



POLITECNICO
MILANO 1863

Apache Kafka

Alessandro Margara

`alessandro.margara@polimi.it`

`https://margara.faculty.polimi.it`

Communication semantics

Communication semantics

- In theory, message delivery can be provided with different guarantees
 1. At most once semantics: no duplicates, but messages can be lost
 2. At least once semantics: messages can be delivered more than once, but they are not lost
 3. Exactly once semantics: the system behaves as if each message was delivered once and exactly once

Communication semantics

- Exactly once semantics is theoretically impossible to guarantee in the presence of failures
 - It is equivalent to a distributed consensus problem
 - Possible in practice under some reasonable assumptions
- Consider a communication between a sender and a receiver
 - The receiver fails ...
 - ... but is re-instantiated later in time
- The receiver acknowledges messages ...
- ... and the sender does not receive an acknowledgement
 - It can send the message again
 - But if the message was received before the failure, it might be duplicated
 - At least once semantics
 - It can avoid sending the message
 - But if the message was not received before the failure, it is lost
 - At most once semantics

Communication semantics

- Which guarantees does Kafka offer?
- In Kafka, communication takes place in two steps
 - From producers to brokers
 - From brokers to consumers
- In recent versions, Kafka offers exactly once semantics (EOS) for both steps

EOS on the producer

- The producer can be configured to be idempotent
 - An idempotent operation can be performed many times without causing any different effect than being performed once
 - This removes the possibility of duplicate messages being delivered to a topic
- Implementation
 - The producer waits for an ack from the broker
 - If it does not receive the ack, after a timeout it sends the message again (possible duplication of messages)
 - The producer attaches sequence numbers to messages ...
 - ... that the broker can use to discard duplicates

EOS on the consumer

- A consumer repeatedly performs three actions
 - Reads a message
 - Processes the message
 - Produces some results
- EOS requires that the results are produced once and only once

EOS on the consumer

- Kafka provides EOS in the consumer under the following assumptions
 - Results are written to a Kafka topic
 - Kafka topics are reliable (achieve through replication)
- Implementation using transactional writes
 - Possibility to write multiple messages as part of a transaction
 - Either all or none of the messages are written
 - The consumer can write the results and advance the offsets of consumed messages within a transaction
 - If the offset is updated, the results are written
 - If the results are not written, the offset is not updated

End-to-end EOS

- This solution enables end-to-end exactly once semantics
 - In the case of many intermediate processing steps ...
 - ... if each step writes its results in Kafka topics ...
 - ... each step guarantees exactly once semantics ...
 - ... resulting in exactly once semantics being guaranteed end-to-end!
- Useful for services that require multiple processing steps
 - Used to implement Kafka Streams

Transactions

- EOS relies on transactional (atomic) updates of offsets
- Transactions are implemented using a transaction coordinator
 - Writes metadata about transactions in a special Kafka log
 - Same guarantees as other logs
 - Ordered, idempotent, fault-tolerant
- Consumers can choose between two different isolation levels
 - Read committed: read only writes that have been committed as part of a transaction
 - Read uncommitted: read also writes that have not been yet committed

Log compaction

- To achieve EOS, consumers need to store any state/result in Kafka
 - Kafka has no control on state that is outside the system (e.g., in memory or in external databases)
- In many cases, new events have update semantics
 - Their effect is to update/overwrite the previous state of a consumer
- Kafka offers a special message storage configuration to deal with this specific (and frequent) situation

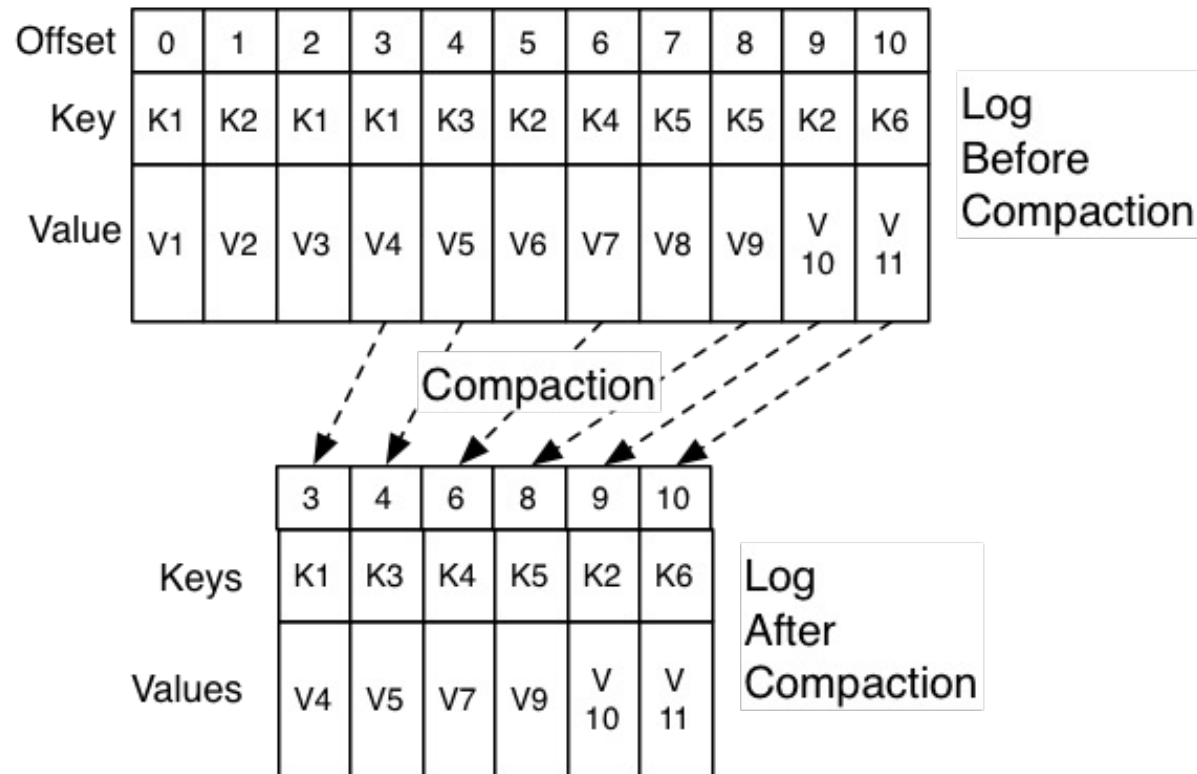
Log compaction

- By default, all messages are stored in Kafka topics for a configurable retention time
 - This is reasonable for all those scenarios where we want to store the history of messages
 - However, it is not optimal when old messages become irrelevant with the arrival of new messages

Log compaction

- To support scenarios where we are only interested in the most recent state, Kafka introduces log compaction
 - A specific retention policy
 - Configurable per topic
 - Allows Kafka to only keep the last value for each key
 - Preserves the offset and order of messages
 - Simply deletes messages when a more recent one with the same key becomes available

Log compaction



Log compaction

- Log compaction is not performed as soon as new messages are received
 - Log is split in two parts (head and tail)
 - Messages are marked for deletion only in the tail
 - Actual removal of messages triggered periodically

