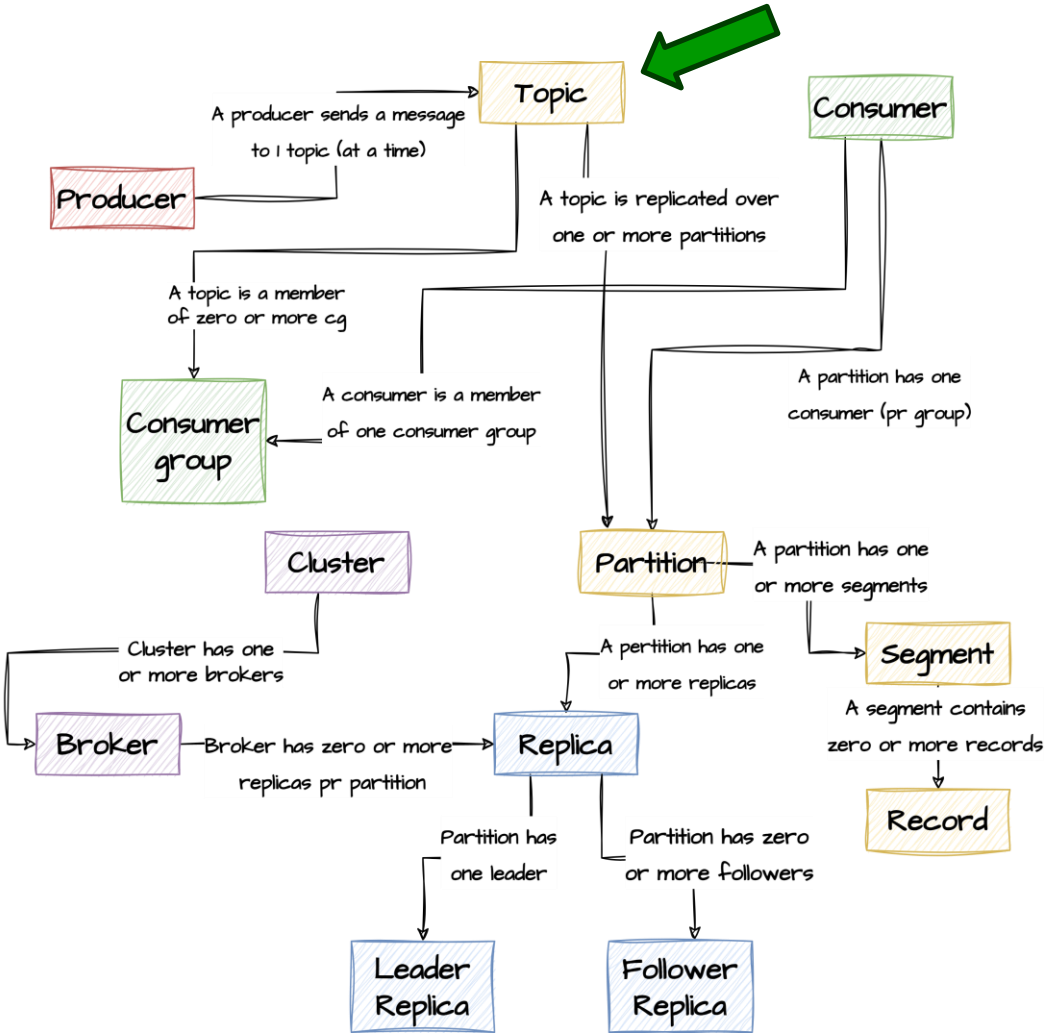
Lund&Bendsen
developing developers

Kafka Broker

Topics

Lund&Bendsen
developing developers

- Topics
- Partitions
- Log segments
- Optimizations
- Misc



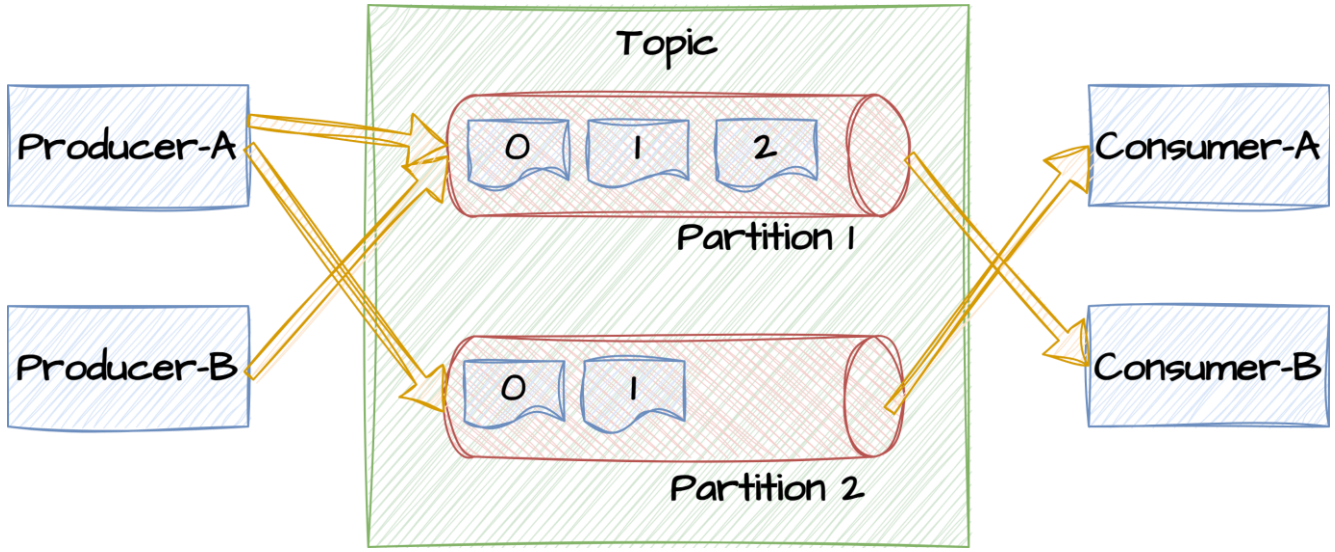
Topics

Lund&Bendsen
developing developers

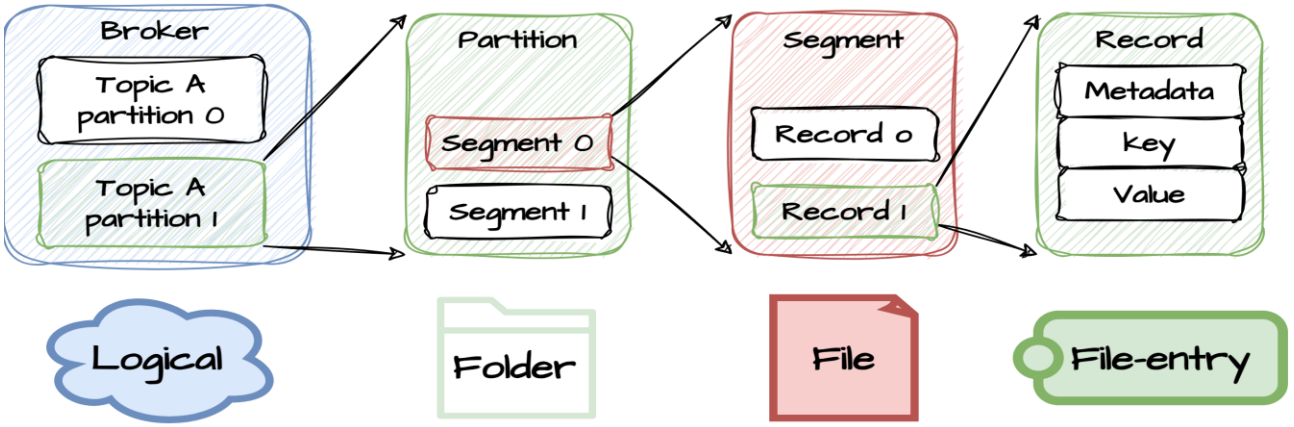
In Apache Kafka, topics are fundamental constructs that play a central role in organizing and categorizing messages (events) within the Kafka messaging system. Topics provide a way to classify and publish messages, making it easier for producers to send data to specific categories and for consumers to subscribe to the data they are interested in.

Kafka topics act as logical channels or categories to which messages are published by producers and from which messages are consumed by consumers.

Topics are used to categorize and organize messages, making it possible to segregate different types of data within a Kafka cluster.



Topics – partitioning



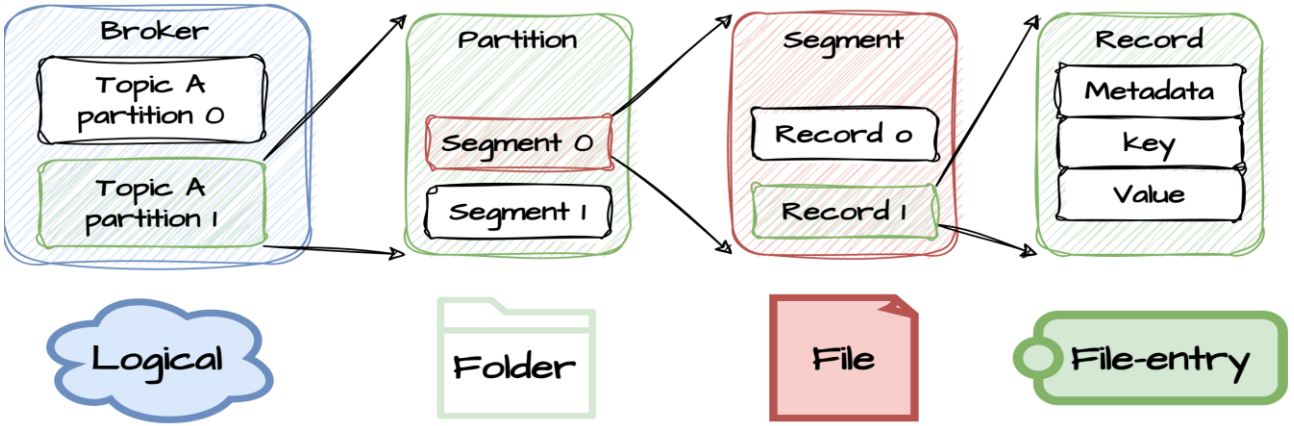
Named and Identified:

- A topic is a logical grouping of partitions of data located on different brokers
- Topics are identified by a unique name within the Kafka cluster. Producers and consumers refer to topics by their names when sending or receiving messages.
- Topic names are case-sensitive and are typically configured as strings.

Partitioning:

- Kafka topics can be divided into multiple partitions. Partitions allow for parallelism, scalability, and efficient message processing.
- Each partition is an ordered, immutable sequence of records. Records within a partition are assigned a unique offset that indicates their position within the partition.

Topics – parallelism & retention



Parallelism and Scalability:

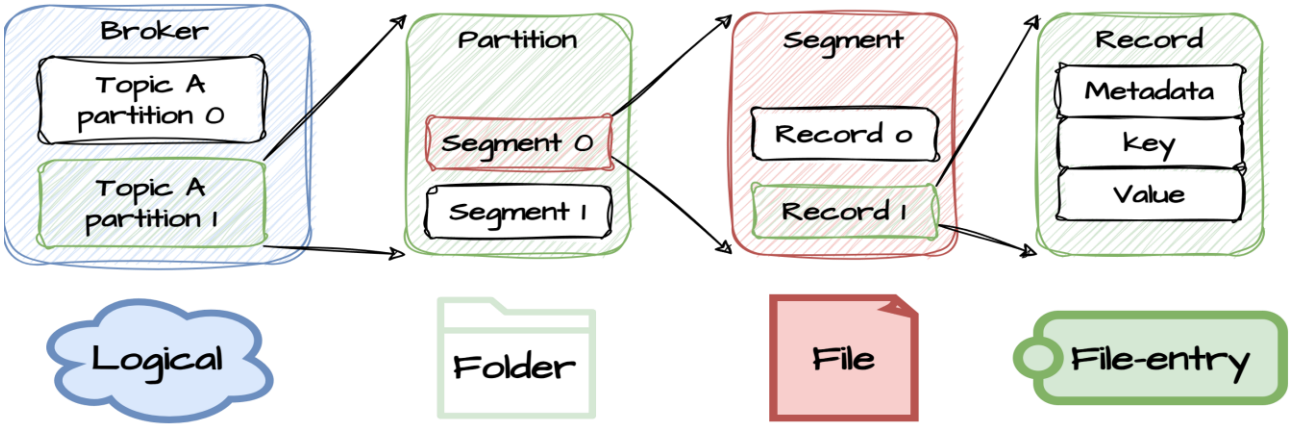
- Partitions enable parallel processing of messages. Kafka brokers can distribute partitions across multiple servers, allowing for high throughput and scalability.
- Producers can send messages to different partitions in parallel, and consumers can read from multiple partitions concurrently.

Configurable Retention:

- Each topic can have its own retention policy, which determines how long segments within the topic are retained before they are eligible for deletion.
- Retention can be configured based on time (e.g., retain messages for 7 days) or size (e.g., retain messages up to a certain disk space usage).

Topics- pub/sub & fault tolerance

Lund&Bendsen
developing developers



Publish-Subscribe Model:

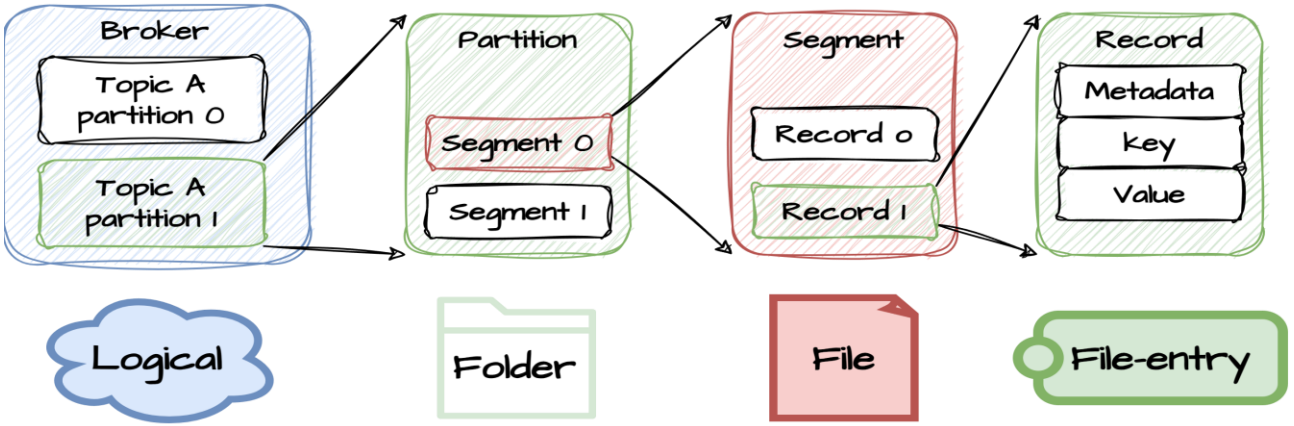
- Kafka follows a publish-subscribe messaging model. Producers publish messages to topics, and consumers subscribe to topics to receive messages.
- Multiple consumers can subscribe to the same topic, allowing for the broadcast of messages to all interested consumers.

Durability and Fault Tolerance:

- Kafka's replication mechanism ensures the durability of messages. Each partition has multiple replicas distributed across different brokers, providing fault tolerance.
- Even if some brokers or partitions become unavailable, messages can still be retrieved from the available replicas.

Topic - order & durability

Lund&Bendsen
developing developers



Message Ordering:

- Messages within a partition are strictly ordered based on their offsets. Kafka guarantees that messages are stored and delivered to consumers in the order they were produced within each partition.
- While ordering is guaranteed within a partition, there is no global ordering across partitions.

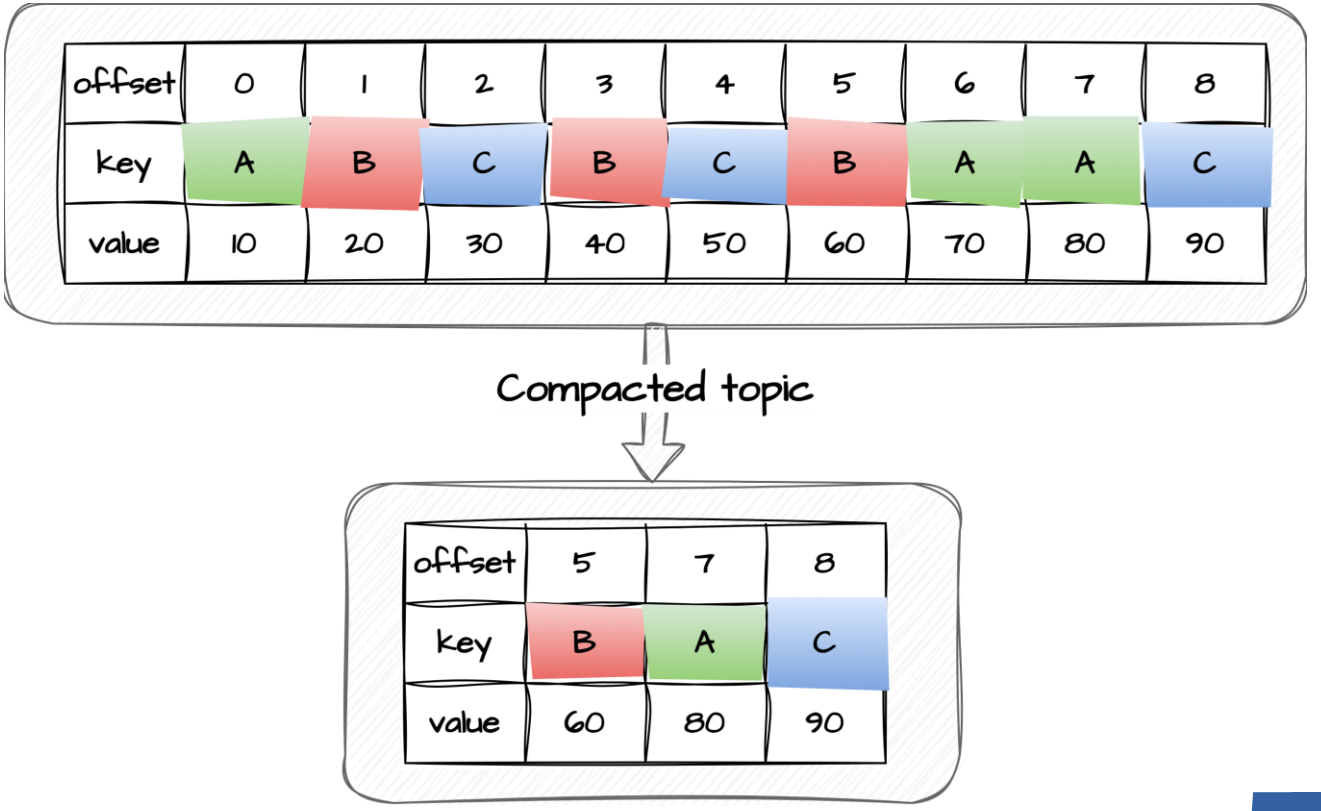
Durable Storage:

- Kafka stores messages within topics in a durable and fault-tolerant manner. Messages are written to log segments, and log segments are replicated across Kafka brokers.

Compacted topic

Lund&Bendsen
developing developers

Compacted topics only hold the latest instance of a given key – much like a regular database table. This can be efficient in some cases when you are not interested in the entire history of events pr key, but just lookeing for the latest update. Compacted topics should have few keys as they are quite expensive to maintain. When creating a topic you can also specify the `cleanup.policy` property to be `compact`



Lund&Bendsen
developing developers

Starting again at 12:30

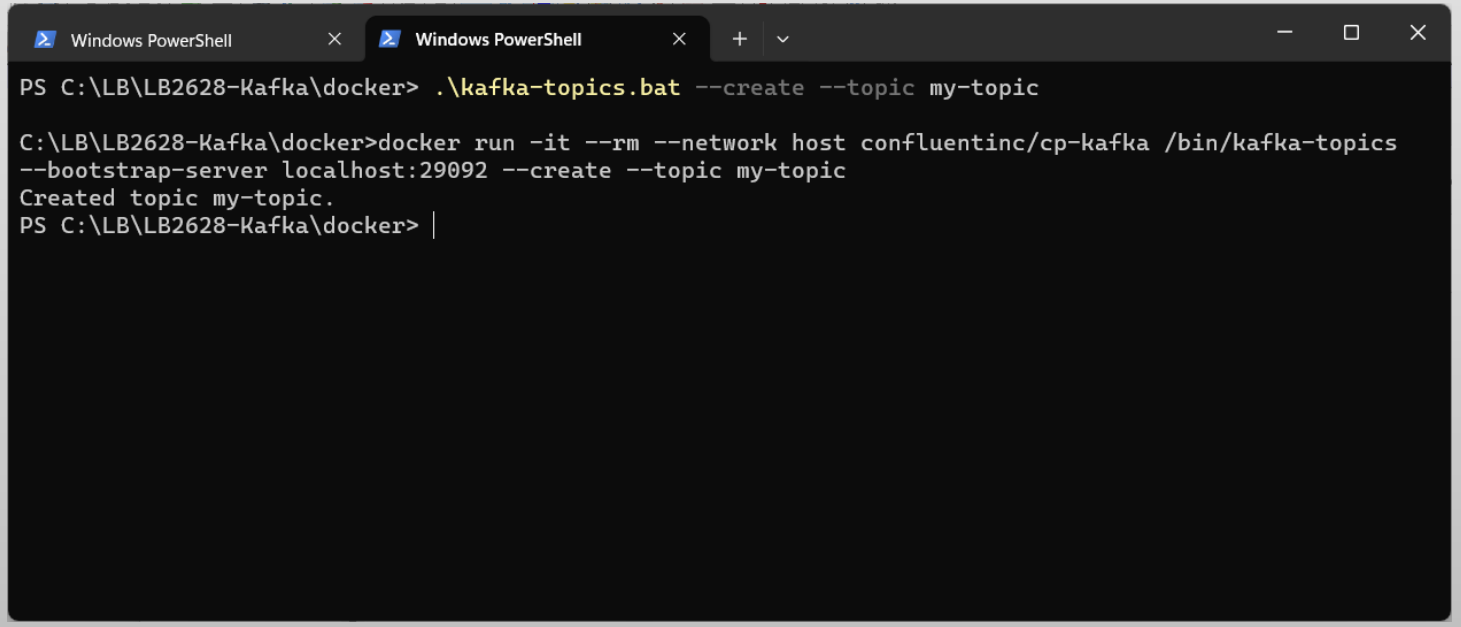
Please raise your hand when you are back!

Lund&Bendsen
developing developers

4.1

Create Kafka topic

- Open powershell and navigate to <project-folder>/docker
- Type **.\kafka-topics.bat --topic my-topic --create**
- A topic is now created. You should see a confirmation like the below:

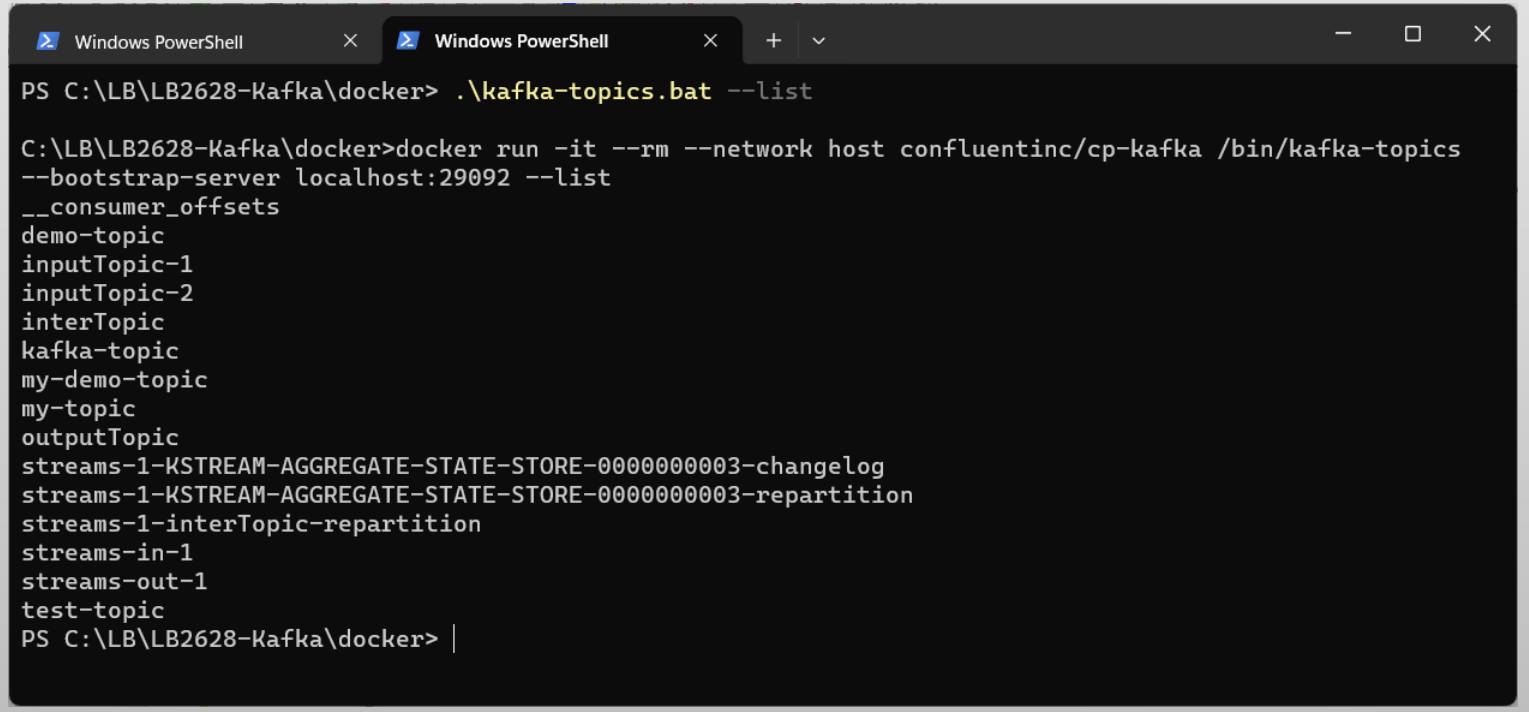


Lund&Bendsen
developing developers

4.2

List Kafka topics

- Open powershell and navigate to <project-folder>/docker
- Type **.\kafka-topics.bat --list**
- You should now be able to see a list of topics in the cluster



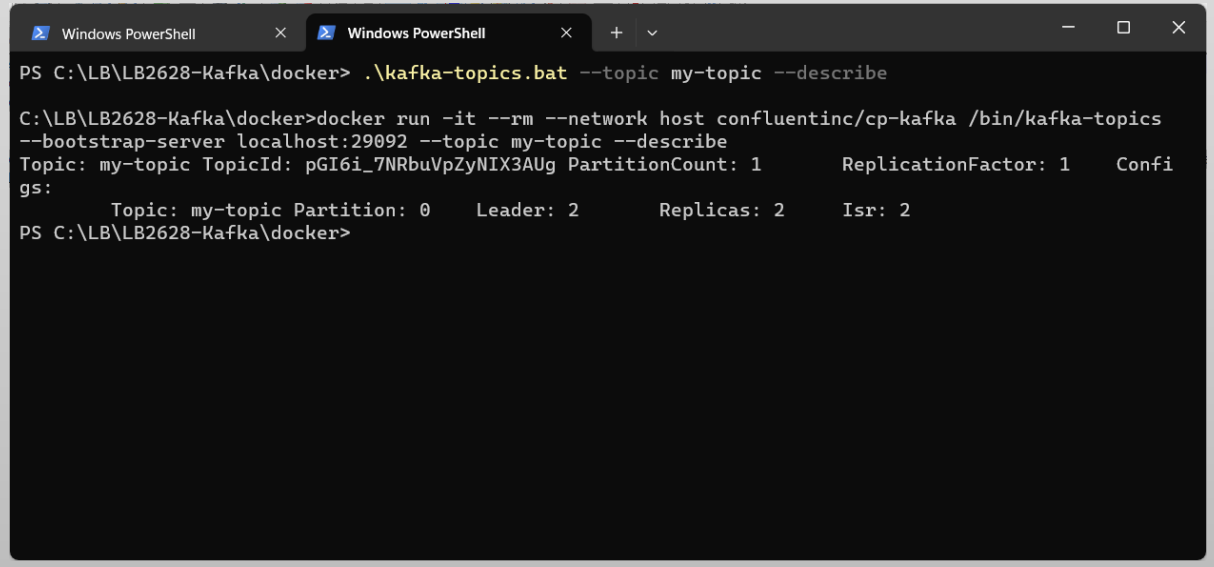
Exercise

Lund&Bendsen
developing developers

4.3

Describe Kafka topics

- Open powershell and navigate to <project-folder>/docker
- Type **.\kafka-topics.bat --topic my-topic --describe**
- You should be able to see number of partitions, the replication factor.
- For each partition you can see the leader broker, the replica brokers and the ISR brokers.
- Note: `default.replication.factor` and `num.partitions` control how many partitions/replicas are created if nothing else is specified



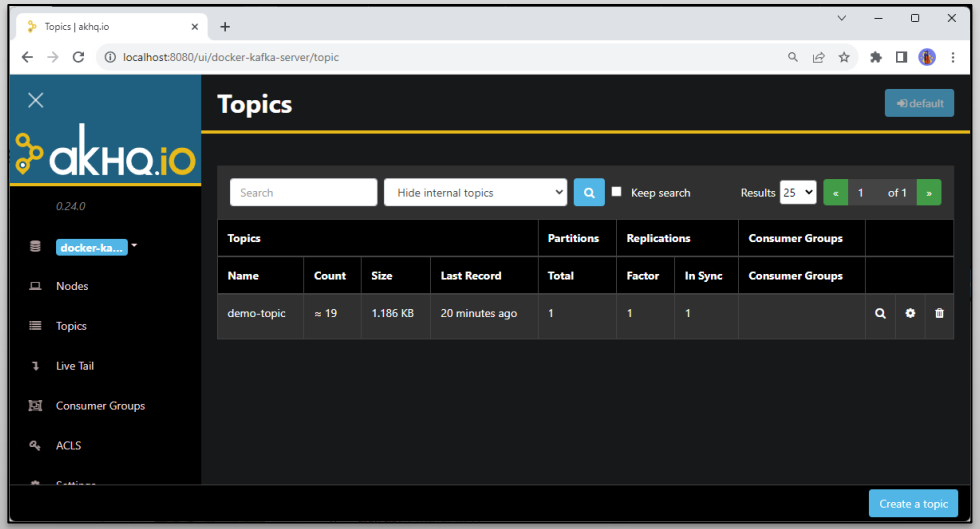
Exercise

Lund&Bendsen
developing developers

4.4

Open akhq.io

- Open a browser and type **localhost:8080**
- This will open a web-based kafka management UI.
- You should see something like below
- Find the topics you just created and check partition and replica settings
- Find the nodes (brokers) running and check the details



Exercise

Lund&Bendsen
developing developers

Delete Kafka topics

- Open powershell and navigate to <project-folder>/docker
- Type **.\kafka-topics.bat --topic my-topic --delete**
- The topic is now deleted. List the topics to confirm

Create Kafka topics through akhq.io

- Create the my-topic again from akhq.io. Notice the different settings that can be applied when creating a topic

Broker - topic configuration

Lund&Bendsen
developing developers

auto.create.topics.enable
Enable automatic creation of topics on the server. If this property is set to true, then attempts to produce, consume, or fetch metadata for a nonexistent topic automatically create the topic with the default replication factor and number of partitions. The default is enabled.

default.replication.factor
Specifies default replication factors for automatically created topics. For high availability production systems, you should set this value to at least 3.

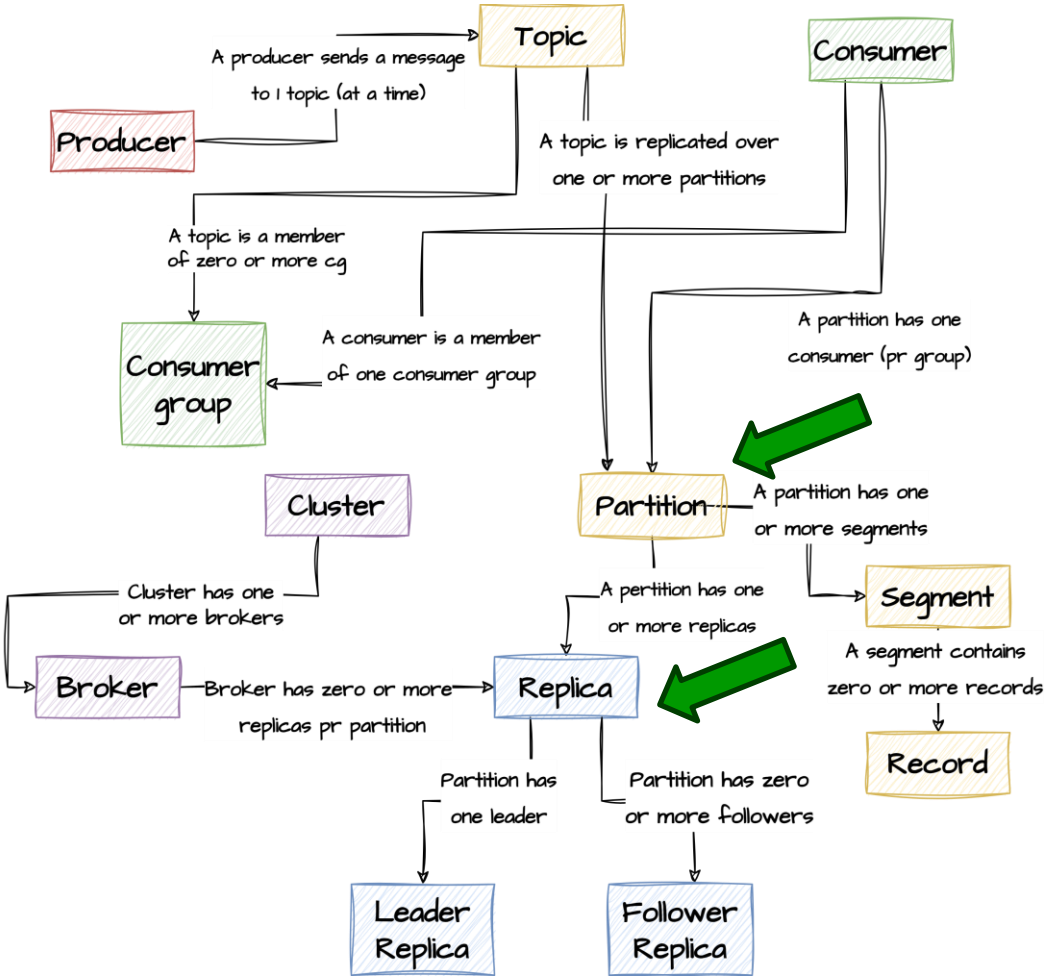
num.partitions
Specifies the default number of log partitions per topic, for automatically created topics. The default value is 1. Change this setting based on the requirements related to your topic and partition design.

delete.topic.enable
Allows users to delete a topic from Kafka using the admin tool. Deleting a topic through the admin tool will have no effect if this setting is turned off.

Partitions & Replicas

Lund&Bendsen
developing developers

- Topics
- Partitions
- Log segments
- Optimizations
- Misc



Partitions

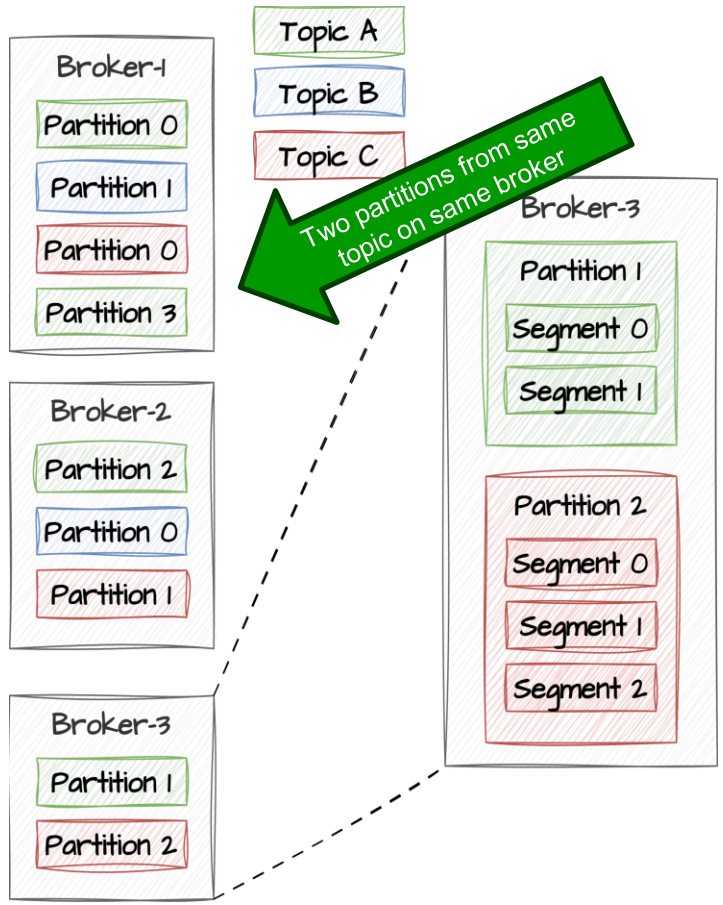
Lund&Bendsen
developing developers

Division of Topics:

- Kafka topics are divided into one or more partitions. Each partition is an independent, ordered sequence of messages.
- The division of topics into partitions allows Kafka to distribute the workload and provide parallelism and scalability.

Ordered Sequence of Messages (Per Partition):

- Messages within a Kafka partition are strictly ordered based on their offsets. Each message within a partition is assigned a unique offset, indicating its position in the sequence.
- Kafka guarantees that messages are stored and delivered to consumers in the order they were produced within each partition.



Partitions – parallelism & scalability

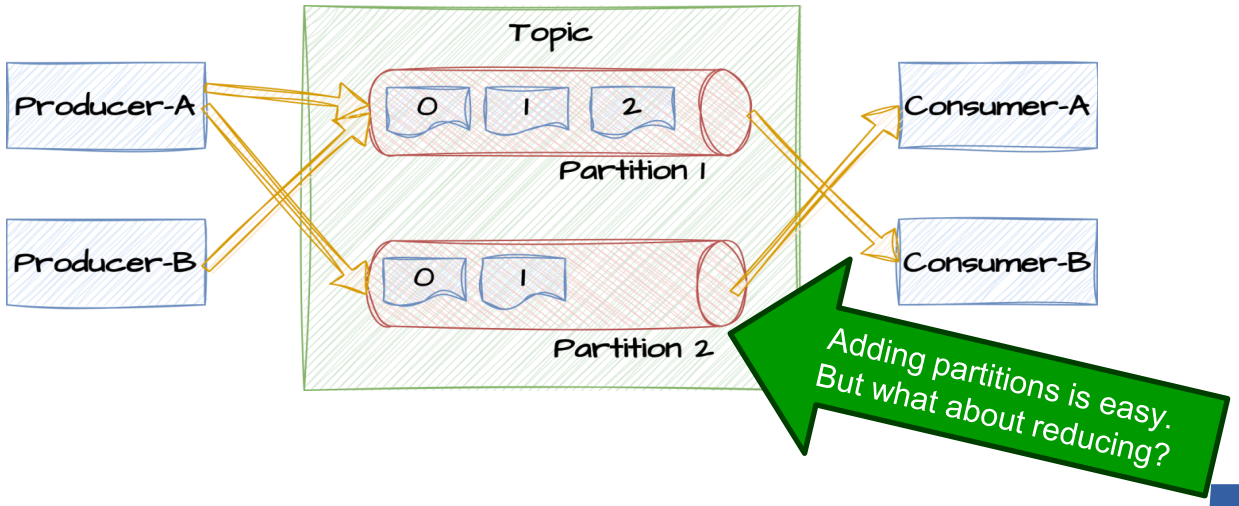
Lund&Bendsen
developing developers

Parallel Processing:

- Because partitions are distributed on different brokers, this allows for parallel processing of messages. Consumers can read from different partitions of the same topic concurrently, allowing for high throughput and efficient message handling.
- Producers can also send messages to different partitions in parallel, distributing the data load.

Scalability:

- Kafka brokers can host multiple partitions of multiple topics. This horizontal scaling capability allows Kafka to handle large volumes of data and high throughput.
- As data volume and traffic grow, additional partitions can be added to topics to distribute the load evenly.



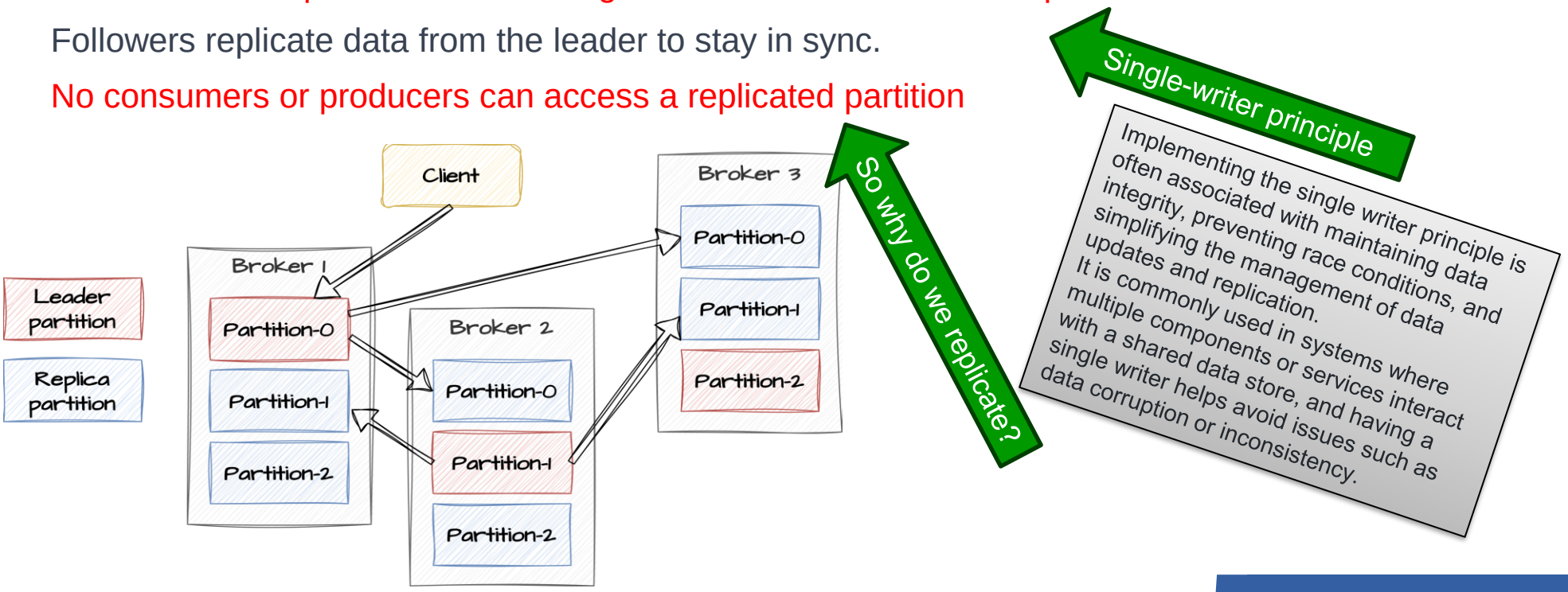
Replication of partitions

Lund&Bendsen
developing developers

Partition replication is a core feature of Kafka that provides durability, fault tolerance, and high availability. By maintaining multiple copies of data across Kafka brokers, it ensures that data is not lost in case of hardware failures or network issues

Leader and Follower Replicas:

- Each partition within a Kafka topic has one leader replica and zero or more follower replicas.
- The leader is responsible for handling all reads and writes for that partition.
- Followers replicate data from the leader to stay in sync.
- No consumers or producers can access a replicated partition

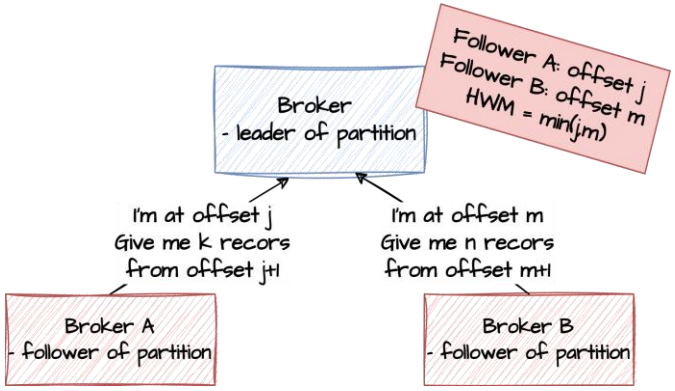


Replication - ISR

Lund&Bendsen
developing developers

An In-Sync Replica (ISR) is a replica of a Kafka partition that is (almost) caught up with the partition's leader replica. The leader is always an ISR. Due to network latency followers cannot always be fully caught up.

How do we decide when not ISR anymore?

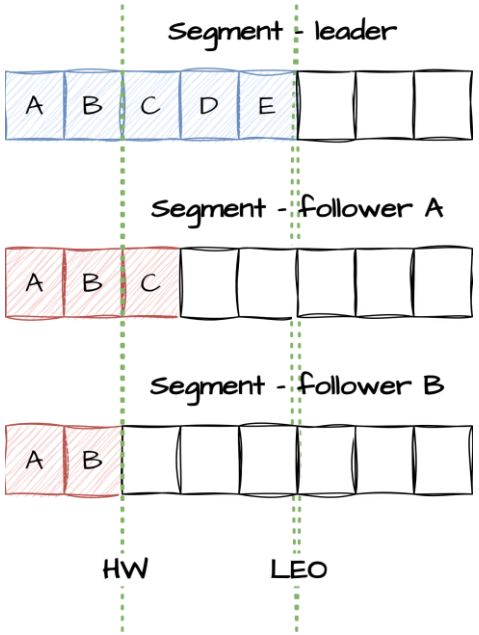


High Watermark (HWM):

- The high watermark is a marker that indicates the highest offset (position) of a message that has been successfully replicated to all in-sync replicas (ISRs) of a partition.
- The high watermark represents the point up to which data is guaranteed to be available and durable.
- In case of failure the new elected leader will resume from HWM. Any entries above HWM will be discarded

Log End Offset (LEO):

- The latest offset of messages on the leader partition.
- Consumer only reads up to the high watermark.



Consistency - Why?

ISR configuration

Lund&Bendsen
developing developers

min.insync.replicas

What can you do to handle NotEnoughReplicasExceptions?

When a producer sets acks to "all", min.insync.replicas specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an exception.

When used together, min.insync.replicas and producer acks allow you to enforce stronger durability guarantees.

You should set min.insync.replicas to 2 for replication factor equal to 3.

replica.lag.time.max.ms

If a follower hasn’t sent any fetch requests or hasn’t consumed up to the leaders log end offset for at least this time, the leader will remove the follower from ISR

unclean.leader.election.enabled

Indicates whether to enable replicas not in the ISR set to be elected as leader as a last resort, even though doing so may result in data loss

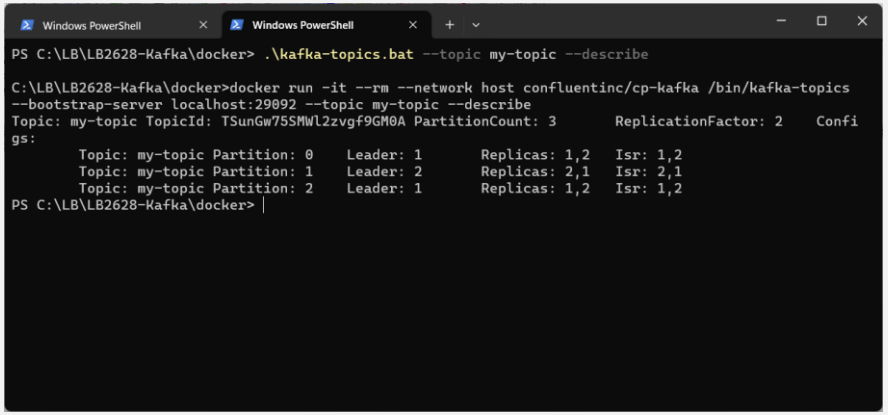
Exercise

Lund&Bendsen
developing developers

4.6

Create Kafka topics with multiple partitions

- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-topics.bat --topic my-topic --create --partitions 3 --replication-factor 2**
- A topic is now created with the partition configuration specified in the command line.
- **Type .\kafka-topics.bat --topic test-topic –describe**
- You can see that we now have 3 partitions for out test-topic.
- Verify the same thing in akhq.io



Replication – Leader election

Lund&Bendsen
developing developers

Leader Election:

- When a new partition is created or when a leader replica becomes unavailable (e.g., due to a broker failure), Kafka initiates a leader election process.
- During the leader election, one of the in-sync replicas (ISRs) is selected as the new leader. ISRs are a subset of replicas that are fully caught up with the leader.

What if there are no ISR?

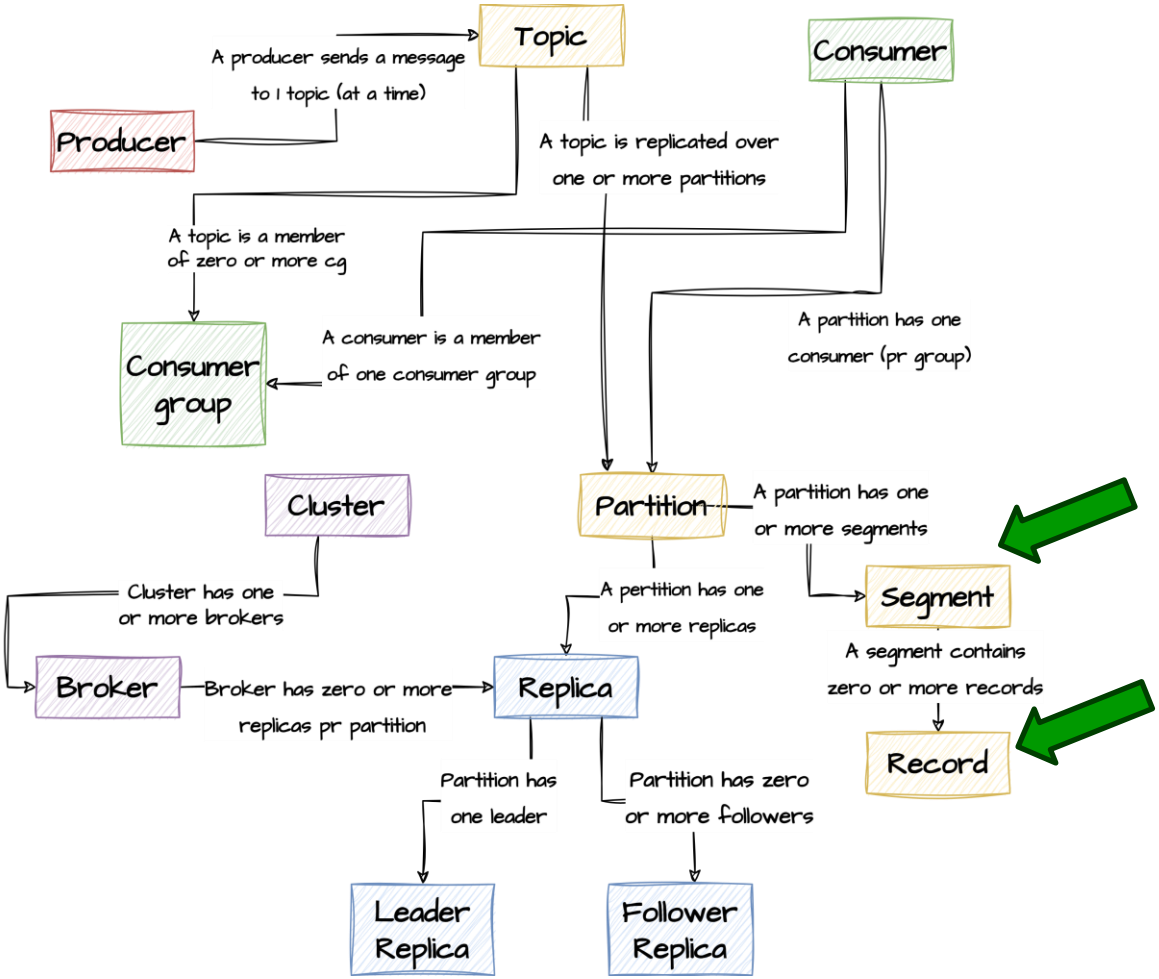
Data Replication from Leader to Followers:

- Once a new leader is elected, it begins to serve client requests. Simultaneously, it replicates messages to its follower replicas.
- Data replication is done asynchronously, meaning the leader does not wait for acknowledgments from followers before responding to produce requests.
- Followers request data from the leader and catch up by fetching and applying the missing messages.

Log segments

Lund&Bendsen
developing developers

- Topics
- Partitions
- Log segments
- Optimizations
- Misc



Log segments

Lund&Bendsen
developing developers

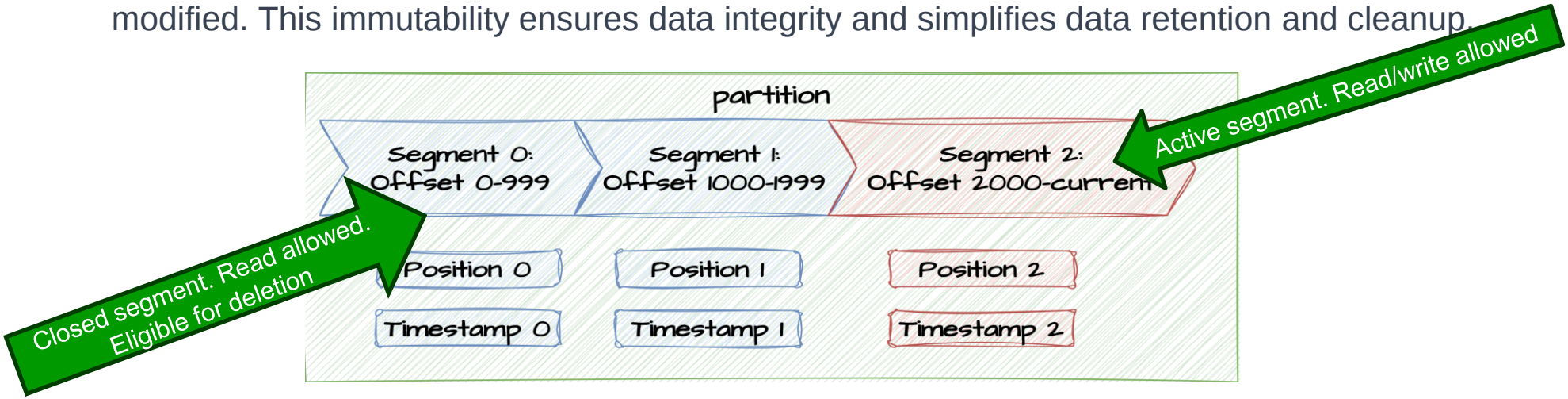
In Apache Kafka, log segments are the fundamental storage units used to store messages (events) within Kafka topics. Log segment data files have a specific structure. Index files are maintained to allow for efficient lookups in the data file based on offset and timestamp.

Segment Files:

- A log segment is represented as an append-only file on the disk. Each log segment contains a sequence of messages.
- Kafka stores these segment files in the directory specified in the broker's configuration.

Append-Only Nature:

- Log segments are append-only, meaning that once data is written to a segment, it is never modified. This immutability ensures data integrity and simplifies data retention and cleanup.



Log segments

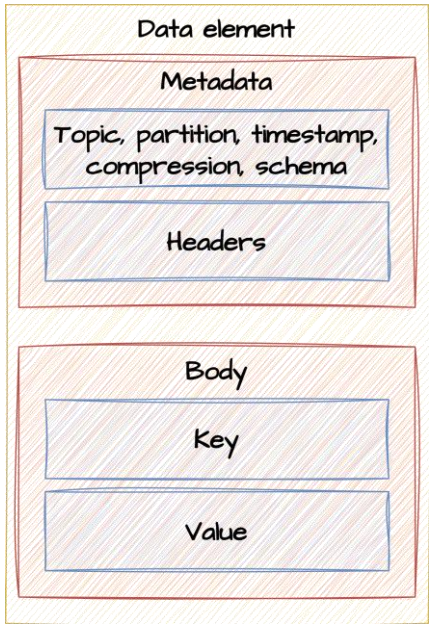
Lund&Bendsen
developing developers

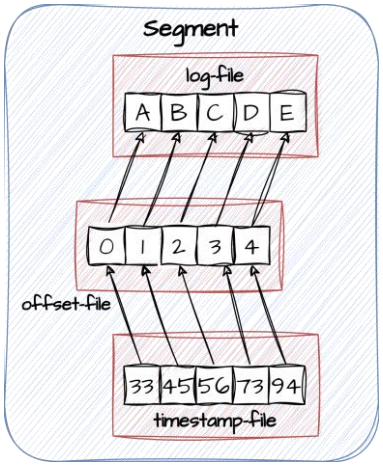
Offset Range:

- Log segments are associated with an offset range, which defines the range of message offsets contained within the segment.
- For example, a segment with an offset range of 0 to 123 contains messages with offsets from 0 to 123.

Index Files:

- To facilitate efficient message retrieval, Kafka maintains index and timestamp files alongside log segments. These index files allow Kafka to quickly look up messages based on their offsets.
- Index files are memory-mapped for fast access.





Log segments

Lund&Bendsen
developing developers

Time-Based and Size-Based Segment Rolling:

- Log segments are rolled (closed) based on two primary criteria: time and size.
- Time-Based Rolling: Log segments may be rolled periodically, ensuring that older data is eventually archived and deleted based on a retention policy.
- Size-Based Rolling: Log segments may be rolled when they reach a maximum size, as configured in the broker settings.

Maximum Size:

- Each log segment has a configurable maximum size, typically specified in broker configuration settings. Once a segment reaches this size, it is closed, and a new segment is created for incoming messages.

Log configuration

Lund&Bendsen
developing developers

log.roll.hours
The maximum time, in hours, before a new log segment is rolled out. The default value is 168 hours (seven days).
This setting controls the period of time after which Kafka will force the log to roll, even if the segment file is not full. This ensures that the retention process is able to delete or compact old data.

log.retention.hours
The number of hours to keep a log file before deleting it. The default value is 168 hours (seven days).
When setting this value, take into account your disk space and how long you would like messages to be available.
An active consumer can read quickly and deliver messages to their destination.
The higher the retention setting, the longer the data will be preserved. Higher settings generate larger log files, so increasing this setting might reduce your overall storage capacity.

Log configuration

Lund&Bendsen
developing developers

log.dirs
A comma-separated list of directories in which log data is kept. If you have multiple disks, list all directories under each disk.
Review the following setting in the Advanced kafka-broker category, and modify as needed:

log.retention.bytes
The amount of data to retain in the log for each topic partition. By default, log size is unlimited. Note that this is the limit for each partition, so multiply this value by the number of partitions to calculate the total data retained for the topic.
If *log.retention.hours* and *log.retention.bytes* are both set, Kafka deletes a segment when either limit is exceeded.

log.segment.bytes
The log for a topic partition is stored as a directory of segment files. This setting controls the maximum size of a segment file before a new segment is rolled over in the log. The default is 1 GB.

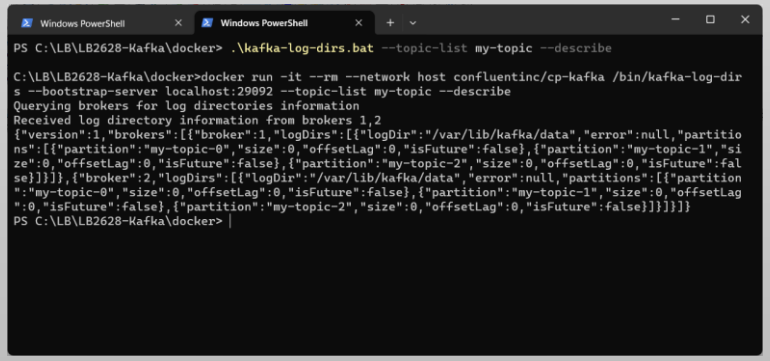
Exercise

Lund&Bendsen
developing developers

4.7

List log information

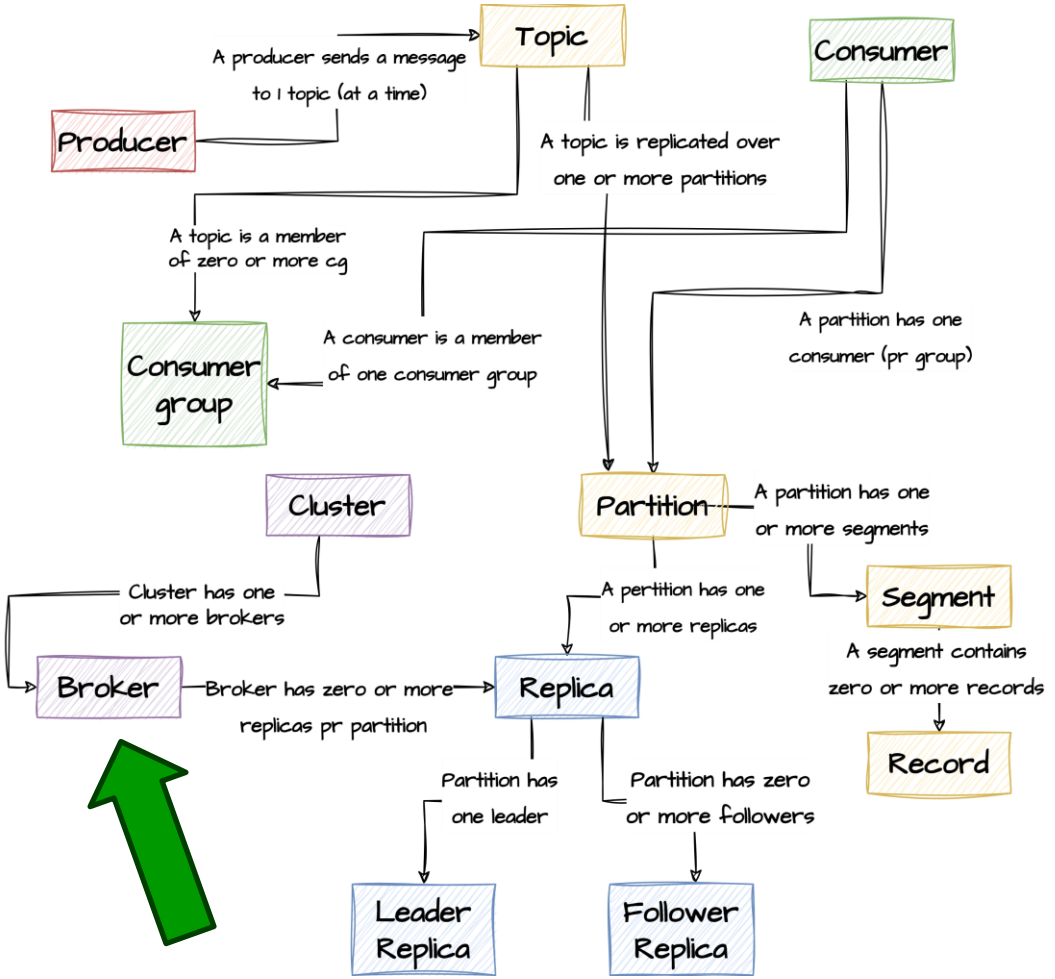
- Open powershell and navigate to <project-folder>/docker
- Type **.\kafka-log-dirs.bat --topic-list my-topic --describe**
- You should now be able to see a description of how partitions for the topic have been created on the two brokers, like the below dump:
- Type **docker exec kafka1 ls /var/lib/kafka/data/** to see all partition folders in the broker. Choose one of them and navigate into it
- **NB:** you can also use Docker Desktop to navigate the file-structure of the kafka container



Agenda

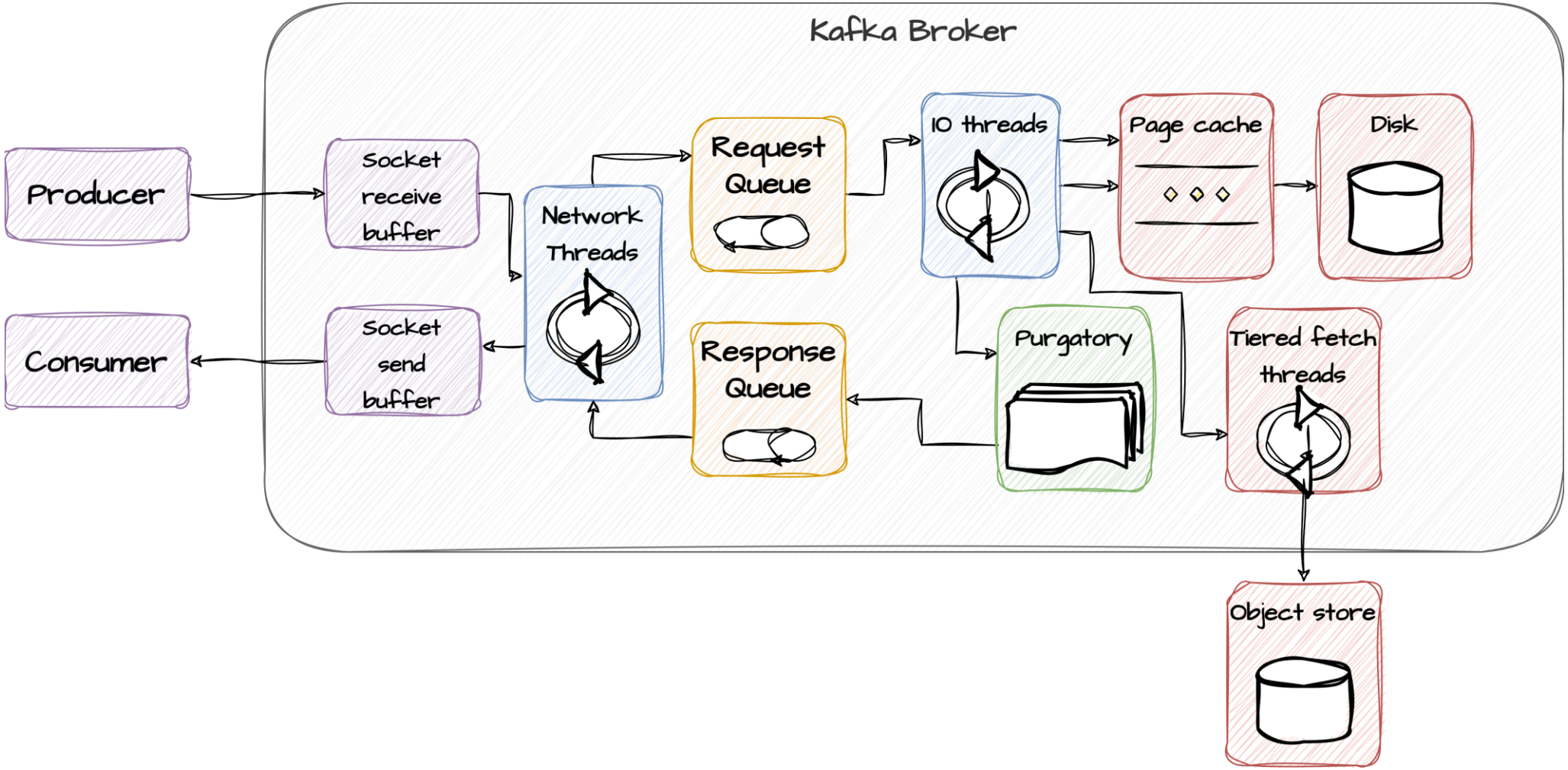
Lund&Bendsen
developing developers

- Topics
- Partitions
- Log segments
- Optimizations
- Misc



Broker internals

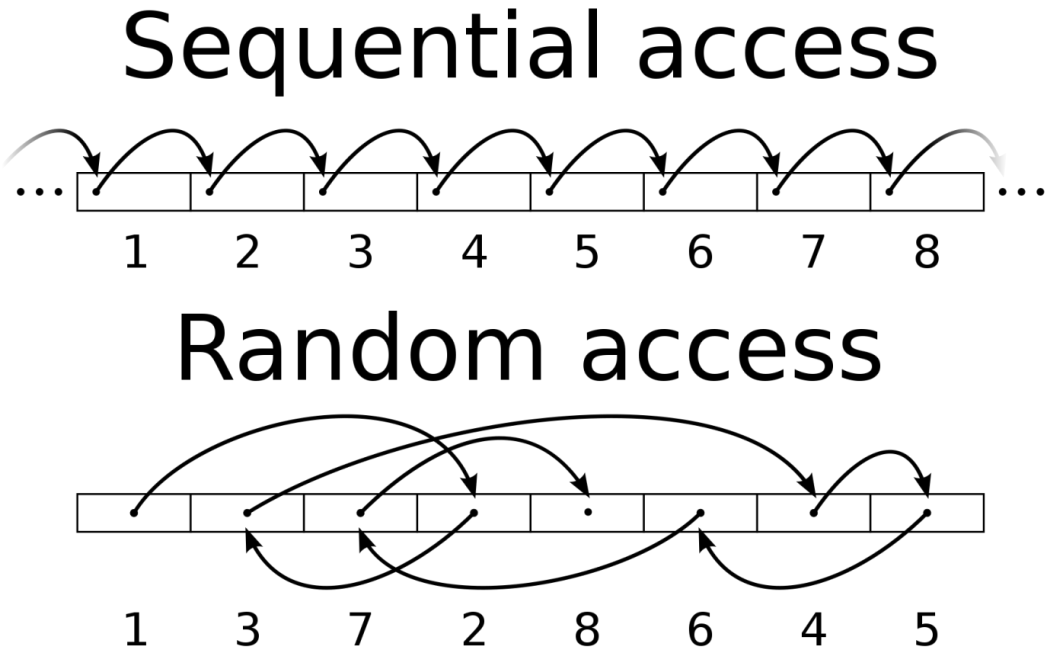
Lund&Bendsen
developing developers



Optimization – Sequential I/O

Lund&Bendsen
developing developers

On disk data is structured in blocks each with its own unique address. When reading and writing to disk the computer uses the block address to locate the data. The disk head is optimized to move in straight lines and as the performance benefits are quite substantial if I/O operations can be done on contiguous blocks of data. This is referred to as sequential access as opposed to random access where the disk head moved back and forth adds a lot of overhead to the I/O operations.



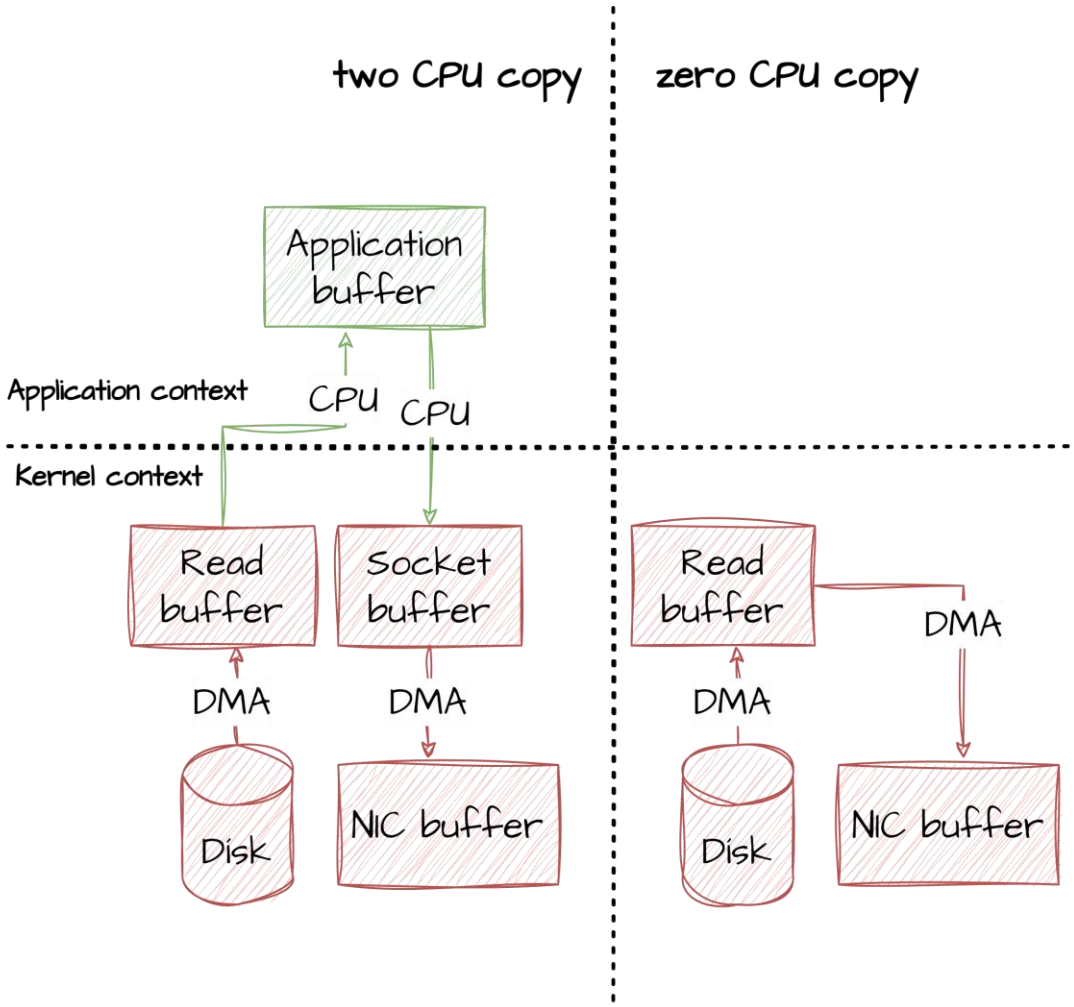
Kafka uses an append-only write strategy which means that all read and write operations are done sequentially

Optimization - Zero copy

Lund&Bendsen
developing developers

Zero-copy is used in computer operations to avoid involving the CPU in data-copying tasks between memory areas. It also eliminates unnecessary data copies, saving CPU cycles and memory bandwidth. This approach is particularly useful for tasks that involve transferring large amounts of data, such as high-speed file transmission over a network.

Kafka uses zero copy extensively in read operations when tranferrring data from segments to network interfaces

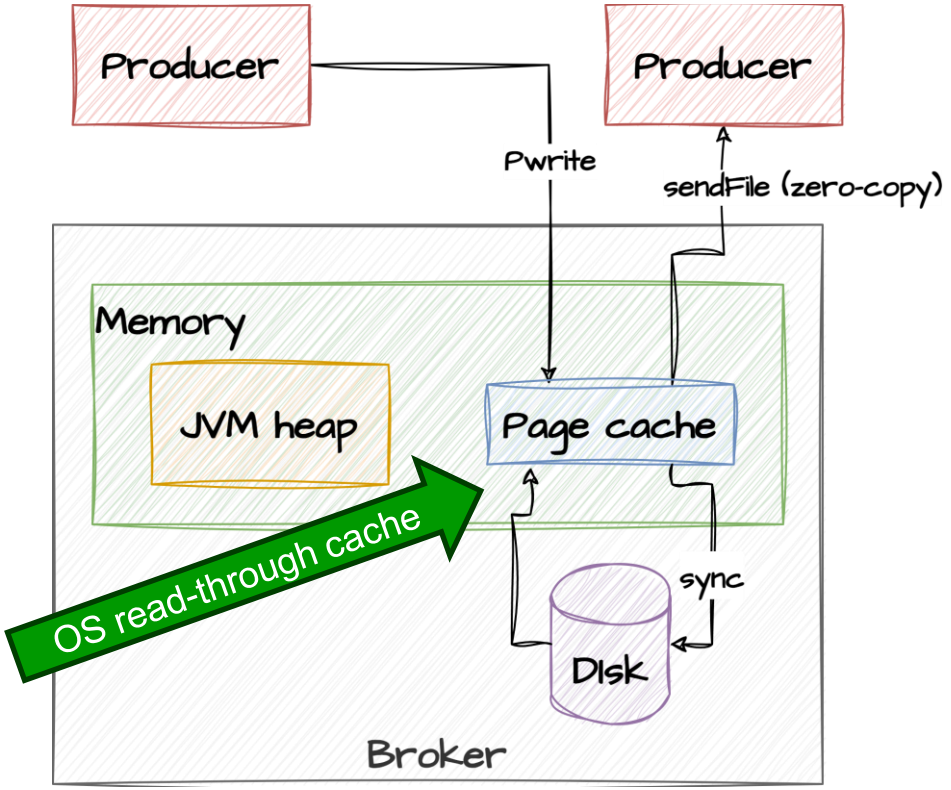


Optimization - Page cache

Lund&Bendsen
developing developers

A page cache is a component of an operating system's memory management system that stores copies of frequently accessed data from storage devices, such as hard drives or SSDs, in the system's main memory (RAM). The primary purpose of a page cache is to speed up read operations by reducing the need to access data directly from slower storage media.

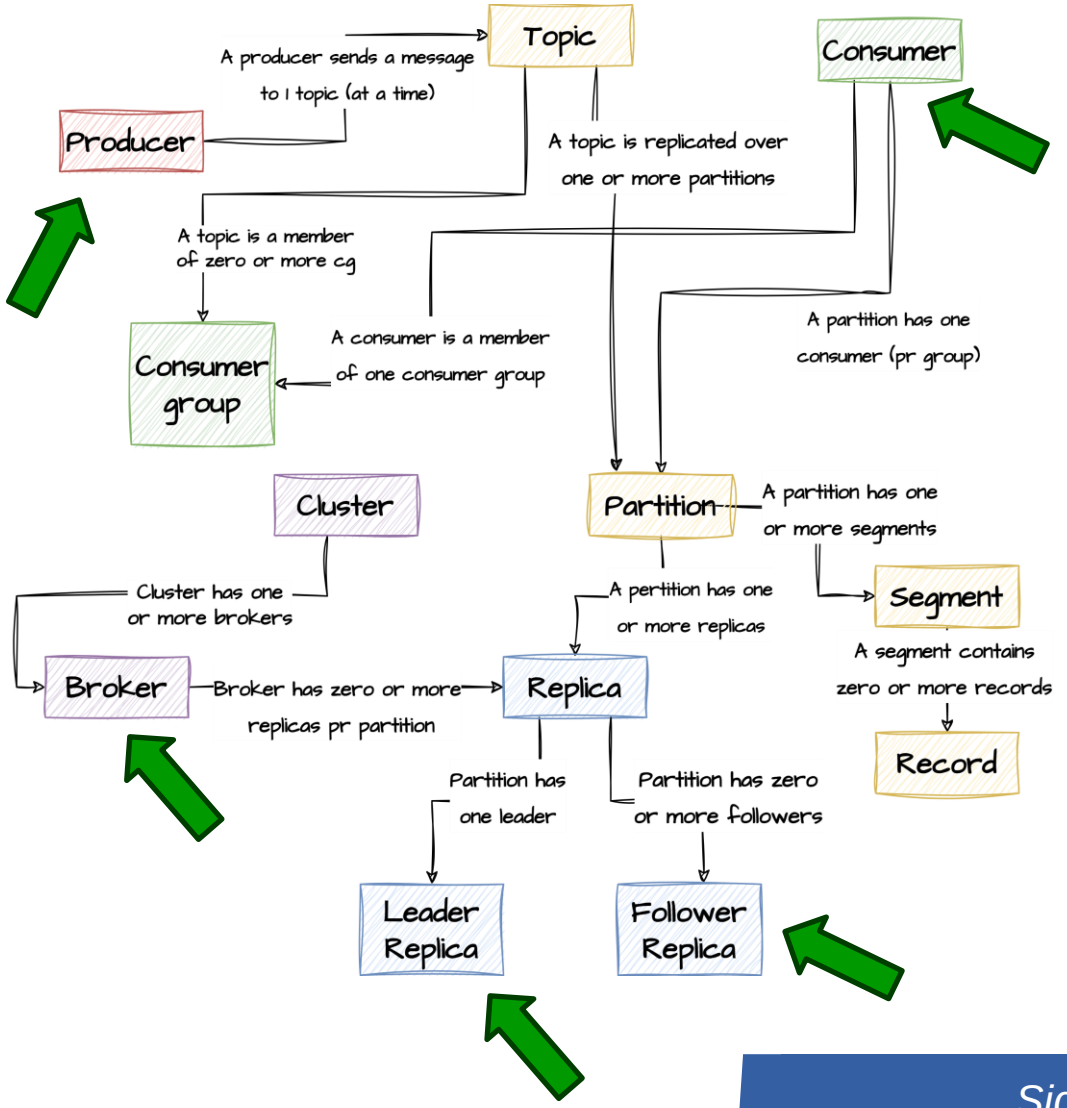
Kafka uses the page cache to keep live segments in memory for fast read access for consumers. Fast retrieval of data is guaranteed if consumers request data from memory mapped segments which is usually the case when consumers are not lagging behind



Agenda

Lund&Bendsen
developing developers

- Topics
- Partitions
- Log segments
- Optimizations
- Misc

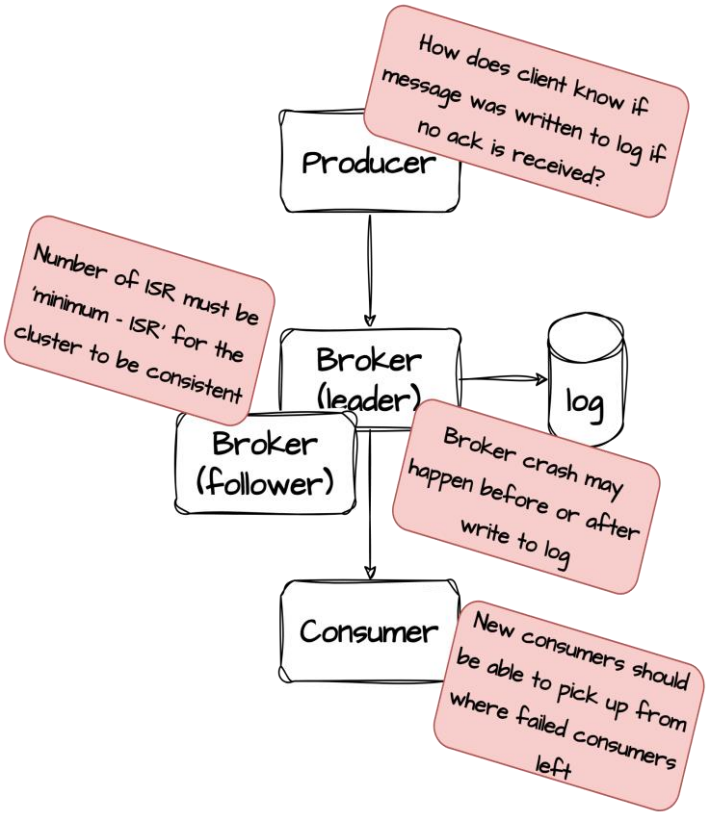


Failures

Lund&Bendsen
developing developers

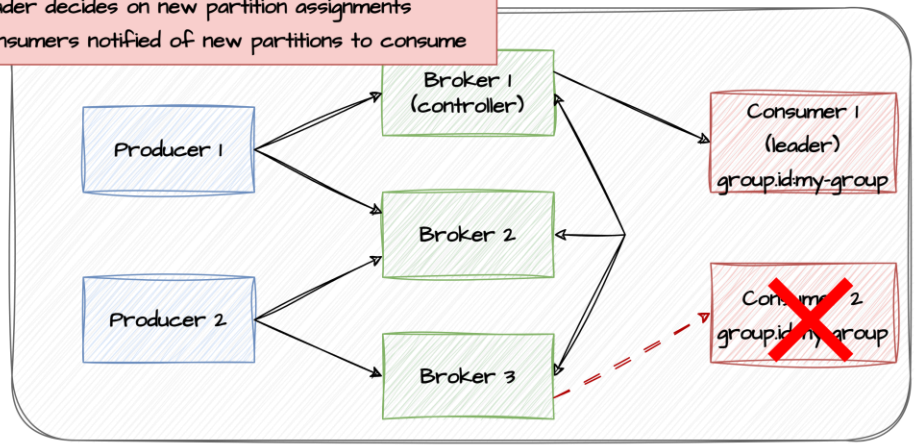
Kafka can tolerate *n-1* broker failures, meaning that a partition is available if there is at least one broker available. Kafka's replication protocol guarantees that once a message has been written successfully to the leader replica, it will be replicated to all available replicas.

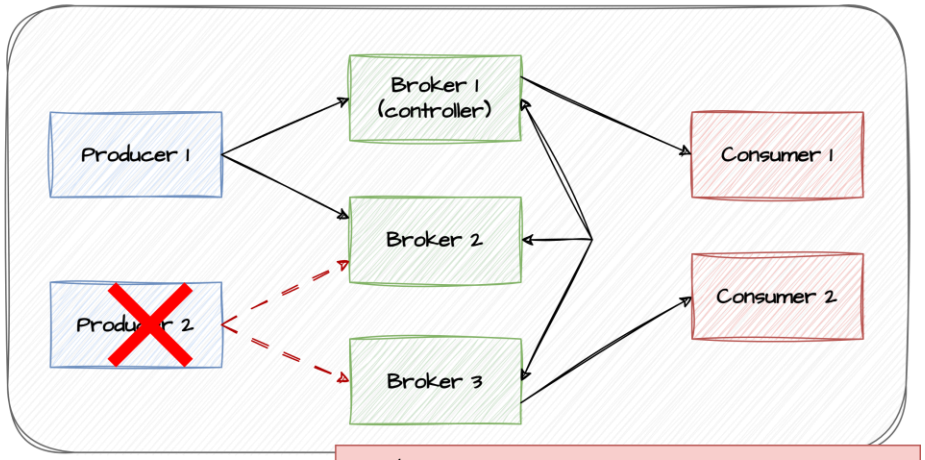
Durability in Kafka depends on the producer receiving an ack from the broker. Failure to receive that ack does not necessarily mean that the request itself failed. The broker can crash after writing a message but before it sends an ack back to the producer. It can also crash before even writing the message to the topic. Since there is no way for the producer to know the nature of the failure, it is forced to assume that the message was not written successfully and to retry it. In some cases, this will cause the same message to be duplicated in the Kafka partition log, causing a consumer to receive it more than once. (Can be solved with idempotent producers)



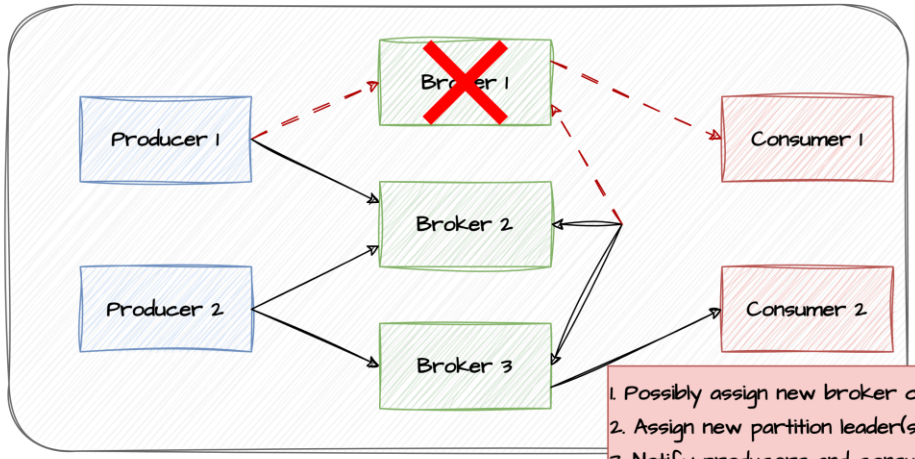
Failures

1. Consumer crashes and heart-beat stops
2. Controller triggers rebalance of consumer group
3. Leader decides on new partition assignments
4. Consumers notified of new partitions to consume





1. Producer crashes
2. If new producer needs to pick up from somewhere, this needs to be a custom implementation



1. Possibly assign new broker controller
2. Assign new partition leader(s) from followers
3. Notify producers and consumer about new partition leaders
4. Rebalance consumers groups (leader and controller involved)
5. Notify consumers of new partitions to consume

Transactions

Lund&Bendsen
developing developers

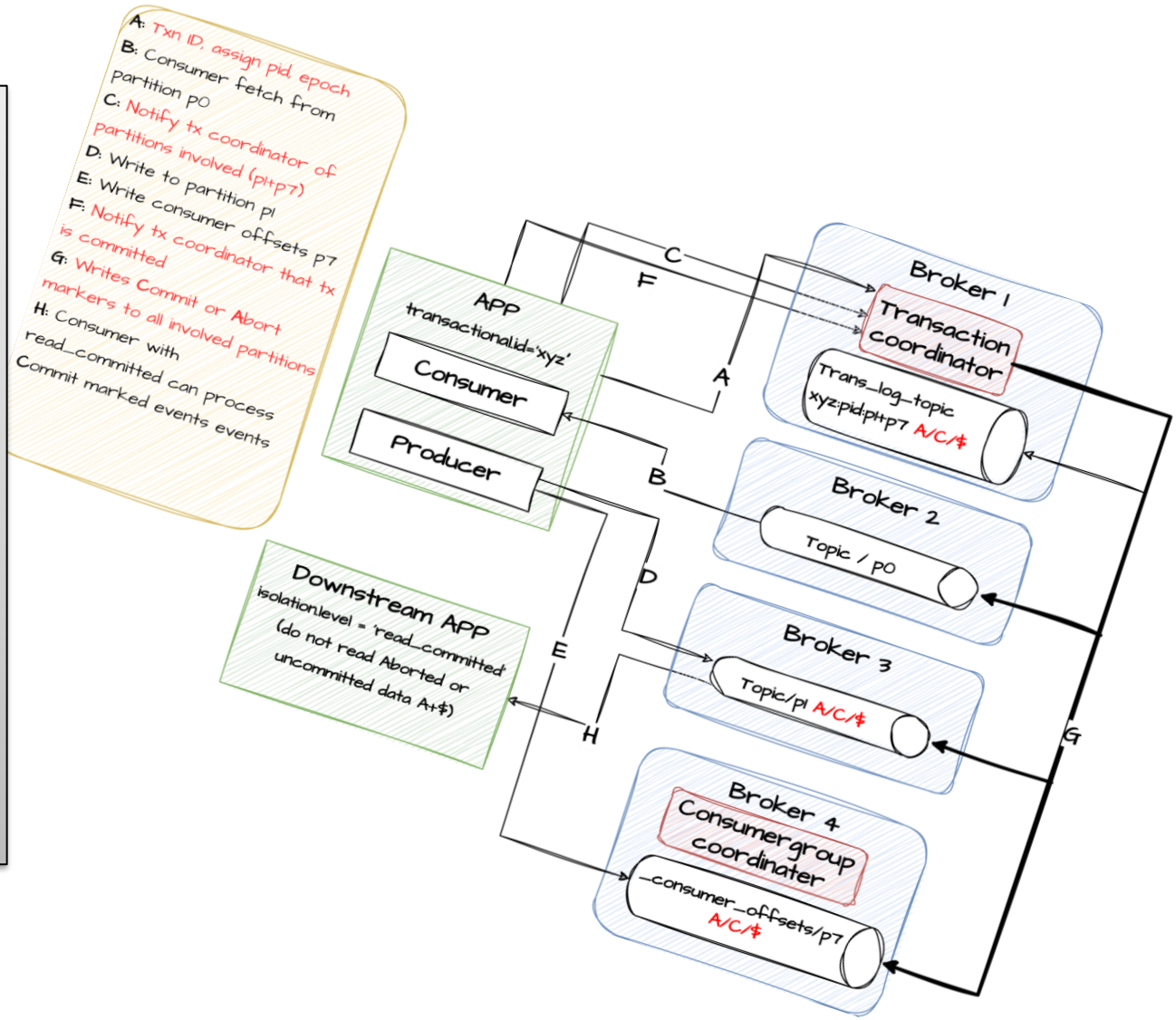
```
KafkaProducer producer = createKafkaProducer(
  “bootstrap.servers”, “localhost:9092”,
  “transactional.id”, “my-transactional-id”);

producer.initTransactions();

KafkaConsumer consumer = createKafkaConsumer(
  “bootstrap.servers”, “localhost:9092”,
  “group.id”, “my-group-id”,
  "isolation.level", "read_committed");

consumer.subscribe(singleton(“inputTopic”));

while (true) {
  ConsumerRecords records =
              consumer.poll(Long.MAX_VALUE);
  producer.beginTransaction();
  for (ConsumerRecord record : records) {
    producer.send(producerRecord(“outputTopic”,
              record));
   producer
.sendOffsetsToTransaction(currentOffsets(consumer)
, group);
  producer.commitTransaction();
}
```



Security

Lund&Bendsen
developing developers

Out of the box Kafka runs with very little security turned on. But if in most cases its not acceptable that any client can connect to the cluster and start sending / consuming messages. Kafka uses that flexibility of the java security framework, which means that new security providers can be plugged into the system.

Authentication:

Kafka supports various authentication mechanisms, including:

- SSL/TLS: Encrypts data in transit and authenticates clients and brokers. (two-ways)
- SASL (Simple Authentication and Security Layer): Supports authentication via mechanisms like PLAIN, GSSAPI, and more.
- Custom authentication: Allows for custom authentication providers to be implemented.

Authorization:

- Kafka employs an Access Control List (ACL) mechanism to restrict which clients can read from or write to specific topics.
- Fine-grained access control can be established by specifying ACLs for users, groups, or IPs.

Security example

Lund&Bendsen
developing developers

Broker:

```
# Configure SASL_SSL if SSL encryption is enabled, otherwise configure
SASL_PLAINTEXT
security.inter.broker.protocol=SASL_SSL

# With SSL encryption
listeners=SASL_SSL://kafka1:9093
advertised.listeners=SASL_SSL://localhost:9093
```

Producer / Consumer:

```
props.put("security.protocol", "SASL_SSL");
props.put("sasl.mechanism", "SCRAM-SHA-256");
props.put("sasl.jaas.config",
"org.apache.kafka.common.security.scram.ScramLoginModule required username=”kafka-
client" password=”secret";");
```

auto.leader.rebalance.enable

Enables automatic leader balancing. A background thread checks and triggers leader balancing (if needed) at regular intervals. The default is enabled.

unclean.leader.election.enable

This property allows you to specify a preference of availability or durability. This is an important setting: If availability is more important than avoiding data loss, ensure that this property is set to true. If preventing data loss is more important than availability, set this property to false.

This setting operates as follows:

If unclean.leader.election.enable is set to true (enabled), an out-of-sync replica will be elected as leader when there is no live in-sync replica (ISR). This preserves the availability of the partition, but there is a chance of data loss.

If unclean.leader.election.enable is set to false and there are no live in-sync replicas, Kafka returns an error and the partition will be unavailable.

This property is set to true by default, which favors availability.

If durability is preferable to availability, set unclean.leader.election to false.

message.max.bytes

Specifies the maximum size of message that the server can receive. It is important that this property be set with consideration for the maximum fetch size used by your consumers, or a producer could publish messages too large for consumers to consume.

Note that there are currently two versions of consumer and producer APIs. The value of *message.max.bytes* must be smaller than the *max.partition.fetch.bytes* setting in the new consumer, or smaller than the *fetch.message.max.bytes* setting in the old consumer. In addition, the value must be smaller than *replica.fetch.max.bytes*.

replica.fetch.max.bytes

Specifies the number of bytes of messages to attempt to fetch. This value must be larger than message.max.bytes.