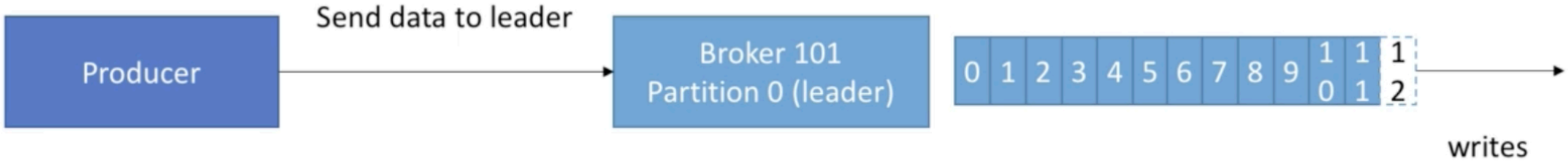
Producers Acks Deep Dive
acks = 0 (no acks)

- No response is requested
- If the broker goes offline or an exception happens, we won't know and will lose data



- Useful for data where it's okay to potentially lose messages:
  - Metrics collection
  - Log collection

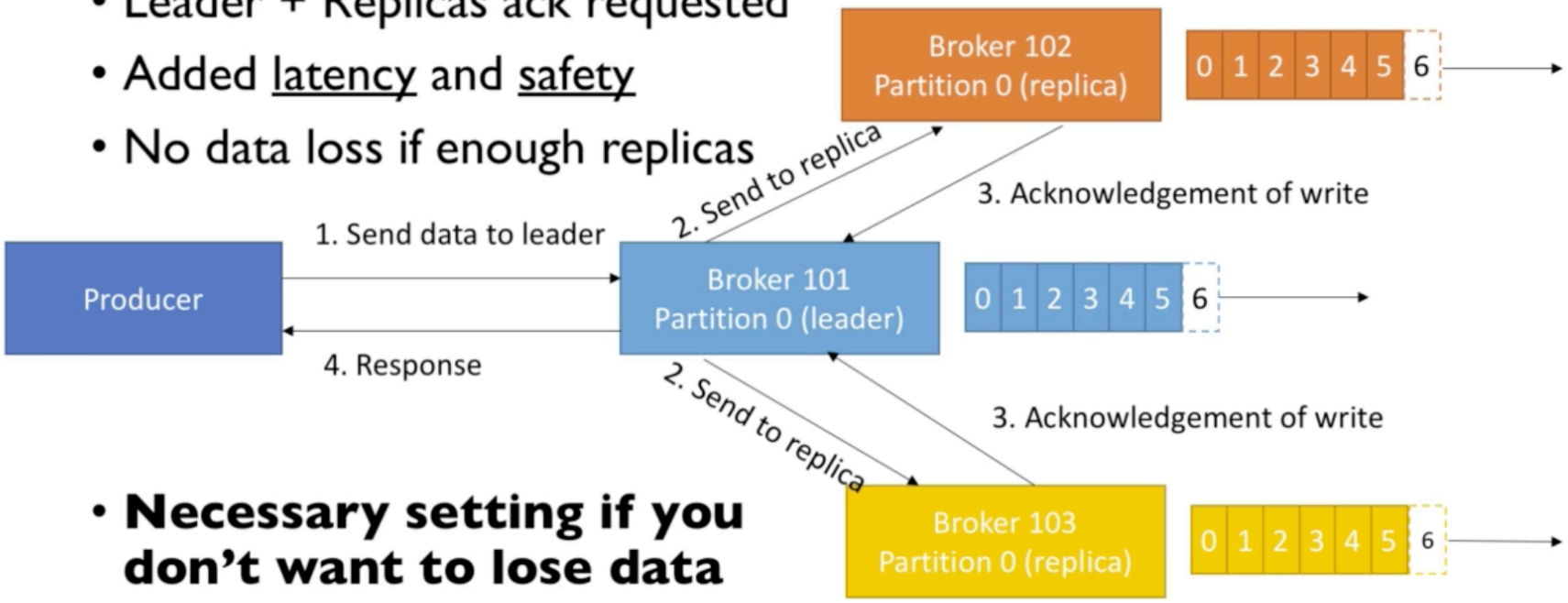
Producers Acks Deep Dive
acks = 1 (leader acks)

- Leader response is requested, but replication is not a guarantee (happens in the background)
- If an ack is not received, the producer may retry

- If the leader broker goes offline but replicas haven't replicated the data yet, we have a data loss.

Producers Acks Deep Dive
acks = all (replicas acks)

- Leader + Replicas ack requested
- Added latency and safety
- No data loss if enough replicas

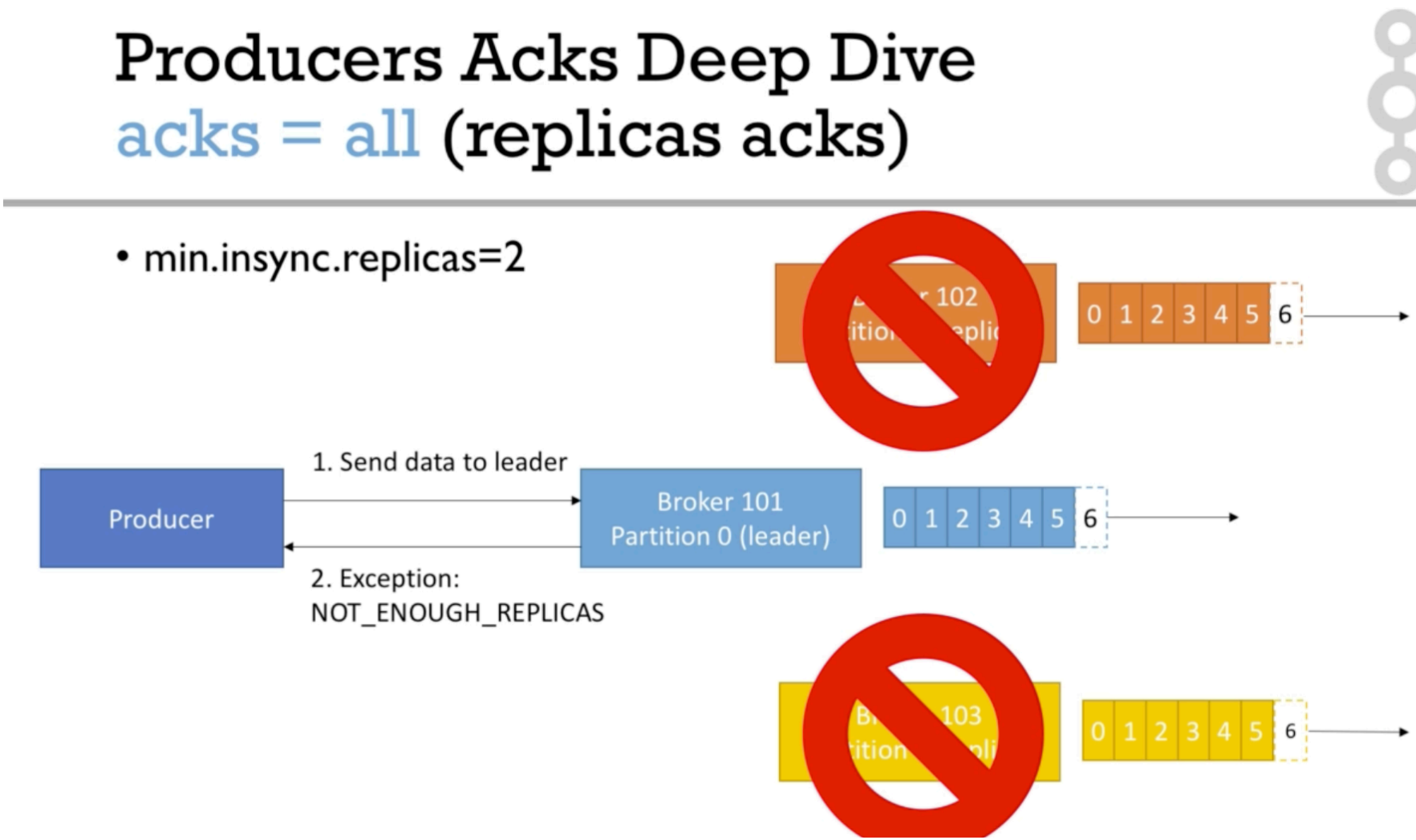


- **Necessary setting if you don't want to lose data**

Producers Acks Deep Dive
acks = all (replicas acks)

- Acks=all must be used in conjunction with min.insync.replicas.
- min.insync.replicas can be set at the broker or topic level (override).
- min.insync.replicas=2 implies that at least 2 brokers that are ISR (including leader) must respond that they have the data.
- That means if you use replication.factor=3, min.insync=2, acks=all, you can only tolerate 1 broker going down, otherwise the producer will receive an exception on send.

Producers Acks Deep Dive
acks = all (replicas acks)



Producer retries

- In case of transient failures, developers are expected to handle exceptions, otherwise the data will be lost.
- Example of transient failure:
  - NotEnoughReplicasException
- There is a "retries" setting
  - defaults to 0
  - You can increase to a high number, ex Integer.MAX_VALUE

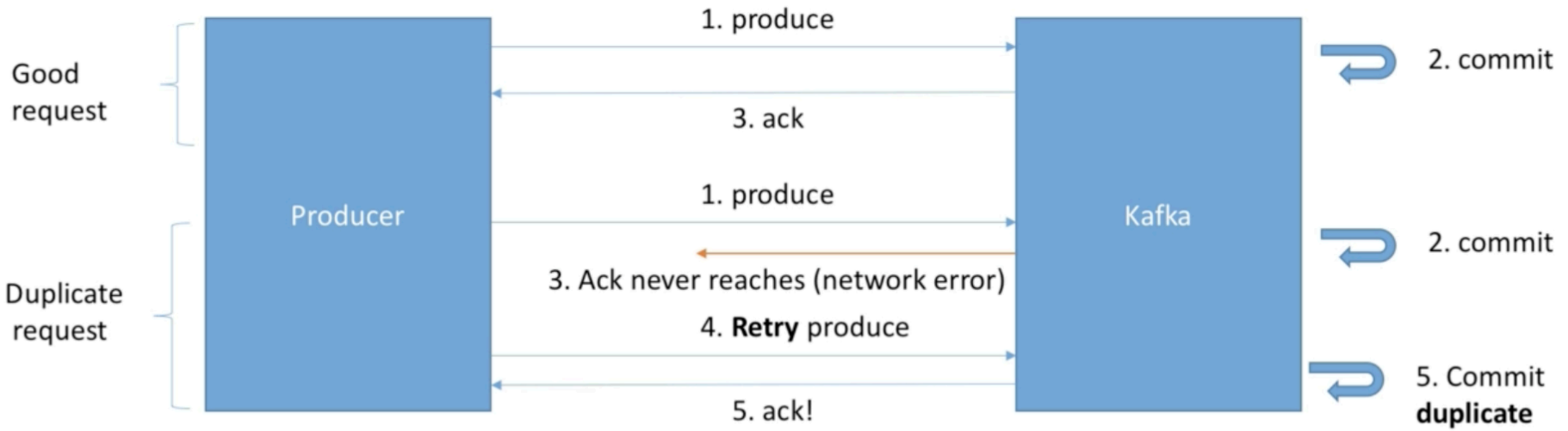
Producer retries

- In case of retries, by default there is a chance that messages will be sent out of order (if a batch has failed to be sent).
- If you rely on key-based ordering, that can be an issue.
- For this, you can set the setting while controls how many produce requests can be made in parallel: max.in.flight.requests.per.connection
  - Default: 5
  - Set it to 1 if you need to ensure ordering (may impact throughput)

- In Kafka >= 1.0.0, there's a better solution!

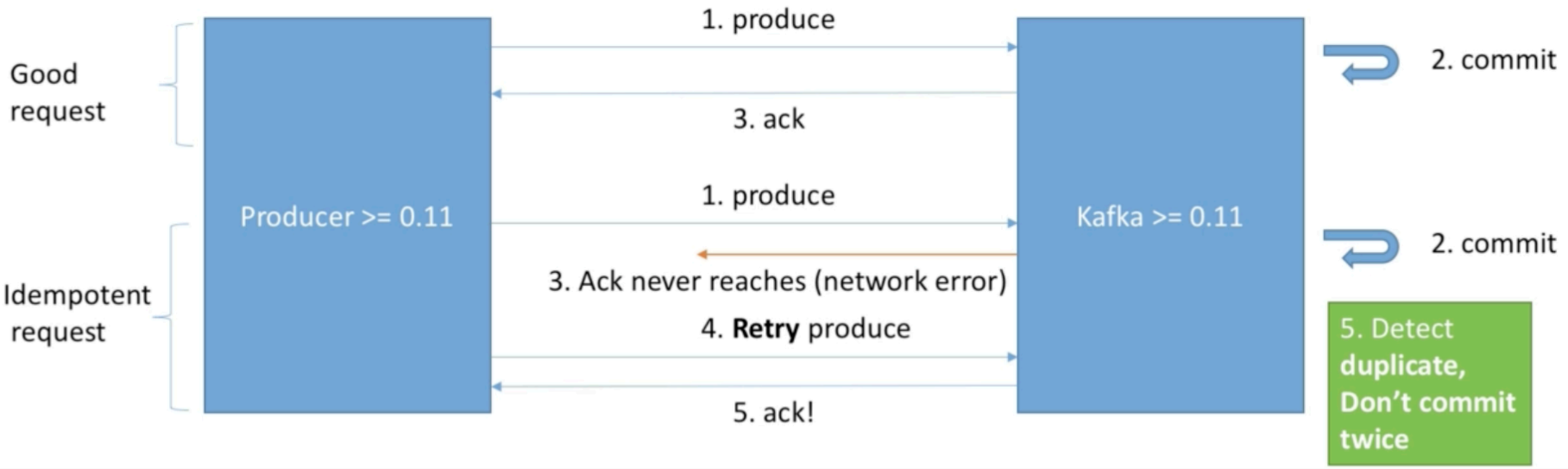
Idempotent Producer

- Here's the problem: the Producer can introduce duplicate messages in Kafka due to network errors



Idempotent Producer

- In Kafka >= 0.11, you can define a "idempotent producer" which won't introduce duplicates on network error



Idempotent Producer

- Idempotent producers are great to guarantee a stable and safe pipeline!

- They come with:
  - retries = Integer.MAX_VALUE (2^31–1 = 2147483647)
  - max.in.flight.requests=1 (Kafka >= 0.11 & < 1.1) or
  - max.in.flight.requests=5 (Kafka >= 1.1 – higher performance)
  - acks=all

- Just set:
  - **producerProps.put("enable.idempotence", true);**

Safe producer Summary & Demo

Kafka < 0.11

- acks=all (producer level)
  - Ensures data is properly replicated before an ack is received
- min.insync.replicas=2 (broker/topic level)
  - Ensures two brokers in ISR at least have the data after an ack
- retries=MAX_INT (producer level)
  - Ensures transient errors are retried indefinitely
- max.in.flight.requests.per.connection=1 (producer level)
  - Ensures only one request is tried at any time, preventing message re-ordering in case of retries

Kafka >= 0.11

- enable.idempotence=true (producer level) + min.insync.replicas=2 (broker/topic level)
  - Implies acks=all, retries=MAX_INT, max.in.flight.requests.per.connection=5 (default)
  - while keeping ordering guarantees and improving performance!

- Running a "safe producer" might impact throughput and latency, always test for your use case

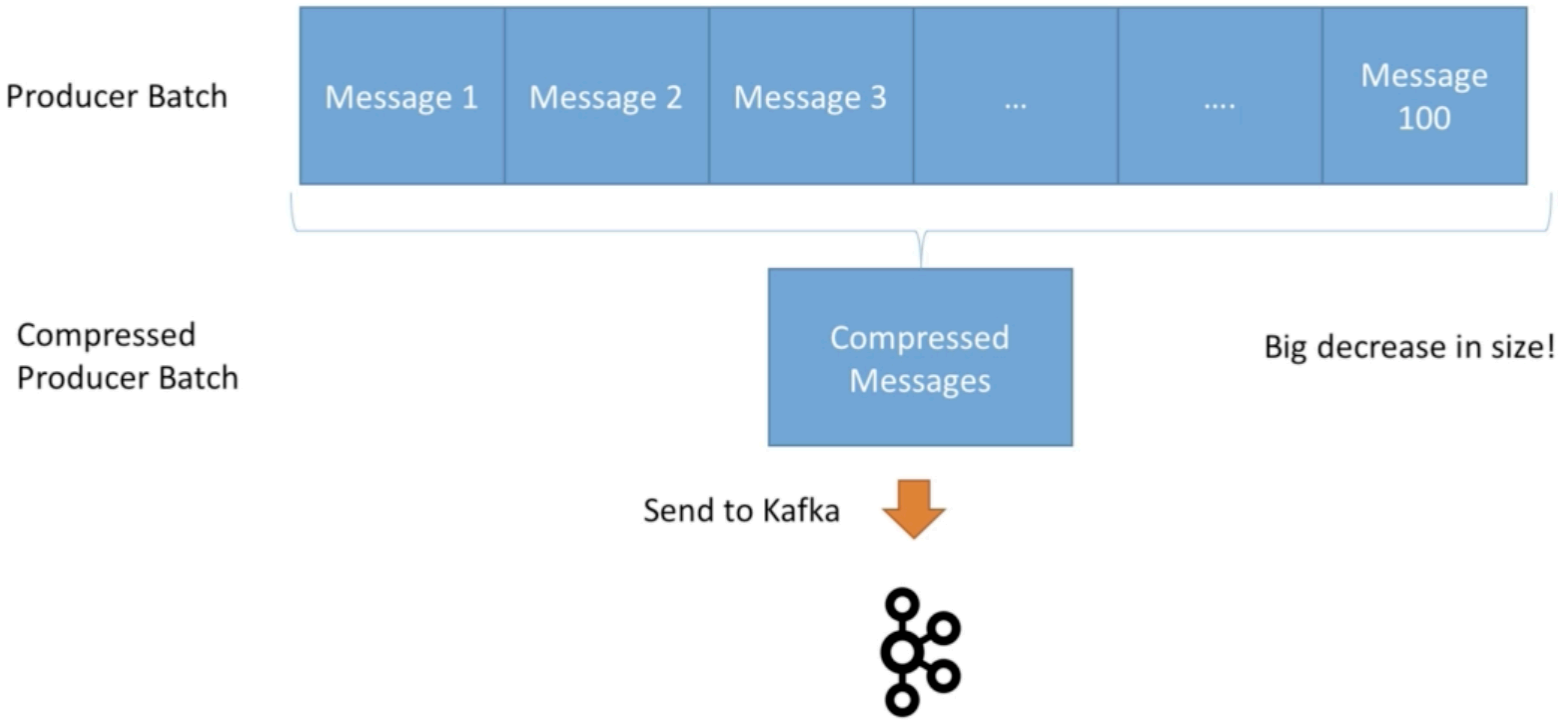
Message Compression

- Producer usually send data that is text-based, for example with JSON data
- In this case, it is important to apply compression to the producer.

- Compression is enabled at the Producer level and doesn't require any configuration change in the Brokers or in the Consumers
- "compression.type" can be 'none' (default), 'gzip', 'lz4', 'snappy'

- Compression is more effective the bigger the batch of message being sent to Kafka!
- Benchmarks here: https://blog.cloudflare.com/squeezing-the-firehose/

Message Compression



Message Compression

- The compressed batch has the following advantage:
  - Much smaller producer request size (compression ratio up to 4x!)
  - Faster to transfer data over the network => less latency
  - Better throughput
  - Better disk utilisation in Kafka (stored messages on disk are smaller)
- Disadvantages (very minor):
  - Producers must commit some CPU cycles to compression
  - Consumers must commit some CPU cycles to decompression
- Overall:
  - Consider testing snappy or lz4 for optimal speed / compression ratio

Message Compression Recommendations

- Find a compression algorithm that gives you the best performance for your specific data. Test all of them!
- Always use compression in production and especially if you have high throughput
- Consider tweaking linger.ms and batch.size to have bigger batches, and therefore more compression and higher throughput

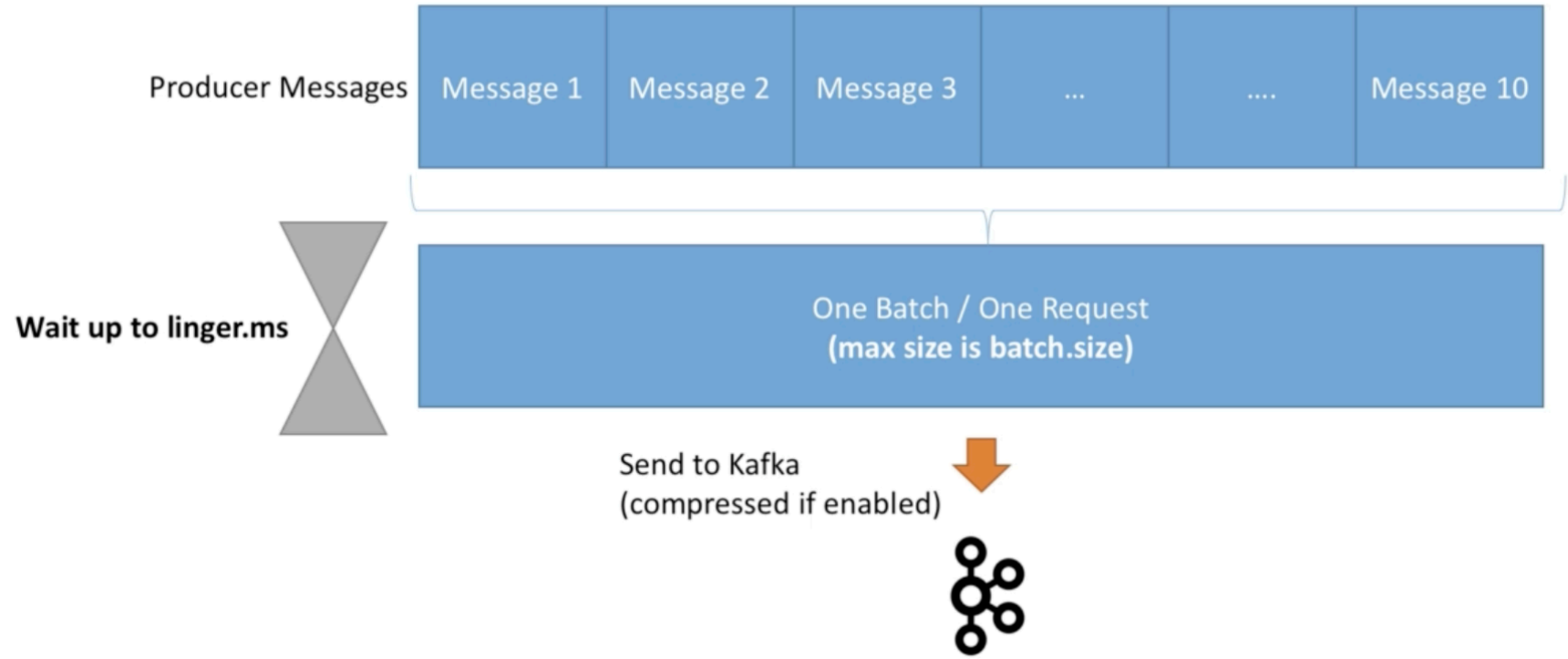
Linger.ms & batch.size

- By default, Kafka tries to send records as soon as possible
  - It will have up to 5 requests in flight, meaning up to 5 messages individually sent at the same time.
  - After this, if more messages have to be sent while others are in flight, Kafka is smart and will start batching them while they wait to send them all at once.
- This smart batching allows Kafka to increase throughput while maintaining very low latency.
- Batches have higher compression ratio so better efficiency
- So how can we control the batching mechanism?

Linger.ms & batch.size

- **Linger.ms:** Number of milliseconds a producer is willing to wait before sending a batch out. (default 0)
- By introducing some lag (for example linger.ms=5), we increase the chances of messages being sent together in a batch
- So at the expense of introducing a small delay, we can increase throughput, compression and efficiency of our producer.
- If a batch is full (see batch.size) before the end of the linger.ms period, it will be sent to Kafka right away!

Linger.ms



Linger.ms & batch.size

- **batch.size:** Maximum number of bytes that will be included in a batch. The default is 16KB.

- Increasing a batch size to something like 32KB or 64KB can help increasing the compression, throughput, and efficiency of requests
- Any message that is bigger than the batch size will not be batched
- A batch is allocated per partition, so make sure that you don't set it to a number that's too high, otherwise you'll run waste memory!
- (Note: You can monitor the average batch size metric using Kafka Producer Metrics)

Producer Default Partitioner and how keys are hashed

- By default, your keys are hashed using the "murmur2" algorithm.
- It is most likely preferred to not override the behavior of the partitioner, but it is possible to do so (partitioner.class).
- The formula is:
  targetPartition = Utils.abs(Utils.murmur2(record.key())) % numPartitions;
- This means that same key will go to the same partition (we already know this), and adding partitions to a topic will completely alter the formula

Max.block.ms & buffer.memory

- If the producer produces faster than the broker can take, the records will be buffered in memory
- **buffer.memory=33554432 (32MB):** the size of the send buffer

- That buffer will fill up over time and fill back down when the throughput to the broker increases

Max.block.ms & buffer.memory

- If that buffer is full (all 32MB), then the .send() method will start to block (won't return right away)

- **max.block.ms=60000**: the time the .send() will block until throwing an exception. Exceptions are basically thrown when
  - The producer has filled up its buffer
  - The broker is not accepting any new data
  - 60 seconds has elapsed.

- If you hit an exception hit that usually means your brokers are down or overloaded as they can't respond to requests