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 2, 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

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 a ESP platform will be used as it is designed orienting to the concept of streaming. However, in order to choose the suitable platform, user 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].

Some other surveys cover more platforms and a wider range of evaluating aspects such as the comparisons of Apache Kafka, Apache Pulsar and RabbitMQ [4] [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 need of user and adapted to individual use case to determine the most suitable platform for that case according to its priorities.

1.4 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

2.1.1 Microservices with Event-Driven Approach

With microservices architecture, an application is disassembled into services according to different business capabilities [6]. These services are self-contained and loosely-coupled with each other. Each of them maintains a separate database and expose its data with other only via a mutually agreed contract. Since every service itself is an independent deployable, it can have its own development cycle and technology stack. Services also allow finer-grained scaling of the application. This is very useful for reducing cost when deploying the application to the cloud [7].

In microservices architecture, 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 [8].

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

Event
An event represents something occurred in the past and cannot be altered or deleted [9]. It is simply a fact stating what has happened in the system. Whenever a service updates its state, it sends out an event. Any other service 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, services of the system can be more loosely coupled. New service can be easily plugged in the system and start to consume events while other services remain unmodified. There are three

common ways to use events, namely, event notification, event-carried state transfer and event sourcing [10].

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 service.

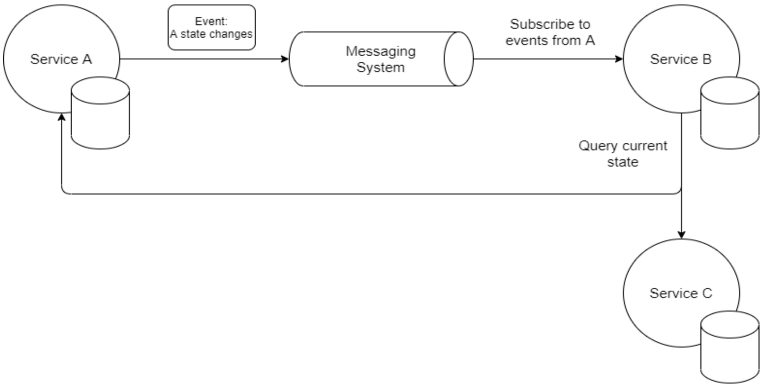

Figure 2.1: Services coordination with event notification.

The approaches of event notification and request-driven ensure a minimum level of coupling by letting each service manage its own data and only share when being requested. Another advantage is that the state of each service is consistent throughout the entire system since it only exists in one place. Nevertheless,when services grow with more functionalities, they need to expand the service contract to expose more data to outside which then leads to higher coupling. This is known as the dichotomy of data and services [11]. Therefore, the two following patterns tackle this problem by proactively allowing services to openly share their data instead of encapsulating it within each service.

Event-Carried State Transfer
With this pattern, each event encloses more detail information about what has been changed as well as how it has been modified. As a result, current state of any service 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.

When a service 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 multiple copies of data across the system also means that the system can be in inconsistent state temporarily or even worse permanently if it is not designed carefully. This

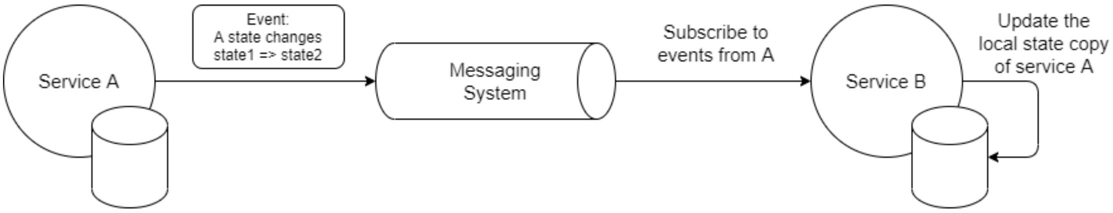

Figure 2.2: Services coordination with event-carried state transfer.

concern is closely related to the problem of how to atomically update the local state and publish a corresponding event [12].

Event Sourcing
The event sourcing pattern takes one step further. Events are not used to only notify and help replicate states among services but they are now the primitive form of data store [13]. Instead of updating the current state, services 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 the entire system. Services can derive the state of any entity from the events stream and keep it in memory or a cache to query when needed. Snapshot of the current state can be periodically created to avoid processing through all events after every restart.

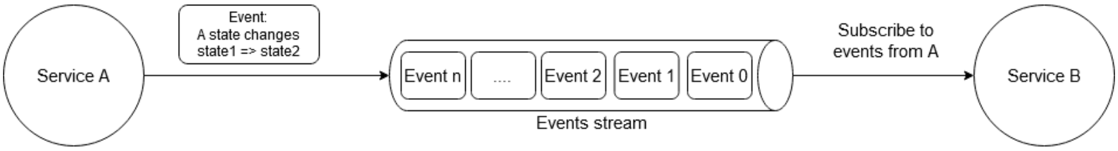

Figure 2.3: Services coordination with event sourcing.

This approach eliminates the need of atomically updating state and generating event as in event-carried state transfer since there is now only publishing of events. Apart from providing a loosely-coupled and scalable way for coordination among services, 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 [14].

Event-driven architecture and its related patterns are useful but it also comes with a cost of increasing complexity in the system [15] [16]. As can be seen from the above short summary, 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 in a data warehouse. 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.

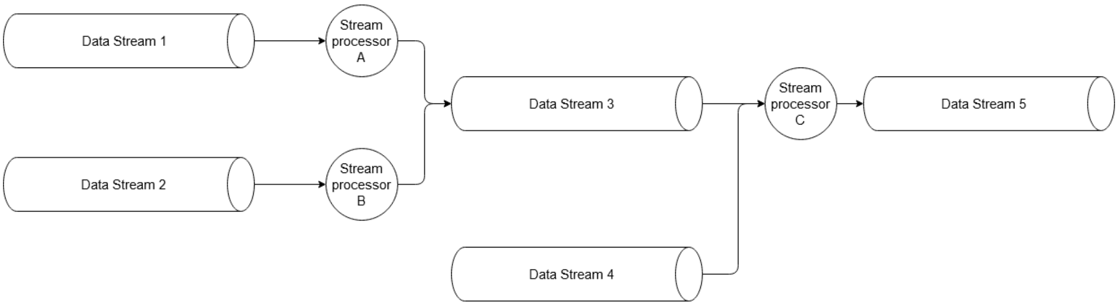

Figure 2.4: Stream processing concept.

By processing this influx of data continuously as they arrives, events and patterns 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 [17]. 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 [18]. As stream processing engines grow more mature and accurate, the demand for such system is lessened [19].

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 on the organizational level 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 emerges not only between services of applications but also between these processing and data systems on a bigger scale. 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 [20].

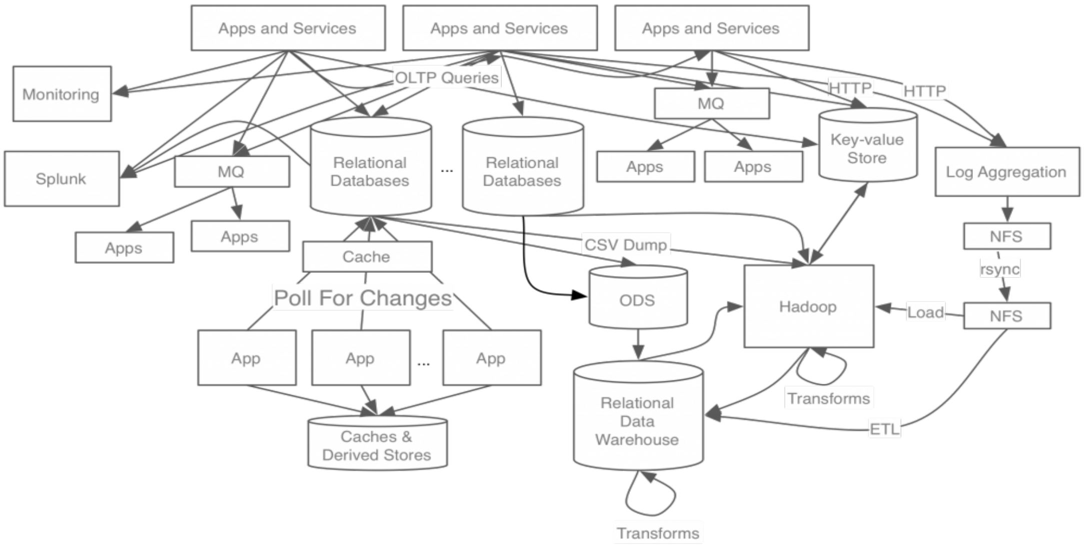

Figure 2.5: The tangled data systems at LinkedIn in the old day [20].

Jay Kreps described in his article a log-centric infrastructure that can solve this problem [21]. Every event recorded by any data system in the organization, is written orderly to an append-only log. This becomes the backbone for all data systems in the organization similar to using event-sourcing in a distributed system but on a much bigger scale. Each system has the flexibility to derive different data structures 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.

Processing streams of events on an organizational scale is 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.

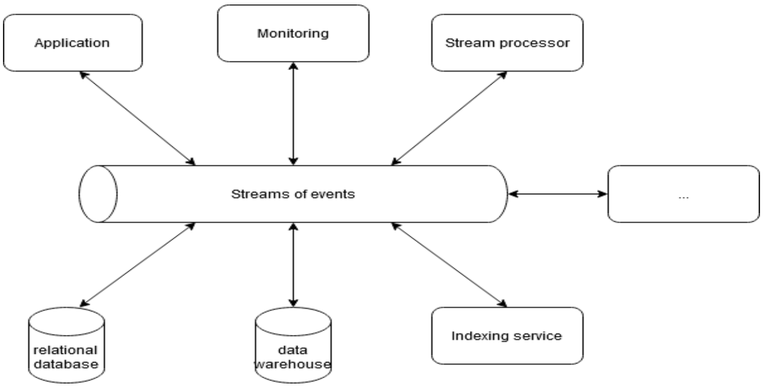

Figure 2.6: System with streams of events as the single source of truth.

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 [20]. 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.

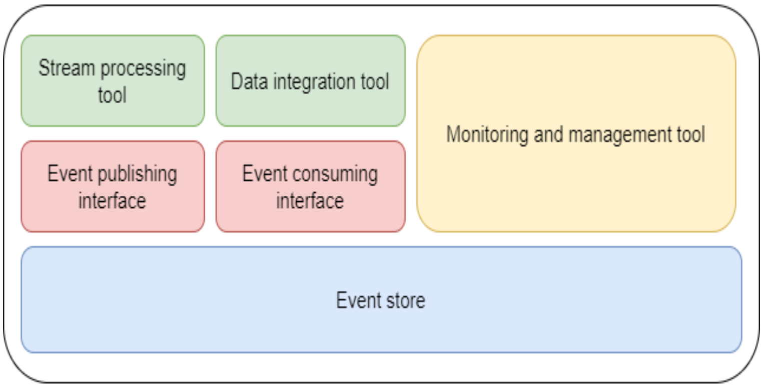
Stream processing tool
Data integration tool
Monitoring and management tool
Event publishing interface
Event consuming interface
Event store

Figure 2.7: Capabilities of an ESP platform.

All of these capabilities should be in a real-time, high throughput, scalable and highly reliable fashion so that the platform would not become the bottleneck in the infrastructure.

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 [22]. 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 [23]. The company Alibaba also joins the trend by open sourcing their RocketMQ to Apache Foundation [24]. 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 [25]. It is currently an incubating project of Cloud Native Computing Foundation. Pravega is also a quite new open-source project in the field [26]. 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 criteria indicating the maturity, the size of the active community, the popularity of the platform and the 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 active 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
Active community	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 in the community. 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 [27]. 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 [28]. Therefore, most common and related patterns of message delivery will be used as evaluating criteria for these platforms.

2.1: Publish-Subscribe [29]: 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 [30]: 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) [30]: 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 [13]. 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 [31]: 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. However, in distributed systems, order guarantee is very hard. 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: 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.

3.3: at-least-once delivery semantics: In case of failures, message can be resent until success 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 [32]. 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: Community size: for usability and learnability, the size of the active community is very important. A community of many users and active developers could be a great source of support. Moreover, apart from learnability, a platform with a big community can gain more contributions such as detecting issues, adding new features. This can help speed up 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 section, 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	Community size
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

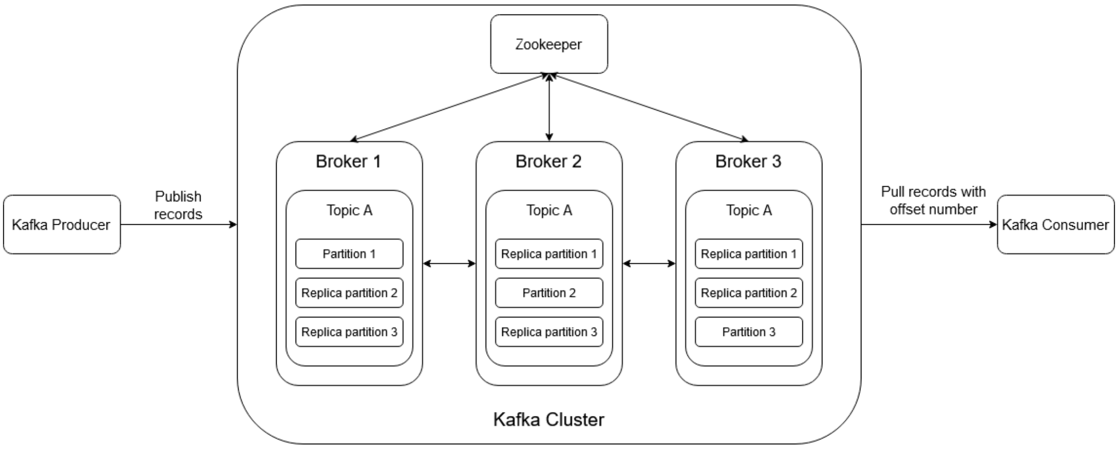

Figure 4.1: General concept of Apache Kafka.

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 [33] 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 [34].

- 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 [35]. 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

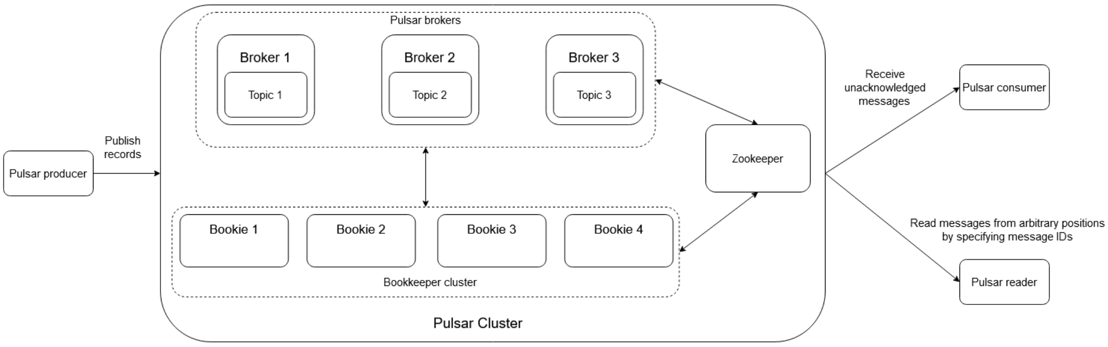

Figure 4.2: General concept of Apache Pulsar.

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 [36]: 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

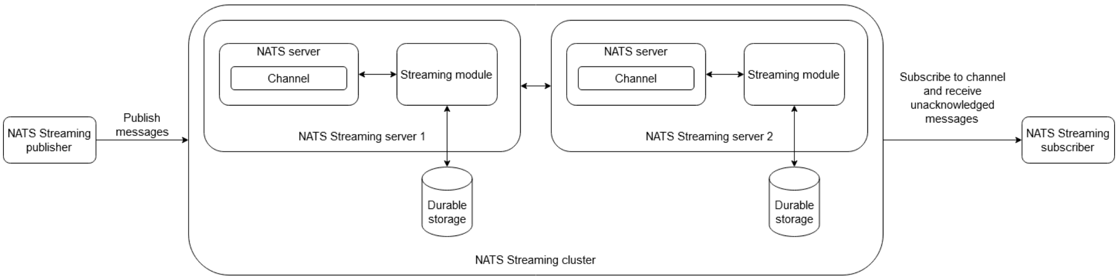

Figure 4.3: General concept of NATS Streaming.

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 streaming module to receive and persist message on NATS server to a pluggable durable storage. 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 [37]. 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 [38]. 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 [39]. 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.

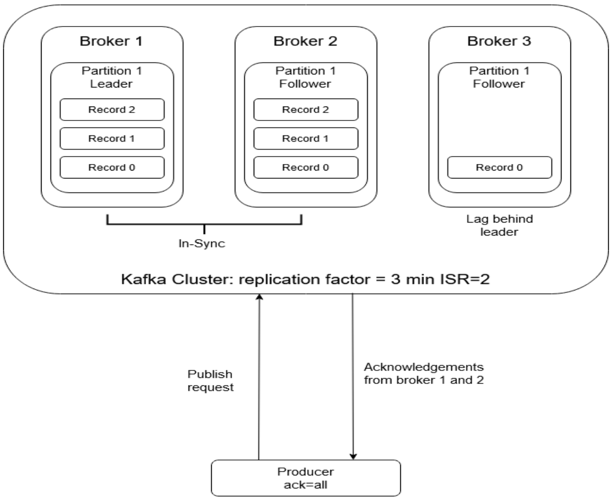

Figure 4.4: Data replication model for fault tolerance of event storage on Kafka.

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 acknowledgment. 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 acknowledgment 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 [40]. 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 but only if the order is not older than 3 months, it is reasonable to this combination of retention strategies.

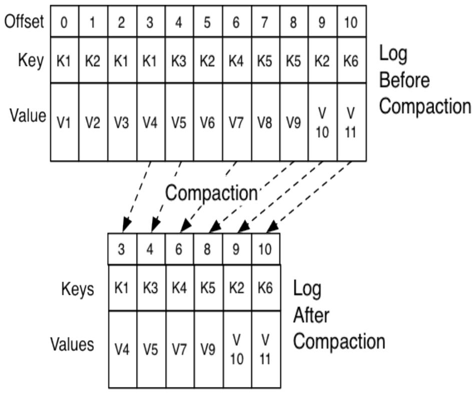

Figure 4.5: Log compaction on Kafka (taken from Kafka documentation [41]).

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 on the ledger are written and replicated on Bookkeeper cluster [42]:

- 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.

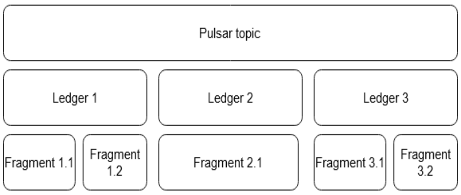

Figure 4.6: Underlying storage layers of a Pulsar topic.

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 [43], 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.

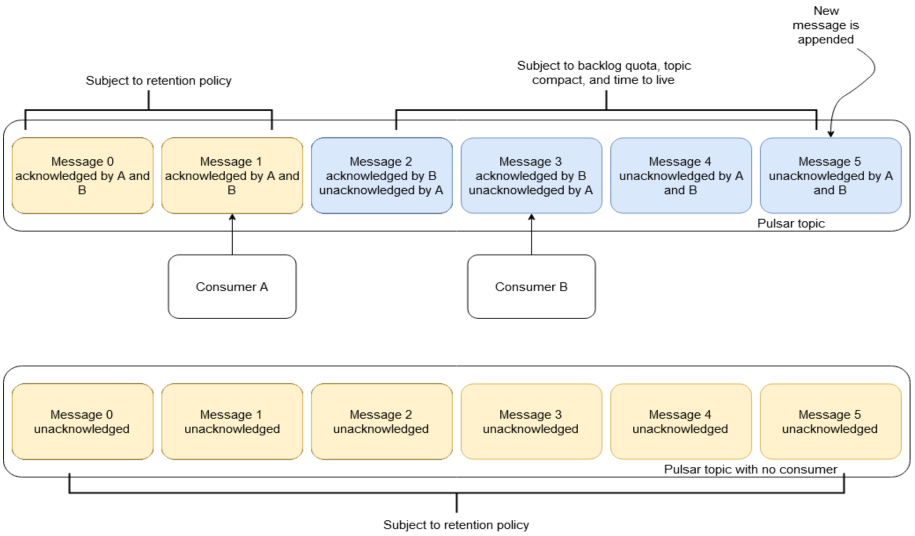
New message is appended
Subject to retention policy
Subject to backlog quota, topic compact, and time to live
Message 0 acknowledged by A and B
Message 1 acknowledged by A and B
Message 2 acknowledged by B unacknowledged by A
Message 3 acknowledged by B unacknowledged by A
Message 4 unacknowledged by A and B
Message 5 unacknowledged by A and B
Pulsar topic
Consumer A
Consumer B
Message 0 unacknowledged
Message 1 unacknowledged
Message 2 unacknowledged
Message 3 unacknowledged
Message 4 unacknowledged
Message 5 unacknowledged
Pulsar topic with no consumer
Subject to retention policy

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 [44]. 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 [45]. 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 [46]. 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. 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 (EFS).

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 [47]. 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.

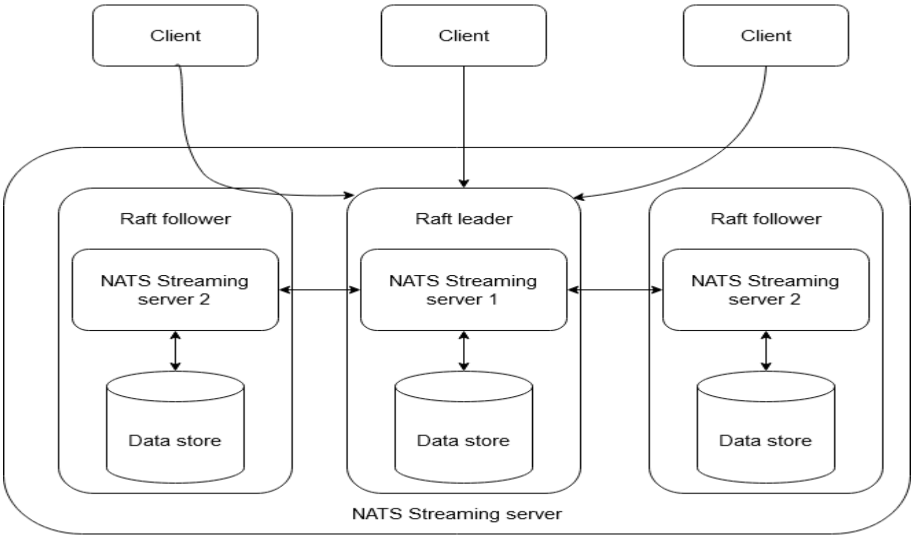

Figure 4.8: Fault tolerance of event storage on NATS Streaming in clustering mode.

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 [48].

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 [49]. 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 [49]. 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

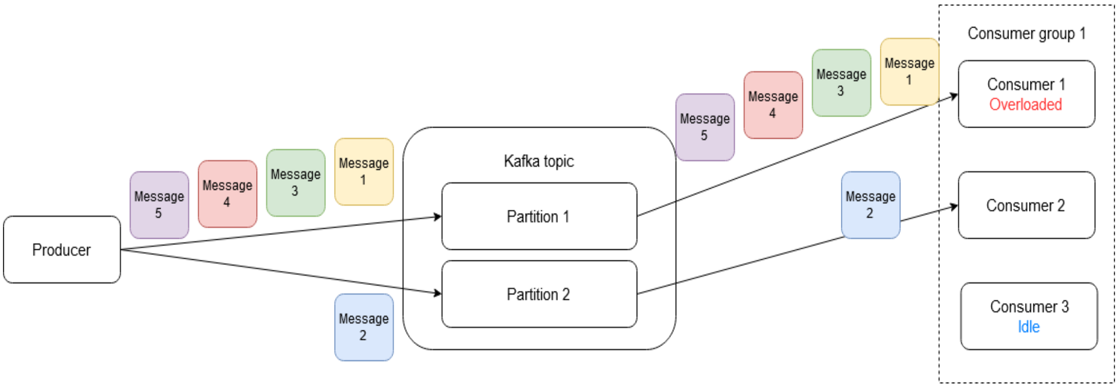

Figure 4.9: Competing consumers pattern with Kafka.

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 [50].

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.

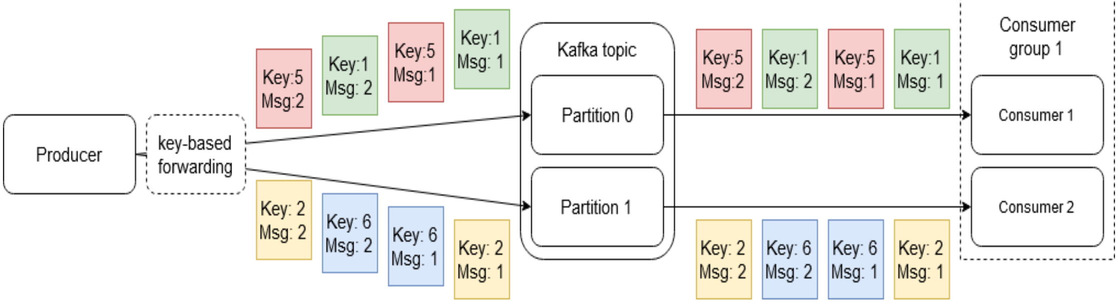

Figure 4.10: Consumer group pattern with Kafka.

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 [51].

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.

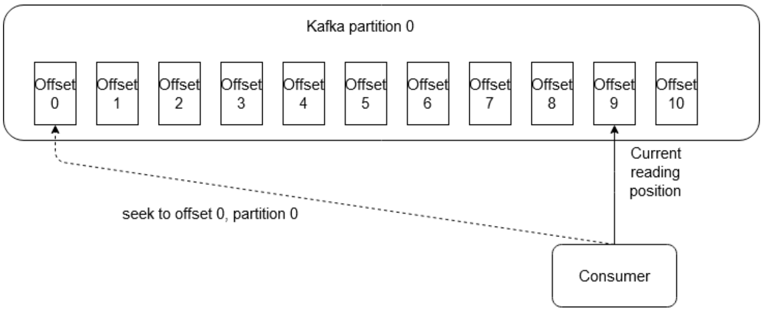

Figure 4.11: Event playback with Kafka.

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 [51]. 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 [52]. Acknowledgment 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 [53]. 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 [54]. 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 [55]. 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.

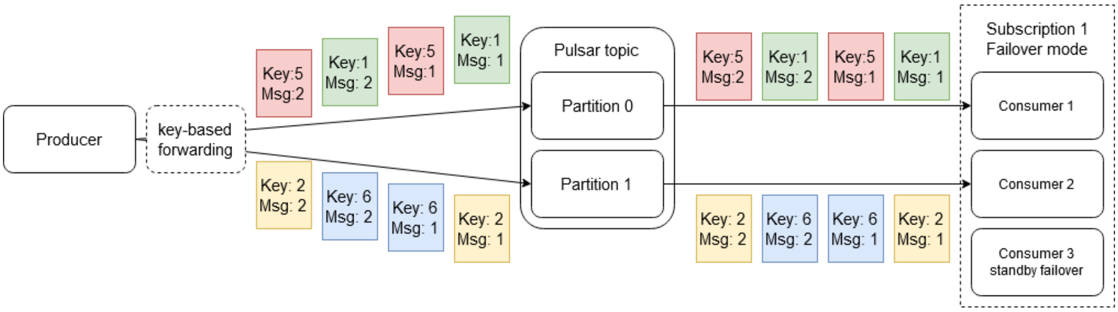

Figure 4.12: Consumer group pattern with Pulsar.

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 acknowledgment, 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 acknowledgments 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 [56]. 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 [57]. 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.

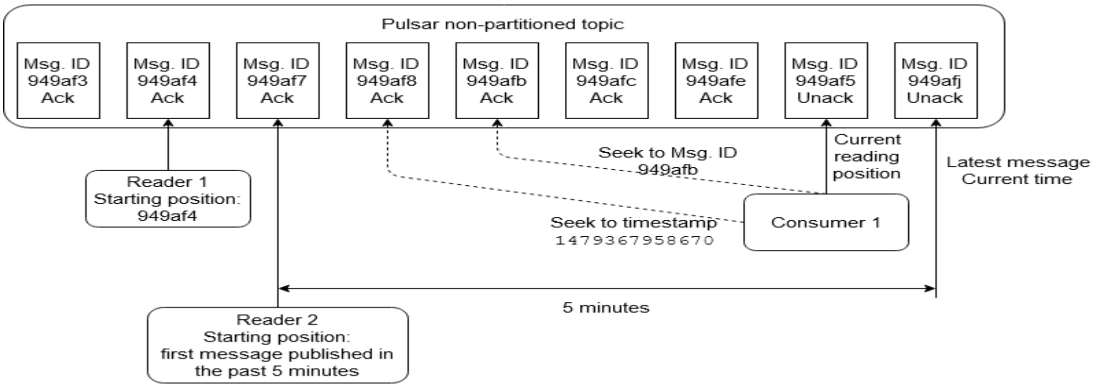

Figure 4.13: Event playback with Pulsar.

NATS Streaming

With NATS Streaming client API, a consumer must create a subscription to a channel to be able to start consuming messages [58]. 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 acknowledgments. 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 [59]. 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.

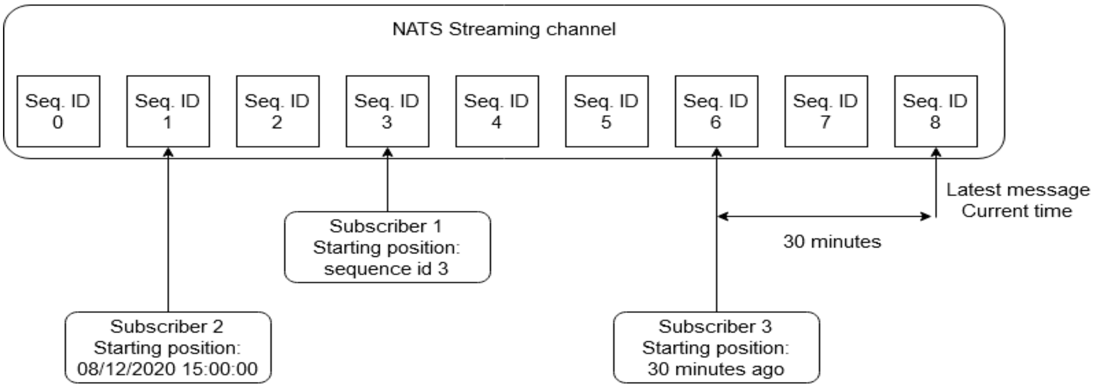

Figure 4.14: Event playback with NATS Streaming.

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

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 4.15).

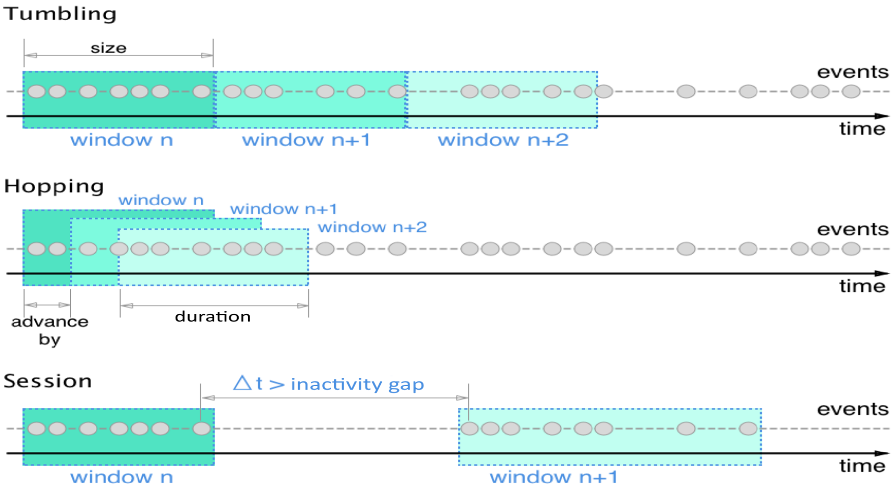

Figure 4.15: Supported window types for stream processing with Kafka Streams (taken from [61]).

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

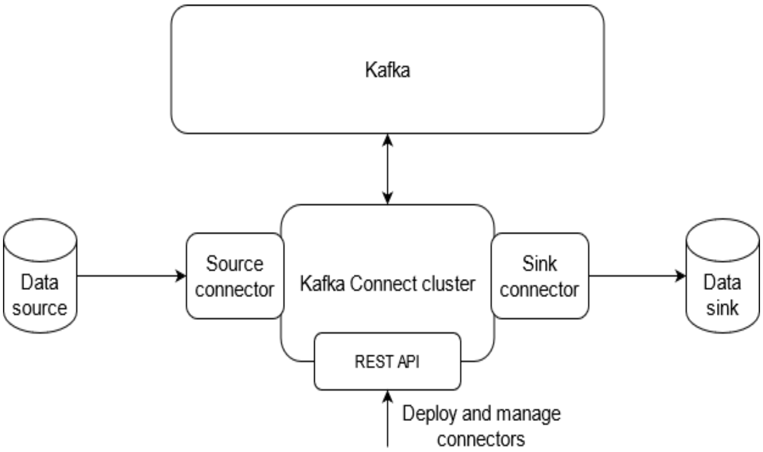

Figure 4.16: Kafka Connect cluster to integrate data between Kafka and external data systems.

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.

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 [70].

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 [71]. 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 [72]. 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 [72]. 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 [73]. Kafka brokers use Dropwizard Metrics [74] to collect statistics about their current statuses and expose this information to users using Java Management Extension (JMX) [75]. 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 [76] [77]. For instance, there are Confluent Control Center, Lenses, Cluster Manager for Apache Kafka (CMAK) tool. 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 [78]. 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 [79]. 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 [80]. 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 [81].

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.

Acronyms

CQRS Command Query Responsibility Segregation.

ESP Event Stream Processing.

List of Figures

List of Tables

List of Listings

Bibliography

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

[2] S. Intorruk and T. Numnonda, "A comparative study on performance and resource utilization of real-time distributed messaging systems for big data," in *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 102–107, IEEE, 2019.

[3] C. Bartholomew, "Performance comparison between apache pulsar and kafka: Latency." `https://medium.com/swlh/performance-comparison-between-apache-pulsar-and-kafka-latency-79fb0367f407`. Accessed: 2020-11-12.

[4] "Kafka vs. pulsar vs. rabbitmq: Performance, architecture, and features compared." `https://www.confluent.io/kafka-vs-pulsar/`. Accessed: 2020-11-12.

[5] 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.

[6] J. Lewis and M. Fowler, "Microservices." `https://martinfowler.com/articles/microservices.html`. Accessed: 2020-11-18.

[7] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 179–182, IEEE, 2016.

[8] 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.

[9] G. Young, "Cqrs documents." `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`, 2010. Accessed: 2020-11-23.

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

[11] B. Stopford, *Designing Event-Driven Systems Concepts and Patterns for Streaming Services with Apache Kafka*, ch. 8, pp. 79–81. O'Reilly Media, Inc., 2018.

[12] C. Richardson, "Pattern: Event-driven architecture." `https://microservices.io/patterns/data/event-driven-architecture.html`. Accessed: 2020-11-22.

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

[14] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*, ch. Reference 3: Introducing Event Sourcing. Microsoft patterns & practices, 2013.

[15] H. Rocha, "What they don't tell you about event sourcing." `https://medium.com/@hugo.oliveira.rocha/what-they-dont-tell-you-about-event-sourcing-6afc23c69e9a`, 2018. Accessed: 2020-11-23.

[16] 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.

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

[18] N. Marz, "How to beat the cap theorem." `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`. Accessed: 2020-11-18.

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

[20] 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.

[21] 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.

[22] `https://kafka.apache.org/`. Accessed: 2020-11-19.

[23] `https://pulsar.apache.org/`. Accessed: 2020-11-19.

[24] `https://rocketmq.apache.org/`. Accessed: 2020-11-19.

[25] `https://docs.nats.io/nats-streaming-concepts/intro`. Accessed: 2020-11-19.

[26] `https://pravega.io/`. Accessed: 2020-11-19.

[27] “Iso/iec 25010.” https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed: 2020-11-22.

[28] G. Hohpe and B. Woolf, “Enterprise integration patterns: Messaging patterns.” https://www.enterpriseintegrationpatterns.com/patterns/messaging/Messaging.html. Accessed: 2020-11-23.

[29] 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.

[30] G. Hohpe and B. Woolf, “Enterprise integration patterns: Messaging patterns: Competing consumers.” https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html. Accessed: 2020-11-23.

[31] 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.

[32] T. Treat, “You cannot have exactly-once delivery.” https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/. Accessed: 2020-11-24.

[33] “Apache zookeeper.” https://zookeeper.apache.org/. Accessed: 2020-11-26.

[34] “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.

[35] “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.

[36] “Apache bookkeeper.” https://bookkeeper.apache.org/. Accessed: 2020-11-26.

[37] “Kafka log.” https://kafka.apache.org/documentation/#log. Accessed: 2020-11-26.

[38] “Kafka flush messages.” https://kafka.apache.org/documentation/#flush.messages. Accessed: 2020-11-26.

[39] “Kafka replication.” https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication. Accessed: 2020-11-27.

[40] “Kafka replication.” https://kafka.apache.org/documentation/#cleanup.policy. Accessed: 2020-11-27.

[41] “Kafka documentation.” https://kafka.apache.org/documentation/. Accessed: 2020-11-26.

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

[43] "Using autorecovery." http://bookkeeper.apache.org/docs/4.11.1/admin/autorecovery/. Accessed: 2020-11-29.

[44] "Message retention and expiry." https://pulsar.apache.org/docs/en/cookbooks-retention-expiry/. Accessed: 2020-11-30.

[45] "Topic compaction." https://pulsar.apache.org/docs/en/concepts-topic-compaction/. Accessed: 2020-11-30.

[46] "Nats streaming: Persistence." https://docs.nats.io/nats-streaming-server/configuring/persistence. Accessed: 2020-12-01.

[47] 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.

[48] "Nats streaming: Store limits." https://docs.nats.io/nats-streaming-server/configuring/storelimits. Accessed: 2020-12-01.

[49] "Design: The consumer." http://kafka.apache.org/documentation.html#theconsumer. Accessed: 2020-12-02.

[50] 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.

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

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

[53] "Pulsar messaging: Subscriptions." https://pulsar.apache.org/docs/en/concepts-messaging/#subscriptions. Accessed: 2020-12-04.

[54] "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.

[55] "Pulsar clients: Reader interface." https://pulsar.apache.org/docs/en/concepts-clients/#reader-interface. Accessed: 2020-12-06.

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

[57] “Javadoc: Pulsar reader.” https://pulsar.apache.org/api/client/2.6.0-SNAPSHOT/org/apache/pulsar/client/api/ReaderBuilder.html. Accessed: 2020-12-07.

[58] “Nats streaming: Subscriptions.” https://docs.nats.io/nats-streaming-concepts/channels/subscriptions. Accessed: 2020-12-07.

[59] “Nats streaming: Receiving messages from a channel.” https://docs.nats.io/developing-with-nats-streaming/receiving. Accessed: 2020-12-07.

[60] “Kafka streams: Core concepts.” https://kafka.apache.org/26/documentation/streams/core-concepts. Accessed: 2020-12-09.

[61] “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] “Pulsar functions overview.” https://pulsar.apache.org/docs/en/functions-overview/. Accessed: 2020-12-09.

[64] “Github issue: Unable to create pulsar function.” https://github.com/apache/pulsar/issues/8469. Accessed: 2020-12-09.

[65] “Kafka connect.” https://kafka.apache.org/documentation.html#connect. Accessed: 2020-12-10.

[66] “Kip-318: Make kafka connect source idempotent.” https://issues.apache.org/jira/browse/KAFKA-6080. Accessed: 2020-12-10.

[67] “Open issue: Transactional eos for source connectors.” https://docs.confluent.io/5.5.0/connect/kafka-connect-hdfs/index.html#features. Accessed: 2020-12-10.

[68] “Kafka: Clients.” https://cwiki.apache.org/confluence/display/KAFKA/Clients. Accessed: 2020-12-10.

[69] “Pulsar connector overview.” https://pulsar.apache.org/docs/en/io-overview/. Accessed: 2020-12-10.

[70] 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.

[71] “Pulsar client libraries.” https://pulsar.apache.org/docs/en/client-libraries/. Accessed: 2020-12-10.

[72] "Nats streaming: clients and utilities." https://nats.io/download. Accessed: 2020-12-10.

[73] "Kafka operations: Monitoring." https://kafka.apache.org/documentation/#monitoring. Accessed: 2020-12-12.

[74] "Dropwizard metrics v4.1.2." https://metrics.dropwizard.io/4.1.2/. Accessed: 2020-12-12.

[75] "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.

[76] 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.

[77] G. Shilo, "Kafka administration and monitoring ui tools." https://dzone.com/articles/kafka-administration-and-monitoring-ui-tools. Accessed: 2020-12-12.

[78] "Pulsar metrics." https://pulsar.apache.org/docs/en/reference-metrics/. Accessed: 2020-12-15.

[79] "Pulsar manager." https://pulsar.apache.org/docs/en/administration-pulsar-manager/. Accessed: 2020-12-15.

[80] "Nats streaming: Monitoring." https://docs.nats.io/nats-streaming-concepts/monitoring. Accessed: 2020-12-15.

[81] "Nats: Monitoring." https://docs.nats.io/nats-server/configuration/monitoring. Accessed: 2020-12-15.