Bachelor Thesis Business and IT

Patrick Japink

University of Twente

Redis, Apache Kafka, RabbitMQ: Effect of choosing Event-streaming middleware upon architectural constraints defined by Non-functional requirements in Event-driven microservice architectures.

Patrick Japink, University of Twente, The Netherlands

ABSTRACT

Event-driven architectures are gaining industry support. The comparison of Event-streaming brokers which have been established and which have been newly developed are compared with respect to their architectural constraints defined by non-functional requirements. Redis, Apache Kafka and RabbitMQ compared and representative architecture based on the use-case of inters-service communication is devised. The evaluation of non-functional requirements is reviewed and applied to the three software solutions.

Additional Key Words and Phrases: Non-functional requirements, Event-driven architecture

INTRODUCTION

The adoption of microservice architecture and more specifically, event-driven microservice architectures that utilize real-time data to support business objectives, is increasing. Companies across industries have seen an increase in the presence of data sources and volume which dovetails with these developments. Message-oriented middleware is an essential part of many distributed systems and microservice architectures. While traditional message queueing systems are common in microservice architectures, it is important to note that brokers with streaming capabilities play a distinct role in event-driven microservice architectures [3] [4]. Event-streaming brokers and traditional message queues generally differ in distinct aspects, for example, in that Event-streaming brokers offer non-destructive consumer semantics and that they offer a degree of backwards accessibility of past data. [5]. Generally, in Event-driven microservices, the role of Event-streaming brokers is to receive events from producers, store them in logically separate streams and make them available to one or more consumers [6]. The key data structure at the heart of a stream can be described as an append-only log [7]. For large-scale systems, groups of Event-stream broker instances together receive, store and provide events in clusters to fulfill the necessary requirements [4]. The streams of logically separate topics, or append-only logs, can thus be partitioned across separate machines [8]. The focus of this research are Event-streaming solutions used to enable Event-driven microservice architectures. In the last 5 years, some of the most well established messaging solutions, Redis and RabbitMQ have added data streaming capabilities to their software. Other established event streaming solutions, like Apache Kafka, have seen continued adoption in a wide variety of industries. Along with the benefits of adopting new architectural styles, new challenges and trade-offs have to be navigated in order for the possible benefits of these new adaptations to be realized. Changes in business strategies, like the aforementioned trends, prompt further innovations in architectures, which bring about new developments and extensions of existing applications.

The comparison of these three software products as Event-streaming brokers represents a gap in the research, as a lot of the development related to this is fairly recent. Redis, Apache Kafka and RabbitMQ are widely used software products, each with distinct features that make them applicable for different use-cases to varying degrees.

This research aims to compare them by their intersecting quality as Event-streaming solutions, and to analyze the implications that their differences have for their non-functional requirements, in the context of Event-driven microservice architectures. The leading question would therefore be: How do Redis, Apache Kafka and RabbitMQ differ with respect to their architectural constraints defined by non-functional requirements, used to facilitate asynchronous inter-service communication in event-driven microservice architectures?

This research aims to address this matter with the following research questions:

1) What are relevant non-functional requirements and features of Event-stream brokers used for asynchronous communication in microservice architectures?
2) What are the different properties of the three Event-stream brokers and what is their appropriate architecture and configuration for this use-case?
3) What are important insights for the evaluation of non-functional requirements and which one's out of those collected should be analyzed?
4) How do the three Event-broker solutions compare with respect to the selected non-functional requirements?

The first research question will be answered in section one through a surveying of the relevant contemporary literature and research. Section two covers the second research question. A description and comparison of the three Event-streaming brokers properties will be based on their documentation. The architecture and configuration will be informed by the results in the first section in combination with the documentations. The third research question, answered in the third section, consists of a survey of the relevant research, as well as the content of the first section to select a subset of the non-functional requirements. Section number four contains the answers to the fourth research question. The non-functional requirements elicited in research question three will be evaluated comparatively for the three Event-broker solutions, with support of the information in section two.

This analysis will be conducted based on the open-source versions of these applications, as opposed to the commercially supported options. It will also be based on the out-of-the-box features provided by said open source versions and not take into direct account possible custom extensions.

SECTION 1 – EVENT-STREAMING BROKERS

1.1 Requirements

For the elicitation of important non-functional requirements and the support of their analysis, materials from three related areas can be relied upon. Firstly, literature on distributed systems can be relied upon due to microservices being a subcategory of distributed systems. Furthermore, contemporary literature from industry professionals about Event-driven microservices are an appropriate source. Thirdly, due to their aforementioned similar use-case, we can rely on research that deals with the evaluation and comparison of message queues and publish/subscribe systems in order to gather insights.

Relevant work was perfumed by Bellemare in "Building Event-driven Microservices". Descriptions of four essential requirements for Event-brokers, with a focus on large-scale system capabilities have been provided by Bellemare [6]. The features listed and their provided contextualization by Bellemare [6] are the following:

Scalability - " Additional event broker instances can be added to increase the cluster's production, consumption, and data storage capacity*",*

Durability - " Event data is replicated between nodes. This permits a cluster of brokers to both preserve and continue serving data when a broker fails.*",*

*High Availability - "*A cluster of event broker nodes enables clients to connect to other nodes in the case of a broker failure. This permits the clients to maintain full uptime.*",*

*High Performance - "*Multiple broker nodes share the production and consumption load. In addition, each broker node must be highly performant to be able to handle hundreds of thousands of writes or reads per second.*"*

Four different papers that evaluate messaging systems based on a queue and publish/subscribe paradigm have been selected. *Table 1* shows a matrix of the included non-functional requirements per research paper. The term "performance" has been designated to include all papers which took the factors of "latency" and "throughput" into account. In some of the papers by the authors listed in *table 1*, this has been noted under "efficiency", in others as separate terms. The categories of "durability" mentioned by Bellemare [6] and the of "reliability" listed by Fu, Yu and Zhang [9] both refer to the same aspects, the usage of multiple nodes in a cluster and the replication of data between them in order to manage the failure of single nodes without interruption of the service as a whole. The term "reliability" will be used to represent both.

Table 1. Comparison of research papers

Authors	Dobbelaere and Esmaili	Fu, Yu and Zhang	Hedge and Nagaraja	Sharvari and Sowmya	Bellemare
performance	x	x	x	x	x
availability	x		x	x	x
scalability	x	x	x	x	x
usability		x			
reliability		x			x
compatibility		x			

The overlapping requirements named by Bellemare and the selected research papers are Scalability, Availability, Performance and Durability. The remaining requirements named by Zhang, Fu and Yu, are usability defined by "easiness of installation", "completeness of documentation" and "management & monitoring functionality" and compatibility [9]. The aspect of compatibility focuses on the connection of message queues to different types of long-term storage, which is not applicable to this context. Mairiza, Zowghi, and Nurmuliani performed research on which non-functional requirements are most commonly listed in certain domains [10], they named extensibility as one requirement spanning all system and application domains. Extensibility is also covered as a design goal of distributed systems in the work of Tanenbaum, under the umbrella term of "openness" [11]. The only non-functional requirement not included so far, of the five most common, listed by [10] and also by [12] is that of security. The survey of research and literature thus results in the following list of preliminary non-functional requirements:

1) Scalability
2) Availability
3) Performance
4) Reliability
5) Usability
6) Extensibility
7) Security

1.2 Features

In the domain of messaging systems, qualities of service (QOS), describe conditions which can apply to the delivery of messages. In [13], [9], [14] and [15], the relevant QOS of "delivery guarantees" has been named. This refers to the guarantees regarding the number of times the same message will be delivered to the same consumer. The delivery guarantees referred to are *at-least-once, at-most-once* and *exactly-once*. The two other QOS listed are that of *message persistence* and *message ordering.* The differing feature

requirements of more traditional message brokers make these less relevant categories for the evaluation of Event-streaming brokers. The persistence of events is an expected requirement, while the ordering of events may only differ when stored across different partitions, depending on the use-case. Bellemare further states in his work on Event-streaming brokers as important features of their underlying storage facilities: *Partitioning, Strict ordering, Indexing, Infinite retention and Replayability* [6].

Due to limitations of storage capacity, assuming continuous production of events only a subset of all historical data can be stored locally [16]. The rule-sets which dictate which data to remove and when are referred to as the *retention policy.* Since storage of events is an assumed feature, it is also an expected quality of the system. *Publisher acknowledgements* and *Consumer acknowledgements* are possible features which allow the publishing and consumption of events to be confirmed. Its implementation has implications for the consistency of data and message delivery guarantees. The position in the respective partition which consumers have last accessed can be stored by consumers or by the Event-streaming system [17]. As in the case of databases, multiple distinct operations, such as the adding of multiple events, can be viewed as a single operation, semantically. Just as with databases, such a conditional grouping can be named a *transaction* and represents another possible feature [18].

SECTION 2 – PROPERTIES, ARCHITECTURE AND CONFIGURATION

2.1 Properties and Architecture

Redis, Apache Kafka and RabbitMQ are software solutions with differing and overlapping capabilities. *Table 2* in the appendix A1 contains a general comparison of the products properties and streaming related features. Redis is a multifunctional, in-memory data store application with optional extended storage settings and different data-structures. Event-streaming capabilities were added starting with version 5.0. Apache Kafka is a high performance Event-streaming and Event-stream processing platform, with an extensive ecosystem. RabbitMQ is a feature-rich messaging application which supports a multitude of protocols. Event-streaming functionalities, including an optional new binary streaming protocol have been in development, starting with version 3.9.

Section one elaborated on common requirements and some of the architectural practices related to them. To adequately compare the different Event-streaming solutions with one another, the appropriate architecture for each needs to be determined. This architecture must be chosen according to the overall use-case of Event-driven microservice communication, based on the elicited requirements, while providing the needed features, across all three Event-streaming brokers.

The requirements are based on the assumption of a large-scale, distributed context. This is in line with the requirement of *performance*. Different requirements are positively related, meaning that certain architectural decisions benefit them similarly. *Scalability, Availability, Durability and Performance* all can all increase in a clustered, distributed architecture. This comes with the increased complexity of maintaining other requirements, such as data consistency, traceability, maintainability and others.
Across multiple sources, the concepts of clustering, scalable design, data replication across nodes and the possibility for automatic failovers and recovery from failures have been mentioned [4] [6] [9]. The chosen architecture will be a cluster made up of three main nodes. An uneven number of nodes is advantageous for certain leader election methods, because it helps to prevent split-brain scenarios [19] during which multiple nodes assume the leader role, leading to data inconsistencies. When configuring each of the three applications, the aim is to cater to the requirements and features elicited in section one. Because of the differing properties of the three Event-streaming broker solutions, the optimal and possible adjustments are distinct for each.

2.2 Configuration

This section describes the configuration appropriate for all three software products, such that they converge to fulfill the architectural requirement in a similar fashion.

2.2.1 Redis

In the Redis architecture, depicted in *Figure 1* of the appendix A2, data is replicated from master node to slave nodes. Each of the three master nodes will have two replica nodes. This results in an odd number of nodes which results in the option of an quorum [20] to aid in prevent split-brain conditions during network partitions to avoid data inconsistencies. Because the first line of storage for Redis is random-access memory, to fulfill the requirements, durability options need to be enabled. This involves setting append only file (AOF) and RDB snapshots, with a strong fsync policy/. These settings result in a lower performance but make it so that Redis fulfills the necessary durability requirements.

2.2.2 RabbitMQ

The RabbitMQ architecture is depicted in *Figure 2* of the appendix A3. To take advantage of the higher throughput and all stream related features such as offset tracking and publisher message deduplication, the stream plugin must be activated. Recovery mode should be set as "autoheal" to enable the automated failover during network partitions.

2.2.3 Kafka

Figure 3 of the appendix A4 shows the Apache Kafka architecture. The publisher acknowledgement level is set to the highest level. This causes events to only be acknowledged once they have been replicated to a majority of nodes present, resulting in their full durability in case of a complete failure of one node.

SECTION 3 – TRAITS AND SELECTION OF NON-FUNCTIONAL REQUIREMENTS

3.1 Traits

This section described the examination of relevant research papers with the goal of eliciting important factors that can provide guidance for the definition and evaluation of non-functional requirements.

The research conducted by Chung and Leite [21], as well as the research by Mairiza, Zowghi and Nurmuliani [10] provides information to further guide our evaluation of non-functional requirements. [21] and [10] compare and contrast various conceptualizations of non-functional requirements. Chiefly, the appropriate working definition of a non-functional requirement is that of any architectural constraint or quality that judges a system based on a certain condition. What is agreed upon by many authors is that there is no clear consensus when it comes to their scope, definition, elicitation, verification and the nomenclature used to describe them [10]. A consequence of this stated in [10] is that for practical purposes, non-functional requirements need to be defined with respect to the use-case. The NFR framework [22] and others discussed [10] propose a goal-oriented definition, appropriate for the system at hand. The authors of [10] describe broadly that non-functional requirements can be transformed from abstract requirements "into more concrete terms and detail" with the usage of "means, methods and operations". Another concept discussed in the work of [21] which was included in several classification schemes is that of "interdependencies" among non-functional requirements. Expressed more concretely, in relation to the framework in [22], this is further conceptualized as positive and negative correlations between non-functional requirements [21]. The analysis of [10] also concluded that certain non-functional requirements are defined as an attribute of other non-functional requirements. They provide the example of the non-functional requirement *availability* being an attribute belonging to the non-functional requirement of *security*.

Thus the evaluation with respect to non-functional requirements should take into account two aspects. Firstly, the definition related to the use-case, meaning that the non-functional requirement should be operationalized by stating goals that apply to the context. Secondly, the positive and negative relations of non-functional requirements amongst themselves should be taken into account.

3.2 Selection

For the evaluation, we will focus on a non-functional requirement. Due to its relevance in [22] [9] and [6], as well as its relation to the distributed architecture of using Event-streaming brokers in large-scale systems the selected non-functional requirement will be that of *reliability*.

SECTION 4 – DEFINITION AND EVALUATION

4.1 Definition

In the analysis of [10] *availability* is listed as an attribute of *reliability*. This interaction thus relates two of the requirements from section one. Another attribute listed in [10] is that of *fault tolerance*. This aligns with the descriptions of *reliability* given in [6] and [9]. In the use-case of Event-streaming broker in the given architecture *reliability* can thus be operationalized with the following goals:

-> the continued availability of service in case of failure defined by one or more unavailable nodes

-> the consistency and durability of data in case of failure defined by one or more unavailable nodes

-> the efficient failover and/or recovery in case of failure defined by one or more unavailable nodes

-> the guarantees above, additionally in case of failure defined by partitions in the network

4.2 Evaluation

This section examines the three Event-streaming brokers with respect to the qualities elicited in the previous section.

4.2.1 Redis

Redis uses a full-mesh cluster, where in a cluster on N nodes, each node is connected to N-1 nodes via TCP. Nodes utilize a gossip protocol to determine their mutual availability. Failover involves the election of new master nodes. In Redis, this is handled by master and replica nodes together in a majority vote fashion. After a node is found to be unreachable for a majority of the nodes, the master node no longer receives writes. To replace it, one of its replica nodes is promoted to be a master. In case of a network partition, only the majority portion which is connected will be available after a short time-out period. Thus Redis can only handle minor partitions before becoming unavailable. To be able to form a quorum and to help avoid a data inconsistency caused by a split brain condition, it is recommended to have an uneven number of nodes in a cluster. For three master nodes, two replicas each result in a total of nine nodes. Redis is single-threaded and therefore cannot benefit considerably in throughput by scaling vertically, which is why algorithmic sharding is used to distribute data across different nodes in the cluster, using hashing. This enables the parallel processing of operations across the cluster. Shards and their number can be redistributed when the number of nodes in the cluster changes. Data from a shard can be replicated to one or more replica instances in the cluster. Replication of data happens asynchronously between master nodes and their dedicated replicas. Because replication occurs asynchronously, writes are acknowledged to publishers before they are shared with replica instances. This leads to two possible data-loss scenarios. Firstly, a master node can acknowledge a write before it is replicated further and then fail. Secondly, a master node fails and gets marked as unreachable but then becomes reachable and receives a write before being demoted to a replica itself by a new master. From this follows that Redis cannot provide strong consistency of the data.

4.2.2 RabbitMQ

In the cluster, for each stream exists a leader node to which writes are directed. The remaining nodes receive replicates of the data and

serve read requests. This entails that a node can be a leader or replica depending on the distinct stream in question.
To tolerate network partitions, a quorum of nodes must be reachable in the cluster. For a three node cluster, this results in RabbitMQ being able to tolerate one complete node failure, while retaining availability. When a new stream is created, the leader node for that stream will be the node with the least leader roles. In case of the failure of a node, a replicate node will become the leader for the stream. When a node is reachable, it can rejoin and continue as a replica node for the stream. The acknowledgement of publisher messages in RabbitMQ occurs when the data has been replicated to a quorum of nodes in the cluster. Once received, data is persisted to disk and not kept in memory, however, data is not written to disk synchronously using fsync. Instead RabbitMQ relies on the operating system to schedule the writing on the operating system page cache. Thus it is possible that data is lost during a node failure, however, if another node which the data has been replicated to has not failed, the data will be replicated again. Should the node fail before the publisher receives an acknowledgement, then the aforementioned guarantees don't apply

4.2.3 Kafka

Certain aspects of clusters in Kafka are in part managed by an application called Zookeeper, such as naming, but in general the nodes, referred to as brokers, handle most of the functionality. Data in Kafka is replicated across all brokers. Publisher message acknowledgements, as configured, only confirm messages to publishers once they have been replicated across all three nodes. Through replication of data along with extensive publisher acknowledgements, Kafka is able guarantee that messages will not be lost as long as there is one node alive with a replica. Streams are not fully contained in any one node, but rather are partitioned in ordered segments across all nodes. One partition node is always considered the leader. All reads and writes occur through the leader for each specific stream topic. There are two conditions nodes must fulfill to be considered functional. Firstly, it should respond to the heartbeat requests managed by Zookeeper to signal its aliveness. Secondly, it should successfully receive writes from leader nodes in the cluster. Leader nodes maintain a list of functional replicas. Nodes that fulfill their functioning criteria are called "in-sync". A controller node is used to manage the registration of active brokers in the cluster and removes nodes which are not in-sync. Apache Kafka does not rely on a majority vote method to elect new leader nodes, instead all in-sync nodes are eligible to become leaders. This is possible due to the requirement of writes across all replication nodes, which guarantees they are not behind. In case that a node fails, Kafka does not require it to be consistent in order to rejoin. Before a reachable node can join the cluster again, it is required to become in-sync by default. This still implies that Kafka is not immune to failure due to a too high amount of network partitions. While Kafka guarantees data integrity with at-least one replica node being in-sync, this does not hold when all nodes become unavailable. The default strategy implemented in Kafka here is to wait until the first replica node that was listed as in-sync becomes reachable again and to promote it to the leader. This causes the system to be unavailable until this occurs. The consistency of the data is dependent on the state of the new leader, if the node incurred a form of disk error, the data is possibly inconsistent.

CONCLUSION

The three Event-streaming brokers Redis, Apache Kafka and RabbitMQ all employ different strategies to ensure reliability. They differ most in the degree of certainty they provide with regard to publisher acknowledgements with Apache Kafka providing the highest guarantees, followed by RabbitMQ and then Redis. Similar among all of them is that a large enough amount of network partition or node failures will lead to a sustained loss of availability. Apache Kafka and RabbitMQ provide a selection of settings to determine the degree of publisher acknowledgement and setting with regards to failover procedures which enable users to prioritize availability or partition consistency, while Redis is strongly focused on high availability, providing disaster recovery options rather than consistency. The choice of the in-sync set approach for leader election used by Apache Kafka allows for the loss of one more one in a three node cluster when compared to the majority vote approach used by Redis and RabbitMQ.

FURTHER RESEARCH AND THEATS TO VALIDITY

The research carried out in this paper is focused on the single requirement of reliability, in a general manner. Different cases of failure (hardware, software, malicious intent) should be taken into consideration. More requirements and their interrelation with reliability should be examined to get a clearer view of the design implications of the respective applications. The conclusions in this paper have been drawn based on the information found in the official documentation, this implies that a more thorough verification of the claims made therein could lead to different results. Guarantees and behaviors might deviate from documentation based on bugs and implementation errors.

REFERENCES

[1] Newman, S. (2021). In Building microservices: Designing fine-grained systems (p. 19). essay, Sebastopol, CA: O'Reilly Media.

[2] Rocha, H. a. O. (2022). Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices (p. 94) https://link.springer.com/content/pdf/10.1007/978-1-4842-7468-2.pdf

[3] Newman, S. (2021). In Building microservices: Designing fine-grained systems (p. 48). essay, Sebastopol, CA: O'Reilly Media.

[4] Bellemare, A. (2020). *Building Event-Driven Microservices (p. 28)*. O'Reilly

Media

[5] Bellemare, A. (2020). Building Event-Driven Microservices (p. 31,32). O'Reilly Media.

[6] Bellemare, A. (2020). Building Event-Driven Microservices (p. 28,29). O'Reilly Media.

[7] Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. (446) "O'Reilly Media, Inc."

[8] Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. (447) "O'Reilly Media, Inc."

[9] Fu, Guo & Zhang, Yanfeng & Yu, Ge. (2020). A Fair Comparison of Message Queuing Systems. IEEE Access. PP. 1-1. 10.1109/ACCESS.2020.3046503.

[10] Mairiza, D., Zowghi, D., & Nurmuliani, N. (2010). An investigation into the notion of non-functional requirements. https://doi.org/10.1145/1774088.1774153

[11] Tanenbaum, A. S., & Van Steen, M. (2023b). Distributed Systems (p. 15). Maarten Van Steen.

[12] Tanenbaum, A. S., & Van Steen, M. (2023b). Distributed Systems (p. 21). Maarten Van Steen.

[13] Dobbelaere, Philippe & Sheykh Esmaili, Kyumars. (2017). Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. 227-238. 10.1145/3093742.3093908.

[14] Hegde, R.G. (2020). Low Latency Message Brokers.

[15] [15] T, S. (2019, December 8). A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming. arXiv.org. https://arxiv.org/abs/1912.03715

[16] Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. (p. 450) "O'Reilly Media, Inc."

[17] Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. (p. 449) "O'Reilly Media, Inc."

[18] Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. (p. 221) "O'Reilly Media, Inc."

[19] Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (p. 352, 367) "O'Reilly Media, Inc."

[20] Joshi, U. (n.d.). Quorum. martinfowler.com. https://martinfowler.com/articles/patterns-of-distributed-systems/quorum.html last accessed: 25.6.2023

[21] Chung, L., & Leite, J. C. S. D. P. (2009). On Non-Functional Requirements in Software Engineering. In Springer eBooks (pp. 363–379). https://doi.org/10.1007/978-3-642-02463-4_19

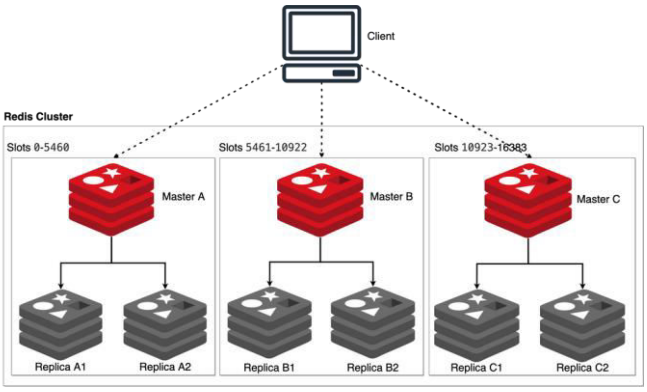
[22] Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. International Series in Software Engineering, vol. 5, p. 476. Springer, Heidelberg (1999)

APPENDIX

A1 – Table 2

property	Redis	Kafka	RabbitMQ
data streaming functionality release	2018	2011	2021
software license	Three clause BSD license	Apache 2.0	Mozilla Public License Version 2.0
version	open-source 7.0.11	open-source 3.11.17	open-source 3.4.0 using stream plugin
streaming protocol	RESP protocol	Kafka protocol	RabbitMQ Streams protocol
client	lettuce 6.2.4	kafka-client 3.4.0	stream-client 0.10.0
client language	Java	Java	Java
written in	C	Java / Scala	Erlang
stream retention policy options	by size, by age	by size, by age	by size, by age , by size and age
stream consumer semantics	non-destructive	non-destructive	non-destructive
stream semantic identifier	key	topic	stream name
OOTB publisher message deduplication	no	yes	no
message acknowledgement	yes	yes, configurable extend	yes
server-side consumer offset tracking	yes	yes	yes
replay / time-traveling	yes	yes	yes
optional TLS	yes	yes	yes
clustering support	yes	yes	yes

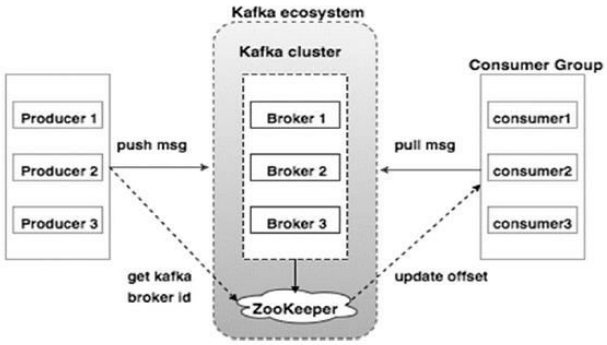
A2 – Figure 1



A3 – Figure 2


Stream 1 Leader
Stream 2 Replica
Stream 3 Replica
RabbitMQ node
Stream 1 Replica
Stream 2 Leader
Stream 3 Replica
RabbitMQ node
Stream 1 Replica
Stream 2 Replica
Stream 3 Leader
RabbitMQ node

A4 – Figure 3


Kafka ecosystem
Kafka cluster
Consumer Group
Producer 1
Producer 2
Producer 3
push msg
Broker 1
Broker 2
Broker 3
pull msg
consumer1
consumer2
consumer3
get kafka
broker id
ZooKeeper
update offset