Increasing Resiliency in Kafka Clients

Implementation of Write-Ahead Log, Consumer Pause Capability, and a Remote Control Center

Abstract

This white paper proposes an approach to enhance the resiliency of applications using Kafka clients, concentrating on the development of a reusable Kafka producer and consumer library extending beyond the current capabilities of Spring Kafka.

Firstly, we introduce a Write-Ahead Log (WAL) into the producer library, a feature designed to increase durability and minimise data loss during system crashes. This novel application of the WAL simplifies the traditionally complex process of implementing the outbox pattern, with the library itself safeguarding against potential message loss.

Furthermore, we establish an on-demand pause and resume functionality for Kafka consumer operations, enabling greater flexibility and system control. This is complemented by the development of a Remote Control Center Application, through which applications using the consumer library can register their consumer groups. Authenticated users can then remotely control the status of consumers per application, enhancing overall system adaptability.

The consumer library uses a Global KTable implementation provided by Kafka Streams to facilitate the remote broadcasting of start/stop commands, effectively controlling consumer containers as needed. The performance, fault tolerance, and recovery times of this comprehensive system are rigorously tested through extensive simulations.

The expected findings indicate a substantial enhancement in system resilience, with minimised risk of data loss and optimised system uptime. By embedding the WAL into the library, we alleviate the necessity for client-side outbox pattern implementation, thus offering a more efficient and robust solution.

Problem Statement

When using tradition Spring or Native Kafka clients to produce and consume messages and employing conventional design patterns there are some issues including the 2 listed below. This white paper describes patterns and solutions to these problems.

1. While consuming message from Kafka the standard error handling process is to retry the message ’n' of times and once that is exhausted push the message to an error queue (dead letter queue) for further handling. But in cases of extended outages like infrastructure issues with extremely high volume systems., we may end up pushing a lot of messages into the error queues after exhausting retries. This will overwhelm the error queue and subsequent processing. This paper proposes a different approach to this problem by giving user the ability to pause the Kafka consumer either programatically or on-demand via a remote control centre so that we can temporarily pause message flow through the system till it heals.
2. While producing messages from a Kafka producer client - an often overlooked fact is the potential for message loss when the network connectivity between broker and client is compromised. This often leads to message loss in the most unexpected manner. One out of the box solution to handle this is to use a sync producer that will wait for broker acknowledgment for each message sent. However this is not a performance solution. This paper proposes a novel solution that will allow the Kafka producer client to sent a message reliably by using the write ahead log pattern to ensure ZERO message loss.

Proposed Solution:

The proposed solution is broken down into 2 separate sections. 1 section will describe the solution for the Kafka consumer problem while the other section will describe the solution for the Kafka producer issues listed above.

Kafka Consumer Solutions:

This white paper proposes a solution to the above mentioned problem statement by providing the client with a re-usable Kafka library on top of what the Spring Kafka framework currently provides. This library will provide support for the following -

1. Inbuilt logic to register the Kafka consumer into a topic that is then materialised as a Global KTable.
2. Inbuilt scheduled task that queries this Global KTable for any changes to the state of the listener container. If the listener container state changes in the store the scheduled task will accordingly pause / resume the Kafka listeners in the library.
3. A remote control centre UI that will provide insight into the available consumer groups across applications using the library and current status of the consumers. The UI will also provide the ability to start / stop these consumer groups on demand.

Kafka Consumer Solution Implementation (Method):

Overall consumer library features are described below. In addition to providing the feature set mentioned above - it will also standardise client configurations by providing an opinionated view of the configurations. The client using the library can override this if required. Another value add is that it also sources client consumer side metrics and feeds this information into a reporting system like Prometheus/Grafana, App D etc. for better visibility. This is a significant value addition the clients as an extension to existing Kafka monitoring tools like the Kafka / Confluent control centre.

The diagram shown below- explicitly showcases how the remote pause / start feature works in an application that is using this library. The steps are listed in detail below.

1. During application start up the library will register all consumers into a Kafka topic. They will simply publish data as a key, value pair where the key is the consumer group name / id and the value is indicator of whether it is currently PAUSED / ACTIVE. This is shown as step (1) in the diagram.
2. Once this data is on the topic - the control application will read and materialise this data as a Global KTable. This is indicated by step (2) in the diagram. This data can they be queried and shown on the UI. The user with the right access to this control UI can now see the available consumer groups and the current state of the consumer group.
3. Now if there is an issue - or an extended outage and the user wants to pause the Kafka consumer., he/she can simply send a command to pause the consumer via UI. This command will be sent as a message on the same Kafka topic in the same key, value format. The consumer group name with state now to be set as PAUSED. This is indicated as step (3) on the diagram.
4. Now the client application library has an inbuilt poller / scheduled task that will scan for these kind of changes on the topic. Once it sees the change state instruction for that consumer group it will read and process the message. If the user request is to PAUSE the consumer group - this library will access the spring Kafka listener container and set it to pause. This is indicated as step(4) on the diagram.
5. In this manner, the user has complete control over the Kafka consumers of their application. Each consumer can be configured to register with a unique application ID during startup so that is can be identified in the control UI and also be targeted uniquely for start / stop during extended production issues.

Kafka Producer Solutions:

This white paper proposes a solution to the Kafka producer message loss case by providing a re-usable client producer library on top of what Spring Kafka currently provides. This library will provide an implementation of the write ahead log or outbox pattern as part of standard functionality so that client need not worry about implementing this by themself. They can get the added benefit of 100% message delivery guarantee by using the feature while not sacrificing performance as well.

Problem in detail - when the Kafka producer writes a message to a Kafka topic in an async manner., the client thread is acknowledged immediately but the message is not actually sent to the Kafka broker. It is just put into a client buffer. A separate thread will wait for the buffer to fill up and then flush to broker as one network call. This is how Kafka achieves high degree of performance. However - since the send and produce happen in 2 different threads., if the producer crashes between send and produce then the messages in the buffer will be lost permanently even though they are successfully acknowledged to the client. The client thinks the write to Kafka was successful when in fact the message is lost. This is a catastrophic message loss scenario. One way to avoid this is to use a blocking sync send call from the producer client to the Kafka broker. But this is a blocking call and will use network resources for every message being produced which will kill the performance that Kafka provides.

This white paper solution prescribes a solution by using a write ahead log for recovery of such lost messages to allow the client to send messages in an async manner while still ensuring high performance that Kafka provides due to batching.

Kafka Producer Solution Implementation (Method):

Overall producer library features are described below. In addition to providing the feature set mentioned above - it will also standardise client configurations by providing an opinionated view of the configurations. The client using the library can override this if required.

The diagram shown below shows explicitly how the producer write ahead log (WAL) feature works. See below:

1. A client using the producer library must enable this feature via configuration and provide a persistent store - this can be a database URL where the application has permissions to create a table or a file store where the application can create a file.
2. When the client application sends a message to Kafka via this producer library - the message will be capture and written to the database or the file first. This is shown as step (1) in the diagram below. This is just a persistent record of the message.
3. Now the library will proceed to use the standard Spring Kafka template to send the message to Kafka in an async manner as indicated by step (2) in the diagram.
4. The library also implements an async callback producer listener that will wait for the actual acknowledgement from the Kafka cluster. This is indicated as step (3) in the diagram.
5. Once this acknowledgement is received, the client library async producer listener implementation will update the record in the write ahead log (database or file) as success or error based on the response from Kafka broker. If no response is received from Kafka broker or response is lost nothing is updated in the write ahead log. This is indicated as step (4) in the diagram.
6. Now the final response after the WAL has been updated is propagated to the client async producer listener.
7. How does this help with message loss - assume that a producer crash happens during the buffer stage, where the message has been sent to client buffer but not yet flushed to Kafka broker. In this case, when the application restarts., the producer library will scan the WAL for data that has not received any response from the Kafka broker and will proceed to rewrite them to Kafka broker. This might potentially result in a duplicate message being sent to the Kafka Broker but that in any case is better than losing a message altogether.
8. Another important aspect to consider is that the WAL is a significantly complex implementation - in a distributed system., it is possible that several instances of the producer library will crash and restart at the same time in which case the way the WAL is processed has to be carefully managed. The distributed instances must not process the same messages and each instance of the producer library must uniquely handle each message that is still pending in the WAL.

Conclusion:

The intent of the solution in this white paper is to take on what spring framework has built to integrate with Kafka and add some more features that will make this library even more resilient. It also hopes to integrate some design patterns that will be of signifiant value add in real production outage scenarios where it is better to just pause the Kafka pipelines rather than keep on trying and failing. It also hopes to address most common developer misunderstanding of Kafka configurations by taking an opinionated stance and also providing implementation for resiliency patterns that would be too complex for client application to build by themselves like the write ahead log and the control centre for metrics and control of Kafka consumers.