Chapter 1. Meet Kafka

The enterprise is powered by data. We take information in, analyze it, manipulate it, and create more as output. Every application creates data, whether it is log messages, metrics, user activity, outgoing messages, or something else. Every byte of data has a story to tell, something of import that will inform the next thing to be done. In order to know what that is, we need to get the data from where it is created to where it can be analyzed. We then need to get the results back to where they can be executed on.

The faster we can do this, the more agile and responsive our organizations can be. The less effort we spend on moving data around, the more we can focus on the core business at hand. This is why the pipeline is a critical component in the data-driven enterprise. How we move the data becomes nearly as important as the data itself.

> *Any time scientists disagree, it's because we have insufficient data. Then we can agree on what kind of data to get; we get the data; and the data solves the problem. Either I'm right, or you're right, or we're both wrong. And we move on.*
>
> —Neil deGrasse Tyson

Publish / Subscribe Messaging

Before discussing the specifics of Apache Kafka, it is important for us to understand the concept of publish-subscribe messaging and why it is important. Publish-subscribe messaging is a pattern that is characterized by the sender (publisher) of a piece of data (message) not specifically directing it to a receiver. Instead, the publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages. Pub/sub systems often have a broker, a central point where messages are published, to facilitate this.

How It Starts

Many use cases for publish-subscribe start out the same way: with a simple message queue or inter-process communication channel. For example, you write an application that needs to send monitoring information somewhere, so you write in a direct connection from your application to an app that displays your metrics on a dashboard, and push metrics over that connection, as seen in Figure 1-1.

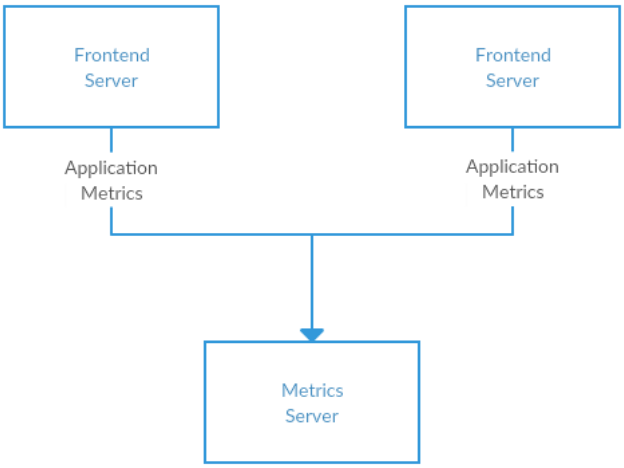

Figure 1-1. A single, direct metrics publisher

Before long, you decide you would like to analyze your metrics over a longer term, and that doesn't work well in the dashboard. You start a new service that can receive metrics, store them, and analyze them. In order to support this, you modify your application to write metrics to both systems. By now you have three more applications that are generating metrics, and they all make the same connections to these two services. Your coworker thinks it would be a good idea to do active polling of the services for alerting as well, so you add a server on each of the applications to provide metrics on request. After a while, you have more applications that are using those servers to get individual metrics and use them for various purposes. This architecture can look much like Figure 1-2, with connections that are even harder to trace.

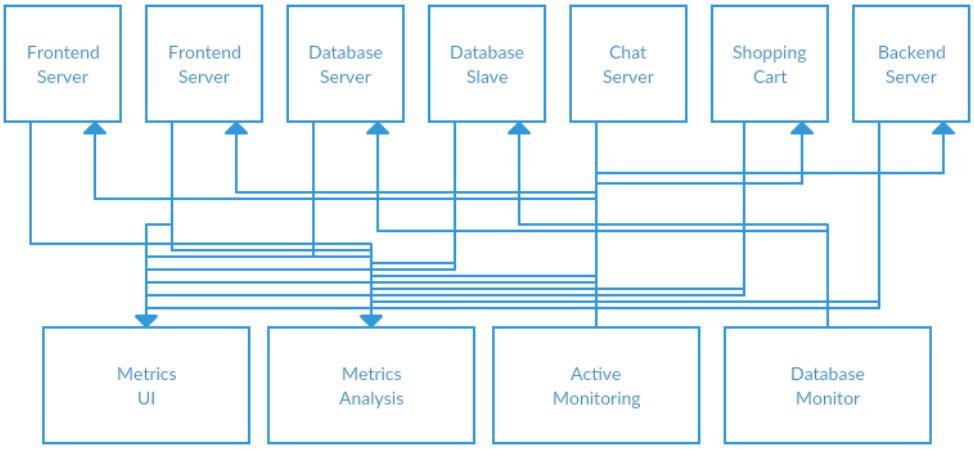

Figure 1-2. Many metrics publishers, using direct connections

The technical debt built up here is obvious, and you decide to pay some of it back. You set up a single application that receives metrics from all the applications out there, and provides a server to query those metrics for any system that needs them. This reduces the complexity of the architecture to something similar to Figure 1-3. Congratulations, you have built a publish-subscribe messaging system!

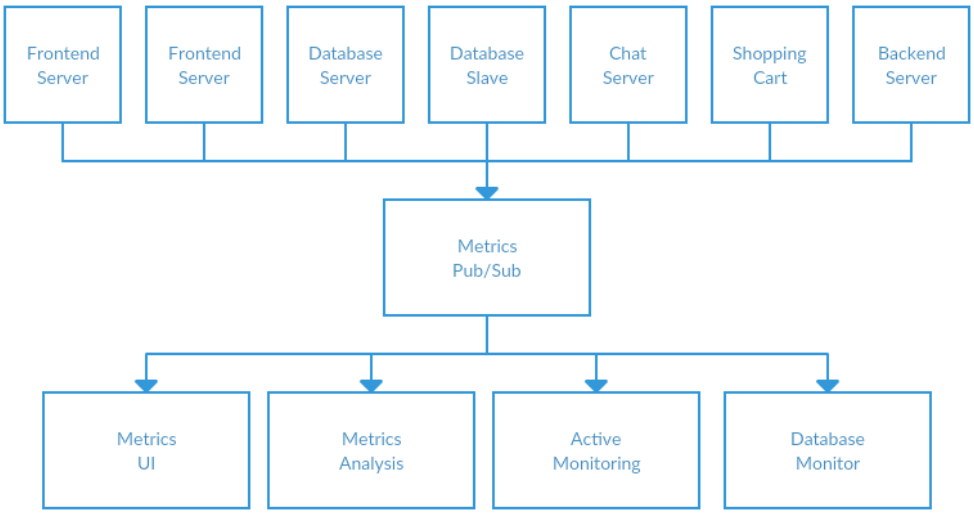

Figure 1-3. A metrics publish/subscribe system

Individual Queue Systems

At the same time that you have been waging this war with metrics, one of your coworkers has been doing similar work with log messages. Another has been working on tracking user behavior on the front-end website and providing that information to developers who are working on machine learning, as well as creating some reports for management. You have all followed a similar path of building out systems that decouple the publishers of the information from the subscribers to that information. Figure 1-4 shows such an infrastructure, with three separate pub/sub systems.

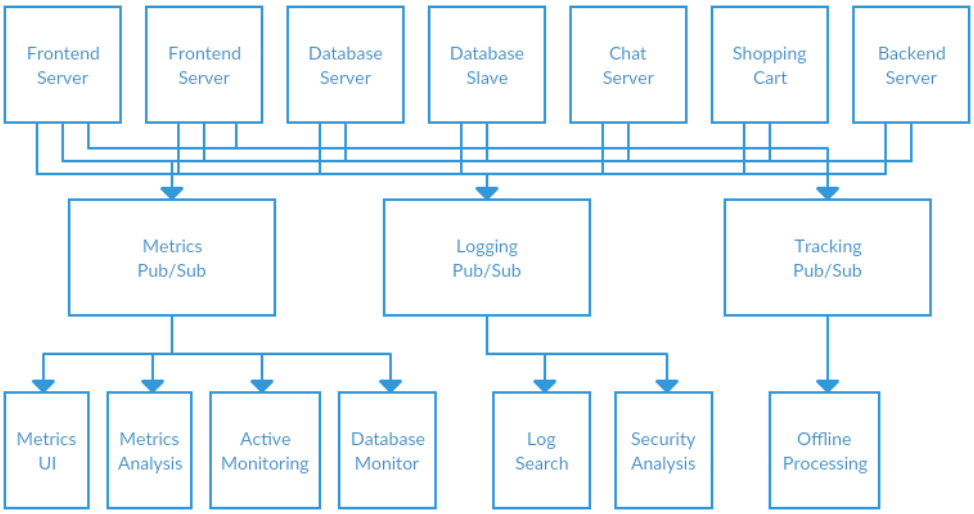

Figure 1-4. Multiple publish/subscribe systems

This is certainly a lot better than utilizing point to point connections (as in Figure 1-2), but there is a lot of duplication. Your company is maintaining multiple systems for queuing data, all of which have their own individual bugs and limitations. You also know that there will be more use cases for messaging coming soon. What you would like to have is a single centralized system that allows for publishing of generic types of data, and that will grow as your business grows.

Enter Kafka

Apache Kafka is a publish/subscribe messaging system designed to solve this problem. It is often described as a "distributed commit log" or more recently a "distributing streaming platform". A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consistently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Messages and Batches

The unit of data within Kafka is called a *message*. If you are approaching Kafka from a database background, you can think of this as similar to a *row* or a *record*. A message is simply an array of bytes, as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka. Messages can have an optional bit of metadata which is referred to as a key. The key is also a byte array, and as with the message, has no specific meaning to Kafka. Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to treat partitions as a hash ring, and assure that messages with the same key are always written to the same partition. Usage of keys is discussed more thoroughly in Chapter 3.

For efficiency, messages are written into Kafka in batches. A *batch* is just a collection of messages, all of which are being produced to the same topic and partition. An individual round trip across the network for each message would result in excessive overhead, and collecting messages together into a batch reduces this. This, of course, presents a tradeoff between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, which provides for more efficient data transfer and storage at the cost of some processing power.

Schemas

While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure be imposed on the message content so that it can be easily understood. There are many options available for message *schema*, depending on your application's individual needs. Simplistic systems, such as Javascript Object Notation (JSON) and Extensible Markup Language (XML), are easy to use and human readable. However they lack features such as robust type handling and compatibility between schema versions. Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop. Avro provides a compact serialization format, schemas that are separate from the message payloads and that do not require generated code when they change, as well as strong data typing and schema evolution, with both backwards and forwards compatibility.

A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled. When these tasks are tightly coupled, applications which subscribe to messages must be updated to handle the new data format, in parallel with the old format. Only then can the applications that publish the messages be updated to utilize the new format. New applications that wish to use data must be coupled with the publishers, leading to a high-touch process for developers. By using well-defined schemas, and storing them in a common repository, the messages in Kafka can be understood without coordination. Schemas and serialization are covered in more detail in Chapter 3.

Topics and Partitions

Messages in Kafka are categorized into *topics*. The closest analogy for a topic is a database table, or a folder in a filesystem. Topics are additionally broken down into a number of *partitions*. Going back to the "commit log" description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic generally has multiple partitions, there is no guarantee of time-ordering of messages across the entire topic, just within a single partition. Figure 1-5 shows a topic with 4 partitions, with writes being appended to the end of each one. Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide for performance far beyond the ability of a single server.

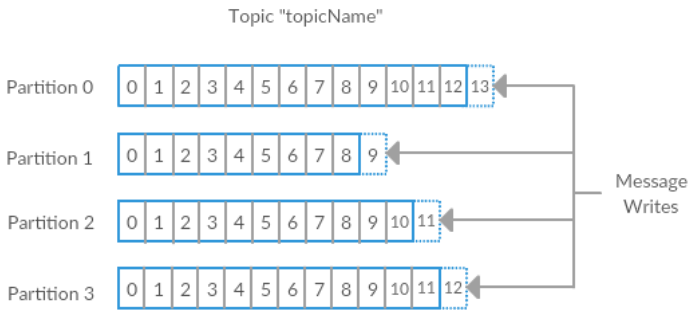

Figure 1-5. Representation of a topic with multiple partitions

The term *stream* is often used when discussing data within systems like Kafka. Most often, a stream is considered to be a single topic of data, regardless of the number of partitions. This represents a single stream of data moving from the producers to the consumers. This way of referring to messages is most common when discussing stream processing, which is when frameworks, some of which are Kafka Streams, Apache Samza, and Storm, operate on the messages in real time. This method of operation can be compared to the way offline frameworks, namely Hadoop, are designed to work on bulk data at a later time. An overview of stream processing is provided in Chapter 11.

Producers and Consumers

Kafka clients are users of the system, and there are two basic types: producers and consumers. There are also advanced client APIs - Kafka Connect API for data integration and Kafka Streams for stream processing. The advanced clients use producers and consumers as building blocks and provide higher level functionality on top.

Producers create new messages. In other publish/subscribe systems, these may be called *publishers* or *writers*. In general, a message will be produced to a specific topic. By default, the producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly. In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition. The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions. Producers are covered in more detail in Chapter 3.

Consumers read messages. In other publish/subscribe systems, these clients may be called *subscribers* or *readers*. The consumer subscribes to one or more topics and reads the messages in the order they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the *offset* of messages. The offset is another bit of metadata, an integer value that continually increases, that Kafka adds to each message as it is produced. Each message within a given partition has a unique offset. By storing the offset of the last consumed message for each partition, either in Zookeeper or in Kafka itself, a consumer can stop and restart without losing its place.

Consumers work as part of a *consumer group*. This is one or more consumers that work together to consume a topic. The group assures that each partition is only consumed by one member. In Figure 1-6, there are three consumers in a single group consuming a topic. Two of the consumers are working from one partition each, while the third consumer is working from two partitions. The mapping of a consumer to a partition is often called *ownership* of the partition by the consumer.

In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the missing member. Consumers and consumer groups are discussed in more detail in Chapter 4.

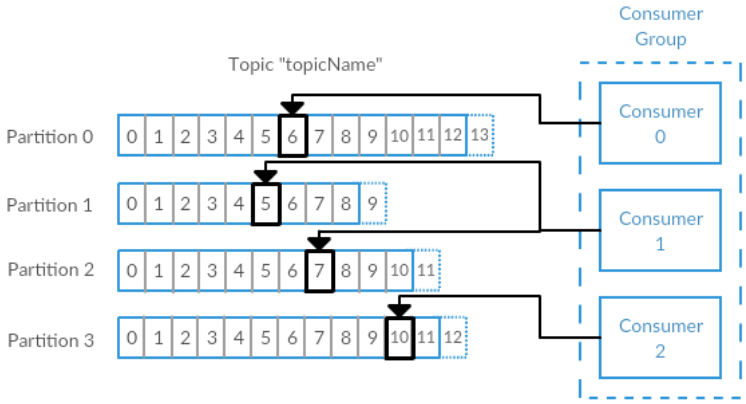

Figure 1-6. A consumer group reading from a topic

Brokers and Clusters

A single Kafka server is called a *broker*. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been committed to disk. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.

Kafka brokers are designed to operate as part of a *cluster*. Within a cluster of brokers, one will also function as the cluster *controller* (elected automatically from the live members of the cluster). The controller is responsible for administrative operations, including assigning partitions to brokers and monitoring for broker failures. A partition is owned by a single broker in the cluster, and that broker is called the *leader* for the partition. A partition may be assigned to multiple brokers, which will result in the partition being replicated (as in Figure 1-7). This provides redundancy of messages in the partition, such that another broker can take over leadership if there is a broker failure. However, all consumers and producers operating on that partition must connect to the leader. Cluster operations, including partition replication, are covered in detail in Chapter 6.

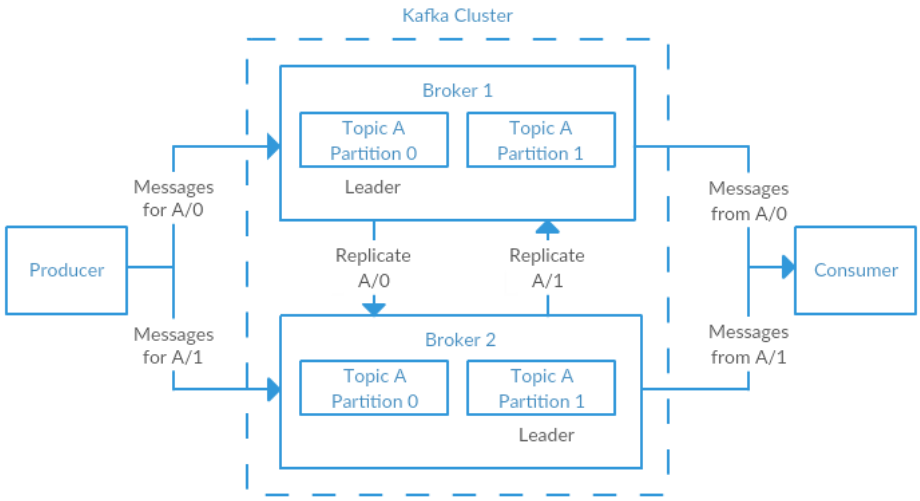

Figure 1-7. Replication of partitions in a cluster

A key feature of Apache Kafka is that of *retention*, or the durable storage of messages for some period of time. Kafka brokers are configured with a default retention setting for topics, either retaining messages for some period of time (e.g. 7 days) or until the topic reaches a certain size in bytes (e.g. 1 gigabyte). Once these limits are reached, messages are expired and deleted so that the retention

configuration is a minimum amount of data available at any time. Individual topics can also be configured with their own retention settings, so messages can be stored for only as long as they are useful. For example, a tracking topic may be retained for several days, while application metrics may be retained for only a few hours. Topics may also be configured as *log compacted*, which means that Kafka will retain only the last message produced with a specific key. This can be useful for changelog-type data, where only the last update is interesting.

Multiple Clusters

As Kafka deployments grow, it is often advantageous to have multiple clusters. There are several reasons why this can be useful:

- Segregation of types of data
- Isolation for security requirements
- Multiple datacenters (disaster recovery)

When working with multiple datacenters, in particular, it is usually required that messages be copied between them. In this way, online applications can have access to user activity at both sites. Or monitoring data can be collected from many sites into a single central location where the analysis and alerting systems are hosted. The replication mechanisms within the Kafka clusters are designed only to work within a single cluster, not between multiple clusters.

The Kafka project includes a tool called *Mirror Maker* that is used for this purpose. At it's core, Mirror Maker is simply a Kafka consumer and producer, linked together with a queue. Messages are consumed from one Kafka cluster and produced to another. Figure 1-8 shows an example of an architecture that uses Mirror Maker, aggregating messages from two "Local" clusters into an "Aggregate" cluster, and then copying that cluster to other datacenters. The simple nature of the application belies its power in creating sophisticated data pipelines, however. All of these cases will be detailed further in Chapter 7.

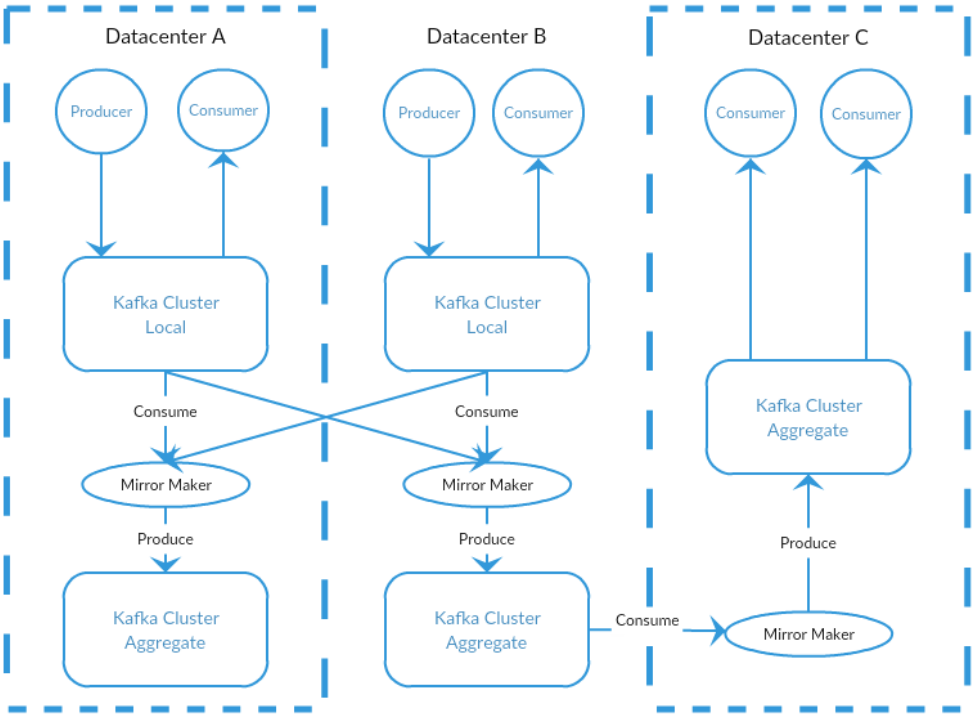

Figure 1-8. Multiple datacenter architecture

Why Kafka?

There are many choices for publish/subscribe messaging systems, so what makes Apache Kafka a good choice?

Multiple Producers

Kafka is able to seamlessly handle multiple producers, whether those clients are using many topics or the same topic. This makes the system ideal for aggregating data from many front end systems and providing the data in a consistent format. For example, a site that serves content to users via a number of microservices can have a single topic for page views which all services can write to using a common format. Consumer applications can then received one unified view of page views for the site without having to coordinate the multiple producer streams.

Multiple Consumers

In addition to multiple producers, Kafka is designed for multiple consumers to read any single stream of messages without interfering with each other. This is in opposition to many queuing systems where once a message is consumed by one client, it is not available to any other client. At the same time, multiple Kafka consumers can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Disk-based Retention

Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time. Messages are committed to disk, and will be stored with configurable retention rules. These options can be selected on a per-topic basis, allowing for different streams of messages to have different amounts of retention depending on what the consumer needs are. Durable retention means that if a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data. It also means that maintenance can be performed on consumers, taking applications offline for a short period of time, with no concern about messages backing up on the producer or getting lost. The consumers can just resume processing where they stopped.

Scalable

Flexible scalability has been designed into Kafka from the start, allowing for the ability to easily handle any amount of data. Users can start with a single broker as a proof of concept, expand to a small development cluster of 3 brokers, and move into production with a larger cluster of tens, or even hundreds, of brokers that grows over time as the data scales up. Expansions can be performed while the cluster is online, with no impact to the availability of the system as a whole. This also means that a cluster of multiple brokers can handle the failure of an individual broker and continue servicing clients. Clusters that need to tolerate more simultaneous failures can be configured with higher replication factors. Replication is discussed in more detail in Chapter 6.

High Performance

All of these features come together to make Apache Kafka a publish/subscribe messaging system with excellent performance characteristics under high load. Producers, consumers, and brokers can all be scaled out to handle very large message streams with ease. This can be done while still providing sub-second message latency from producing a message to availability to consumers.

The Data Ecosystem

Many applications participate in the environments we build for data processing. We have defined inputs, applications that create data or otherwise introduce it to the system. We have defined outputs, whether that is metrics, reports, or other data products. We create loops, with some components reading data from the system, performing operations on it, and then introucing it back into the data infrastructure to be used elsewhere. This is done for numerous types of data, with each having unique qualities of content, size, and usage.

Apache Kafka provides the circulatory system for the data ecosystem, as in Figure 1-9. It carries messages between the various members of the infrastructure, providing a consistent interface for all clients. When coupled with a system to provide message schemas, producers and consumers no longer require a tight coupling, or direct connections of any sort. Components can be added and removed as business cases are created and dissolved, while producers do not need to be concerned about who is using the data, or how many consuming applications there are.

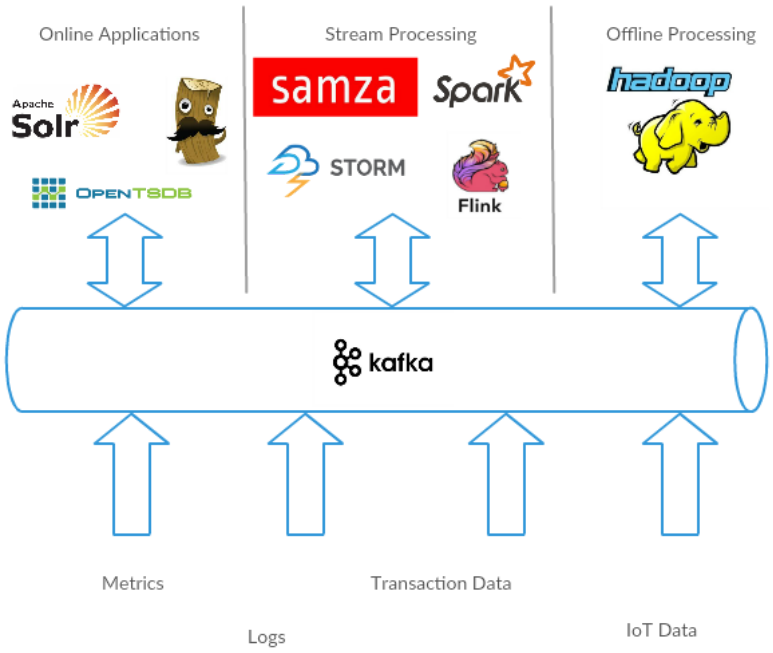

Figure 1-9. A Big data ecosystem

Use Cases

ACTIVITY TRACKING

The original use case for Kafka is that of user activity tracking. A website's users interact with front end applications, which generate messages regarding actions the user is taking. This can be passive information, such as page views and click tracking, or it can be more complex actions, such as adding information to their user profile. The messages are published to one or more topics, which are then consumed by applications on the back end. In doing so, we generate reports, feed machine learning systems, and update search results, among myriad other possible uses.

MESSAGING

Another basic use for Kafka is messaging. This is where applications need to send notifications (such as email messages) to users. Those components can produce messages without needing to be concerned about formatting or how the messages will actually be sent. A common application can then read all the messages to be sent and perform the work of formatting (also known as decorating) the messages and selecting how to send them. By using a common component, not only is there no need to duplicate functionality in multiple applications, there is also the ability to do interesting transformations, such as aggregation of multiple messages into a single notification, that would not be otherwise possible.

METRICS AND LOGGING

Kafka is also ideal for the collection of application and system metrics and logs. This is a use where the ability to have multiple producers of the same type of message shines. Applications publish metrics about their operation on a regular basis to a Kafka topic, and those metrics can be consumed by systems for monitoring and alerting. They can also be used in an offline system like Hadoop to perform longer term analysis, such as year over year growth projections. Log messages can be published in the same way, and can be routed to dedicated log search systems like Elastisearch or security anlysis applications. Kafka provides the added benefit that when the destination system needs to change (for example, it's time to update the log storage system), there is no need to alter the front end applications or the means of aggregation.

COMMIT LOG

As Kafka is based on the concept of a commit log, utilizing Kafka in this way is a natural use. Database changes can be published to Kafka and applications can monitor this stream to receive live updates as they happen. This changelog stream can also be used for replicating database updates to a remote system, or for consolidating changes from multiple applications into a single database view. Durable retention is useful here for providing a buffer for the changelog, meaning it can be replayed in the event of a failure of the consuming applications. Alternately, log compacted topics can be used to provide longer retention by only retaining a single change per key.

STREAM PROCESSING

Another area that provides numerous types of applications is stream processing. This can be thought of as providing the same functionality that map/reduce processing does in Hadoop, but it operates on a data stream in real time, where Hadoop usually relies on aggregation of data over a longer time frame, either hours or days, and then performing batch processing on that data. Stream frameworks allow users to write small applications to operate on Kafka messages, performing tasks such as counting metrics, partitioning messages for efficient processing by other applications, or transforming messages using data from multiple sources. Stream processing is covered separate from other case studies in Chapter 11.

The Origin Story

Kafka was born from necessity to solve the data pipeline problem at LinkedIn. It was designed to provide a high-performance messaging system which could handle many types of data, and provide for the availability of clean, structured data about user activity and system metrics in real time.

> *Data really powers everything that we do.*
>
> —Jeff Weiner, CEO of LinkedIn

LinkedIn's Problem

As described at the beginning of this chapter, LinkedIn had a system for collecting system and application metrics that used custom collectors and open source tools for storing and presenting the data internally. In addition to traditional metrics, such as CPU usage and application performance, there was a sophisticated request tracing feature that used the monitoring system and could provide introspection into how a single user request propagated through internal applications. The monitoring system had many faults, however. This included metric collection based on polling, large intervals between metrics, and no self-service capabilities. The system was high-touch, requiring human intervention for most simple tasks, and inconsistent, with differing metric names for the same measurement across different systems.

At the same time, there was a system created for collecting user activity tracking information. This was an HTTP service that front-end servers would connection to periodically and publish a batch of messages (in XML format). These batches were then moved to offline processing, which is where the files were parsed and collated. This system, as well, had many failings. The XML formatting was not consistent, and parsing it was computationally expensive. Changing the type of activity created required a significant amount of coordinated work between front-ends and offline processing. Even then, the system would be broken constantly with changing schemas. Tracking was built on hourly batching, so it could not be used in real-time for any purpose.

Monitoring and user activity tracking could not use the same back-end service. The monitoring service was too clunky, the data format was not oriented for activity tracking, and the polling model would not work. At the same time, the tracking service was too fragile to use for metrics, and the batch-oriented processing was not the right model for real-time monitoring and alerting. However, the data shared many traits, and correlation of the information (such as how specific types of user activity affected application performance) was highly desirable. A drop in specific types of user activity could indicate problems with the application that services it, but hours of delay in processing activity batches meant a slow response to these types of issues.

At first, existing off-the-shelf open source solutions were thoroughly investigated to find a new system that would provide real-time access to the data and scale out to handle the amount of message traffic needed. Prototype systems were set up with ActiveMQ, but at the time it could not handle the scale. It was also a fragile solution in the way LinkedIn needed to use it, hitting many bugs that would cause the brokers to pause. This would back up connections to clients and could interfere with the ability of the applications to serve requests to users. The decision was made to move forward with a custom infrastructure for the data pipeline.

The Birth of Kafka

The development team at LinkedIn was led by Jay Kreps, a principal software engineer who was previously responsible for the development and open source release of Voldemort, a distributed key-value storage system. The initial team also included Neha Narkhede and was quickly joined by Jun Rao. Together they set out to create a messaging system that would meet the needs of both systems and scale for the future. The primary goals were:

- Decouple the producers and consumers by using a push-pull model
- Provide persistence for message data within the messaging system to allow multiple consumers
- Optimize for high throughput of messages
- Allow for horizontal scaling of the system to grow as the data streams grow

The result was a publish/subscribe messaging system that had an interface typical of messaging systems, but a storage layer more like a log aggregation system. Combined with the adoption of Apache Avro for message serialization, this system was effective for handling both metrics and user activity tracking at a scale of billions of messages per day. Over time, LinkedIn's usage has grown to in excess of one trillion messages produced (as of August 2015), and over a petabyte of data consumed daily.

Open Source

Kafka was released as an open source project on GitHub in late 2010. As it started to gain attention in the open source community, it was proposed and accepted as an Apache Software Foundation incubator project in July of 2011. Apache Kafka graduated from the incubator in October of 2012. Since that time, it has continued to have active development from LinkedIn, as well as gathering a robust community of contributors and committers outside of LinkedIn. As a result, Kafka is now used in some of the largest data pipelines at many organizations. In the fall of 2014, Jay Kreps, Neha Narkhede, and Jun Rao left LinkedIn to found Confluent, a company centered around providing development, enterprise support, and training for Apache Kafka. The two companies, along with ever-growing contributions from others in the open source community, continue to develop and maintain Kafka, making it the first choice for big data pipelines.

The Name

A frequent question about the history of Apache Kafka is how the name was selected, and what bearing it has on the application itself. On this topic, Jay Kreps offered the following insight:

> *I thought that since Kafka was a system optimized for writing using a writer's name would make sense. I had taken a lot of lit classes in college and liked Franz Kafka. Plus the name sounded cool for an open source project.*
>
> *So basically there is not much of a relationship.*
>
> —Jay Kreps

Getting Started With Kafka

Now that we know what Kafka is, and have a common terminology to work with, we can move forwards with getting started with setting up Kafka and building your data pipeline. In the next chapter, we will explore Kafka installation and configuration. We will also cover selecting the right hardware to run Kafka on, and some things to keep in mind when moving to production operations.

PREV Preface

NEXT 2. Installing Kafka