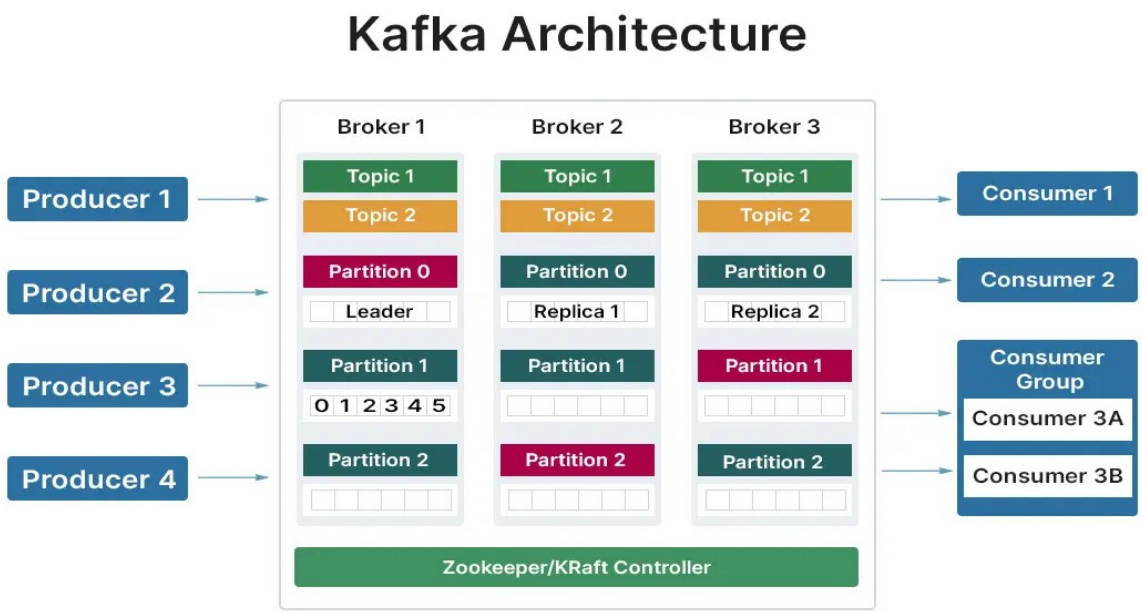
Chapter 1 - The Inner Workings of Apache Kafka

Objectives

In this chapter, participants will learn about:

- Apache Kafka's internals
  - Topics and partitions
  - Brokers
  - Consumer groups
  - Log compaction

1.1 A Kafka Cluster High-Level Interaction Diagram



1.2 Topics & Partitions

- A Kafka topic is a mechanism for message classification and is often referred to as a distributed commit log; it can also be viewed as
  - Materialized event view into streamed events at rest
- Topics are made up of partitions, which are ordered "commit logs" (physical files) where events (messages/records) are stored
  - By default, on Linux, partitions are stored in the */tmp/kafka-logs/* directory. Partition files are named <topic>-<partion_id>, e.g. orders-0, orders-1, etc. Operational topic partitions are being served by the (partition) leader broker
- For fault tolerance, each partition can be replicated to one or more brokers in the cluster
  - The replication factor (the *replication.factor* configuration parameter) is configured at the topic's creation time; the recommended value in production is 3 replicas

1.3 The Terms Event/Message/Record

- The terms *event/message/record* are, for the most part, used interchangeably to refer to a unit of data moved or persisted within Kafka
  - The term *record* is more often used to refer to a persisted message/event
- Internally, Kafka uses instances of the *ProducerRecord* class for published messages (https://tinyurl.com/2eb8r8vk)

1.4 Message Offset

- A message offset in Kafka is used as a message id
- It is a per-partition monotonically increasing integer [0,1,2,3, ...] that is unique within a partition
- An offset along with the partition id and broker node id uniquely identifies

the location of the message on a Kafka cluster

1.5 Message Retention Settings

- During topic creation, you can configure message retention (applicable to all topic's partitions) using these two options:
  - **Retention time**
    - ✔ Controlled by the *retention.ms* parameter (-1 for unlimited storage time), or
  - **Retention size**
    - ✔ Controlled by the *retention.bytes* parameter (-1 for uncapped size)
  - **Note:** Those topic-level settings can be combined together such that whichever one comes first takes effect

1.6 Deleting Messages

- Messages identified as exceeding the established retention threshold, are either
  - Deleted (the default setting), or
  - Compacted (more on compaction a bit later ...)
  - This strategy is controlled by the *cleanup.policy* parameter configured at the topic creation time
- When messages from a partition file are deleted (e.g. due to resetting the logs after the retention period), the new record's offset is set to the last recorded offset used + 1

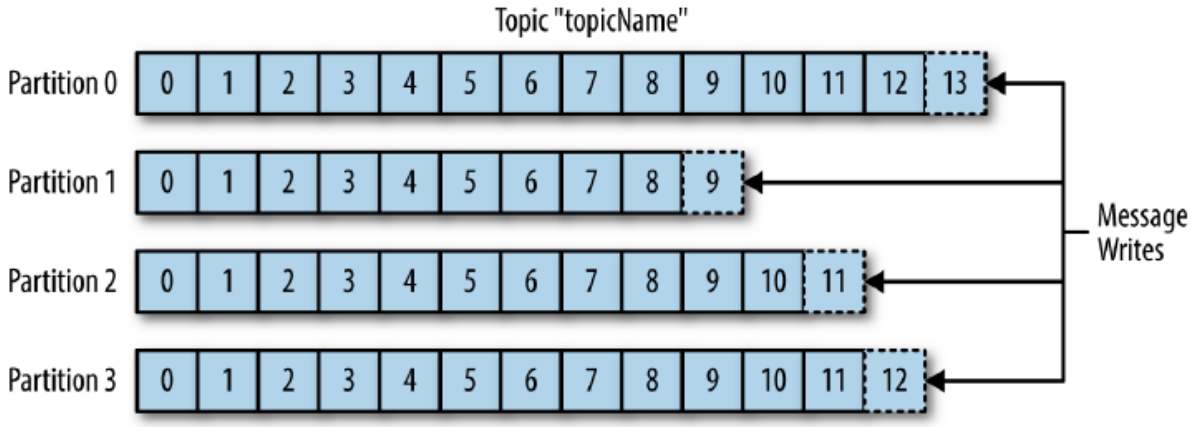
1.7 The Flush Policies

- Flushing is forcing a disk write operation (the hard *fsync* operation on Linux) on messages that could potentially be cached by the IO subsystem and lazily flushed by the system

- Not flushing messages in time may result in lost messages
- The important trade-offs to be aware of:
  - *Durability*: Unflushed data may be lost if you are not using replication
  - *Latency*: Very large flush intervals may lead to latency spikes
  - *Throughput*: The flush comes at a cost – small flush intervals will flood your disk controller
- You have an option to flush over an interval or after a number of messages (or both at the same time)
- The flush policy is controlled in the *server.properties:*
  - It can be overridden on a per-topic basis
  - *log.flush.interval.messages=10000*
  - *log.flush.interval.ms=1000*

1.8 Writing to Partitions

- Messages are written to a partition in an append-only fashion
- Producer sends the data directly to the (partition) lead broker, which takes care of most of the housekeeping tasks



Source: *Kafka: The Definitive Guide*

1.9 Batches

- To improve overall Kafka throughput, brokers write messages to partitions in batches
  - Batches are typically compressed for more efficient data transfer
  - Messages are batched by topic-partition
- Batching can be configured as either a fixed number of messages or a batching window (e.g. 128K or 50 ms)
- **Note**: Using batches results in message buffering which introduces latency between a message production and the time it becomes a persistent record in Kafka (a factor contributing to a consumer lag situation)

1.10 Batch Compression

- For further efficiency, Kafka uses batch compression
- The batch of messages is written to the log in compressed form
- Consumers decompress the batch on read
- Kafka supports GZIP, Snappy, and LZ4 compression protocols

1.11 Partitions as a Unit of Parallelism

- A partition acts as a unit of write/read parallelism
  - More partitions allow for greater parallelism for message production and consumption at the expense of more files being stored across broker nodes
    - ✔ Adds complexity in partition replicated environments that may lead to saturating network bandwidth in the cluster and cases of under-replicated partitions → data loss!

1.12 Message Ordering

- Kafka provides strict record ordering within a single partition

- The onus of ordering across topic's partitions is on developers who would need to use record timestamps for application-controlled ordering in their apps

1.13 Kafka Default Partitioner

- Kafka default partitioner, which is part of the Kafka client library, uses the following logic for determining the target partition of a message created by the producer:
  - If the producer provides a partition number in the message record, that partition is used
  - If a producer provides a key (a message meta-parameter), the partitioner will calculate the partition id based on a hash value of the key
  - When no partition number or key is present, Kafka default partitioner will pick a partition in a round-robin fashion
- Developers can create custom partitioners to better accommodate their problem domain requirements

Note:

Key hash-based partitioning logic can be illustrated by this Java code snippet:

```
import org.apache.kafka.common.utils.Utils;
...
return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
```

1.14 The Load Balancing Aspect

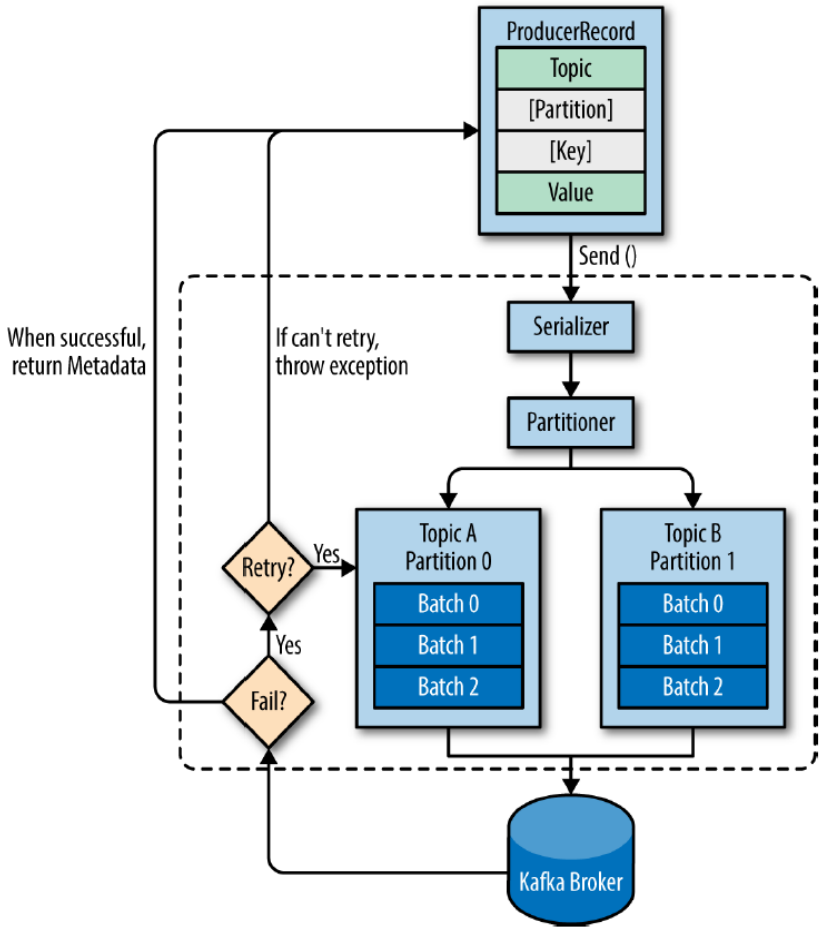
- ***Key***-less messages (the ***key*** meta-parameter is null) are written to a partition randomly (Kafka provides round-robin load-balancing) unless a key is provided as message metadata
  - That means that partitioning can also help with load balancing over brokers
- A Kafka message can have an optional ***key*** meta-parameter that makes message distribution more deterministic: messages with the same key are routed by the broker to the same partition

- This can be of great value in apps that, for example, use the event sourcing pattern

Note:

To accomplish simple load balancing, a simple approach would be for the client to just round robin requests over all brokers. Alternatively, in an environment where there are many more producers than brokers, would be to have each client choose a single partition at random and publish to that. This later strategy will result in far fewer TCP connections.

1.15 Kafka Message Production Schematics



Source: *Kafka: The Definitive Guide*

Notes:

The dashed section in the figure delineates the scope of the Kafka client library's functionality.

1.16 Reading from a Topic

- A consumer (message reader from a topic) subscribes to one or more topics and performs sequential or random reads
- To read a record, a consumer needs three pieces of information:
  - Topic name,
  - Partition id within the topic, and
  - Offset of the record within the partition (the "offset pointer")
  - For efficiency, a consumer can request a block of records
- The "fetch" request is processed by the broker leading the partition
- A consumer can go back and re-read a previously fetched record
  - The system retains the record ids (offsets) a consumer has read

1.17 Consumer Lag

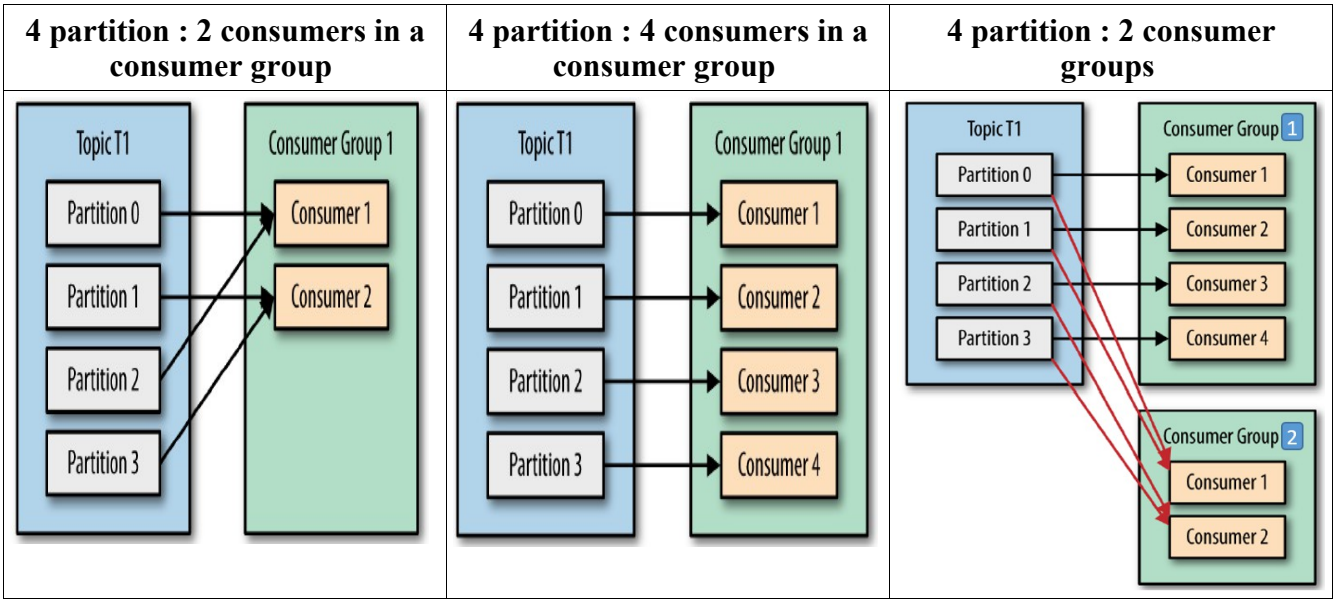
- A metric that defines how far the consumer is from reading the latest message committed to the partition
- The ideal lag value is 0 with many factors contributing to larger values

1.18 Consumer Group

- In essence, a bunch of consumers with a common group id
- Grouping of consumers helps parallelize reads across different consumers in the same group
- Kafka guarantees a message is only read by a single consumer in a group
  - Each topic partition is assigned to a single consumer in the group
  - The following constraint applies:
    - ✔ $\sum$ (consumers in a group) <= $\sum$ (partitions in a topic)
- Kafka checks for liveliness of consumers in a group
- In case of a consumer failure, Kafka rebalances the partitions between the remaining consumers

- Consumer group metadata is maintained in ZK

1.19 Consumer Group Diagram



Adapted from *Kafka: The Definitive Guide*

1.20 The Broker

- Acks the message receipt from the producer
- Commits the message to disk
  - Consumers can now read the message
- Replicates data
  - One copy is committed to the “main” partition, the other replicas are sent to machines on the cluster
  - Under-replicated data may lead to data loss
- In a cluster, one broker service is dedicated to act as the controller, performing administrative tasks like managing the states of partitions and replicas
- One broker process that handles all client (producers and consumers) reads and writes for a partition is called the "(partition) leader"

- more on this concept in a moment ...

Notes:

According to LinkedIn, in production, they are using:

Dual quad-core Intel Xeon machines with 24GB of memory

You need sufficient memory to buffer active readers and writers. You can do a back-of-the-envelope estimate of memory needs by assuming you want to be able to buffer for 30 seconds and compute your memory need as write_throughput*30

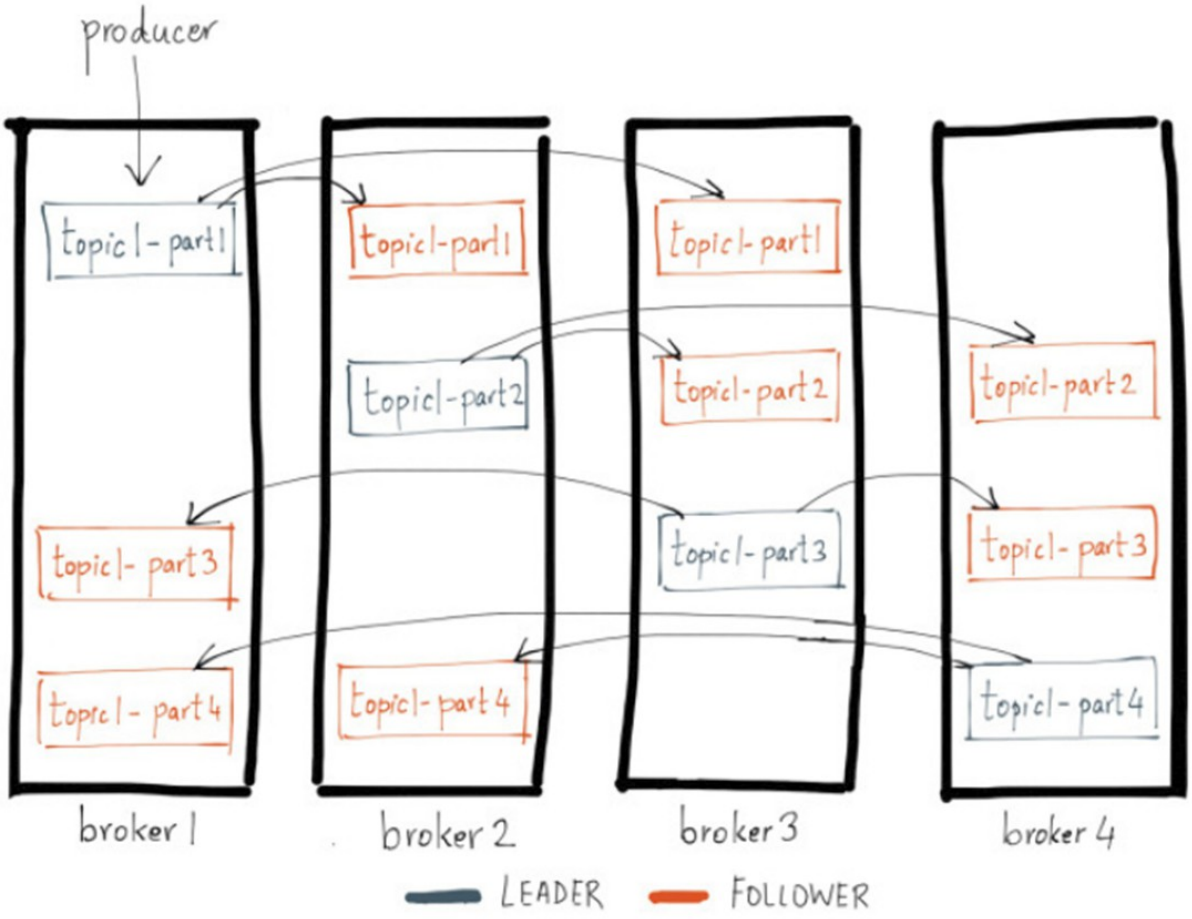
8x7200 rpm SATA drives

Kafka recommends using multiple drives to get good throughput and not sharing the same drives used for Kafka data with application logs or other OS filesystem activity to ensure good latency. You can either RAID these drives together into a single volume or format and mount each drive as its own directory.

1.21 The Leader and Followers Pattern

- The broker process managing the main partition replica acts as the "leader"; zero or more servers act as "followers"
  - Followers exist when the replica is set > 1
- The leader handles all client (producers and consumers) reads and writes
  - The followers passively replicate the leader's contents
- In case of the leader's failure, one of the followers will be automatically elected the "leader" (done by ZK)
- Each broker hosting partitions can act as a leader for some of its partitions and a follower for others
  - This arrangement ensures load balancing within the cluster
- If there is a broker conflict on owning the main partition (due to a system failure), the partition is marked as leaderless and it is taken offline

1.22 Partition Replication Diagram



- Topic configured with *replication.factor = 3*

Source: Apache Kafka Confluent Documentation

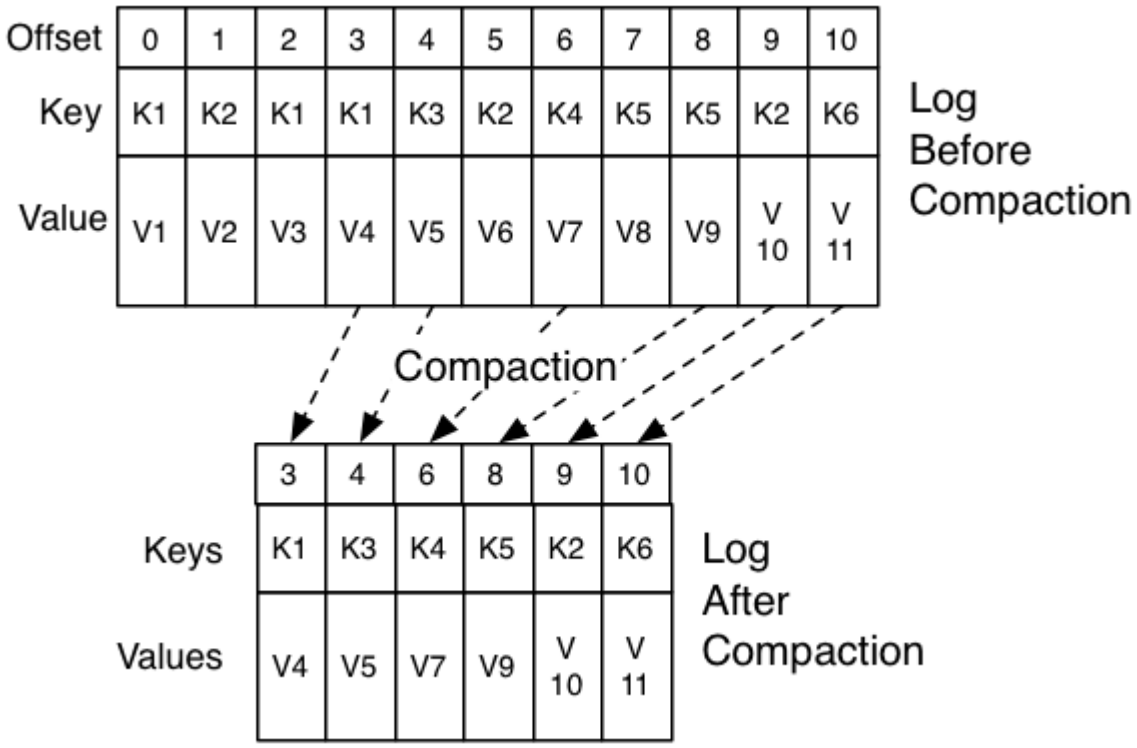
1.23 Controlling Message Durability with Minimum In-Sync Replicas

- This is done from the two sides: the producer's and the broker's
- A producer can set the *acks* request attribute to "all" (or "-1"), the server can further adjust this request through the value of the *min.insync.replicas* broker configuration parameter, which specifies the minimum number of replicas that must acknowledge a write for the write to be considered successful
- This parameter should be chosen based on the availability vs consistency trade-off analysis
  - Consult business stakeholders if in doubt

1.24 Log Compaction

- Log compaction is an optional feature that allows you to remove old records where more recent records with the same keys exist
  - Similar to the SQL's update operation
  - If you request a compacted record, you will get the latest record with the same key
- It gives you a per-record retention control, leading to smaller log sizes
- Message ordering is preserved

1.25 Log Compaction



Source: Apache Kafka Documentation

1.26 Operational Problems

- Most frequently problems arise as a result of:
  - **Rebalancing**
    - ✔ Assignment of consumers in consumer groups to partitions (brokers) is dynamic -- new consumers can join the consumer group, or die. Any such group change events results in rebalancing: re-assignment of the active consumers to partitions, which may take a while to accomplish during which time consumers are unassigned from the original partitions and left in off-line mode (no processing happens at this time!)
  - **Consumer lag**
    - ✔ The consumer side problem:
      - ✗ Slow message processing (possibly, unoptimized GC policy)

- Network issues
- Rebalancing!!

1.27 Summary

- In this chapter, we reviewed a number of important internal details of Kafka's functionality