Kafka Connect Architecture

Kafka Connect's goal of copying data between systems has been tackled by a variety of frameworks, many of them still actively developed and maintained. This section explains the motivation behind Kafka Connect, where it fits in the design space, and its unique features and design decisions.

Motivation

Why build another framework when there are already so many to choose from? A lot of effort has already been invested in building connectors for many systems, so why not simply reuse them?

In short, most of these solutions do not integrate optimally with a stream data platform, where streaming, event-based data is the lingua franca and Apache Kafka® is the common medium that serves as a hub for all data. Given a centralized hub that other systems deliver data into or extract data from, the ideal tool will optimize for individual connections between that hub (Kafka) and each other system.

To see why existing frameworks do not fit this particular use case well, we can classify them into a few categories based on their intended use cases and functionality.

1. **Log and metric collection, processing, and aggregation**

   Examples: Flume, Logstash, Fluentd, Heka

   These systems are motivated by the need to collect and process large quantities of log or metric data from both application and infrastructure servers. This leads to a common design using an *agent* on each node that collects the log data, possibly buffers it in case of faults, and forwards it either to a destination storage system or an aggregation agent which further processes the data before forwarding it again. In order to get the data from its source format into a format suitable for the destination, these systems have a framework for decoding, filtering, and encoding events.

   This model works very nicely for the initial collection of logs, where data is necessarily spread across a large number of hosts and may only be accessible by an agent running on each host. However, it does not extend well to many other use cases. For example, these systems do not handle integration with batch systems like HDFS well because they are designed around the expectation that processing of each event will be handled promptly, with most failure handling left to the user.

   These systems are also operationally complex for a large data pipeline. Collecting logs requires an agent per server anyway. However, to scale out copying data to systems like Hadoop requires manually managing many independent agent processes across many servers and manually dividing the work between them. Additionally, adding a new task may require reconfiguring upstream tasks as well since there is no standardized storage layer.
2. **ETL for data warehousing**

   Examples: Gobblin, Chukwa, Suro, Morphlines, HIHO

   These systems are trying to bridge the gap from a disparate set of systems to data warehouses, most popularly HDFS. Focusing on data warehouses leads to a common set of patterns in these systems. Most obviously, they focus primarily on batch jobs. In some systems these batches can be made quite small, but they are not designed to achieve the low latency required for stream processing applications. This design is sensible when loading data into a data warehouse, but does not extend to the variety of data replication jobs that are required in a stream data platform.

   Another common feature is a flexible, pluggable data processing pipeline. In the context of ETL for a data warehouse this is a requirement if processing can not be performed earlier in the data pipeline. Data must be converted into a form suitable for long term storage, querying, and analysis before it hits HDFS. However, this greatly complicates these tools -- both their use and implementation -- and requires users to learn how to process data in the ETL framework rather than use other existing tools they might already be familiar with.

   Finally, because of the very specific use case, these systems generally only work with a single sink (HDFS) or a small set of sinks that are very similar (e.g. HDFS and S3). Again, given the specific application domain this is a reasonable design tradeoff, but limits the use of these systems for other types of data copying jobs.
3. **Data pipelines management**

Expand Content

Examples: NiFi

These systems try to make building a data pipeline as easy as possible. Instead of focusing on configuration and execution of individual jobs that copy data between two systems, they give the operator a view of the entire pipeline and focus on ease of use through a GUI. At their core, they require the same basic components (individual copy tasks, data sources and sinks, intermediate queues, etc.), but the default view for these systems is of the entire pipeline.

Because these systems "own" the data pipeline as a whole, they may not work well at the scale of an entire organization where different teams may need to control different parts of the pipeline. A large organization may have many mini data pipelines managed in a tool like this instead of one large data pipeline. However, this holistic view allows for better global handling of processing errors and enables integrated monitoring and metrics for the entire data pipeline.

Additionally, these systems are designed around generic processor components which can be connected arbitrarily to create the data pipeline. This offers great flexibility, but provides few guarantees for reliability and delivery semantics. These systems often support queuing between stages, but they usually provides limited fault tolerance, much like the log and metric processing systems.

With the benefits and drawbacks of each of these classes of related systems in mind, Kafka Connect is designed to have the following key properties:

- **Broad copying by default** -- Quickly define connectors that copy vast quantities of data between systems to keep configuration overhead to a minimum. The default unit of work should be an entire database, even if it is also possible to define connectors that copy individual tables.
- **Streaming and batch** -- Support copying to and from both streaming and batch-oriented systems.
- **Scales to the application** -- Scale down to a single process running one connector in development, testing or a small production environment, and scale up to an organization-wide service for copying data between a wide variety of large scale systems.
- **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 functionality. Correspondingly, data copied by Kafka Connect must integrate well with stream processing frameworks.
- **Parallel** -- Parallelism should be included in the core abstractions, providing a clear avenue for the framework to provide automatic scalability.
- **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 and quickly get data flowing between systems. 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.

Architecture

Kafka Connect has three major models in its design:

- **Connector model**: A connector is defined by specifying a `Connector` class and configuration options to control what data is copied and how to format it. Each `Connector` instance is responsible for defining and updating a set of `Tasks` that actually copy the data. Kafka Connect manages the `Tasks`; the `Connector` is only responsible for generating the set of `Tasks` and indicating to the framework when they need to be updated. `Source` and `Sink` `Connectors` / `Tasks` are distinguished in the API to ensure the simplest possible API for both.
- **Worker model**: A Kafka Connect cluster consists of a set of `Worker` processes that are containers that execute `Connectors` and `Tasks`. `Workers` automatically coordinate with each other to distribute work and provide scalability and fault tolerance. The `Workers` will distribute work among any available processes, but are not responsible for management of the processes; any process management strategy can be used for `Workers` (e.g. cluster management tools like YARN or Mesos, configuration management tools like Chef or Puppet, or direct management of process lifecycles).
- **Data model**: Connectors copy streams of messages from a partitioned input stream to a partitioned output stream, where at least one of the input or output is *always* Kafka. Each of these streams is an ordered set messages where each message has an associated offset. The format and semantics of these offsets are defined by the Connector to support integration with a wide variety of systems; however, to achieve certain delivery semantics in the face of faults requires that offsets are unique within a stream and streams can seek to arbitrary offsets. The message contents are represented by `Connectors` in a serialization-agnostic format, and Kafka Connect supports pluggable `Converters` for storing this data in a variety of serialization formats. Schemas are built-in, allowing important metadata about the format of messages to be propagated through complex data pipelines. However, schema-free data can also be use when a schema is simply unavailable.

The connector model addresses three key user requirements. First, Kafka Connect performs **broad copying by default** by having users define jobs at the level of `Connectors` which then break the job into smaller `Tasks`. This two level scheme strongly encourages connectors to use configurations that encourage copying broad swaths of data since they should have enough inputs to break the job into smaller tasks. It also provides one point of **parallelism** by requiring `Connectors` to immediately consider how their job can be broken down into subtasks, and select an appropriate granularity to do so. Finally, by specializing source and sink interfaces, Kafka Connect provides an **accessible connector API** that makes it very easy to implement connectors for a variety of systems.

The worker model allows Kafka Connect to **scale to the application**. It can run scaled down to a single worker process that also acts as its own coordinator, or in clustered mode where connectors and tasks are dynamically scheduled on workers. However, it assumes very little about the *process management* of the workers, so it can easily run on a variety of cluster managers or using traditional service supervision. This architecture allows scaling up and down, but Kafka Connect's implementation also adds utilities to support both modes well. The REST interface for managing and monitoring jobs makes it easy to run Kafka Connect as an organization-wide service that runs jobs for many users. Command line utilities specialized for ad hoc jobs make it easy to get up and running in a development environment, for testing, or in production environments where an agent-based approach is required.

The data model addresses the remaining requirements. Many of the benefits come from coupling tightly with Kafka. Kafka serves as a natural buffer for both **streaming and batch** systems, removing much of the burden of managing data and ensuring delivery from connector developers. Additionally, by always requiring Kafka as one of the endpoints, the larger data pipeline can leverage the many tools that integrate well with Kafka. This allows Kafka Connect to **focus only on copying data** because a variety of stream processing tools are available to further process the data, which keeps Kafka Connect simple, both conceptually and in its implementation. This differs greatly from other systems where ETL must occur before hitting a sink. In contrast, Kafka Connect can bookend an ETL process, leaving any transformation to tools specifically designed for that purpose. Finally, Kafka includes partitions in its core abstraction, providing another point of **parallelism**.

Internal Connect Offsets

As connectors run, Kafka Connect tracks **offsets** for each one so that connectors can resume from their previous position in the event of failures or graceful restarts for maintenance. These offsets are similar to Kafka's offsets in that they track the current position in the stream of data being copied and because each connector may need to track many offsets for different **partitions** of the stream. However, they are different because the format of the offset is defined by the system data is being loaded from and therefore may not simply be a `long` as they are for Kafka topics. For example, when loading data from a database, the offset might be a transaction ID that identifies a position in the database changelog.

Users generally do not need to worry about the format of offsets, especially since they differ from connector to connector. However, Kafka Connect does require persistent storage for configuration, offset, and status updates to ensure it can recover from faults, and although Kafka Connect will attempt to create the necessary topics when they don't yet exist, users may choose to manually create the topics used for this storage. These settings, which depend on the way you decide to run Kafka Connect, are discussed in the next section.

© Copyright 2019, Confluent, Inc. Privacy Policy | Terms & Conditions. Apache, Apache Kafka, Kafka and the Kafka logo are trademarks of the Apache Software Foundation. All other trademarks, servicemarks, and copyrights are the property of their respective owners.

Please report any inaccuracies on this page or suggest an edit.

13 Votes

Last updated on Oct 17, 2019.