KIP-DRAFT: Error Handling in Connect

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *"Under Discussion"*

Discussion thread: TBD

JIRA: KAFKA-6738

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

There are several places in Connect during which failures may occur. Any failure to deserialize, convert, process, or read/write a record in Kafka Connect can cause a task to fail. Although some errors can be addressed with SMTs or custom converters that check for malformed data, in general it is difficult to ensure correct and valid data or to tell Connect to skip problematic records.

Connect should allow users to configure how bad data should be handled during all phases of processing records. Some failures such as the lack of availability of some external components may be resolved by simply retrying, while other errors should be recorded and the problem records simply skipped. Where possible, Connect should be able to record the error and optionally include the problematic records and configuration states of the connector and transform. And since no single solution works for everyone, all of this error handling behavior should be configurable.

This proposal aims to change the Connect framework to allow it to automatically deal with errors while processing records in a Connector. By default, Connect will fail immediately when an error occurs, which is the previous Connect behavior. Therefore, all new behaviors must be explicitly enabled.

Public Interfaces

Several new behaviors for handling and reporting errors are introduced, and all must be configured in the individual connector configurations.

Retry on Failure

Retry the failed operation a configurable number of times, with backoff between each retry. The number of retries and backoff can be configured using the following new properties:

Configuration Name	Description	Default Value	Domain
errors.retries.limit	The maximum number of retries before failing.	0	[-1, 1, ... Long.MAX_VALUE], where -1 means infinite retries.
errors.retries.delay.max.ms	The maximum duration between two consecutive retries (in milliseconds).	60000	[1, ... Long.MAX_VALUE]

Task Tolerance Limits

Tolerate up to a configurable number of failures in a task. If the task fails to successfully perform an operation on a record within tolerance limit, the record is skipped. Once the tolerance limit (overall or rate) is reached, the task will fail. Tolerance limits can be configured using the following new properties:

Config Option	Description	Default Value	Domain

errors.tolerance.limit	Fail the task if we exceed specified number of errors overall.	-1	[-1, 0, 1, ... Long.MAX_VALUE], where a value of -1 means infinite failures will be tolerated.
errors.tolerance.rate.limit	Fail the task if we exceed specified number of errors in the observed duration.	-1	[-1, 0, 1, ... Long.MAX_VALUE], where a value of -1 means infinite failures will be tolerated in the observed window.
errors.tolerance.rate.duration	The duration of the window for which we will monitor errors.	minute	minute, hour, day

Log Error Context

The error context and processing information can be logged along with the standard application logs using the following configuration properties:

Config Option	Description	Default Value	Domain
errors.log.enable	Log the error context along with the other application logs	false	Boolean
errors.log.include.configs	Include the (worker, connector) configs in the log.	false	Boolean
errors.log.include.messages	Include the Connect Record which failed to process in the log.	false	Boolean

Produce Error Context to a Dead Letter Queue (Kafka Topic)

Produce a message which contains the processing context and error information to a Kafka topic. By default, the worker properties will be used to locate the Kafka cluster. This can be overridden using the `errors.dlq.producer.*` configs as shown below:

Config Option	Description	Default Value	Domain
errors.dlq.enable	Write the error context along into a Kafka topic	false	Boolean
errors.dlq.topic.name	The name of the topic where these messages are written to	""	A valid Kafka topic name
errors.dlq.topic.partitions	Number of partitions for this topic	5	[1, 2 .. Integer.MAX_INT]
errors.dlq.topic.replication.factor	The replication factor for this topic	3	[1, 2 .. Integer.MAX_INT]
errors.dlq.include.configs	Include the (worker, connector) configs in the log.	false	Boolean
errors.dlq.include.messages	Include the Connect Record which failed to process in the log.	false	Boolean
errors.dlq.producer.*	Config for the KafkaProduce to produce to this topic (for example: `errors.dlq.producer.bootstrap.servers` will set the bootstrap servers of the Kafka cluster).		
errors.dlq.converter	Converter class used to convert the error context between Kafka Connect format and the serialized form that is written to Kafka.	org.apache.kafka.connect.json.JsonConverter	Any class which implements org.apache.kafka.connect.storage.Converter

Metrics

The following new metrics will monitor the number of failures, and the behavior of the response handler. Specifically, the following set of counters:

- for counting the failures at various stages.
- for how the framework handled this error.
  - number of messages retried
  - number of records skipped
  - number of messages logged to file or the dead letter queue

MBean name: `kafka.connect:type=task-error-metrics,connector=([-.\w]+),task=([-.\w]+)`

Metric/Attribute Name	Description	Implemented
record-failures	Total number of failures seen by this task.	2.0.0
records-skipped	Total number of records skipped by this task.	2.0.0
total-retries	Total number of retries made by this task.	2.0.0
failures-logged	The number of messages that was logged into either the dead letter queue or with Log4j.	2.0.0
dlq-records-produced	Number of records successfully produced to the dead letter queue.	2.0.0

dlq-produce-failures	Number of records which failed to produce correctly to the dead letter queue.	2.0.0
last-failure-timestamp	The timestamp when the last failure occurred in this task.	2.0.0

Proposed Changes

A connector consists of multiple stages. For source connectors, Connect retrieves the records from the connector, applies zero or more transformations, uses the converters to serialize each record's key, value, and headers, and finally writes each record to Kafka. For sink connectors, Connect reads the topic(s), uses the converters to deserialize each record's key, value, and headers, and for each record applies zero or more transformations and delivers the records to the sink connector. In this proposal, we will specifically deal with the following failure scenarios which can occur during these stages:

1. Connector tasks fail with `RetriableException` when reading/writing records from/to external system.
2. SMTs fail and throw exceptions while processing records.
3. Converters fail to correctly serialize/deserialize records.
4. `Exceptions` while producing/consuming to/from Kafka topics in the Connect framework.

There are two behavioral changes introduced by this KIP. First, a failure in any stage will be reattempted, thereby "blocking" the connector. This helps in situations where time is needed to manually update an external system, such as manually correcting a schema in the Schema Registry. More complex causes, such as requiring code changes or corruptions in the data that can't be fixed externally, will require the worker to be stopped, data to be fixed and then the Connector to be restarted. In the case of data corruption, the topic might need to be cleaned up too. If the retry limit for a failure is reached, then the tolerance limit is used to determine if this record should be skipped, or if the task is to killed.

The second behavioral change is introduced in how we log these failures. Currently, only the exception which kills the task is written with the application logs. With the additions presented in this KIP, more context will be logged:

1. The record which failed to process (if available).
2. The Exception along with the stack trace.
3. Information about the stage which failed to process the record (for example: classname of the transform and its configs).
4. The various stages in the connector, along with their configurations and order of processing.
5. The number of attempts, and time taken for these attempts.

Logging errors with application logs is convenient and requires no additional setup. The log messages are informative but not made easily actionable. For example, it is hard to collect log files from various machines, parse them and take appropriate actions. By introducing a dead letter queue, we can overcome these problems. For sink connectors, a developer can consume bad records from this topic, correct them and write the corrected records back to the original Kafka topics. Similarly, for source connectors, the developer can write the corrected records back to the original source.

While logging the error context, it might be worthwhile to take precautions to hide sensitive content. For example, some of the configs might contain sensitive information such as usernames or passwords. To prevent logging critical information, we provide configuration options to disable logging the messages (`errors.dlq.include.messages`) and/or configs (`errors.dlq.include.configs`).

Example 1: Fail Fast

To maintain backward compatibility, by default a Connector task will fail immediately upon an error or exception. This reflects the same behavior as earlier releases, ensuring that existing installations work the same way. Although it is not necessary to add extra configuration properties, the following properties may be added to a connector configuration to achieve this older behavior:

```
# disable retries on failure
errors.retries.limit=0

# do not log the error and their contexts
errors.log.enable=false

# do not record errors in a dead letter queue topic
errors.dlq.enable=false

# Fail on first failure
errors.tolerance.limit=0
```

Example 2: Record and Skip

The following configuration shows how to setup error handling with multiple retries, logging both to the application logs and a Kafka topic with infinite tolerance:

```
# retry at most 100 times waiting up to 5 minutes between consecutive failures
errors.retries.limit=100
errors.retries.delay.max.ms=300000

# log error context with application logs, but do not include configs and messages
errors.log.enable=true
errors.log.include.configs=false
errors.log.include.messages=false

# produce error context into the Kafka topic
errors.dlq.enable=true
errors.dlq.topic.name=my-connector-errors
errors.dlq.topic.partitions=25
errors.dlq.topic.replication.factor=3
errors.dlq.include.configs=true
errors.dlq.include.messages=true

# Tolerance all errors.
errors.tolerance.limit=-1
errors.tolerance.rate.limit=-1
errors.tolerance.rate.duration.ms=60000
```

Example 3: Record to separate Kafka cluster

In the previous example, errors are recorded in the log and in a separate "dead letter queue" (DLQ) Kafka topic in the same broker cluster that Connect is using for its internal topics. It is possible to record the errors in a DLQ on a *separate* Kafka cluster by defining extra `errors.dlq.producer.*` configurati on properties. Here is the same set of connector configuration properties as in Example 2, except with the additional `errors.dlq.producer.*` properties:

```
# retry up to 100 times waiting up to 5 minutes between consecutive failures
errors.retries.limit=100
errors.retries.delay.max.ms=300000

# log error context with log4j appender, but do not include configs and messages
errors.log.enable=true
errors.log.include.configs=false
errors.log.include.messages=false

# produce error context into a secure Kafka topic
errors.dlq.enable=true
errors.dlq.topic.name=my-connector-errors
errors.dlq.topic.partitions=25
errors.dlq.topic.replication.factor=3
errors.dlq.include.configs=true
errors.dlq.include.messages=true
errors.dlq.producer.bootstrap.servers=secure-broker:9093
errors.dlq.producer.acks = 0
errors.dlq.producer.security.protocol=SSL
errors.dlq.producer.ssl.truststore.location=/var/private/ssl/kafka.client.truststore.jks
errors.dlq.producer.ssl.truststore.password=test1234
errors.dlq.producer.ssl.keystore.location=/var/private/ssl/kafka.client.keystore.jks
errors.dlq.producer.ssl.keystore.password=test1234
errors.dlq.producer.ssl.key.password=test1234

# Tolerance all errors.
errors.tolerance.limit=-1
errors.tolerance.rate.limit=-1
errors.tolerance.rate.duration.ms=60000
```

Compatibility, Deprecation, and Migration Plan

The current behavior in Connect is to kill the task on the first error in any stage. As mentioned above, this will remain the default behavior if connector configurations are not changed.

Rejected Alternatives

Correcting records in the handler: the handler will not be responsible for providing corrected records. For sink records, the user can correct records and write the corrected records back to the origin Kafka topics using the dead letter queue mentioned above. For source records, the user can analyze the error messages and fix the data at the source.

Allow per-stage error handler: This would have provided finer grained error handling. But comes at the expense of more configuration, and users having to ensure that the different stages are using compatible error handling. It is also not evident that this is more useful than simply taking the most flexible of handlers and applying it across all stages.

Interceptors for Erroneous Records: Similar to ProducerInterceptor and ConsumerInterceptor, we could potentially add ErrorInterceptors too. But given that the ErrorHandler subsumes the functionalities here, we decided to not provide this feature.

Defining and configuring properties in the worker config: Firstly, it is out of place to specify error handling properties in a worker config when it will never be used by a worker (since all the failures are handled at the Connector level). Secondly, adding inheritance in configurations adds a level of complexity which can be avoided at this stage of development.