Università degli Studi di Messina

Data Science

Big Data Acquisition

-

Kafka

Prof. Daniele Ravi

dravi@unime.it

Main tools in Hadoop

Hadoop Distributed File System (HDFS)

Role:

- HDFS is the storage layer in Hadoop. It’s a distributed file system designed to store vast amounts of data across multiple nodes.

Connections:

- Many components in the Hadoop ecosystem rely on HDFS as a foundational layer to store both input and output data.

Hadoop NoSQL Database (HBase)

Role:

- HBase is a distributed, scalable NoSQL database built on top of HDFS.
- It’s designed for real-time read and write access to large datasets.

Connections:

- HBase stores data in HDFS and can interact with MapReduce for batch processing.

- Additionally, applications like Hive can query HBase tables to combine data stored in HBase with data stored in HDFS.

Yet Another Resource Negotiator (YARN)

Role:

- YARN is Hadoop’s resource management layer, allocating resources across the cluster and managing job scheduling.

Connections:

- YARN manages resources for applications like MapReduce, Spark, and others running on Hadoop.

- It ensures that jobs receive adequate resources to operate efficiently on distributed nodes.

MapReduce

Role:

- MapReduce is a programming model used for processing large datasets with a distributed algorithm on a cluster.

Connections:

- MapReduce jobs run on YARN-managed resources and typically read from and write to HDFS, making it one of the primary processing models in Hadoop.

Spark Streaming

Role:

- Spark Streaming is a component of Apache Spark that enables real-time data processing and analytics, leveraging **in-memory computing** for faster results.
- It can process data in both batch and streaming modes, giving users flexibility in how they analyze data.

Connections:

- Spark Streaming can run on YARN, access data from HDFS, HBase, or other sources, and process it in near real-time, complementing batch processing models like MapReduce.

Today: Data Ingestion

- **Components**: Apache Kafka, Apache Flume, etc.

- **Process**:
  - Data is ingested from various sources, such as logs, sensors, user interactions, and external databases.

  - Kafka acts as a **message broker** that collects and organizes real-time streams of data, while Flume is typically used for **collecting and aggregating** log data from different sources.

  - The ingested data can be sent directly to HDFS for batch processing or to other processing engines like Spark Streaming for real-time analysis.

Next module: Data Querying and Analysis

- **Components:** Hive, Pig.

- **Process**:
  - Users write queries in SQL-like language (HiveQL), and Hive takes care of optimizing and executing them. Hive translates SQL-like queries into MapReduce jobs that are executed on YARN.

  - Pig allows users to write scripts in Pig Latin, which are then converted into MapReduce jobs to process data in HDFS.

Putting It All Together

- **Data Ingest Systems** like Kafka and Flume bring in data.

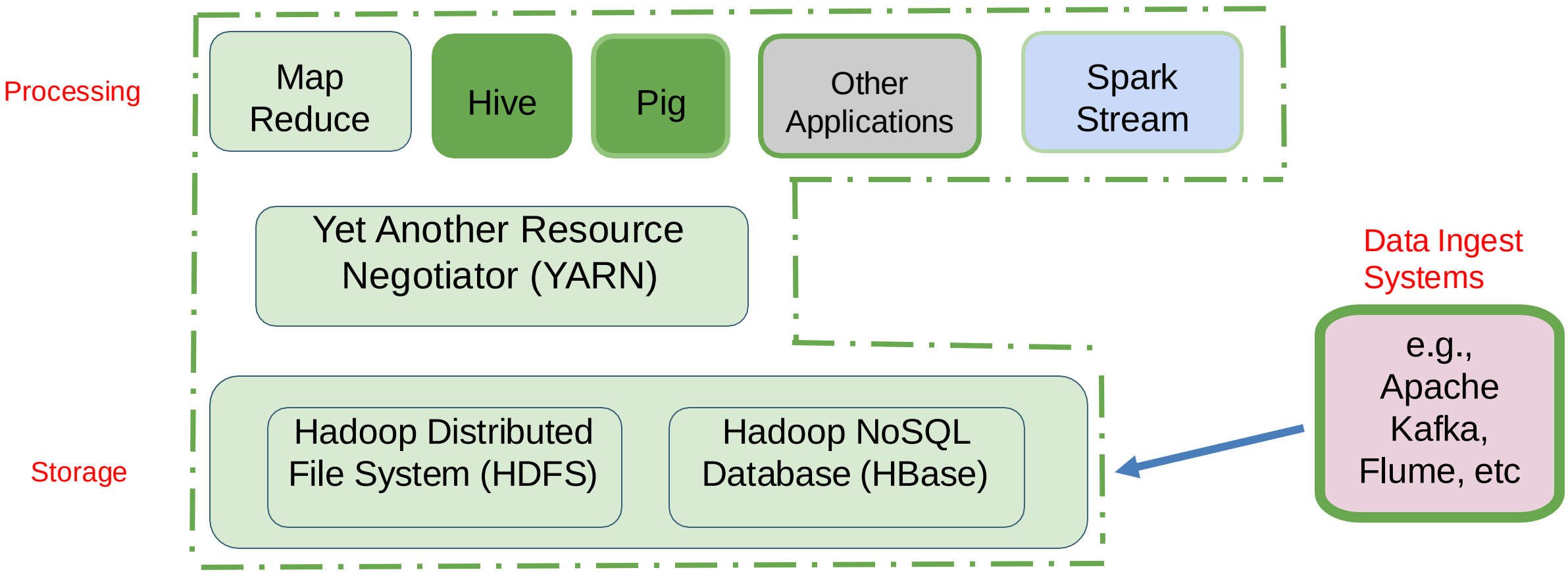
- **HDFS** stores the incoming data in a distributed, fault-tolerant way.
- **HBase** is used when real-time read/write access is needed on top of HDFS

- **YARN** manages resources for running applications.

- **MapReduce** or **Spark** (with Spark Streaming, for real-time) processes data, often in batch or real-time.

- **Hive** and **Pig** provide higher-level abstractions and make querying or processing large datasets easier.

Hadoop Ecosystem



Data Flow

- In Hadoop ecosystem system data moves from **ingestion** to **storage** and **processing**, enabling both batch and real-time analytics.

- This is a typical data flow:

1. **Ingestion**:
    1. Data from a web application is sent to Kafka.
    2. Flume collects server logs and streams them to HDFS.

2. **Storage**:
    1. Kafka writes messages to HDFS.
    2. The data is partitioned and replicated for fault tolerance.

3. **Processing**:
    - Hive query can be issued to analyze data in HDFS, which translates to a MapReduce job executed by YARN.
    - Spark Streaming can be used to processes real-time data from Kafka for immediate insights.
    - The results of the Hive query or Spark Streaming can be stored back in HDFS

Apache Hive and Pig

Apache Hive: SQL on MapReduce

Hive is an abstraction layer on top of Hadoop (MapReduce/Spark)

Use Cases:

- Data Preparation
- ETL Jobs (Data Warehousing)
- Data Mining

Apache Hive: SQL on MapReduce

Hive is an abstraction layer on top of Hadoop (MapReduce/Spark)

- ➢ Hive uses a SQL-like language called HiveQL
- ➢ Facilitates reading, writing, and managing large datasets residing in distributed storage using SQL-like queries
- ➢ Hive executes queries using MapReduce (*and also using Spark*)
  - HiveQL queries → Hive → MapReduce Jobs

Apache Hive

➢ Structure is applied to data at time of read → No need to worry about formatting the data at the time when it is stored in the Hadoop cluster

➢ Data can be read using any of a variety of formats:

- Unstructured flat files with comma or space-separated text
- Semi-structured JSON files (a web standard for event-oriented data such as news feeds, stock quotes, weather warnings, etc)
- Structured HBase tables

➢ Hive should be used for “data warehousing” tasks, not arbitrary transactions.

Apache Pig: Scripting on MapReduce

Pig is an abstraction layer on top of Hadoop (MapReduce/Spark)

- ➢ Use Cases:
  - Data Preparation
  - ETL Jobs (Data Warehousing)
  - Data Mining

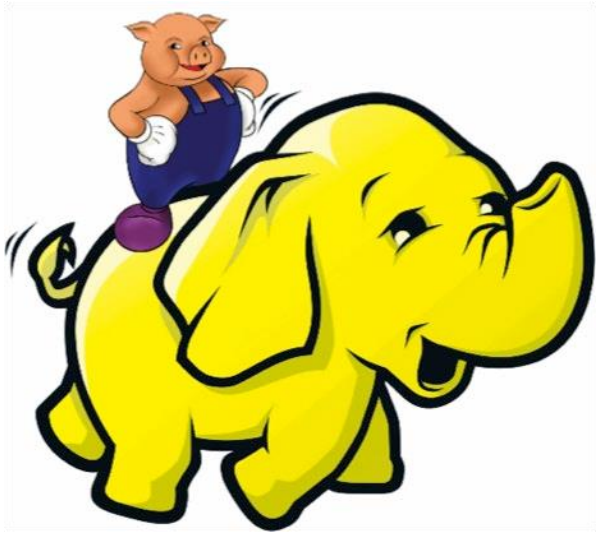
Apache Pig: Scripting on MapReduce

Pig is an abstraction layer on top of Hadoop (MapReduce/Spark)

- ➢ Code is written in Pig Latin “script” language (a data flow language)
- ➢ Facilitates reading, writing, and managing large datasets residing in distributed storage
- ➢ Pig executes queries using MapReduce (*and also using Spark*)
  - ○ Pig Latin scripts → Pig → MapReduce Jobs

Apache Hive & ApachePig

- Instead of writing Java code to implement MapReduce, one can opt between Pig Latin and Hive SQL to construct MapReduce programs
- Much fewer lines of code compared to MapReduce, which reduces the overall development and testing time



Apache Hive vs Apache Pig

- Declarative SQL-like language (HiveQL)
- Operates on the server side of any cluster
- Better for structured Data
- Easy to use, specifically for generating reports
- Data Warehousing tasks
- Facebook

- Procedural data flow language (Pig Latin)
- Runs on Client side of any cluster
- Best for semi structured data
- Better for creating data pipelines
  - allows developers to decide where to checkpoint data in the pipeline
- Incremental changes to large data sets and also better for streaming
- Yahoo

Apache Hive vs ApachePig: example

Job: Retrieve data from sources (specifically "users" and "clicks"), join and filter this data, then join it with a third source ("geoinfo"), aggregate the resulting data, and store it in a table named `ValuableClicksPerDMA`.

```
insert into ValuableClicksPerDMA
select dma, count(*)
from geoinfo join (
select name, ipaddr
from users join clicks on
(users.name = clicks.user)
where value > 0;
) using ipaddr
group by dma;
```

```
Users    = load 'users' as (name, age, ipaddr);
Clicks   = load 'clicks' as (user, url, value);
ValuableClicks   = filter Clicks by value > 0;
UserClicks = join Users by name, ValuableClicks by
user;
Geoinfo  = load 'geoinfo' as (ipaddr, dma);
UserGeo  = join UserClicks by ipaddr, Geoinfo by
ipaddr;
ByDMA    = group UserGeo by dma;
ValuableClicksPerDMA   = foreach ByDMA generate group,
COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```

Comment: “Client side”??

When we say “runs on client side” we don’t mean “runs on the iPhone”. Here the client is any application using Hadoop.

So the “client side” is just “inside the code that consumes the Pig output”

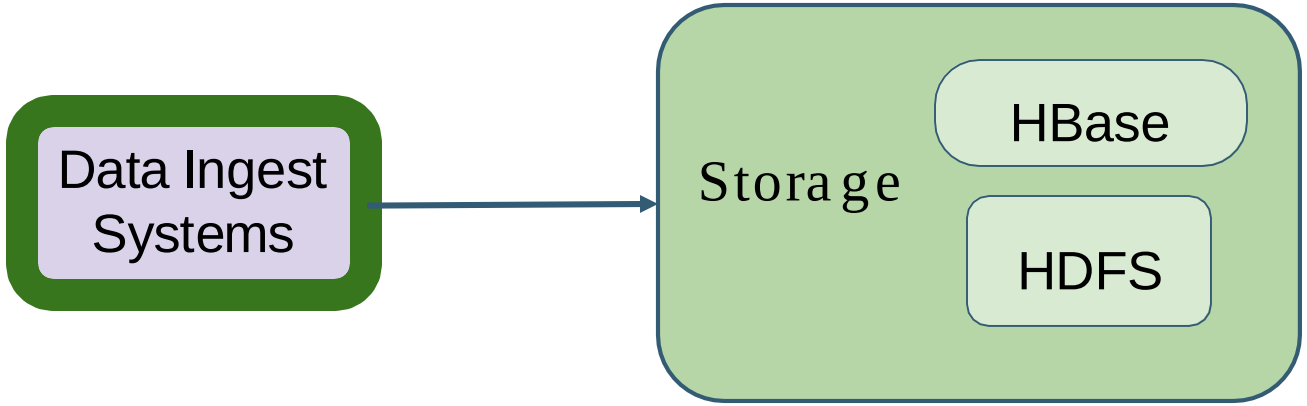
In contrast, the “server side” lives “inside the Hive/HDFS layer”

Apache Kafka

Data Ingestion Systems/Tools (1)

Hadoop typically ingests data from many sources and in many formats:

- Traditional data management systems, e.g. databases
- Logs and other machine generated data (event data)
- e.g., Apache Sqoop, Apache Fume, **Apache Kafka** (focus of this class)



Data Ingestion Systems/Tools (2)

- Apache Sqoop
  - High speed import to HDFS from Relational Database (and vice versa)
  - **Supports many database systems,**
    e.g. Mongo, MySQL, Teradata, Oracle

- Apache Flume
  - Distributed service for ingesting streaming data
  - Ideally suited for event data from multiple systems, for example, log files

Cooperation Among Instances in a Pool

- The Apache ecosystem is pretty elaborate.
- It has many “tools”, and several are implemented as separate μ-services (microservices).
- The μ-services run in pools: we configure the cloud to automatically add instances if the load rises, reduce if it drops

Models for cooperation

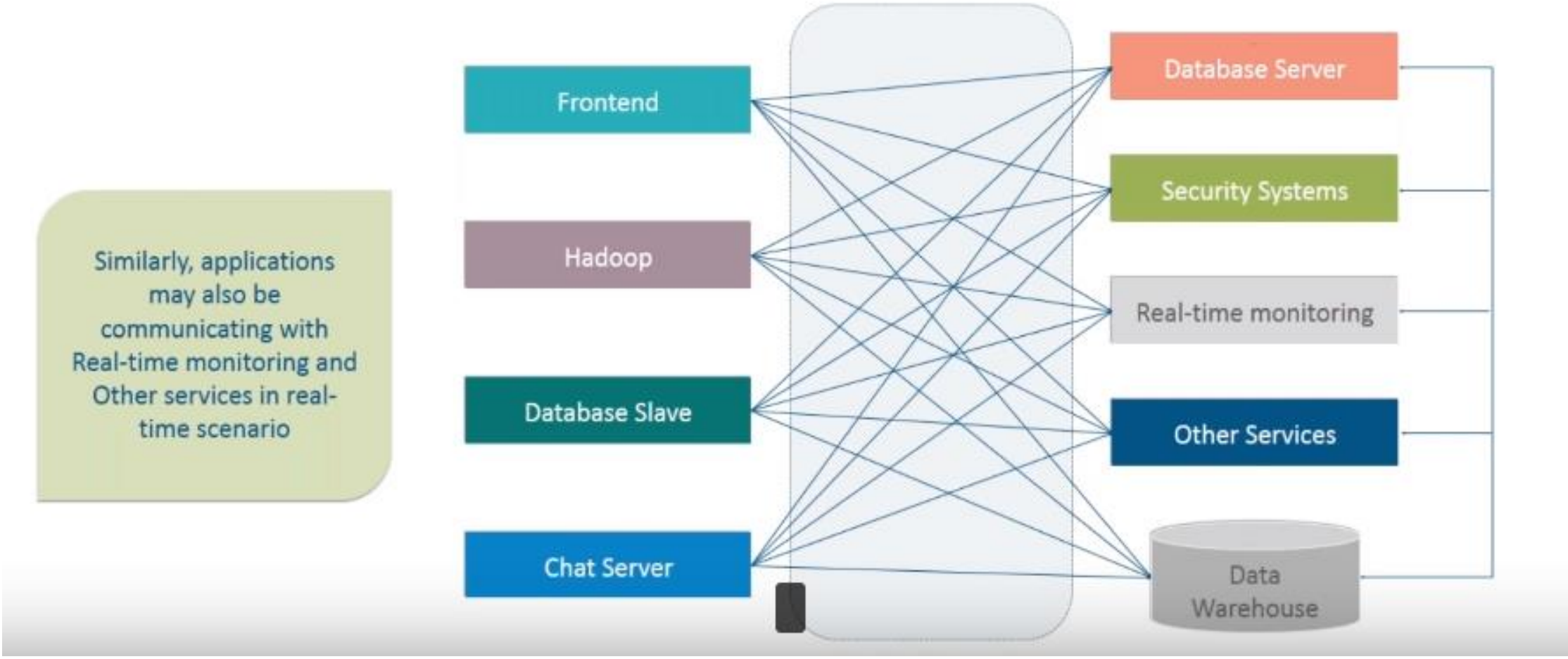
- When instances run in a pool, they may need to coordinate with each other to avoid working in isolation and instead contribute to a collaborative process.

- In such cases, **groups can be explicitly defined**, with each worker or process aware of all other members in the group.

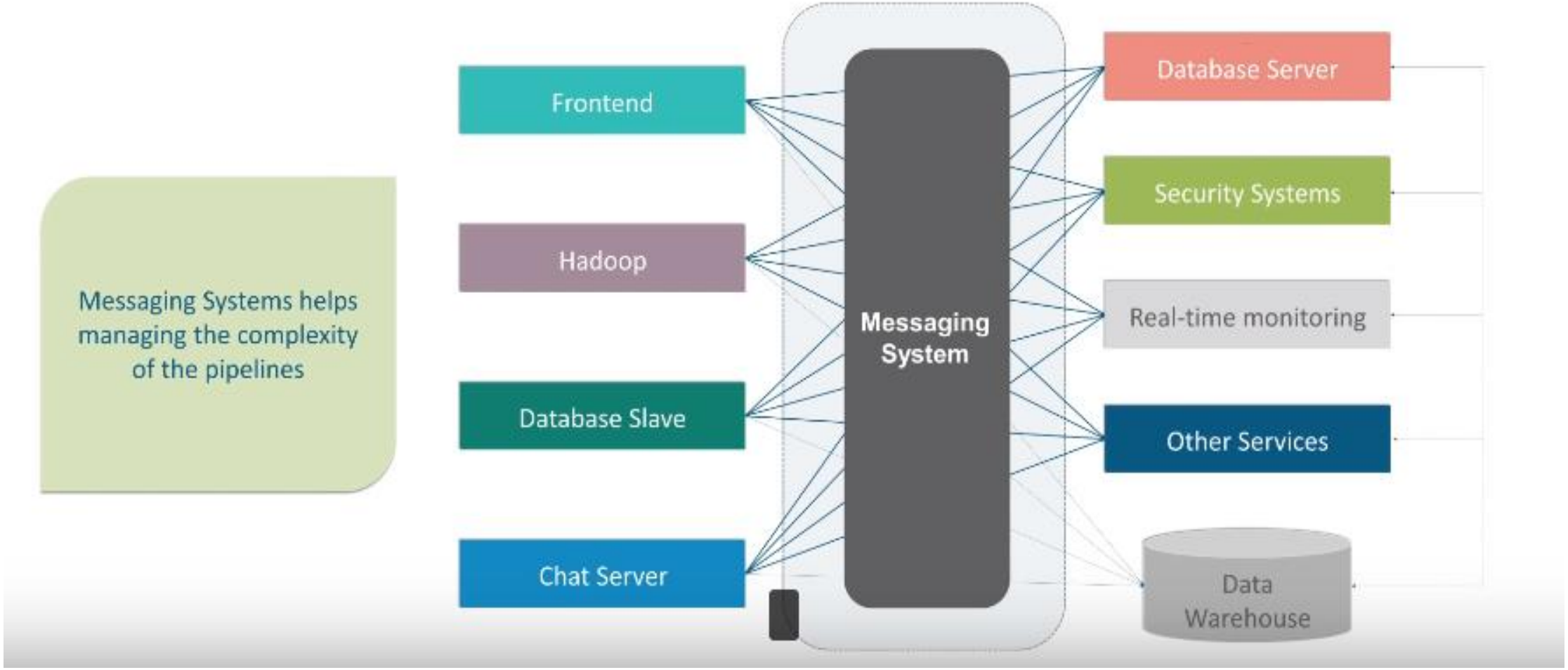
- This setup allows for structured and predictable cooperation based on a shared view of the task list and group membership

- However, this approach can be overly complex for certain use cases.
- A simpler option is loose coordination, where workers independently pick tasks from a shared list, complete them, and then report completion, without needing to track each other directly.

Cooperation in a Complex Data Pipeline



Solution to the Complex Data Pipeline



Messaging System

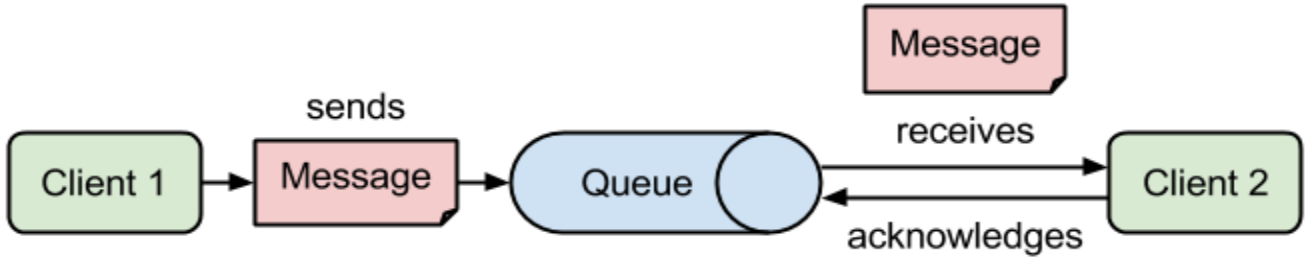
There are two types of Messaging System:

1. **Point to Point System**
2. **Publish-Subscribe System**

1. Point to Point System

Messages are persisted in a queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue.

The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time.
The following diagram depicts the structure.



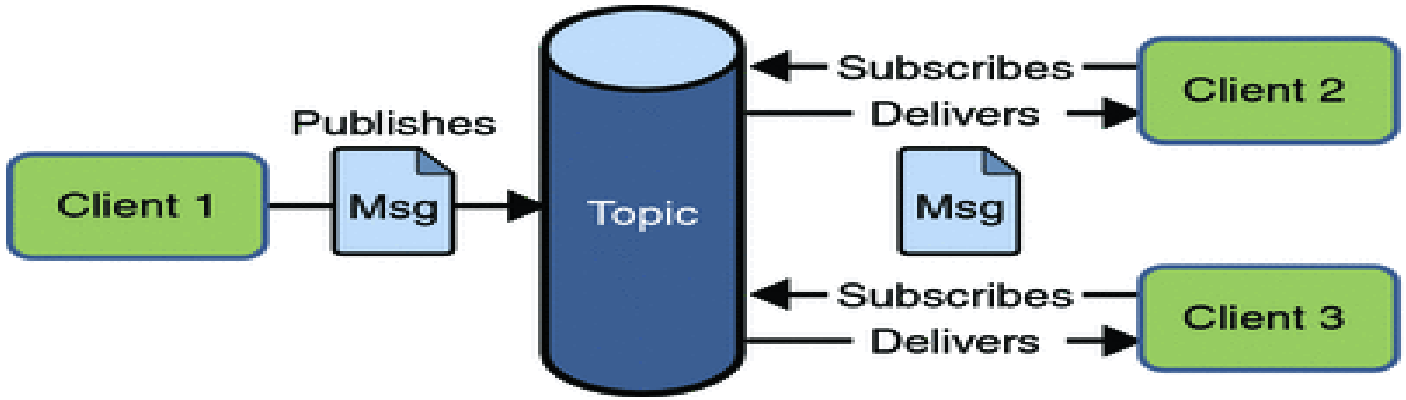
Messaging System

2. Publish-Subscribe System

Messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic.

In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers.

A real-life example is a TV system, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



How Publish-Subscribe Works

In a Publish-Subscribe system, instances (or services) communicate by "publishing" and "subscribing" to messages on specific **topics** or **channels**.

Here’s how it generally works:

- **Publishers** send messages to a topic without needing to know who (or how many) will receive them.
- **Subscribers** express interest in specific topics and receive any messages that are published to those topics.

- This approach decouples the sender and receiver, allowing for scalable and flexible communication among services.

How Pub-Sub Facilitates Cooperation in Pools

With the Pub-Sub model, instances in a pool can cooperate and stay in sync without directly communicating with each other.

Event Broadcasting:

- When a significant event occurs (like a data update or state change), an instance can publish a message on a topic.

- Other instances subscribed to that topic can immediately receive the update and react to it.

- For example, if one instance updates a user profile, other services listening to the "user-profile-updates" topic can respond to the change.

How Pub-Sub Facilitates Cooperation in Pools

Task Distribution:

- By using Pub-Sub, a pool of instances can balance workloads without a central coordinator.

- For example, a load balancer can publish tasks to a Kafka topic, and instances can consume tasks from that topic based on their capacity.

- This model allows new instances to automatically join the pool, start subscribing to the topic, and contribute to the workload as demand rises.

How Pub-Sub Facilitates Cooperation in Pools

Resilience and Fault Tolerance:

- Pub-Sub tools can store published messages so that even if a subscribing instance is temporarily unavailable, it can retrieve and process messages when it comes back online.
- This ensures that messages aren't lost if an instance fails or is temporarily down.

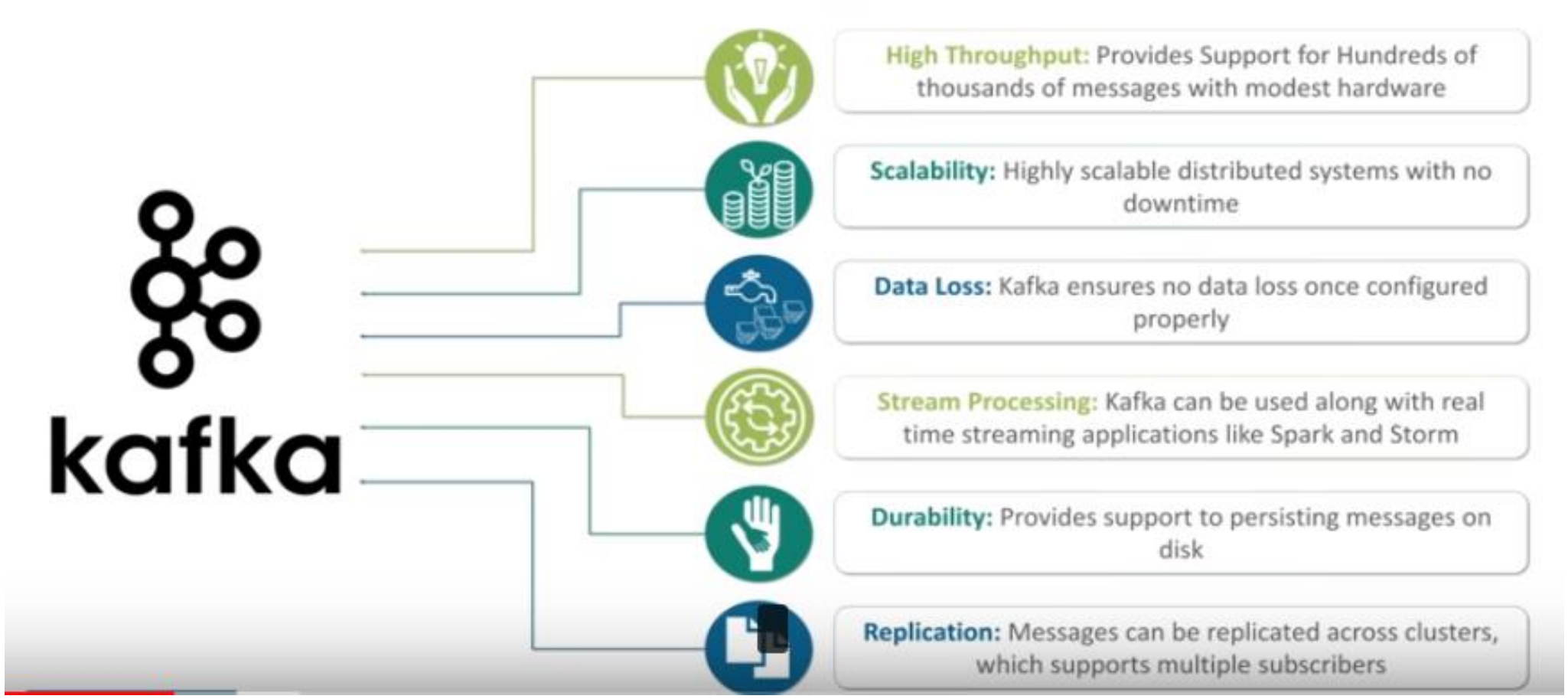
Summary

- The Pub-Sub model, enables instances in a pool to **cooperate by publishing and subscribing to events** rather than making direct requests to each other.
- This makes the system more modular, scalable, and resilient.
- Instances in a pool can dynamically react to workload changes, share updates in real-time, and coordinate actions through event streams without direct coupling.

Apache Kafka

- Functions like a distributed publish-subscribe messaging system
  - Distributed, reliable publish-subscribe system
- Originally developed by LinkedIn, now widely popular
- Features: Durability, Scalability, High Availability, High Throughput
- **One of the most widely used Pub-Sub tools**

Apache Kafka features



Apache Kafka features

Kafka is a Distributed, Replicated Commit Log system

- **Distributed**: Kafka runs on a cluster of nodes for **fault tolerance** and **scalability**.
- **Replicated**: Messages are **replicated across multiple nodes** to prevent data loss.
- **Commit Log**: Kafka stores messages in **partitioned, append-only logs** called topics, keeping a time-ordered record of events.
- **Kafka's Advantage**: This log-based design enables high throughput and scalability, making Kafka ideal for handling continuous streams of data in real-time.

Why is Kafka so fast?

- Fast **writes**:
  - While Kafka persists all data to disk, all writes go to the **page cache** or RAM.

- Fast **reads**:
  - Very efficient to transfer data from page cache to a network **socket**

- On a Kafka cluster you will see no read activity **on the disks as** they will be serving data entirely from cache.

What is Apache Kafka used for? (1)

- The original use case (@LinkedIn):
  - To track user behavior on websites.
  - Site activity (page views, searches, or other actions users might take) is published to central topics, with one topic per activity type.
- Effective for two broad classes of applications:
  - Building **real-time streaming data pipelines** that reliably get data between systems or applications
  - Building **real-time streaming applications that transform** or react to the streams of data

What is Apache Kafka used for? (2)

- **Messaging**
- Kafka has a hight throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large-scale message processing applications.

- **Metrics**
- For operational monitoring data, Kafka includes aggregating statistics from distributed applications to produce centralized feeds of operational data.

- **Event Sourcing**
- Since it supports very large stored log data, Kafka is an excellent backend for applications of event sourcing.

Apache Kafka: Fundamentals

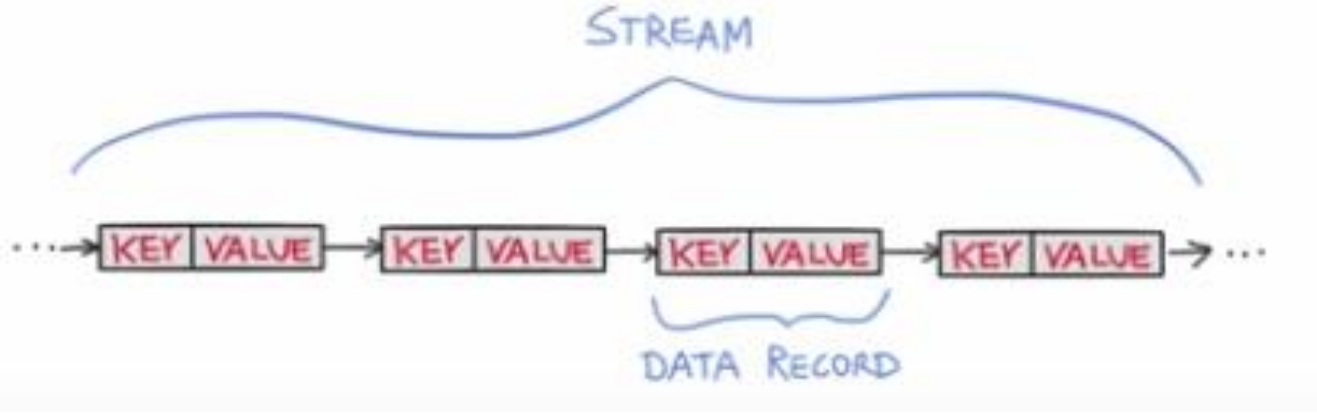
- Kafka is run as a cluster on one or more servers
- The Kafka cluster stores streams of *records* in categories called *topics*
- Each record (or message) consists of a key, a value, and a timestamp

- Publish-Subscribe: Messages are persisted in a topic, consumers can subscribe to one or more topics and consume all the messages in that topic

Apache Kafka: Components

Logical Components:

- Topic: The named destination of partition
- Partition: One Topic can have multiple partitions and it is an unit of parallelism
- Record or Message: Key/Value pair (+ Timestamp)



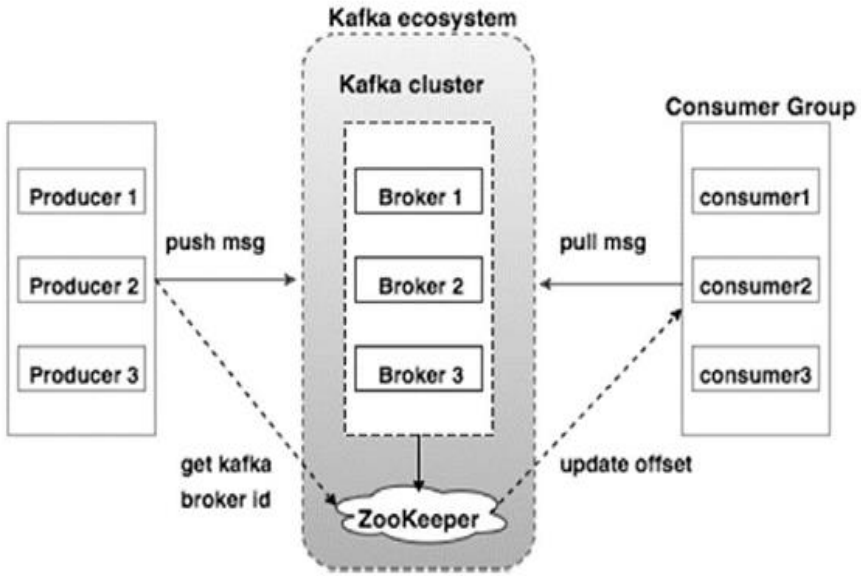
Apache Kafka: Components

Physical Components:

- **Producer**: Send message to broker
  - It publishes messages to a Kafka topic. The producer is responsible for choosing which record to assign to which partition within the topic.
- **Consumer**: Receive message from broker
  - This component subscribes to a topic(s), reads and processes messages from the topic(s).
- **Broker**: One node of Kafka cluster
  - Kafka Broker manages the storage of messages in the topic(s). If Kafka has more than one broker, that is what we call a Kafka cluster.
- **ZooKeeper**: Coordinator of Kafka cluster and consumer groups
  - To offer the brokers with metadata about the processes running in the system and to facilitate health checking and managing and coordinating, Kafka uses Kafka zookeeper.

Apache Kafka as a Messaging System

- **Producers** write data to **brokers**.
- **Consumers** read data from **brokers**.
- All this is distributed.
- Data is stored in **topics**.
- **Topics** are split into **partitions**, which are **replicated**.



Apache Kafka Work-Flow

1. Producers send message to a topic at regular intervals.

2. Kafka broker stores all messages in the partitions configured for that particular topic. **It ensures the messages are equally shared between partitions.**
   If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second.

3. Consumer subscribes to a specific topic.

4. Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper.

5. Consumer will request new messages in a regular interval (like 100 Ms)

Apache Kafka Work-Flow

6. Once Kafka receives the messages from producers, it forwards them to the consumers

7. Consumer will receive the message and process it.

8. Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.

9. Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper.

10. This above flow will repeat until the consumer stops the request.

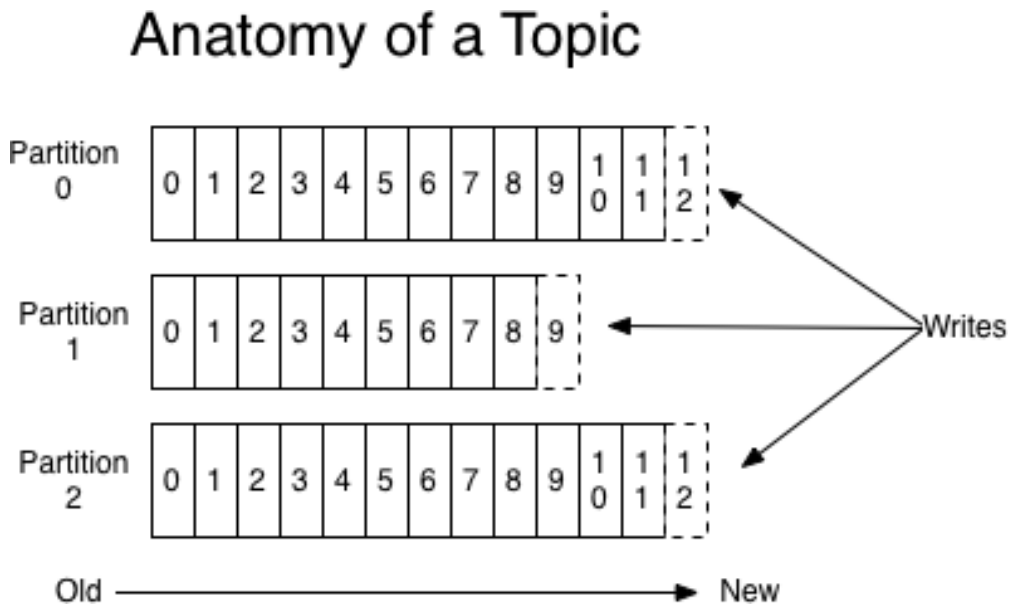
11. Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages

Apache Kafka: Topics & Partitions (1)

- A stream of messages belonging to a particular category is called a topic
- Data are stored in topics.
- Topics in Kafka are always multi-subscriber -- a topic can have zero, one, or many consumers that subscribe to the data written to it
- Topics are split into partitions. Topics may have many partitions, so it can handle an arbitrary amount of data

Apache Kafka: Topics & Partitions (2)

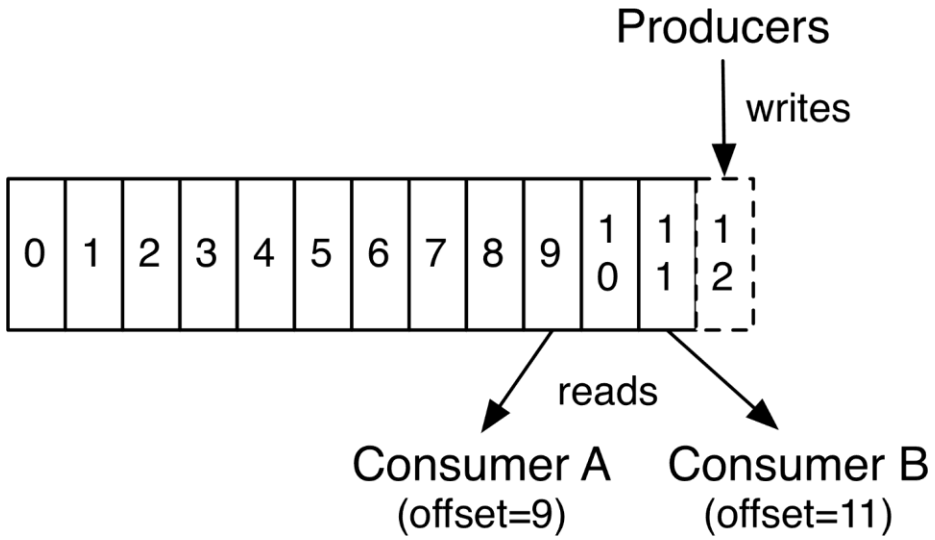
- For each topic, the Kafka cluster maintains a partitioned log that looks like this:



- Each partition is an ordered, immutable sequence of records that is continually appended to -- a structured commit log.
- *Partition offset*: The records in the partitions are each assigned a sequential id number called the ***offset*** that uniquely identifies each record within the partition.

Apache Kafka: Topics & Partitions (3)

- The only metadata retained on a per-consumer basis is the *offset* or position of that consumer in the log.
- This offset is controlled by the consumer -- normally a consumer will advance its offset linearly as it reads records (but it can also consume records in any order it likes)



Apache Kafka: Topics & Partitions (4)

The partitions in the log serve several purposes:

- Allow the log to scale beyond a size that will fit on a single server.
- Handles an arbitrary amount of data -- a topic may have many partitions
- Acts as the unit of parallelism

Apache Kafka: Topics & Partitions (4)

- The Kafka cluster durably persists all published records—whether or not they have been consumed—using a configurable retention period.

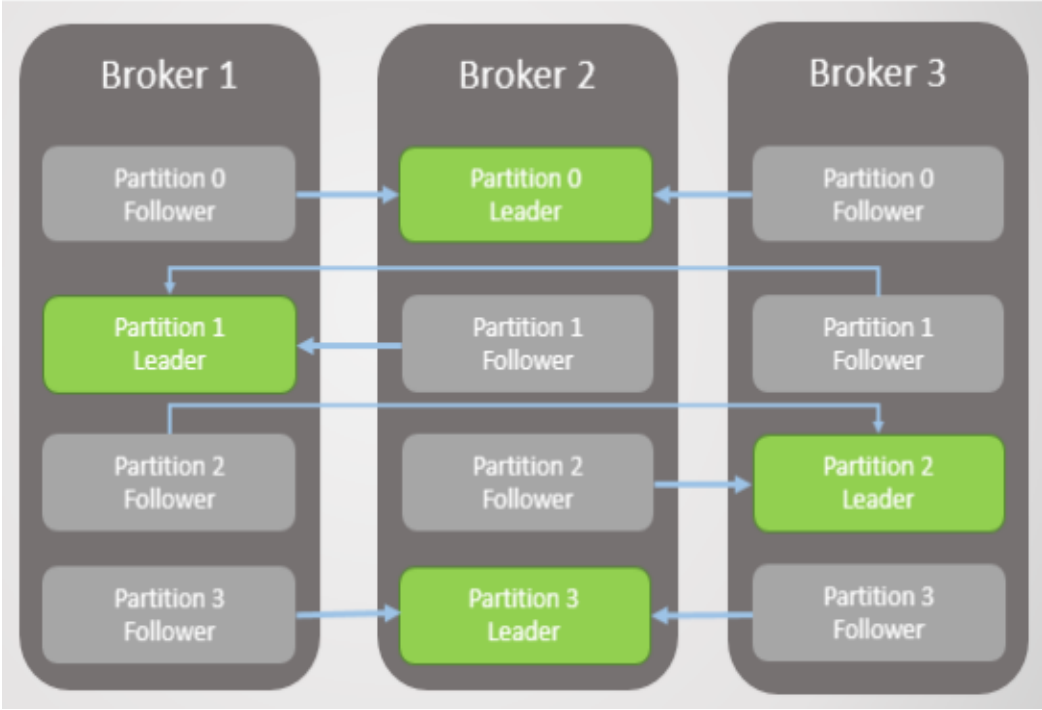
- For example, if the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space.

- **Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem**.

- This is one of the biggest difference between Kafka and other similar system (RabbitMQ/ActiveMQ)

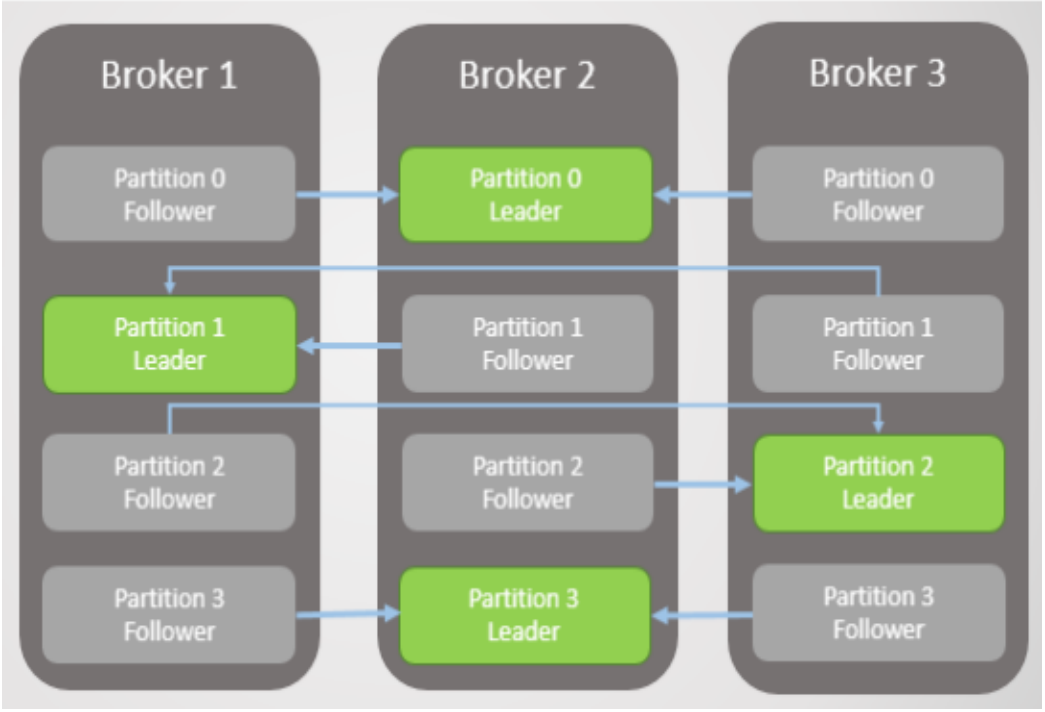
Apache Kafka: Distribution of Partitions(1)

- The partitions are distributed across servers in the Kafka cluster, with each partition replicated for fault tolerance.
- Each partition has one server that acts as the “leader” and zero or more servers that act as “followers.”
- The leader handles all read and write requests for the partition, while followers passively replicate the leader.
- If the leader fails, one of the followers automatically becomes the new leader.



Apache Kafka: Distribution of Partitions(2)

- Partitions for the same topic are spread across multiple brokers in the cluster, and the number of replicas is a configurable parameter.
- Each server acts as a leader for some partitions and as a follower for others, providing load balancing within the cluster.
- Producers decide which partition to assign messages to within a topic based on a key assigned to each message.



Apache Kafka: Distribution of Partitions(3)

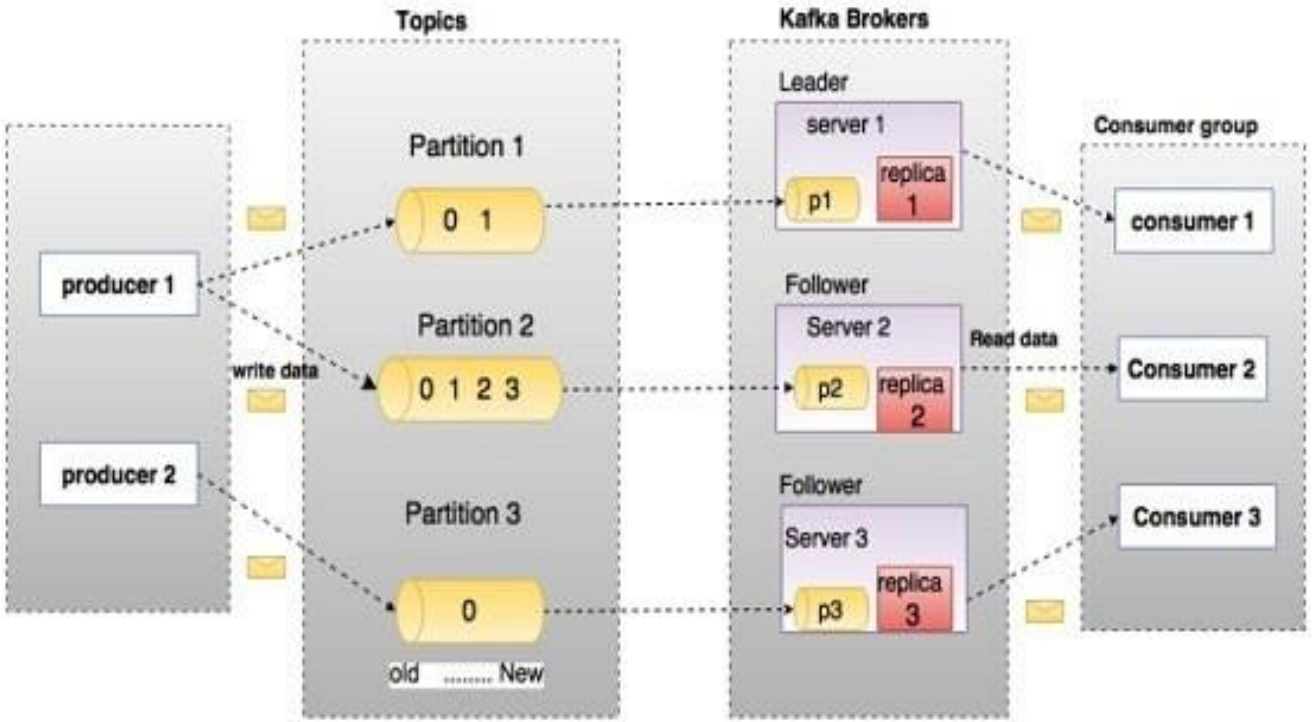
Here, a topic is configured into three partitions.

Partition 1 has two offset factors 0 and 1.

Partition 2 has four offset factors 0, 1, 2, and 3.

Partition 3 has one offset factor 0.

The id of the replica is same as the id of the server that hosts it.



Apache Kafka: Components

Logical Components:

- Topic: The named destination of partition
- Partition: One Topic can have multiple partitions and it is an unit of parallelism
- Record or Message: Key/Value pair (+ Timestamp)

Physical Components:

- Producer: The role to send message to broker
- Consumer: The role to receive message from broker
- Broker: One node of Kafka cluster
- ZooKeeper: Coordinator of Kafka cluster and consumer groups

Apache Kafka: Producers

- **Producers** are applications or processes that send (or "publish") data to topics.
- **Choosing Partitions**:
  - The producer must decide which **partition** within the topic each data record (message) should go to.
- **Partition Assignment Methods**:
  - **Round-Robin**: The producer assigns records to partitions in a **round-robin** manner, distributing messages evenly across all partitions to balance the load. This method doesn't consider the content of each record; it simply aims for an even distribution.
  - **Semantic Partitioning**: Alternatively, the producer can use a **partition function** based on specific characteristics (or "semantics") of each record.
  - For example, records with the same key (like a user ID) might go to the same partition to ensure that all messages related to that key are processed in order.
  - This approach allows logical grouping of related records.

Apache Kafka: Consumers

- A **consumer group** is a collection of consumers working together to consume data from a topic.
- Each consumer instance in a group is assigned the same **consumer group name**

- **Benefits of Consumer Groups:**
  - Allows multiple consumers to process data in parallel.
  - Supports scalability by dividing workload among consumers.

- Records published to a topic are **delivered to only one consumer instance within each subscribing group**.

- Ensures records are distributed across consumers within the group without duplication.

Apache Kafka: Consumers

- **Single Group, Multiple Consumers**:
  - If all consumers subscribe with the **same consumer group name**, Kafka **distributes records** across these instances.

- **Partition Assignment**:
  - Each consumer instance receives records from **different partitions**, balancing the workload.

- **Parallel Processing**:
  - This configuration allows multiple consumer instances to process data in parallel for faster throughput.

Apache Kafka: Consumers

- **Broadcast Setup for Data Access - Independent Grouping**:
  - If each consumer has a **unique group name**, each group receives **all records** from the topic.

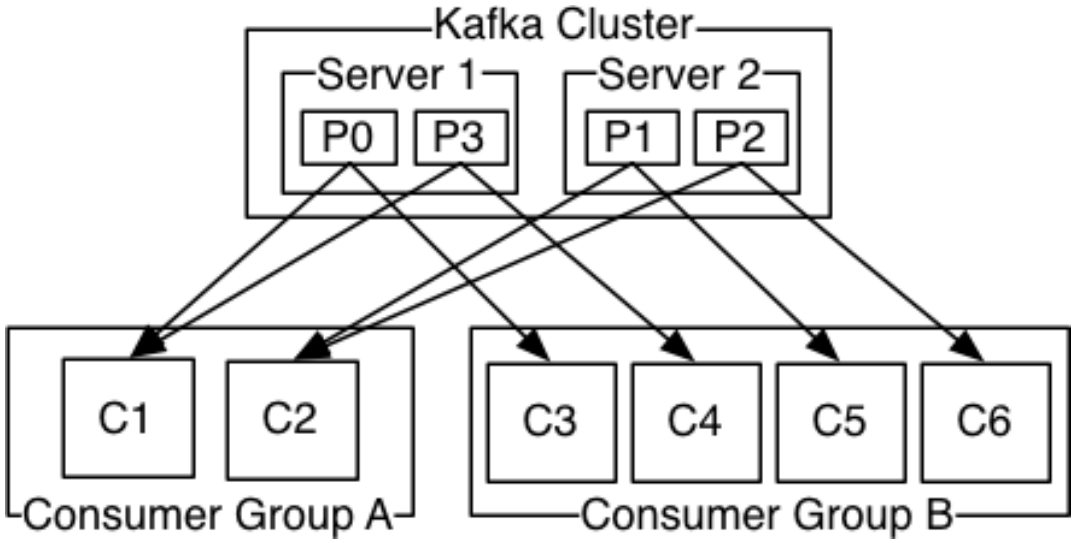
- **Benefits of Broadcast**:
  - Useful when **multiple teams or applications** require access to the same data.

  - **Redundant Data Delivery**:
    - Each group receives a full copy of the data, supporting independent processing needs.

Apache Kafka: Producers & Consumers

Example:

A two server Kafka cluster hosting four partitions (P0 to P3) with two consumer groups (A & B).



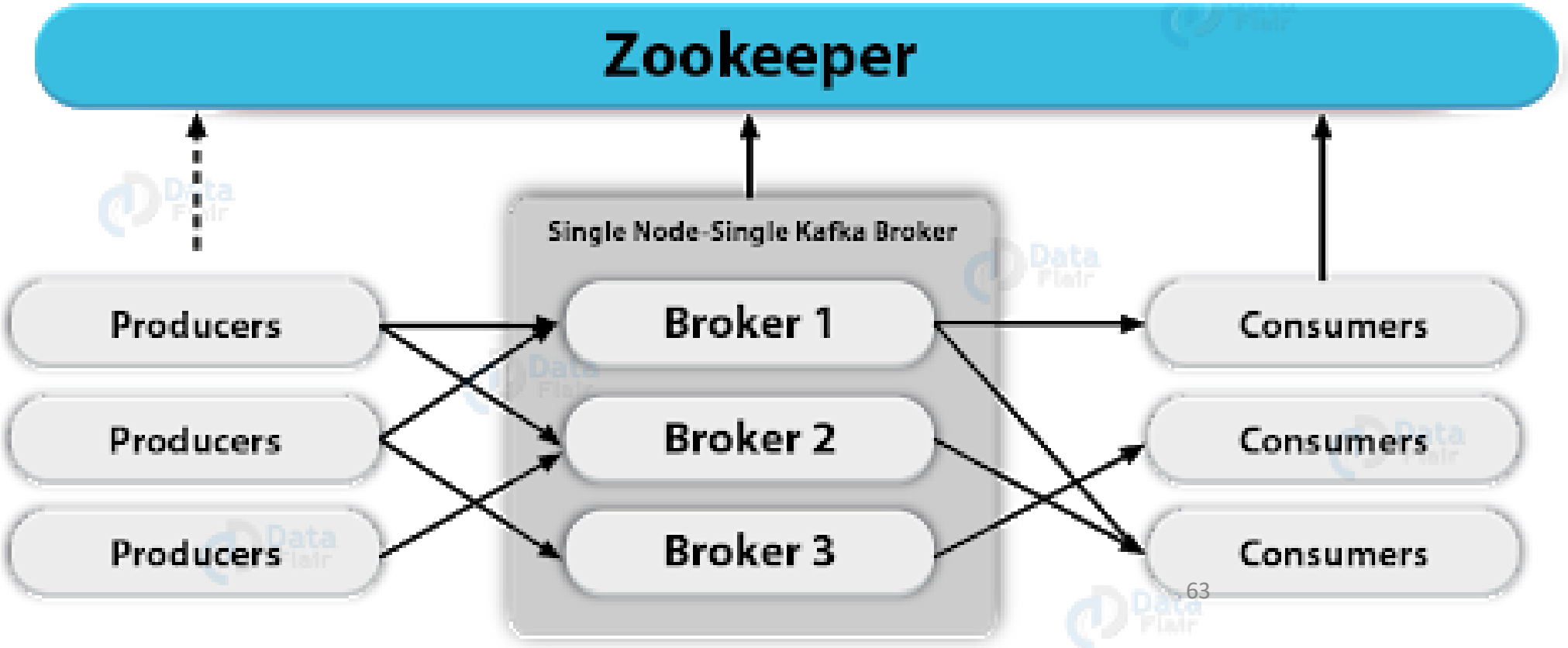
Consumer group A has two consumer instances (C1 & C2) and group B has four (C3 to C6).

Zookeeper

- It is a service for providing configuration and synchronization for distributed systems

- Very stable service, used in many distributed frameworks
- Zookeeper is used in Kafka to:
  - Elect a broker as a leader for a partition after a previous leader fails
  - Manage the topology of the cluster and send information about changes in the topology. For example, it informs all nodes when a new broker joins the cluster, when a broker dies or leaves the cluster, when a new topic is added or removed etc.
  - Provide synchronization and an overview of the current state of the cluster and its configuration

Zookeeper


Zookeeper
Single Node-Single Kafka Broker
Producers
Producers
Producers
Broker 1
Broker 2
Broker 3
Consumers
Consumers
Consumers

Roles of Zookeeper

In Kafka Brokers:

- State: Zookeeper checks the status of each broker by sending messages to check if the broker is alive
- Replicas: It keeps information about the location of each partition replica for a topic and has the responsibility of choosing a new leader in case of a failure of a previous leader
- Nodes and Topics: Zookeeper has information about which Kafka broker holds which topic

In Kafka Consumers:

- Offsets: Zookeeper holds information the offset of each consumer

Apache Kafka: Design Guarantees (1)

- Records (or Messages) sent by a producer to a particular topic partition will be appended in the order they are sent.

- A consumer instance sees records in the order they are stored in the log.

- For a topic with replication factor N, kafka will tolerate up to N-1 server failures without losing any records committed to the log.

Apache Kafka: Design Guarantees (2)

Message Delivery Semantics:

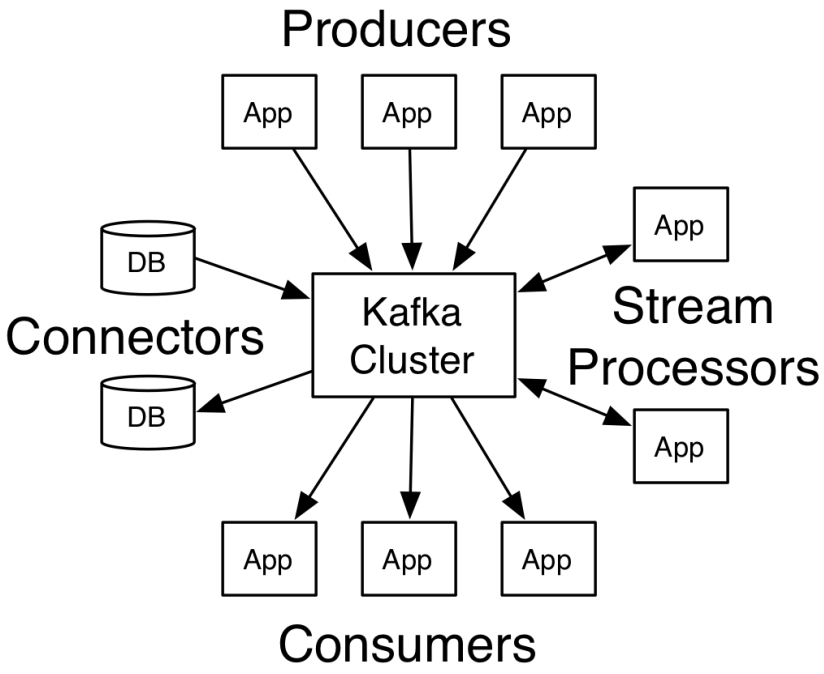
- ➢ At most once: Messages may be lost but are never redelivered.
- ➢ At least once: Messages are never lost but may be redelivered.
- ➢ Exactly once: Each message is delivered once and only once

Apache Kafka: Four Core APIs (1)

Producer API: Allows an application to publish a stream of records to one or more Kafka topics

Consumer API: Allows an application to subscribe to one or more topics and process the stream of records produced to them

Streams API: Allows an application to act as a *stream processor* -- consuming an input stream from one or more topics and producing an output stream to one or more output topics

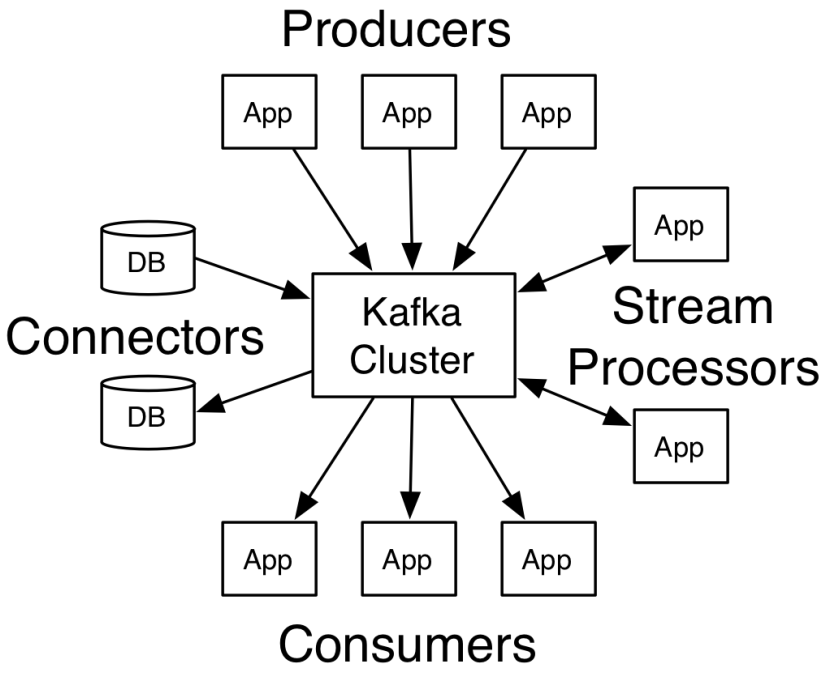


Apache Kafka: Four Core APIs (2)

Connector API:

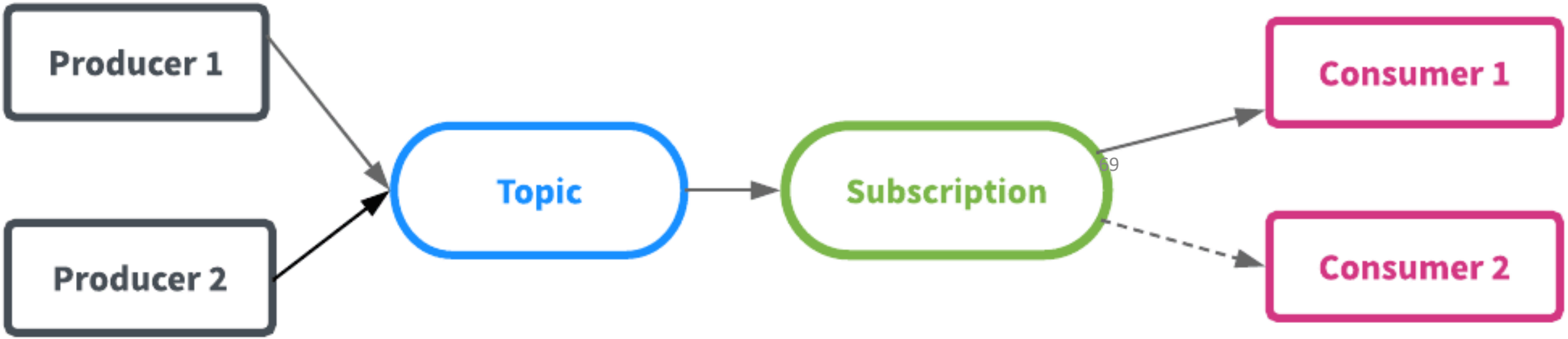
Allows building and running producers or consumers that connect Kafka topics to existing applications or data systems.

For example, a connector to a relational database might capture every change to a table.



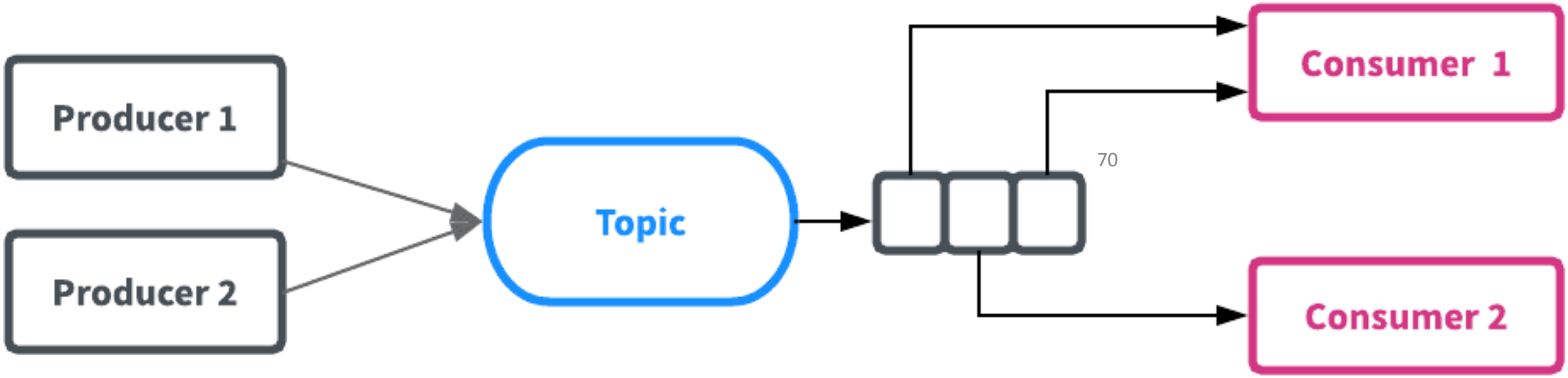
Kafka messaging models

- Queueing (messaging) : producers write messages to a topic, each message is consumed by only one consumer
- Publish - subscribe: producers write messages to a topic, all subscribed consumers receive all the messages



Kafka as a Queueing System

- Why use Kafka and not another messaging system?
  - Kafka uses more reliable clustering and data replication mechanisms than most conventional messaging systems



Kafka as a Storage system

- Messages are stored in the brokers until they are acknowledged
- Due to the replication that Kafka provides, it is ensured that a message will not get deleted even if a server fails
- Parallelization techniques of Kafka allow for good scaling. Good performance is guaranteed even for large volumes of data
- For these reasons, one could say that Kafka is a special purpose distributed filesystem

Kafka for Stream Processing

- Simple stream processing can be done using the producer and consumer APIs.
- If we want to do more complex processing, we can use the Streams API to give the data to a specialized stream processing application
- It allows to build programs that do computations using streams and do more complex operations such as joining streams together
- Example: an application which sells clothes could take as input streams containing information about sales and output a stream containing information about changes in prices, computed off of the input

Tricky aspects?

- Using the publish-subscribe model for fault-tolerant request-reply interactions is actually not so simple.

- Someone posts a request (easy), but now a random member of the worker pool wants to grab the request: a race condition.

Tricky aspects? (cont)

- **Race Condition**: When a request is posted, multiple workers in the pool may attempt to claim and process it, leading to a race condition. To resolve this, the server publishes an announcement declaring which worker has taken on the task.

- **Trust in Announcements**: The system relies on all participants trusting the first published announcement about a worker claiming a task. Subsequent announcements about the same task are ignored.

Tricky aspects? (cont)

- **Failure Detection**: To handle worker failures, a failure detection service (like Apache Zookeeper) is integrated. This service publishes announcements about workers joining or failing, allowing the system to detect if a worker that claimed a task has crashed.

- **Task Reactivation/Reissue**: If a worker crashes after claiming a task, the system can reactivate or reissue that task to ensure it gets completed.