FRANKFURT UNIVERSITY OF APPLIED SCIENCES
FACULTY 2: COMPUTER SCIENCE AND ENGINEERING

HIGH INTEGRITY SYSTEMS

Master Thesis

AN EVALUATION OF DIFFERENT OPEN SOURCE ESP PLATFORMS TOWARDS CONSTRUCTING A FEATURE MATRIX

Student:	Vo Duy Hieu
Matriculation number:	1148479
Supervisor:	Prof. Dr. Christian Baun
2^{nd} Supervisor:	Prof. Dr. Eicke Godehardt

February 22, 2021.

Official Declaration

I declare that this thesis has been written solely on my own. I herewith officially ensure that the work presented in the thesis is my own. I certify, to the best of my knowledge, my thesis does not violate anyone's copyright. Any external sources from the work of other people included in my thesis are fully acknowledged with citation and referencing.

Date	Signature

Acknowledgement

Abstract

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 2
- 1.3 Contribution . . . 2
- 1.4 Novatec Consulting GmbH . . . 2
- 1.5 Organization of this Thesis . . . 3

2 Background **4**
- 2.1 Event Driven Architecture . . . 4
- 2.2 Stream Processing . . . 6
  - 2.2.1 Stream Processing and Event-Driven Architecture . . . 7
- 2.3 Event Stream Processing Platform . . . 9

3 Evaluation Scheme **10**
- 3.1 Considered Platforms . . . 10
- 3.2 Evaluation Metrics . . . 12

4 Evaluation of Platforms **20**
- 4.1 General concepts . . . 20
- 4.2 Event storage . . . 23
- 4.3 Messaging patterns . . . 30
- 4.4 Messaging semantics . . . 38
  - 4.4.1 General prerequisite . . . 39
  - 4.4.2 Evaluation . . . 41
- 4.5 Stream processing . . . 48
- 4.6 Data integration and Interoperability . . . 51
- 4.7 Monitoring and Management . . . 54
- 4.8 Scalability . . . 55
- 4.9 Event storage . . . 60
- 4.10 Usability and Community . . . 65
- 4.11 Performance . . . 67
- 4.12 Feature matrix . . . 69

5 Implementation of Highest Delivery Guarantee **70**
- 5.1 Overview . . . 70
  - 5.1.1 Failure scenarios . . . 72

5.2 Implementation 74
5.2.1 Clients 74
5.2.2 Failure injection 80
5.2.3 Running environment 82
5.3 Results and Discussion 82

List of Acronyms **86**

List of Figures **87**

List of Tables **89**

List of Algorithms **90**

List of Listings **90**

Bibliography **91**

1 Introduction

Nowadays, the explosion of the number of digital devices and online services comes along with an immense amount of data that is auto generated or collected from the interactions of users. For instance, from 2016, the Netflix company already gathered around 1.3 PB of log data on a daily basis [1]. With this unprecedented scale of input data, companies and organizations have tremendous opportunities to utilize them to create business values. Many trending technologies such as Big Data, Internet of Things, Machine Learning and Artificial Intelligent all involves handling data in great volume. However, this also brings about a challenge to collect these data fast and reliably.

Once the data is ingested into the organization, it needs to be transformed and processed to extract insights and generate values. In the context of enterprise applications, as the systems grows over time with more services, the need for an effective data backbone to serve these huge amount of data to these services and to integrate them together while maintaining a good level of decoupling becomes inevitable.

Moreover, all these steps of collecting, processing and transferring data must be done in real-time fashion. One of the prominent methods is Event Stream Processing (ESP) which treats data as a continuous flow of events and use this as the *central nervous system* of the software systems with event-driven architecture.

1.1 Motivation

To develop a system evolving around streams of events, the primary basis is a central event store which can ingest data from multiple sources and serve this data to any interested consumer. Usually an ESP platform will be used as it is designed orienting to the concept of streaming. However, in order to choose the suitable platform, users will usually be burdened by a plethora of questions which need to be answered. The concerns include how well is the performance and reliability of the platform, does the platform provide necessary functionalities, will it deliver messages with accuracy that meets the requirements, can the platform integrate with the existing stream processing framework in the infrastructure, to name but a few.

As there are many platforms now available on the market both open-source and commercial with each having different pros and cons, it could be challenging and time-consuming to go through all of them to choose the most suitable option that matches the requirements. It would be greatly convenient to have a single standardized evaluation

of these platforms which can be used as a guideline during the decision-making process. Therefore, the goal of this thesis is to derive a feature matrix to help systematically determine the right open-source ESP platform based on varying priority in different use cases.

1.2 Related Work

There are a number of articles and studies which compare and weigh different platforms and technologies. However, most of them only focus on evaluating the performance between platforms [2] [3] [4].

Some other surveys cover more platforms and a wider range of evaluating aspects such as the comparison of Apache Kafka, Apache Pulsar and RabbitMQ from the company Confluent [5]. However, these assessments are conducted only briefly on the conceptual level. Apart from these studies, there is still lack of in-depth investigation into the differences of ESP platforms and their conformability with event-driven use cases and this is where the thesis will fill in.

1.3 Contribution

In this thesis, three open source ESP platforms, namely, Apache Kafka, Apache Pulsar and NATS Streaming are selected for evaluation based on preliminary measures and reasoning. Each platform is assessed against a set of criteria covering all important quality factors. The results are summarized in the form of a feature matrix with adjustable weighting factors of quality categories and features. Therefore, the matrix can be tailored to the needs of different users and adapted to individual use case to determine the most suitable platform for that case according to its priorities.

1.4 Novatec Consulting GmbH

This thesis is written with the support and supervision of Novatec Consulting GmbH. Novatec is an independent IT specialist with the headquarter located in Leinfelden-Echterdingen and 8 more branches across Germany and Spain. Its portfolio includes both development services and consultation in various aspects such as IT Architecture and Cloud, Data Intelligence, Business Process Management, Security.

The motivation for this thesis also arises from the real-world demand of the company to have a good and comprehensive evaluation of the most common ESP platforms for consultation purpose.

1.5 Organization of this Thesis

The thesis is organized as follows. Chapter 2 gives a short theoretical background of the topics event-driven architecture, stream processing and ESP platforms. Chapter 3 enumerates prominent open-source ESP platforms currently available and furthermore derives criteria to choose the top three platforms which are considered in this thesis. Moreover, it also includes the elaboration of comparison metrics. After that, chapter 4 presents the evaluation of each platform against the comparison scheme and gives a discussion on the resulted feature matrix. Finally, the conclusion summarizes and proposes future improvement for the matrix.

2 Background

In order to conduct the comparison effectively, it is necessary to first lay a good theoretical basis of event-driven architecture, event stream processing and the concrete roles of an ESP platform. Based on that, a comprehensive set of evaluation metrics can be determined.

2.1 Event Driven Architecture

In distributed systems or microservices applications, systems or services need to have a mechanism to coordinate and work together to achieve end results. There are typically two approaches for this task, namely, request-driven and event-driven [6].

In the first approach, a system sends command to request for state change or queries current state in other systems. This method is hard to scale because the control logic concentrates on the sender of requests. Adding a new system usually involves code change in others to include it in the control flow. For the latter approach, systems communicate with each other using events.

Event
An event represents something occurred in the past and cannot be altered or deleted [7]. It is simply a fact stating what has happened in the system. Whenever a system updates its state, it sends out an event. Any other system can listen and operate on this event without the sender knowing about it. This leads to inversion of control when the receiver of events now dictates the operational logic. As a result, the systems can be more loosely coupled. New service or system can be easily plugged and start to consume events while other services remain unmodified. There are three common patterns to use events, namely, event notification, event-carried state transfer and event sourcing [8].

Event Notification
The events are used only for notifying about a state change on the sender. Receiver decides which operation should be executed upon receiving the events. This usually involves querying for more information from either the publisher of events or another system.

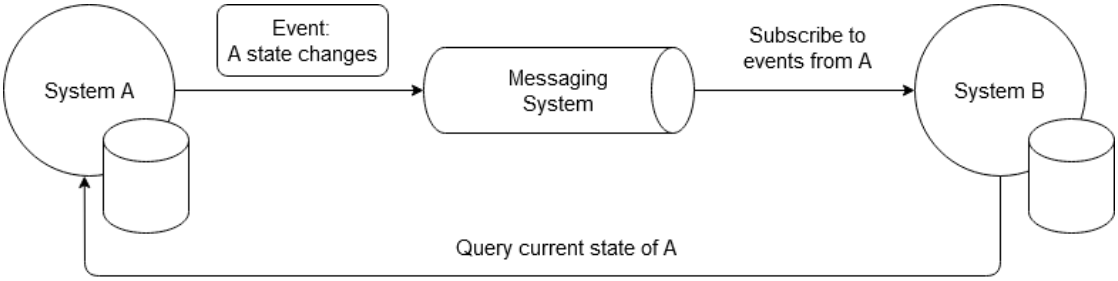


Figure 2.1: Systems coordination with event notification.

The approaches of event notification ensure a minimum level of coupling by letting each system manage its own data and only share when being requested. Another advantage is that the state of each system is consistent since it exists and can be queried from one place. Nevertheless, when systems grow with more functionalities, they need to expand the service contract to expose more data to outside which then leads to higher coupling. Therefore, the two following patterns tackle this problem by proactively allowing systems to openly share their data instead of encapsulating it within themselves.

Event-Carried State Transfer
With this pattern, each event encloses more detail information about what and how something has been changed. As a result, current state of any system can be reconstructed anywhere by applying its published events on the same initial state in the same order. Therefore, every subscriber can retain a local state replica and keep it synchronized with the source of events.

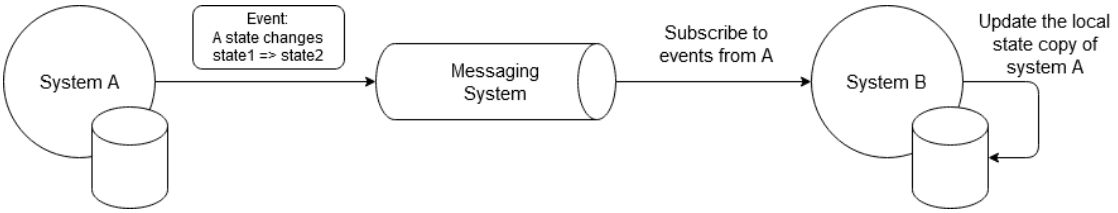


Figure 2.2: Systems coordination with event-carried state transfer.

When a system keeps a state copy of another locally, it can access this data faster and becomes independent of the online status of the source of data. Nevertheless, having copies of data across multiple systems also means that these copies can be in inconsistent state temporarily or even worse permanently if it is not designed carefully.

Event Sourcing
The event sourcing pattern takes one step further. Events are not used to only notify and help replicate states among systems but they are now the primitive form of data

store [9]. Instead of updating the current state, systems record every state change as events which are stored in the order of occurring in append-only fashion. This stream of events becomes the source of truth for all services and systems. Systems can derive the state of any entity from the events stream to query when needed. Snapshot of the current state can be periodically created to avoid processing through all events after every restart.

Figure 2.3: Systems coordination with event sourcing.

Apart from providing a loosely-coupled and scalable way for coordination among services and systems, event sourcing comes with additional benefits. The stream of events retains the entire history of the system and therefore can be used for auditing, extracting more value from past events, troubleshooting problem during production as well as for testing and fixing errors by simply replaying the events with the new system.

Event-driven architecture and its related patterns are useful but it also comes with a cost of increasing complexity in the system [10]. One of the most significant added values of this approach is to reduce the coupling and lay the foundation for systems with high demand for scalability. If this is not the case, a simpler approach should be chosen to not overcomplicate the system.

2.2 Stream Processing

With the increasing amount of data, the demand about how data is processed and analyzed also evolves over time. With the batch processing paradigm, data is collected over a predefined period of time and stored in a big bounded batch. Some scheduled batch jobs will then go over the entire batch of data to generate insights and reports tying to the needs of organizations. However, this type of data processing gradually cannot keep up with the need of faster analysis to allow companies to response more timely to changes. Therefore, the concept of stream processing begins to emerge.

Unlike its batch counterpart, stream processing aims at handling unbounded data which is a better form for representing the flow of events in real world given their continuous and boundless nature. Stream processing frameworks are designed to handle this type of data with high throughput and horizontal scalability. A stream processor can ingest one or more input streams and execute its business logic. It can also generate new data to other output streams which subsequently can be consumed by other stream processors. The

data flow is continuous and new stream processors can always be added to incorporate new processing logic and generate more results.

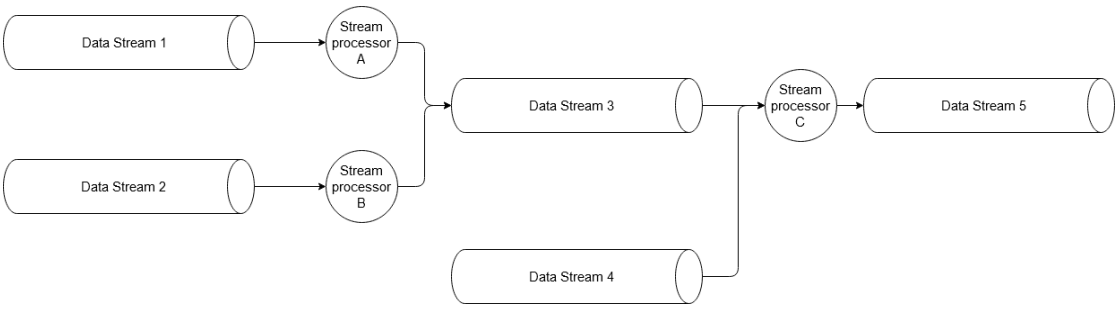


Figure 2.4: Stream processing concept.

By processing this influx of data continuously as they arrives, events and changes can be detected with low latency making stream processing more suitable for real-time use cases. Moreover, the input data can come from numerous sources with varying transmission rates. Therefore, data may arrive late and out of order with respect to the time it is generated at the source. In this case, for it sees data in an endless fashion, stream processing gives more tolerance for late data and more flexibility to assemble data into windows and sessions to extract more useful information. It is even suggested that a well-designed stream processing system with guarantee of correctness and the capability to effectively manage the time semantics could surpass batch processing system [11]. Back in the time when using stream processing was a trade-off between accuracy and low-latency, it was a popular idea to run two batch and stream processing pipelines in parallel to achieve both fast and correct results [12]. As stream processing engines grow more mature and accurate, the demand for such system is lessened .

2.2.1 Stream Processing and Event-Driven Architecture

Stream processing is not merely a data processing paradigm to achieve low latency result. Applying the concept of streaming and event-driven architecture also has the potential to help build more scalable and resilient software infrastructure for organizations. A big institution using software in every operational aspect usually has a sophisticated software infrastructure providing different functionalities such as business applications, query and analytical services, monitoring system, data warehouse. In this case, the problem of coupling between these processing and data systems will eventually emerge as the systems grows. Data needs to be shared and synchronized between these components. Without an efficient way to integrate system-wide data, the entire system can quickly turns into a big tangled mesh. As an example, this problem was experienced at LinkedIn as their system became increasingly complex [13].

Jay Kreps described in his article an approach that can solve this problem by building an infrastructure revolving around event streams [14]. Every event recorded by any

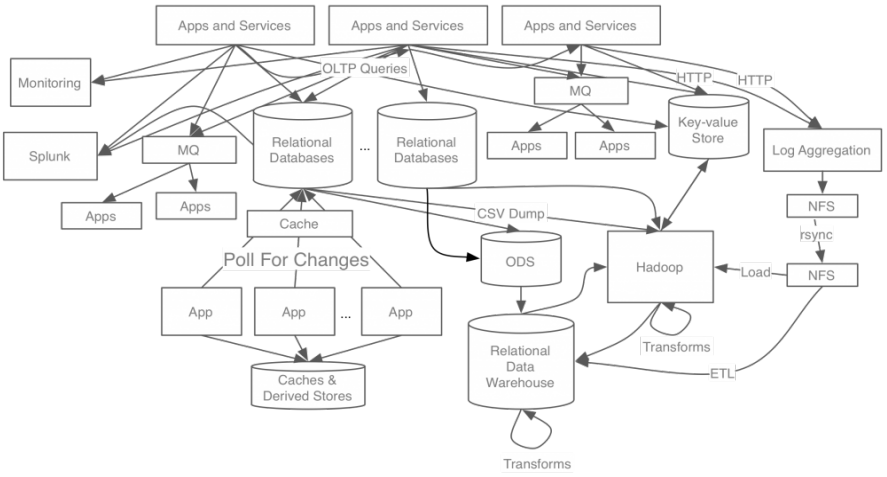


Figure 2.5: The tangled data systems at LinkedIn in the old day [13].

data system in the organization, is written orderly to an append-only log. This becomes backbone for all data systems in the organization. Each system has the flexibility to derive different data representations from the raw events to match its specific access pattern. The task of data representation is now done at individual system instead of in the central data store.

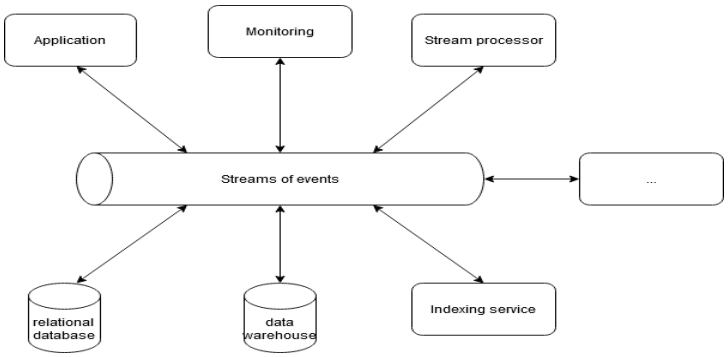


Figure 2.6: System with streams of events as the single source of truth.

This is basically applying event sourcing pattern on a much bigger scale. As a result, processing streams of events on an organizational scale becomes a non-trivial task. This is where stream processing fits in by providing a scalable tool to handle such amount of data. Applications and their decoupled services can use stream processing to continuously consume, process and generate new events while data systems can use streaming tool to continuously integrate new data from the streams to their data representation. The result is a more neatly organized infrastructure with every system synchronizes and communicates via the event streams using stream processing (figure 2.6).

2.3 Event Stream Processing Platform

An ESP platform must facilitate the construction of software systems revolving around streams of events. According to Jay Kreps, it has two main uses [13]. Firstly, it must provide the necessary infrastructure and tools for applications to work and coordinate with each other on top of events streams. Secondly, the platform can serve as the integration point where various data systems can attach themselves into and synchronize data among them continuously.

To fulfill these two uses, the following fundamental capabilities are required. A platform must have an events storage layer. Optimally, it should also support the option to persist events for an infinite period of time since this will be the single source of truth that all data systems depend on. Accessing interface must be provided for applications to publish and consume events. Based on this, custom applications and services can already be self-implemented to do stream processing as well as integrating data to different destinations. However, such platform only provides minimal functionalities and burdens developers with many low-level implementation tasks.

In order to be more useful and easier to be integrate into the infrastructure of organizations, an ESP platform must provide more supporting tools. More particularly, the platform should also support the processing of events streams either by providing a native stream processing tool or being able to integrate with external stream processing framework. For the data integration, the platform should come with ready-to-be-used tools to integrate with a wide range of existing data systems effortlessly including also legacy systems. Moreover,the platform should have a rich set of utility tools for monitoring and management.

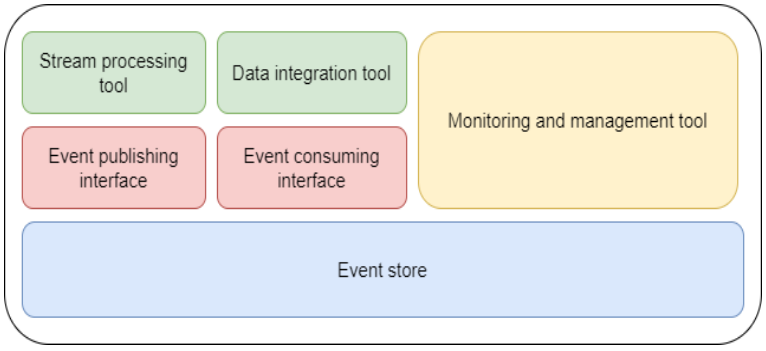


Figure 2.7: Capabilities of an ESP platform.

More importantly, all of these capabilities should be in a real-time, high throughput, scalable and highly reliable fashion as the entire infrastructure will rely on the platform for events coordination.

3 Evaluation Scheme

3.1 Considered Platforms

As the concept of using stream as a source of truth gains more attention and becomes more popular, many projects aiming at creating a processing platform evolving around streams started to take shape. Many companies first started their projects as in-house products and then later open-sourced them to enhance the development pace with the help of community. Kafka, which was first developed at LinkedIn, is the first prominent name in the field [15]. It was later open sourced to the Apache Software Foundation. Yahoo! also created their own stream processing platform named Pulsar and it is now also an Apache project [16]. The company Alibaba also joins the trend by open sourcing their RocketMQ to Apache Foundation [17]. In addition, there is the NATS streaming server, which is developed and maintained by the company Synadia on top of NATS messaging system to provide stream processing capability [18]. It is currently an incubating project of Cloud Native Computing Foundation. Pravega is also a quite new open-source project in the field [19]. All these platforms are released under Apache-2.0 License.

Since an adequate evaluation for all these platforms could not be contained within the scope of the thesis, only three platforms will be selected based on a set of preliminary criteria indicating the maturity, the size of community, the popularity of the platform and quality of documentation.

Maturity is evaluated based on the project stage in an open source foundation if it belongs to one and the date of the first release. Because each project has different in-house phase before open-sourcing, the date of release on GitHub is chosen as the standardized criterion.

The size of community is determined by the number of contributors of the project on GitHub and the number of related questions on Stackoverflow.

To assess the popularity, the number of GitHub stars and Google trend points of each project in the last 12 months are used.

The result of the preliminary evaluation is summarized in Table 3.1. The data and number presented in the table were collected in 11/2020.

According to these criteria, Apache Kafka and Apache Pulsar are the leaders in all preliminary categories. Among the three remaining platforms, RocketMQ is the most

Table 3.1: Preliminary evaluation of 5 open-source ESP platforms.

		Apache Kafka	Apache Pulsar	RocketMQ	NATS Streaming	Pravega
Maturity	Project stage in open-source foundation	Graduated project of Apache Software Foundation	Graduated project of Apache Software Foundation	Graduated project of Apache Software Foundation	Incubating project of Cloud Native Computing Foundation	Sandbox project of Cloud Native Computing Foundation
	The first release date on GitHub	11/2012	09/2016	02/2017	06/2016	09/2017
Community size	Number of contributors on GitHub	727	327	211	34	65
	Number of related questions on Stackoverflow	21 447	144	57	47	0
Popularity	GitHub stars	17 300	6800	12700	2100	1300
	Google trend points	40	72	5	2	2
Documentation	Quality of the content	Very Good Update regularly with each release Cover in detail design and features Blogposts and books from Confluent, a company founded by the creators of Kafka	Good Update regularly with each release Cover in detail design and features	Bad Do not update regularly, some content is even from 2016 Cover only a few examples	Medium Update regularly Good coverage of general concept and features	Medium Update regularly with each release Good coverage of general concept and features

popular. However, the document page of RocketMQ is unfortunately very inadequate and outdated. This could become a great problem during the evaluation. Therefore, RocketMQ will not be considered in the thesis. Regarding Pravega, despite the promising set of rich features, it is not mature enough since it is still in the sandbox stage at the Cloud Native Computing Foundation. Moreover, the community of Pravega is still too small. On the other hand, although NATS streaming has a moderate community, it has a reasonable maturity and quality of documentation. Moreover, it is built on top of NATS messaging system which is used in industry by many companies such as Siemens, Pivotal, GE and therefore can inherit its stability. The evaluation of NATS streaming can be greatly beneficial for organizations which already have NATS messaging in the infrastructure to integrate NATS streaming more easily.

As a result, the three platforms which will be inspected in depth are Apache Kafka, Apache Pulsar and NATS streaming.

3.2 Evaluation Metrics

The evaluation categories for distinct functionalities of an ESP platform are derived based on the necessary capabilities in section 2.3. Since there are not many major scientific works on evaluating ESP platforms, concrete criteria which are directly related to ESP platforms cannot be found. Therefore, criteria in these categories are determined mainly by self-deducing from the literature research of related technology to each capability of an ESP platform or event-driven concept and from interviewing with experts at the company Novatec who work intensively with ESP platforms.

For general functionalities such as security and non-functional criteria, the comparison categories are adapted from ISO25010 software quality model to have a good coverage of main quality aspects [20]. However, non-applicable characteristics from the standard such as maintainability which refers to the inner structure and complexity of the system will not be included. As a result, 11 main comparison categories are determined.

1: Event storage: this category is directly corresponding to the event store capability of the platforms. In this category are the criteria to evaluate this storage functionality.

1.1: Durable storage: with event-driven application, the event stream is the backbone that all application and services depend on. Therefore, events on an ESP platform must be stored in non-volatile storage. It must be guaranteed that once an event is confirmed to be persisted, it must survive system crash or power outage.

1.2: Flexible data retention policy: For event-driven application, events are the source of truth of the systems. They can be consumed, replayed by numerous services and applications at different rates and times. Therefore, the platform must support a long data retention period to give services enough time for events consumption. The platform should also support selective retention of data since this can be useful for certain use cases to save disk storage.

1.3: Data archive in cheap storage (hot/cold storage): For use cases where the entire history of events must be retained, it is also desirable to have the option to automatically offload old and low-demanded data to cheap storage to save cost. On the other hand, newer data can still be retained in hot storage on the platform to be served to clients faster.

In order to evaluate the capabilities to allow publishing and subscribing events of the platform, two evaluation categories are derived, namely, messaging patterns and messaging semantics. These categories include evaluation criteria taken from the concept of messaging systems which are corresponding to this functionality on the ESP platforms.

2: Messaging patterns: In this category, the evaluation of various possibilities to deliver events from producer to consumer with the platform as the messaging middleware is considered. This is directly related to the concept of asynchronous messaging and is dictated by the messaging patterns supported by the platform. Therefore, most common and related patterns of message delivery will be used as evaluating criteria for these platforms.

2.1: Publish-Subscribe [21]: With this pattern, a message is delivered to multiple consumers, each of which will receive the same set of messages. This pattern is very relevant in the context of event stream processing since one event stream can be consumed by numerous independent receivers and each of them requires an entire history of events for a different processing logic. Therefore, it must be supported by the platform.

2.2: Competing consumers [22]: This pattern is very important to allow multiple consumers to consume a messaging channel to balance the load and increase throughput. Each message will be delivered to only one of the competing consumers. This is suitable for the event-driven use case where each event in the stream is self-contained and can be processed separately such as using event to trigger a reaction. The number of events to be processed can be significant, the capability to scale with competing consumer is very important and therefore this pattern is included in the evaluation.

2.3: Publish-Subscribe + Competing consumers (Consumer group) [22]: With Competing consumers pattern, a message can be delivered to any available consumer to maximize concurrency and throughput. In this scenario, each message can be interpreted and processed individually. For certain event-driven patterns such as event-sourcing, it is essential to have the entire history of events to reconstruct the system state. The Publish-Subscribe pattern is more suitable for this purpose. However, an event stream can retain large data volumes with events of many different entities which could overwhelm the processing capacity of a subscriber. Therefore, it can be very useful if the platform supports the combination of two patterns Pub-Sub and Competing consumer to scale each subscriber of an event stream. In this way, events can be consumed by several competing consumers where each consumer will receive only events from a subset of the entities belong to that event stream. This pattern is sometimes referred as consumer group.

2.4: Event playback: This is not directly related to asynchronous messaging but is very important in event-driven architecture. With event streams as the source of truth, it is also very important for any consumer to be able to re-consumes older events. The ability to replay events is indeed one of the key features of Event Sourcing pattern [9]. Therefore, the platform must support this access pattern for replaying past events with various starting points, namely, specific position in the event streams or specific time point.

2.5: Content-based routing [23]: With this pattern, the consumer of a message channel can selective receive based on some information in the messages. With event-driven approach, each event stream usually retains event of the same type but from multiple source entities. It is very common that some downstream consumer only wants to receive events coming from a specific source. If the pattern is supported on the platform, it can be very useful in such scenarios.

3: Messaging semantics: This category focuses on the concern of correctness of delivered data. Forwarding message from producer to consumer alone is not enough. The messages must also arrive on the consumer side correctly with a certain level of delivery guarantee and be interpretable by the consumer.

3.1: Strong ordering guarantee: For events, the order is very important to correctly reproduce the system state from event streams because events represent state changes in the. Events in different orders will result in different system states. In distributed systems, failures can happen anytime causing delays, retries and hence out-of-order. To really achieve order guarantee requires the cooperation between the platform and clients and also compromise on throughput. The platform must lay a good basis to allow collaboration to guarantee order of events traversing from producer, through the platform and to the end consumer.

Message delivery guarantee is another hard problem in distributed systems. In an unreliable environment, failures are unavoidable which can lead to connection disruption while sending messages. There are three different levels of guarantee in such scenarios a messaging system can provide, namely, at-least-once, at-most-once and exactly-once with different tradeoffs between performance and reliability. An ESP platform should allow users to choose different tradeoffs depending on their use cases. Therefore, these guarantee levels are adapted as evaluation criteria for the platforms.

3.2: at-most-once delivery semantics: If message lost can be tolerated, at-most-once delivery guarantee is sufficient. In case of failures, the message can simply be dropped which results in message loss but there will be no duplication on the consumer side. For example, for stock prices or temperature measurements, the new values are updated very frequently and therefore it is acceptable to miss some values occasionally.

3.3: at-least-once delivery semantics: If a message must never be lost and always be processed, at-least-once guarantee should be chosen. In case of failures, message must be resent which may lead to message duplication. The consumer is guaranteed to receive each message at least once.

3.4: exactly-once semantics: exactly once delivery on the other hand cannot be physically achieved because systems communicating over an unreliable channel like network will never be ensured about the status of the published message [24]. This criterion actually evaluates seemingly exactly-once processing capability instead of exactly-once delivery guarantee. More particularly, it is possible that a message can be redelivered and reprocessed by consumer multiple times. However, the result of processing should be the same

as when message is received and processed exactly once. This feature is very important in mission-critical application such as financial transaction.

3.5 and 3.6: Support schema records, Schema management and evolution: events or messages in general usually have certain structure with specified fields. This is very important for consumer of messages to be able to interpret them. Therefore, the platform should support sending and receiving schema records with various common schema data serializing systems such as Avro, Protobuf, Json. In distributed systems where producers and receivers of data are independent, it is very important that they are in agreement about the schema of records to avoid mismatch interpretation which can lead to non-processable records. Moreover, the structure of the messages and their schema can be upgraded over time to adapt new requirements. Therefore, it is necessary to have a schema management system which handles the evolution of schema overtime and enforce validation rules to ensure the compatibility of new schemas to both producer and consumer. Thus, platforms should also support schema management mechanism.

4: Stream processing: The criteria in this category are used to evaluate how well the platform provide the stream processing capability on its persisted event streams.

4.1: Native stream processing: This can be a very useful feature to develop stream processing applications that integrate better with the platform. If it is provided by the platform, detail elaboration and evaluation will be conducted to see if important functionalities are supported such as: windowing function, time semantics, aggregation functions.

4.2: Integration with external stream processing frameworks: Apart from native stream processing, it should be also possible to integrate the platform with existing and mature stream processing frameworks such as Apache Spark, Apache Flink. This criterion will help determine the available integrable frameworks of each platform.

4.3: Simple, high-level stream processing: In addition to native stream processing and external stream processing frameworks, it would be useful if the platform also supports stream processing capability in the form of simple query to help people with little software development background quickly gain insight into the events streams.

5: Data integration and Interoperability: in this category, the ability of the platform to integrate and share data with different types of systems and clients are evaluated.

5.1: Connectors to external systems: since the streams will be the data backbone of the system, the platform should have a standard framework to ingest and export data with various data systems. Moreover, a wide range of ready-to-be-used connectors should also be supported to ease the need of self-implementing integration service.

5.2: Supported programming languages for client: to help services and applications integrate easier, an ESP platform should support a wide range of clients in different programming languages. This factor could also be important for the decision making of a company to use it or not. This criteria will be evaluated based on number of currently supported clients for each platform.

6: Monitoring and Management: in this category, the set of operational tools provided by each platform will be inspected.

6.1: Technical monitoring: during operation of the platform, it is very important to keep track of the health of nodes in the cluster via metrics such as: CPU usage, Memory used, messages throughput so quickly react to changes. Therefore, it is very important to be able to set monitoring system for the platforms.

6.2: Event tracing monitoring: in addition to technical monitoring, it can also be very useful to have monitoring systems on a higher level to give an overview and quick inspection of the flow of events through the platform and also the content of each event.

6.3: Admin API: to help user manage and configure the platform, administration API should be exposed and rich set of managing tools should be provided.

6.4: Professional support: although the evaluated technologies are all open-source, it is not always convenient for users to self-manage everything to maintain and operate with the platform. Users might want to delegate the tasks of deployment, management of the platform to service providers to focus more on developing their business logic. Therefore, the number of available managed services of these platforms is also a good evaluation criterion to consider.

7: Scalability: as the data backbone, an ESP platform will usually have to handle a huge amount of data. Therefore, it should be easily scalable to quickly adapt to new demand of data volumes.

7.1: Scalability of storage and computing server: the two fundamental functionalities of an ESP platform are persisting events and serving read/write requests from clients. The demands for storage and messaging consumption could vary greatly. Therefore, these two layers should be scalable and optimally can be scaled independently with minimum manual administering from users.

8: Security: this category of criteria is adapted from the ISO25010 quality model. This is very important for any data system in general. In case of ESP platform, security is a very important factor since the platform is the data backbone for the organization which can retain many sensitive data which needs to be protected.

8.1: Authentication mechanism: this is one of the fundamental security mechanisms to verify the legitimacy of clients connected to a system. The platform should support authentication mechanisms. They could be built-in mechanisms which only work within the

platform. More optimally, the platform should support pluggable mechanisms of common authentication scheme such as Simple Authentication and Security Layer (SASL) so that it can be integrated more easily to the existing security infrastructure in the organization.

8.2: Authorization mechanisms: different clients of the platform can have different levels of access rights. Therefore, the platform should support authorization mechanisms to control access of clients. There are two common types of access control, namely, Role-Based Access Control (RBAC) which defines access rights based on business roles, and Access Control List (ACL) which allows more fine-grained control on the level of individual clients. The platform should support these mechanisms.

8.3: Encryption: this criterion covers the confidentiality of data on the platform. There are different levels of encryption, namely, encrypted transmission, encryption of data at rest and end-to-end encryption. The evaluation will be conducted based on the number of supported encryption levels on each platform.

8.4: Multi-tenant: In case multiple teams and departments rely on an ESP platform to operate, multi-tenant feature needs to be supported. Otherwise, different cluster will have to be set up for each individual team which then increase the cost of operating and maintenance. Therefore, this is a nice feature to have on an ESP platform.

9: Reliability/Recoverability: this non-functional category is adapted from the quality model and aims at evaluating the failover mechanisms of the platforms in case of failure of different components, namely, data store, client, serving layer, and the entire data center.

9.1: Event storage is fault tolerant: As the data integration point for the system, there must be no single of point failure for the event storage. Therefore, the platform should provide failover in case of failure of data storage so that the events can continue to be served to applications and services.

9.2: Consumer group failover mechanism: If the platform supports the consumer group in criterion 3.3, there should be failover mechanism within the group in case of one or more consumers fail to avoid disrupted consumption.

9.3: Broker failover mechanism: the platform should handle the case when one or more server instances fail. In this case, the requests from clients should not be disrupted.

9.4: Geo-replication: for significant systems that span across the globe, an ESP platform should support out-of-the-box data replication between different data center not only to increase the response time for client but also for fault-tolerance and high availability in case one of the data center is down.

10: Usability and Community: this category is also derived from the ISO25010 standard. This refers to how fast and easy for users to get used to the platform and

start to integrate it into their system. This can be a very important factor when choosing an ESP platform.

10.1: Active community: for usability and learnability, the activeness of the community is very important in addition to its size. A community of many active developers could be a great source of support. Moreover, a platform with an lively community can gain more contributions such as detecting issues, adding new features. This can help speed up significantly the development and enhance the quality of the platform.

10.2: Available training courses: in addition to available document, there should be a good selection training courses available.

10.3: The ease to start the development: It should be easy for developers to quickly start development, build prototype applications with the platform and so forth.

11: Performance: this category is taken from the ISO25010 quality model. This focuses on how good the platform can perform its functions while guaranteeing an acceptable processing speed.

11.1 and 11.2: End-to-end latency, Throughput: since the demand for low latency in stream processing is very high, the platforms should have reasonable end-to-end latency to deliver messages from producer to consumer. Moreover, as the data backbone, huge data volumes from all kinds of systems and applications will pass through the platform and therefore, high throughput must be guaranteed. These two criteria will be evaluated based on literature research since there are already many works on comparing time behavior of these ESP platforms. The platforms will be sorted based on their time behaviors with the platform with the best performance in the first place and the platform with the lowest performance in the third place. The grading will be done accordingly.

After deriving this set of criteria, a discussion was held with experts at Novatec to determine which criteria in each category are more essential for an ESP platform or of greater importance in their daily works with an ESP platform. These criteria are marked as high priority. In the scope of the thesis, only these criteria will be the focus and be evaluated in-depth.

For different use cases, the priority might be different. Therefore, despite not being considered in detail, the other criteria could serve as a general guideline to compare and evaluate in different scenarios.

The assessment of the platforms will be organized according to the evaluation categories. However, for the Reliability / Recoverability category, since it is closely related to the non-functional behaviors of other functionalities, the criteria in this section will be considered together with other categories when fitted.

Table 3.2: Considered evaluation criteria in the thesis.

No.	Evaluation category / Evaluation criteria
1	**Event storage**
1.1	Durable storage
1.2	Flexible data retention policy
2	**Messaging patterns**
2.1	Publish-Subscribe
2.2	Competing consumers
2.3	Publish-Subscribe + Competing consumers (Consumer group)
2.4	Event playback
3	**Messaging semantics**
3.1	Strong ordering guarantee
3.2	At-most-once delivery semantics
3.3	At-least-once delivery semantics
3.4	Exactly-once semantics
4	**Stream processing**
4.1	Native stream processing
5	**Data integration and Interoperability**
5.1	Connectors to external systems
5.2	Supported programming languages for client
6	**Monitoring and Management**
6.1	Technical monitoring
7	**Scalability**
7.1	Scalability of storage and computing server
8	**Security**
8.1	Authentication mechanisms
8.2	Authorization mechanisms
8.3	Encryption
9	**Reliability / Recoverability**
9.1	Event storage is fault tolerant
9.2	Consumer group failover mechanism
9.3	Broker failover mechanism
10	**Usability and Community**
10.1	Active community
11	**Performance**
11.1	End-to-end latency
11.2	Throughput

4 Evaluation of Platforms

4.1 General concepts

Before going into comparison, the general concepts of the three platforms will be quickly recapped to provide a basis about the working mechanism of these platform. More detail elaboration of their features will be presented during the evaluation.

Apache Kafka

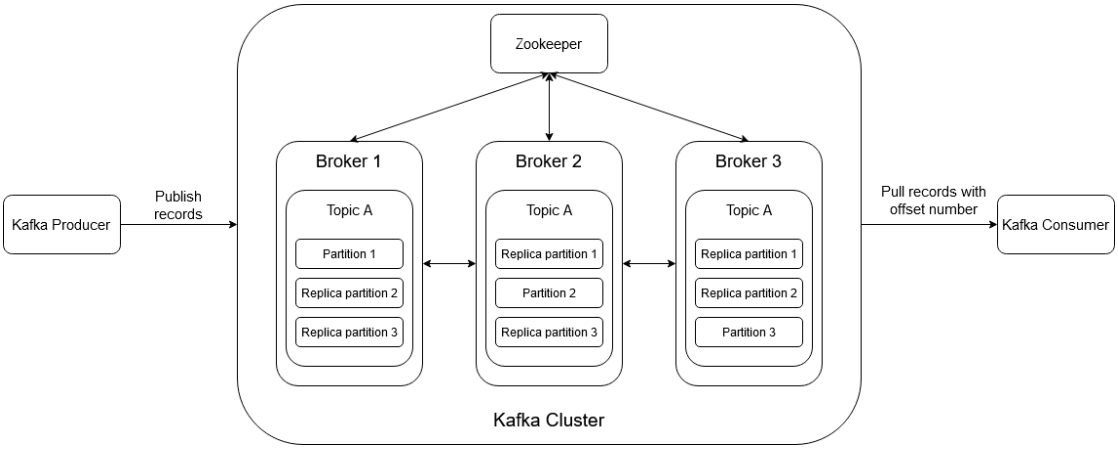


Figure 4.1: General concept of Apache Kafka.

Apache Kafka is a platform designed specifically for event streaming. There are a number of fundamental concepts and core components of Kafka:

- Record: this is the name for message published to Kafka. Each record can have a key, a value, and some metadata. This is also usually referred as event or message.
- Kafka broker: this is the heart of Kafka. A broker is in charge of serving read/write requests and persisting published messages from clients to its disk. Kafka usually runs in cluster with three or more broker. In the current release of Kafka, the cluster also include one or more Apache Zookeeper [25] nodes to maintain metadata of these brokers. However, Zookeeper will soon be removed completely from Kafka in later release and metadata will be maintained natively on Kafka broker instead [26].

- Topic and partition: Records are organized into different topics on Kafka. A topic further comprises of multiple partitions, each of which is an append-only and immutable log. New records will be appended to the end of the log. Each record in a partition is uniquely identified by an incremental offset number. There are two types of partition, namely, leader and follower. Read and write operations will be done on the active leader partition. Follower partitions are replicas of the leader which reside on different brokers in the cluster and cannot serve requests. Since Kafka 2.4, it is possible to read records from follower replicas as well [27]. Nevertheless this feature is disabled by default.
- Kafka clients: There is Producer API which is used to publish records to Kafka topics. On the other hand, Consumer API is used to read records from a Kafka topic.

In this thesis, the current release 2.6.0 of Kafka will be evaluated.

Apache Pulsar

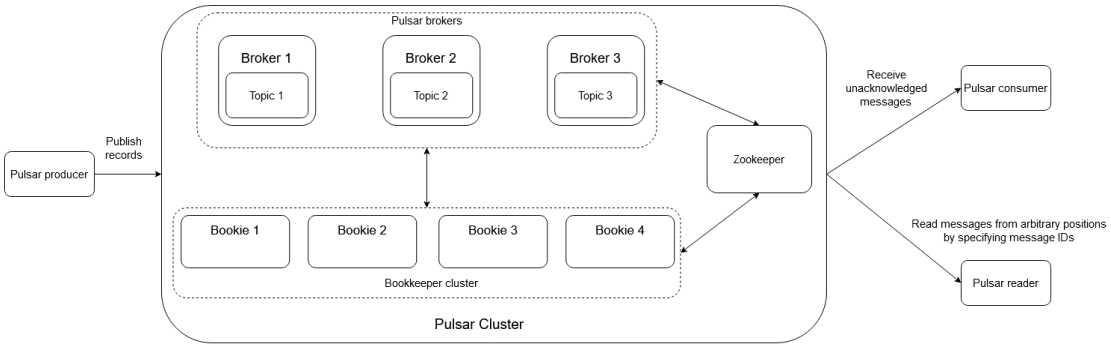


Figure 4.2: General concept of Apache Pulsar.

Apache Pulsar is designed as a multiple-purposed platform by combining the concept of traditional messaging and event streaming. Following are the main concepts of Pulsar:

- Message: a message published to Pulsar has a key-value format along with some metadata.
- Pulsar broker: this component is responsible for serving read/write requests from clients and send persisting request of messages to the persistence layer. There are usually many brokers run together in a cluster.
- Apache Bookkeeper [28]: this is the persistence layer of Pulsar. Bookkeeper handles the durable storage of messages upon receiving requests from the Pulsar broker. Messages are stored on Bookkeeper ledgers which are immutable logs with new records being appended to the end. The Bookkeeper usually runs in a cluster with multiple nodes which are called Bookies.

- A Pulsar cluster is made from a cluster of Pulsar brokers, a cluster of Bookkeeper nodes and also a number of Zookeeper nodes for metadata management of these clusters.
- Topic and partition: Apache Pulsar organize records into different topics. Each broker is responsible for read/write request of a different subset of topics. A topic can also be split further into multiple partitions. However, a partition is internally also a normal Pulsar topic which is managed transparently to user by Pulsar. Records in a Pulsar partition or a non-partitioned topic are uniquely identified by message IDs.
- Pulsar clients: Messages can be published to Pulsar topic with Pulsar Producer API. For messages consumption, there are two different client APIs, namely, Consumer API and Reader API. Each of these consumption clients has different level of flexibility and can be used in different cases. The detail comparison of Consumer and Reader is given in the evaluation section of messaging patterns.

The current release 2.7.0 of Pulsar will be evaluated.

NATS Streaming

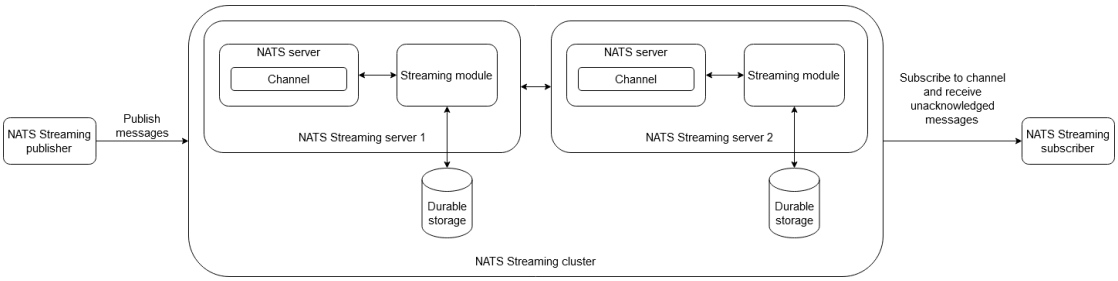


Figure 4.3: General concept of NATS Streaming.

NATS Streaming is an event streaming add-on built on top of NATS server which is a messaging system without persistent layer. It has some core concepts:

- Message: a message published to NATS Streaming contains a value and some metadata.
- NATS Streaming server: A NATS Streaming server comprises of a normal NATS server as the messaging layer and a a separated streaming module which is internally a normal NATS client to receive and persist message on NATS server to a pluggable durable storage. Because of this special structure, the clients and streaming module only interacts indirectly via the intermediate NATS Server and all requests must first traverse through the NATS Server. NATS Streaming comes with an embedded NATS server but can also be configured to work with an existing NATS server. A number of NATS Streaming servers can be grouped together into a cluster in two modes: clustering and fault tolerance. In the first mode, each

server retains a full copy of all messages in a separated data store. In the latter mode, nodes in the cluster share a single data store.

- Channel: On NATS Streaming, messages are organized into channels which internally are made from append-only message logs. A message in a channel can uniquely be identified with an incremental sequence number.
- NATS Streaming client: NATS streaming provides client API to publish and subscribe to messages on channels.

In the thesis, the latest version 0.19.0 of NATS Streaming is used for assessment.

Moreover, each platform provides different implementations of its clients in different programming languages. Nevertheless, to have a uniform benchmark, when the evaluation involves comparing clients, the Java implementations will be used since this programming language is officially supported by all three platforms.

4.2 Event storage

Apache Kafka

Durable storage

In Kafka, records are stored in partitions each of which is stored on the disk as a number of segment files [29]. The broker receives records and writes them sequentially to these files. When a segment file reaches its configurable maximum size, a new file will be created for writing new records. Since all records are only appended to these files, the read/write of records only requires sequential I/O on the disk. This is one of the key design features of Kafka to help maintain fast performance even on cheap hard disks.

However, a Kafka broker can already acknowledge writing requests when records are written to I/O buffer and not necessarily when records are persisted to disk. Therefore, durability is not guaranteed, and message loss can still happen when the broker fails before flushing records to disk. User can force disk flush to ensure durability whenever a message is received but this is not recommended by Kafka since it can reduce the throughput of the system [30]. To achieve durability, a more common approach is combining this unflushed write feature with redundant write on other brokers in the cluster which is the fault-tolerance feature for storage provided by Kafka. This is elaborated in more detail in the next section.

Event storage is fault tolerant

As briefly mentioned in the general concept, there are two types of Kafka partitions, namely, leader and follower. The leader partition is active and can serve read/write requests from clients while followers are standby replicas of the leader. For fault tolerance, Kafka supports data replication among brokers in the cluster [31]. For each topic, user

can specify a replication factor which determines the number of existing copies of records on Kafka. When replication factor is 2 or more, every partition of the topic will have 1 leader and 1 or more followers. For each partition, each of its copies will be distributed on a different broker in the cluster. Therefore, there will be no single point of failure for record storage. The replication factor must be equal or smaller than the number of brokers in the Kafka cluster.

However, enabling only data replication on the broker cannot guarantee that all records are safely replicated on the Kafka cluster. By default, a Kafka producer sends a record and only waits for the acknowledgement of successful write from the leader partition. If the broker with the leader partition goes down before the record is flushed to its disk and replicated to other follower replications, the record may be lost without the producer knowing about it for resending. Therefore, the producer must be strictly configured to wait for acknowledgements from the leader partition as well as other follower replicas. In this case, the leader partition will also wait for acknowledgements from its followers before confirming with client. By having the redundant acknowledgements, durability is guaranteed even when messages are not yet persisted to disks given that all brokers retain the partition do not fail simultaneously.

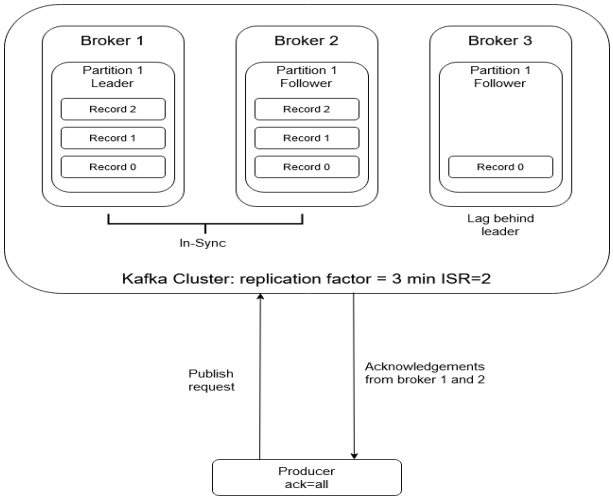


Figure 4.4: Data replication model for fault tolerance of event storage on Kafka.

For each replicated partition, the follower periodically fetch data from leader to stay in-sync. If the producer waits for acknowledgements from all replicas, some followers may fall too far behind the leader for instance due to network connection and increase the waiting time of the producer. Therefore, the leader of the partition dynamically maintains a list of in-sync replicas (ISR) which contains itself and all followers which currently stays synchronized with it. This list is stored on Zookeeper. When a follower does not catch up with the leader by sending fetch request after a configured amount of time, it will be removed from the ISR of the partition. The slow follower can rejoin ISR

later when it has fully caught up with the leader partition. In practice, the producer will only wait for acknowledgements from the ISR instead of all replicas. This aims at balancing between the durability, fault-tolerance of published records and the latency for acknowledgement. As a result, a message acknowledged to producer can survive up to *ISR-1* failed nodes and still be available to consumer.

It could be possible that all followers of a partition are out-of-sync with the leader. In this case, producer only receives one acknowledgement from the leader which brings back the problem of losing messages. Therefore, Kafka also provides the option to configure the minimum number of in-sync replicas. If the ISR falls below this number, new writing requests will be rejected, and availability is compromised to ensure the durability.

Durability and fault tolerance of data storage on Kafka are closely related to each other and can only be achieved with the right configurations on both Kafka brokers and Kafka producers. In addition, Kafka provides many configuration options to give users the flexibility to choose different priorities for their systems such as availability, durability, latency.

Flexible data retention policy
All records published to Kafka will be retained. Old data can be cleaned up with different cleanup policies [30]. These policies can be configured on the broker level which will then be applied to all topics or they can be configured differently for individual topic. There are two basic strategies to retain data:

- Delete: All records are retained for a period of time or based on a maximum size and then they will be deleted.
- Compact: This strategy is only applicable to records with key values. The topic will be compacted. Only the latest record of each key value is retained.

For the first strategy, user can choose different retention policies for a partition based on maximum size or for a segment file of a partition based on retention period. Once the retention limit is exceeded, oldest segment file of the partition will be deleted. By default, there is no limit on the size of a partition and the retention period is 7 days. User can also configure infinite retention period if necessary.

For the second strategy, the background cleaner thread of Kafka will regularly scan the segment files and keep only the latest record for each key value (Figure 4.5). Therefore, this only works with records with non-empty key value. Writing request of record without a key to a topic configured with compact cleanup policy will be rejected.

The two strategies can also be combined. With this setup, topics will be compacted regularly but when the retention maximum size or maximum retention period is reached, the data will also be removed regardless of being compacted or not. For example, in case of an online shopping application where each order is kept track by a sequence of events published to Kafka as records, when users are interested in the latest status of an order

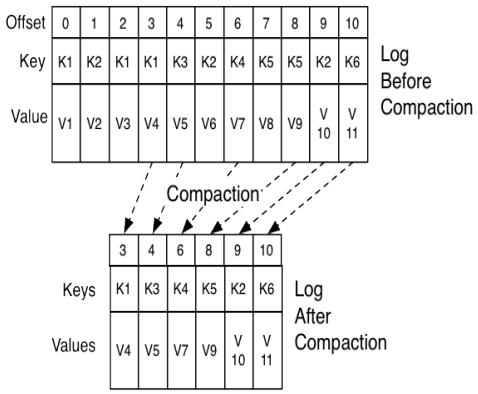


Figure 4.5: Log compaction on Kafka (taken from Kafka documentation [32]).

but only if the order is not older than 3 months, it is reasonable to this combination of retention strategies.

To sum up, Kafka provides a flexible way to retain all records or only selective data. Users can choose the appropriate strategy based on their use cases.

Apache Pulsar

Durable storage

A Pulsar topic can be persistent or non-persistent which must be specified by user when creating the topic. Messages on non-persistent topics are only kept in-memory on the Pulsar brokers. For persistent topics, Pulsar provides the persistence layer using Bookkeeper. A Pulsar broker has a Bookkeeper client internally. When receiving writing requests from clients, Pulsar broker sends persistent requests to the Bookkeeper cluster. Bookkeeper provides the storage abstraction called ledger. A Pulsar topic is made up from one or more ledgers. New messages will be appended to the end of a ledger. Once a ledger reaches its maximum size or the Bookkeeper node (Bookie) is restarted, a new ledger will be created. Internally, a Bookie store messages of a ledger in an entry log file on its disk. Durability of data on a Bookie is guaranteed once an acknowledgement is sent back to Pulsar broker. The broker then can confirm the successful write with client.

Event storage is fault tolerant

Pulsar supports replication of messages of a topic on multiple Bookies for fault-tolerance. To achieve this, Pulsar utilizes the built-in replication mechanism of Bookkeeper. A Pulsar topic comprises of one or more Bookkeeper ledgers. Each ledger can be further made from one or more fragments. When a ledger is created, it must have three important configuration options which control how messages are written and replicated on Bookkeeper cluster [33]:

- Ensemble size (E): An ensemble is a set of Bookies which are selected randomly from the Bookkeeper cluster to persist records for a fragment of the ledger. Whenever one node in the ensemble fails to accept write requests, a new fragment with a different ensemble without the failed node is created for the ledger to ensure that there are enough available Bookies for writing. The ensemble size can be configured by user and must be equal or smaller than the number of nodes in the Bookkeeper cluster.
- Write quorum size (Qw): Every record in a fragment will be written to Qw nodes in the ensemble so that each record will have Qw copies for fault tolerance. Qw can be equal or smaller than E. If it is smaller than E, every record will be written to a different subset of nodes in the ensemble.
- Acknowledge quorum size (Qa): This number specify the number of nodes in the Qw set which must acknowledge before a message is considered to be successfully persisted. Write request is acknowledged by Pulsar broker when Qa Bookie nodes have confirmed receiving the message. This option provides a possibility to balance between performance and the persistence guarantee. With this configuration, it is guaranteed that a message can still survive and be available in case $Qa - 1$ Bookies are destroyed.

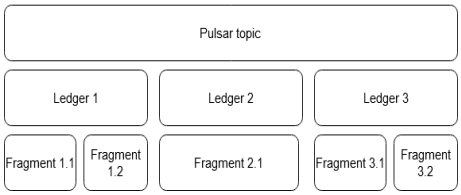


Figure 4.6: Underlying storage layers of a Pulsar topic.

Although all these configuration options are from Bookkeeper which is used internally by Pulsar, Pulsar also allows users to configure them using its administrator tool to achieve the required level of fault tolerance in different use cases. When there are not enough Bookies to meet the configured ensemble size and quorum, writing request from clients will return an error. Moreover, whenever a Bookie node dies, fragment with records written on that node will not have enough Qw copies. In that case, if the auto recovery is enabled [34], the Bookkeeper cluster can auto detect the failed node and replicate records on that Bookie to others to maintain Qw replicas for each record.

Flexible data retention policy As briefly mentioned in the general concept, a Pulsar topic can be read using Pulsar consumer. Whenever a consumer receives and processes successfully a message, it needs to send an acknowledgement back to the Pulsar broker. Based on that, Pulsar has two kinds of messages:

- Able to delete: Messages which have been acknowledged by all consumers of the topic and messages on topic with no active consumers.

- Unable to delete: Messages which have not been acknowledge by all consumers of the topic.

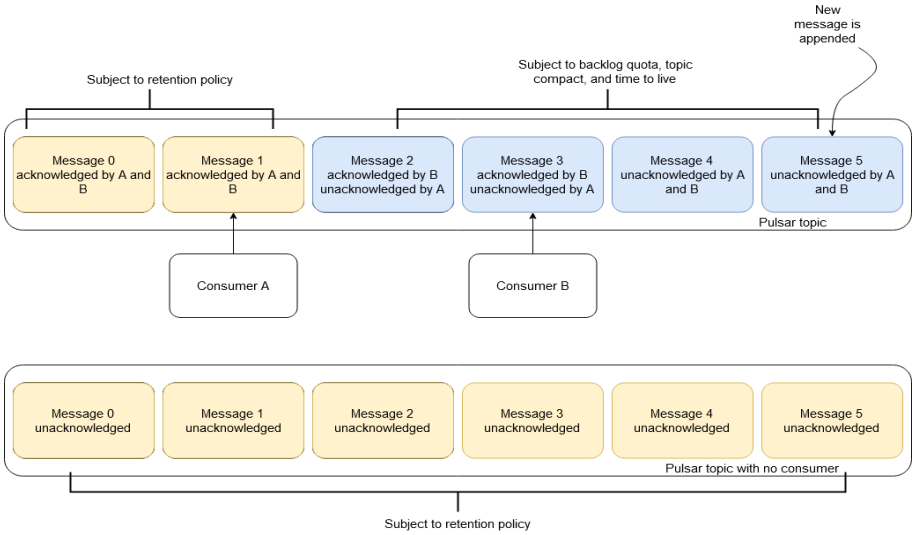


Figure 4.7: Message retention policy on Pulsar.

By default, messages of the first type will be deleted the next time Pulsar does cleanup. Pulsar can be configured to retain these messages using retention policy [35]. User can set a time limit or a size limit for the retained messages on the topic. Whenever they exceed this limit, old messages will be deleted to keep the messages always within the limit. The size and time limits can be configured to be unlimited as well.

The second type of messages will always be retained by default. However, their size can grow too large. This can be controlled using backlog quota or time to live (TTL).

The backlog quota set a size limit on allowed unacknowledged messages of a topic. If this limit is exceeded, user can choose to reject new write request to topic or delete oldest unacknowledged messages. The main purpose of this configuration is not to save disk space. It aims at regulating the sending rate of producer in case slow consumers fall behind when consuming new messages by rejecting new sending request or reduce the number of unread messages.

On the other hand, TTL option focus on saving the disk space. It set a living time for unacknowledged messages. When the time expires, messages will be auto acknowledged to be subject to delete.

Pulsar also provides the option for topic compaction [36]. However, this is completely unrelated to saving disk space and in fact will increase the disk usage. In the compaction process, Pulsar will scan through the unacknowledged messages and make a new copy containing only latest message of each key value. Messages without key values will be

overlooked by the compaction. This new compacted copy can be read by consumer when only latest values are relevant to speed up the processing.

These retention policies are only relevant to Pulsar consumer with its acknowledgement mechanism. In case of Pulsar reader, it does not acknowledge the consumption of messages to Pulsar and therefore does not determine which messages will be retained or deleted.

In summary, by default, messages on Pulsar will be deleted after being consumed by Pulsar consumer. However, it can be configured to retain messages as long as needed. Moreover, there is no option to selectively keep only latest messages to save disk space.

NATS Streaming

Durable storage

NATS streaming provides a pluggable persistence layer for durable storage of messages [37]. However, by default, persistence storage is disabled and NATS streaming server only stores messages in-memory. NATS Streaming must be explicit configured to use durable storage.

There are currently two storage options supported out-of-the-box which are file store and relational database store which are referred as SQL store in the NATS document. With file store, messages are stored in files on the disk of the server or in a network filesystem (NFS) mounted on the server. A directory is created for each channel in which messages of that channel are stored in log files. On the other hand, with SQL store, messages are persisted as records in an external relational database. All messages published to NATS Streaming are persisted in a messages table and each of them is uniquely identified by the ID number of the channel to which it belongs and an incremental sequence number. With these two provided storage options, once the producer of a message receive acknowledgement from the server, it is guaranteed that the message is durably persisted. Moreover, NATS Streaming also provides a storage interface which can be self-implemented by user to connect NATS to a different data store.

Event storage is fault tolerant

NATS Streaming has two different clustering modes [18]. In fault tolerance mode, all server instances are mounted to the same shared data store such as NFS or a shared database table depending on which persistence layer is used by the servers. NATS Streaming does not support data replication for fault tolerance of data with this mode. The shared data store can become the single point of failure. Once it goes down, data is not accessible or even worse lost. If fault tolerance of data is required, it relies entirely on users. For instance, users can implement data replication on the persistence layer themselves or use a fully managed storage service such as Amazon Elastic File System.

Fault tolerance of event store is provided out-of-the-box by NATS Streaming with the clustering mode. In this mode, each node maintains a full copy of data in a separated

data store. The nodes in NATS streaming cluster use Raft Consensus algorithm to replicate data [38]. Only the Raft leader node can take care of receiving requests for all channels and make copies on other nodes. Users only have to start up a NATS streaming cluster with a number of nodes and data will be auto replicated among them. Producers of message will receive acknowledgement once the replication process is finished. With Raft Consensus algorithm, a cluster with *2n+1* node can tolerate up to n node failures and continue to operate while guaranteeing no message loss. If there are more failures, the NATS Streaming cluster cannot accept new messages. It is recommended by NATS to limit the cluster size to only 3 or 5 nodes.

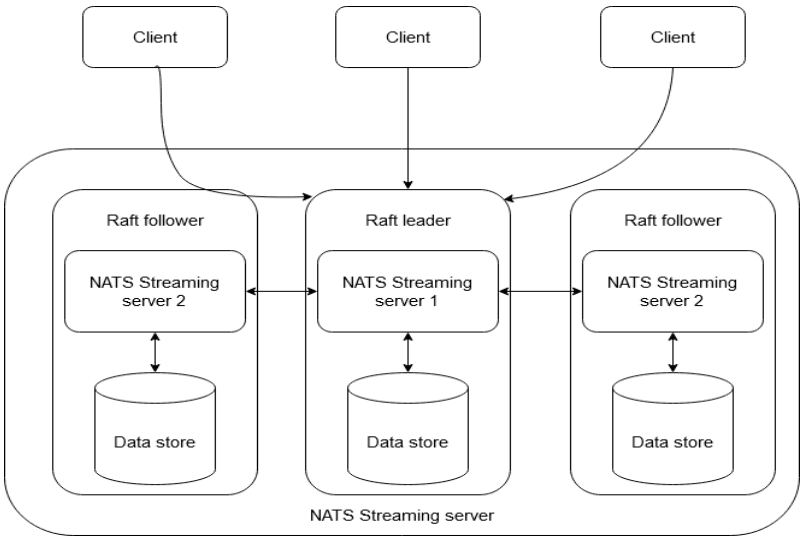


Figure 4.8: Fault tolerance of event storage on NATS Streaming in clustering mode.

Flexible data retention policy
If persistence storage is enabled, all messages published to NATS Streaming will be retained whether they are consumed or not. User can specify the maximum number of channels, maximum size or number of messages of a channel, maximum retained time of each message [37].

When the limits are exceeded, oldest messages will be deleted until the retained data falls below the maximum limitation. All of these policies can be set to unlimited to retain messages forever. However, there is no option to selectively retain messages on NATS Streaming server.

4.3 Messaging patterns

Apache Kafka

Records on a Kafka topic can be read using Kafka consumer [39]. Each consumer belongs

to one consumer group. A topic on Kafka can be simultaneously consumed by multiple consumer groups, each of which will receive all messages from that topic.

A consumer group can comprise of multiple consumers and each message on the subscribed topic will be delivered to only one of them. The important point is that the messages are not distributed randomly to consumers in the group. Instead, each partition of the topic is assigned to one consumer in the group. At any time, each partition will be assigned and read by only one consumer in the group.

Each new record appended to a partition is assigned an offset number to be uniquely identified. This offset number is also used by Kafka consumer to indicate its current reading position on the partition. Unlike many traditional messaging systems, Kafka uses pull model to deliver messages to consumer. That means Kafka broker does not keep track of what has been consumed by consumer with acknowledgement and therefore does not actively deliver unread messages to consumer. It is the responsibility of consumer to provide an offset number indicating its current reading position on every request to pull new batch of messages from that point onward from Kafka. With this approach, the consumer has full control about its reading position. For durability of the consumption status, Kafka supports automatic or manual committing the offset numbers of consumers to internal Kafka topics. Consumer also can maintain its reading position in a different durable storage such as an external relational database.

Publish-Subscribe
It is very straightforward to realize the publish-subscribe pattern with the concept of consumer group of Kafka. New subscriber for a topic can be created by creating a new consumer group with only one consumer and all messages on all partitions of that topic will be delivered to this subscriber.

Competing consumers

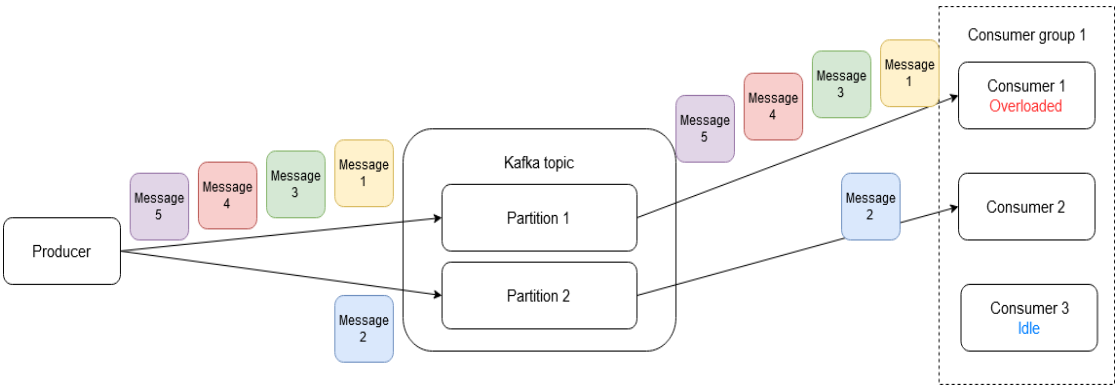


Figure 4.9: Competing consumers pattern with Kafka.

The competing consumers can be realized using consumers within the same group. By

adding multiple consumers to one group, they can concurrently consume messages of the subscribed topic and each message will only be read by one consumer. However, since the partition is the unit for parallel consumption in one consumer group, the number of competing consumers will be limited by the number of partitions of the topic. If there are more consumers in the group than partitions, some consumers will be unoccupied. More partitions can allow more parallel consumers. However, if there are too many partitions, this could degrade the performance of the system [40].

Moreover, if the messages are not evenly distributed across the partitions with some retain more messages than the others, some consumers will have to handle more workload while the others are assigned with empty partitions will remain idle. Therefore, the competing consumers pattern provided by Kafka is quite rigid and cannot be scale freely.

Publish-Subscribe + Competing consumers (Consumer group)
Although it can be used as either publish-subscribe or competing consumers pattern for normal messaging scenarios, the concept of consumer group and partition itself is a combination of both these patterns provided by Kafka specifically for the event-driven use cases. By assigning each partition of a topic to a consumer in the consumer group, events of the same type published to the topic can be read and processed in parallel.

Kafka provides a forwarding mechanism on the producer side to guarantee that events from the same entity will be delivered to the same consumer. Messages in Kafka have the key-value form. If message key is defined, it will be used to create a hash value which will then be used to determine the destination partition of the message. When an identifier is assigned to each event source and used as the key for published events, these events will end up in the same partition on the destination topic. As a result, the consumption of events can be scaled by tuning the number of partitions and consumers in the group while ensuring that each consumer will receive the entire history of events from a specific entity.

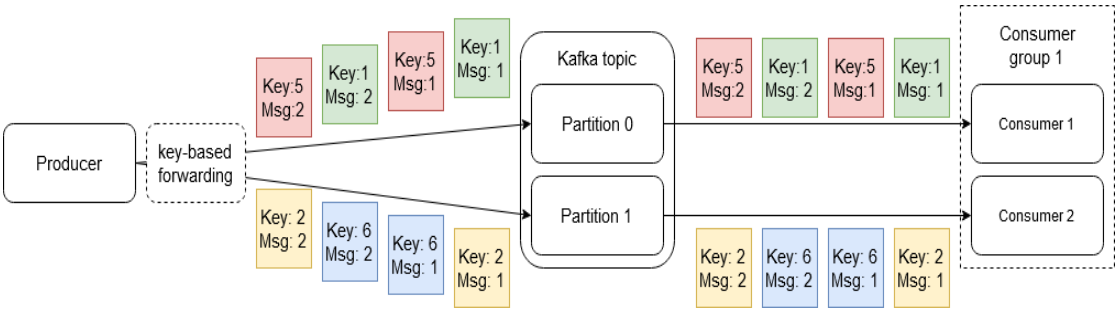


Figure 4.10: Consumer group pattern with Kafka.

Consumer group failover mechanism
Consumers in the group periodically send heartbeats to Kafka to indicate their liveness.

Moreover, they must also regularly send new requests for messages from Kafka. If a consumer fails to meet either condition, it is considered to be failed and will be removed by Kafka from the group [41].

In case a consumer fails, its partitions will be reassigned to other consumers in the group automatically. The failover consumer will continue to process messages from the latest reading position of the failed consumer. If this reading position is checkpointed in Kafka, the task of retrieving the consumption status of failed consumer and sending it to the taking-over consumer will be managed transparently by Kafka. On the other hand, if the failed consumer persists its position in an external datastore, Kafka provides the callback *ConsumerRebalanceListener* for consumer to determine when partition rebalancing occurs and retrieve the last reading position on newly assigned partition.

Event playback
A Kafka consumer must specify its reading position using offset number in every request to pull new messages from Kafka. Therefore, it is very straightforward to re-consume older records in Kafka by simply using an older offset number for the new consuming request.

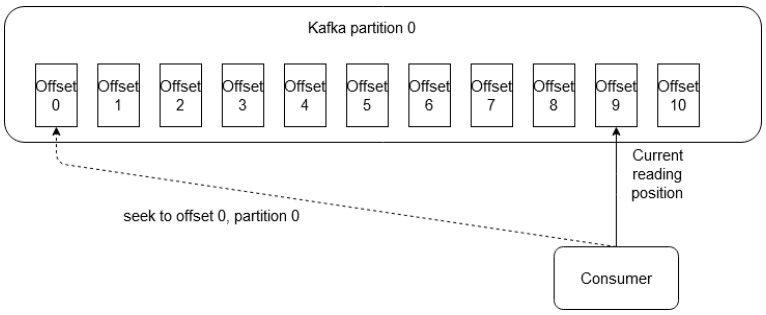


Figure 4.11: Event playback with Kafka.

By default, if a consumer is started with an existing consumer group, its reading position on the assigned partition starts from the latest committed offset number on either Kafka or external datastore, and the position will advance after every request for new messages. However, a Kafka consumer can also reset its reading to an arbitrary position [41]. This reset operation is done on partition basis. A consumer can only reset the offset of partitions that it is currently assigned to. Next time the consumer pulls more records from this partition, it will start from the reset offset position.

Apache Pulsar

There are two ways to consume message from Pulsar, namely, using Pulsar consumer and Pulsar reader.

The consumption of messages with Pulsar consumer is quite similar to traditional messaging system. Pulsar uses a combination of push and pull models to deliver messages

to consumer [42]. Acknowledgement must be sent back to Pulsar broker upon successful processing of a message. The broker will keep track of the consumption status of consumers and will send unacknowledged messages to a queue on the consumer side when it receives permission from the consumer. Messages on this queue is dequeued and consumed gradually by the application. Once the queue is halved, Pulsar consumer sends a new request for new messages to be pushed to the queue again. Moreover, Pulsar broker also uses the acknowledgements to mark messages as being able to be deleted.

Consumers subscribe to a topic by creating a subscription [43]. Multiple consumers can be grouped together with the same subscription to consume messages on the topic together. There are four different subscription modes, namely, Exclusive, Failover, Shared and Key_Shared each of which is relevant to a different messaging pattern supported by Pulsar. However, with the current release of Pulsar, the Key_Shared subscription mode is still unstable [44]. Therefore, this mode will not be considered further in the thesis.

On the other hand, the Pulsar reader is internally a Pulsar consumer with special configurations [45]. The Pulsar broker does not monitor the consumption status of reader and does not require acknowledgement of messages. Therefore, Pulsar broker does not control which messages to be delivered to reader. On starting up, the reader must define a specific starting position by giving the ID of the first message to be read and only messages from that point onward will be delivered to the reader. This reader is designed to give users more control over which messages to be read from Pulsar and is quite similar to Kafka consumer. If the reader needs durable consumption status, it has to maintain its own current reading position in a durable storage and manually retrieve that on every startup. Pulsar does not support auto-checkpointing reader position natively on Pulsar as in Kafka.

However, there is no possibility like subscription to group multiple readers together. Each reader connects directly to a topic and starts consuming all messages individually. Therefore, it is not possible to realize messaging patterns of shared consumption such as competing consumers and consumer group with Pulsar reader. Moreover, at the moment with Pulsar 2.7.0, a Pulsar reader can only read from non-partitioned topics.

Publish-Subscribe
Pulsar provides the Publish-Subscribe pattern with the Exclusive subscription mode of consumer. In this mode, at any time, only one consumer is allowed in the subscription, other consumers which join the subscription later will be rejected. All messages on the subscribed topic will be delivered to the single consumer in the subscription. Different subscribers can create a new exclusive subscription to the topic with different subscription names.

The publish-subscribe pattern can also be realized with the Pulsar reader. Multiple readers on a non-partitioned topic can simply be started and every message on the topic

will be delivered to all of them.

Competing consumers
This pattern is realized on Pulsar with the Shared subscription mode. Multiple consumers can be grouped together using this mode. Messages will be distributed to the consumers in a round-robin fashion and each message will be delivered to only one consumer in the group.

Publish-Subscribe + Competing consumers (Consumer group)
Like Kafka, messages on Pulsar have the key-value form. If a message is created without a key, it will be delivered to a random partition of the topic. But if a key value is assigned, its hashed value will be used to determine the target partition for the message on the topic. On Pulsar, to enable the combination of Publish-Subscribe and Competing consumers, on the producer side, key values must be assigned to all generated events. They will be used as identifiers for events from the same source and help deliver them to only one consumer.

On consumer side, Failover subscription mode can be used to realize this pattern. In this mode, multiple consumers can be grouped together into the same subscription. Each partition of the subscribed topic will be assigned to only one consumer in the group. Only if a consumer fails, its partitions will be taken over by another consumer in the subscription. By adding key to each event on the producer side, it is ensured that events of the same entity will be on the same partition and therefore a consumer in the subscription will receive all of them. This is similar to the consumer group in Kafka and it requires the topic to be partitioned to enable this parallel consumption. The level of parallelism is also limited by the number of partitions of the topic.

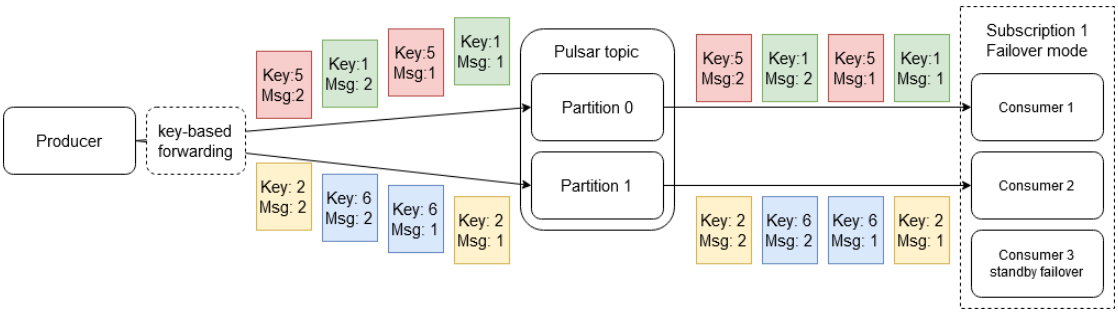


Figure 4.12: Consumer group pattern with Pulsar.

Consumer group failover mechanism
In Failover mode, each partition of the topic is assigned to one consumer in the subscription. Pulsar broker automatically monitors the connection to every consumer.

If a consumer disconnects, its partitions will be assigned to other consumers automatically by Pulsar. Moreover, because Pulsar broker keeps track of the consumption statuses of all consumers with acknowledgement, it will know which messages have not

been read by the failed consumer and continue to deliver them to the failover consumer.

Event playback
For Apache Pulsar, event playback is possible with both Consumer and Reader API.

Pulsar consumer must acknowledge the consumption of each message. It is the responsibility of the broker to decide which messages will be delivered based on the acknowledgements received from the consumer. By default, if a consumer is started with an existing subscription, the reading will automatically be started from the earliest unacknowledged message onward. Nevertheless, it is also possible to reset the consuming position of the subscription with Consumer API [46]. User can reset reading position to a message with a specific message ID. The message ID must be known beforehand. It is also possible to reset the reading position of the subscription to message published at the time equal or greater than a specified timestamp.

With Pulsar reader, a starting position of messages to be read from the topic must be specified every time it is started. As a result, the reader has the flexibility to jump to arbitrary positions and replay events on the topic whenever needed [47]. The starting position can be defined by providing a message ID. User can also specify a rollback duration to rewind the reading position to a specific time point. The times semantics of both consumer and reader is when the messages were published and were automatically assigned by the Pulsar producer.

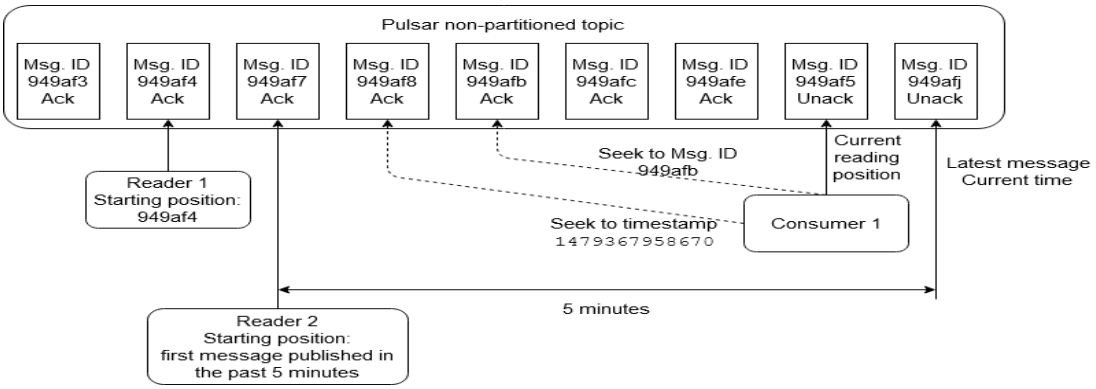


Figure 4.13: Event playback with Pulsar.

NATS Streaming

With NATS Streaming client API, a consumer must create a subscription to a channel to be able to start consuming messages [48]. NATS Streaming provide two subscription modes. The first is normal mode which means there is only one consumer per subscription, and it will receive all messages on the channel. The second mode is queue mode where a subscription has multiple competing consumers which will read the same topic and each message will be read by only one consumer.

NATS streaming uses push model to deliver messages to subscriber. That means the server controls which messages to be sent to consumer. The server keeps up with the current consumption status of each consumer by receiving acknowledgements. NATS server actively pushes next unacknowledged messages to the consumer and expects that the receiving application is available to process these messages right away. Consumer must acknowledge with server about the successful consumption of every message within a predefined time window. Otherwise, unacknowledged messages will be redelivered to subscriber after timeout.

By default, NATS server only retains the consumption status of a consumer during the connection session. When being disconnect, the current reading position of the consumer will not be remembered by the server. In this case, if the consumer wants to resume its consumption from the previous session, it must maintain its own reading position in an external storage and provides it as the starting position to read when reconnecting to the server.

To have a durable consumption status on the server which will be automatically resumed when the consumer reconnects, the subscription must be strictly configured to be durable. When a subscriber starts consuming messages using an existing durable subscription, it will resume its consumption of the unacknowledged messages from the previous connection session.

Publish-Subscribe
In NATS Streaming, a new subscriber to a channel can be set up by simply starting a consumer in normal mode with a new client ID and all messages on that channel will be delivered to this new consumer.

Competing consumers
Competing consumers for a topic can be grouped together into a subscription with queue mode. NATS Streaming will deliver messages to consumers in the group randomly and each message will be sent to only one consumer. The number of consumers in a queue group can be scaled freely.

Publish-Subscribe + Competing consumers (Consumer group)
NATS Streaming offers no combination of Publish-Subscribe and Competing consumers patterns. The subscriber in Publish-Subscribe pattern is not scalable. On the other hand, with the competing consumers pattern, it is not guaranteed that all events from one source will be delivered to the same consumer because of the random distribution of message in the group. If parallel consumption of events is needed, it must be self-implemented by user on the channel level. In this case, all the task of partitioning events of the same type into multiple channels, forwarding events from the same source to the same channel and assigning each channel to one subscriber will have to be done manually.

Consumer group failover mechanism
Since the consumer group pattern is not supported by NATS Streaming, there is also no

failover mechanism.

Event playback
Event playback with NATS Streaming is very straightforward. When creating a subscriber to a NATS channel, user has the ability to choose different starting positions for the subscription [49]. Consumer can start consuming messages from a specific sequence number on the channel, from a specific starting time or from a rewind duration. The time semantics here is when a message is stored on the NATS server and is managed by the server.

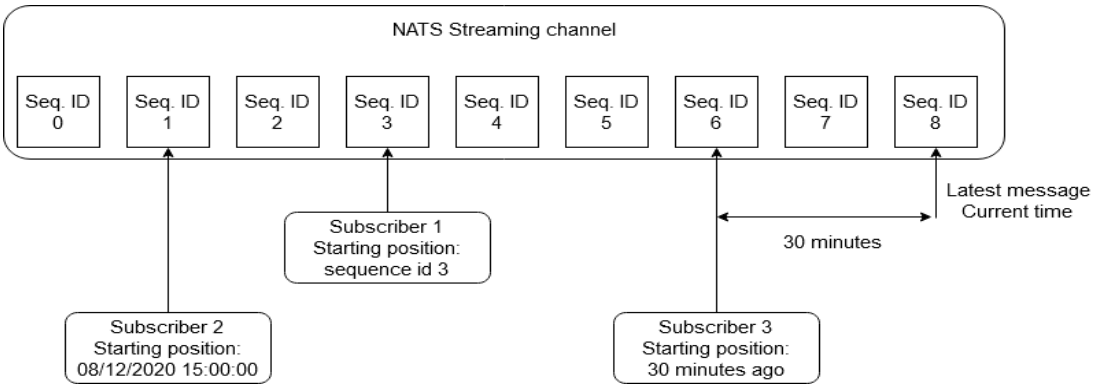


Figure 4.14: Event playback with NATS Streaming.

By default, the subscription on NATS Streaming is not durable. Thus, when a subscriber disconnects to the server, its current reading position will be lost. As a result, user can choose a different starting position every time the subscriber is restarted and can freely jump to different messages on the channel. In case of durable subscription, the current reading position of the subscriber is kept track by the server. Therefore, to replay events, subscriber must first unsubscribe the durable subscription before it can specify a new reading position using the same subscription name.

4.4 Messaging semantics

In systems with asynchronous messaging, messaging semantics such as ordering guarantee, delivery guarantee cannot be provided by the message broker alone. The producers and consumers of messages must collaborate with the messages broker to achieve different levels of semantics. Therefore, in this section, before going into the evaluation, the general prerequisite for collaboration of the clients, which is true for all ESP platforms, is briefly recapped.

4.4.1 General prerequisite

Strong ordering guarantee
To achieve ordering guarantee of messages, the throughput must be compromised. On the message publishing side, at any time, messages from the same source must be published from only one producer instance. Otherwise, since concurrent producers publish events independently and are subject to different network conditions, the order of events cannot be preserved when they arrive at the ESP platform. Similarly for consumers, to guarantee that messages from the same source are processed in the correct order, concurrent consumption with multiple application instances is not permitted.

To have both concurrency and ordering guarantee, the concurrent instances must coordinate with each other to enforce the right order when publishing or consuming messages. Nevertheless, this method can introduce more overhead to the system.

Moreover, when a producer asynchronously send a message and proceed to the next one before receiving acknowledgement from the broker, if a message fails to reach the broker and is resent later, it is possible that another message may arrive in between causing their orders to be interchanged.

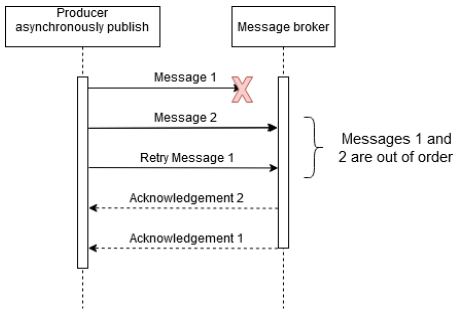


Figure 4.15: Asynchronous publishing and retrying causes out-of-order messages.

Therefore, in case the broker does not support detect out-of-order of asynchronously published messages, producer should only synchronously publish message one by one for ordering guarantee.

At-most-once and At-least-once delivery semantics
When a message is published, it can be lost along the way from producer to broker. Therefore, it is the responsibility of the producer to keep tracks of the sending status and resend messages if necessary. On the receiving side, after receiving and processing a message received from the broker, a consumer usually must perform two operations, namely, durably persisting the result of the processing and updating the current reading position on the source message stream to indicate that it has finished with the current message and will not receive it again. Since failures can non-deterministically occur

between two operations, different order of these operations can result in different delivery guarantees.

With at-most-once semantics, a producer can simply send messages in a fire-and-forget fashion without waiting for acknowledgements of successfully receiving from the message broker. Therefore, one of the key benefits of at-most-once guarantee is higher throughput since the producer is not impeded by the overhead introduced by acknowledging. When failures occur, messages may not reach the platform and are lost. When consuming a message, consumer must update its current reading position before committing the calculated result from the message. In this way, if the consumer goes down before the result is persisted, it will continue with the next message upon restarting while the previous message will not be received again, and the corresponding result is simply lost.

For at-least-once semantics, messages producer must wait for acknowledgement from server for every message and is responsible for resending to avoid message loss. As a result, within one connection session, a message can appear on the message pipeline multiple times in case it is already received by the broker but the acknowledgement to the producer is lost. On the consumption side, for each received message, the consumer must mark it as being consumed and update the reading position only after it has handled the message and durably persisted the result. In this way, if the consumer crashes after the result is stored but before the current reading position is updated, it will receive and process the message again after restarting and a new result will be generated along with the previously persisted result. In this case, duplication will be introduced between two connection sessions.

Figure 4.16: Duplication of messages with at-least-once delivery semantics.

Exactly-once semantics

Although it is physically not possible to ensure that each message is delivered and processed exactly once in a distributed system given the non-reliable nature of network [24], exactly-once processing semantics can still be realized on top of at-least-once guarantee.

With at-least-once semantics, duplication can be introduced within one connection session or between two consecutive connection sessions. One possible solution is having the processing logic of the application to be idempotent so that multiple retries do not affect the end result. However, the feasibility of this approach depends on specific use cases.

Another approach is implementing deduplication mechanism on either the client or the message broker. The evaluation will focus on inspecting the support of this feature on the ESP platforms.

4.4.2 Evaluation

Apache Kafka

Strong ordering guarantee

A Kafka topic comprises of one or more partitions. Kafka can guarantee that when two records are written to the same partition, they will be stored and delivered to consumers in the same order as their arriving order [50].

On the consumer side, in a Kafka consumer group, each partition of the topic is assigned to only one consumer. There are no other competing consumers for each partition. As a result, the delivery model of Kafka guarantee that messages on a partition are delivered and processed in the right order.

On the producer side, all events from the same source must be sent to the same partition where order is guaranteed by Kafka. By default, records with the same key value will be published to the same partition. Therefore, producers must assign appropriate key value for each published message.

Asynchronous publishing of messages while ensuring order is also possible with Kafka. From Kafka 0.11, the idempotent producer is introduced [51]. When this feature is enabled on a producer, for each targeted partition, each writing request will be assigned an incremental sequence number which will be used for deduplication on the broker in case of retrying. Thanks to this sequence number, Kafka can also tolerate up to 5 asynchronous publishing requests from one producer.

At-most-once and At-least-once delivery semantics

With Kafka producer, users have the flexibility to control the level of durability of published messages. Producer can be configured to wait for acknowledgements from only the leader partition, from all in-sync replicas or not wait for any confirmation from Kafka broker at all [52]. For the at-most-once guarantee, producer can be configured to send a record and immediately proceed to the next one without waiting for acknowledgement from the server. With the at-least-once guarantee, durability of published message must be ensured. Therefore, as elaborated in section 4.2, the producer must be configured to wait for the acknowledgements from all in-sync replicas to ensure that the published message is safely received and replicated by the Kafka server. Internally, if Kafka producer is configured to wait for acknowledgement from brokers, it will retry in case of failure until timeout. Therefore, there is no need to implement the retry logic in the application.

On the Kafka consumer, by default, the offset number indicating the current reading position will be periodically persisted to Kafka. To have a better control over when

to update the reading position, this should be disabled. The offset number should be manually committed before or after persisting the calculated result depending on the chosen delivery semantics.

To sum up, Kafka supports both at-most-once and at-least-once delivery semantics with its clients.

Exactly-once semantics
From release 0.11 [51], Kafka provides mechanisms to protect against both types of duplication introduced by at-least-once semantics (figure 4.16) and realize exactly-once semantics.

To avoid duplicated messages by producer resending, the idempotent producer is introduced. Whenever a new producer instance is created and connected to Kafka, it is assigned a unique ID by the broker. When publishing messages to a partition, producer will add an incremental sequence number to every message along with its ID. By using these two values, the broker can uniquely identify a specific message sent by a producer on each partition and can discard any duplicated messages resulted from resending.

To prevent duplication across two different connection sessions, Kafka provides the transaction feature. When a Kafka producer writes records to multiple partitions in a transaction, either all produced records are safely persisted or none of them is available to be read by other Kafka consumers. When Kafka consumer persists its current reading position to Kafka, internally this value is written to a special Kafka topic. Therefore, the act of updating reading position can be put in a transaction similarly to producing a normal record.

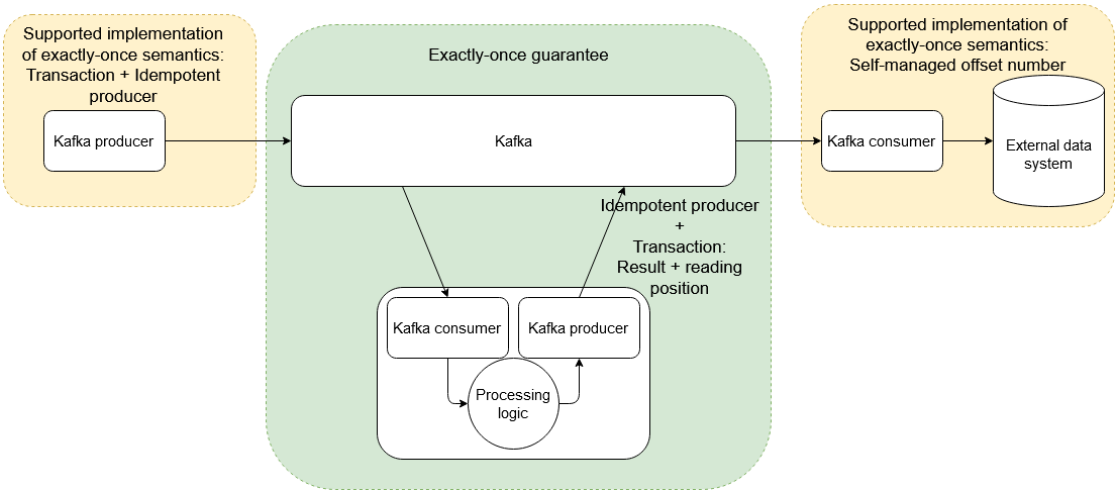


Figure 4.17: Support of exactly-once semantics on Kafka (*green*: fully supported, *yellow*: partly supported, *red*: not supported).

For application which obtains data from Kafka topics with Kafka consumer and then produces new records back to Kafka, it can use Kafka producer to persist the result to output topics and commit the current reading position on input topics in a single transaction. In this way, even if the application crashes while producing records back to Kafka, when it restarts, it can start consuming records again from a clean state with the latest successfully committed result and reading position. Any result generated in the middle of a failed transaction will be discarded.

In case of edge application which obtains data from an external system and produces records to Kafka, the application can produce records along with their related reading positions on the external system in a transaction. Upon restarting, application can retrieve the last committed position on Kafka and resume the processing on the external system accordingly.

However, for edge consuming application which will write result to an external system, the transaction cannot span across its Kafka boundary. In this case, one possible approach is persisting atomically the reading position of the consumer along with the calculated result in the targeted external data system given that it supports transactions. Upon restarted, the consumer can retrieve the committed offset number and resume consumption accordingly. The delivery model of Kafka makes it easier to implement this since the reading position is only an offset number and can be controlled and persisted anywhere by the consumer.

Apache Pulsar

Strong ordering guarantee

A topic on Pulsar can either be non-partitioned or partitioned. Internally, a partition is also a normal non-partitioned topic. Pulsar guarantees that messages published to the same partition/non-partitioned topic will always be delivered to consumption side in the order in which they arrived at Pulsar [43].

If Pulsar consumer is used to consume messages, the Explicit and Failover subscription mode can be used to ensure the order of arriving messages. In the Explicit mode, there are no competing consumers for a topic regardless of being partitioned or not. With the Failover mode, similarly to Kafka consumer group, each partition will be assigned to only one consumer in the subscription while other consumers only take over when failures occur. Thus, each consumer in these two modes can receive all messages on a partition from Pulsar in the right order. If Pulsar reader is used, order of consumed messages is guaranteed since each reader can only be connected to a non-partitioned topic and grouping multiple readers for parallel reading of a topic is not possible.

On producer side, if the topic is partitioned, appropriate key values must be assigned to every message to ensure that events from the same source will be produced to the same partition where order of delivery is guaranteed by Pulsar.

Regarding asynchronous publishing of messages, Pulsar also claims that a producer can have multiple asynchronous requests on the same connection to Pulsar while the order of messages received on the broker is still guaranteed [53].

At-most-once and At-least-once delivery semantics
When using Pulsar producer, it is not possible to publish messages in a fire-and-forget way. Internally, the producer always waits for confirmation from the broker and retries sending the messages if no acknowledgement is received until a specified timeout. As a result, a message can be duplicated on the stream. The Pulsar broker supports deduplication of resent messages. However, this will introduce overheads on both the producer for resending and the brokers for deduplication which does not comply with the key benefit of low overhead of at-most-once semantics. Therefore, in general, Pulsar producer does not support at-most-once. On the other hand, the Pulsar producer is very suitable for the at-least-once guarantee. Since the producer always waits for acknowledgement and retries sending message internally, no self-implementation of retry logic is needed.

On the consuming side, the application must update the reading position (by either acknowledging if Pulsar consumer is used or persisting message ID of the consumed message if Pulsar reader is used) and durably persist the calculated in the appropriate order for the delivery semantics.

Considering both publishing and consuming sides, Pulsar clients only support at-least-once delivery semantics.

Exactly-once semantics
From release 1.20.0, Pulsar provides idempotent producer to discard any duplicated message introduced by producer retrying [54]. With this feature, each producer must be assigned a globally unique name. Each message published to a Pulsar topic will be automatically assigned an incremental sequence ID by the producer. Pulsar broker will use the producer name and the sequence ID to uniquely identify each message published by a producer on a topic and discard any duplicated message resulted from retrying.

For deduplication of duplicated messages created by consumer between two connection sessions, from release 2.7.0, Pulsar provides the transaction feature for Pulsar producer and consumer [55]. With this new feature, users can publish messages to multiple Pulsar topics as well as acknowledge consumed message on numerous Pulsar source topics in one atomic operation. If the application fails during the transaction, all half-committed values will be discarded.

For application which reads messages from Pulsar and produces results back to Pulsar, exactly-once semantics can be achieved. The transaction feature can be used to ensure that the application will always recover from a crash with a clean state on both source and output Pulsar topics.

For application which processes data on external system and produces messages to Pulsar, it can use Pulsar producer to send messages and the corresponding reading positions on the external system to Pulsar in one transaction. In this way, the application can resume the consumption on the external system after failure with the committed position on Pulsar to ensure the exactly-once semantics.

In case of application which reads messages from Pulsar and writes result to an external system, the Pulsar transaction cannot span on both systems. In this case, exactly-once must be self-implemented. For instance, the Pulsar transaction can be integrated into the two-phase commit protocol [56] to acknowledge consumed messages on Pulsar and produce new messages to external system. This approach is used in the implementation of this connector to connect Apache Pulsar and Apache Flink [57]. However, this method depends on the availability of support for two-phase commit on destination system. Users can also use Pulsar reader to have more flexible control on the reading position on Pulsar. In this case, the message ID of the currently consume message on Pulsar can be persist together with the corresponding result to the external system to achieve atomicity similarly to Kafka.

The transaction feature is still in technical review phase. Therefore, this feature should not be used for production and exactly-once semantics is considered to be only partly supported on Apache Pulsar at the moment.

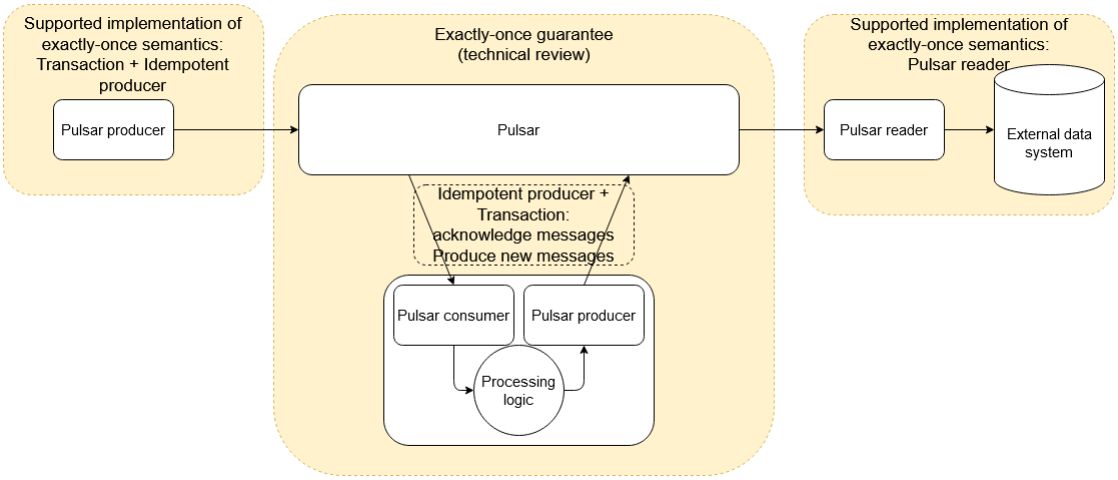


Figure 4.18: Support of exactly-once semantics on Pulsar (*green*: fully supported, *yellow*: partly supported, *red*: not supported).

NATS Streaming

Strong ordering guarantee

Messages on NAT Streaming are organized into different channels. There is no lower level of partitioning of messages. NATS streaming can guarantee that when a subscriber consumes from a channel, messages are delivered in the same order as when they are published and persisted to the channel. This is not explicitly written in the

document, but it is explained by NATS development team in discussions on GitHub [58].

On the subscriber, to guarantee the order of messages, only normal subscription mode can be used. In this mode, each subscription has only one consumer without any other competing consumer [48]. Therefore, this single consumer can receive all messages on the channel in right order and process them sequentially.

Moreover, with its delivery model, NATS streaming actively pushes messages to subscriber and expects that these messages will be processed right away and acknowledged shortly after. Therefore, in case the consumption of a message takes longer time than the timeout for acknowledgement, the message can be resent out-of-order along with new messages. If the processing speed of client does not match the delivery of server, the sequence of delivered messages will not retain the right order anymore.

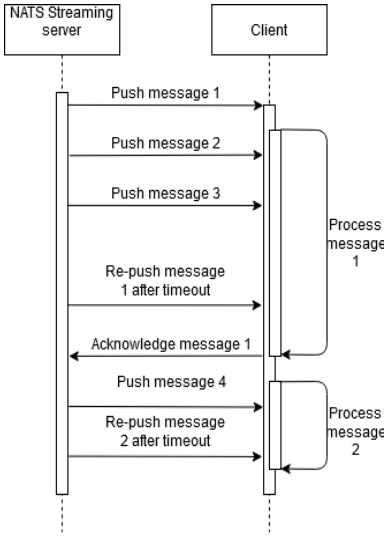


Figure 4.19: NATS Streaming server redelivers messages out-of-order: message 1 and 2 are processed longer than the acknowledgement timeout and therefore are resent by server. Client sees the sequence of messages 1, 2, 3, 1, 4, 2.

Therefore, the sending rate of the server must be limited to only one unacknowledged message at a time necessary to avoid out-of-order redelivery [59]. As a result, it is not possible to receive messages in batch with NATS Streaming.

On the producer side, messages must be published to the same channel where order is retained. Moreover, producer must synchronously send the messages one after another since the NATS server does not support protection against out-of-order caused by resending messages.

At-most-once and At-least-once delivery semantics
Internally NATS producer always wait for acknowledgement from server after publishing a message. However, unlike Kafka and Pulsar, this producer does not retry internally to resend a message if it does not receive acknowledgement from the server. Thus, to have at-most-once guarantee on the publishing side, the publishing status returned by the producer can be simply ignored to realize fire-and-forget publishing. On the other hand, for at-least-once semantics, returned error from the producer should be handled with self-implemented retry logic to guarantee that the deliveries of all messages are successful.

NATS subscriber must acknowledge every consumed messages so that the server can continue to push next unacknowledged messages. However, the acknowledgement by NATS client to server is fire-and-forget [58]. The acknowledgment can be lost without the subscriber knowing about it which will cause the the message to be resent. If a consumer can perform deduplication to process each message exactly once, end-to-end at-most-once guarantee can still be realized. Nevertheless, as will be seen in the next evaluation section, end-to-end exactly-once guarantee cannot be achieved on NATS Streaming. As a result, at-most-once delivery guarantee is also not realizable by NATS Streaming.

On the other hand, at-least-once consumption of messages is possible with NATS clients. Consumer must update its reading position (by acknowledging the message with the server for durable subscription or checkpointing the reading position to external system for non-durable subscription) after it has durably published the calculated result. In case of non-durable subscription, the consumer still needs to send acknowledgement to the server. Nevertheless, when to acknowledge is irrelevant since the consumer only relies on the external checkpoint to resume the consumption.

Exactly-once semantics
Currently, NATS Streaming server does not support detect and discard duplication within one connection session when a client resends a message multiple time. If deduplication is needed, it must be self-implemented on the application layer.

For duplication introduced between connection sessions when client crashes before updating its current reading position, NATS Streaming also provides no deduplication mechanism. For an application which reads messages from NATS and publishes the results to another NATS channel, it is not possible to atomically publish new message and update reading position on source channel. Therefore, it is always possible that a message whose result is already published will be re-consumed and duplicated result will be generated. The same applies to application which publishes messages obtained from an external system to NATS Streaming.

For application which consumes messages from NATS and writes results to an external storage, it can use non-durable subscription and rely on the transactional feature of the sink system to store the sequence ID of consumed message and output value to achieve exactly-once. However, NATS Streaming server can always redeliver a message when it

does not receive acknowledgement from client after timeout. There are also duplicated messages introduced to the channel from the upstream applications. Application must handle the deduplication of these messages as well.

As a result, in general, exactly-once semantics is not supported by NATS Streaming.

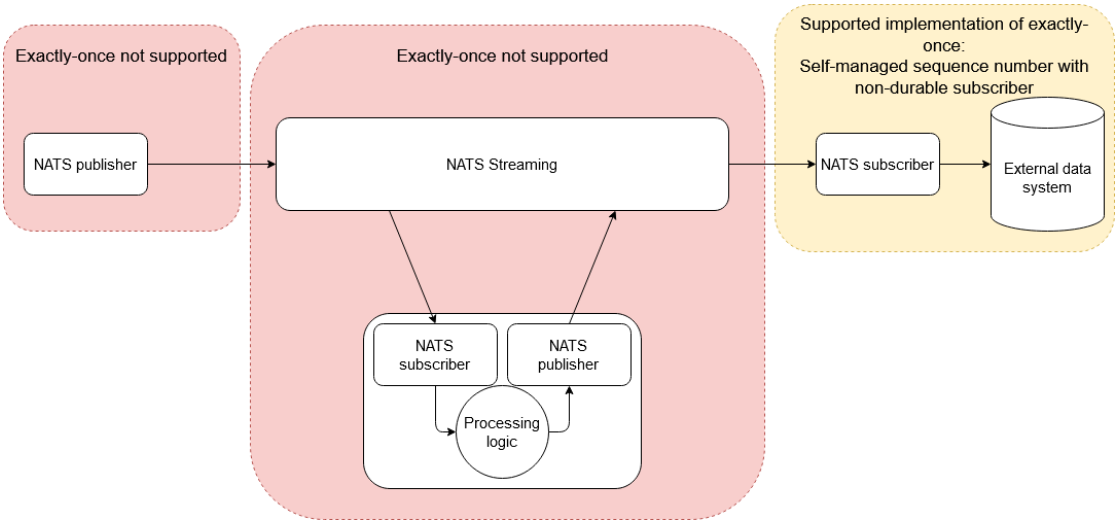


Figure 4.20: Support of exactly-once semantics on NATS Streaming (*green*: fully supported, *yellow*: partly supported, *red*: not supported).

4.5 Stream processing

Apache Kafka

Native stream processing

From release v0.10, Kafka provides the Kafka Streams to support native stream processing [60]. This is a Java library built on top of Kafka producer and consumer. Users can use this library to implement and deploy stream processor which reads input records from Kafka and produces calculated results back to Kafka as a normal Java application.

One of the main advantages of Kafka Streams is that users does not need to set up a separated cluster for stream processing. Normal Java application instance can be simply started anywhere to do the stream processing with Kafka. Moreover, since Kafka Streams is developed from normal Kafka client, it inherits the parallelism and failover concept the Kafka consumer group. Multiple instances of a stream processor can be started for scalability. Each instance is assigned a set of partitions from the input topics. This also guarantees that records on the same partitions are processed by only one instance in the same order as when they are sent to Kafka. When the number

of instances is changed due to failure or scaling up with more instances, the partition will be automatically rebalanced similar to the Kafka consumer group. Moreover, Kafka Streams also benefits from the features of idempotent producer and transaction of Kafka client. Therefore, exactly-once processing guarantee can be achieved with this library.

Kafka Streams library provides the Streams Domain Specific Language (DSL) which supports a rich set of functionalities. The DSL provides logical abstraction of Kafka topics as *KStreams* and *KTables*. A *KStream* consists of all records in a Kafka topic while a *KTable* only presents the latest values of each record key in the topic which is similar to a normal database table. The table representation can be very useful when doing lookup of the current state of an entity.

The DSL supports a rich set of both stateless and stateful operations on top of its abstractions. There are many stateless operations such as filtering, mapping each record to a set of corresponding outputs. For stateful processing, the DSL support aggregating multiple records to extract result, joining *KStreams* and *KTables*, grouping records into window for processing. Users also have the flexibility to choose the semantics for the time boundaries of the windows such as event time (i.e. the time a record is actually generated at the source) or processing time (i.e. when the records are received by the stream processor). Users also have the possibility to choose different types of windows, control how the windows are advanced over time (Figure 4.21).

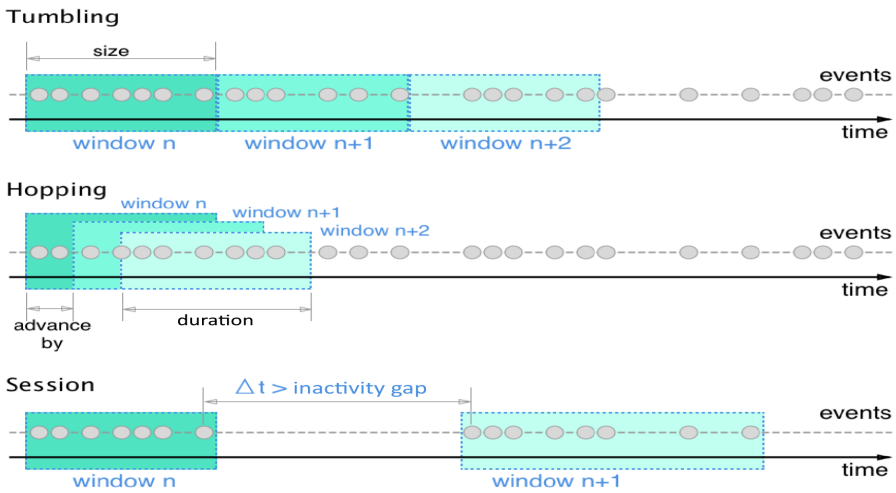


Figure 4.21: Supported window types for stream processing with Kafka Streams (taken from [61]).

Each Kafka Streams application instance retains the current state of the stateful operations in a local state store. This has a number of advantages such as faster state query than remote storage, better isolation between the instances [62]. Moreover, Kafka

Streams also supports checkpointing the local state store to a Kafka topic for fault-tolerance which is managed completely transparent to users.

Apart from DSL, Kafka Streams also provides a lower-level Processor API. With this API, users have more flexibility to implement more sophisticated stateless and stateful operations which are not provided by the DSL.

To sum up, Kafka Streams is a powerful tool to do stream processing on top of Kafka topics. Stream processors which are implemented using Kafka Streams not only benefit from various useful functions of the library, they can also integrate seamlessly with Kafka for scalability and fault-tolerance.

Apache Pulsar

Native stream processing

Apache Pulsar provides the Pulsar Functions for native stream processing [63]. The concept of Pulsar Functions is similar to serverless function as a service (FaaS). Users can start a number of function workers on the Pulsar cluster. These workers can be run within the normal Pulsar brokers or they can be grouped into a separated cluster running on different host machines.

Functions can be implemented using Java, Python or Go and deployed to the function workers. While deploying, the input topics and output topics of the function can be specified. Users can also configure the number of instances of the function to be run in parallel for scalability. Each instance is assigned to a function worker in the cluster where it is executed. Whenever a message arrives from one of the input topics, a function instance is triggered, and result is generated to the output topics.

Internally, Pulsar Function uses Pulsar producer and consumer to read and write messages to Pulsar. By default, the internal Pulsar consumer uses Shared subscription mode to maximize the parallelism. In this case, messages on the input topics are distributed evenly across the instances of the function. Therefore, these messages can be processed out-of-order. To ensure that messages are handled in the right order, user must strictly enable that when deploying the function. If this is enabled, function will choose the Failover subscription mode for its consumer. In this way, each partition of a topic will be assigned to only one function instance and messages will be delivered and processed by the instance in the right order as when they are published to Pulsar.

With Pulsar Functions, users can enforce stateless processing logic on each incoming message such as transforming, filtering, or routing to different output topics based on its content. The function also supports simple stateful stream processing. A function instance can aggregate input messages and persist the current state to Bookkeeper. Unlike local state store of Kafka Streams, the full state of a Pulsar function is maintained centrally in Bookkeeper. This state can be query directly by users using the REST API of the function worker or the command line tool provided by Pulsar.

Pulsar Functions also supports grouping messages into windows and generate corresponding results for each window. Nevertheless, this feature is still not properly documented and is also still unstable with unresolved issues [64]. More sophisticated stateful operation such as joining messages from two topics are not supported by Pulsar Functions. Moreover, Pulsar Functions currently does not support exactly-once processing guarantee since the transaction feature is still in technical review phase and not integrated into the function.

In summary, Pulsar Functions provides a simple and convenient way to quickly deploy functions to apply simple processing logic on the stream of messages on Pulsar. Because the functions run within the Pulsar cluster, no additional administrative tasks are required. Nevertheless, the available functionalities of Pulsar Functions are quite limited and cannot support more sophisticated streaming operations.

NATS Streaming

Native stream processing

NATS Streaming does not provide any tool for native stream processing. Users must implement the stream processing application from the ground up with NATS Streaming client or rely on external streaming processing framework such as Apache Spark or Apache Flink.

4.6 Data integration and Interoperability

Apache Kafka

Connectors to external system

Kafka provides the Kafka Connect framework to import data from external data systems to Kafka and export records from Kafka topics to sink storage systems [65]. By using this framework, connectors for data integration can be quickly developed, deployed, and run in a scalable and fault tolerant way.

Kafka Connect provides a uniform and high-level Java programming interface to develop connectors. Once the executable connector is built, it can be deployed to a Kafka Connect cluster. The Kafka Connect framework provides a REST API to deploy and manage connectors on the cluster. A Kafka Connect cluster runs independently from Kafka cluster and can comprise of multiple worker nodes. The tasks of copying data defined in the implementation of the connector can be distributed among these nodes for scalability.

Internally, the workers in the connect cluster use Kafka clients to publish or consume data from Kafka. These worker coordinate with each other using the same balancing and failover mechanism as the Kafka consumer group. The framework automatically and periodically checkpoints the current processing positions on the source system of the copying task in Kafka. Developers do not have to take care of managing the consumption status of connector.

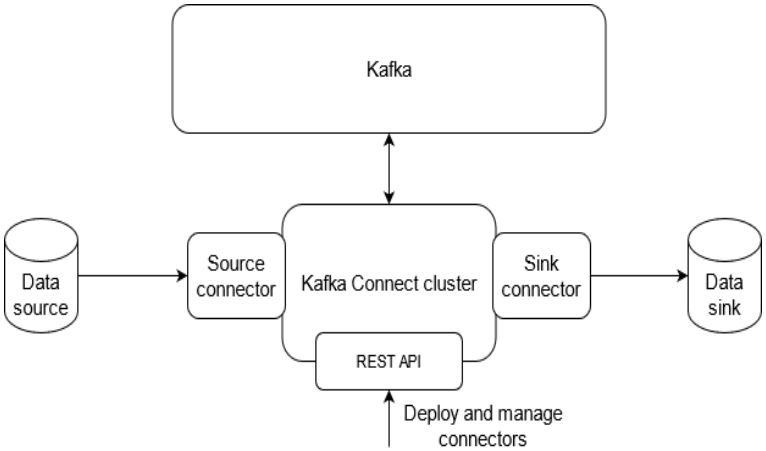


Figure 4.22: Kafka Connect cluster to integrate data between Kafka and external data systems.

For the source connector which imports data into Kafka, the auto-management of consumption status cannot provide exactly-once guarantee in case of failure. This is because the framework does not use transactional feature of Kafka producer to save processing position along with the imported records [66]. Therefore, if a worker node crashes before it checkpoints the consumption status on source system to Kafka, it is possible that some records will be reprocessed again when the copying task is resumed.

With the sink connector which exports data from Kafka, users have the possibility to flush the offset number of the currently consumed record on the Kafka topic along with the actual data in a single atomic action to external data systems. Therefore, when worker node fails, the task can be resumed with the committed offset number on the external system instead of using the checkpointed position on Kafka. As a result, exactly-once semantics can be guaranteed on the sink system. For example, the sink connector to export data from Kafka to HDFS uses this approach to guarantee exactly-once semantics [67].

With Kafka connector, users can also define a chain of simple and pluggable transformation operations to modify the records one after the other before they are written to destination systems [65]. This can serve as a quick preliminary adaption of records so that they can match with the processing logic in their destination.

There are already many off-the-shelf connectors available for various data systems. The Confluent Hub, which is managed by the Confluent company, is a central repository to share Kafka connectors with both open-sourced and commercial licenses. There are connectors of many common data systems such as relational database, Hadoop distributed file system (HDFS), Cassandra, change data capture (CDC) on the source system. All these connectors can be quickly configured and deployed to a Kafka Connect cluster

without any implementation requires. This can further offload the development burden on users to integrate Kafka with other data systems.

Supported programming languages for clients

The official release of Kafka includes only the Java client [68]. Kafka relies on different smaller groups of developers for providing clients in different programming languages. There are many third-party projects with more than 15 different supported programming languages such as C/C++, Python, Go. Most of these projects are very active and regularly updated along with the new release from Kafka.

Apache Pulsar

Connectors to external system

Pulsar supports the automation of moving data in and out of Pulsar with the Pulsar IO connectors [69]. The concept of Pulsar IO is similar to Pulsar Functions. Developers can implement connectors using a Java programming interface provided by Pulsar.

The executable files of connectors can then be deployed to the cluster of function workers on the Pulsar cluster using the REST API of the cluster or the admin command line tool of Pulsar. The connector is then run and scaled among the nodes of the function workers. Internally, Pulsar IO also uses normal Pulsar producer and consumer to interact with the Pulsar cluster. However, unlike the Kafka Connect framework, the management of reading position on the source system is not done transparently to users by Pulsar IO. Developers must handle the task of checkpointing the current consumption status of a connector and retrieving it when the connector is restarted. Therefore, the delivery semantics of a connector depends on the developers and which mechanism they use to commit the reading position. With the newly released Pulsar transaction, developers can utilize this feature to achieve exactly-once semantics for the connector. For instance, transaction is used in the implementation of sink connector to export data from Pulsar to Fink [57].

In the official release of Pulsar, there are many ready-to-be-used source- and sink-connectors for different data systems such as relational databases, Kafka, HDFS, NoSQL databases. Users can simply start these connectors on the Pulsar cluster without having to download and deploy these connectors manually. There are also many other connectors developed and maintained by third-party organizations. For instance, Streamnative, which is a company offering managed Pulsar as a service, has a central hub with many Pulsar connectors.

Supported programming languages for clients

Officially Apache Pulsar supports 7 different clients in different programming languages including some most popular languages such as Java, Python, and Node.js [70]. Moreover, there are also other clients in 4 different languages all of which are actively maintained by third-party contributors.

NATS Streaming

Connectors to external system

Currently there is not any general framework to move data in and out of NATS Streaming servers. There are only a few other projects which bridge NATS Streaming with some specific data systems such as Kafka and IBM-MQ [71]. Moreover, these off-the-shelf connectors have to be deployed, managed and scaled manually by users. Other than that, if users want to connect NATS Streaming with external data systems, they need to handle the task of implementing, deploying and operating the connectors themselves. Therefore, integrating data with other systems is generally not supported by NATS Streaming.

Supported programming languages for clients

Syndia, which is the company actively develops and maintains the NATS Streaming project, officially supports clients in 7 different programming languages with some common languages: Java, C, Node.js [71]. In addition, there are some other clients maintained by the community. Nevertheless, these projects are very inactive and outdated. Therefore, these third-party clients will not be included in the evaluation.

4.7 Monitoring and Management

Apache Kafka

Technical monitoring

Kafka supports monitoring on both the brokers and clients [72]. Kafka brokers collect numerous metrics about their current statuses and expose this information to users using Java Management Extension (JMX) [73]. There are many useful metrics such as request rate, memory usage, connection status. Kafka clients and tools such as producer, consumer, Kafka stream processor and connector support monitoring by reporting numerous built-in metrics with JMX such as request rate, response rate from the server, network rate.

The metrics exposed by JMX can be displayed and monitored using JMX-compliant tools such as jconsole, Prometheus with the JMX exporter. With this monitoring mechanism, Kafka gives users the flexibility to plug in the metrics into different monitoring systems without being tied to a specific tool.

Apart from relying on the built-in monitoring mechanism of Kafka, there are also many monitoring tools for Kafka health-check developed by third-party organizations which require only minimal setup and can be quickly started [74]. Each tool has a different license and provides a different set of functionalities. Therefore, users have a wide selection of available tools to match their needs in different cases.

Apache Pulsar

Technical monitoring

Components in a Pulsar cluster expose their monitoring metrics via HTTP ports in

Prometheus format [75]. The Pulsar brokers provide numerous statistics about their current health and statuses such as usage of CPU, memory and network bandwidth, number of currently connected producers and consumers, total throughput. The Zookeeper and Bookkeeper shipped together with the Pulsar release also report many metrics on the opened HTTP ports. These metrics can be directly collected by Prometheus to monitor and create alerts based on the current health of the Pulsar cluster.

In addition, Pulsar also provides an off-the-shelf and open-source tool named Pulsar-Manager to manage and monitor Pulsar clusters [76]. It is a tool with web UI which can be quickly deployed and connected to a running Pulsar cluster to provide insight into the current status of the cluster. Moreover, users can also dynamically manage and configure the cluster via the UI of the tool such as updating configuration of Pulsar brokers, creating new topics, resetting reading position of consumers.

NATS Streaming

Technical monitoring

To support monitoring, a NATS Streaming server exposes statistics about its current status via an opened HTTP port [77]. The metrics are returned to users in form of JSON. NATS Streaming also provides a Prometheus exporter to convert the metrics from the server monitoring port into Prometheus format to help users display and monitor the server more conveniently.

The metrics provided by the streaming server via the monitoring endpoints cover mostly information on the high level such as number of currently connected clients, number of channels, total messages received and persisted by the server, current consumption status of subscribers. Users can also keep track of the more detailed information such as memory and CPU usage, message throughput of the underlying NATS server of the streaming server via the same monitoring port [78].

Although the supported metrics of NATS Streaming server are not as many as Apache Kafka or Apache Pulsar, they can still provide a generally good insight into the current status of the streaming server given the simplicity of NATS Streaming compared to the other two platforms.

4.8 Scalability

Apache Kafka

Scalability of storage and computing server

A Kafka broker serves read/write requests from clients as well as persisting the records on its disk. Therefore, the storage and processing layers are closely related and cannot be scaled independently.

Topic partition is the basis element to achieve scalability on Kafka. Each partition can have one or more replicas, namely, a leader and a number followers for fault-tolerance.

Each replica of a partition resides completely on the disk of a broker in the cluster. Read and write requests from client can only be done on the leader of a partition. Therefore, by distributing the leaders of partitions evenly among the brokers, the load on the serving layer can be balanced. Moreover, for each partition, to have even distribution, a different subset of brokers is selected to keep its replicas on their disks. A new broker can always be added to the cluster to scale up the capacity of both processing and persisting. When new partitions are created, their replicas will be automatically assigned to the new broker for load balancing.

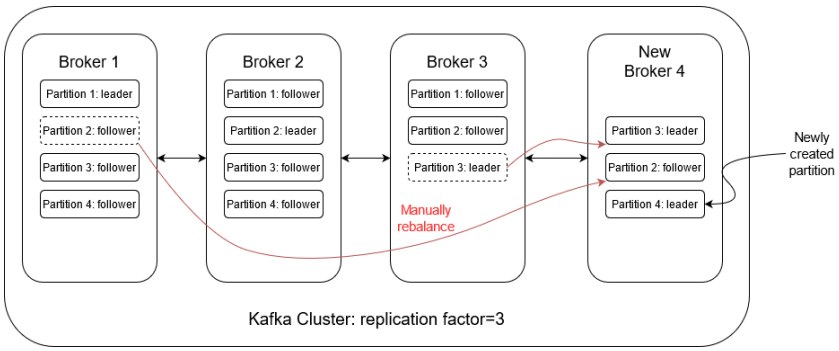


Figure 4.23: Kafka clustered is scaled up with a new broker.

However, for partitions which were created before the new broker is started, their loads are not automatically balanced. In figure 4.23, replicas of partitions 1, 2 and 3 are not automatically offloaded to the new broker. Users must manually conduct the rebalancing using the tools provided by Kafka.

Moreover, the tied coupling between the processing and persisting layers also limits the scalability of Kafka. It could be difficult to optimize the scaling to meets the different requirements on each layer. Therefore, scalability is supported by Kafka. However, the scaling mechanism is quite rigid and requires some manual works from users to rebalance the load.

Broker failover mechanism

In a Kafka cluster, each broker holds the leadership of a subset of partitions and serve requests to these partitions. In case a broker fails, the leadership of each partition will be transferred automatically to one of the brokers retaining the in-sync replica of that partition [31]. This helps to ensure that the new leader of the partition will have all committed records and guarantee the data consistency. When the failed broker comes back online, by default, users must manually rebalance the leadership again.

To manage the leadership of partitions among the available brokers, a broker in the cluster is elected as the controller [79]. At any time, only one broker can have the role

of controller. Currently, Kafka relies on the consistency guarantee of Zookeeper to avoid inconsistent state with more than one active controllers.

When a broker goes down, the partitions with replicas persisted on the failed broker will become temporarily under-replicated because no auto data re-replication mechanism is provided by Kafka. This may lead to unavailability even when there are still many online brokers as the minimum number in-sync replicas cannot be met.

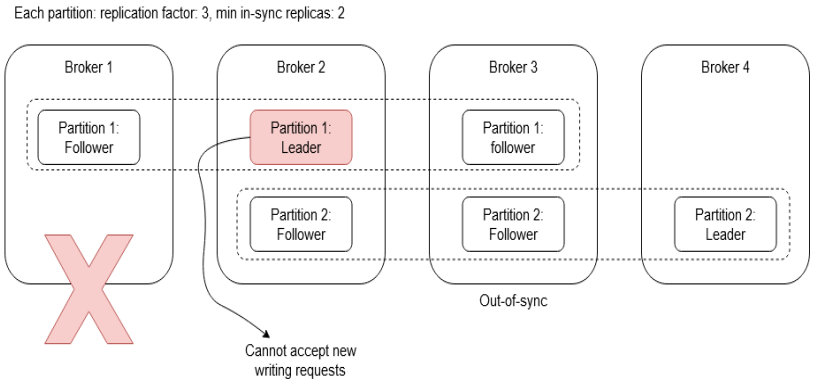


Figure 4.24: Kafka clustered is scaled up with a new broker.

In figure 4.24, when broker 1 fails and the replica of partition 1 on broker 3 becomes out-of-sync with the leader, broker 2 cannot accept new writing requests for partition 1 since it cannot meet the minimum in-sync replicas of two. Broker 4 cannot support to ensure the availability since it does not retain the copy of partition 1. In this case, the availability is compromised for the reliability of data persistence.

If all in-sync replicas including the leader fail at the same time, the remaining replicas of the partition are all lagged behind without having the latest committed messages. In this case, users can choose between availability and consistency of data [31]. Users can either allow one out-of-sync replica to become the new leader to not disrupt the requests handling and risk losing some records or wait until one in-sync replica goes back online and reject all requests to the failed partition in the meantime.

Since a Kafka broker also serves the persistence of records, there is a trade-off between its availability and consistency of data. The failover mechanism of Kafka give users the flexibility to manage this trade-off.

Apache Pulsar

Scalability of storage and computing server

In a Pulsar cluster, the cluster of Pulsar brokers is in charge of handling requests from clients while the Bookkeeper cluster is used for data storage. There two layers of processing and storage can be scaled independently.

On the requests-serving layer, each topic is owned by only one broker in the cluster and all requests to this topic will be directed to this broker only. When a new topic is

created, the load manager in the Pulsar cluster assigns it to the broker with the least load. Moreover, whenever a broker is overloaded or goes down, the load manager also automatically triggers the balancing to distribute the topics to other brokers [80]. As a result, scaling the cluster of brokers is very straightforward.

On the storage layer, scalability is achieved by the built-in mechanism of Bookkeeper. As elaborated in section 4.2, messages are replicated on Bookkeeper cluster in fragments. Each fragment has a different ensemble which is determined by the Pulsar broker. When a new Bookkeeper nodes is started, they will automatically be considered when choosing the ensemble for the next fragment and can start to share the load with other existing nodes.

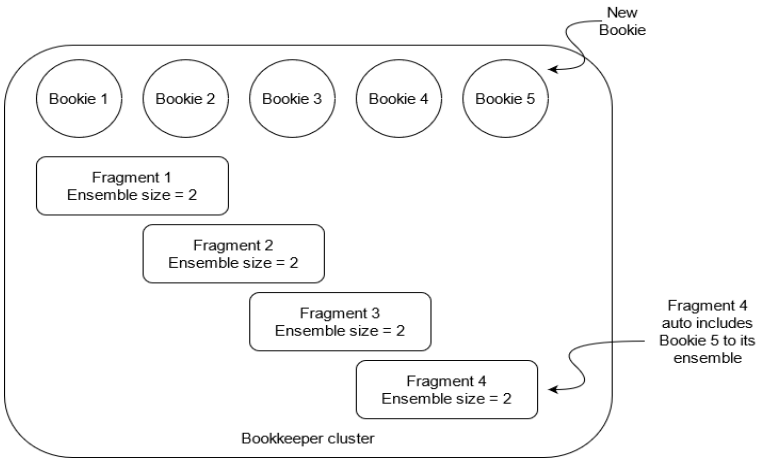


Figure 4.25: Storage layer of Pulsar can be scaled up seamlessly by starting new Bookkeeper instances.

If a Bookkeeper node is shut down, some fragments with messages persisted on this node will be under-replicated. However, the auto recovery feature of Bookkeeper can detect this and re-replicate these messages to other running nodes to maintain the replication quorum [34].

Broker failover mechanism

The brokers are registered to the Zookeeper and regularly kept track of its online status. When a broker is detected to be failed by Zookeeper, its topic will be assigned to other online brokers in the cluster. Apache Pulsar uses the fencing mechanism of BookKeeper to prevent the disconnected broker from writing new message to the topic [33]. This ensures that at any time, only one broker can serve writing requests to a topic.

Because the processing and persisting layers of Pulsar are completely separated, the failover mechanism does not have to consider the reliability of data persistence as in Kafka. As long as there is an online broker, new requests can continue to be accepted.

If the requests are handled successfully, clients will receive confirmations from broker and be ensured about the reliability of data.

NATS Streaming

Scalability of storage and computing server

In both operational modes of NATS Streaming cluster, namely clustering and fault-tolerance, only one server in the cluster can handle read and write requests from clients [18]. To support horizontal scaling of requests-serving layer, NATS Streaming provides the partitioning feature [81]. This feature is not compatible with clustering mode. With this feature, the normal fault tolerant cluster can be partitioned into a number of smaller sub-clusters, each of which is independently in charge of handling requests for a subset of channels and has a separated datastore to persist messages of these channels.

Each sub-cluster must be strictly configured with a fixed set of channels. Requests to different channels will be balanced among the server instances in the cluster. New sub-cluster can always be added to the cluster to handle more channels.

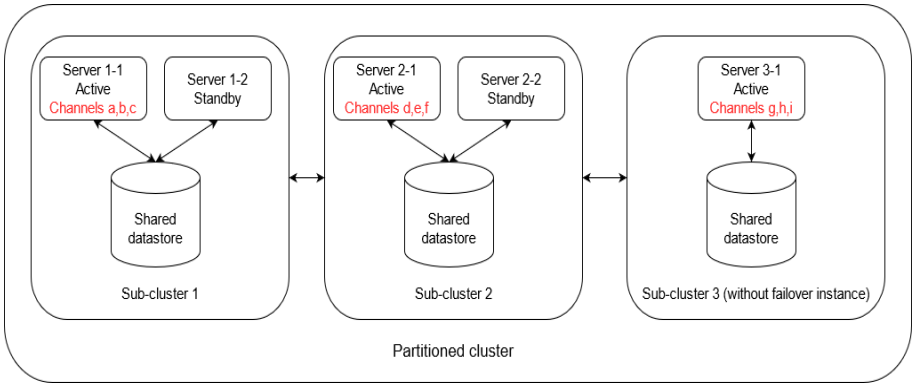


Figure 4.26: Request-serving layer of NATS Streaming can be scaled with the partitioning feature.

Nevertheless, this scaling mechanism has many limitations. Firstly, this feature cannot be used together with clustering mode. Therefore, it cannot use the data replication feature provided by NATS Streaming. The set of assigned channels for each sub-cluster cannot be changed in runtime. If new channels are required, the sub-cluster must be stopped for reconfiguration. Moreover, because each sub-cluster has a completely separated storage location for all assigned channel, it is not possible to dynamically transfer the ownership of some channels on one sub-cluster to another for load balancing.

On the storage layer, the scalability depends on the chosen pluggable data store. For instance, if *file* store is chosen as the persistence layer, a scalable network filesystem can always be used. When *SQL* store is used, the scalability is governed by the relational

database technology. However, this relies entirely on the setup of users without any supported feature from NATS Streaming.

In short, scalability on NATS Streaming is possible for both storage and requests-serving layers. However, most of the configurations and administration tasks are not covered by NATS but instead must be handled manually by users.

Broker failover mechanism
In the clustering mode, the servers in the cluster coordinate with each other using RAFT consensus algorithm [38]. Only the one elected leader in the cluster can serve requests from clients. In case the leader fails, one of the followers will be elected as the new leader. This mode is resilient to split-brain since the Raft algorithm uses majority vote for both leader election and acknowledgement of writing requests. However, if more than half of the servers in the cluster fails, there will be not enough nodes for leader election in case of failure. In this case, the cluster cannot accept new request even when there are still online servers. The availability is automatically compromised for data consistency.

In the fault-tolerance mode, several server instances are attached to a share datastore [18]. At any time, only one active node holds the exclusive lock on the datastore to serve requests. If the active node fails, all standby instances will try to obtain the lock on the data storage and the first to be successful will become the new active node. This failover mechanism requires less overhead as in the clustering mode since no leader re-election is required. In this mode, as long as there is an online server, new requests can be accepted.

4.9 Event storage

Apache Kafka

Authentication mechanisms
In a Kafka cluster, there are Kafka brokers and Zookeeper nodes. Only legitimate communication peers, namely, reliable Kafka clients, tools which also use Kafka clients internally such as Kafka Connect cluster for data integration and other Kafka brokers in the cluster can initiate connections to a Kafka broker. On the other hand, Zookeeper should be accessed only by Kafka brokers to maintain their metadata. Moreover, Kafka tools also expose REST endpoints for configuration and management such as deploying connectors in case of Kafka Connect. Kafka provides authentication mechanism to control access to all these endpoints [82].

In order to control access to a Kafka broker, Kafka supports two authentication mechanisms. The first is Transport Layer Security/Secure Sockets Layer (TLS/SSL) protocol [83]. Users can configure each communication peer with a valid certificate signed by a

mutual trusted Certification Authority (CA) and this will be used for 2-way authentication when establishing the connections between Kafka brokers or between a broker and a client.

The second mechanism is Simple Authentication and Security Layer (SASL) [84]. Kafka supports 4 different SASL mechanisms out-of-the-box, namely, GSSAPI (Kerberos), PLAIN, SCRAM, and OAUTHBEARER. Users have the flexibility to choose the mechanism which fits to the security system in their existing infrastructures. A broker can also be configured to use multiple authentication mechanisms to support different groups of clients.

With the Zookeeper soon to be removed from Kafka [26], the connection to Zookeeper will also become irrelevant. Nevertheless, in the current release 2.6.0, the connection to Zookeeper must still be protected by either TLS/SSL or SASL. For authentication with SASL, Zookeeper supports two mechanisms: Kerberos and DIGEST-MD5 [85]. However, the DIGEST-MD5 mechanism is already obsolete and should not be used in production.

Regarding the REST endpoints of the Kafka tools, Kafka supports authentication of connected users with TLS/SSL certificates.

Authorization mechanisms
Kafka allows users to control different level of privilege and a different set of allowed operations for each connected client by providing an Access Control List (ACL) authorizer [82]. This authorizer utilizes the identity of communication peers obtained from the authentication process to determine their access rights. The authorizer supports fine-grained access control for each client with each type of operation on each different resource. Users can define the access right for clients with the general format: *Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP*. In the current release, the access rights list is persisted by Kafka on Zookeeper.

However, with the ACL authorizer, users must specify access rights for each and every client individually. Higher level of control, namely, Role-Based Access Control (RBAC) which allows to group multiple clients into one role adhered to only one access rule, is not possible. If this is needed, users must plug in a self-implemented authorizer which can bind the identities to different roles and define different access rules.

On the other hand, for the connections from Kafka brokers to Zookeeper, Kafka automatically applies the access rules which allow only authenticated brokers to modify metadata stored on Zookeeper. This helps prevent unauthorized modification of metadata which may lead to cluster disruption.

Encryption
Kafka only supports encryption when transferring data using TLS/SSL [82]. More specifically, the connections between Kafka brokers and clients, between two brokers, from bro-

ker to Zookeeper, and connections to REST endpoints of Kafka tools can be encrypted using the Public-Key Cryptography. Other than that, Kafka supports neither encryption of data at rest nor end-to-end encryption between two end clients.

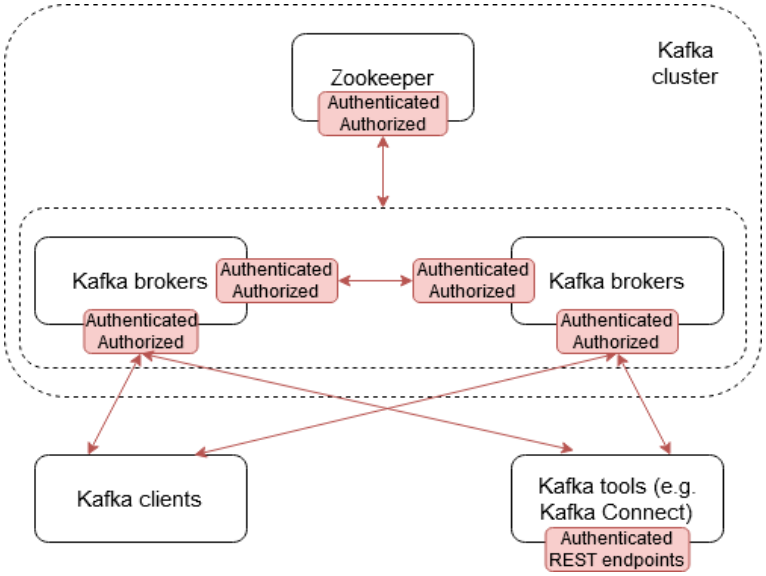


Figure 4.27: Secured Kafka cluster. Red links are encrypted.

Apache Pulsar

Authentication mechanisms

A Pulsar cluster comprises of multiple components. There are the Pulsar brokers, the Bookkeeper cluster and the Zookeeper nodes. In addition, when Pulsar Function or Pulsar IO is used and configured to be run separately from the brokers, there are also nodes of function workers in the cluster. All these components are interconnected. Pulsar provides different authentication mechanisms to protect these components against unauthorized accesses [86].

For the Pulsar brokers and function worker nodes, Pulsar supports different authentication mechanisms. Clients connecting to a broker can be verified using mutual authentication of TLS/SSL protocol [83]. SASL authentication [84] can also be used. Pulsar currently supports two SASL mechanisms which are OAUTHBEARER and GSSAPI (Kerberos). Moreover, with the OAUTHBEARER mechanism, Pulsar also provides a built-in tool to generate and verify token for client identification without having to rely on an additional authorization server. Authentication on Pulsar can also be done with Athenz, which is an open-source platform developed by Yahoo for authentication and authorization based on certificates

A Pulsar broker can be configured with multiple authentications mechanisms. Any client which wants to connect to this broker such as Pulsar producer, consumer, admin client,

another broker or function worker must provide its identity with one of the supported authentication schemes of the broker.

For the connections from Pulsar brokers to Zookeeper and Bookkeeper nodes, they can be protected with either SASL Kerberos authentication mechanism or TLS/SSL certificate-based authentication. Finally, for the connection from Bookkeeper to Zookeeper, users can also configure the authentication using TLS/SSL to control access.

Authorization mechanisms
To control access of authenticated clients to resources on Pulsar brokers and function workers, Pulsar provides a built-in authorizer to define ACL [86].

The resources provided by Pulsar brokers are hierarchical. At the lowest level is topic from which clients can produce and consume messages. The next level is namespace which is a group of related topics. Then there is tenant resource which is a administrative unit consisting of multiple namespaces. Finally, at the highest level is the cluster. User can grant access to different type of resources on different levels for each authenticated client.

Pulsar does not support RBAC. Currently, there is not any mechanism to define and bind roles to clients and control access based on that.

For resources on Zookeeper and Bookkeeper, it is not documented. Therefore, it can be assumed that Pulsar manages the authorization internally and does not expose the control to users.

Encryption
All connections in a Pulsar cluster can be encrypted with the TLS/SSL protocol. Moreover, Pulsar clients also supports end-to-end encryption [86]. This is achieved by combining symmetrical encryption and public-key encryption methods. A Pulsar producer can dynamically generate a symmetric key to encrypt payloads of messages. This symmetric key is then encrypted using a different public key in a public-private key pair and included in the header of the messages. On the consumer side, the consumer can use the corresponding private key to decrypt and retrieve the symmetric key used for payload encryption and then use that to read the messages. However, the distribution of the public-private key pair to the clients is not covered by Pulsar. Users must handle this task manually.

NATS Streaming
Authentication mechanisms
A NATS Streaming server comprises of a NATS Server and a separated streaming module to provide durability of messages on top of the volatile NATS messaging system.

NATS Streaming relies on the authentication mechanisms of normal NATS Server to control incoming accesses from clients [78]. In addition, since the streaming module is also a NATS client internally, it must also use the provided authentication mechanisms

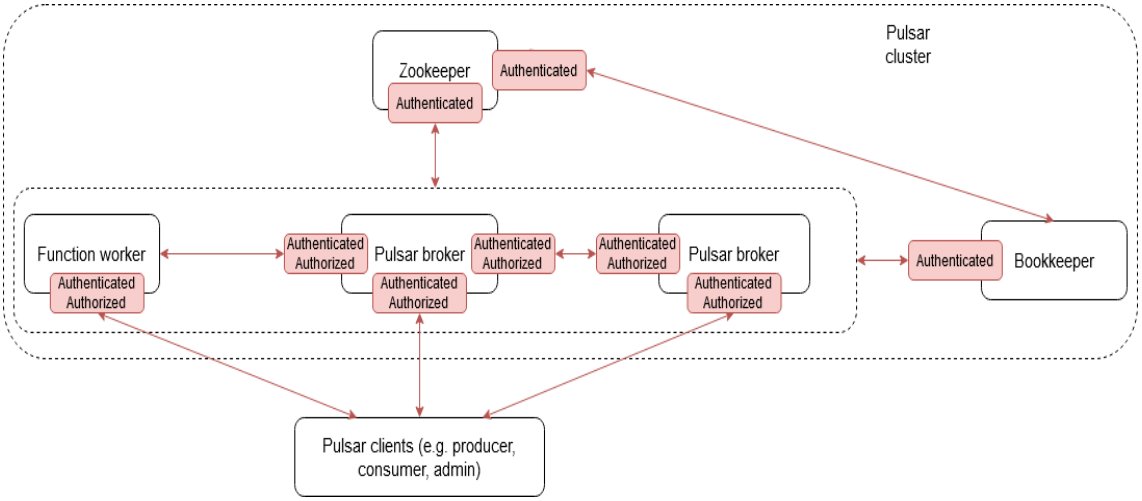


Figure 4.28: Secured Pulsar cluster. Red links are encrypted.

to connect to the NATS Server and serve requests from clients. On the other hand, since the durable storage is pluggable, it is beyond the scope of NATS Streaming to provide authentication for this component. Users must utilize the mechanism provided by the selected datastore.

Clients connected to the NATS server can be identified with mutual authentication using TLS/SSL protocol. NATS also provides a number of built-in authentication mechanisms. Users can configure the server and clients with a shared secret which will be used for authentication when a connection is established. NATS also provides a NATS-exclusive authentication mechanism which is public-key based and called NKey. Upon receiving connection request from a client, the server randomly generates a random value and challenges the client to encrypt that with its private key. The server can then verify the response from client by using the public key of the clients. With this authentication mechanism, NATS provides its own tool to generate public-key pairs and verify their legitimacy during authentication.

Nevertheless, apart from the authentication with TLS/SSL, the other authentication mechanisms are only NATS-specific. This lack of support for pluggable and common authentication mechanisms limits the adaptability of NATS Streaming to infrastructures with existing authentication services. In these cases, the security of the NATS system must be configured separately which can cause extra works on administration and maintenance.

Authorization mechanisms
The underlying NATS server of the streaming server also provides authorization mechanism to control access of individual clients on the resources [78]. Nevertheless, this mechanism is not compatible with the NATS Streaming [87].

More particularly, all subscription requests from clients to streaming channels are internally directed to the streaming module via a single non-persistent NATS channel designated for streaming subscriptions. The authorization mechanisms of NATS can only either block all requests of a client to this internal channel or allow any request to go through. As a result, it is not possible to allow a streaming client to subscribe to only a selective set of channels.

Therefore, authorization is currently not supported on NATS Streaming.

Encryption
Connections from clients to the NATS streaming server and from streaming module to the NATS server can be encrypted with the TLS/SSL protocol [78]. The connection from streaming module to the datastore is not covered by NATS. If this connection needs to be encrypted, users must choose the appropriate encryption scheme which is supported by the selected datastore.

Moreover, NATS Streaming also supports encryption of data at rest. The server can be configured with an encryption key and that will be used by the streaming module to encrypt the payload of all messages before persisting them in the durable storage [88]. However, more fine-grained encryption such as different encryption keys for messages from different clients is not possible on NATS Streaming.

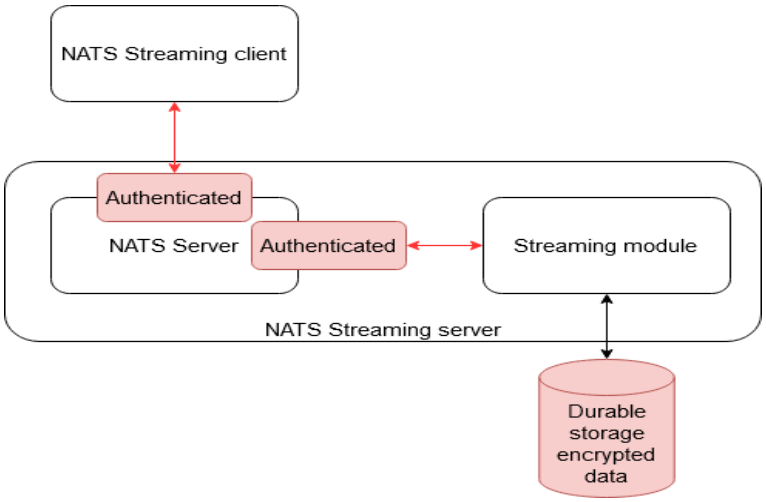


Figure 4.29: Secured NATS Streaming cluster. Red links are encrypted.

4.10 Usability and Community

Apache Kafka
Active community

The community of Kafka is very active. Since it is created in 2012, the GitHub repository of Kafka has been regularly committed. Starting from the end of 2015, the activity of the community increases significantly with an average of roughly 20 commits per day. In the top 30 contributors, each of them has made more than 50 commits to the project. Many of the contributors are from Confluent which is the company founded by the creators of Kafka. Nevertheless, there are also numerous contributors coming from different organizations and institutions as well.

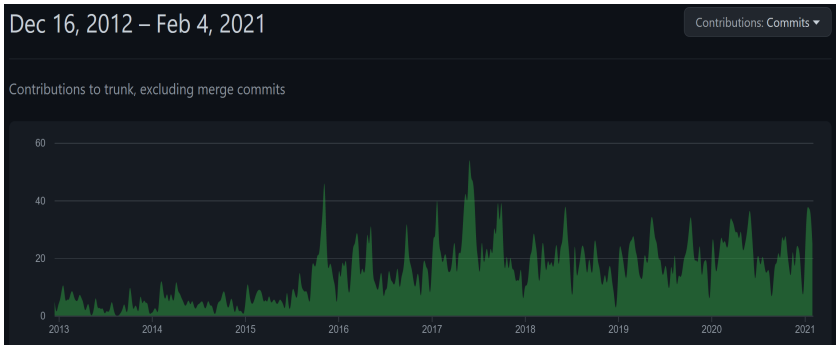


Figure 4.30: Contributions to Kafka on GitHub [89].

In December 2020, more than 50 contributors made 160 commits to the repository. In this month, there are also around 50 new pull requests and more than 100 merged pull requests on the project.

Apache Pulsar

Active community

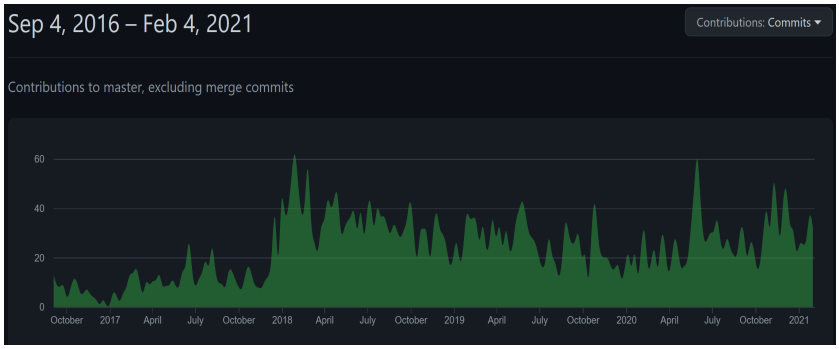


Figure 4.31: Contributions to Pulsar on GitHub [90].

Since 2016, the Apache Pulsar has been steadily contributed by the community. From 2018, Apache Pulsar has gained more interest from the community with the increase in commits to about 30 commits per day. Each of the top 20 contributors have made more than 50 commits to the project. Many top contributors come from Splunk and

Streamnative which are two companies providing managed services based on Apache Pulsar. In addition, there are also many contributors from the different organization actively engaging in the project.

In December 2020, 360 commits have been made by 56 contributors on the Pulsar project. Moreover, more than 200 pull requests are active during this time.

NATS Streaming

Active community

From the data on GitHub, the NATS Streaming project is not very active, especially since 2018 with an average of around 3 to 4 commits per day. Moreover, only the top 3 contributors have made most of the commits to the project and two of them are from Synadia which is the company creating the NATS system. The other contributors only make minors commit to the project.

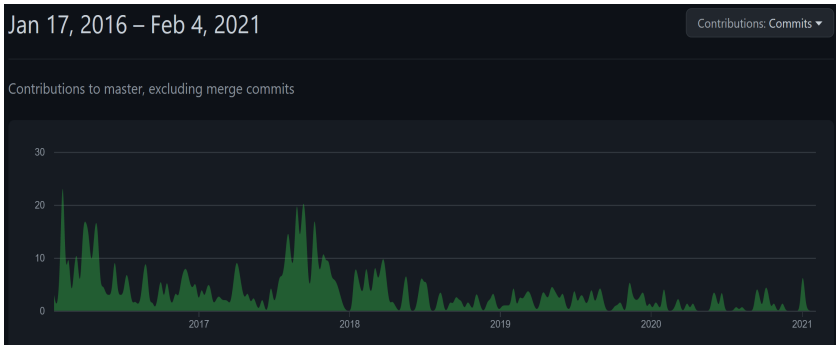


Figure 4.32: Contributions to NATS Streaming on GitHub [91].

In the month December 2020, no commit has been made on the NATS Streaming repository. In addition, during this month, no pull request is created or merged.

4.11 Performance

There are many comparisons on the time behaviors of most common message delivery systems including Apache Kafka, Apache Pulsar and NATS Streaming. However, the benchmarking results from the comparison of the company SoftwareMill [3] are used in this thesis for the evaluation based on a number of reasons. Moreover, this test takes into consideration all three platforms and provides a uniform benchmark setup for comparison. Finally, since this comparison is done by an independent company, the result can be more objective compared to the benchmarking done by the company Confluent which endorses Kafka [2] or the comparison by the StreamNative company which advocates Pulsar [4].

The setup of the test includes a cluster of servers and a number of client instances (producers and consumers) all running on AWS EC2 instances. In the benchmarking,

each message is required to have three replicas on different server nodes. Therefore, the cluster of each ESP platform is set up accordingly.

Apache Kafka: Three Kafka brokers running on three different EC2 instances. In addition, there is also a Zookeeper instance running on the same EC2 node with each broker. The replication factor is configured to 3 and the minimum in-sync replica is set to 2.

Apache Pulsar: There are three Bookies running on three different EC2 instances. Two Pulsar brokers are started on two different EC2 instances. In addition, there are three separated Zookeeper nodes. The write quorum and acknowledge quorum are set to 3 and 2 respectively.

NATS Streaming: Three NATS Streaming server instances are started on three different EC2 instances. The servers are configured to use file store to persist messages on their mounted disks and to run in clustering mode for data replication.

In the benchmarking scenario, producers and consumers use one topic or channel to send and receive messages. The producers are configured to synchronously wait until the server acknowledge that the published messages are safely persisted and replicated. With Kafka, the producer has to wait for acknowledgment from all in-sync replicas. For Pulsar, On the consumers side, at-least-once delivery guarantee is chosen. The consumers can asynchronously acknowledge the consumption of messages to the server.

The throughput of the messages on both publishing and consuming sides are measured in messages per second with all messages having the same size. The end-to-end latencies of each message within a 1-minute window are also measured and the 95th percentile value is recorded in the result.

Moreover, in the tests of Apache Pulsar and Apache Kafka, the topic is partitioned to balance the load among the brokers and maximize the utilization of the available processing capability. In the test, the Pulsar topic is only partitioned into 4 because adding more partitions to the topic does not enhance the performance.

Table 4.33: Benchmarking results of Kafka, Pulsar and NATS Streaming [3].

	Partitions	Threads /node	Sender nodes	Receiver nodes	Producing throughput (msgs/s)	Consuming throughput (msgs/s)	End-to-end latency (ms)
Apache Kafka	64	25	8	16	272 828	272 705	48
Apache Pulsar	4	25	8	16	176 439	176 388	50
NATS Streaming	Not applicable	25	8	16	26 699	26 696	148

With the same number of sending and receiving threads, Apache Kafka is outstanding with the throughput of approximately 270 000 messages per second and the 95th percentile value of latency around 48 milliseconds. Next is Apache Pulsar with the throughput of roughly 170 000 messages per second. The end-to-end latency of Pulsar is as good as Kafka with 50 milliseconds. NATS Streaming performance is not as good as the other platforms with throughput of only 26 000 messages per second and latency of 148 milliseconds. This is understandable since only one server instance in the NATS cluster can serve requests.

To summary, with the same hardware capacity, Kafka has the best performance among three platforms in term of both throughput and end-to-end latency. Apache Pulsar comes is in the middle with the same latency as Kafka and lower throughout. NATS Streaming has the lowest throughput and the highest end-to-end latency.

4.12 Feature matrix

Finally, the evaluation results of three platforms are put together and incorporated into the feature matrix. The general structure of the matrix is showed in figure 4.34.

							Apache Kafka	
Evaluation category / Evaluation criteria	**Required**	**Criteria priority**	**Scale factor**	**Evaluation scale**	**Points**	**Scaled points**	**Evaluation**	**Points**
Event storage								
Durable storage		High	3	Supported / Partly supported / Not supported	10	30	Supported	30
Flexible data retention policy		High	3	Supported / Partly supported / Not supported	10	30	Supported	30
Data archive in cheap storage (hot/cold storage)		Not considered	0	Supported / Partly supported / Not supported	10	0		0
Messaging patterns								
Publish-Subscribe		High	3	Supported / Partly supported / Not supported	10	30	Supported	30
Competing consumer		High	3	Supported / Partly supported / Not supported	10	30	Partly supported	15
Publish-Subscribe + Competing consumer (Consumer Group)	TRUE	High	3	Supported / Partly supported / Not supported	10	30	Supported	30
Event playback		High	3	Supported / Partly supported / Not supported	10	30	Supported	30
Content-based routing		Not considered	0	Supported / Partly supported / Not supported	10	0		0

Figure 4.34: General structure of the feature matrix.

In the matrix, users has the flexibility to choose different priority for each criterion.

5 Implementation of Highest Delivery Guarantee

As elaborated in section 4.4, Apache Kafka and Apache Pulsar can support exactly-once semantics as the highest guarantee while on NATS Streaming, only at-least-once delivery semantics can be achieved. In order to analyze in more detail how the platforms achieve their highest level of delivery guarantee, especially exactly-once semantics, a small use case is implemented on all three platforms. Apart from providing insights into the internal workflow of the platforms, this can also serve as a reference for realizing the delivery guarantee on the platforms in production.

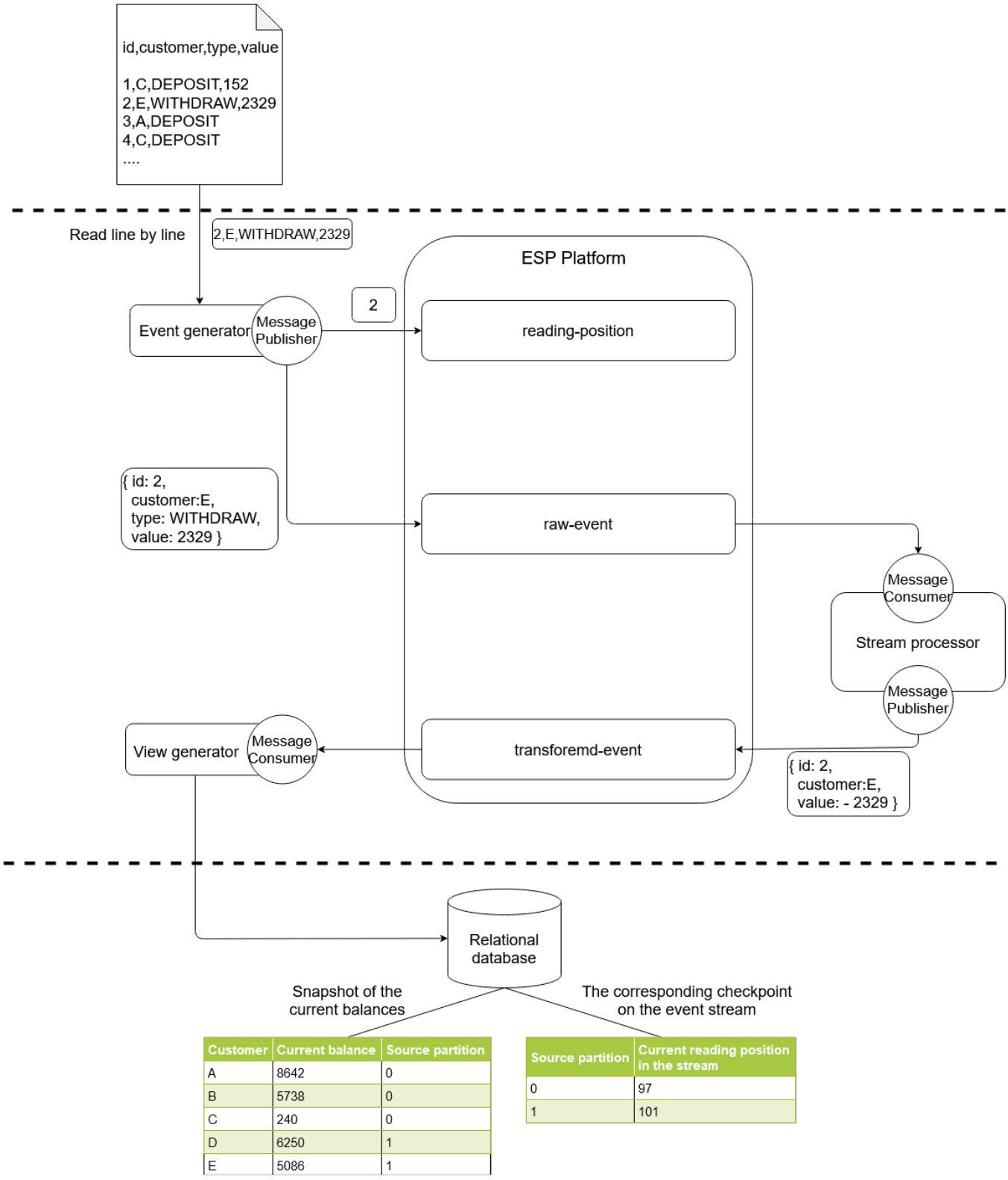
5.1 Overview

For the realization of delivery guarantee, a simple use case of banking transactions is used. Transactional activities of customers are recorded as events and published to the ESP platform. These raw events are transformed and the values of all transactions are aggregated to generate the current account balance for each user. The overview of event flow and the main components in the implementation is showed in figure 5.1.

To simulate the incoming events from customers, an event generator is implemented. The generator reads events from a CSV file which is prepared in advanced with data of 1000 transactional events. The generator then publishes these events to an event stream named *raw-event*. Moreover, the event generator also checkpoints the line number in the CSV file of the published event to another stream named *reading-position* on the ESP platform. In this way, when the event generator restarts, it can retrieve this checkpoint and resume the reading on the source file at the right position.

To transform raw event, a stream processor is implemented. This component will ingest data from *raw-event* stream, extract and transform the transactional value based on the event type and publish this transformed event back into another stream *transformed-event*. This component will rely on the built-in mechanism of the ESP platform to manage the reading position on the *raw-event* stream.

Finally at the end of the event streams pipeline, the view generator component will accumulate the transactional values of each user and persist the resulted current balance to a PostgreSQL relational database. In addition, the view generator also commit the



Customer	Current balance	Source partition
A	8642	0
B	5738	0
C	240	0
D	6250	1
E	5086	1

Source partition	Current reading position in the stream
0	97
1	101

Figure 5.1: Use case to implement delivery guarantee on the platforms.

corresponding reading position on the source stream *transformed-event* to the database so that it can fetch this value and resume the consumption on the stream accordingly upon restarting.

With this setup, message duplication can be easily detected because any transactional event which is processed more than once by any of the three processing components will result in different final current balances. To verify that the configuration and implementation is correct on each platform to achieve the highest message delivery guarantee, different failure scenarios are setup. The final current balances in these scenarios will be verified against the result from the reference scenario without failure.

5.1.1 Failure scenarios

To focus on the components which interact directly with the platform, the source CSV file with events data and the relational database is assumed to be reliable with no failure. Moreover, the platforms are configured to be fault-tolerant with redundancies and assumed to be resilient to failures. Failure scenarios will only be derived for three components: event generator, stream processor, and view generator.

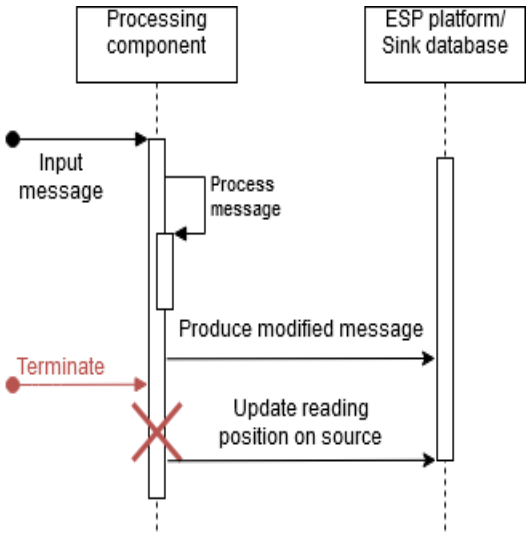


Figure 5.2: Failure scenarios *event-generator-crash*, *stream-processor-crash* and *stream-aggregator-crash*.

These components all have the processing cycle of an event as read-modify-write. After writing the modified event, these components must successfully update the reading position on the source system. Otherwise, failure when updating reading position can lead to the case where the last processed message will be redelivered and reprocessed when

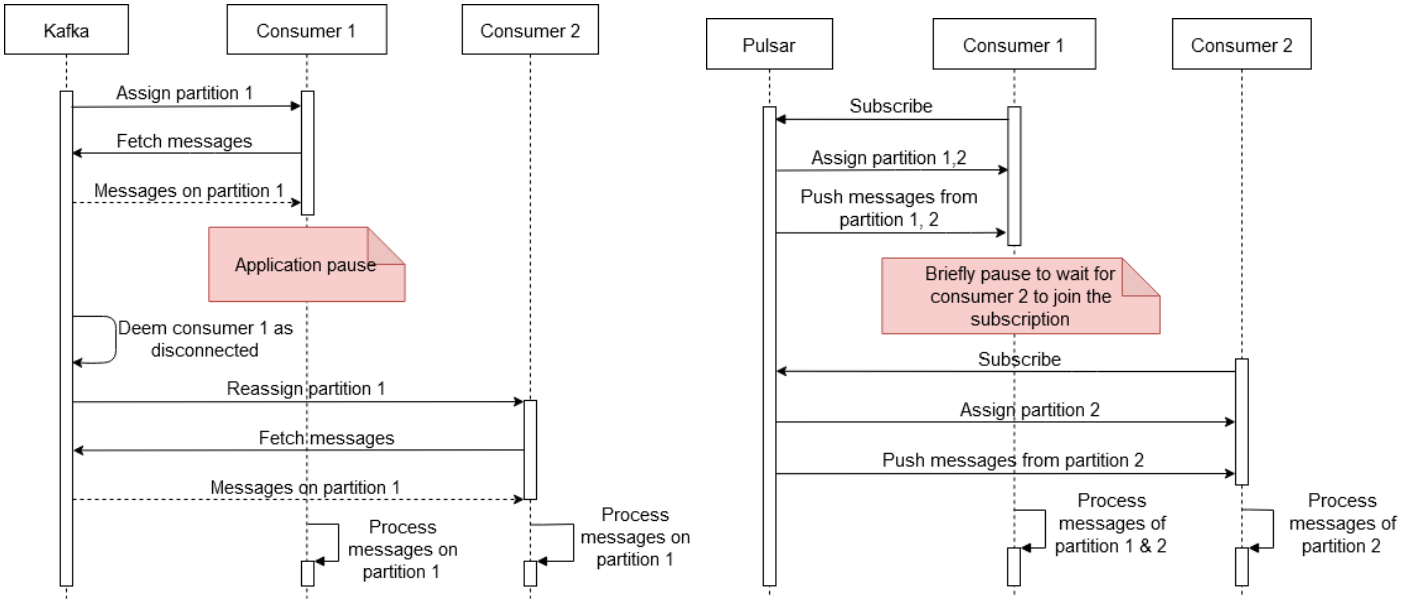
the application restarts. Three failure scenarios, namely, *event-generator-crash*, *stream-processor-crash* and *stream-aggregator-crash*, are derived for three processing components. In each failure scenario, one processing component along the processing pipeline is deliberately terminated before it can checkpoint the reading position on the data source to simulate the application crash (figure 5.2). After that, the processing component will be restarted to continue with its incomplete task.

Kafka and Pulsar support using the consumer group pattern to read a partitioned topic with multiple concurrent consumer instances (section 4.3). In this case, users have the option to rely on the failover mechanism of the consumer group to automatically manage and assign partitions of the topic to consumer instances in the same group. However, this can lead to the case same messages are sent and buffered on two different consumer instances when the platform transfer ownership of a partition from one consumer to another. Therefore, for Apache Kafka and Apache Pulsar, another failure scenario called *duplicated-consumer* is setup.

On Kafka, whenever a consumer send a pulling request for new messages from Kafka, it will be notified about partition reassignment event if there is any. In normal condition, when the partitions are redistributed among the consumers (e.g. when a new consumer joins the consumer group), before the new consumer is notified about its newly assigned partition, the previous owner of the partition always has the chance to gracefully give up the ownership on the partition by committing the offset number of processed messages [41]. Kafka handles the coordination of this transition process transparently to users. Therefore, same messages will not be delivered to two different consumers in this case.

However, ownership on a partition can be revoked before the owning consumer is notified. This is the case when the consumer is stalled for some reason (e.g. garbage collection pause, some messages take longer to process than expected) and fails to send new messages pulling request within the permitted interval. In this case, the consumer will be removed from the group by Kafka and its partition will be reassigned to another consumer in the group. As a result, the messages whose offset numbers have not been committed by old consumer will be redelivered to the new one leading to message duplication on two instances. This scenario can be simulated by simply pausing a consumer instance in the consumer group after it has pulled a new batch of messages and before it starts to process and commit offset for these messages until Kafka deems this consumer to be disconnected (figure 5.3a).

The failure scenario *duplicated-consumer* can also occur with Pulsar. When a consumer subscribe to a Pulsar topic, messages on all topic assigned to this consumer will be pushed to a local queue until it is full [42]. When the consumer has consumed and acknowledged half of the buffered messages, new messages will be delivered. When a new consumer joins the same failover subscription, Pulsar broker will assign some topic partitions to it for load balancing. In this case, if the old consumer has not processed and acknowledged all messages of the reassigned partition on its queue, these messages will be



(a) Failure scenario *duplicated-consumer* on Kafka when a partition is reassigned without notifying the previous owning consumer.

(b) Failure scenario *duplicated-consumer* on Pulsar when a new consumer joins the failover subscription.

Figure 5.3: Failure scenario *duplicated-consumer* on Kafka and Pulsar.

redelivered again to the new consumer which causes message duplication on the queue of two consumers. This scenario can be provoked by introduced a small delay into the first consumer instance before it can start to process the messages on its local queue to give the second consumer instance enough time to join the subscription and receive the unprocessed messages again. This is illustrated in figure 5.3b.

The *duplicated-consumer* scenario will be realized on the stream processor component where the consuming and producing of messages are done within the protection scope of the platform. For the view generator component which reads input from the ESP platform and writes output data to an external data sink, in order to have exactly-once semantics with this scenario, the gap between the platform and the data sink varies and must be bridged differently with different types of external system.

5.2 Implementation

5.2.1 Clients

The three main components are implemented with the low level Java client APIs of the platforms. Consumer clients of the platforms are used to subscribe and read mes-

	Event generator	Stream processor	View generator
event-generator-crash	one instance crash and restart	one instance no failure	one instance no failure
stream-processor-crash	one instance no failure	one instance crash and restart	one instance no failure
view-generator-crash	one instance no failure	one instance no failure	one instance crash and restart
duplicated-consumer *(Kafka and Pulsar only)*	one instance no failure	two instances first instance: is delayed second instance: no failure	one instance no failure

Table 5.4: Summary of 4 failure scenarios.

sages on the streams while producer clients are used to publish messages to the platforms.

Event generator and Stream processor

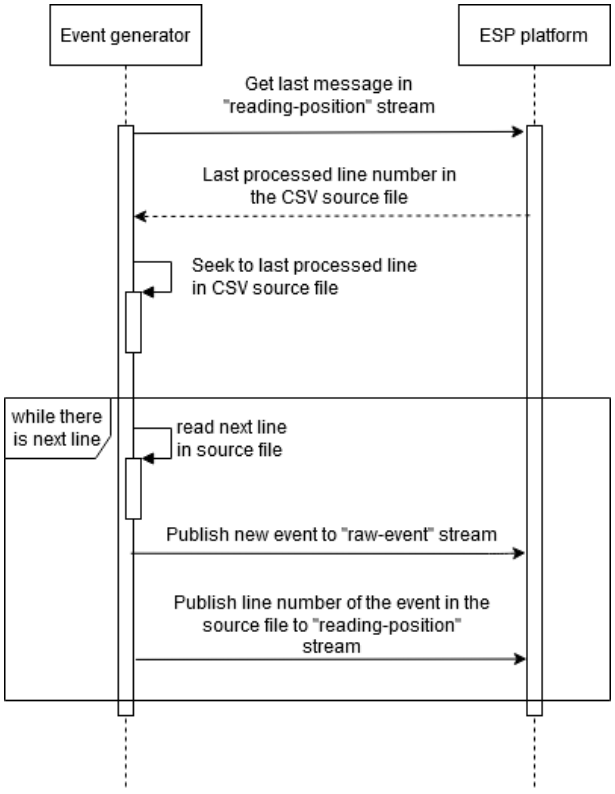


Figure 5.5: Sequence diagram of event generator component.

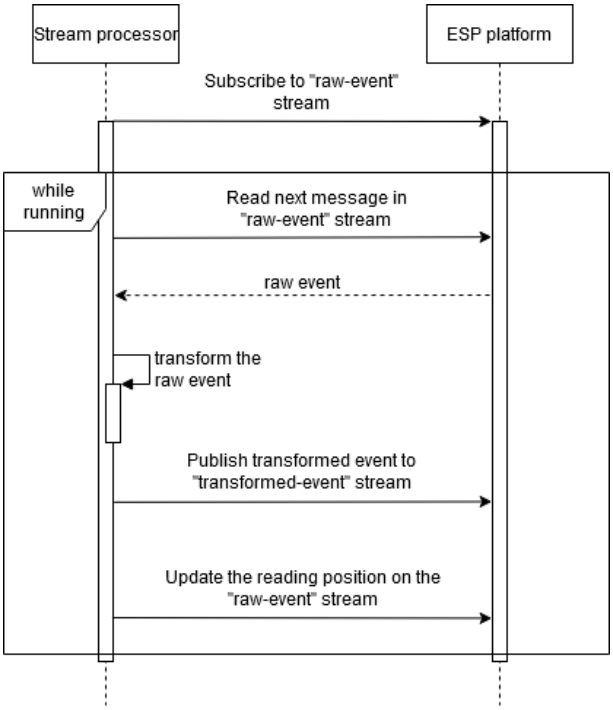


Figure 5.6: Sequence diagram of stream processor component.

Figures 5.5 and 5.6 show the sequence diagrams of the event generator and the stream processor respectively. With Apache Kafka and Apache Pulsar, to achieve exactly-once semantics with these two components, the two operations to publish events and update the reading position on the data source must be atomically combined with the transaction features of the two platforms.

Listing 5.7 shows part of the source code of the stream processor component for Kafka. The transaction, in which transformed events are published to *transformed-event* topic and the offset numbers of consumed messages on *raw-event* topic are committed to Kafka, is started on line 12 and committed on line 29.

```
1  public class KafkaStreamProcessor {
2    private KafkaProducer<String,String> producer;
3    private KafkaConsumer<String,String> consumer;
4    //Initialize producer and consumer, subscribe consumer to Kafka topic
5    public void transformRawEvent(){
6      while(true) {
7        //Pull a new batch of messages from Kafka
8        ConsumerRecords<String, String> consumerRecords = consumer.poll(Duration.ofSeconds(10));
9        //..
10       try{
11         //Begin the transaction
12         producer.beginTransaction();
13
14         //Loop through the pulled messages
15         for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
16           producerRecord = processConsumedRecord(consumerRecord);
17           //Send transformed event to 'transformed-event' topic
18           RecordMetadata recordMetadata=producer.send(producerRecord).get();
19         }
20
21         //Add the hook here to inject custom code during runtime to simulate the application crash
22         bytemanHook(counter);
23
24         //Update the offset on the source topic 'raw-event' in the same transaction
25         producer.sendOffsetsToTransaction(getOffsetToCommitOnSourceTopic(consumerRecords),
26                                 consumer.groupMetadata());
27
28         //Commit the transaction
29         producer.commitTransaction();
30       } catch (ProducerFencedException e) {
31         producer.close();
32       } catch (CommitFailedException e) {
33         producer.abortTransaction();
34       } //..
35     }
36   }
37   //Helper methods
38 }
```

Listing 5.7: Kafka stream processor with the transaction feature.

To ensure exactly-once semantics, Kafka has fencing mechanism in its transactions to prevent duplication of messages and notify users by throwing back exceptions. When there is two producer instances with the same ID trying to publish messages to Kafka,

requests from older instance will be rejected by Kafka and a *ProducerFencedException* will be returned to this instance. When a temporarily disconnected application is replaced by a newer instance and the old instance later reconnects, this could lead to two instances sending message simultaneously. This fencing mechanism helps prevent that scenario.

Fencing mechanism of Kafka also prevents messages duplication when partition reassignment occurs within a consumer group [92]. When the offset number of source topic is committed in a transaction (as in line 25 in listing 5.7), the metadata of the consumer which is used to read the input message must also be sent to Kafka. When this offset committing request is accepted by Kafka, the partition to which the message belongs is locked by the consumer instance declared in the request. If partition reassignment is triggered after the lock is obtained, the new owner of the reassigned partition must wait until the ongoing transaction on the locked partition is either completed, aborted or timeout before it can pull new messages from Kafka. Moreover, Kafka ensures that a partition can only be locked by a consumer with up-to-date metadata. When a consumer is unaware of the latest partition reassignment, its metadata becomes obsolete and any offset committing requests with this consumer will be rejected by Kafka and a *CommitFailedException* will be returned. Any stream processor instance with the consumer which is deemed disconnected by Kafka cannot publish any new message until its consumer rejoins the group and retrieves the latest metadata.

The implementation of the event generator with Kafka transaction is fairly similar. However, since the event generator reads input data from the CSV file rather than a Kafka topic, it only publishes reading position as a normal message to Kafka in the same transaction instead of committing offset number as in the stream processor.

For Apache Pulsar, listing 5.8 shows part of the source code for the stream processor with the transaction feature.

```
1  public class PulsarStreamProcessor {
2      private PulsarClient client;
3      private Producer<String> producer;
4      private Consumer<String> consumer;
5    //Initialize Pulsar client, producer and consumer; subscribe consumer to Pulsar topic
6      public void transformRawEvent(){
7          //..
8          while(true){
9              try {
10                 //Read new record from the buffer queue
11                 message = consumer.receive();
12                 //Transform the raw event
13                 String transformedEvent = processRawEvent(message);
14                 //Create a transaction
15                 Transaction txn = client.newTransaction()
16                                         .withTransactionTimeout(5, TimeUnit.MINUTES)
17                                         .build()
18                                         .get();
19                 //Publish the transformed event to 'transformed-event' topic
```

```
20                producer.newMessage(txn)
21                        .key(customerId)
22                        .value(objectMapper.writeValueAsString(transformedRecord))
23                        .send();
24
25                //Add the hook here to inject custom code during runtime to simulate the application
                        crash
26                bytemanHook(counter);
27
28                //Acknowledge the consumption of message on 'raw-event' topic
29                consumer.acknowledgeAsync(message.getMessageId(),txn);
30                txn.commit().get();
31             } catch (PulsarClientException e) {
32                e.printStackTrace();
33             } //..
34          }
35       }
36    }
```

Listing 5.8: Pulsar stream processor with the transaction feature.

In the transaction from line 15 to line 30, the transformed event and the acknowledgement of the corresponding raw event are sent to Pulsar as an atomic operation. Pulsar handles the case of partition reassignment within a consumer group quite differently from Kafka to ensure exactly-once semantics. When a consumer acknowledge the consumption of a message in a transaction, Pulsar locks this message for that specific consumer [55] until the transaction is completed or aborted. If a message on a partition is locked before partition reassignment occurs, it will not be redelivered to the new owner of the partition. On the other hand, if rebalancing happens before an application instance acknowledges and obtains lock on a message, this message will be redelivered to the new owner of the partition and therefore simultaneously exists on the buffers of two different consumer instances. In this case, the first application instance to finish processing and acknowledge will obtain the lock and cause the other instance to back off with a *TransactionConflictException* when it tries to acknowledge the same message.

The transaction in Pulsar event generator is implemented similarly. Instead of acknowledging message on the source topic, the event generator update the reading position by publishing the line number of published event in the CSV file in the same transaction.

With NATS Streaming, atomically publishing and acknowledging the consumption of multiple messages is not supported. Therefore, the event generator and stream processor are implemented according to the sequence diagrams in figures 5.5 and 5.6 without any special remark.

View generator

Regarding the view generator component, the implementations on all three platforms are fairly similar. To interact with the relational database and map Java object to relational table, the OpenJPA which is an implementation of Java Persistence API (JPA) [93] is

used. Two entities, namely, *CurrentBalance* and *CurrentReadingPosition* (listings 5.9 and 5.10), representing the snapshot of current balances and reading position on the source stream respectively are defined and mapped to two corresponding tables on the PostgreSQL database.

```
@Entity
@Table
public class CurrentBalance {
    @Id
    private String customerId;
    private float currentBalance;
    private int sourcePartition;
  //Constructors, getter, setter methods
}
```

Listing 5.9: Current balance entity.

In the current balance entity, apart from the customer ID and the current balance of the corresponding customer, there is an additional field which indicates the source partition to which events from this customer is published on the ESP platform. This field is applicable in case of Kafka and Pulsar where multiple view generator instances can consume the *transformed-event* topic concurrently. When an application instance is assigned a partition, it must use this source partition field to retrieve the latest snapshots of current balances of all customers on this partition. In case of NATS Streaming, topic cannot be partitioned and therefore this field will simply be assigned a default value.

```
@Entity
@Table
public class CurrentReadingPosition {
    @Id
    private int sourcePartition;
    private long currentReadingPosition;
  //Constructors, getter, setter methods
}
```

Listing 5.10: Current reading position entity.

For the current reading position entity, each partition on the source topic will be mapped to a row in the database table and is uniquely identified by the partition number. In case of NATS Streaming, there is only one row with the default partition number.

Figure 5.11 illustrates the overall workflow of the view generator component.

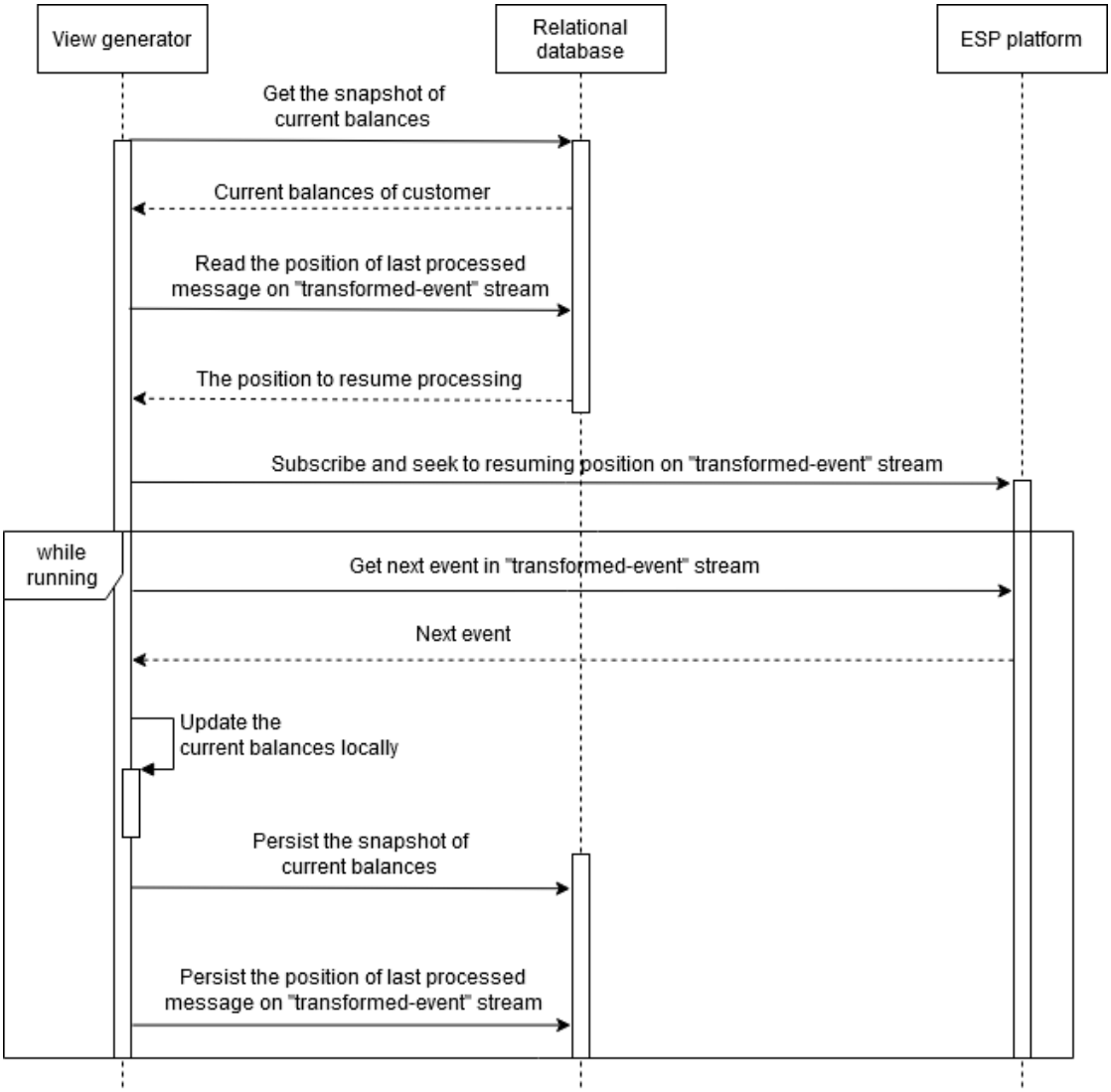


Figure 5.11: Sequence diagram of view generator component.

The two operations to persist snapshot of current balances and corresponding reading position are conducted in a transaction to ensure the atomicity and guarantee exactly-once semantics in case the view generator crashes in the middle on a transaction.

5.2.2 Failure injection

To provoke failure scenarios described in section 5.1.1, custom codes are injected into the the application at runtime to pause or terminate the program using Byteman tracing tool for Java [94]. With this tool, Byteman rule can be defined to inject arbitrary code

into a running application when a condition is met. A Byteman rule contains of three main parts:

- Event: this triggers the rule
- Condition: this part checks the current status of the running application
- Action: this part defines the alternative behavior of the application if the condition is satisfied

```
1 RULE Fault injection to the stream processor, simulate application crash during a transaction to
      published transformed event and commit the offset number of the processed raw event.
2 CLASS kafka.clients.KafkaStreamProcessorNew
3 METHOD bytemanHook(int)
4 AT EXIT
5 BIND counter = $1
6 IF counter >= 120
7 DO debug("Simulate application crash after processing 120 events or more"), killJVM()
8 ENDRULE
```

Listing 5.12: Byteman rule script for stream-processor-crash scenario on Kafka.

In the example in listing 5.12 is the Byteman rule to inject failure during runtime into the stream processor component of Kafka. The event is defined from line 2 to 4. When the method *bytemanHook* in the class *KafkaStreamProcessor* is executed, the rule is invoked. The *bytemanHook* is a predefined helper method to provide a convenient way to specify the injecting position in the class without having to modify much in the application code. In the stream processor class of Kafka (listing 5.7), this method is placed between the operations to publish transformed events and commit offset of processed raw events. The *bytemanHook* method does nothing and accepts as parameter a counter variable indicating the number of messages which have been processed by the processor since started. This counter is used in the condition of the Byteman rules online 6, namely, when the stream processor has processed and published successfully more than 120 messages, the action defined on line 7 which intentionally terminates the application is executed.

Different rules corresponding to different failure scenarios on each processing components are defined similarly for all three platforms. To execute the application with defined Byteman rule, the Java virtual machine (JVM) must be configured to load and run the executable jar file of the Byteman engine when the application is executed. In addition, the script of the rule to be executed must be specified as well.

```
1 java -javaagent:./byteman.jar=script:./byteman_streamprocessor.btm -jar kafkaClient.jar
```

Listing 5.13: Execute the stream processor of Kafka with the Byteman rule.

5.2.3 Running environment

To run the failure scenarios on each platform, Docker compose is used to containerize required infrastructure and processing components and run them locally in a Docker environment [95].

The infrastructure comprises of the ESP platform and the PostgreSQL relational database. The ESP platforms are set up with multiple instances running in different Docker containers to be fault-tolerant:

- Apache Kafka: three Zookeeper instances and three Kafka broker instances.
- Apache Pulsar: three Zookeeper instances, three BookKeeper instances and two Pulsar broker instances.
- NATS Streaming: three NATS Streaming server instances in clustering mode and file-based persistence.

While Apache Pulsar and NATS Streaming have official Docker images, Apache Kafka is not released with a Docker image. Nevertheless, there are a number of Kafka Docker images provided by third-party organizations. In the thesis, the Kafka Docker image from the company Confluent is used as it is regularly updated.

Regarding the processing components, their executable files are containerized and run in the same Docker environment as the infrastructure. For each processing component, different containers are prepared for the normal instance without failure and the fault-injected instance by Byteman.

To conveniently start the failure scenarios, scripts are prepared to start up the Docker containers of the infrastructure, create the necessary streams on the platforms, and run the processing components.

5.3 Results and Discussion

The results from running the failure scenarios on the three platforms are summarized in table 5.14.

	Apache Kafka	Apache Pulsar	NATS Streaming
event-generator-crash	Exactly-once	Exactly-once	At-least-once
stream-processor-crash	Exactly-once	Exactly-once	At-least-once
view-generator-crash	Exactly-once	Exactly-once	Exactly-once
duplicated-consumer	Exactly-once	Exactly-once	Not applicable

Table 5.14: Summary of 4 failure scenarios.

As expected, the transactional features of Kafka and Pulsar help ensure exactly-once semantics in the case of *event-generator-crash* and *stream-processor-crash* failure scenarios. On the other hand, NATS Streaming can only provide at-least-once guarantee in these scenarios.

With the *view-generator-crash* scenario, exactly-once semantics is guaranteed with all three platforms. Nevertheless, this is only resulted from the support of transaction on the sink relational database and requires manual implementation on the application level to bridge the gap between the two data systems.

Regarding the *duplicated-consumer* failure scenario, the locking mechanisms of the transactions on Kafka and Pulsar are effective to guarantee exactly-once semantics even when a message is delivered and processed by two different stream processor instances when partition reassignment occurs.

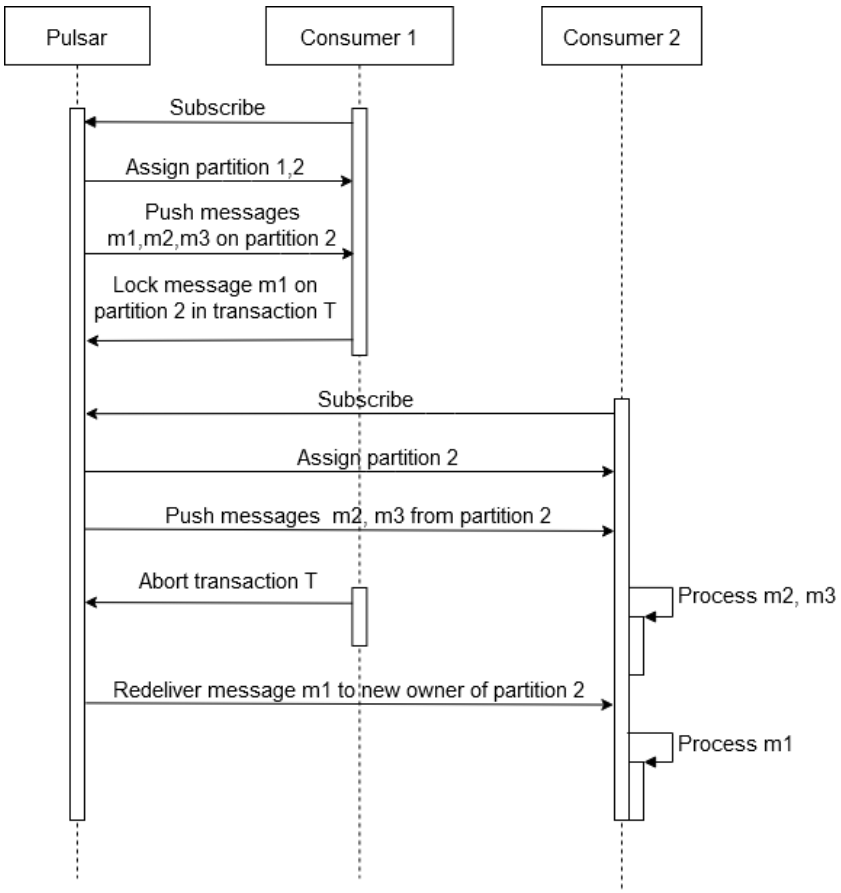


Figure 5.15: Pulsar transaction can result in out-of-order processing of messages when transaction is aborted.

However, the transaction feature on Pulsar can potentially lead to out-of-order process-

ing of messages on the same partition. With the Pulsar transaction, when a message in locked by a consumer instance by being acknowledged in a transaction, this message cannot be redelivered to another consumer instance or cannot be acknowledged by other consumers if it is already delivered to their local queues. However, subsequent messages on the same partition can still be processed and acknowledged by other consumers. In the scenario depicted in figure 5.15, when partition is reassigned to consumer 2 when it joins the subscription, message m1 is already locked by consumer 1 and therefore will not be pushed to consumer 2. However, when consumer 1 fails to complete the transaction, message m1 will return to unacknowledged state and will be redelivered and processed again. This causes message m1 to be processed out-of-order after message m2 and m3 even though they are on the same partition where order is guaranteed.

Moreover, this mechanism of Pulsar to let consumers race to lock messages when they have the same messages buffered locally can lead to unexpected behaviors during runtime when a partition is not guaranteed to be consumed by only one application instance at any time. Nevertheless, since this transaction feature of Pulsar is still actively developed and refined in the technical review phase, this problem could be soon resolved in future releases.

The problem of out-of-order processing is well prevented on Kafka. When a consumer in the consumer group commits the offset number of a message in a transaction, the entire partition to which the message belongs is locked and any other consumer in the same group cannot fetch new messages from this partition even when it is assigned as the new owner of the partition. Only when the ongoing transaction is either completed or aborted, new owner of the partition can start to pull new messages. On the old owner of the partition, after the ongoing transaction is finished, if there is any buffered message left from the last pull request, this application instance cannot continue to commit offset numbers for them because it now has the outdated metadata of the consumer group before the partitions reassignment. When this instance sends a new pulling requests to Kafka, it receives the latest metadata as well as only messages on the partition which it is currently assigned.

Therefore, at any time, only one consumer in a Kafka consumer group can process and commit offset numbers of messages on a partition. This mechanism of Kafka compromises the latency and availability to ensure the consistency when partition reassignment occurs. Moreover, all low-level implementation of transactions with Kafka producers and consumers as in this thesis is provided out-of-the-box with the Kafka Streams library. Users can utilize this library to focus only on the processing logic while still being able to achieve exactly-once semantics on Kafka with minimum configuration.

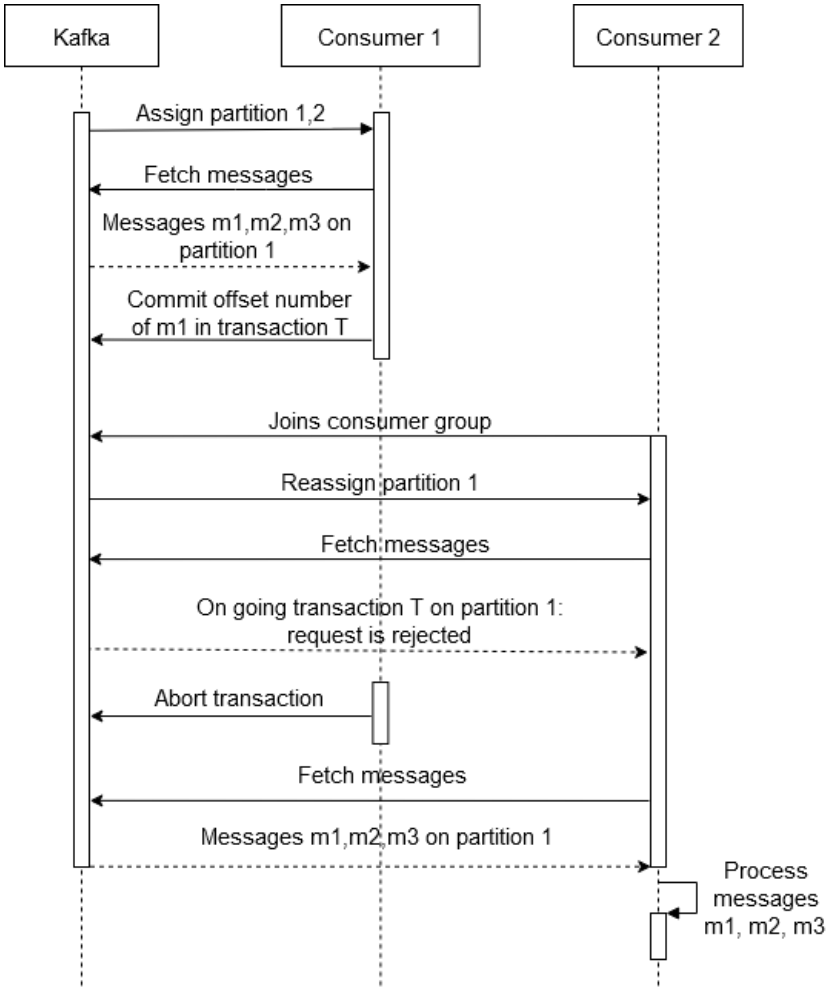


Figure 5.16: Kafka transaction can prevent out-of-order processing of messages when transaction is aborted.

List of Acronyms

ACL Access Control List.

API Application Programming Interface.

ESP Event Stream Processing.

RBAC Role-Based Access Control.

SASL Simple Authentication and Security Layer.

List of Figures

2.1 Systems coordination with event notification. 5
2.2 Systems coordination with event-carried state transfer. 5
2.3 Systems coordination with event sourcing. 6
2.4 Stream processing concept. 7
2.5 The tangled data systems at LinkedIn in the old day [13]. 8
2.6 System with streams of events as the single source of truth. 8
2.7 Capabilities of an ESP platform. 9

4.1 General concept of Apache Kafka. 20
4.2 General concept of Apache Pulsar. 21
4.3 General concept of NATS Streaming. 22
4.4 Data replication model for fault tolerance of event storage on Kafka. . . . 24
4.5 Log compaction on Kafka (taken from Kafka documentation [32]). 26
4.6 Underlying storage layers of a Pulsar topic. 27
4.7 Message retention policy on Pulsar. 28
4.8 Fault tolerance of event storage on NATS Streaming in clustering mode. . 30
4.9 Competing consumers pattern with Kafka. 31
4.10 Consumer group pattern with Kafka. 32
4.11 Event playback with Kafka. 33
4.12 Consumer group pattern with Pulsar. 35
4.13 Event playback with Pulsar. 36
4.14 Event playback with NATS Streaming. 38
4.15 Asynchronous publishing and retrying causes out-of-order messages. . . . 39
4.16 Duplication of messages with at-least-once delivery semantics. 40
4.17 Support of exactly-once semantics on Kafka (*green*: fully supported, *yellow*: partly supported, *red*: not supported). 42
4.18 Support of exactly-once semantics on Pulsar (*green*: fully supported, *yellow*: partly supported, *red*: not supported). 45
4.19 NATS Streaming server redelivers messages out-of-order: message 1 and 2 are processed longer than the acknowledgement timeout and therefore are resent by server. Client sees the sequence of messages 1, 2, 3, 1, 4, 2. . 46
4.20 Support of exactly-once semantics on NATS Streaming (*green*: fully supported, *yellow*: partly supported, *red*: not supported). 48
4.21 Supported window types for stream processing with Kafka Streams (taken from [61]). 49

4.22 Kafka Connect cluster to integrate data between Kafka and external data systems. . . . 52
4.23 Kafka clustered is scaled up with a new broker. . . . 56
4.24 Kafka clustered is scaled up with a new broker. . . . 57
4.25 Storage layer of Pulsar can be scaled up seamlessly by starting new Bookkeeper instances. . . . 58
4.26 Request-serving layer of NATS Streaming can be scaled with the partitioning feature. . . . 59
4.27 Secured Kafka cluster. Red links are encrypted. . . . 62
4.28 Secured Pulsar cluster. Red links are encrypted. . . . 64
4.29 Secured NATS Streaming cluster. Red links are encrypted. . . . 65
4.30 Contributions to Kafka on GitHub [89]. . . . 66
4.31 Contributions to Pulsar on GitHub [90]. . . . 66
4.32 Contributions to NATS Streaming on GitHub [91]. . . . 67
4.34 General structure of the feature matrix. . . . 69

5.1 Use case to implement delivery guarantee on the platforms. . . . 71
5.2 Failure scenarios *event-generator-crash*, *stream-processor-crash* and *stream-aggregator-crash*. . . . 72
5.3 Failure scenario *duplicated-consumer* on Kafka and Pulsar. . . . 74
5.5 Sequence diagram of event generator component. . . . 75
5.6 Sequence diagram of stream processor component. . . . 75
5.11 Sequence diagram of view generator component. . . . 80
5.15 Pulsar transaction can result in out-of-order processing of messages when transaction is aborted. . . . 83
5.16 Kafka transaction can prevent out-of-order processing of messages when transaction is aborted. . . . 85

List of Tables

3.1 Preliminary evaluation of 5 open-source ESP platforms. 11
3.2 Considered evaluation criteria in the thesis. 19

4.33 Benchmarking results of Kafka, Pulsar and NATS Streaming [3]. 68

5.4 Summary of 4 failure scenarios. 75
5.14 Summary of 4 failure scenarios. 82

List of Listings

5.7 Kafka stream processor with the transaction feature. 76
5.8 Pulsar stream processor with the transaction feature. 77
5.9 Current balance entity. 79
5.10 Current reading position entity. 79
5.12 Byteman rule script for stream-processor-crash scenario on Kafka. 81
5.13 Execute the stream processor of Kafka with the Byteman rule. 81

Bibliography

[1] Netflix Inc., "Netflix Technology Blog: Evolution of the Netflix Data Pipeline." https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905. Accessed: 2020-11-14.

[2] V. C. Alok Nikhil, "Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest?." https://www.confluent.de/blog/kafka-fastest-messaging-system/. Accessed: 2020-11-12.

[3] A. Warski, "Evaluating persistent, replicated message queues (2020 edition)." https://softwaremill.com/mqperf/. Accessed: 2020-11-12.

[4] P. Li and S. Guo, "Benchmarking Pulsar and Kafka - A More Accurate Perspective on Pulsar's Performance." https://streamnative.io/en/blog/tech/2020-11-09-benchmark-pulsar-kafka-performance. Accessed: 2020-11-12.

[5] Confluent, "Kafka vs. Pulsar vs. RabbitMQ: Performance, Architecture, and Features Compared." https://www.confluent.io/kafka-vs-pulsar/. Accessed: 2020-11-12.

[6] B. Stopford, Designing Event-Driven Systems Concepts and Patterns for Streaming Services with Apache Kafka, ch. 5, pp. 31–38. O'Reilly Media, Inc., 2018.

[7] G. Young, "CQRS Documents." https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf, 2010. Accessed: 2020-11-23.

[8] M. Fowler, "What do you mean by "Event-Driven"?." https://martinfowler.com/articles/201701-event-driven.html. Accessed: 2020-11-19.

[9] M. Fowler, "Event sourcing." https://martinfowler.com/eaaDev/EventSourcing.html. Accessed: 2020-11-23.

[10] C. Kiehl, "Don't Let the Internet Dupe You, Event Sourcing is Hard." https://chriskiehl.com/article/event-sourcing-is-hard, 2019. Accessed: 2020-11-23.

[11] T. Akidau, "Streaming 101: The world beyond batch." https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/. Accessed: 2020-11-17.

[12] J. Kreps, "Questioning the Lambda Architecture." https://www.oreilly.com/radar/questioning-the-lambda-architecture/. Accessed: 2020-11-18.

[13] J. Kreps, "Putting Apache Kafka To Use: A Practical Guide to Building an Event Streaming Platform (Part 1)." https://www.confluent.io/blog/event-streaming-platform-1/. Accessed: 2020-11-18.

[14] J. Kreps, "The Log: What every software engineer should know about real-time data's unifying abstraction." https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying. Accessed: 2020-11-18.

[15] The Kafka contributors, "Apache Kafka." https://kafka.apache.org/. Accessed: 2020-11-19.

[16] The Pulsar contributors, "Apache Pulsar." https://pulsar.apache.org/. Accessed: 2020-11-19.

[17] The RocketMQ contributors, "Apache RocketMQ." https://rocketmq.apache.org/. Accessed: 2020-11-19.

[18] The NATS Streaming contributors, "NATS Streaming concepts." https://docs.nats.io/nats-streaming-concepts/intro. Accessed: 2020-11-19.

[19] The Pravega contributors, "Pravega." https://pravega.io/. Accessed: 2020-11-19.

[20] "ISO/IEC 25010." https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed: 2020-11-22.

[21] G. Hohpe and B. Woolf, "Enterprise Integration Patterns: Messaging Patterns: Publish-Subscribe Channel." https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html. Accessed: 2020-11-23.

[22] G. Hohpe and B. Woolf, "Enterprise Integration Patterns: Messaging Patterns: Competing Consumers." https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html. Accessed: 2020-11-23.

[23] G. Hohpe and B. Woolf, "Enterprise Integration Patterns: Messaging Patterns: Content-Based Router." https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html. Accessed: 2020-11-23.

[24] T. Treat, "You Cannot Have Exactly-Once Delivery." https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/. Accessed: 2020-11-24.

[25] The Zookeeper contributors, "Apache Zookeeper." https://zookeeper.apache.org/. Accessed: 2020-11-26.

[26] C. McCabe, "KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum." https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum. Accessed: 2020-11-26.

[27] J. Gustafson, "KIP-392: Allow consumers to fetch from closest replica Skip to end of metadata." https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica. Accessed: 2020-11-26.

[28] The BookKeeper contributors, "Apache BookKeeper." https://bookkeeper.apache.org/. Accessed: 2020-11-26.

[29] The Kafka contributors, "Kafka Implementation." https://kafka.apache.org/documentation/#implementation. Accessed: 2020-11-26.

[30] The Kafka contributors, "Kafka Configuration: Topic-Level Configs." https://kafka.apache.org/documentation/#configuration. Accessed: 2020-11-26.

[31] The Kafka contributors, "Kafka Design: Replication." https://kafka.apache.org/documentation.html#replication. Accessed: 2020-11-27.

[32] The Kafka contributors, "Kafka Design: Log compaction." https://kafka.apache.org/documentation/#compaction. Accessed: 2020-11-26.

[33] The BookKeeper contributors, "The BookKeeper protocol." https://bookkeeper.apache.org/docs/4.11.1/development/protocol/. Accessed: 2020-11-29.

[34] The BookKeeper contributors, "BookKeeper administration." http://bookkeeper.apache.org/docs/4.11.1/admin/bookies/. Accessed: 2020-11-29.

[35] The Pulsar contributors, "Message retention and expiry." https://pulsar.apache.org/docs/en/cookbooks-retention-expiry/. Accessed: 2020-11-30.

[36] The Pulsar contributors, "Concepts and Architecture: Topic compaction." https://pulsar.apache.org/docs/en/concepts-topic-compaction/. Accessed: 2020-11-30.

[37] The NATS Streaming contributors, "NATS Streaming Server: Configuring." https://docs.nats.io/nats-streaming-server/configuring. Accessed: 2020-12-01.

[38] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pp. 305–319, 2014.

[39] The Kafka contributors, "Kafka Design: The consumer." https://kafka.apache.org/documentation/#theconsumer. Accessed: 2020-12-02.

[40] J. Rao, "How to choose the number of topics/partitions in a Kafka cluster?." https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/. Accessed: 2020-12-03.

[41] The Kafka contributors, "Javadoc: Kafka consumer." https://kafka.apache.org/26/javadoc/index.html?org/apache/kafka/clients/consumer/KafkaConsumer.html. Accessed: 2020-12-03.

[42] The Pulsar contributors, "Pulsar binary protocol specification." https://pulsar.apache.org/docs/en/develop-binary-protocol/. Accessed: 2020-12-04.

[43] The Pulsar contributors, "Concepts and Architecture: Messaging." https://pulsar.apache.org/docs/en/concepts-messaging/. Accessed: 2020-12-04.

[44] The Pulsar contributors and users, "GitHub issue: Some partitions get stuck after adding additional consumers to the KEY_SHARED subscriptions." https://github.com/apache/pulsar/issues/8115. Accessed: 2020-12-06.

[45] The Pulsar contributors, "Concepts and Architecture: Pulsar Clients." https://pulsar.apache.org/docs/en/concepts-clients/. Accessed: 2020-12-06.

[46] The Pulsar contributors, "Javadoc: Pulsar consumer." https://pulsar.apache.org/api/client/2.6.0-SNAPSHOT/org/apache/pulsar/client/api/Consumer.html. Accessed: 2020-12-07.

[47] The Pulsar contributors, "Javadoc: Pulsar reader." https://pulsar.apache.org/api/client/2.6.0-SNAPSHOT/org/apache/pulsar/client/api/Reader.html. Accessed: 2020-12-07.

[48] The NATS Streaming contributors, "NATS Streaming concepts: Channels." https://docs.nats.io/nats-streaming-concepts/channels. Accessed: 2020-12-07.

[49] The NATS Streaming contributors, "Developing with NATS Streaming: Receiving Messages from a Channel." https://docs.nats.io/developing-with-nats-streaming/receiving. Accessed: 2020-12-07.

[50] Confluent, "Introduction to Kafka." https://docs.confluent.io/home/kafka-intro.html. Accessed: 2020-12-14.

[51] A. Mehta, "KIP-98 - Exactly Once Delivery and Transactional Messaging." https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging. Accessed: 2020-12-14.

[52] The Kafka contributors, "Kafka Configuration: Producer Configs." https://kafka.apache.org/documentation/#producerconfigs. Accessed: 2020-12-12.

[53] The Pulsar contributors, "Pulsar adaptor for Apache Kafka." https://pulsar.apache.org/docs/ja/adaptors-kafka/. Accessed: 2020-12-16.

[54] M. Merli and S. Guo, "PIP 6: Guaranteed Message Deduplication." https://github.com/apache/pulsar/wiki/PIP-6:-Guaranteed-Message-Deduplication. Accessed: 2020-12-18.

[55] S. Guo, "PIP 31: Transaction Support." https://github.com/apache/pulsar/wiki/PIP-31%3A-Transaction-Support. Accessed: 2020-12-18.

[56] Wikipedia contributors, "Two-phase commit protocol." https://en.wikipedia.org/wiki/Two-phase_commit_protocol. Accessed: 2020-12-18.

[57] J. Zhao and J. Huang, "What's New in Pulsar Flink Connector 2.7.0." https://streamnative.io/en/blog/release/2020-12-24-pulsar-flink-connector-270. Accessed: 2020-12-10.

[58] The NATS Streaming contributors and users, "GitHub issue: Order of delivery." https://github.com/nats-io/stan.go/issues/179. Accessed: 2020-12-16.

[59] The NATS Streaming contributors, "Developing with NATS Streaming: Acknowledgements." https://docs.nats.io/developing-with-nats-streaming/acks. Accessed: 2020-12-16.

[60] The Kafka contributors, "Kafka Streams: Core concepts." https://kafka.apache.org/26/documentation/streams/core-concepts. Accessed: 2020-12-09.

[61] Confluent, "ksqlDB: Time and Windows." https://docs.ksqldb.io/en/latest/concepts/time-and-windows-in-ksqldb-queries/. Accessed: 2020-12-10.

[62] J. Kreps, "Why local state is a fundamental primitive in stream processing." https://www.oreilly.com/content/why-local-state-is-a-fundamental-primitive-in-stream-processing/. Accessed: 2020-12-10.

[63] The Pulsar contributors, "Pulsar Functions overview." https://pulsar.apache.org/docs/en/functions-overview/. Accessed: 2020-12-09.

[64] The Pulsar contributors and users, "GitHub issue: Unable to create pulsar function." https://github.com/apache/pulsar/issues/8469. Accessed: 2020-12-09.

[65] The Kafka contributors, "Kafka Connect." https://kafka.apache.org/documentation.html#connect. Accessed: 2020-12-10.

[66] The Kafka contributors, "KAFKA-6080: Transactional EoS for source connectors." https://issues.apache.org/jira/browse/KAFKA-6080. Accessed: 2020-12-10.

[67] Confluent, "HDFS 2 Sink Connector for Confluent Platform." https://docs.confluent.io/5.5.0/connect/kafka-connect-hdfs/index.html#features. Accessed: 2020-12-10.

[68] J. Rao, "Kafka: Clients." https://cwiki.apache.org/confluence/display/KAFKA/Clients. Accessed: 2020-12-10.

[69] The Pulsar contributors, "Pulsar Connector overview." https://pulsar.apache.org/docs/en/io-overview/. Accessed: 2020-12-11.

[70] The Pulsar contributors, "Pulsar client libraries." https://pulsar.apache.org/docs/en/client-libraries/. Accessed: 2020-12-11.

[71] The NATS Streaming contributors, "NATS Streaming: clients and utilities." https://nats.io/download. Accessed: 2020-12-11.

[72] The Kafka contributors, "Kafka Operations: Monitoring." https://kafka.apache.org/documentation/#monitoring. Accessed: 2020-12-12.

[73] Oracle Corporation, "Java SE Monitoring and Management Guide, Chapter 2: Monitoring and Management Using JMX Technology." https://docs.oracle.com/javase/8/docs/technotes/guides/management/agent.html, 2006. Accessed: 2020-12-12.

[74] N. Hagen and T. Dehn, "Comparison of Kafka Monitoring Tools." https://www.novatec-gmbh.de/en/blog/comparison_of_kafka_monitoring_tools/. Accessed: 2020-12-12.

[75] The Pulsar contributors, "Reference: Pulsar Metrics." https://pulsar.apache.org/docs/en/reference-metrics/. Accessed: 2020-12-15.

[76] The Pulsar contributors, "Administration: Pulsar Manager." https://pulsar.apache.org/docs/en/administration-pulsar-manager/. Accessed: 2020-12-15.

[77] The NATS Streaming contributors, "NATS Streaming concepts: Monitoring." https://docs.nats.io/nats-streaming-concepts/monitoring. Accessed: 2020-12-15.

[78] The NATS Streaming contributors, "NATS server: Configuration." https://docs.nats.io/nats-server/configuration. Accessed: 2020-12-15.

[79] S. Kozlovski, "Apache Kafka's Distributed System Firefighter — The Controller Broker." https://stanislavkozlovski.medium.com/apache-kafkas-distributed-system-firefighter-the-controller-broker-1afca1eae302. Accessed: 2020-12-18.

[80] The Pulsar contributors, "Pulsar Administration: Pulsar load balance." https://pulsar.apache.org/docs/en/administration-load-balance/. Accessed: 2020-12-19.

[81] The NATS Streaming contributors, "NATS Streaming concepts: Partitioning." https://docs.nats.io/nats-streaming-concepts/partitioning. Accessed: 2020-12-19.

[82] The Kafka contributors, "Kafka Security." https://kafka.apache.org/documentation/#security. Accessed: 2021-01-03.

[83] Wikipedia contributors, "Transport Layer Security." https://en.wikipedia.org/wiki/Transport_Layer_Security. Accessed: 2021-01-03.

[84] Wikipedia contributors, "Simple Authentication and Security Layer." https://en.wikipedia.org/wiki/Simple_Authentication_and_Security_Layer. Accessed: 2021-01-03.

[85] R. Radhakrishnan, "Client-Server mutual authentication." https://cwiki.apache.org/confluence/display/ZOOKEEPER/Client-Server+mutual+authentication. Accessed: 2021-01-03.

[86] The Pulsar contributors, "Security: Pulsar security overview." https://pulsar.apache.org/docs/en/security-overview/. Accessed: 2021-01-04.

[87] The NATS Streaming contributors and users, "GitHub issue: NATS Streaming Client Authorization." https://github.com/nats-io/nats-streaming-server/issues/788. Accessed: 2021-01-05.

[88] The NATS Streaming contributors, "NATS Streaming concepts: Store encryption." https://docs.nats.io/nats-streaming-concepts/store-encryption. Accessed: 2021-01-05.

[89] GitHub, "Kafka: GitHub repository insights." https://github.com/apache/kafka/graphs/contributors. Accessed: 2021-02-04.

[90] GitHub, "Pulsar: GitHub repository insights." https://github.com/apache/pulsar/graphs/contributors. Accessed: 2021-02-04.

[91] GitHub, "NATS Streaming: GitHub repository insights." https://github.com/nats-io/nats-streaming-server/graphs/contributors. Accessed: 2021-02-04.

[92] J. Gustafson, "KIP-447: Producer scalability for exactly once semantics." https://cwiki.apache.org/confluence/display/KAFKA/KIP-447%3A+Producer+scalability+for+exactly+once+semantics. Accessed: 2021-02-10.

[93] Wikipedia contributors, "Jakarta Persistence." https://en.wikipedia.org/wiki/Jakarta_Persistence. Accessed: 2021-02-10.

[94] JBoss community, "Byteman." https://byteman.jboss.org/. Accessed: 2021-02-11.

[95] Docker Inc. , "Overview of Docker Compose." https://docs.docker.com/compose/. Accessed: 2021-02-12.