Stream processing - part I (Apache Kafka)

Bartosz Łoś
RTB House

Practical Distributed Systems, 2022

What will this lecture be about?

Stream processing

Kafka high-level architecture

- messages, topics, partitions
- brokers, clusters
- producers, consumers
- replication
- log compaction

How does Kafka solve efficiency issues?

How does Kafka ensure delivery guarantees?

Stream processing: events

What an **event** is?

- occurred at some point in time
- immutable object (usually contains a **key** and **value)**
- contains a **timestamp** indicating when it happened
- encoded, stored in database and/or sent over the network in order to process it
- generated by a **producer** and available to be processed by multiple **consumers**
- grouped into a topic or stream

Stream processing: events

What an **stream of events** is?

- container of similar events
- unbounded data, incrementally processed

Uses of streams:

- **data pipelines** that reliably get data between systems or applications
- **real-time applications** that transform or react to streams of data

Stream processing: events

Software processes **events** e.g.:

- server **requests**
- bank **transactions**
- **actions** that users take on a web page
- **messages**
- sensor **measurements**
- web server **logs**
- hardware utilization **metrics** (CPU, memory, disk, network)
- tables **changes** in database, write-ahead logging (WAL)

Stream processing: concept of pub/sub (step 1)

Publish/subscribe messaging system (in short: pub/sub):

- allows for publishing messages by (multiple) producers and subscribing to them by (multiple) consumers
- runs message brokers (servers) optimized for handling message streams with some durability guarantees

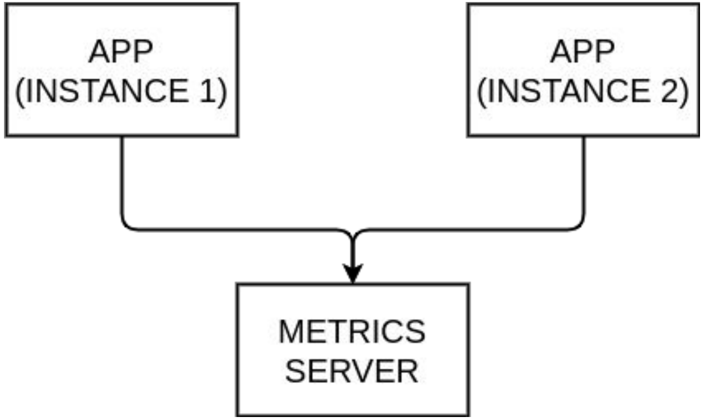
Many use cases for pub/sub start out the same way:

- with a need to implement a **simple message queue** or **interprocess communication** channel

Example: an application that needs to send monitoring information

Stream processing: concept of pub/sub (step 1)

Solution: **creating a direct connection from an application to a monitoring application** (that displays metrics on a dashboard) and pushing metrics over that connection



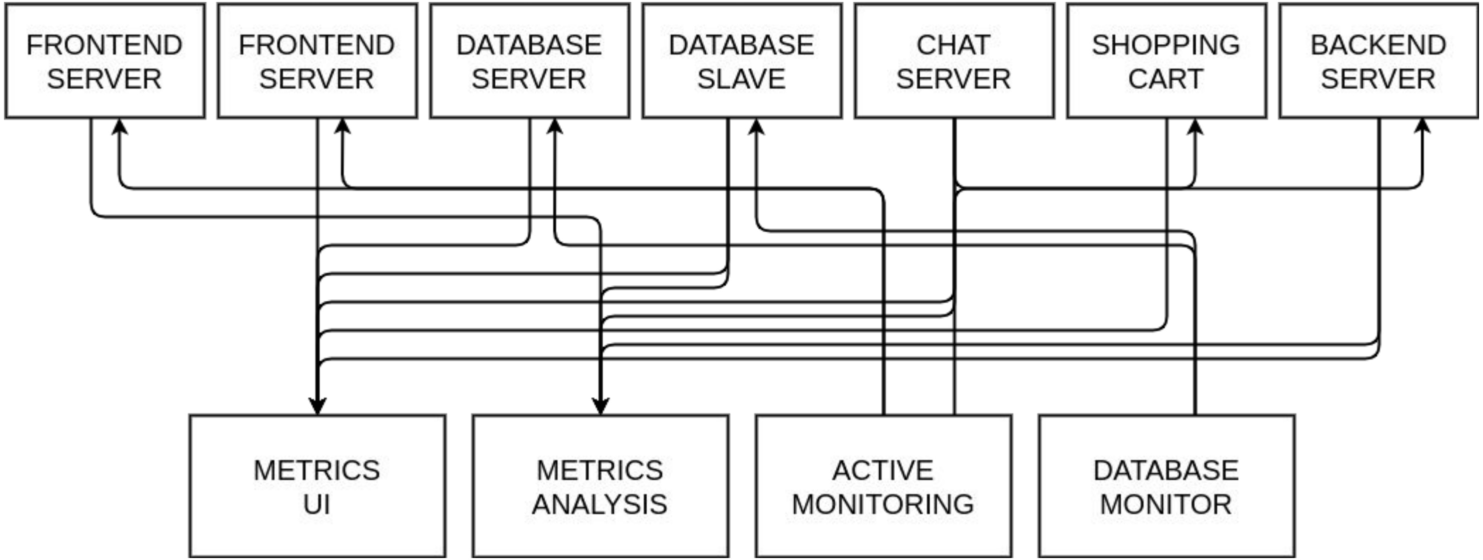
Stream processing: concept of pub/sub (step 2)

New needs: **new business apps and new monitoring apps** (that are getting metrics from individual apps and are using them for various purposes):

- analyzing metrics over a longer term (doesn’t work well in the dashboard)
- active polling of the services for an alerting solution (e.g. for someone on maintenance duty)

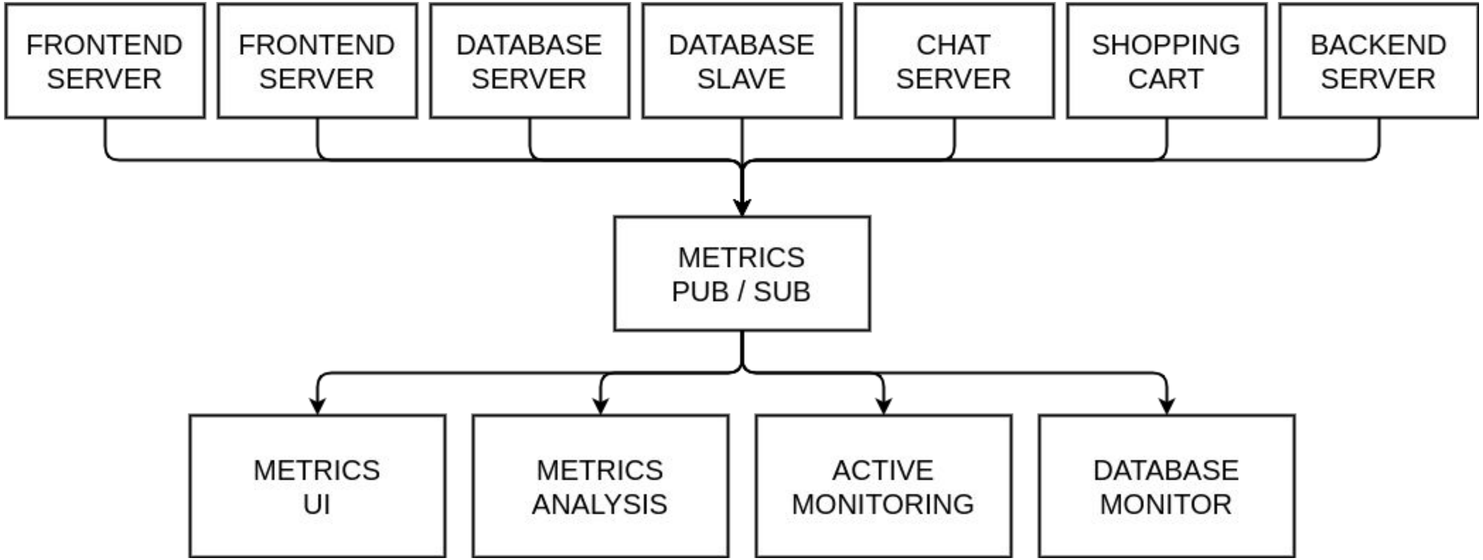
Stream processing: concept of pub/sub (step 2)

Solution: complex **point-to-point connections** between different apps



Stream processing: concept of pub/sub (step 3)

Solution for a complex architecture: setting up a **central application** that receives metrics from all the applications and allows any system to query those metrics



Stream processing: concept of pub/sub (step 4)

New needs: processing various types of data e.g.:

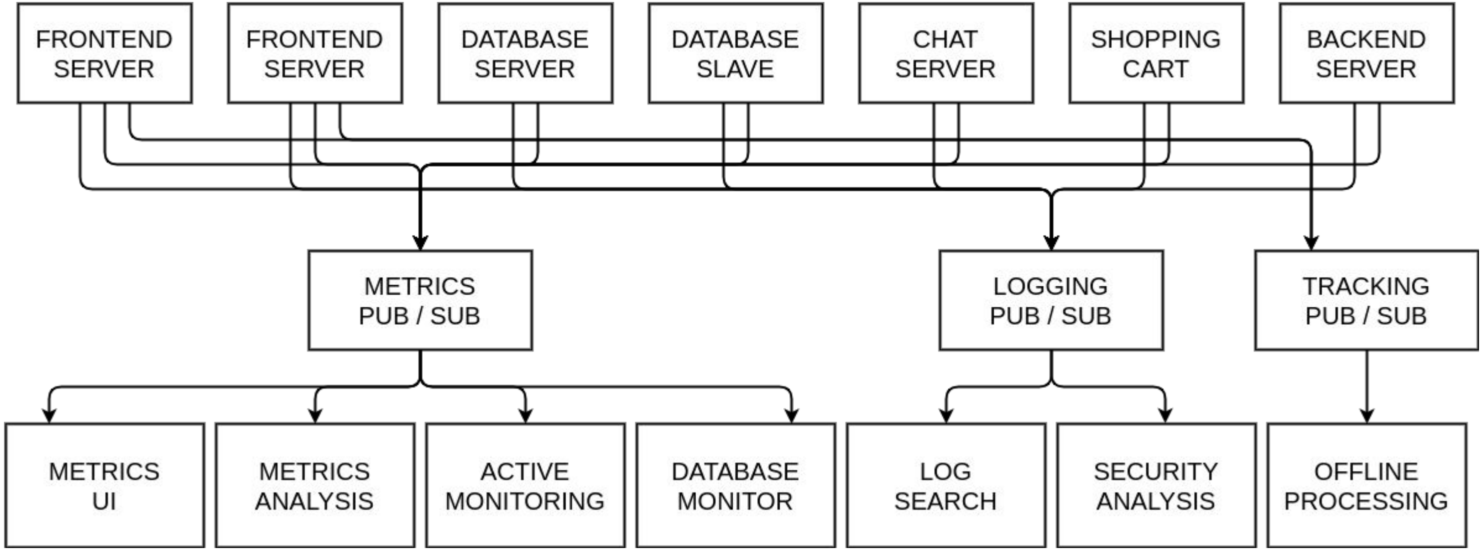
- metrics
- log messages
- system requests

for various purposes e.g.:

- system monitoring
- machine learning
- reports

Stream processing: concept of pub/sub (step 4)

Solution: **a separate pub/sub system for every different flow**



Stream processing: concept of pub/sub (step 5)

Problem:

- problem: maintaining **duplicated** systems for queuing data, all of which have their own individual bugs and limitations

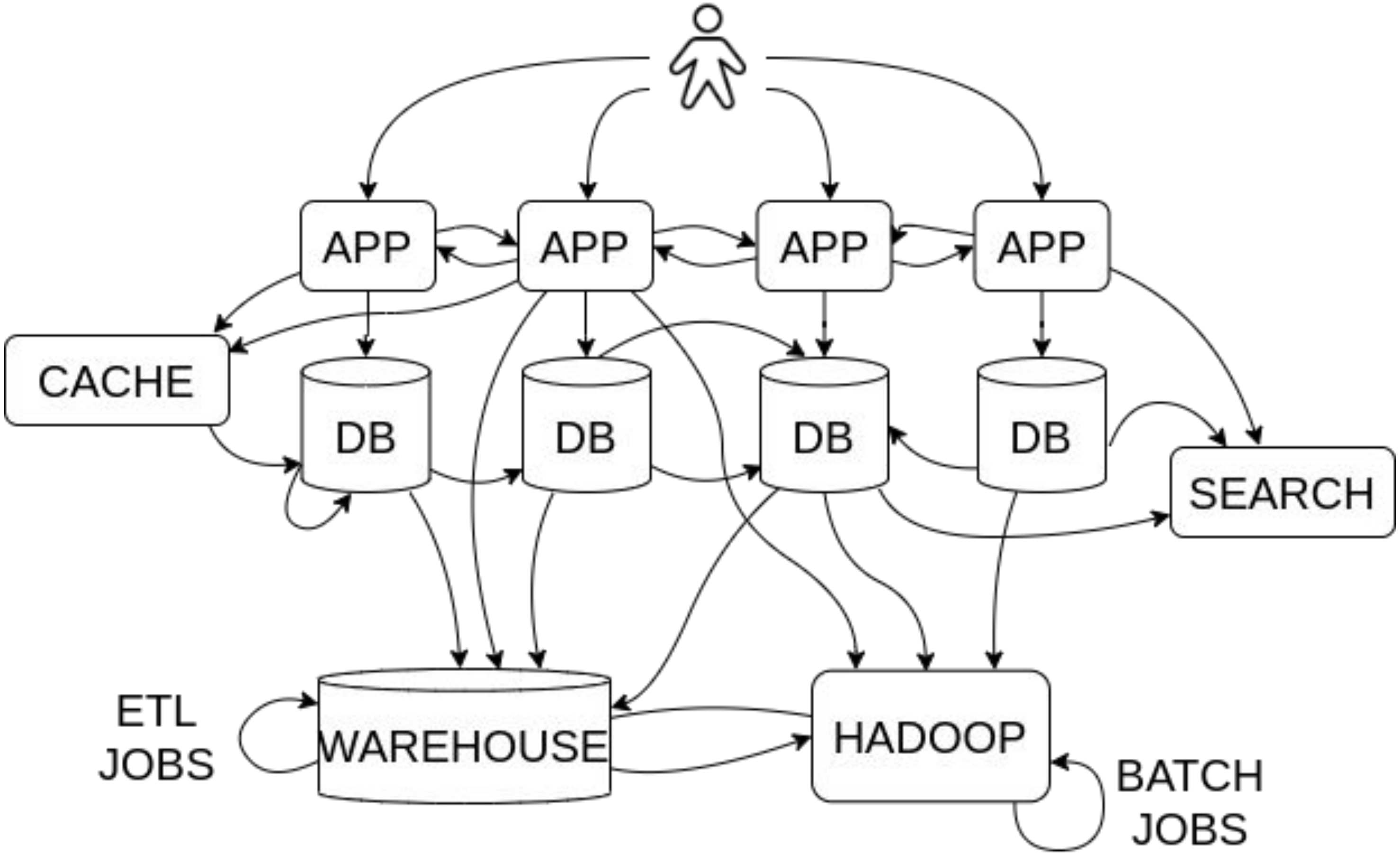
Goal:

- **a centralized system that allows for processing generic types of data** (which will grow as the business grows) and **easier integration** of different applications with a streaming platform in the middle

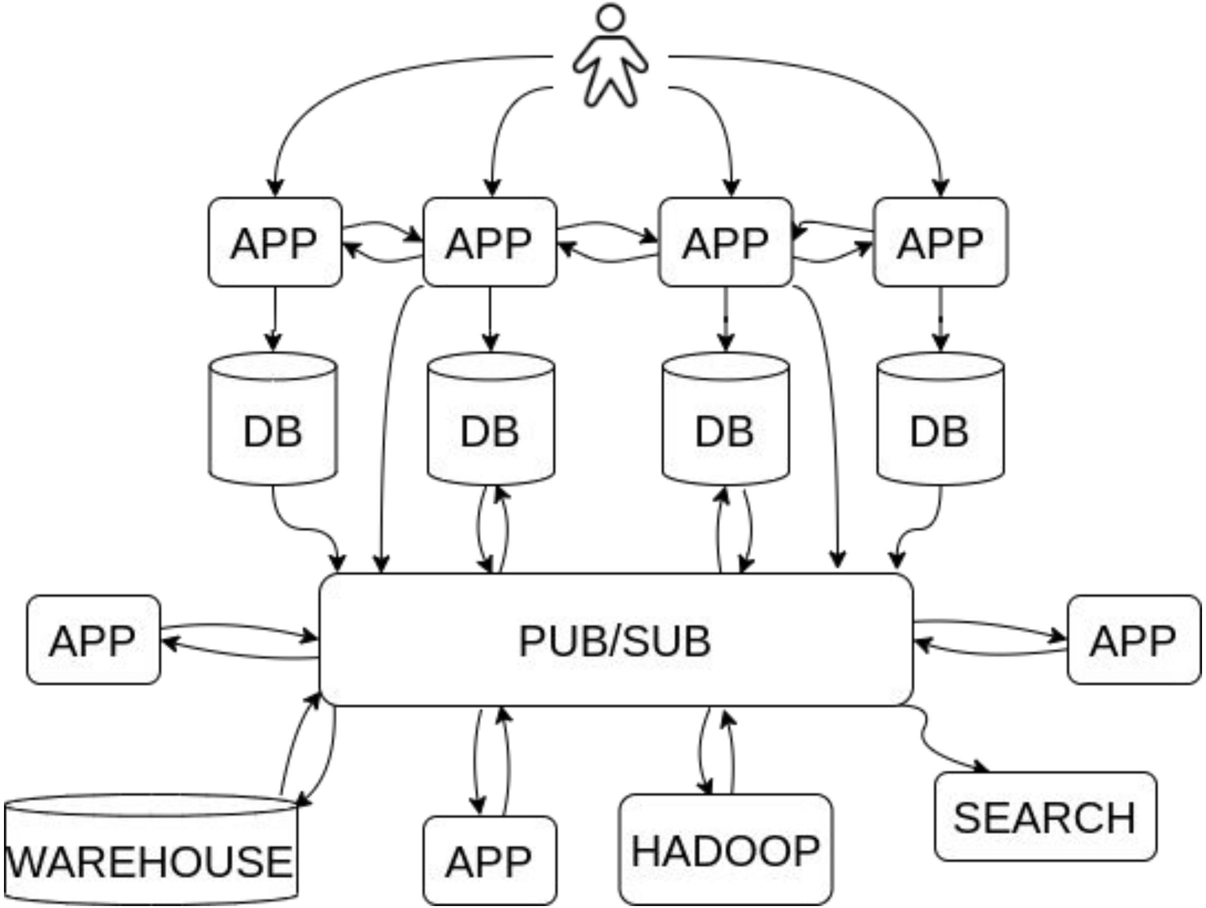
Stream processing: concept of pub/sub (step 5)

Other potential goals:

- shift from system based on the current state to **system which process data as events** (event-driven architecture)
- an **alternative to batch processing** (a lot of data is unbounded)



Practical Distributed Systems, 2022


APP
APP
APP
APP
DB
DB
DB
DB
APP
PUB/SUB
APP
WAREHOUSE
APP
HADOOP
SEARCH

Stream processing: guarantees and issues

Producer and consumer separation: what happens if the producers send messages faster than the consumers can process them?

- dropping messages
- **backpressure** (blocking the producer from sending more messages)
- **asynchronous processing** (buffering in producer-consumer queue)

  producer waits for the broker to confirm that it has buffered the message and does not wait for the message to be processed by consumers

  what happens as that queue grows? does it write messages to the disk? how does the disk access affect the performance?

Stream processing: guarantees and issues

Fault-tolerance: what happens in case of server failures or temporary unavailability of consumers?

- durability requires writing to disk and/or **replication**
- delivery (reliability) semantics: **exactly-once**, **at-least-once** or **at-most-once**

Performance:

- high-throughput (periodic data loads) and low-latency

Scalability:

- scalability of messaging system and **distribution** of processing

Kafka high-level architecture

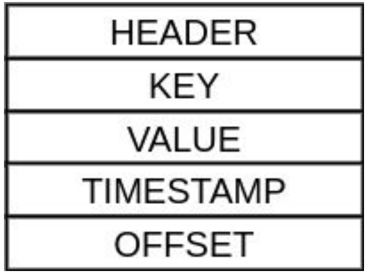
Apache Kafka:

- originally developed at LinkedIn, open-sourced
- an event streaming platform
- used to collect, store and process real-time data streams at scale
- use cases: distributed logging, stream processing, pub-sub messaging

Kafka high-level architecture: messages

Message (also called event or record):

- a **key-value** pair
- keys and values are internally **serialized as byte arrays**
- a key is **not necessarily a unique identifier** for the message

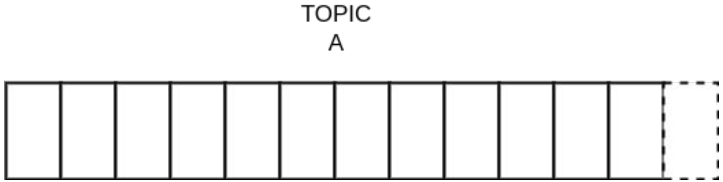


HEADER
KEY
VALUE
TIMESTAMP
OFFSET

Kafka high-level architecture: topics

Messages in Kafka are categorized into **topics**:

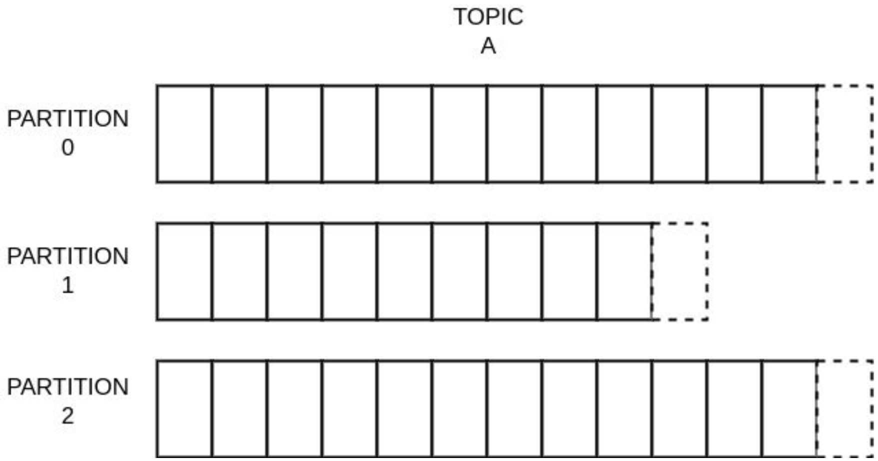
- **durable logs** of messages
- **immutable, append only**
- **can only seek by offset**
- **stateless**: events are not deleted after consumption
- **topic retention**: retaining messages for some period of time (e.g. 7 days) or until the topic reaches a certain size in bytes (e.g. 1 GB)



Kafka high-level architecture: partitions

Topics are broken down into a number of **partitions** (a partition is a single log)

- each partition can be hosted on a different server (a single topic can be **scaled horizontally**)
- client applications (consumers and producers) **read/write the data from/to many brokers** at the same time

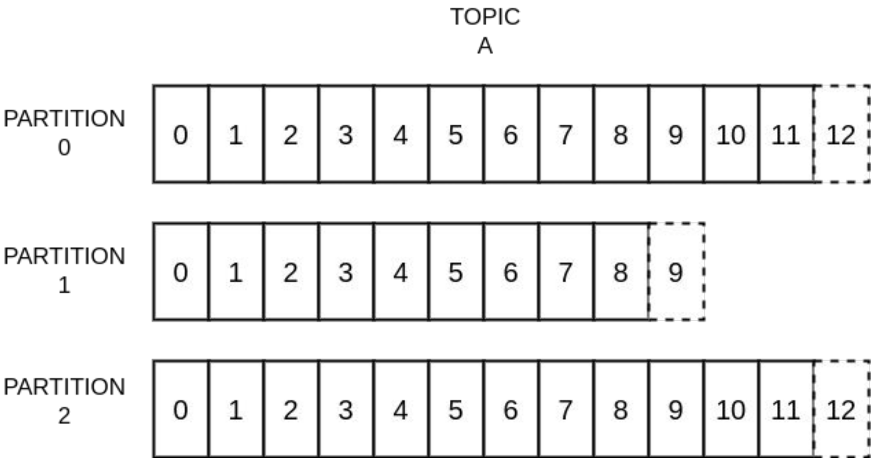


Kafka high-level architecture: partitions

Each message in a given partition has a unique **offset** (an integer value that continually increases for each partition)

Kafka gives the following guarantees:

- messages sent by a producer to a particular partition will be appended **in the order they are sent**
- a consumer instance sees messages **in the order they are stored** in the log



Kafka high-level architecture: brokers and clusters

Kafka is run as a cluster of one or more **brokers** (servers). Each broker:

- receives messages from producers, assigns offsets to them, and writes the messages to storage on disk
- responds to fetch requests for partitions (from consumers) and returns messages

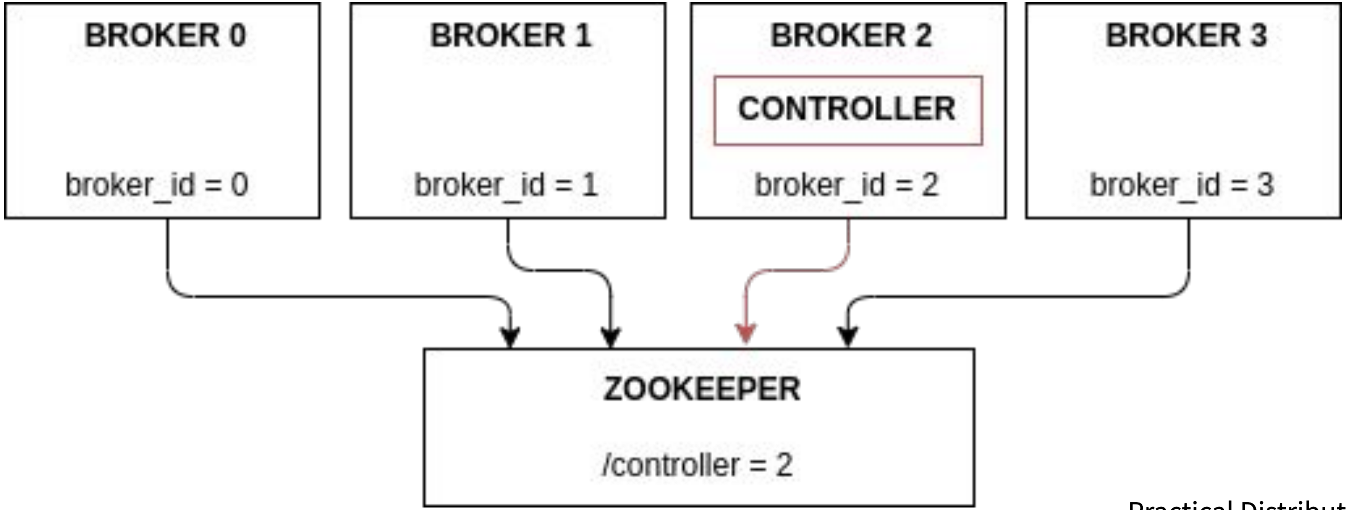
One of the brokers functions as the **cluster controller** responsible for administrative operations like:

- assigning partitions to brokers
- monitoring for broker failures

Kafka high-level architecture: brokers and clusters

A key-value store (Apache Zookeeper) is used to elect the controller:

- every broker tries to write its value to /controller path in Zookeeper
- the one which wins the creation of this path becomes the controller



Kafka high-level architecture: producers

A **producer** (also called publisher or writer) produces messages, **controls which partition it publishes messages to** and sends messages directly to the broker

Default **partitioner:**

- if no partition is specified but a key is present, choose a partition based on a hash of the key (messages with the same key are always written to the same partition):

  partition(event) = hash(event.key) % number of partitions

- if no partition or key is present, choose the sticky partition that changes when the batch is full, balancing messages (batches) over all partitions of a topic evenly

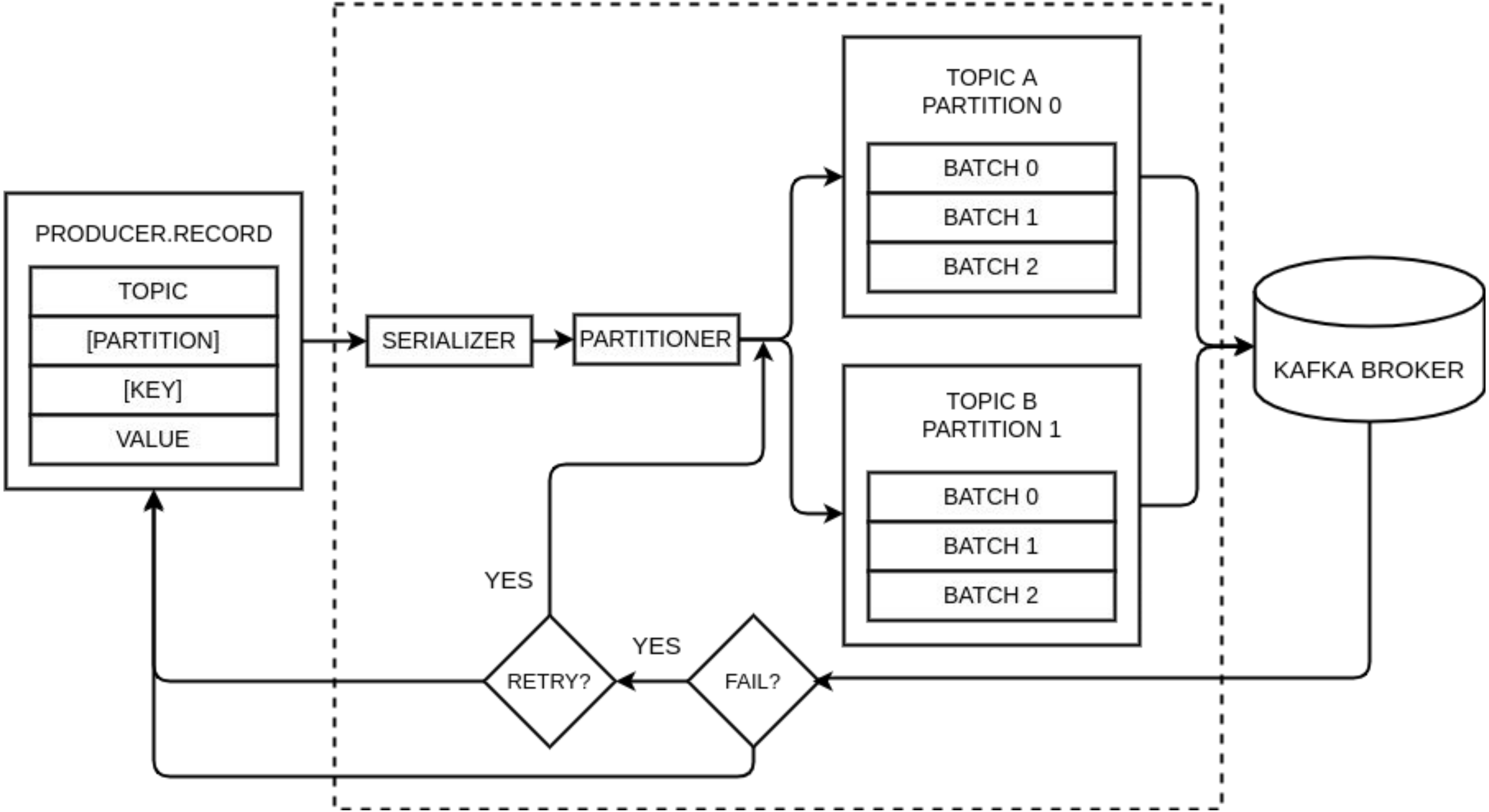
Kafka high-level architecture: producers

Producer accumulates data in memory and sends out larger **batches in a single request**, can be configured (additional latency vs better throughput):

- to accumulate no more than a fixed number of messages (e.g. 64k)
- to wait no longer than some fixed time (e.g. 10 ms)

The broker sends back a response:

- when successful: RecordMetadata (topic, partition and message offset)
- when unsuccessful: error (producer may retry sending the message a few more times)



Practical Distributed Systems, 2022

Kafka high-level architecture: consumers

A **consumer** (also called subscriber or reader):

- subscribes to one or more topics and gets messages in the order in which they were produced
- keeps track of which messages it has already consumed and issues fetch requests specifying its **offset** in the log

Pull based consumption (optimal batching without introducing unnecessary latency):

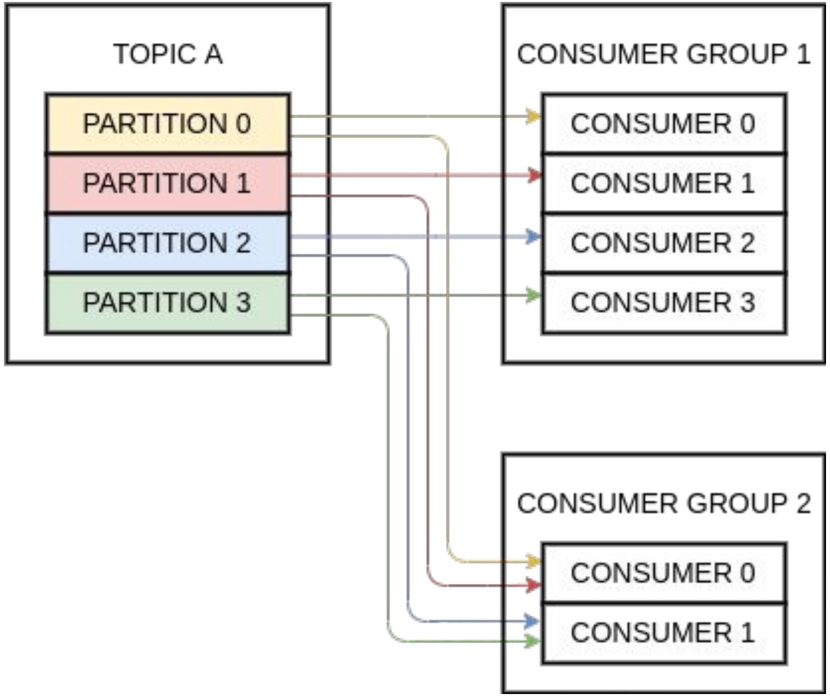
- consumer always pulls all available messages after its current position in the log (or up to some configurable max size)
- to avoid polling in a tight loop request could be blocked waiting until data arrives

Kafka high-level architecture: consumers

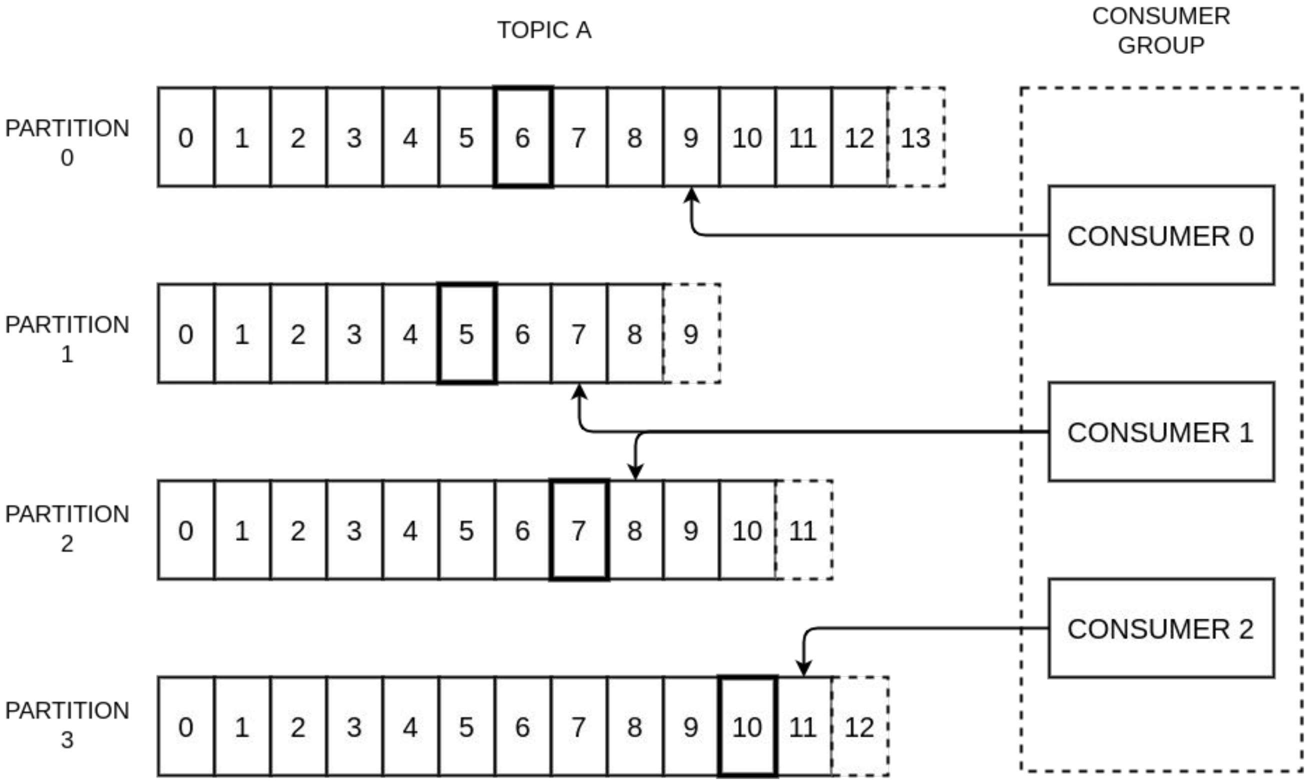
Consumers work as part of a **consumer group**:

- one or more consumers that work together to consume a topic, the group ensures that each partition is only consumed by one member
- if a single consumer fails, the remaining members of the group will rebalance and continue from the last **committed offset**

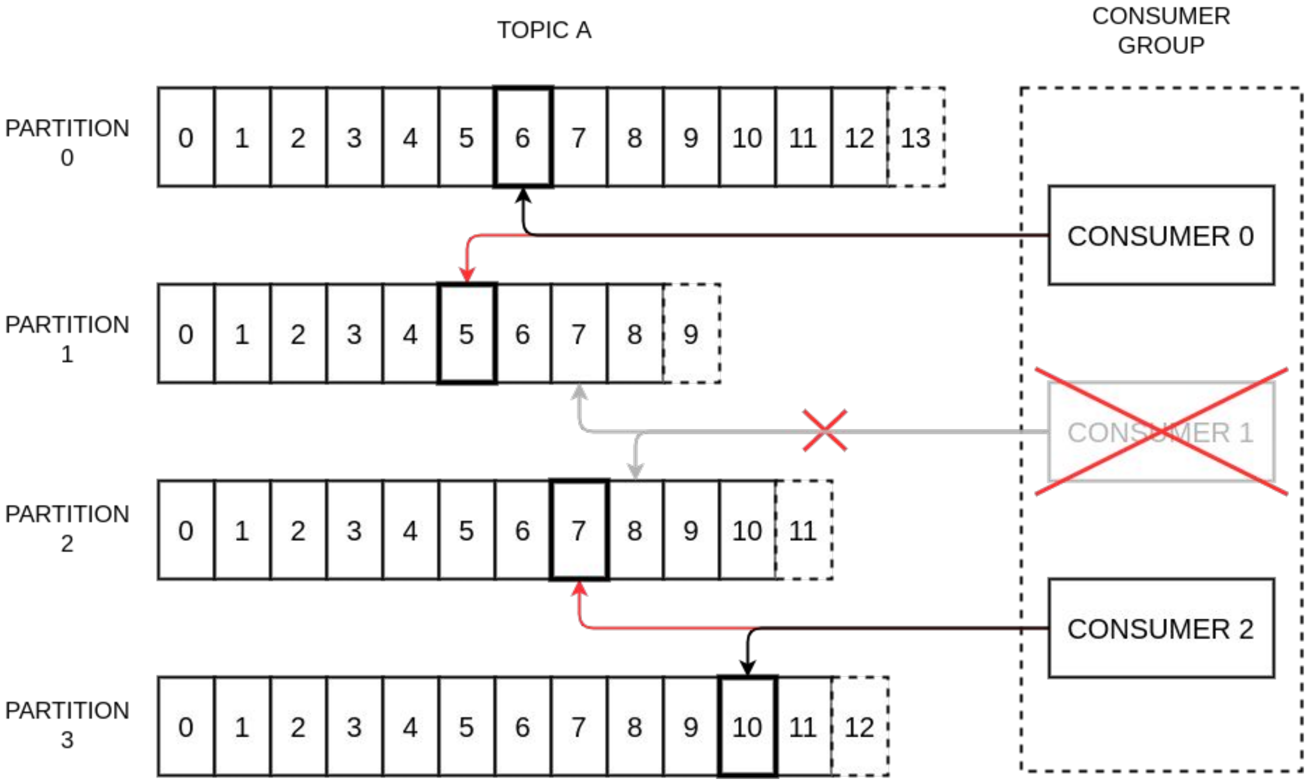
Consumers maintain membership in a consumer group by sending heartbeats to a **group coordinator** (one of the brokers, different for different consumer groups)



Kafka high-level architecture: consumers



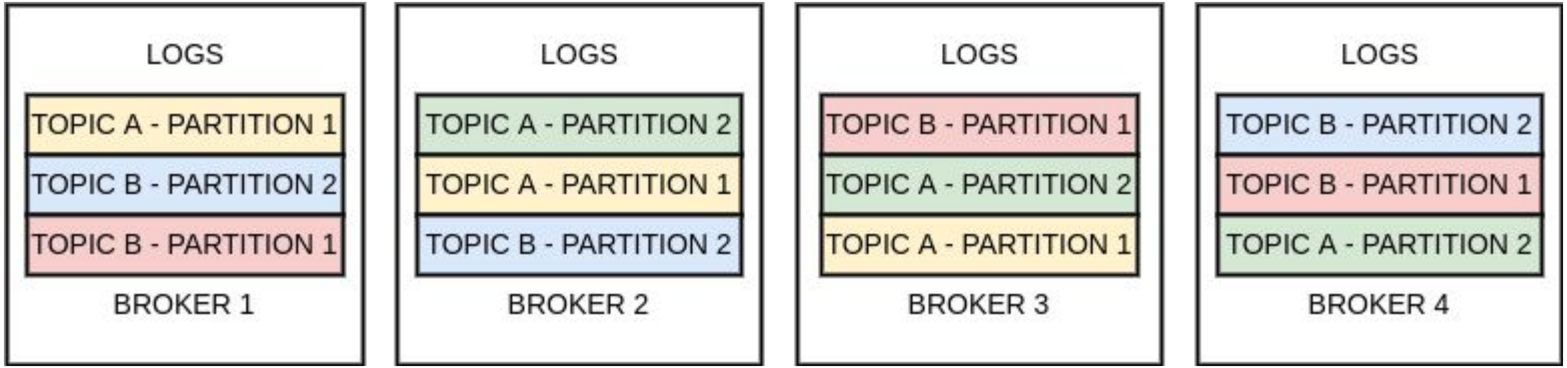
Kafka high-level architecture: consumers



Practical Distributed Systems, 2022

Kafka high-level architecture: replication

A partition may have **replicas** distributed among brokers



LOGS

TOPIC A - PARTITION 1
TOPIC B - PARTITION 2
TOPIC B - PARTITION 1

BROKER 1

LOGS

TOPIC A - PARTITION 2
TOPIC A - PARTITION 1
TOPIC B - PARTITION 2

BROKER 2

LOGS

TOPIC B - PARTITION 1
TOPIC A - PARTITION 2
TOPIC A - PARTITION 1

BROKER 3

LOGS

TOPIC B - PARTITION 2
TOPIC B - PARTITION 1
TOPIC A - PARTITION 2

BROKER 4

Kafka high-level architecture: replication

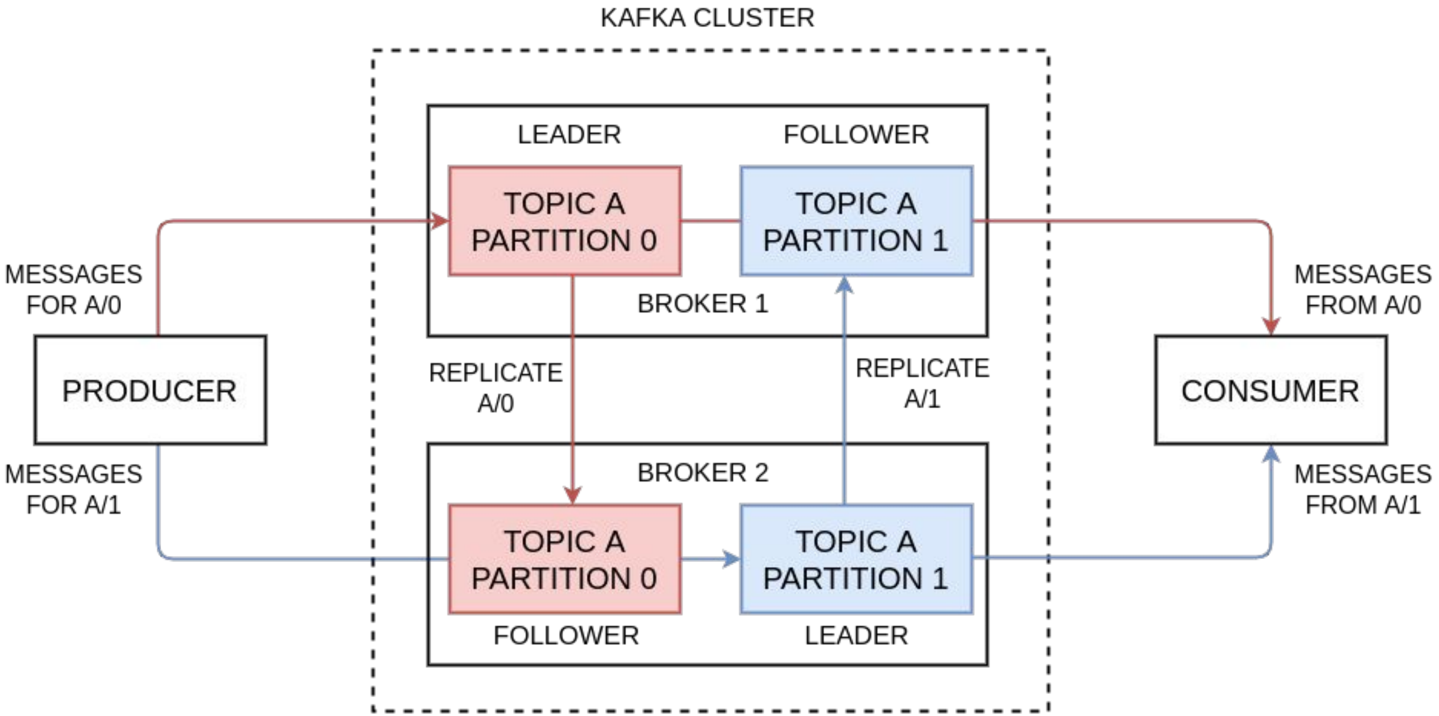
Each partition has a single **leader** and (zero or more) **followers**:

- all consumers and producers operating on that partition must connect to the leader (write to and read from)
- followers consume messages from the leader (just as normal consumers), the logs on the followers are identical to the leader's log

Another broker can take over leadership if there is a broker failure:

- for a topic with **replication factor** (the total number of replicas including the leader) N, Kafka will tolerate up to N-1 server failures without losing any **committed** messages
- the controller is responsible for electing a new leader and propagating this information

Kafka high-level architecture: replication



Kafka high-level architecture: replication

In sync replica:

- **a node is alive**: must be able to maintain its session with ZooKeeper
- **not too far behind the leader**: must replicate the writes happening on the leader

The leader keeps track of the set of **in sync replicas (ISR)** persisted to ZooKeeper:

- when the leader dies, only members of this set are eligible for election as leader
- in case of crash before rejoining, it must fully resync again

Kafka high-level architecture: replication

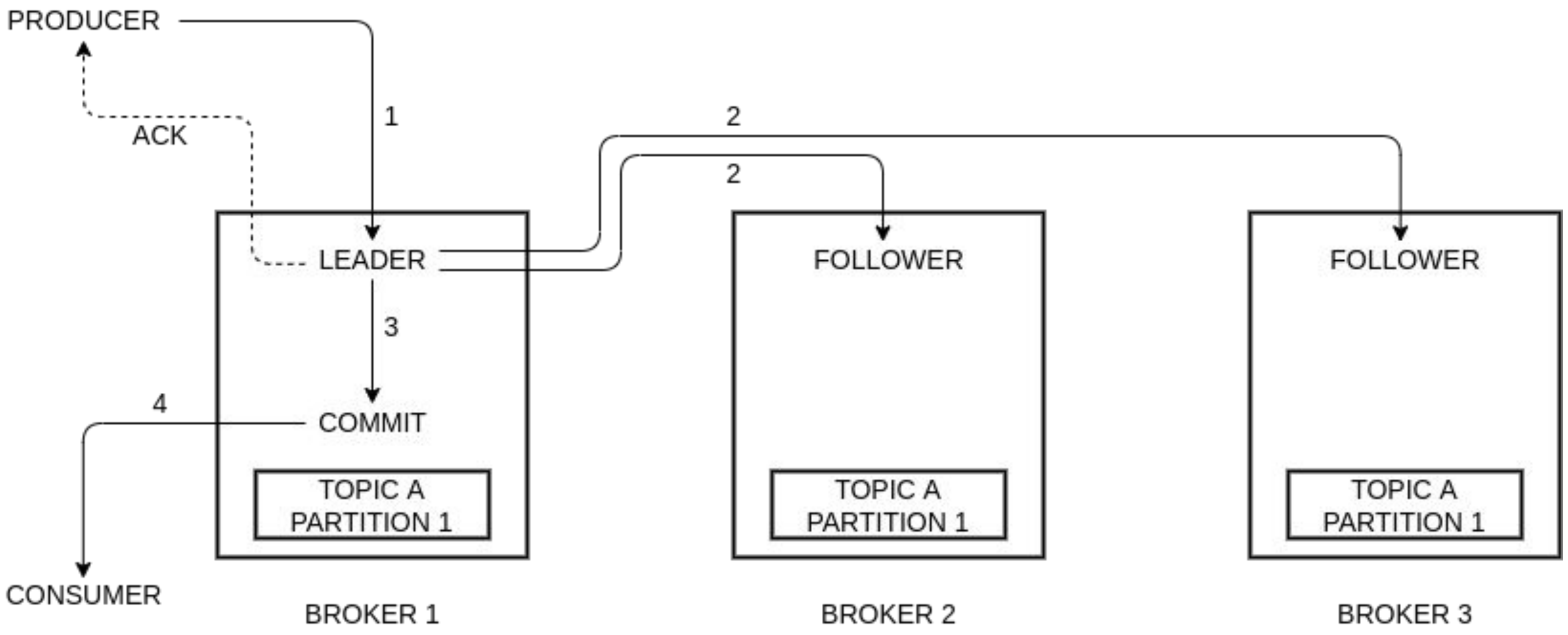
Only committed messages are ever given out to the **consumer**:

- a message is **committed** when all in sync replicas have applied it to their log

Producer can choose (controlled by the **acks** setting) whether it waits for the message to be acknowledged by:

- **0** replicas
- **1** replicas
- **all (-1)** replicas (means all the current in-sync replicas)

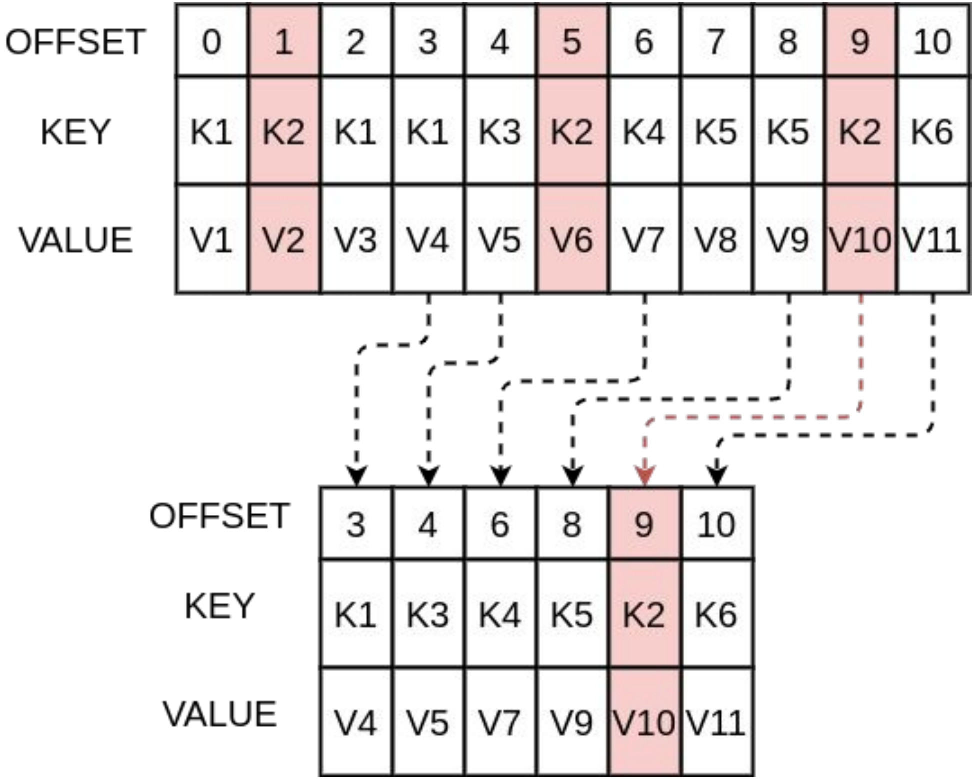
Kafka high-level architecture: replication



Kafka high-level architecture: log compaction

Log compaction:

- retains the **last message** produced with a specific key (when only the last update is interesting)
- allows for deletes (**tombstone** is a message with a key and a null value)



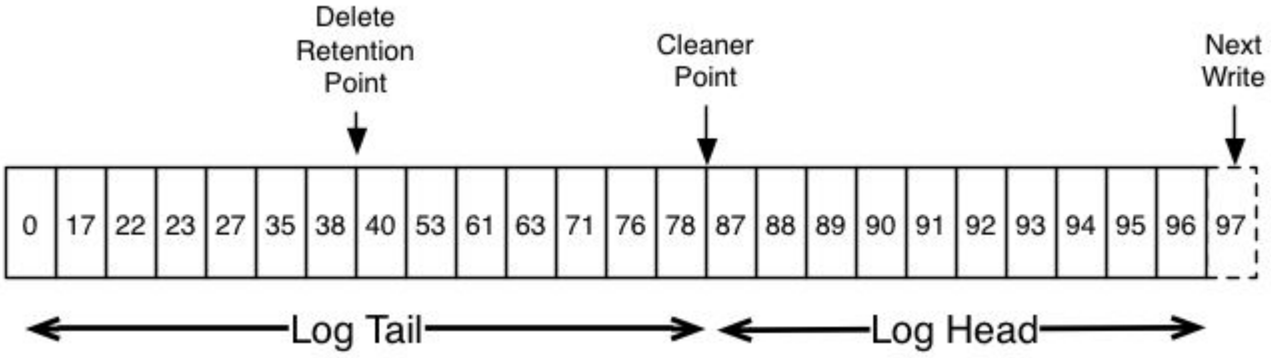
OFFSET	0	1	2	3	4	5	6	7	8	9	10
KEY	K1	K2	K1	K1	K3	K2	K4	K5	K5	K2	K6
VALUE	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11

OFFSET	3	4	6	8	9	10
KEY	K1	K3	K4	K5	K2	K6
VALUE	V4	V5	V7	V9	V10	V11

Kafka high-level architecture: log compaction

The compaction is done in the background by periodically recopying log segments:

- the **head** (retains all messages with sequential offsets)
- the **compacted tail** (retains the original offset assigned to the message)



How does Kafka solve efficiency issues?

Poor disk access patterns:

- **reads and appends to files** (instead of random access data structures like BTree)

  context: HDD performance, throughput of linear writes vs latency of disk seeks (e.g. 600MB/sec vs 100k seeks/sec in case of 7200rpm SATA RAID-5)

- using **pagecache instead of maintaining an in-memory cache** (all data is immediately written to a persistent log on the filesystem without flushing it)

  context: OS optimizations (read-ahead and write-behind, all disk reads and writes except for direct I/O go through OS pagecache)

How does Kafka solve efficiency issues?

Poor disk access patterns:

- storing a **compact byte structure** rather than individual objects

  context: JVM memory overhead of objects, slow GC as the in-heap data increases

- **no instant deletes** (retain messages for some time)

  context: reads do not block writes or each other

How does Kafka solve efficiency issues?

Too many small I/O operations (between client and server and in the server's own operations):

- protocol allows for **grouping messages into batches**

As a result:

- larger network packets and network round trips limited
- larger sequential disk operations and contiguous memory blocks
- larger linear chunks fetched by consumer

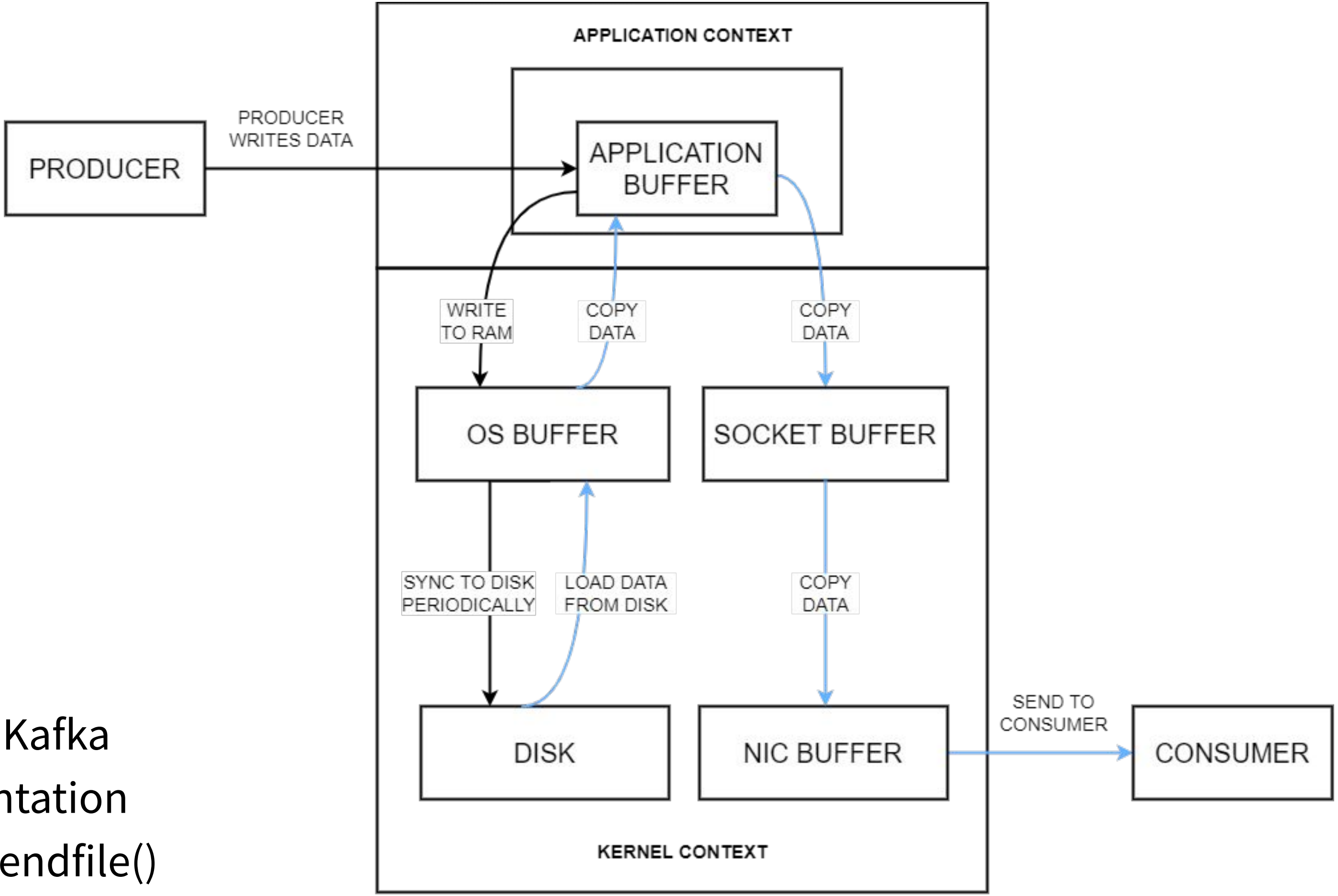
How does Kafka solve efficiency issues?

Excessive byte copying:

- **common binary message format** (shared by the producer, the broker and the consumer; data chunks can be transferred without modification between them)
- **allowing the OS to send the data from pagecache to the network directly** with sendfile() system call

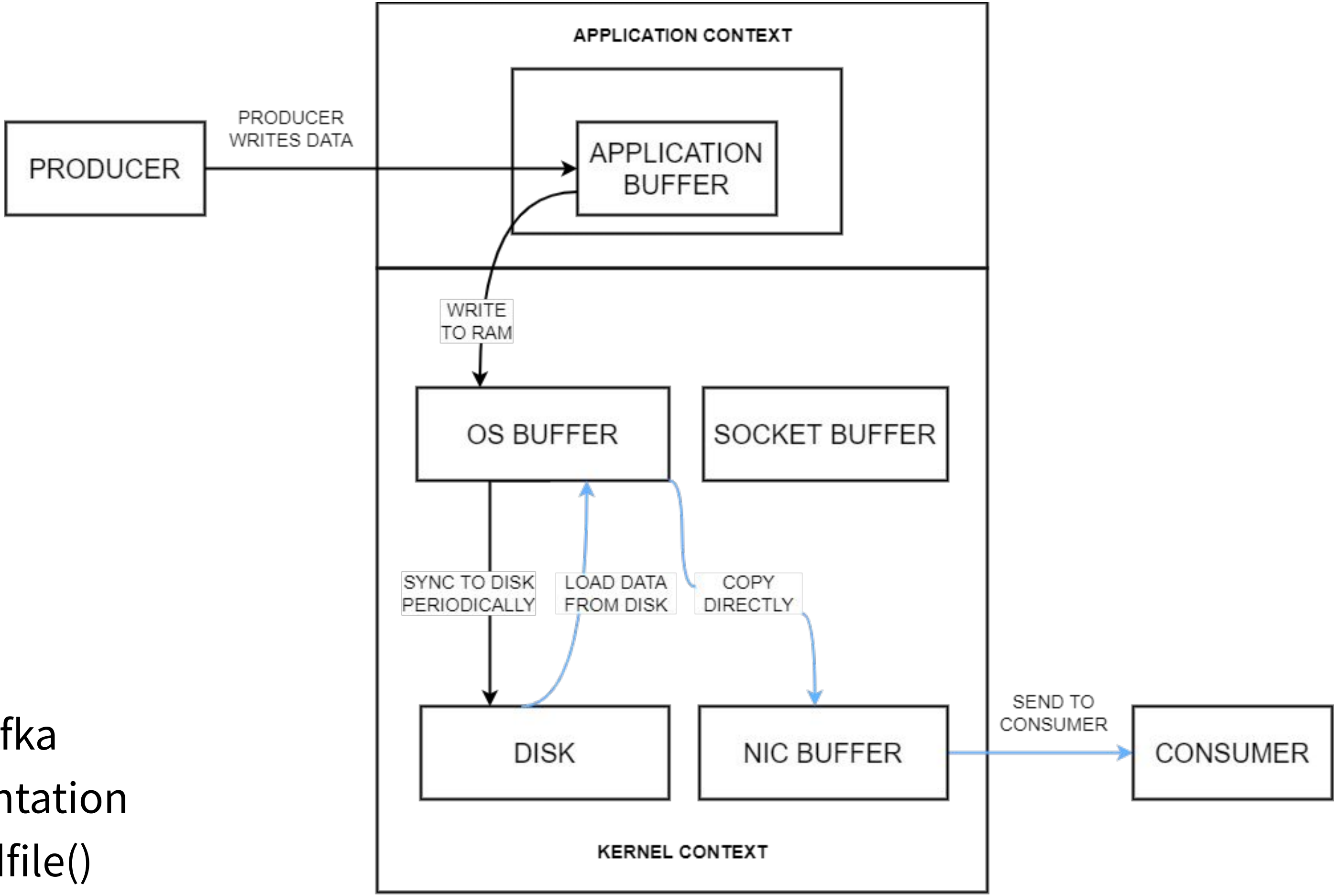
Context: common data path for transfer of data from file to socket:

- OS reads data from the disk into pagecache in kernel space
- app reads the data from kernel space into a user-space buffer
- app writes the data back into kernel space into a socket buffer
- OS copies the data from the socket buffer to the NIC buffer (network interface controller) where it is sent over the network



Potential Kafka implementation without sendfile() system call

Practical Distributed Systems, 2022



Actual Kafka implementation with sendfile() system call

How does Kafka solve efficiency issues?

Network bandwidth optimization: **end-to-end batch compression**

- efficient compression requires **compressing multiple messages together** rather than compressing each message individually

The batch of messages will be:

- sent by the producer in compressed form
- written in compressed form to the log
- decompressed by the consumer

How does Kafka ensure delivery guarantees?

“There are only two hard problems in distributed systems:
2. Exactly-once delivery 1. Guaranteed order of messages 2. Exactly-once delivery”

Two issues:

- the **durability guarantees** for publishing a message
- the **processing guarantees** when consuming a message

How does Kafka ensure delivery guarantees?

If a producer attempts to publish a message and experiences a network error it cannot be sure if this error happened before or after the message was committed:

- **resending** the message means **at-least-once** delivery semantics

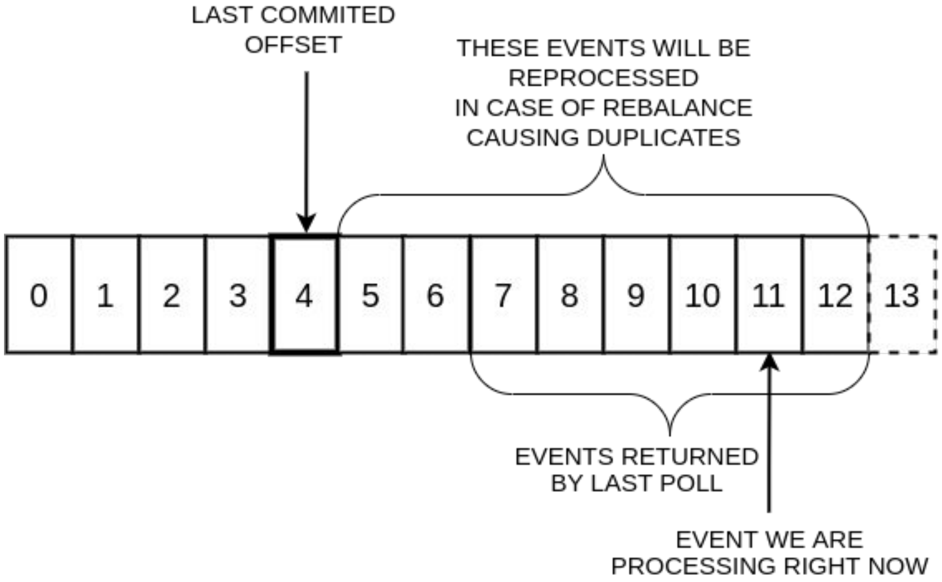
The Kafka producer supports an **idempotent** delivery option which guarantees that resending will not result in duplicate entries in the log:

- the broker assigns each producer an ID and deduplicates messages using a sequence number that is sent by the producer along with every message

How does Kafka ensure delivery guarantees?

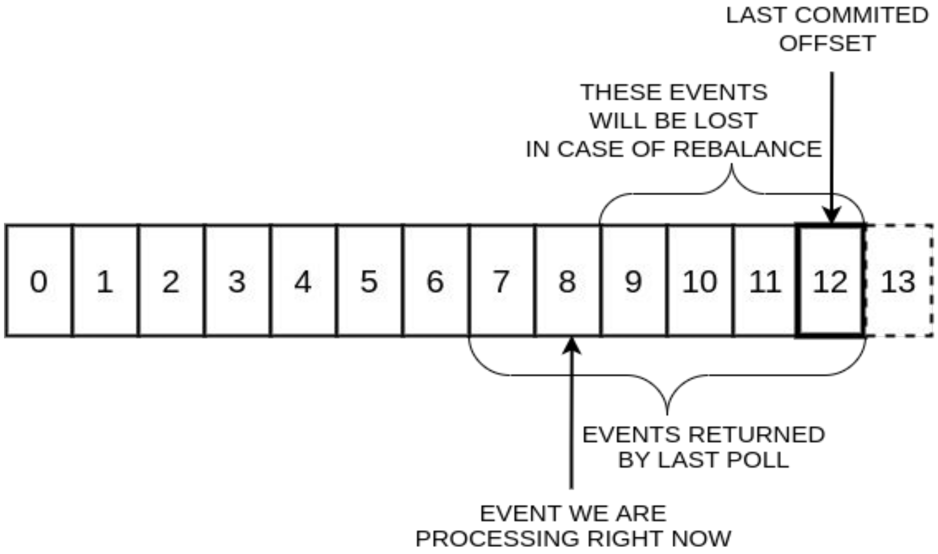
If a consumer never crashed it could just store this position in memory, but if the consumer fails and we want this partition to be taken over by another process the new process will need to choose an appropriate position from which to start processing

At-least-once: read the messages, process the messages, and finally save its position


LAST COMMITED OFFSET
THESE EVENTS WILL BE REPROCESSED IN CASE OF REBALANCE CAUSING DUPLICATES
0
1
2
3
4
5
6
7
8
9
10
11
12
13
EVENTS RETURNED BY LAST POLL
EVENT WE ARE PROCESSING RIGHT NOW

How does Kafka ensure delivery guarantees?

At-most-once: read the messages, then save its position in the log, and finally process the messages


LAST COMMITED OFFSET
THESE EVENTS WILL BE LOST IN CASE OF REBALANCE
0
1
2
3
4
5
6
7
8
9
10
11
12
13
EVENTS RETURNED BY LAST POLL
EVENT WE ARE PROCESSING RIGHT NOW

How does Kafka ensure delivery guarantees?

Exactly-once when consuming from a Kafka topic and producing to another topic:

- the ability to send messages to multiple partitions using transactions
- the consumer's position is stored as a message in a topic, so we can write the offset to Kafka in the same transaction as the output topics receiving the processed data
- in the default **read_uncommitted** isolation level all messages are visible to consumers even if they were part of an aborted transaction
- in the **read_committed** isolation level the consumer will only return messages from transactions which were committed (and any messages which were not part of a transaction)

How does Kafka ensure delivery guarantees?

Exactly-once when writing to an external system

Option 1:

- a two-phase commit between the storage of the consumer position and the storage of the consumers output

Option 2:

- letting the consumer store its offset in the same place as its output

Summary

We have discussed:

- stream processing as a paradigm and potential applications of event-driven architectures
- Kafka high-level architecture and motivations behind design decisions
- Kafka implementation details with regard to its performance and delivery guarantees