KIP-26 - Add Kafka Connect framework for data import /export

Status

Current state: *Accepted*

Discussion thread: here, here

JIRA: KAFKA-2365

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

Motivation

Kafka has become a standard storage system for large scale, streaming data. However, the current user experience when trying to adopt Kafka is poor because Kafka provides little support for getting data into or out of Kafka.

Consider some of these common use cases:

- Stream processing existing data (import) - the user has existing data from another source (such as logs or a database change log) and wants to use a stream processing framework on that data.
- Load streaming data into other storage systems (export) - the user has data stored in Kafka and wants to load it into HDFS/S3 for long-term storage and batch processing, Elasticsearch/Solr for text indexing and search, or Druid/InfluxDB or time series.
- Experiment with Kafka (import/export) - after a quickstart, new users need non-trivial data to experiment with, and would usually prefer using some data from their own application and might want to take the resulting data and integrate it with other existing services.

Although there are individual solutions for each of these use cases, users are left uncertain how to best accomplish these tasks because Kafka offers little guidance. They must find third party code or not uncommonly end up creating redundant standalone tools for each import/export task. There are many such connectors and are easily discovered (a quick search will turn up multiple options for Hadoop, ElasticSearch, Cassandra, Couchbase, and more), but they vary widely in functionality, robustness, scalability, and quality. Adopting any one connector is usually not difficult, but adopting just three or four can quickly turn into an integration and operational nightmare as connectors are written for different frameworks or as standalone solutions and have different requirements and dependencies which are often costly to setup or manage (e.g. Mesos, YARN). Further, many solutions to these problems are suboptimal because they are implemented in existing data copying/processing frameworks which cannot leverage Kafka-specific details and don't allow for strong semantics that can be achieved with Kafka such as certain delivery guarantees.

Kafka Connect addresses these problems by providing a standard framework for Kafka connectors. It abstracts the common problems these connectors need to solve: fault tolerance, partitioning, offset management and delivery semantics, operations, and monitoring. This makes developing connectors much simpler and adopting connectors easier, especially making each incremental addition trivial.

Proposed Changes

This KIP proposes a standardized framework for handling import and export of data from Kafka called Kafka Connect. This framework can address a variety of use cases including those described in the motivation section; would make adopting Kafka much simpler for users with existing data pipelines; would encourage an ecosystem of tools for integration of other systems with Kafka using a unified interface; and would provide a better user experience, guarantees, and scalability than other frameworks that are not Kafka-specific are able to.

The rest of this section covers some design goals derived from the motivation & use cases, discusses why this tool should be made part of Kafka rather than a third party tool, why it should be a Kafka-specific tool, and provides a rough, high-level system design.

Design Goals

The goal of adding the Kafka Connect framework is to make it as easy as possible to connect Kafka to other systems by copying data between Kafka and these systems. The Kafka Connect tool needs to support a variety of input/output systems and use cases, but they all share the common need of copying data, often at scale. To that end, there are a few important design goals any such framework should strive to achieve:

- Focus on copying data only – Focus on reliable, scalable data copying; leave transformation, enrichment, and other modifications of the data up to frameworks that focus solely on that process. Because some very simple transformations may be broadly applicable (e.g. drop a field, obfuscate sensitive data), it may make sense to include a very minimal hook or settings to perform these modifications. However, these additions must be added carefully and not come at the cost of other key design goals.
- Copy broadly by default – Endless configuration tweaking can quickly destroy the effectiveness of a tool like Kafka Connect. Connectors should be quick to define and be able to copy vast quantities of data between systems. For example, the default unit of work should be an entire database, even if it is possible to define connectors that copy individual tables.
- Parallel – Parallelism should be included in the core abstractions, providing a clear avenue for the framework to provide automatic scalability. Although some sources or sinks may naturally have no parallelism (e.g. a database change log), many others have significant parallelism (e.g. metrics, logs), and the framework should be capable of -- and encourage -- taking advantage of that parallelism.
- Strong semantics – If possible, it is better to provide exactly-once delivery than weaker semantics; it is preferable to provide at-least or at-most once delivery than best effort. The framework should make it easy to provide stronger semantics when the connector system supports it.
- Capture metadata – Many systems provide data with a well-defined structure and types. The framework should be able to capture this metadata and preserve it through an entire data pipeline as long as connectors also preserve it. However, the framework should also be able to handle systems which do not include or do not provide this metadata.
- Accessible connector API – It must be easy to develop new connectors. The API and runtime model for implementing new connectors should make it simple to use the best library for the job, quickly get data flowing between systems, and still get all the benefits of the framework. Where the framework requires support from the connector, e.g. for recovering from faults, all the tools required should be included in the Kafka Connect APIs.
- Streaming and batch – Kafka Connect must be able to integrate well with both streaming and batch-oriented systems. Kafka's ability to interact efficiently with both these types of systems is one of its unique features and one which Kafka Connect can take advantage of to make integrating these types of systems seamless and easy.
- Scale to the application – Although Kafka Connect should support copying large scale data, it should also scale easily to the application or environment. It should be easy to run a single process with a single connector in development, testing or a small production environment, but also scale up to an organization-wide service for copying data between a wide variety of large scale systems.

Why make Kafka Connect Kafka-specific?

Kafka Connect is designed specifically for Kafka and one endpoint in every Kafka connector is always Kafka. In contrast, there are already a variety of frameworks for copying and processing data that provide highly generic interfaces and already have plugins for Kafka (examples: fluentd, Flume, Logstash, Heka, Apache Camel). However, this generic approach misses out on a lot of important features of Kafka.

First, Kafka builds parallelism into its core abstraction: a partitioned topic. Fixing Kafka as one half of each Kafka connector leverages and builds upon this parallelism: sources are expected to handle many parallel input sequences of data that produce data to many partitions, and sinks are expected to consume from many Kafka partitions, or even many topics, and generate many output sequences that are sent to or stored in the destination system. In contrast, most frameworks operate at the level of individual sequences of records (equivalent to a Kafka partition), both for input and output (examples: fluentd, Flume, Logstash, Morphlines, Heka). While you can achieve parallelism in these systems, it requires defining many tasks for each individual input and output partition. This can become especially problematic when the number of partitions is very large; Kafka Connect expects this use case and allows connectors to efficiently group a large number of partitions, mapping them to a smaller set of worker tasks. Some specialized cases may not use this parallelism, e.g. importing a database changelog, but it is critical for the most common use cases.

Second, Kafka Connect can take advantage of Kafka's built-in fault tolerance and scalability to simplify Kafka Connect, both operationally and it's implementation. More general frameworks usually standardize on the lowest common denominator abstraction -- a single partition that may be persisted to disk but is not fault tolerant in a distributed sense -- because the burden of implementing connectors for many systems would be too high if they did not (examples: Logstash, Heka, Fluentd; Flume's storage is configurable and can be backed by Kafka). By requiring Kafka as one endpoint, Kafka Connect can leverage Kafka features such as consumer groups, which provide automatic partition balancing and fault tolerance, without any additional code. Connectors must be aware of the semantics provided by Kafka, but their code can remain very simple.

Third, by working directly with Kafka, Kafka Connect provide flexible delivery guarantees (at most once, at least once, and exactly once) without additional support from connectors. This is possible to achieve with the right set of primitives in a more general framework (flush, offset tracking, offset commit), but requires pushing more of that functionality into connectors rather than implementing it once in the framework. Many frameworks cannot provide delivery guarantees (examples: Logstash, Heka), some can provide some of these guarantees under some situations given careful configuration (example: fluentd, Flume), but none provide an out-of-the-box solution that makes it easy for the user to achieve different semantics without requiring a deep understanding of the the framework's architecture.

Besides leveraging Kafka-specific functionality, there are drawbacks to adopting any of the numerous existing general-purpose frameworks. Most are not actually general purpose because they were initially designed around a specific use case (e.g. logs, ETL) and later generalized; however, their designs -- and limitations -- clearly highlight their origins. Many of these systems also grew broader in scope over time, making them more complex to learn, deploy, and making it harder to provide useful guarantees, such as data delivery guarantees. Another issue is that many tools, especially those focused on ETL, tend to require a specific runtime model/environment, e.g. they require YARN. Such a requirement is impractical for a general purpose tool as the number of cluster resource management strategies is quickly growing, let alone traditional process management strategies. It is better to be agnostic to the use of these tools rather than depending on them. Finally, many of these tools do not fit in well with the technology stack Kafka requires. For some users they may be preferable if they match their existing stack and infrastructure (e.g. Ruby for fluentd), but for many users a tool that fits well with the stack (and knowledge) that Kafka already requires would be preferable.

Why add Kafka Connect to Kafka?

Kafka Connect as described should not rely on any Kafka internals. While there is no technical requirement that Kafka Connect be included directly with Kafka, doing so has a number of benefits.

First, standardizing on a framework and tool for performing data copying allows the project to deeply integrate it in documentation. Many aspects of getting started with or using Kafka become much simpler for both new and experienced users. Beyond the quickstart examples that currently use the console-producer and console-consumer, new users will have some guidance on how to best get real data into or out of Kafka. The tool will already be included with the binary distribution they have downloaded.

Second, it encourages a healthy ecosystem of connectors in the Kafka ecosystem. Currently, connectors are spread across many one-off tools or as plugins for other frameworks. This makes it more difficult to find relevant connectors since the user needs to find a framework that supports both Kafka and their source/sink system. An ecosystem of connectors specifically designed to interact well with Kafka is increasingly important as more users adopt Kafka as an integral part of their data pipeline and want a large fraction or all of their data flowing through Kafka.

Finally, Kafka connectors will generally be better (e.g. better parallelism, delivery guarantees, fault tolerance, scalability) than plugins in other frameworks because Kafka Connect takes advantage of being Kafka-specific, as described in the previous subsection. Kafka Connect will benefit by being closely tied to Kafka development, and vice versa, by ensuring Kafka APIs, abstractions, and features coevolve with Kafka Connect, which represents an important use case.

Design

Kafka Connect’s design can be broken into three key components: the data model that defines the structure of data that Kafka Connect manages, the connector model which defines the interface to external systems, and the worker model which defines how connectors are executed and how the system implements various aspects such as coordination, configuration storage, offset storage, and offset commit management for different delivery guarantees.

Data Model

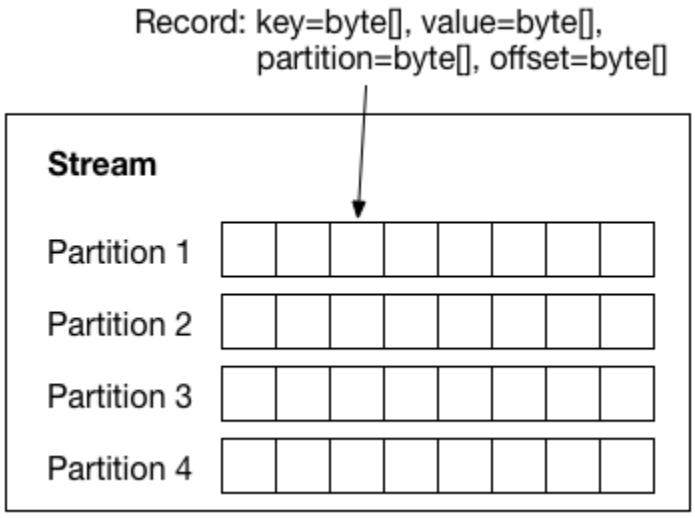
Kafka Connect’s job is to copy events (or messages, records, changes, or your preferred terminology) from one system to another. To do so, it needs a generic representation for structured data that is not dependent on any particular system (data storage system or data serialization system). This is a significant departure from Kafka which is completely agnostic to data formats and, aside from the serializer interfaces provided for convenience, only operates on byte arrays.

Kafka Connect’s data model only assumes an in-memory runtime format, leaving the details of (de)serialization to a pluggable component, much like the existing serializer interface. The data model must be able to represent complex data (object/record/dictionary types), but must also allow for standalone primitive types that will commonly be used as keys for events. Except for primitive types, all records include schemas which describe the format of their data. Including this schema facilitates the translation of the runtime data format to serialization formats that require schemas.

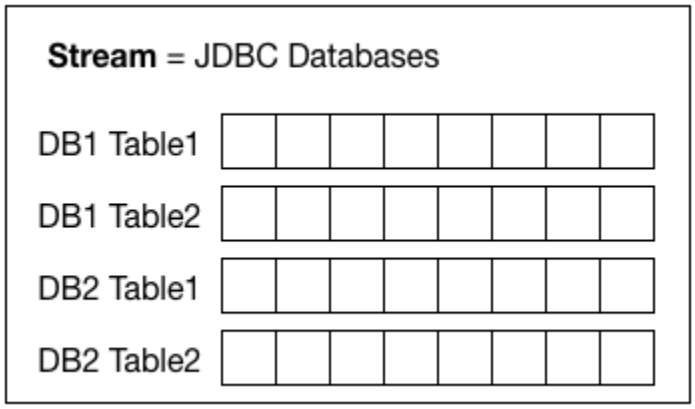
Records and Partitioned Streams

The basic structure that all Kafka Connect source and sink systems must be mapped to is the partitioned stream of records. This is a generalization of Kafka's concept of topic partitions. The stream is the complete set of records, which are split into independent infinite sequences of records. Each record can contain:

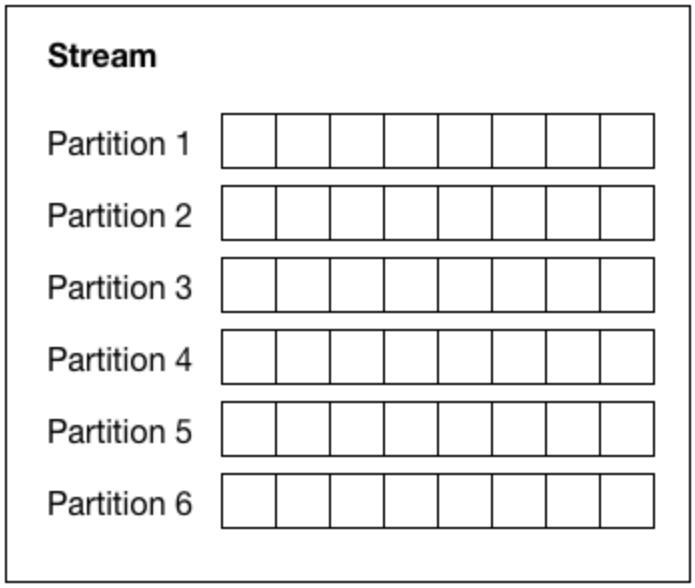
- key and value - the event data, which after serialization are both byte[] (or null)
- partition - and partition identifier. This can be arbitrarily structured (again, after serialization it will be a byte[]). This is a generalization of Kafka integer partition IDs.
- offset - a unique identifier indicating the position of the event in the partition. This can be arbitrarily structured (also a byte[]). This is a generalization of Kafka long offsets.



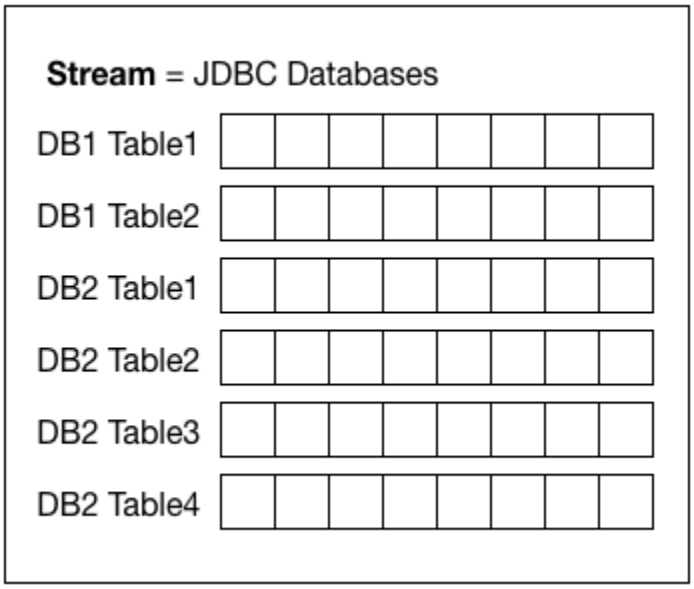
As a concrete example, we might model a collection of databases we want to import via JDBC as a stream. Each table is assigned its own partition, and each record in a partition will contain one update to one row of the table. As shown below, since the stream covers many databases, partitions are labeled by the combination of the database and table.



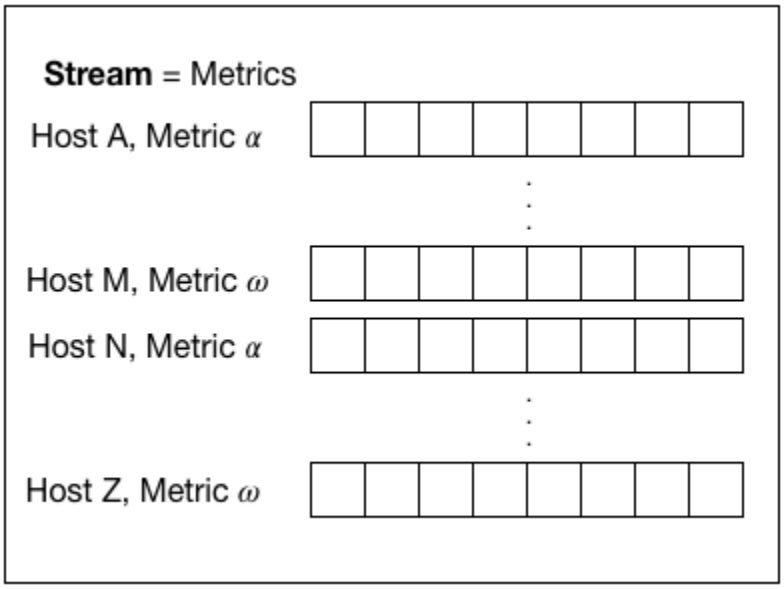
Streams may have a dynamic number of partitions. Over time, a stream may grow to add more partitions or shrink to remove them:



Although this is possible in Kafka, it is not very common. In Kafka Connect, it may be more common. For example, in the JDBC example, new partitions will be added when new tables are created, as shown here where DB2 has added Table3 and Table 4.



The number of partitions may also be very large and Kafka Connect does not require that all partitions be listed. A somewhat extreme example of this would be metrics collected from a large number of hosts, e.g. application stats collected via JMX or OS-level stats collected via ganglia. Logically, we can represent these as a very large number of partitions, with one partition per host per metric (perhaps hundreds of thousands or even millions across a data center) and Unix timestamps as offsets.



Connector Model

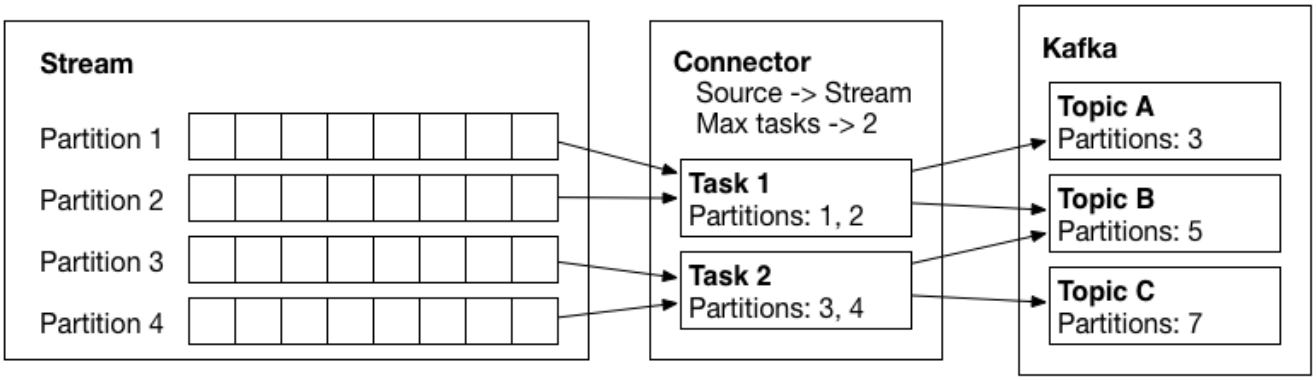
The connector model defines how third-party developers create connector plugins which import or export data from another system. The model has two key concepts: a Connector, which divides up the work, and Tasks, which are responsible for producing or consuming records.

Connectors

Connectors are the largest logical unit of work in Kafka Connect and define where data should be copied to and from. This might cover copying a whole database or collection of databases into Kafka, or a set of topics matching a regex into HDFS. However, the connector does not perform any copying itself. Instead, it is responsible for using the configuration to map the source or sink system to the partitioned stream model described above and grouping partitions into coarser-grained groups called tasks. This is an ongoing task because the partitioning may be dynamic; the connector must monitors the input or output system for changes that require updating that distribution of partitions.

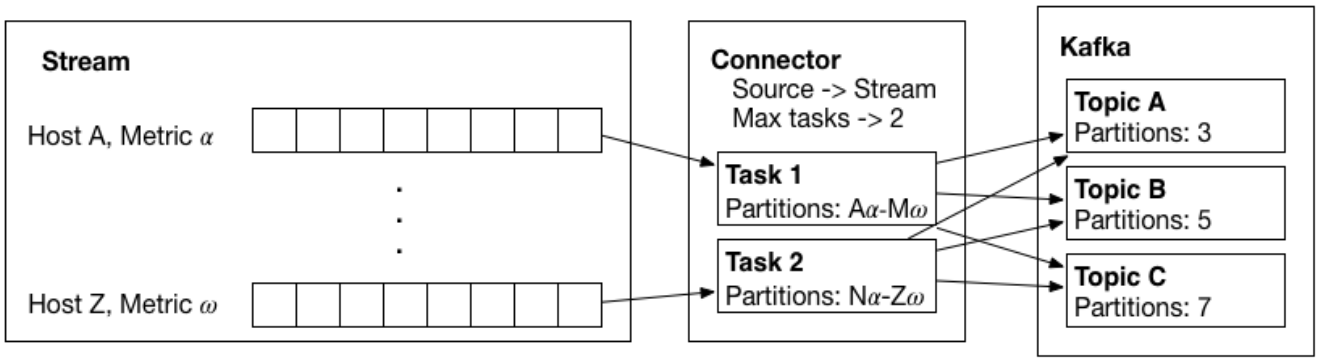
Tasks

Tasks are responsible for producing or consuming sequences of Kafka ConnectRecords in order to copy data. They are assigned a subset of the partitions in the stream and copy those partitions to/from Kafka. Tasks also provide control over the degree of parallelism when copying data: each task is given one thread and the user can configure the maximum number of tasks each connector can create. The following image shows the *logical* organization of a source connector, its tasks, and the data flow into Kafka. (Note that this is not the *physical* organization, tasks are not executing inside Connectors.)



Partitions are balanced evenly across tasks. Each task reads from its partitions, translates the data to Kafka Connect's format, decides the destination topic (and possibly partition) in Kafka. Note that partitions in the input system do not need to be mapped to a single topic or partition in Kafka; the connector may perform arbitrary shuffling and data transformation.

In this simple case, we can easily list all the partitions and assign them simply by dividing the list between the tasks. However, this approach is not required. The Connector is responsible for assigning partitions to tasks and it can use any approach that makes sense. For example, the metrics example from earlier might look like this:



Instead of listing every single metric across all hosts or application processes, the connector might only divide work between tasks at the granularity of hosts and might even specify this as a range of hosts rather than actually listing the full set of hosts. So in the example, the connector could generate configs that specify the range of hosts the task should handle (e.g. server.range=a-m and server.range=n-z) and tasks, which are implemented as part of the same connector plugin, know to handle all metrics for all servers with hostnames in that range. It is important that each metric be its own partition so that offsets can be tracked for each individually (enabling correct handling of failures), but the Kafka Connect framework does not need to know the full list of partitions or exactly how they are assigned to tasks.

Aside from lifecycle events, source and sink tasks have different APIs. Source tasks are pull-based and provide an API for polling for new input records from the source system. The framework will call this method continuously, then handle serializing the records and producing them to Kafka. The Kafka Connect framework will also track the input partition offsets included with the records and manage the offset commit process, ensuring data has been fully flushed to Kafka.

Sink tasks are push based and provide an API for accepting input records that have been read and parsed from a Kafka consumer. They may either use Kafka's built-in offset commit combined with a flush method they provide to have the framework manage offset commits, or they can manage offsets entirely in the output data store.

Worker Model

The worker model represents the runtime in which connectors and tasks execute. This layer decouples the logical work (connectors) from the physical execution (workers executing tasks) and abstracts all the management of that scheduling and coordination between workers.

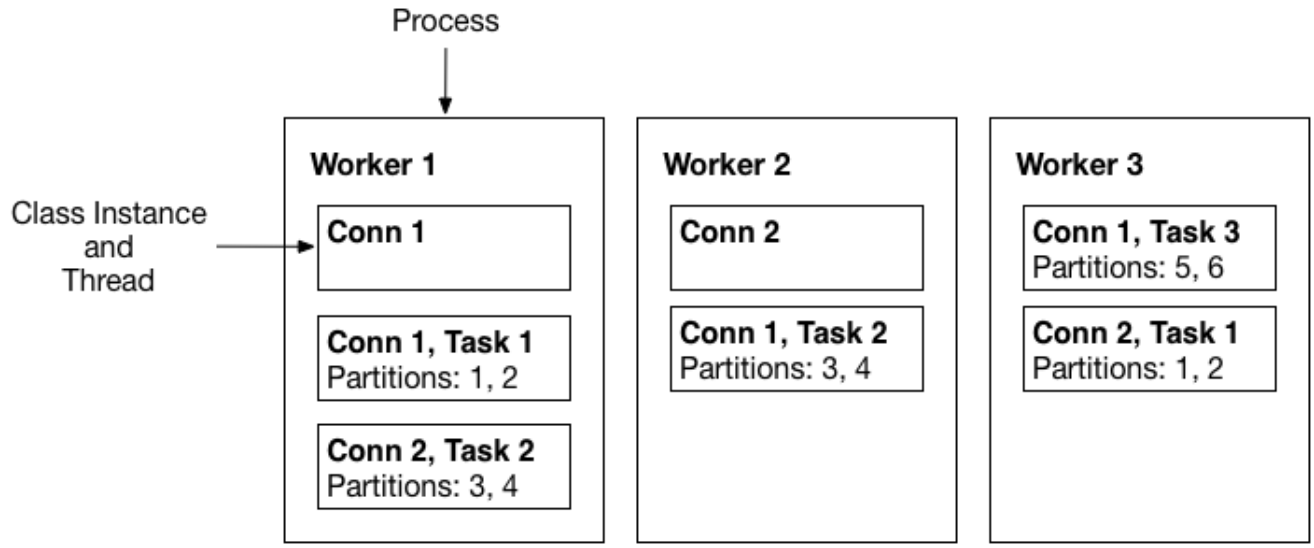
Workers

Workers are processes that execute connectors and tasks. Each worker is a single process, and how these processes are spawned or managed is outside the scope of Kafka Connect. They do not do any resource management -- they just run the work they are assigned by instantiating the appropriate classes, passing in configuration, and managing the components lifecycle. Workers provide a way to run connectors and tasks without assuming any particular process or cluster management tool (although workers themselves may be running under any one of these tools). They also allow many tasks to share the overhead of a single process since many tasks may be lightweight in terms of CPU and memory usage. Workers are assumed to have homogeneous resources so that balancing tasks across them can be kept simple.

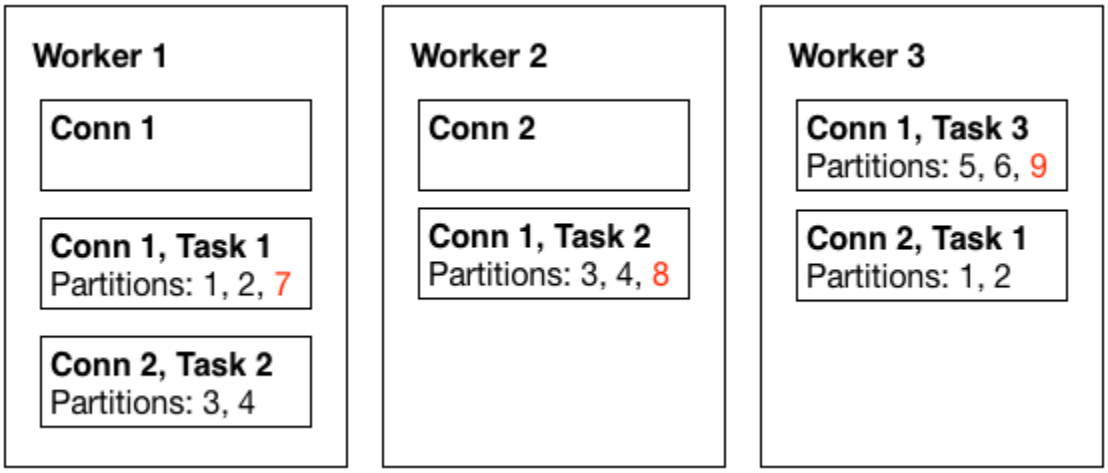
Balancing Work

Since connectors and tasks may be lightweight, they are not executed in their own processes. Instead, they are assigned to workers, which execute many in the same process. The framework manages this process by tracking which workers are currently running and balancing work across them. The exact implementation is not specified here. However, this does require some coordination between a collection of worker processes, both to distribute the work and to share some persistent state to track the progress of each connector (i.e. offset commit).

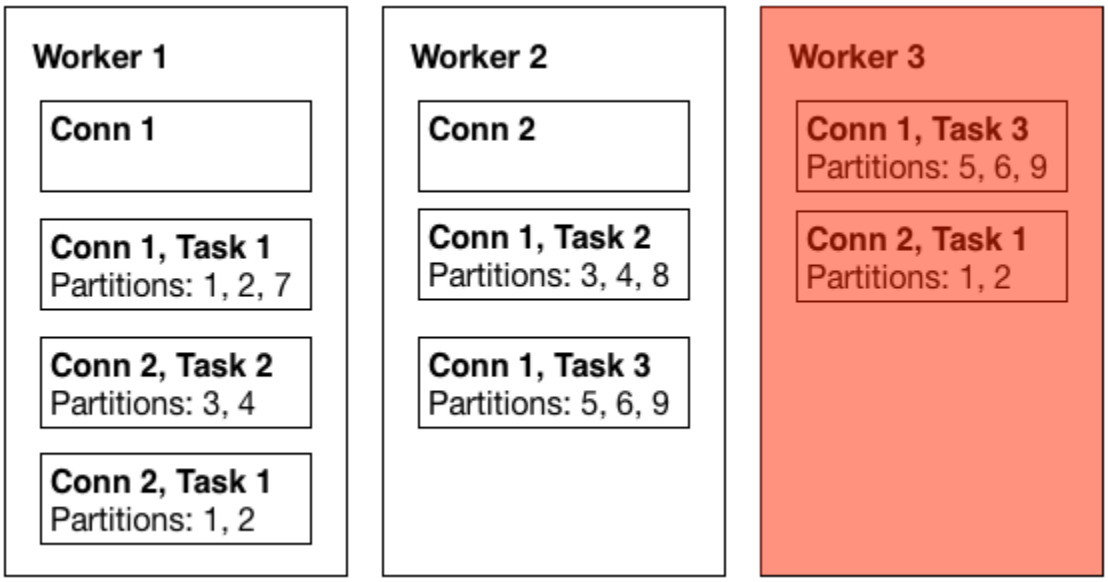
Here is a simple example of a cluster of 3 workers (processes launched via any mechanism you choose) running two connectors. The worker processes have balanced the connectors and tasks across themselves. Note that although the partition assignments are labeled in this image, the workers are not aware of them.



If a connector adds partitions, this causes it to regenerate task configurations. Although this can create new tasks in some cases, usually it will just require the connector to change the assignment of partitions to tasks. In this case, as shown below, the tasks need to be reconfigured, but no changes are made in the mapping of tasks to workers.



If one of the workers fails, the remaining workers rebalance the connectors and tasks so the work previously handled by the failed worker is moved to other workers:



The same mechanism is used to add or remove capacity by starting or stopping worker processes, but the process will be handled gracefully.

This load balancing process performed by the workers can use a simple assignment policy (e.g. uniform distribution) and should be relatively infrequent (comparable to consumer group partition rebalancing), only occurring when:

- A new worker process is started
- A worker process shuts down gracefully or fails
- A user reconfigures the maximum number of tasks for a connector
- The set of partitions for a connector changes *and* the resulting assignment of partitions to tasks generated by the connector results in a different number of tasks than the previous configuration. This should be unusual since it requires that the number of partitions is smaller than the user's setting for maximum number of tasks. For example, if a user specifies max.tasks=20 for a JDBC connector where there are only two tables to copy initially it would only generate two tasks, each assigned one table. If a third table was added, the new configuration would have 3 tasks. In most cases, since partitions can be fine-grained, the number of partitions will be much larger than the maximum number of tasks.

Although the implementation is not specified here, it must provide an interface that can be used in embedded mode or by the REST API to accomplish the following operations:

- Get a list of connectors.
- Create a new connector with a provided configuration.
- Get the current state of a connector and its tasks.
- Get a connector configuration.
- Update a connector configuration.
- Get the configuration and state of a connector task.
- Delete a connector.
- Get a list of active workers and their state & tasks.

At least two implementations should be provided: a trivial standalone implementation (single process, no coordination) for agent-style applications and a distributed implementation which is required for any non-agent mode deployment.

Data Storage

In order to be fault tolerant, Kafka Connect needs to store three types of data: connector configurations (provided by the user), task configurations (generated by connectors), and offset data.

This KIP will not specify the implementation of this storage. However, to support the different execution modes, there are likely to be two implementations. The first will use local file system storage and is useful for standalone mode to store offset data. This mode does not require connector or task configuration storage because there is only one process and it is expected that the connector configuration is provided each time Kafka Connect is executed. The second implementation is distributed and will likely store data in Zookeeper or Kafka since both are readily available. The Kafka Connect implementation can cleanly isolate this functionality to an interface so alternative implementations may be possible.

Because Kafka already provides offset storage for consumers, the need for a new storage system for offset data warrants further discussion. Kafka Connect extends the idea of offsets to a more general form, allowing partition IDs and offsets for source connectors to have arbitrary structure. These cannot easily be mapped directly to the topic-partition offsets that Kafka provides for consumers, even with the recent addition of metadata that can be attached to each of the offsets. Therefore, at least for source connectors, another more general mechanism will be necessary. However, sink tasks may commonly be able to use Kafka's existing offset commit, and will not need to consider offsets at all since this process can be managed by the Kafka Connect framework.

Delivery Guarantees

Kafka Connect can support three different levels of delivery guarantees between a source and sink system: at least once, at most once, and exactly once. Each of these require certain set of functionality for the input and output systems.

At least once: each messages is delivered to the output system at least one time, but possibly multiple time. This requires the output system to support a flush operation, which is followed by offset commit in the input system. For Kafka Connect sources, KafkaProducer is the output and already supports flush, and the Kafka Connect framework provides storage and offset commit for the general form offsets generated from the source system. For sinks, the task is the sink and must implement a flush method, and the source is the KafkaConsumer and already supports offset commit stored in Kafka.

At most once: each message is delivered to the output system at most once, but may not be delivered at all. This requires buffering of records in the framework so that offsets can be committed in the input system (where the offsets may not be available until the data is read) and then records are sent to the output system. Offset commit is provided by the same components as described in the at least once section.

Exactly once: each message is guaranteed to be delivered to (or stored in) the output system exactly once. The implementation will depend on the output system, requiring either idempotency or that input system offsets are tracked in the output system. For source connectors, the output system is Kafka and should be able to achieve exactly once semantics with the idempotent producer; for sink connectors, it can be achieved in a system-specific way, e.g. using <topic>-<partition>-<offset> as a document ID in a document store like Elasticsearch or atomically committing data & offset using a file rename in HDFS.

Integration with Process Management & Cluster Resource Managers

Process and resource management is orthogonal to Kafka Connect's runtime model. The framework is not responsible for starting/stopping/restarting processes and can be used with any process management strategy, including YARN, Mesos, Kubernetes, or none if you prefer to manage processes manually or through other orchestration tools.

Here are some examples of how Kafka Connect might be deployed and how resource management frameworks can be applied:

- **Kafka Connect as a service**: Run a collection of Kafka Connect worker processes (via any mechanism, e.g. using Chef/Puppet/Ansible/Salt). Users submit connectors via the REST API. Internally, the worker processes instantiate the connectors and tasks and balance them over the available workers. A resource manager can be used to make the cluster self-healing (respawning workers if they die); however, even if workers are not automatically restarted, connectors continue to run properly even if a worker fails because the connectors and tasks that were on that worker are relocated to the remaining live workers.
- **Resource constrained connectors**: To apply resource constraints to specific connectors, start the desired number of Kafka Connect workers with resource constraints via YARN, Mesos, or Kubernetes (or the appropriate framework, such as Slider or Marathon). Only one connector is submitted to this cluster. Internally, tasks are still balanced across worker processes and the workers will handle failure of one of the processes as usual. The restriction to a single connector is only required to allow the resource manager to apply constraints to a single connector.
- **Small resource constrained connectors**: If the data to be copied is small enough to be handled by one process, you want to rely on a resource manager to handle failures via restart, and you want to apply resource constraints, then you can run Kafka Connect in standalone mode with a single connector configuration under a resource manager. Note, however, that offset storage, managed in a local file in standalone mode, must be available across restarts.
- **Embedded application**: Using the embedded API, the application starts a source connector to import data to Kafka and is returned a list of topics where the data will be stored. Using Kafka's client APIs or a higher level stream processing API, it consumes from the topic, transforms the data, and produces the resulting data to a new topic. Many instances of the application can be executed in parallel, e.g. by deploying with Chef/Puppet /Ansible/Salt. The embedded API internally works just like a worker process and connectors and tasks will be automatically balanced across the active set of application processes, and the consumer will independently automatically balance the partitions across the application processes.

Examples

To make some of these concepts more concrete, this section describes a few possible connectors, their implementations, and applications.

JDBC Import

A JDBC connector would provide a generic way to import data from relational databases. The default scope for copying would be an entire collection of databases, and by default each table would be a partition (or a query would be a partition if the user configured one, e.g. to include a join). The connector can poll the database periodically to check for new tables, resulting in new partitions.

To generate records, tasks poll each table for updates, using timestamps, autoincrement columns, or a combination of both to detect additions and changes. Schemas are derived from the table's columns and the records will have a trivial, flat record structure. The partition ID for records will be a combination of the database name, the table name and the offsets will be the timestamp, autoincrement ID, or a tuple containing both. These offsets are used to construct a query on each poll of the table that restricts the query results to new data.

HDFS Export

HDFS export is a very common use case for long-term storage and batch analysis of data. The input for this connector would be a list of topics or a regex subscription. Since this will usually cover a large quantity of data, this will generally be used in clustered mode. Dynamic inputs are handled automatically by the Kafka consumer. This connector can provide exactly once delivery by managing offsets with the data stored in HDFS. It can write to temporary files as the data streams in. To commit data, it closes the file and performs an atomic rename. Offset information is included with the file (or in the filename) to ensure atomicity of committing both the data and the offsets. After faults, instead of resuming wherever the Kafka consumer left off, it will check the offsets in HDFS and seek the underlying consumer to the correct position before writing new data.

Log Import

Log file import is a common use case and some applications may not be able to deliver logs directly to Kafka. A log file connector would run in standalone mode since it needs to run on each server that logs are collected from. The configuration would provide a list of files or regexes for files to load. Each file would become an input partition and offsets could be recorded as byte offsets into that file. The simplest implementation would create one record for each line (and records would have trivial structure). A more complex implementation might allow for a simple regex specification of the format of the logs.

Mirror Maker

The existing mirror maker tool can be thought of as a special case where rather than just one endpoint being Kafka, both the input and output are Kafka. This could potentially be implemented as either a source or a sink connector. One of the connections, managed by the framework, would use the Kafka cluster that is set globally for the entire Kafka Connect cluster; the second connection would be managed by the connector itself and use a remote Kafka cluster. If the implementations are sufficiently compatible, it might make sense to eventually deprecate the original mirror maker tool and provide a compatible Kafka Connect wrapper script. A mirror maker connector would be a good candidate for a built-in connector since it is a commonly needed tool and requires no additional dependencies.

Public Interfaces

This section describes the different public, user-facing interfaces that Kafka Connect will include, but these are not intended to necessarily represent the final interface. Rather, the goal of this section is to give a sense of the scope and usage of the Kafka Connect tool.

CLI

The primary interface for Kafka Connect is the REST interface. However, many users' first exposure to Kafka Connect will be via the command line because that is the natural interface to run a simple, standalone Kafka Connect agent.

This KIP introduces two command line tools. However, since these do not provide access to all Kafka Connect functionality, it is likely that we will want to add more command line utilities to query and modify Kafka Connect. The commands included are the minimum required to provide baseline functionality in conjunction with the REST interface, but additional commands providing CLI access to functionality only exposed by the REST interface in this KIP can be added in subsequent KIPs.

copycat

The copycat command runs Kafka Connect in standalone (agent) mode. This mode runs one or more Kafka Connect jobs in a single process:

```
copycat [...storage options...] connector.properties [connector2.properties …]
```

Functionally, this behaves as if you started a single node cluster and submitted each connector properties file as a new connector, however:

- It acts completely standalone and does not need to coordinate with other Kafka Connect workers. This allows it to run on any machine without extra configuration or any more access than any other Kafka client would normally have.
- The connector configuration is not placed in persistent storage. It is expected that if you kill the process, you will provide the connector definition again if you want to resume it.
- Offset storage is local and unreplicated. The data will be persisted to disk by default, allowing resumption in the case of failure, reboots, or other causes of process restarts. Extra command line arguments can be passed to control how offset storage is managed, e.g. to disable it entirely for testing purposes.
- A trivial implementation of work balancing is used because there is only one worker process and no real coordination is required.

copycat-worker

The copycat-worker command starts a worker in cluster mode:

```
copycat-worker <... config options for distributed mode ...>
```

In contrast to the standalone mode, this mode:

- Requires some configuration to setup work balancing and shared storage of connector/task configs and offset storage.
- The command does not include a connector; it only starts a worker. All workers will be running a REST interface which can be used to submit /modify/destroy persistent connectors.

REST Interface

All workers running in cluster mode run a REST interface that allows users to submit, get the status of, modify, and destroy connector. API calls can be made on any worker, but may require coordination with other workers to execute. The following is a sketch of the key API endpoints:

```
GET /connectors
```

Get a list of connectors.

```
POST /connectors
```

Create a new connector by passing its configuration as a set of string key-value pairs.

```
GET /connectors/<id>
```

Get the current state of a connector, including the current list of tasks for the connector.

```
GET /connectors/<id>/config
```

Get the connector configuration.

```
PUT /connectors/<id>/config
```

Update the connector configuration. Updating requires reconfiguring all tasks and this call will block until reconfiguration is complete or timed out. Because this may require flushing data, this call may block for a long time.

```
GET /connectors/<id>/tasks/<tid>
```

Get the configuration and state of a connector task.

```
DELETE /connectors/<id>
```

Destroy a connector. This will try to cleanly shutdown and so may block waiting for data to flush from connectors.

Embedded API

Kafka Connect can also be run in an embedded mode, allowing you to adopt it quickly for a specific application without complicating deployment & operations. This mode supports distributed mode so it can scale with your application and automatically balance work across the active set of processes. Although this KIP does not specify the complete API, here is a simple example of what this might look like:

EmbeddedSample.java

```
// Start embedded Kafka Connect worker with a unique ID for the Kafka Connect
// cluster. Any application that uses the embedded API with the same
// ID will join the same group of Copycat processes, across which
// connectors and tasks will be balanced.
final Copycat copycat = new Copycat(“app-id”);
copycat.start();

// Start import to load data into Kafka. Here we show how this could return
// information about where the data is being produced. For simple cases you
// simply know the output (here there is a single output topic). In other
// cases it may be dynamic and connector-specific, so you may need to get the
// information from Kafka Connect after the connector has been started.
Properties importConfig = new Properties();
importConfig.setProperty("connector.class", "org.apache.kafka.copycat.JDBCSourceConnector");
importConfig.setProperty("connector.tasks.max", "20");
importConfig.setProperty("jdbc.connection", "...");
importConfig.setProperty("jdbc.table.whitelist", "some_table"); // Only load a single table
importConfig.setProperty("jdbc.topic.prefix", "my-database-"); // Prefix for output Kafka topics, i.e. the one
table's output topic will now be my-database-some_table
 ... other connector settings ...
String[] inputTopics = copycat.addConnector(importConfig);


// Now we can use a stream processing layer to do some transformations and
// then write the resulting data back into Kafka. These APIs are not from
// an existing stream processing framework and are only intended to demonstrate
// how embedded Kafka Connect could combine nicely with an embeddable stream
// processing layer built on Kafka's existing clients. We could also do this
// using the consumer and producer directly.
Streaming streaming = new Streaming(config);
streaming.start();
Stream<K,V> inputStream = streaming.from(inputTopics);
inputStream.map((key,value) -> new KeyValue<K,V>(key, value*2))
           .filter((key,value) -> value != null)
           .sendTo("output");

// Finally, we can create Start export to get data from “output” topic to, e.g., HDFS.
// Since this is a sink connector, we don't need to capture any information that
// is returned when we add the connector like we did with the source connector -- the
// set of input topics and the destination (e.g. HDFS folder) are both specified in
// the config.
Properties exportConfig = new Properties();
exportConfig.setProperty("connector.class", "org.apache.kafka.copycat.HDFSSinkConnector");
exportConfig.setProperty("connector.tasks.max", "20");
exportCOnfig.setProperty("hdfs.url", "hdfs://my-hdfs-server:9001/export/"); // Where to store data
 ... other connector settings ...
copycat.addConnector(exportConfig);

// Handle shutdown cleanly.
Runtime.getRuntime().addShutdownHook(new Thread() {
  @Override
  public void run() {
    copycat.stop();
    streaming.stop();
  }
});

copycat.join();
streaming.join();
```

In this way you can easily and intuitively set up an entire application pipeline and contain everything in one set of processes even though it combines functionality from multiple frameworks. The embedded version of Copycat supports all the same functionality (and under the hood is the same implementation) as the distributed version. You can start as many (or as few) of these processes as desired and each of the components scales up and down automatically, rebalancing work as needed. Note that in embedded mode we still specify a setting for the maximum number of tasks each connector should use. This allows the user to control parallelism (number of threads) regardless of the number of application processes currently running. It may also be useful to have a special setting which always creates one task per server, which more closely matches the new consumer model.

Compatibility, Deprecation, and Migration Plan

This KIP only proposes additions. There should be no compatibility issues.

Rejected Alternatives

Make Kafka Connect an external third-party framework

Kafka Connect is a new framework which should not rely on any internals of Kafka. Therefore, it could be maintained as an external framework which happens to be Kafka-specific. We've rejected this because:

- One of the primary goals of Kafka Connect is to ease adoption of Kafka by making it easier to get data into and out of Kafka. Shipping Kafka Connect as part of a normal Kafka release makes it immediately available to every user that downloads Kafka.
- Perhaps more importantly, it encourages deep integration of Kafka Connect with the Kafka documentation.
- Feature/mission creep in the project is a danger in adding new tools like this. However, besides custom integrations or consumer applications, which the client libraries serve well, getting other types of data that already exist in other systems is a popular enough need that the project should directly provide some support for doing so. The impact of this change is kept minimal by only including the core Kafka Connect framework in Kafka, leaving connector plugins as third-party plugins.

Maintain connectors in the project along with framework

We could potentially maintain some or all connectors within Kafka alongside the framework. However, this has a number of drawbacks:

- Additional load on reviewers, project mailing lists, and general noise in the project
- Overhead in managing organization of multiple repositories, or organizing one repository to make development of each tool work well without impeding development of other components
- Would need to select which subset to maintain in Kafka, have a mechanism for promotion from third-party to "official" plugin, etc.

In practice, maintaining any connectors in the project (aside from a very simple example connector) shouldn't be necessary. The project gets all the benefits by providing the core framework and tools, and can choose to document available connectors or even endorse some as standard if users need further guidance in finding the connectors for their use cases. Additionally, it forces the framework to immediately think through how to handle third-party plugins well.

The major drawback to this approach is that all use cases require using some third-party code, which will require getting the appropriate JARs on the classpath. Additionally, it will not be possible to provide centralized, unified documentation. However, documentation for connector plugins will mostly be limited to configuration options.

Use existing stream processing framework

Some people are already using stream processing frameworks as connectors for other systems. For example, some users do light processing using something like Spark Streaming and then save the output directly to HDFS. There are a couple of problems with this approach.

- Lack of separation of concerns. Stream processing frameworks should be focused on transformation and processing of data. Philosophically, import/export and transformation should be separate, and they effectively are in these frameworks.
- Stream processing frameworks face a similar problem as frameworks like fluentd, flume, logstash, and others: in order to support a wide variety of systems, and arbitrary combinations of systems, they have to make some sacrifices in their abstractions and implementations. Kafka Connect can avoid making these sacrifices because it is specific to Kafka, and since Kafka is a de facto standard data store for stream processing and supported by all the major stream processing frameworks, it will quickly benefit all frameworks.
- There are multiple good stream processing systems, and having Kafka choose one external project to support does not make sense. Further, by providing a set of high quality connectors, Kafka can alleviate the development and integration burden of these other projects because they can leverage these connectors instead of writing their own. Instead, they will be able to focus on their core contribution, a framework for stream processing.

Support push-based source connectors

Some systems push data to a service which collects and logs that information. For example, many metrics systems such as statsd work this way: applications generate the data, send it to the collector, and likely do not save the data locally in any way. To support this use case, Kafka Connect would have to act as a server in a stable location (set of servers and fixed ports). In contrast, most source connectors pull data from the data sources they're assigned, and so can easily be placed on any worker node. This flexibility keeps the Kafka Connect distributed runtime much simpler.

Supporting push-based source connectors makes the task scheduling sufficiently complex and requires integration with configuration and deployment of applications that it does not make sense to support this in Kafka Connect. Instead, if push-based reporting is being used the recommended integration is to have Kafka Connect collect the data from whatever service is collecting the data (e.g. statsd, graphite, etc.)

Note that this does not exclude running a standalone, agent-style Kafka Connect instance when the connector cannot function in cluster mode. Examples of this use case are collecting log files (where logs can't be sent directly to Kafka) or collecting a database change log (which might require being colocated with the database). Further, while push-based connectors are not supported directly by the framework, connectors can be written to listen for network events in order to support these applications locally.