BEST PRACTICES FOR EXCEPTION HANDLING IN APACHE KAFKA

Apache Kafka is a distributed streaming platform that enables the processing of large amounts of data in real time. It is designed to handle high-volume, high-velocity data streams, and provides fault tolerance, scalability, and durability.

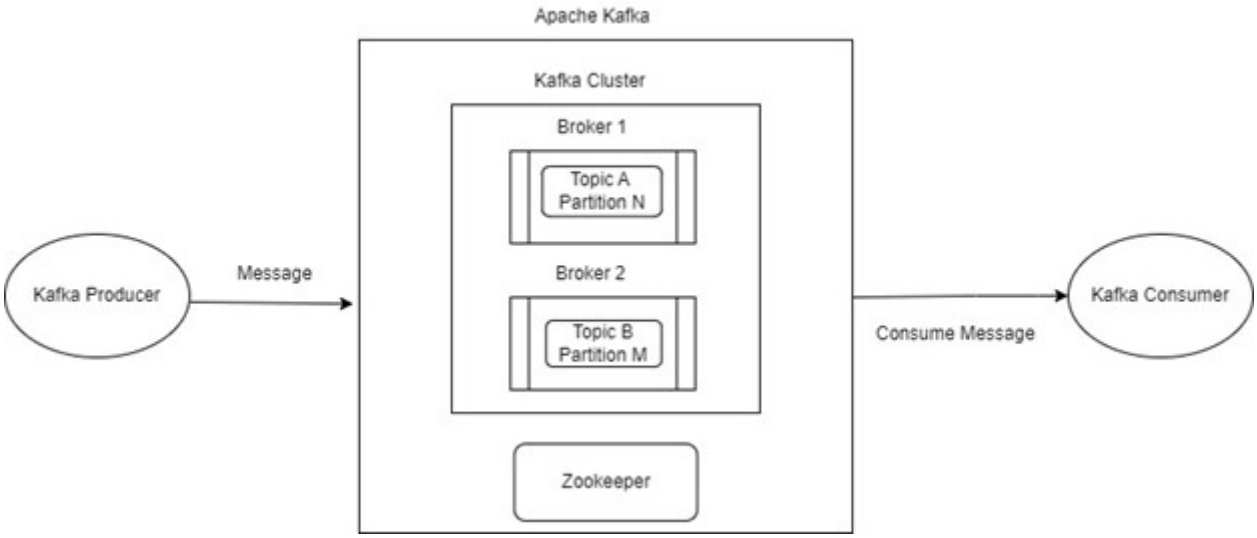


Figure 1. Apache Kafka architecture.

It provides two main client libraries for data processing: Kafka Consumer and Kafka Streams.

The default, **Kafka Consumer**, is a client library that allows users to read data from Kafka topics. It is a poll-based system, which means that consumers must request data from Kafka topics by polling for new messages. Once a message is consumed, the consumer can process it according to the required business logic.

On the other hand, **Kafka Streams** is a client library that provides stream processing capabilities. It allows users to read data from Kafka topics, transform the data, and write it back to Kafka topics or external systems. Kafka Streams is built on top of the Kafka Consumer library and provides additional functionalities like stateful processing, windowing, and aggregation. It is a stateful processing engine that allows users to perform complex data transformations on continuous data streams in real time.

Kafka is a powerful tool for processing and analyzing streaming data, but like any distributed system, it can encounter errors and exceptions. Proper exception handling is crucial for maintaining the reliability and fault tolerance of your Kafka Streams application. This blog post covers different patterns and best practices for handling errors and exceptions in your event streaming applications.

Kafka error scenarios

Broadly speaking, we face two kinds of errors while processing the data streams: transient errors and non-transient errors.

A **transient error** is an error that occurs once or at unpredictable intervals. Examples of such errors are a temporary network glitch or temporarily unavailable end points. An obvious solution in such a scenario is to retry the processing.

Non-transient errors are more persistent in nature, such as when a Kafka message fails with the same error again and again, no matter how many times you retry. Examples of such errors are bugs in application processing logic and parsing errors. You need to provide a mechanism to gracefully handle these errors when they occur, so you can recover automatically when possible or shut down when it's truly unrecoverable.

Using Kafka offset commit management to manage errors

To effectively handle Kafka error scenarios, we need to understand the mechanism of offset commit management. In Kafka, consumers can commit their current position in the partition to Kafka brokers to ensure that they have consumed all the messages up to a certain point in the partition. This process of committing the offset can be done either automatically or manually, and there are trade-offs associated with each approach.

By default, the consumer is configured to auto-commit offsets. Using auto-commit gives you "at least once" delivery: Kafka guarantees that no message will be missed, but duplicates are possible. Auto-commit works as a cron with a period set through the *auto.commit.interval.ms* configuration property. If the consumer crashes, then, after a restart or a rebalance, the position of all partitions owned by the crashed consumer will be reset to the last committed offset. When this happens, the last committed position may be as old as the auto-commit interval itself. Any messages that have arrived since the last commit will have to be read again.

Clearly, if you want to reduce the window for duplicates, you can reduce the auto-commit interval, but some users may want even finer control over offsets. The consumer therefore supports a

commit API, which gives you full control over offsets. Note that when you use the commit API directly, you should first disable auto-commit in the configuration by setting the *enable.auto.commit* property to false.

The Kafka Streams client commits offsets based on *commit.interval.ms* configs (default is 30 seconds). So, even if you request a commit, commits happen regularly. In general, it's sufficient to rely on Kafka Streams' implicit commits (requesting commits explicitly is not necessary for most applications). The Kafka Streams client will always disable or turn off auto-committing.

Noisy Neighbor issue

This is the major problem that one encounters when there are errors/exceptions in processing Kafka messages in a multi-tenancy environment. When one or more Runtime exceptions are not handled, they bubble up all the way to the Kafka (Streams) thread, causing Kafka Streams to crash. In such a scenario, since an offset is not committed, Kafka may end up re-reading the same messages again and again, continuing to fail with the same exceptions, and crashing. This is a "Stop the World" (STW) scenario. An important point to note here in a **multi-tenancy** environment: This may cause a "**Noisy Neighbor**" issue by blocking the entire stream.

For example, if the consumer client happens to read data from a partition that is not in an expected format, then it can throw an error and choose not to commit the offset. In such a scenario, the offset is not committed, and the consumer ends up reading the same message again and again, which triggers continuous failures ad infinitum. In a multi-tenant environment, this can cause Noisy Neighbor issues because partitions have data from more than one tenant. Bad data from one tenant can impact the processing of data from other tenants.

Three best practices for Kafka error handling

To avoid STW and Noisy Neighbor issues, there are some design approaches one can use.

1. **Error log and discard the message**: To avoid STW scenarios, many developers catch the generic exception and error log it. The problem with this approach is that even though it avoids the STW scenario, it results in data loss because we have dropped that message and committed the offset at the same time. This may result in functional issues. This is definitely not a suitable approach for exceptions caused by transient errors. Also, as exceptions are caught in the background, errors may go unnoticed.
2. **Dead letter topic**: This approach is more suitable for non-transient errors. In scenarios where some of the messages cannot be processed by applications, they are routed to error topics and the main stream continues. We need to have monitoring and alerting in place for error topics to resolve the issue later. This will avoid a Noisy Neighbor issue and, at the same time, errors and exceptions won't go unnoticed.

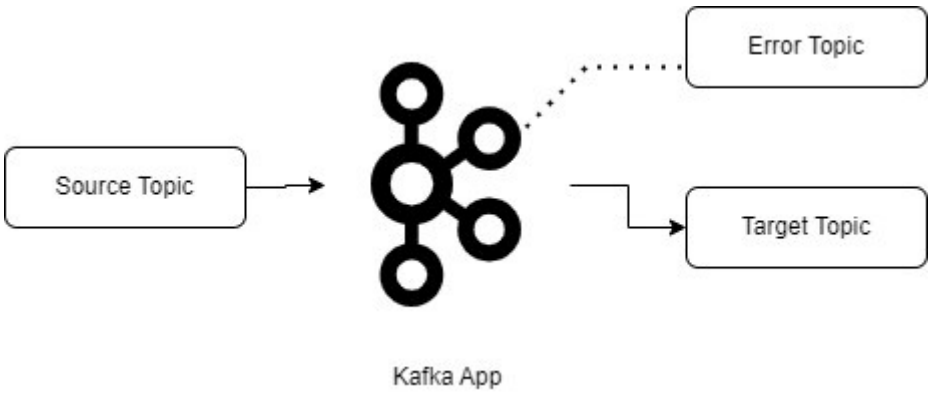
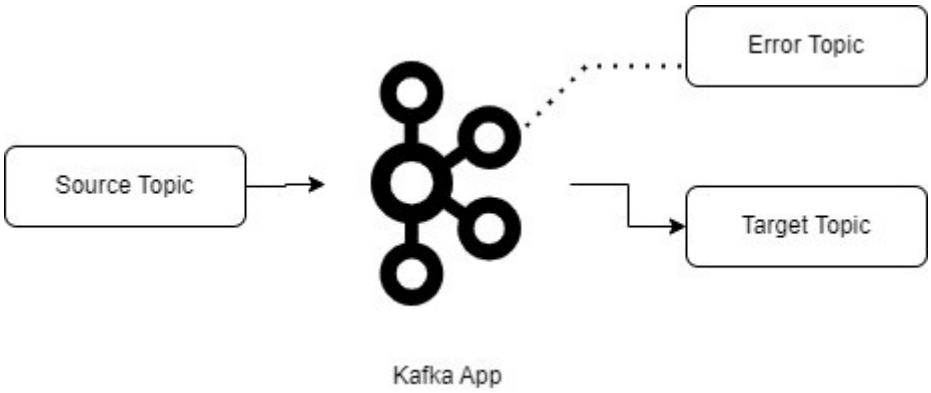


Figure 2. Dead letter topic.

3. **Retry topic and retry consumer**: This approach is more suitable for transient errors. In generic terms, we can say the conditions required to process the message are not available when the application tries to process the event. Adding a retry topic provides the ability to process most messages right away, while delaying the processing of other messages until the required conditions are met. This approach has some challenges: Related messages may get processed out of order and duplicate processing of messages may occur (fortunately, Kafka Streams has a mechanism to handle duplicate events).



3. Retry topic and retry consumer.

Kafka Streams error handling mechanisms

Kafka Streams defines three main categories where errors may occur:

1. Consuming records
2. Processing records
3. Producing records

Kafka Streams provides error handlers for each of these categories. In some cases, the best approach is to acknowledge and continue; other times, it is more prudent to shut down.

Consuming records (entry level)

Kafka Streams provides this entry level exception handler: **DeserializationExceptionHandler**.

This error handler allows you to manage any messages that fail to deserialize, which can be caused by corrupt data, incorrect serialization logic, or unhandled message types.

The implemented exception handler needs to return a FAIL or CONTINUE depending on the message and the exception thrown. Returning FAIL will signal that Kafka Streams should shut down, and returning CONTINUE will signal that it should ignore the issue and continue processing.

Kafka Streams provides built-in handlers for the same. The default configuration for this handler is **LogAndFailExceptionHandler**. This exception handler will log the error and shut down Kafka Streams until the user chooses to react to it or resume it.

Another option is to use **LogAndContinueExceptionHandler**. This exception handler will continue processing the next records instead of shutting down the entire stream.

Add this error handler through Streams config:

```
streamsConfiguration.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,SendToDeadLetterQueueExceptionHandler.class);
```

Let us understand the custom error handler implementation along with dead letter topic design. Here is the code snippet that implements the **DeserializationExceptionHandler** and the override handle method. This class will collect the messages for which a deserialize exception has occurred. We will produce these messages in dead letter topic, which will have monitoring and alerting in place. Kafka Streams will continue processing the next records, which will prevent the Noisy Neighbor issues as well.

Figure 4. DeserializationExceptionHandler.

Producing records

ProductionExceptionHandler can be used in a scenario where errors occur while trying to produce records from Kafka Streams to Kafka Broker.

This is very similar to **DeserializationExceptionHandler**, where a developer can choose to log and continue processing or shut down Kafka Streams. This only applies to exceptions that are not handled by Kafka Streams, such as **RecordsTooLargeException**. Here again, one needs to implement the **ProductionExceptionHandler** interface and override the handle method. You also need to add this error handler through Kafka Streams configuration:

```
streamsConfiguration.put(StreamsConfig.DEFAULT_PRODUCTION_EXCEPTION_HANDLER_CLASS_CONFIG, StreamProducerExceptionHandler.class);
```

Processing records

Kafka Streams provides this message-processing-level exception handler: **StreamsUncaughtExceptionHandler**.

This works for exceptions that are not handled by Kafka Streams. The processing phase can have many types of errors, such as application bugs, transient network issues, database-related errors, and so on. In these instances, developers must decide under which circumstances they need to continue processing and in which circumstances they need to stop the entire stream.

You can provide an error handler for the message processing phase by implementing the **StreamsUncaughtExceptionHandler** interface and overriding the handle method. The handle method has three options to respond in case of error situations:

1. **Replace the stream thread**: This is a type of retry mechanism for transient errors, where Kafka Streams kills the current stream thread and spawns a new stream thread, which again consumes records from the last committed offset. This reprocesses the same record with the hope of not seeing the transient error again. This can result in duplicate records depending on the application's processing mode determined by the **PROCESSING_GUARANTEE_CONFIG**
2. **Shut down the individual stream instance**: This shutsdown the individual consumer thread of the Kafka Streams application experiencing the exception.
3. **Shut down all stream instances**: This shuts down all instances of a Kafka Streams application *with the same application-id*. Kafka Streams uses a rebalance to instruct all application instances to shut down, so even those running on another machine will receive the signal and exit.

Below is a code snippet for the **StreamsUncaughtExceptionHandler** interface. This implementation of the **StreamsUncaughtExceptionHandler** will keep track of the number of errors that occur within a given timeframe. If processing errors occur within the expected timeframe, then these messages are sent to dead letter topic and the error is logged. If the number of errors exceeds the threshold within the provided timeframe, then the entire application will shut down.

```
private StreamThreadExceptionResponse logThreadCrashException(Throwable exception,int currentFailureCount,Instant previousErrorTime) {
  KafkaProducer<byte[], byte[]> dlqProducer;
  currentFailureCount++;
  Instant currentErrorTime = Instant.now();
  if (previousErrorTime == null) {
    previousErrorTime = currentErrorTime;
  }
  long millisBetweenFailure = ChronoUnit.MILLIS.between(previousErrorTime, currentErrorTime);
  if (currentFailureCount >= maxFailures) {
    if (millisBetweenFailure <= maxTimeIntervalMillis) {
      return StreamThreadExceptionResponse.SHUTDOWN_APPLICATION;
    } else {
      currentFailureCount = 0;
      previousErrorTime = null;
    }
  }else{
    log.warn("Exception caught during Deserialization, sending to the dead queue topic; " +
                  "taskId: {}, topic: {}, partition: {}, offset: {}",
          context.taskId(), record.topic(), record.partition(), record.offset(),
          exception);
    dlqProducer.send(new ProducerRecord<>(deadLetterTopic, record.timestamp(), record.key(), record.value(), record.headers())).get();
  }
}
```

Figure 5. StreamsUncaughtExceptionHandler.

Effective error handling in Kafka applications involves practices such as using error topics, implementing error logging, handling errors gracefully, utilizing retry mechanisms, establishing monitoring systems, conducting thorough testing, and ensuring proper serialization. These practices

enhance the reliability and fault tolerance of Kafka applications. It's also important to test your exception handling code to ensure that it is working as expected. This can be done by writing unit tests that simulate different exception scenarios and verify that the appropriate actions are taken.

By implementing effective exception handling strategies, Kafka developers can enhance the fault tolerance, scalability, and overall reliability of their applications. Being prepared for potential failures and gracefully handling exceptions will help ensure smooth data processing and deliver a seamless streaming experience with Kafka.