POLITECNICO
MILANO 1863

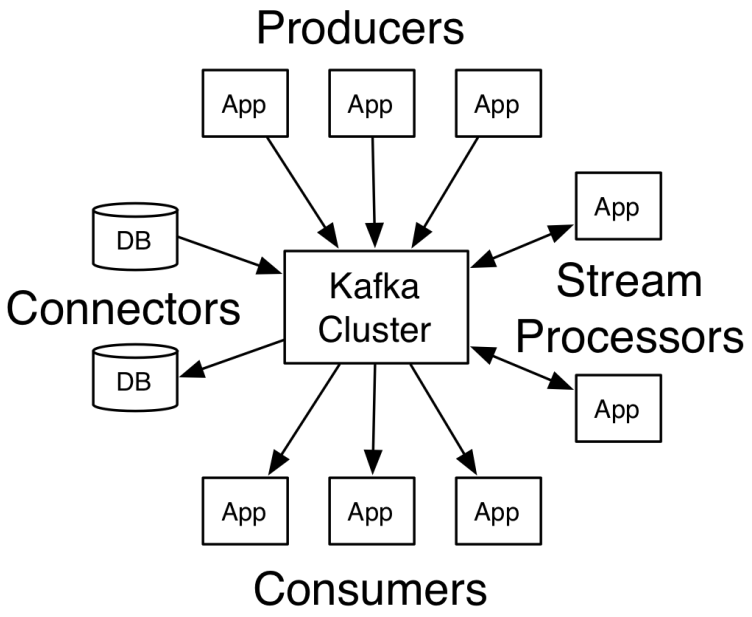
Apache Kafka

Alessandro Margara
alessandro.margara@polimi.it
https://margara.faculty.polimi.it

Apache Kafka: overview

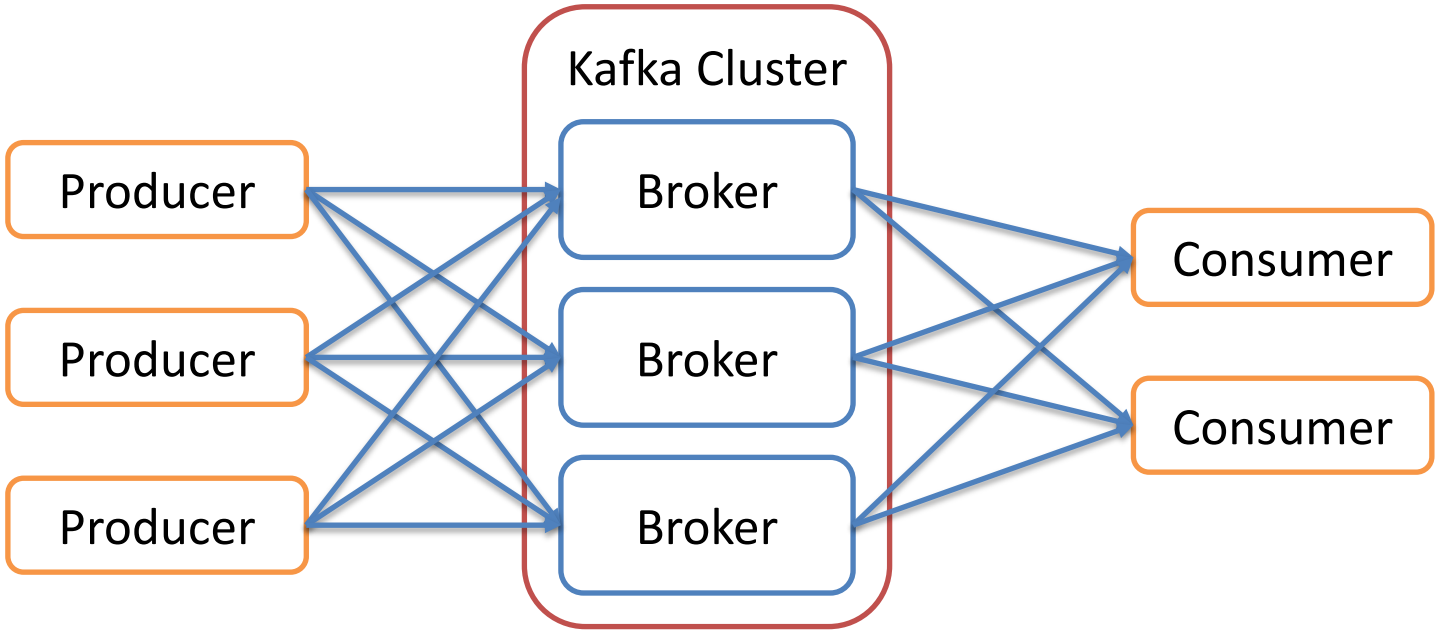
Apache Kafka terminology

- Producer API enables components to write messages to Kafka topics
- Consumer API enables components to read messages from Kafka topics
- Connectors are pre-baked adapters to 3rd party systems
  - Databases that output changes (write log)
  - Sources and sinks of data analytics tools
- More recently, Kafka Streams / KTable API introduced high-level abstractions to develop stream processing applications
  - Consume from one or more input topics
  - Join, aggregate, transform events
  - Write results to one or more output topics



High-level view

- A Kafka cluster consists of multiple servers/brokers
  - Storage and messaging components



Messages

- The basic unit of data in Kafka is a message (or record, or event)
  - Producers write messages to brokers
  - Consumers read messages from brokers

- A message is a key-value pair
  - Keys and values can be any data type
  - All data is stored in Kafka as byte arrays
  - Producers provide serializers to convert the key and value to byte arrays

Communication

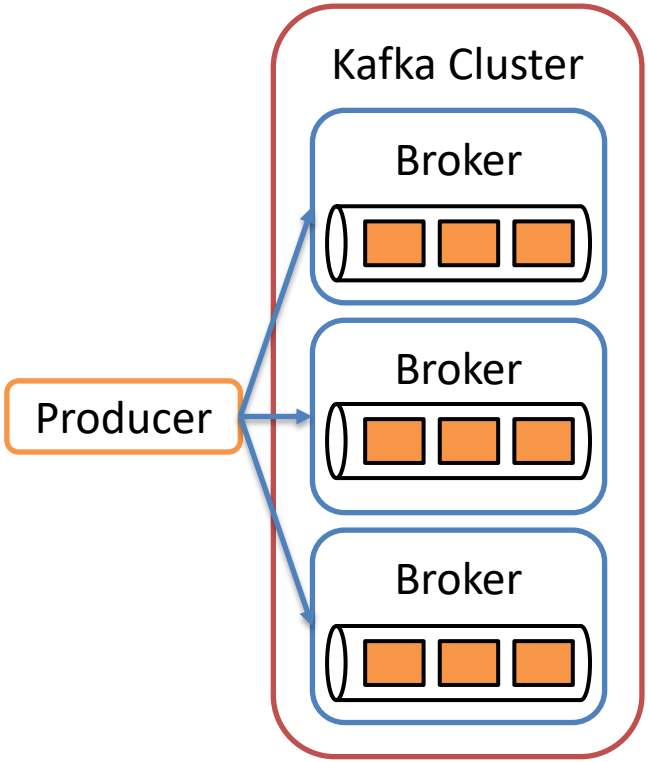
- Communication protocol on top of TCP
- Language-independent
  - Producer and consumer API offered for many languages
- Designed for efficiency

Topics

- Kafka maintains streams of messages called *topics*
- Topics categorize messages into groups
- Developers decide which topics exist
  - By default, a topic is created when it is first used
- One or more producers can write to one or more topics
- There is no limit to the number of topics that can be created

Topics and partitions

- Producers shard data over a set of *partitions*
  - Each partition contains a subset of the topic's messages
  - Each partition is an ordered, immutable log of messages
- Partitions are distributed across brokers



Topics and partitions

- The message key is used to determine which partition a message is assigned to
- Records with the same key are guaranteed to be stored in the same partition
- The partitioning strategy is specified by the producer
  - Default strategy is a hash of the message key
    - Aims to achieve load balance between partitions
  - Developers can provide a custom partitioner class
  - Semantic partitioning: user-specified keys allow locality of data with the same key

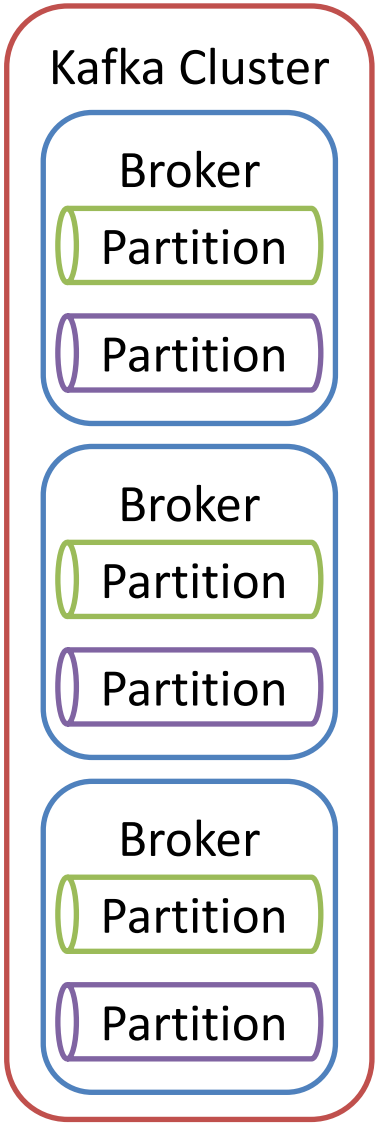
Topics

- Kafka stores all messages for a configurable time
  - Both consumed and not consumed messages

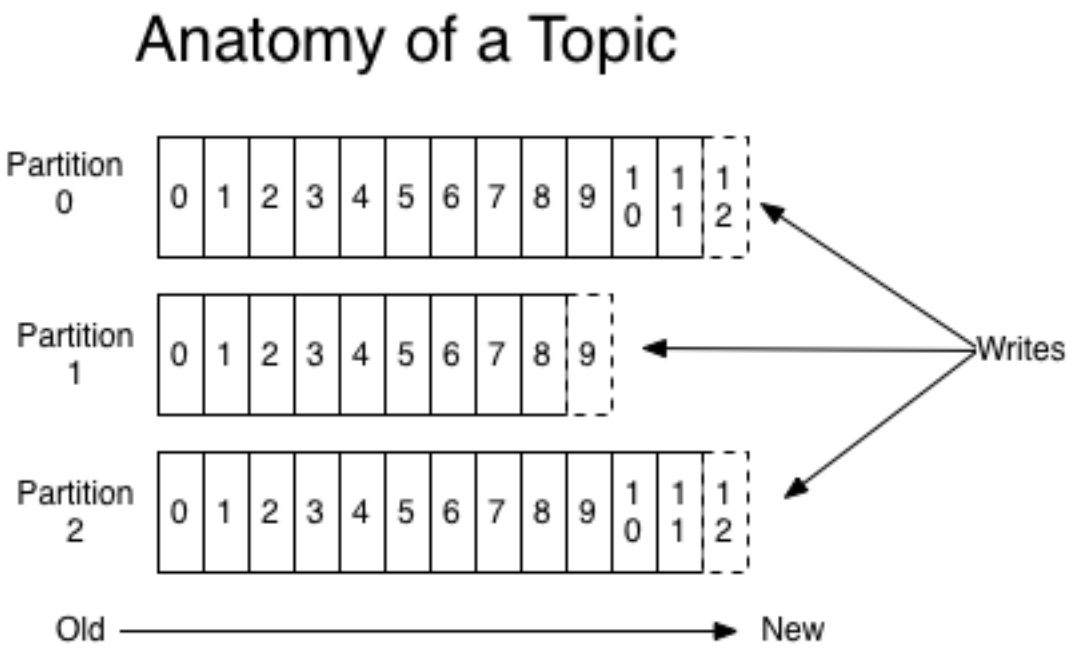
- The performance of Kafka is almost constant and independent on the size of the data stored
  - Consequence: storing data for long time is feasible and inexpensive
  - We will see how this is achieved

Topics and brokers

- Messages in a topic are spread across partitions in different brokers
  - Typically, a broker manages multiple partitions
- Each partition is stored on the broker's disk as one or more log files
- Each message in the log is identified by its offset number
  - A monotonically increasing value
- Kafka provides a configurable retention policy for messages to manage log file growth
  - Retention policies can be configured per topic



Kafka topic


Anatomy of a Topic
Partition 0
0 1 2 3 4 5 6 7 8 9 10 11 12
Partition 1
0 1 2 3 4 5 6 7 8 9
Partition 2
0 1 2 3 4 5 6 7 8 9 10 11 12
Writes
Old
New

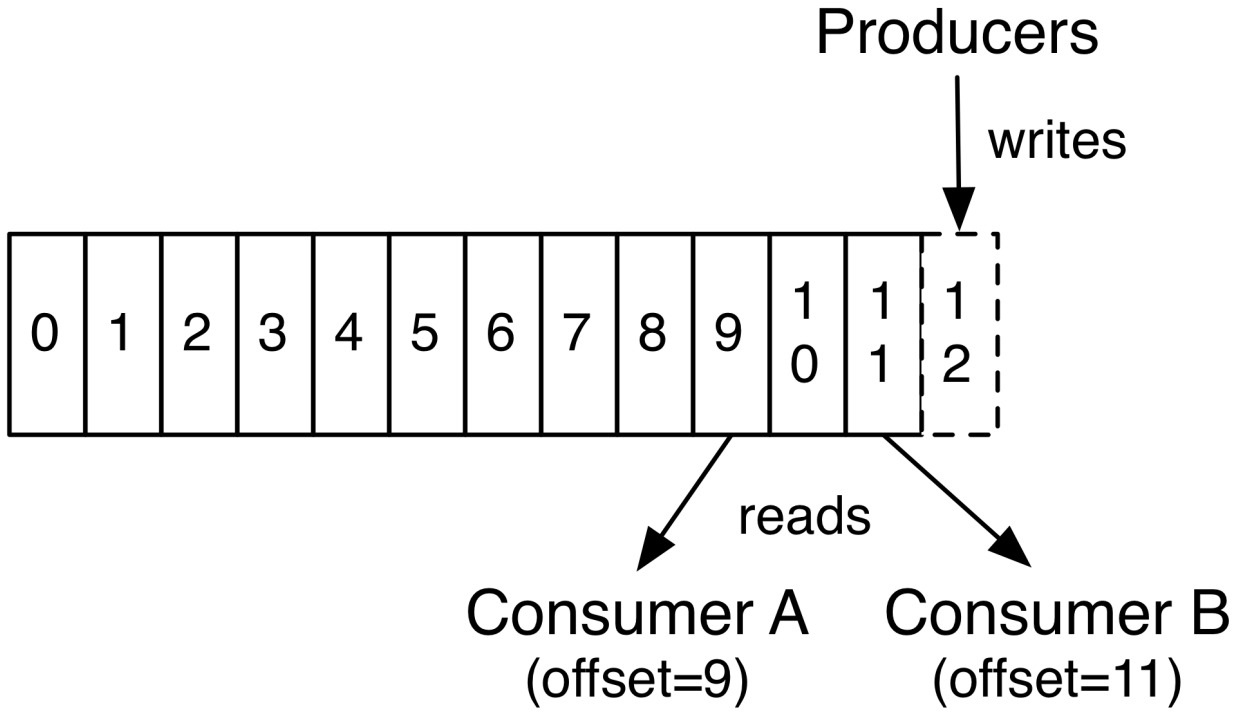
Topics and replication

- Partitions can be replicated across brokers
  - The number of replicas is specified for each topic
- Kafka automatically handles replication
  - Replication only used for fault-tolerance (backup)
  - A leader replica and zero, one or more followers
  - The leader replica handles all requests (both read and write) and propagates changes to the followers
- Replication provides fault-tolerance in case a broker fails
  - A follower takes over in the case the leader fails

Consumers

- Consumers pull messages from one or more topics in the cluster
  - As messages are written to a topic, the consumer will automatically retrieve them
- The *consumer offset* keeps track of the latest message read
  - If necessary, the consumer offset can be changed
    - For example, to re-read old messages
  - It is responsibility of the consumer to store its offset
    - No state management overhead for brokers
    - By default, the consumer offset is also stored in a special Kafka topic

Consumers


Producers
writes
0
1
2
3
4
5
6
7
8
9
10
11
12
reads
Consumer A
(offset=9)
Consumer B
(offset=11)

Consumers

- Different consumers can read data from the same topic
  - By default, each consumer will receive all messages in the topic
  - Easy to add new consumers without impacting on other consumers
- Multiple consumers can be combined into a consumer group
  - Consumer groups provide scaling capabilities
  - Each consumer in a consumer group is assigned a subset of the partitions for consumption

Consumers

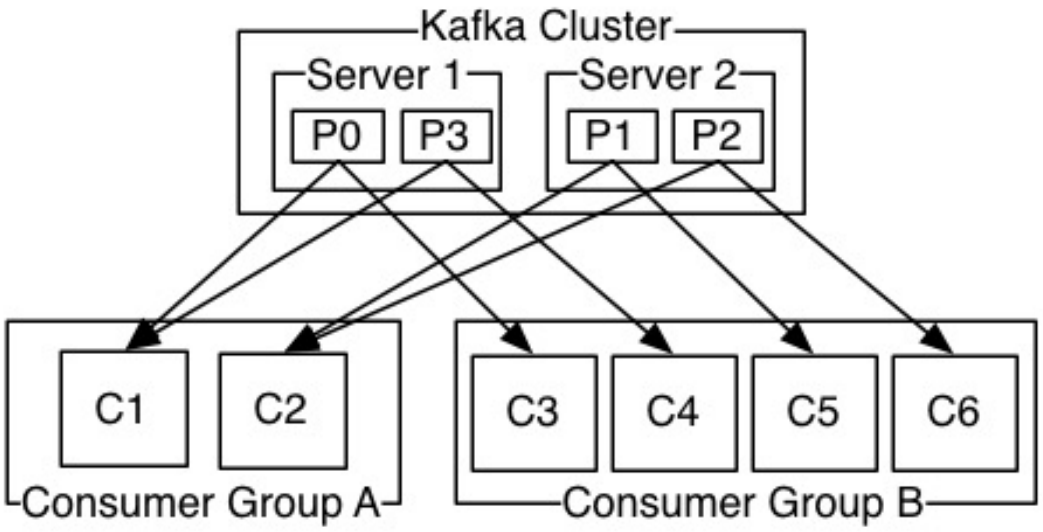
- In summary, partitioning of topics serves different purposes
  - Scaling beyond the disk space of a single node
  - Allowing consumers in a consumer group to read in parallel from different partitions
    - Thus, enabling parallel processing of messages
    - Consumers in a consumer group can be different processes, possibly hosted on different machines
    - Limitation: the number of useful consumers in a consumer group is constrained by the number of partitions on the topic

Consumers

- In practice, each consumer group conceptually represents a single logical consumer
- Each message will be delivered to *each* consumer group interested in the topic
- The message will be delivered to *one* consumer in the consumer group
  - Depending on how the topic is partitioned, …
  - … and how partitions are associated to the consumers within one group

Consumers

- Example
  - Four partitions (P0, P1, P2 , P3)
  - Two consumer groups (A, B)



Guarantees

- Kafka offers the following guarantees

1. Messages from a given producer to a given topic partition are added to that partition in FIFO order

2. The offset within a partition reflects the order in which messages are added to that partition
   - Consumers can read in order from each partition

3. Messages with the same key are guaranteed to be stored in the same partition

4. A topic with replication factor N can tolerate the failure of up to N-1 brokers without losing any information

Zookeeper

- Kafka requires well known, standard protocols to manage the cluster of distributed brokers
  - Cluster membership
  - Failure detection and recovery
  - Agreement on available topics and their configuration
  - …
- To do so, it relies on Zookeeper
  - Open-source Apache project
  - Offers an efficient implementation for several widely used protocols for distributed applications (e.g., membership and leader election)
- This motivates why you need to start Zookeeper when running Kafka
  - Recent versions of Kafka are building internal alternatives to implement the same functionalities under the name of KRaft

Design

Why Kafka is different

- Most traditional event-based / message queuing systems
  - Work in main memory
  - Do not persist data at all …
  - … or delete data once it is consumed
- Kafka offers more functionalities
  - Persistency
  - Replay old messages
- Often with better performance
  - Throughput
  - Latency
  - Scalability

Multiple consumers and scalability

- Some traditional queuing systems remove events after they have been read …
- … instead, Kafka persists messages
  - Can be read by multiple consumers

- Multiple brokers, multiple topics, multiple consumers in consumer groups
  - Flexible design that enables distribution and scalability at different levels

Pull architecture

- Kafka consumers pull messages from brokers
  - This contrasts with many messaging systems, which use a push-based interaction

- Advantages of pulling
  - Brokers do not store any state for the consumers
    - Consumers are responsible for storing the offset of the last position they have read
    - Add more consumers to the system without reconfiguring the cluster
  - Possibility for a consumer to go offline, resuming from where it left off
  - No problem if the consumer is overloaded
    - The consumer can pull and process the data at the speed it can sustain
    - No effects on the producers or on the cluster

Caching for performance

- Unlike other messaging systems, Kafka does not require a lot of memory
  - Logs are stored on disk and read when required
- Simple data structure: sequential log
  - Unlike most databases, which use tree-based data structure for indexed access
  - Sequential reads are the most efficient operations for disks …
- Kafka takes advantage of the operating system's page cache to hold recently-used data
  - Typically, recently-produced data is the data that consumers are requesting
  - Serving the same data to many consumers becomes less expensive
  - This cache remains warm even if the Kafka server process fails and needs to be restarted!

Caching for performance

- Kafka does not deserializes / copies the data into the main memory
  - It uses zero-copy data transfer from the page cache to the network
- A Kafka broker running on a system with a reasonable amount of RAM for the OS to use as cache will typically provide enough performance to saturate its network connection
  - In practice, the network, not Kafka itself, will be the limiting factor on the speed of the system
- In my opinion, these design choices alone are the key factors the contribute to make Kafka so popular
  - Simple data structure on disk
  - Exploit OS memory optimizations …
  - … and avoid replicating them at the application level