Event Streaming

> Event streaming is the digital equivalent of the human body’s central nervous system.

> Event streaming is the practice of capturing data in real – time from event sources like databases, sensors, mobile devices, cloud services & software applications in the form of streams of events;

🡪 storing these event streams durable for later retrieval;

🡪 manipulating, processing & reacting to the event streams in real – time as well as retrospectively;

🡪 & routing the event streams to different destination technologies as needed.

**Event streaming thus ensures a continuous flow & interpretation of data so that the right information is at the right place, at the right time.**

> Event Streaming is applied to a wide variety of use cases across various industries like

* To process payments & financial transactions in real – time, such as in stock exchanges, banks & insurances.
* To track & monitor cars, trucks, & shipments in real – time, such as in logistics & the automotive industry.
* To continuously capture & analyze sensor data from IoT devices or other equipment, such as in factories & wind parks.
* To collect & immediately react to customer interactions & orders, such as in retail, the hotel & travel industry, & mobile applications.
* To monitor patients in hospital care & predict changes in condition to ensure timely treatment in emergencies.
* To connect, store & make available data produced by different divisions of a company.
* To serve as the foundation for data platforms, event – driven architectures & microservices.

**Apache Kafka is an event streaming platform**

> Kafka combines **three key capabilities** so you can implement your use cases for event streaming end – to – end with a single battle – tested solution:

1. To **publish** (write) & **subscribe to** (read) streams of events, including continuous import / export of your data from other systems.
2. To **store** streams of events durably & reliably for as long as you want.
3. To **process** streams of events as they occur or retrospectively.

> All this functionality is provided in a distributed, highly scalable, elastic, fault – tolerant, & secure manner.

> Kafka can be deployed on bare – metal hardware, virtual machines, & containers, & on – premises as well as in the cloud.

> You can choose between self – managing your Kafka environments & using fully managed services offered by a variety of vendors.

**How does Kafka work in a nutshell?**

> Kafka is a distributed system consisting of servers & clients that communicate via a high – performance TCP network protocol.

> **Servers**: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the **brokers**. Other servers run **Kafka Connect** to continuously import & export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters.

* To let you implement mission – critical use cases, a Kafka cluster is highly scalable & fault – tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

> **Clients**: They allow you to write distributed applications & microservices that read, write & process streams of events in parallel, at scale, & in a fault – tolerant manner even in the case of network problems or machine failures.

* Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java & Scala including the higher – level Kafka Streams library, for Go, Python, C / C++, & many other programming languages as well as REST APIs.

**Use Cases**

**a) Messaging**

> Kafka works well as a replacement for a more traditional message broker.

> Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc.)

> In comparison to most messaging systems, Kafka has better throughput, built – in partitioning, replication, & fault – tolerance which makes it a good solution for large scale message processing applications.

> In our experience messaging uses are often comparatively low – throughput, but may require low end – to – end latency & often depend on the strong durability guarantees Kafka provides.

> In this domain Kafka is comparable to traditional messaging systems such as ActiveMQ or RabbitMQ

**b) Website Activity Tracking**

> The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real – time publish – subscribe feeds.

> This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real – time processing, real – time monitoring, & loading into Hadoop or offline data warehousing systems for offline processing & reporting.

> Activity tracking is often very high volume as many activity messages are generated for each user page view.

**c) Metrics**

> Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

**d) Log Aggregation**

> Many people use Kafka as a replacement for a log aggregation solution.

> Log aggregation typically collects physical log files off servers & puts them in a central place (a file server or HDFS perhaps) for processing.

> Kafka abstracts away the details of files & gives a cleaner abstraction of log or event data as a stream of messages. This allows for lower – latency processing & easier support for multiple data sources & distributed data consumption.

> In comparison to log – centric systems like Scribe or Flume, Kafka offers equally good performance, stronger durability guarantees due to replication & much lower end – to – end latency.

**e) Stream Processing**

> Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics & then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow – up processing.

e.g., A processing pipeline for recommending news articles might crawl article content from RSS feeds & publish it to an “article” topic;

Further processing might normalize or deduplicate this content & publish the cleansed article content to a new topic;

A final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real – time data flows based on the individual topics.

> Starting in 0.10.0.0, a light – weight but powerful stream processing library called **Kafka Streams** is available in Apache Kafka to perform such data processing.

> Apart from Kafka Streams, alternative open – source stream processing tools include **Apache Storm** & **Apache Samza**.

**f) Event Sourcing**

> **Event sourcing** is a style of application design where state changes are logged as a time – ordered sequence of records.

> Kafka’s support for very large stored log data makes it an excellent backend for an application built in this style.

**g) Commit Log**

> Kafka can serve as a kind of external commit – log for a distributed system.

> The log helps replicate data between nodes & acts as a re – syncing mechanism for failed nodes to restore their data.

> The log compaction feature in Kafka helps supports this usage.

**Main Concepts & Terminology**

**a)** An **event** records the fact that “something happened” in the world or in your business. It is also called **Record** or **Message**.

> When you read or write data to Kafka, you do this in the form of **events**. Conceptually, an event has a key, value, timestamp & optional metadata headers.

e.g.

* Event key: "Alice"
* Event value: "Made a payment of $200 to Bob"
* Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

**b)** **Producers** are those client applications that publish (write) events to Kafka, & **Consumers** are those that subscribe to (read & process) these events.

> In Kafka, producers & consumers are fully decoupled & agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for.

e.g., Producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly – once.

**c)** Events are organized & durably stored in **Topics**. Very simplified, a topic is similar to a folder in a filesystem, & the events are the files in that folder.

e.g., Topic name could be “payments”.

> **Topics** in Kafka are always multi – producer & multi – subscriber: a topic can have zero, one or many producers that write events to it, as well as zero, one or many consumers that subscribe to these events.

> Events in a topic can be read as often as needed – unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per – topic configuration setting, after which old events will be discarded.

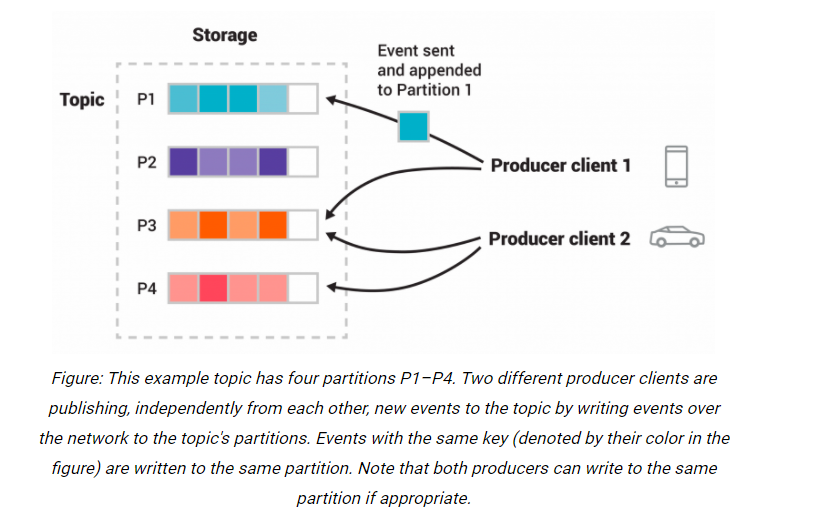
> Kafka’s performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

d) Topics are **partitioned**, meaning a topic is spread over a number of “**buckets**” located on different Kafka brokers.

> This distributed placement of your data is very important for scalability because it allows client applications to both read & write the data from/to many brokers at the same time.

> When a new event is published to a topic, it is actually appended to one of the topic’s partitions.

> Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, & Kafka guarantees that any consumer of a given topic – partition will always read that partition’s events in exactly the same order as they were written.



> To make your data fault – tolerant & highly – available, every topic can be **replicated**, even across geo – regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, & so on.

> A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic – partitions.

**Kafka APIs**

> In addition to command line tooling for management & administrations tasks, Kafka has 5 core APIs for Java & Scala:

1. The **Producer API** to publish (write) a stream of events to one or more Kafka topics.

> It allows applications to send streams of data to topics in the Kafka cluster.

1. The **Consumer API** to subscribe to (read) one or more topics & to process the stream of events produced to them.

> It allows applications to read streams of data from topics in the Kafka cluster.

1. The **Kafka Streams API** to implement stream processing applications & microservices. It provides higher – level functions to process event streams, including transformations, stateful operations like aggregations & joins, windowing, processing based on event – time, & more. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams to output streams.

> It allows transforming streams of data from input topics to output topics.

1. The **Kafka Connect API** to build & run reusable data import/export connectors that consume (read) or produce (write) streams of events from & to external systems & applications so they can integrate with Kafka.

e.g., A connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don’t need to implement your own connectors because the Kafka community already provides hundreds of ready – to – use connectors.

> It allows implementing connectors that continually pull from some source system or application into Kafka or push from Kafka into some sink system or application.

1. The **Admin API** allows managing & inspecting topics, brokers, & other Kafka objects.

**Configuration**

> Kafka uses key – value pairs in the property file format for configuration. These values can be supplied either from a file or programmatically.

**Design**

> Kafka is designed to be able to act as a unified platform for handling all the real – time data feeds a large company might have.

> Kafka would have to have high – throughput to support high volume event streams such as real – time log aggregation.

> Kafka would need to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems.

> Kafka also meant the system would have to handle low – latency delivery to handle more traditional messaging use – cases.

> Kafka supports partitioned, distributed, real – time processing of these feeds to create new, derived feeds.

> Kafka guarantees fault – tolerance in the presence of machine failures in cases where the stream is fed into other data systems for serving.