Hadoop & Big Data

Our Customers

FAQs

Blog

Accumulo (1)

Avro (17)

Bigtop (6)

Books (12)

Careers (15)

CDH (158)

Cloud (25)

Cloudera Labs (10)

Cloudera Life (7)

Cloudera Manager (77)

Community (221)

Data Ingestion (22)

Apache Kafka for Beginners

by Gwen Shapira & Jeff Holoman September 12, 2014 **6 comments**

When used in the right way and for the right use case, Kafka has unique attributes that make it a highly attractive option for data integration.

Apache Kafka is creating a lot of buzz these days. While LinkedIn, where Kafka was founded, is the most well known user, there are **many companies** successfully using this technology.

So now that the word is out, it seems the world wants to know: What does it do? Why does everyone want to use it? How is it better than existing solutions? Do the benefits justify replacing existing systems and infrastructure?

In this post, we'll try to answers those questions. We'll begin by briefly introducing Kafka, and then demonstrate some of Kafka's unique features by walking through an example scenario. We'll also cover some additional use cases and also compare Kafka to existing solutions.

What is Kafka?

Kafka is one of those systems that is very simple to describe at a high level, but has an incredible depth of technical detail when you dig deeper. The **Kafka documentation** does an excellent job of explaining the many design and implementation subtleties in the system, so we will not attempt to explain them all here. In summary, *Kafka is a distributed publish-subscribe messaging system that is designed to be fast, scalable, and durable*.

Like many publish-subscribe messaging systems, Kafka maintains feeds of messages in topics. Producers write data to topics and consumers read from topics. Since Kafka is a distributed system, topics are partitioned and replicated across multiple nodes.

Messages are simply byte arrays and the developers can use them to store any object in any format – with String, JSON, and Avro the most common. It is possible to attach a key to each message, in which case the producer guarantees that all messages with the same key will arrive to the same partition. When consuming from a topic, it is possible to configure a consumer group with multiple consumers. Each consumer in a consumer group will read messages from a unique subset of partitions in each topic they subscribe to, so each message is delivered to one consumer in the group, and all messages with the same key arrive at the same consumer.

What makes Kafka unique is that Kafka treats each topic partition as a *log* (an ordered set of messages). Each message in a partition is assigned a unique offset. Kafka does not attempt to track which messages were read by each consumer and only retain unread messages; rather, Kafka retains all messages for a set amount of time, and consumers are responsible to track their location in each log. Consequently, Kafka can support a large number of consumers and retain large amounts of data with very little overhead.

Next, let's look at how Kafka's unique properties are applied in a specific use case.

Kafka at Work

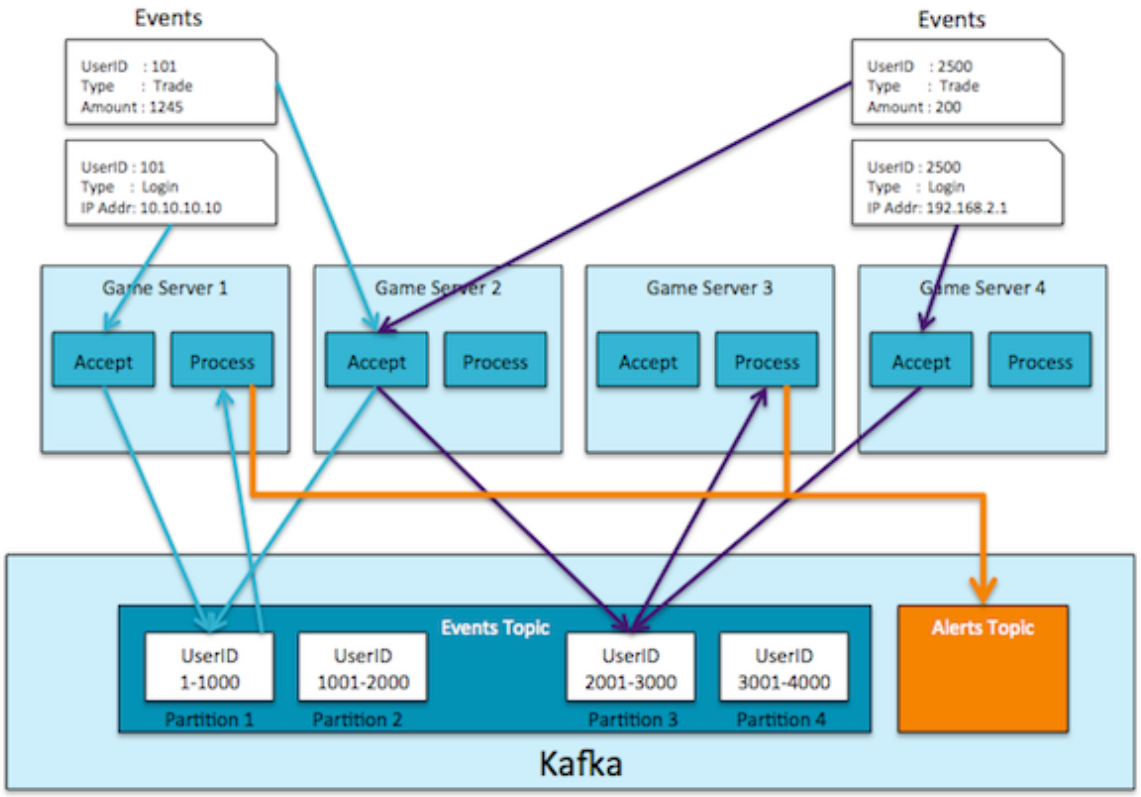
Suppose we are developing a massive multiplayer online game. In these games, players cooperate and compete with each other in a virtual world. Often players trade with each other, exchanging game items and money, so as game developers it is important to make sure players don't cheat: Trades will be flagged if the trade amount is significantly larger than normal for the player and if the IP the player is logged in with is different than the IP used for the last 20 games. In addition to flagging trades in real-time, we also want to load the data to Apache Hadoop, where our data scientists can use it to train and test new algorithms.

For the real-time event flagging, it will be best if we can reach the decision quickly based on data that is cached on the game server memory, at least for our most active players. Our system has multiple game servers and the data set that includes the last 20 logins and last 20 trades for each player can fit in the memory we have, if we partition it between our game servers.

Our game servers have to perform two distinct roles: The first is to accept and propagate user actions and the second to process trade information in real time and flag suspicious events. To perform the second role effectively, we want the whole history of trade events for each user to reside in memory of a single server. This means we have to pass messages between the servers, since the server that accepts the user action may not have his trade history. To keep the roles loosely coupled, we use Kafka to pass messages between the servers, as you'll see below.

Kafka has several features that make it a good fit for our requirements: *scalability*, *data partitioning*, *low latency*, and the *ability to handle large number of diverse consumers*. We have configured Kafka with a single topic for logins and trades. The reason we need a single topic is to make sure that trades arrive to our system after we already have information about the login (so we can make sure the gamer logged in from his usual IP). Kafka maintains order within a topic, but not between topics.

When a user logs in or makes a trade, the accepting server immediately sends the event into Kafka. We send messages with the user id as the key, and the event as the value. This guarantees that all trades and logins from the same user arrive to the same Kafka partition. Each event processing server runs a Kafka consumer, each of which is configured to be part of the same group—this way, each server reads data from few Kafka partitions, and all the data about a particular user arrives to the same event processing server (which can be different from the accepting server). When the event-processing server reads a user trade from Kafka, it adds the event to the user's event history it caches in local memory. Then it can access the user's event history from the local cache and flag suspicious events without additional network or disk overhead.



It's important to note that we create a partition per event-processing server, or per core on the event-processing servers for a multi-threaded approach. (Keep in mind that Kafka was mostly tested with fewer than 10,000 partitions for all the topics in the cluster in total, and therefore we do not attempt to create a partition per user.)

This may sound like a circuitous way to handle an event: Send it from the game server to Kafka, read it from another game server and only then process it. However, this design decouples the two roles and allows us to manage capacity for each role as required. In addition, the approach does not add significantly to the timeline as Kafka is designed for high throughput and low latency; even a small three-node cluster can process close to a million events per second with an average latency of 3ms.

When the server flags an event as suspicious, it sends the flagged event into a new Kafka topic—for example, Alerts—where alert servers and dashboards pick it up. Meanwhile, a separate process reads data from the Events and Alerts topics and writes them to Hadoop for further analysis.

Because Kafka does not track acknowledgements and messages per consumer it can handle many thousands of consumers with very little performance impact. Kafka even handles batch consumers—processes that wake up once

an hour to consume all new messages from a queue—without affecting system throughput or latency.

Additional Use Cases

As this simple example demonstrates, Kafka works well as a traditional message broker as well as a method of ingesting events into Hadoop.

Here are some other common uses for Kafka:

- **Website activity tracking**: The web application sends events such as page views and searches Kafka, where they become available for real-time processing, dashboards and offline analytics in Hadoop
- **Operational metrics**: Alerting and reporting on operational metrics. One particularly fun example is having Kafka producers and consumers occasionally publish their message counts to a special Kafka topic; a service can be used to compare counts and alert if data loss occurs.
- **Log aggregation**: Kafka can be used across an organization to collect logs from multiple services and make them available in standard format to multiple consumers, including Hadoop and Apache Solr.
- **Stream processing**: A framework such as Spark Streaming reads data from a topic, processes it and writes processed data to a new topic where it becomes available for users and applications. Kafka's strong durability is also very useful in the context of stream processing.

Other systems serve many of those use cases, but none of them do them all. ActiveMQ and RabbitMQ are very popular message broker systems, and **Apache Flume** is traditionally used to ingest events, logs, and metrics into Hadoop.

Kafka and Its Alternatives

We can't speak much about message brokers, but data ingest for Hadoop is a problem we understand very well.

First, it is interesting to note that Kafka started out as a way to make data ingest to Hadoop easier. When there are multiple data sources and destinations involved, writing a separate data pipeline for each source and destination pairing quickly evolves to an unmaintainable mess. Kafka helped LinkedIn standardize the data pipelines and allowed getting data out of each system once and into each system once, significantly reducing the pipeline complexity and cost of operation.

Jay Kreps, Kafka's architect at LinkedIn, describes this familiar problem well in a **blog post**:

My own involvement in this started around 2008 after we had shipped our key-value store. My next project was to try to get a working Hadoop setup going, and move some of our recommendation processes there. Having little experience in this area, we naturally budgeted a few weeks for getting data in and out, and the rest of our time for implementing fancy prediction algorithms. So began a long slog.

Diffs versus Flume

There is significant overlap in the functions of Flume and Kafka. Here are some considerations when evaluating the

two systems.

- Kafka is very much a general-purpose system. You can have many producers and many consumers sharing multiple topics. In contrast, Flume is a special-purpose tool designed to send data to HDFS and HBase. It has specific optimizations for HDFS and it integrates with Hadoop's security. As a result, Cloudera recommends using Kafka if the data will be consumed by multiple applications, and Flume if the data is designated for Hadoop.
- Those of you familiar with Flume know that Flume has many built-in sources and sinks. Kafka, however, has a significantly smaller producer and consumer ecosystem, and it is not well supported by the Kafka community. Hopefully this situation will improve in the future, but for now: Use Kafka if you are prepared to code your own producers and consumers. Use Flume if the existing Flume sources and sinks match your requirements and you prefer a system that can be set up without any development.
- Flume can process data in-flight using interceptors. These can be very useful for data masking or filtering. Kafka requires an external stream processing system for that.
- Both Kafka and Flume are reliable systems that with proper configuration can guarantee zero data loss. However, Flume does not replicate events. As a result, even when using the reliable file channel, if a node with Flume agent crashes, you will lose access to the events in the channel until you recover the disks. Use Kafka if you need an ingest pipeline with very high availability.
- Flume and Kafka can work quite well together. If your design requires streaming data from Kafka to Hadoop, using a Flume agent with Kafka source to read the data makes sense: You don't have to implement your own consumer, you get all the benefits of Flume's integration with HDFS and HBase, you have Cloudera Manager monitoring the consumer and you can even add an interceptor and do some stream processing on the way.

Conclusion

As you can see, Kafka has a unique design that makes it very useful for solving a wide range of architectural challenges. It is important to make sure you use the right approach for your use case and use it correctly to ensure high throughput, low latency, high availability, and no loss of data.

Gwen Shapira is a Software Engineer at Cloudera, and a Kafka contributor. Jeff Holoman is a Systems Engineer at Cloudera.

Filed under:

Flume

Kafka

6 Responses

BPS / SEPTEMBER 12, 2014 / 12:23 PM

awesome article, especially the diff between Flume/Kafka

You might want to fix the link to Jay Krep's blog from

https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

> **JUSTIN KESTELYN (@KESTELYN)** / SEPTEMBER 12, 2014 / 1:24 PM
>
> Thanks for the heads-up.

LAURIE / SEPTEMBER 23, 2014 / 8:38 AM

Very helpful information especially putting into context using a specific use case. I also liked the description of the differences between Flume and Kafka.

GOPAL PATWA / SEPTEMBER 28, 2014 / 6:25 PM

Very good article, I was looking for such simple explanation for our internal team.

as others have said, good to know how it differ from Flume, since we already use Flume for HDFS.

BD / OCTOBER 02, 2014 / 11:48 AM

Thanks for the overview! For those who are just starting out their pipeline, it would be great to know how Kafka->Hadoop pipeline works using Camus with Parquet format.

TAUFIQUE HUSSAIN / AUGUST 05, 2015 / 9:03 PM

"Kafka maintains order within a topic, but not between topics." – I disagree with this. Kafka maintains order within a partition of a topic, not between different partitions of a topic. Look at the consumer subsection of documentation.
http://kafka.apache.org/documentation.html#introduction

Leave a comment

Name REQUIRED

Email REQUIRED

(WILL NOT BE PUBLISHED)

Website

Comment

Leave Comment

Prove you're human! *

8 − five =

Cloudera, Inc.
1001 Page Mill Road Bldg 2
Palo Alto, CA 94304

www.cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488