✓ ²
	IP Multicast									✓	
MOM-Consumer Communication Mode	Non-blocking	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Blocking									✓	

(1) Only supported by the [Simple \(or Streaming\) Text Oriented Messaging Protocol \(STOMP\)](#) and [MQTT](#) plugins.

(2) ActiveMQ Artemis supports Websocket transport for the [Advanced Message Queuing Protocol \(AMQP\)](#), [STOMP](#), and [MQTT](#) protocols.

for an acknowledgment. This design avoids blocking on slow consumers and reduces the number of threads used for message forwarding. All of the systems support non-blocking I/O in their implementation.

It is noteworthy that ActiveMQ Classic is the only system that allows the choice of whether the mode of communication should be blocking or non-blocking. Moreover, ActiveMQ Classic allows this configuration to be set per consumer.

10.3 HTTP Interface

Typically, a [Command Line Interface \(CLI\)](#) is offered by MOM systems to manage and configure the system. Some MOM systems offer an HTTP interface that allows users to interact with the system. The HTTP interface allows four types of operations:

- Producing and consuming messages.
- Creating new topics.
- Changing system configurations.

Table 10.2: The operations supported through the HTTP interface.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
HTTP Interface	Produce\Consume			✓				✓	✓	✓	✓
	Create Topics			✓			✓	✓	✓	✓	✓
	Configuration			✓			✓	✓	✓ ¹		✓
	Metrics			✓			✓	✓	✓	✓	✓

(1) Configuration changes through the HTTP interface only affect the current running instance and are not permanent. If the system reboots, then the configuration is lost.

- Monitoring the system’s state and its performance.

Table 10.2 shows the operations each system allows through the HTTP interface. NSQ, Pulsar, RabbitMQ, Ejabberd, ActiveMQ Classic, and ActiveMQ Artemis offer an HTTP interface. Table 10.2 shows that NSQ, RabbitMQ, Ejabberd, and ActiveMQ Artemis support all aforementioned types of operations. Pulsar does not support producing or consuming messages through the HTTP interface. ActiveMQ Classic does not support changing the system configuration through the HTTP interface. Pulsar and ActiveMQ Classic offer a RESTful [API](#).

10.4 Security

Table 10.3 lists several types of support provided for secure client interactions. The majority of the systems we study offer standard security mechanisms for authentication, access control, and secure communication.

- *Authentication*: All of the 10 systems we study support a range of authentication options including username/password, certificate, or access token for each client. Moreover, many of the systems support using an external authentication system, making it easier to integrate them in an institution’s IT infrastructure.

Table 10.3: The security measures supported in the 10 systems we studied.

System	Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Security Measures	Built-in Authentication	✓	✓	✓	✓	✓	✓	✓	✓	✓
	External Authentication			✓	✓	✓	✓	✓	✓	✓
	Access Control List (ACL)	✓	✓		✓	✓	✓	✓	✓	✓
	Secure Transport Layer	✓	✓	✓	✓	✓	✓	✓	✓	✓

- *Access control*: All the systems we study, except NSQ, provide [Access Control List \(ACL\)](#) mechanism. The [ACL](#) specifies what a client can access and with which operations (e.g., produce, subscribe, consume, or manage topics such as create, delete, and configure).
- *Secure communication*: All of the systems we study support using a secure transport layer such as [Transport Layer Security \(TLS\)](#) [104] or [Secure Sockets Layer \(SSL\)](#) [105].

10.5 Summary

In this chapter, we study the implementation details related to client interactions. All [MOM](#) systems we study support TCP, whereas some systems also support UDP, Websocket, and IP Multicast as a message transport protocol. Additionally, all of the systems support non-blocking I/O in their implementation. Some [MOM](#) systems offer an HTTP interface to facilitate client access to the system in addition to the client libraries they typically offer. Lastly, [MOM](#) systems we study support standard mechanisms for client authentication, access control, and secure communication.

Chapter 11

Active Messaging

In this chapter, we discuss the active messaging feature offered by Pulsar and Ejabberd. In this paradigm, messages can be processed using application-specific code while in transit through the [MOM](#) system.

These capabilities can be used to implement complex processing pipelines. Furthermore, active messaging can be used to deploy a flexible dissemination topology and support custom load balancing and filtering techniques.

Pulsar Active Messaging. Active messaging in Pulsar is enabled through Pulsar Functions. Pulsar functions are serverless computing functions written in Java, Python, or Go, and they can consume messages from one or more input topics, apply user-defined processing logic, and produce messages to one or more topics. Pulsar runs the functions at the broker or on a dedicated computational cluster.

Pulsar Functions support chaining, in which the output topic of a function can be the input topic of another function. Pulsar Functions feature three types of processing semantics:

- *At-most-once:* A message is immediately acknowledged after it is consumed by the function, regardless of whether or not the message is successfully processed.
- *At-least-once:* A message is guaranteed to be processed at least once. However, a message can be processed multiple times under some failure scenarios.
- *Effectively-once:* Similar to the at-least-once semantic, a message can be consumed multiple times by a function, but the output of the function will be entered once in the

output topics. This is done by logging the function's status using durable storage and checking the function status before committing the output of a function.

Ejabberd Active Messaging. Ejabberd offers active messaging using a modular system design and leveraging Erlang hot-code swapping. The Ejabberd architecture relies on minimal system core modules, whereas the majority of features are developed as pluggable modules. This modular architecture allows the system to be extended with new modules to provide new features. In addition, Ejabberd allows the system administrator to load new modules, disable a running module, or enable a module during runtime.

Chapter 12

Messaging Protocols

A number of protocols has been developed that specify how clients can interact with MOM systems including [JMS](#), [AMQP](#), [STOMP](#), [MQTT](#), [eXtensible Messaging and Presence Protocol \(XMPP\)](#), and Openwire. These protocols typically define specifications for interacting with a [MOM](#) system, help integrate and interoperate multiple [MOM](#) systems, remove vendor lock-in, and provide a common software-layer that supports popular standards. These protocols provide specifications at different levels, from low level message formats to messaging semantics, to application-level [API](#). Some protocols, such as [JMS](#), offer a high-level Java-based [API](#) that defines how a client can interact with a [MOM](#) system. All of the above-mentioned protocols provide wire-level specifications (i.e., the structure and format of messages). Furthermore, all of the listed protocols specify some of the semantics for the [MOM](#) system, such as access mechanism, transactions, or delivery semantics.

Table [12.1](#) shows the protocols supported by the systems we study, where seven systems use custom protocols, in which five of those systems only use a custom protocol, whereas two support other protocols. [MQTT](#) is the most popular standard and is supported by five of the systems we study. We note that RabbitMQ, ActiveMQ Classic, and ActiveMQ Artemis support five or more protocols.

Table 12.1: The messaging standards supported by the systems we study. P means a standard is supported through a plug-in. J means that the standard is exclusively supported in NAT JetStream.

System		Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	Ejabberd (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Messaging Protocols	JMS							✓P		✓	✓
	AMQP 0.9.1							✓			
	AMQP 1.0							✓P		✓	✓P
	STOMP							✓P		✓	✓P
	MQTT					✓J		✓P	✓P	✓	✓P
	XMPP								✓	✓	
	Openwire									✓	✓P
	Other Standard									✓	✓P
	Custom	✓	✓	✓	✓	✓	✓				✓

Chapter 13

Data Set

For wider impact we have made our data open to the public [1]. For each system, for each characteristic we added an annotation with a note that quotes the source and a link to the source that details the characteristic.

The data set is compiled as a google sheet and available at [1]. Wherever there is an empty cell, this means a feature or option is not supported by the corresponding system. Sometimes an empty cell has a note to either clearly state that the feature is not supported or why it is not supported. An example of such a case is shown in Figure 13.1, where Redis PubSub does not support spilling messages to disk because it essentially keeps messages in memory and drops them immediately once dispatched to consumers.

On the other hand, a cell containing some character indicates that the feature or option is supported by the corresponding system. In such a case, the cell includes a link to the source of this information. To make it easier to find the specific text related to the characteristic, the link highlights the related text at the source web page. An example of such a case is shown in Figures 13.2 - 13.4. The note can appear upon hovering with the cursor over the cell (Figure 13.4).

System	Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	elabard (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Resource Management and Flow Control										
Resource Limits	Rate Limit (ϵ)				ϵ					
	Disk Space Limit (δ)					δ	δ	δ	δ	δ
	Memory Limit (μ)	μ	μ			μ	μ	μ	μ	μ
	Connection Limit (K)	K	K	K	K	K	K	K	K	K
	Exchange Size Limit (σ)	σ		σ	σ	σ	σ	σ	σ	σ
Violation Policy	Spill to Disk								$\mu\sigma$	$\mu\sigma$
	Block Client(s)	μ	μ					μ	$\delta\mu\sigma$	$\delta\mu\sigma$
	Notify	μK	μK					K	$\delta\mu K\sigma$	$\delta\mu K\sigma$
	Refuse / Close Connection	K	K						K	K
	Discard	$\mu\sigma$	μ							σ

*While in Pub/Sub messages are fire and forget and are never stored anyway, and while when using blocking lists, when a message is received by the client it is popped (effectively removed) from the list, streams work in a fundamentally different way. All the messages are appended in the stream indefinitely (unless the user explicitly asks to delete entries); different consumers will know what is a new message from its point of view by remembering the ID of the last message received" - https://redis.io/topics/streams-intro#:~:text=While%20in%20Pub,Last%20message%20received.

Figure 13.1: An empty cell with a note explaining why Redis PubSub cannot support spilling messages to disk.

System	Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	elabard (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Resource Management and Flow Control										
Resource Limits	Rate Limit (ϵ)				ϵ					
	Disk Space Limit (δ)					δ	δ	δ	δ	δ
	Memory Limit (μ)	μ	μ			μ	μ	μ	μ	μ
	Connection Limit (K)	K	K	K	K	K	K	K	K	K
	Exchange Size Limit (σ)	σ		σ	σ	σ	σ	σ	σ	σ
Violation Policy	Spill to Disk			σ			$\mu\sigma$		$\mu\sigma$	$\mu\sigma$
	Block Client(s)	μ	μ		ϵK	$\delta\mu\sigma$	δ	$\delta\mu\sigma$	μ	$\delta\mu\sigma$
	Notify	μK	μK	K	ϵK	$\delta\mu K\sigma$	δK	$\delta\mu K\sigma$	K	$\delta\mu K\sigma$
	Refuse / Close Connection	K	K	K	K	K	K	K	K	K
	Discard	$\mu\sigma$	μ	σ	σ	σ	σ	σ		σ

*When free disk space drops below a configured limit (50 MB by default)

Free Disk Space Alarms ... rabbitmq.com

Figure 13.2: A screenshot showing a link to the cell showing RabbitMQ support of limiting disk space.

Memory Alarms

Memory Threshold: What it is and How it Works

The RabbitMQ server detects the total amount of RAM installed in the computer on startup and when

`rabbitmqctl set_vm_memory_high_watermark fraction` is executed. By default, when the RabbitMQ server uses above 40% of the available RAM, it raises a memory **alarm** and blocks all connections that are publishing messages. Once the memory alarm has cleared (e.g. due to the server paging messages to disk or delivering them to clients that consume and **acknowledge the deliveries**) normal service resumes.

Figure 13.3: A screenshot of the web source detailing the proof of disk space limit in RabbitMQ. This web page can be accessed through the link shown in the previous figure.

System	Redis Streams	Redis PubSub	NSQ	Kafka	NATS	Pulsar	RabbitMQ	elabber (eCS)	ActiveMQ Classic	ActiveMQ Artemis
Resource Management and Flow Control										
Resource Limits	Rate Limit (ϵ)					ϵ				
	Disk Space Limit (δ)						δ	δ	δ	<small>*When free disk space drops below a configured limit (50 MB by default), an alarm will be triggered and all producers will be blocked.*</small>
	Memory Limit (μ)	μ	μ				μ			μ
	Connection Limit (K)	K	K	K	K	K	K	K		K
	Exchange Size Limit (σ)	σ		σ	σ	σ	σ	σ	σ	σ
Violation Policy	Spill to Disk			σ			$\mu \sigma$		$\mu \sigma$	$\mu \sigma$
	Block Client(s)	μ	μ		ϵK	$[\delta \mu \sigma]$	δ	$\delta \mu \sigma$	μ	$\delta \mu \sigma$
	Notify	μK	μK	K	ϵK	$[\delta \mu K \sigma]$	δK	$\delta \mu K \sigma$	K	$\delta \mu K \sigma$
	Refuse / Close Connection	K	K	K	K	K	K	K		K
	Discard	$\mu \sigma$	μ	σ	σ	σ	σ	σ		σ

Figure 13.4: A screenshot of a note showing the text quote of the proof that RabbitMQ supports disk space limit.

Chapter 14

Discussion

Insights and Research Challenges. Two main characteristics define [MOM](#) systems: (a) the incorporation of features central to cloud deployment and applications and (b) extreme flexibility. Many of the [MOM](#) systems we study were designed with extreme flexibility in mind, beginning from a flexible dissemination topology to supporting all possible configuration options and service semantics. This flexibility comes at a cost in system complexity, development, and testing.

Our data set [1] shows that systems like Redis PubSub, NATS Core, and NSQ support the least amount of features in comparison to others. For NSQ, this is resulting from adopting a peer-to-peer brokerless topology that does not offer most of the [MOM](#) communication characteristics including space decoupling, time decoupling, and flexibility. Additionally, the brokerless topology limits the number of features that can be supported, such as replication, subscription recovery, and message ordering per multicast exchange due to the lack of coordination between producers of a certain topic. On the other hand, Redis PubSub and NATS Core do not support features like durable storage, replication, subscription recovery, detection of duplicate messages, consumer acknowledgments, message retention, and flow control because they are designed to be in-memory best-effort [MOM](#) systems.

The data set [1] also shows that Pulsar, ActiveMQ Classic, and ActiveMQ Artemis support most of the features we study. Comparing ActiveMQ distributions to pulsar reveals that ActiveMQ does not support rack-aware replica placement, multicast exchange ordering, a discovery service with strong consistency, or content-based filtering. ActiveMQ distributions lack the active messaging feature that Pulsar offers, which allow implementing flexible topologies, custom content-based filtering, and other application-specific processing

logic. On the other hand, Pulsar lacks the support of brokerless peer-to-peer topology and flexible topology, object messages, message priority, limiting memory utilization, support of standard messaging protocols, and message transfer over IP Multicast.

A large number of systems offer basic transaction support, with many offering non-atomic transactions. Supporting atomic and durable transactions for low-latency and high throughput [MOM](#) systems remains an open research problem.

Active messaging is emerging as a new capability in [MOM](#) systems. This technology is in its infancy and has limited support for protecting a user from competing tenants or misbehaving functions. Building efficient resource management and isolation techniques for active messaging remains an open research problem.

Chapter 15

Conclusions

We provided an in-depth analysis of 10 popular [MOM](#) systems. For each system, we studied 42 features related to various aspects of [MOM](#) systems, including topologies supported, reliability guarantees, service semantics, client interaction mechanisms, resource management, flow control, and active messaging. We present our findings and identify open research problems. Our annotated data set is publicly available at [\[1\]](#) to help researchers understand the state of the art, help developers choose the best system for a certain application, help practitioners understand the semantics of different systems, and help the community understand the capabilities of different systems and focus its effort on fewer number of [MOM](#) systems.

References

- [1] A Survey of Message-Oriented Middleware Systems. <https://docs.google.com/spreadsheets/d/1HrZ7ub19FuuBzA5z4aA6RfR5vnkdnm0bg3hxfADspEA/edit?usp=sharing>. Accessed: October. 2021.
- [2] Hello from Apache BookKeeper — Apache BookKeeper. <https://bookkeeper.apache.org/>. Accessed: October. 2021.
- [3] Mnesia · Elixir School. <https://elixirschool.com/en/lessons/storage/mnesia>. Accessed: October. 2021.
- [4] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. TAO: How Facebook Serves the Social Graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 791–792, New York, NY, USA, 2012. Association for Computing Machinery.
- [5] Brad Calder, Ju Wang, Aaron Ogus, Niranjana Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, pages 143–157, New York, NY, USA, 2011. Association for Computing Machinery.

- [6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.
- [7] Hermes. <http://hermes.allegro.tech/#why>. Accessed: October. 2021.
- [8] Building a Microservices Ecosystem with Kafka Streams and KSQL. <https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/>. Accessed: October. 2021.
- [9] Microservices - why use RabbitMQ? - CloudAMQP. <https://www.cloudamqp.com/blog/why-use-rabbitmq-in-a-microservice-architecture.html>. Accessed: October. 2021.
- [10] Apache Storm. <https://storm.apache.org/>. Accessed: October. 2021.
- [11] KAFKA STREAMS. <https://kafka.apache.org/documentation/streams/>. Accessed: October. 2021.
- [12] An Introduction to Stream Processing with Pulsar Functions - DZone. <https://dzone.com/articles/an-introduction-to-stream-processing-with-pulsar-f>. Accessed: October. 2021.
- [13] Is Kafka the Next Big Thing in the Banking and Financial Sector? - DZone. <https://dzone.com/articles/is-kafka-the-next-big-thing-in-the-banking-and-fin>. Accessed: October. 2021.
- [14] How Kafka Helped Rabobank Modernize Alerting System. <https://www.datanami.com/2017/08/15/kafka-helped-rabobank-modernize-alerting-system/>. Accessed: October. 2021.
- [15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015.
- [16] L. Rodríguez-Gil, J. García-Zubia, P. Orduña, and D. Lopez-de-Ipiña. An Open and Scalable Web-Based Interactive Live-Streaming architecture: The WILSP Platform. *IEEE Access*, 5:9842–9856, 2017.
- [17] Feiyang Wang, Dongyu Zhang, Yuming Lu, and Kai Lei. ”PSVA: A Content-Based Publish/Subscribe Video Advertising Framework”. In Meikang Qiu, editor, *Smart*

Computing and Communication, pages 249–258, Cham, 2018. Springer International Publishing.

- [18] Huy Hoang, Benjamin Cassell, Tim Brecht, and Samer Al-Kiswany. RocketBufs: A Framework for Building Efficient, In-Memory, Message-Oriented Middleware. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 121–132, New York, NY, USA, 2020. Association for Computing Machinery.
- [19] Using Apache Kafka to Drive Cutting-Edge Machine Learning — Confluent. <https://www.confluent.io/blog/using-apache-kafka-drive-cutting-edge-machine-learning>. Accessed: October. 2021.
- [20] A distributed computation system for deep learning experiments with Docker Compose and RabbitMQ. — by Anis Khelif — Deezer I/O. <https://deezer.io/a-distributed-computation-system-for-deep-learning-experiments-with-docker-compose-and-rabbitmq-5ac4ab344406>. Accessed: October. 2021.
- [21] Zhijie Han and Miaoxin Xu. Machine Learning Techniques in Storm. In *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming*, pages 139–142, Nanjing, China, 2015.
- [22] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 351–366, Oakland, CA, 2018.
- [23] Apache Kafka. <https://kafka.apache.org/>. Accessed: October. 2021.
- [24] Hello from Apache Pulsar — Apache Pulsar. <https://pulsar.apache.org/>. Accessed: October. 2021.
- [25] Messaging that just works - RabbitMQ. <https://www.rabbitmq.com/>. Accessed: October. 2021.
- [26] ActiveMQ Artemis The Next Generation Message Broker by ActiveMQ. <https://activemq.apache.org/components/artemis/>. Accessed: October. 2021.

- [27] MQ - IBM MQ - Canada — IBM. <https://www.ibm.com/ca-en/products/mq>. Accessed: October. 2021.
- [28] IronMQ - Serverless Message Queue. <http://www.iron.io/mq>. Accessed: October. 2021.
- [29] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Kaushik Veer-araghavan, Benjamin Wester, and Peter Xie. Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 351–366, USA, 2015. USENIX Association.
- [30] Google Cloud Pub/Sub. <https://cloud.google.com/pubsub>. Accessed: October. 2021.
- [31] Fully Managed Message Queuing – Amazon Simple Queue Service – Amazon Web Services. <https://aws.amazon.com/sqs/>. Accessed: October. 2021.
- [32] Azure Queue storage documentation — Microsoft Learn. <https://docs.microsoft.com/en-us/azure/storage/queues/>. Accessed: October. 2021.
- [33] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
- [34] TIBCO Rendezvous®. <https://www.tibco.com/products/tibco-rendezvous>. Accessed: October. 2021.
- [35] Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. "A Case for Message Oriented Middleware". In Prasad Jayanti, editor, *Distributed Computing*, pages 1–17, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [36] Tarek R. Sheltami, Anas A. Al-Roubaiey, and Ashraf S. Mahmoud. A Survey on Developing Publish/Subscribe Middleware over Wireless Sensor/Actuator Networks. *Wirel. Netw.*, 22(6):2049–2070, Aug 2016.
- [37] Eduardo Souto, Germano Guimares, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, Carlos Ferraz, and Judith Kelner. Mires: A publish/subscribe middleware for sensor networks. *Personal Ubiquitous Comput.*, 10:37–44, 02 2006.

- [38] Jan-Hinrich Hauer, Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. A Component Framework for Content-Based Publish/Subscribe in Sensor Networks. In *Wireless Sensor Networks*, pages 369–385, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [39] Ke Shi, Zhancheng Deng, and Xuan Qin. TinyMQ: A Content-based Publish/Subscribe Middleware for Wireless Sensor Networks. In *The fifth international conference on sensor technologies and applications (SENSORCOMM 2011)*, 01 2011.
- [40] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(1):70–95, 2016.
- [41] Aarush Ahuja, Vanita Jain, and Dharmender Saini. *Characterization and Benchmarking of Message-Oriented Middleware*, pages 129–147. Springer International Publishing, Cham, 2021.
- [42] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance Evaluation of Message-Oriented Middleware Using the SPECjms2007 Benchmark. *Performance Evaluation*, 66:410–434, 08 2009.
- [43] Kai Sachs, Stefan Appel, Samuel Kounev, and Alejandro Buchmann. Benchmarking Publish/Subscribe-Based Messaging Systems. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications, DASFAA’10*, page 203–214, Berlin, Heidelberg, 2010. Springer-Verlag.
- [44] OpenMessaging benchmark. <https://openmessaging.cloud/>. Accessed: October. 2021.
- [45] Vanita Jain, Aarush Ahuja, and Dharmender Saini. Evaluation and Performance Analysis of Apache Pulsar and NATS. In Kavita Khanna, Vania Vieira Estrela, and Joel José Puga Coelho Rodrigues, editors, *Cyber Security and Digital Forensics*, pages 179–190, Singapore, 2022. Springer Singapore.
- [46] Vineet John and Xia Liu. A Survey of Distributed Message Broker Queues. *CoRR*, abs/1704.00411, 2017.
- [47] GitHub - vineetjohn/flotilla: Automated message queue orchestration for scaled-up benchmarking. <https://github.com/vineetjohn/flotilla>. Accessed: October. 2021.

- [48] Redis Streams. <https://redis.io/docs/data-types/streams/>. Accessed: October. 2021.
- [49] Redis Enterprise Software – The Real-Time Data Platform. <https://redis.com/redis-enterprise-software/overview/>. Accessed: October. 2021.
- [50] Redis Pub/Sub. <https://redis.io/docs/manual/pubsub/>. Accessed: October. 2021.
- [51] NSQ A realtime distributed messaging platform. <https://nsq.io/>. Accessed: October. 2021.
- [52] Data in Motion Platform for Enterprise. <https://www.confluent.io/product/confluent-platform/>. Accessed: October. 2021.
- [53] Apache RocketMQ. <https://rocketmq.apache.org/>. Accessed: October. 2021.
- [54] What is Message Queue for Apache RocketMQ? <https://www.alibabacloud.com/help/doc-detail/29532.htm?spm=a2c63.128256.a3.10.48687882EsHNSE>. Accessed: October. 2021.
- [55] NATS.io – Cloud Native, Open Source, High-performance Messaging. <https://nats.io/>. Accessed: October. 2021.
- [56] Synadia. <https://synadia.com/>. Accessed: October. 2021.
- [57] StreamNative BYOC: Pulsar-as-a-Service in the cloud of your choice. <https://streamnative.io/cloud/managed/>. Accessed: October. 2021.
- [58] VMware RabbitMQ Documentation. <https://docs.vmware.com/en/VMware-Tanzu-RabbitMQ/index.html>. Accessed: October. 2021.
- [59] EMQX: The World’s #1 Open Source Distributed MQTT Broker. <https://www.emqx.io/>. Accessed: October. 2021.
- [60] EMQX Enterprise: Enterprise MQTT Platform At Scale. https://www.emqx.com/en/products/emqx?utm_source=emqx.io&utm_medium=referral&utm_campaign=emqxio-header-to-enterprise. Accessed: October. 2021.
- [61] Eclipse Mosquitto. <https://mosquitto.org/>. Accessed: October. 2021.

- [62] TIBCO® Messaging - Eclipse Mosquitto Distribution. <https://www.tibco.com/products/tibco-messaging-eclipse-mosquitto-distribution>. Accessed: October. 2021.
- [63] ejabberd XMPP Server with MQTT Broker SIP Service. <https://www.ejabberd.im/>. Accessed: October. 2021.
- [64] Create Awesome Realtime Systems XMPP Server + MQTT Broker + SIP Service. <https://www.process-one.net/en/ejabberd/#getejabberd>. Accessed: October. 2021.
- [65] Faye: Simple pub/sub messaging for the web. <https://faye.jcoglan.com/>. Accessed: October. 2021.
- [66] Emitter: Scalable Real-Time Communication Across Devices. <https://emitter.io/>. Accessed: October. 2021.
- [67] GitHub - emitter-io/emitter: High performance, distributed and low latency publish-subscribe platform. <https://github.com/emitter-io/emitter#licensing>. Accessed: October. 2021.
- [68] Nchan - flexible pubsub for the modern web. <https://nchan.io/>. Accessed: October. 2021.
- [69] VerneMQ - A MQTT broker that is scalable, enterprise ready, and open source. <https://vernemq.com/>. Accessed: October. 2021.
- [70] Commercial Services - VerneMQ. <https://vernemq.com/services.html>. Accessed: October. 2021.
- [71] ActiveMQ "Classic" The Tried and Trusted Open Source Message Broker. <https://activemq.apache.org/components/classic/>. Accessed: October. 2021.
- [72] ActiveMQ. <https://activemq.apache.org/support#commercial-support->. Accessed: October. 2021.
- [73] Github - moscajs/aedes: Barebone mqtt broker that can run on any stream server, the node way. <https://github.com/moscajs/aedes>. Accessed: October. 2021.
- [74] GitHub - hivemq/hivemq-community-edition: HiveMQ CE is a Java-based open source MQTT broker that fully supports MQTT 3.x and MQTT 5. It is the foundation of the HiveMQ Enterprise Connectivity and Messaging Platform. <https://github.com/hivemq/hivemq-community-edition>. Accessed: October. 2021.

- [75] Discover the 3 different editions of HiveMQ. <https://www.hivemq.com/hivemq/editions/>. Accessed: October. 2021.
- [76] Federation Plugin — RabbitMQ. <https://www.rabbitmq.com/federation.html>. Accessed: October. 2021.
- [77] Shovel Plugin — RabbitMQ. <https://www.rabbitmq.com/shovel.html>. Accessed: October. 2021.
- [78] Virtual Destinations. <https://activemq.apache.org/virtual-destinations>. Accessed: October. 2021.
- [79] Address Federation · ActiveMQ Artemis Documentation. <https://activemq.apache.org/components/artemis/documentation/latest/federation-address.html>. Accessed: October. 2021.
- [80] GitHub REST API - GitHub Docs. <https://docs.github.com/en/rest>. Accessed: October. 2021.
- [81] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.
- [82] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, page 466–476, New York, NY, USA, 2013. Association for Computing Machinery.
- [83] ZeroMQ. <https://zeromq.org/>. Accessed: October. 2021.
- [84] GitHub - nanomsg/nanomsg: nanomsg library. <https://github.com/nanomsg/nanomsg>. Accessed: October. 2021.
- [85] The WhatsApp Architecture Facebook Bought For \$19 Billion - High Scalability. <http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html>. Accessed: October. 2021.
- [86] Ubisoft — ProcessOne. <https://www.process-one.net/en/customers/ubisoft/>. Accessed: October. 2021.

- [87] How LinkedIn customizes Apache Kafka for 7 trillion messages per day — LinkedIn Engineering. <https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages>. Accessed: October. 2021.
- [88] Can Kafka Handle a Lyft Ride? - Confluent. <https://www.confluent.io/resources/kafka-summit-2020/can-kafka-handle-a-lyft-ride/>. Accessed: October. 2021.
- [89] Scaling with RabbitMQ @ Soundcloud. <https://tanzu.vmware.com/content/blog/scaling-with-rabbitmq-soundcloud>. Accessed: October. 2021.
- [90] Open-sourcing Pulsar, Pub-sub Messaging at Scale — Yahoo Engineering. https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale#notes?ref_url=https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale/embed#=_. Accessed: October. 2021.
- [91] rpc(3) - Linux manual page. <https://man7.org/linux/man-pages/man3/rpc.3.html>. Accessed: October. 2021.
- [92] gRPC. <https://grpc.io/>. Accessed: October. 2021.
- [93] Apache Thrift - Home. <https://thrift.apache.org/>. Accessed: October. 2021.
- [94] Rebtel — ProcessOne. <https://www.process-one.net/en/customers/rebtel/>. Accessed: October. 2021.
- [95] Nimbuzz — ProcessOne. <https://www.process-one.net/en/customers/nimbuzz/>. Accessed: October. 2021.
- [96] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
- [97] Henry F. Korth Avi Silberschatz and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.
- [98] Secure and resizable cloud compute – Amazon EC2 – Amazon Web Services. <https://aws.amazon.com/ec2/>. Accessed: October. 2021.
- [99] Kubernetes. <https://kubernetes.io/>. Accessed: October. 2021.

- [100] Consul by HashiCorp. <https://www.consul.io/>. Accessed: October. 2021.
- [101] etcd. <https://etcd.io/>. Accessed: October. 2021.
- [102] LDAP.com – Lightweight Directory Access Protocol. <https://ldap.com/>. Accessed: October. 2021.
- [103] JGroups - The JGroups Project. <http://www.jgroups.org/>. Accessed: October. 2021.
- [104] Transport Layer Security - Wikipedia. https://en.wikipedia.org/wiki/Transport_Layer_Security. Accessed: October. 2021.
- [105] A.C. Weaver. Secure Sockets Layer. *Computer*, 39(4):88–90, 2006.

Appendix A: Keywords

In this appendix, we list the unique keywords we used to search GitHub for [MOM](#) systems. We used these keywords and combinations of these keywords. The exact list of search terms we use is available in our data set [\[1\]](#).

- [AMQP](#)
- ActiveMQ
- Atom
- Auto
- Broker
- Bus
- Framework
- HTTP
- HornetQ
- [JMS](#)
- Kafka
- [MQTT](#)
- Message
- Messaging

- Middleware
- Notification
- OpenWire
- Platform
- Pub
- Pub-Sub
- PubSub
- Publish
- Publish-subscribe
- Publish/Subscribe
- PublishSubscribe
- Publisher
- PublisherSubscriber
- Qpid
- Queue
- REST
- RSS
- Redis
- RocketMQ
- **STOMP**
- Server
- Storm
- Stream

- Streaming
- Sub
- Subscribe
- Subscriber
- System
- WSIF
- WS Notification
- [XMPP](#)