Apache Kafka

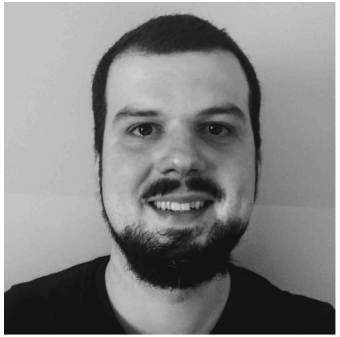
for Java Developers

Organisation and Workshop Agenda

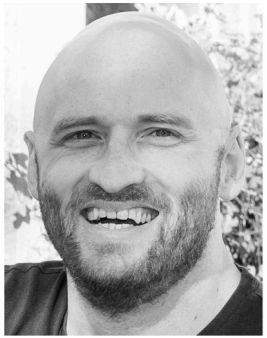
Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

Who are we?

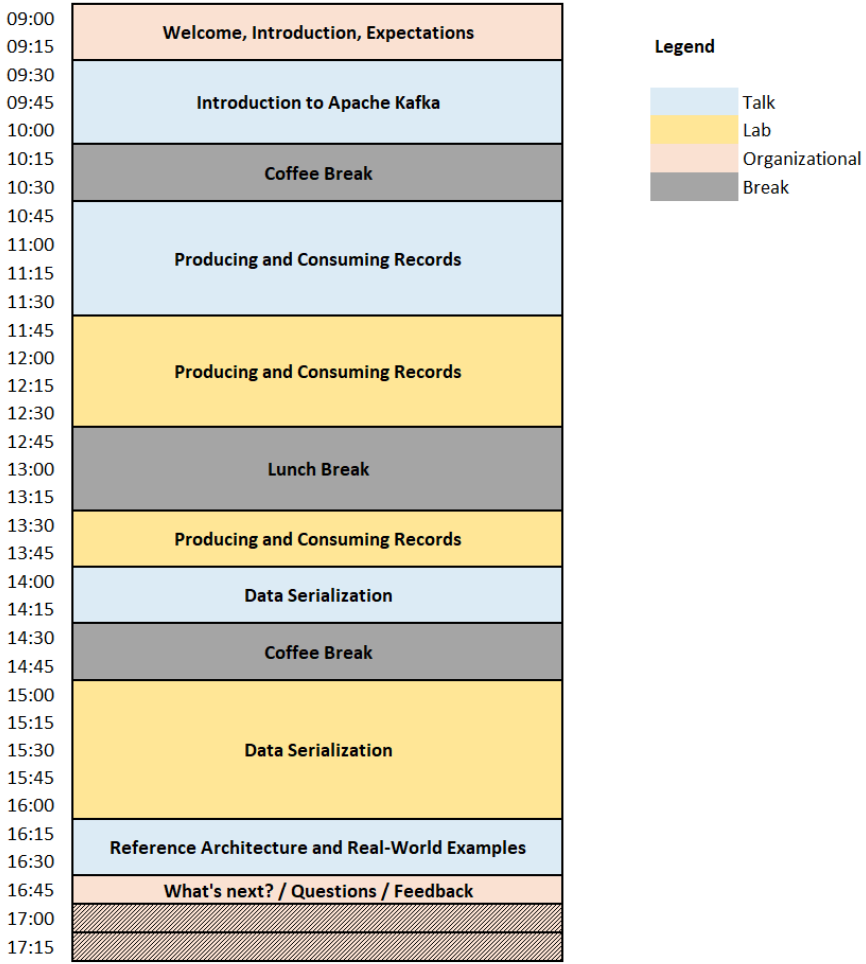


Boris Fresow is a freelance IT consultant and trainer. He mainly deals with distributed systems, messaging solutions, and training on these topics.



Markus Günther is a freelance software developer, architect, and trainer. He supports his clients in implementing robust, scalable systems and also offers seminars & workshops on messaging solutions and modern software methodology.

Apache Kafka for Java Developers



What we'll provide

- Material in terms of slides and accompanying notes
- Assignments for certain topics
  - Isolated GitHub repositories
  - `README.md`: General information
  - `ASSIGNMENT.md`: Tasks
  - `HINTS.md`: In case you're stuck
- Project files and / or Docker-based local environment

What you'll need

- Working installation of
  - JDK 11 (at least)
  - Docker and Docker Compose
- Preferably IntelliJ IDEA
  - doesn't matter if Community or Ultimate Edition

Goals

- Understand core concepts of Apache Kafka

- Learn ...
  - ... how to integrate Kafka into Java applications
  - ... to use the Kafka Client library for Java
  - ... about serialization strategies and their differences
  - ... to reason about event-based architectures

What are your expectations?

Questions?

Apache Kafka

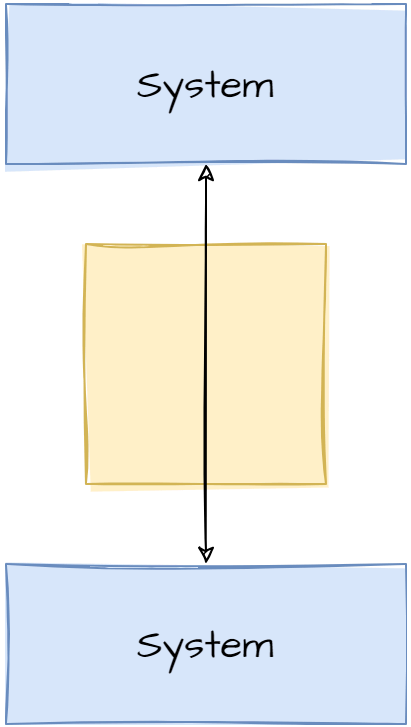
for Java Developers

Introduction to Apache Kafka

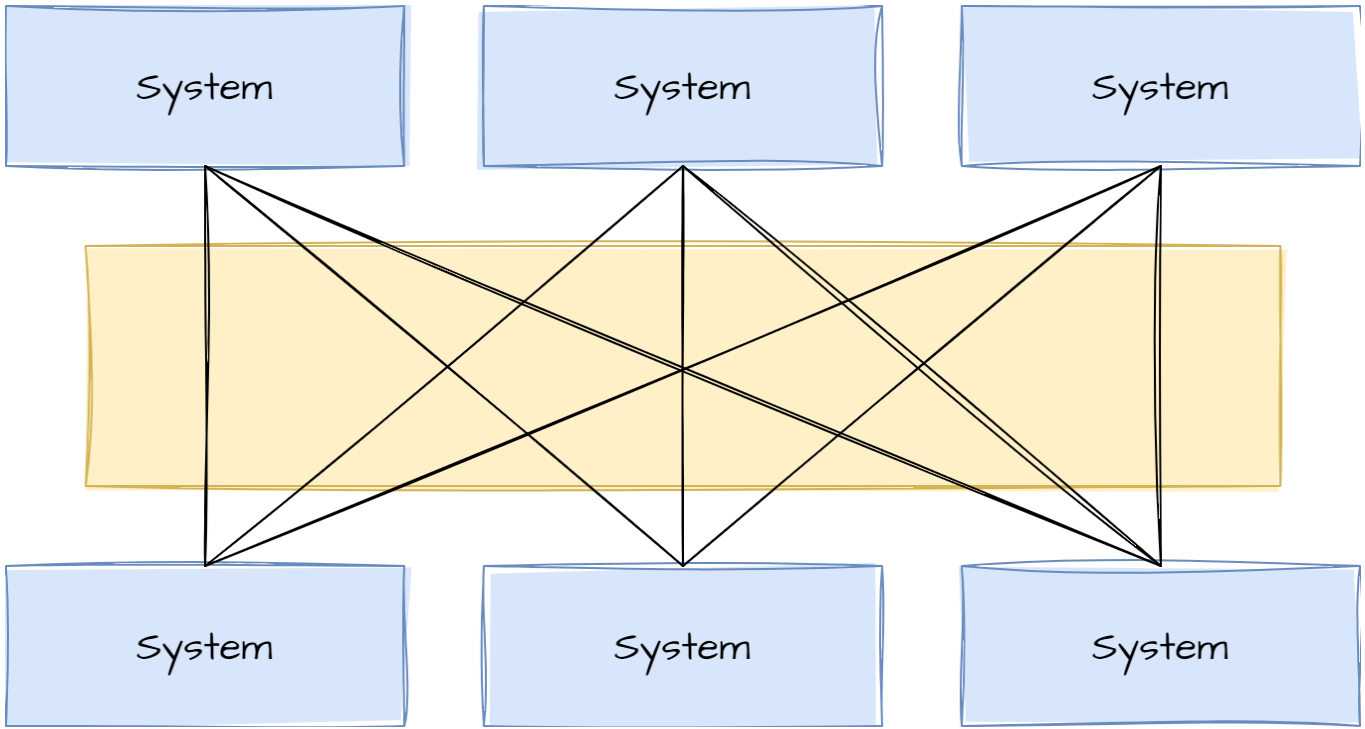
Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

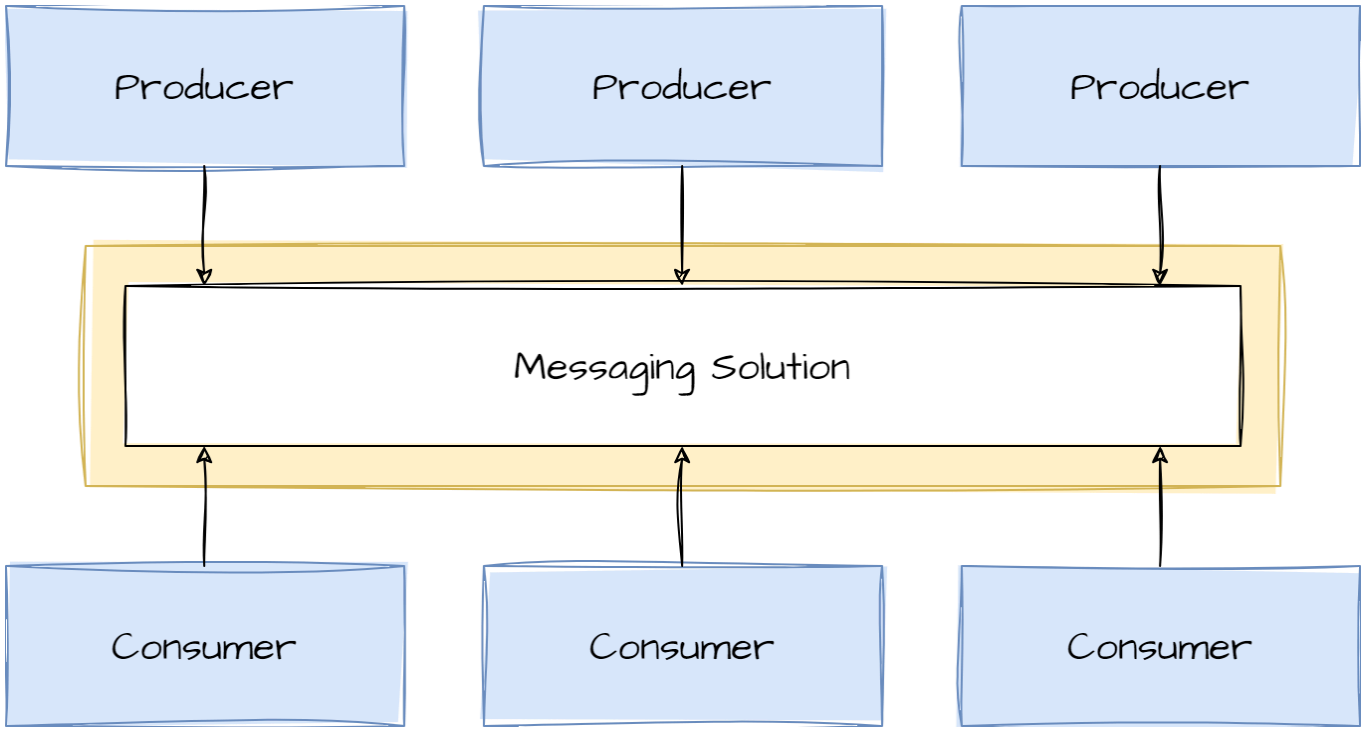
Point-to-point communication is simple to maintain - especially with few systems involved.



More systems increase the complexity of communication channels in this architecture.



A messaging solution decouples producing from consuming systems.



- Systems become decoupled and easier to maintain because they can produce and consume data without knowledge of other systems.

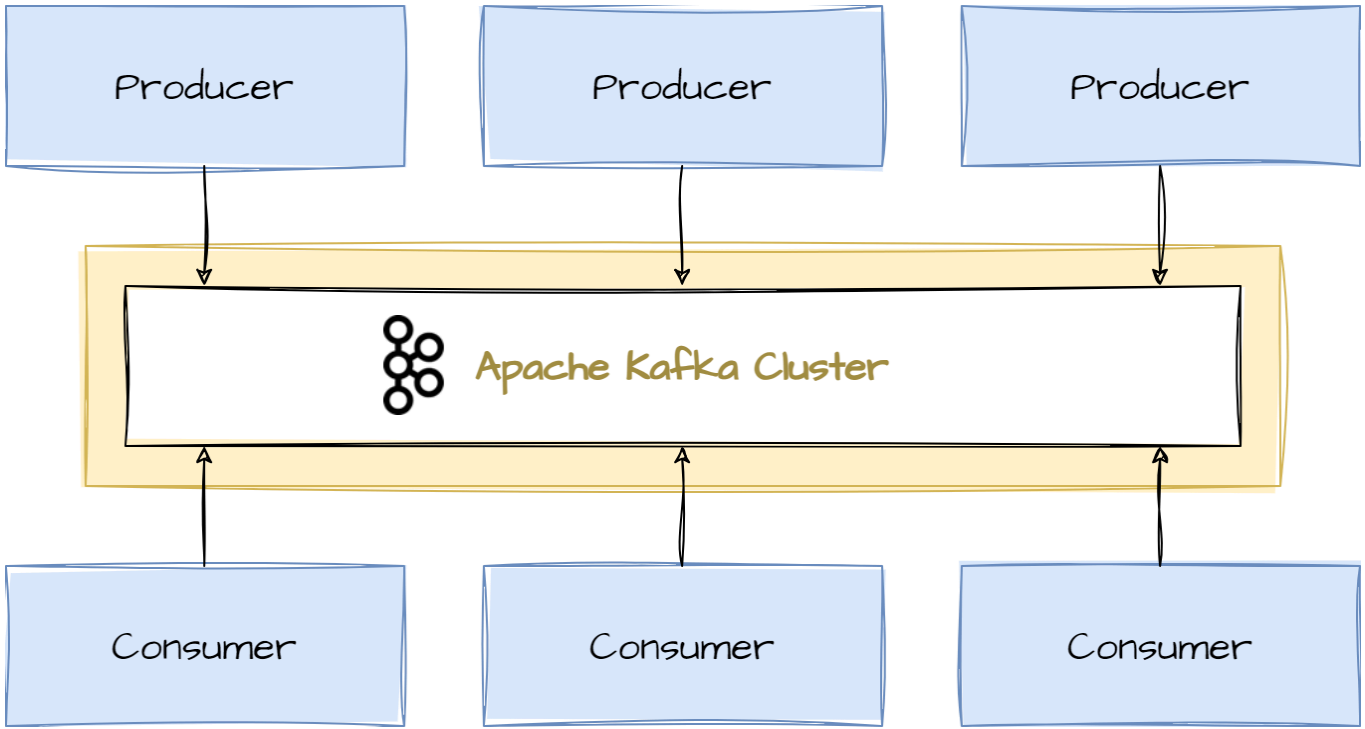
- Asynchronous communication comes with stronger delivery guarantees. If a consumer goes down, it will simply pick up from where it left off when it comes back online again (or shift load to other consumers of a group of consumers).

- Systems can standardize on the communication protocol, as well as scaling strategies and fault-tolerance mechanisms.

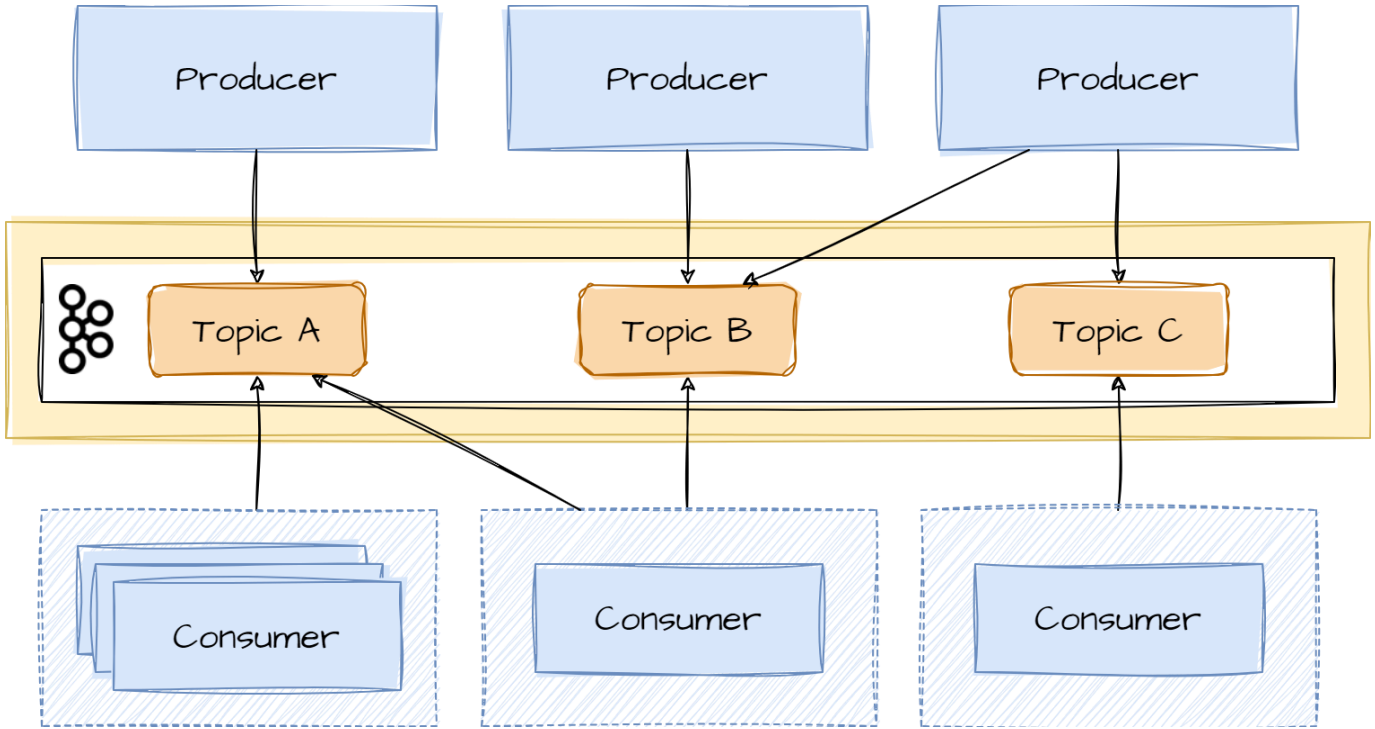
- Consumers consume data at a rate they can handle.

- Systems can rebuild their state anytime by replaying events in a topic.

Apache Kafka supports this communication model.



Producers publish data to topics, consumers subscribe to them and read at their own pace.



- Instead of having multiple systems communicate directly with each other, producers simply publish their data to one or more topics, without caring who comes along to read the data.

- Topics are named streams (or channels) of related data that are stored in a Kafka cluster.

- Topics serve a similar purpose as tables in a database. In fact, there is an underlying duality between a stream and a table that is important when we talk about stream processing for analytics.

- The data in a topic imposes no particular schema as far as Kafka is concerned. Kafka stores raw binary data. You can of course enforce schemas at the application level.

Apache Kafka is a distributed pub-sub messaging system with topic access semantics.

History

- Apache Kafka originated at LinkedIn
- Maintained by the Apache Foundation
- Confluent drives further development
- Confluent provides various system components that enrich the Kafka ecosystem

Apache Kafka is a distributed pub-sub messaging system with topic access semantics. (cont.)

Intentions

- Designed for near-real-time processing of events
- Supports multiple delivery semantics
  - At-least-once
  - Exactly-once (well, not quite)
- Optimized binary protocol for client-to-broker communication
  - No integration with JMS ...

Apache Kafka is a distributed pub-sub messaging system with topic access semantics. (cont.)

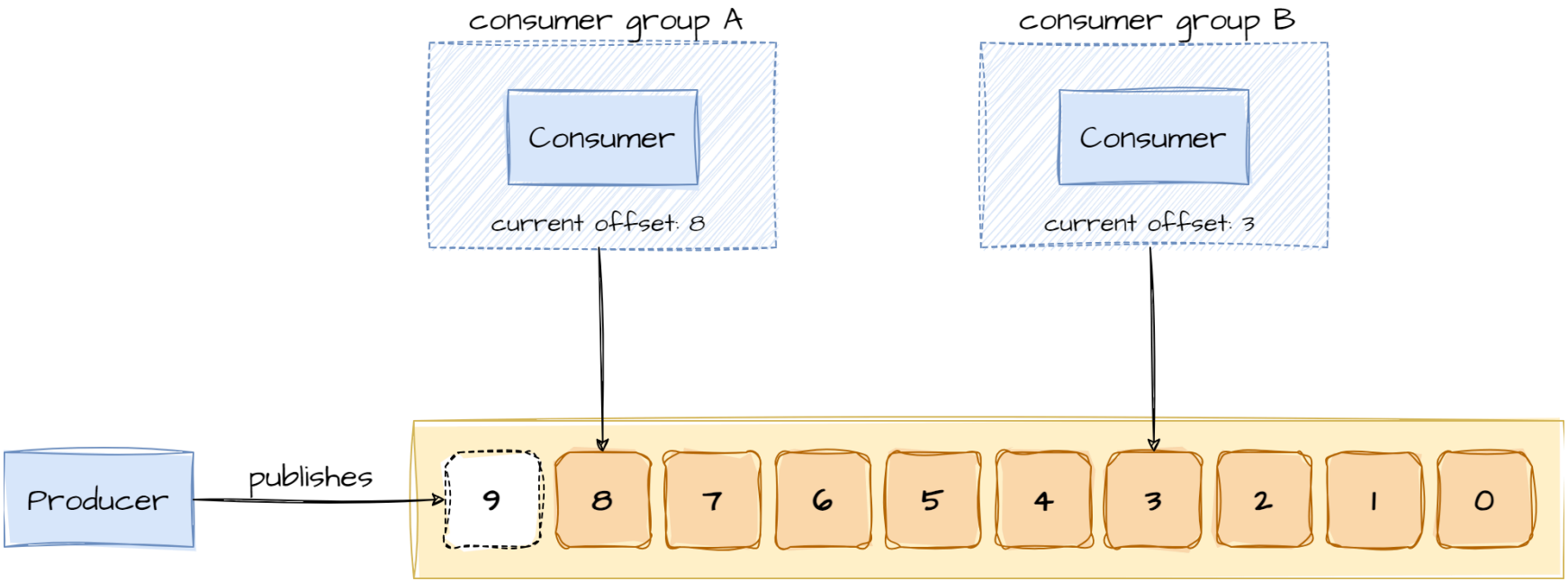
Innovations

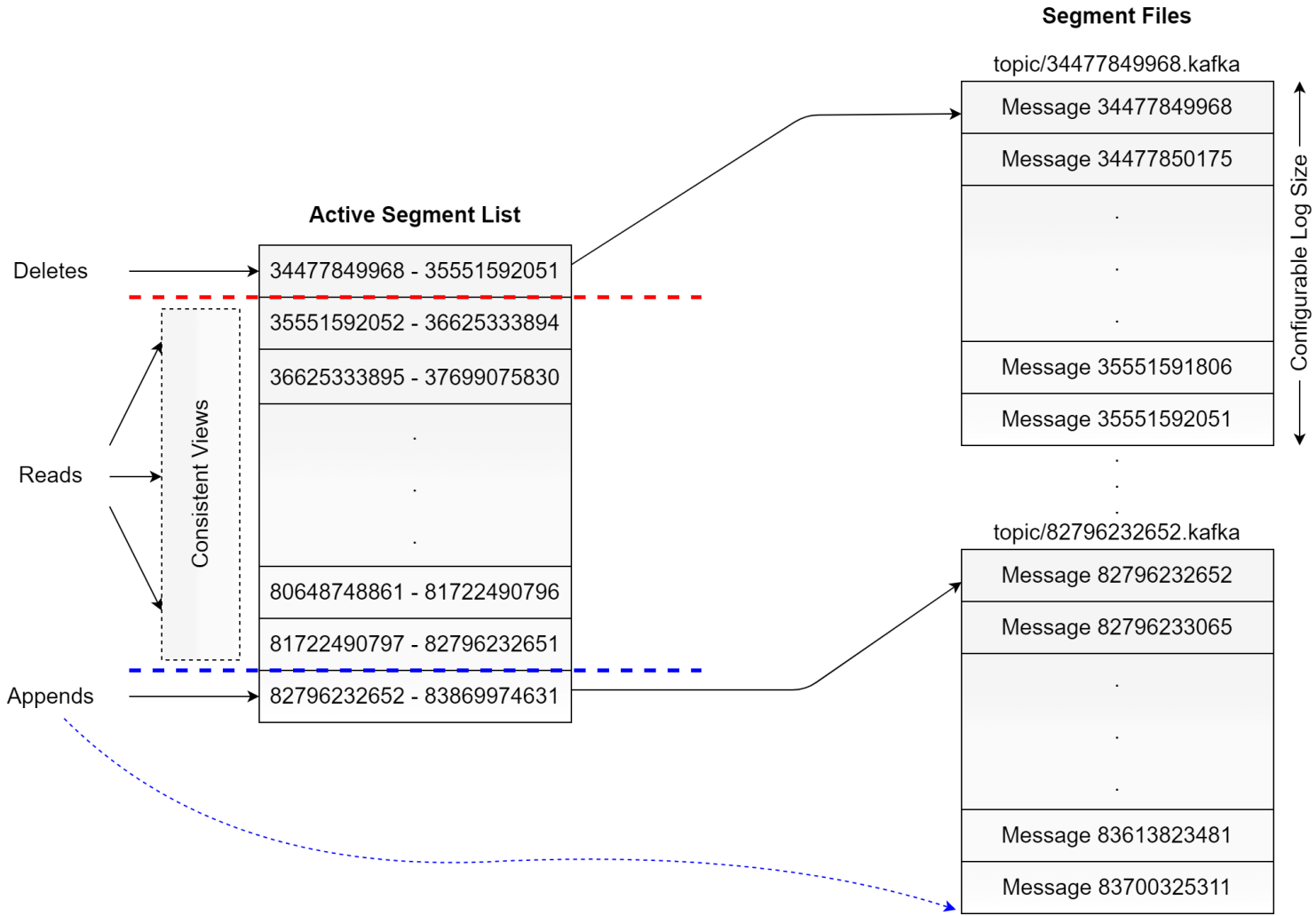
- Messages are acknowledged in order
- Messages are persisted for days / weeks / indefinite
- Consumers manage their offsets
- Very powerful configuration settings allow ...
  - different read/write strategies on the same topic
  - performance tuning on a very low level

- Messages are acknowledged in order: This eliminates the need to track acknowledgements on a per-message, per-listener basis and allows a reader's operation to be very similar to reading a file.
- Messages are persisted for days / weeks / indefinite: This eliminates the requirement to track when readers have finished with particular messages by allowing the retention time to be set so long that readers are almost certain to be finished with messages before they are added.
- Consumers manage their offsets: Consumers control from which offset they continue to read. Applications can even manage their offsets outside of Kafka entirely.

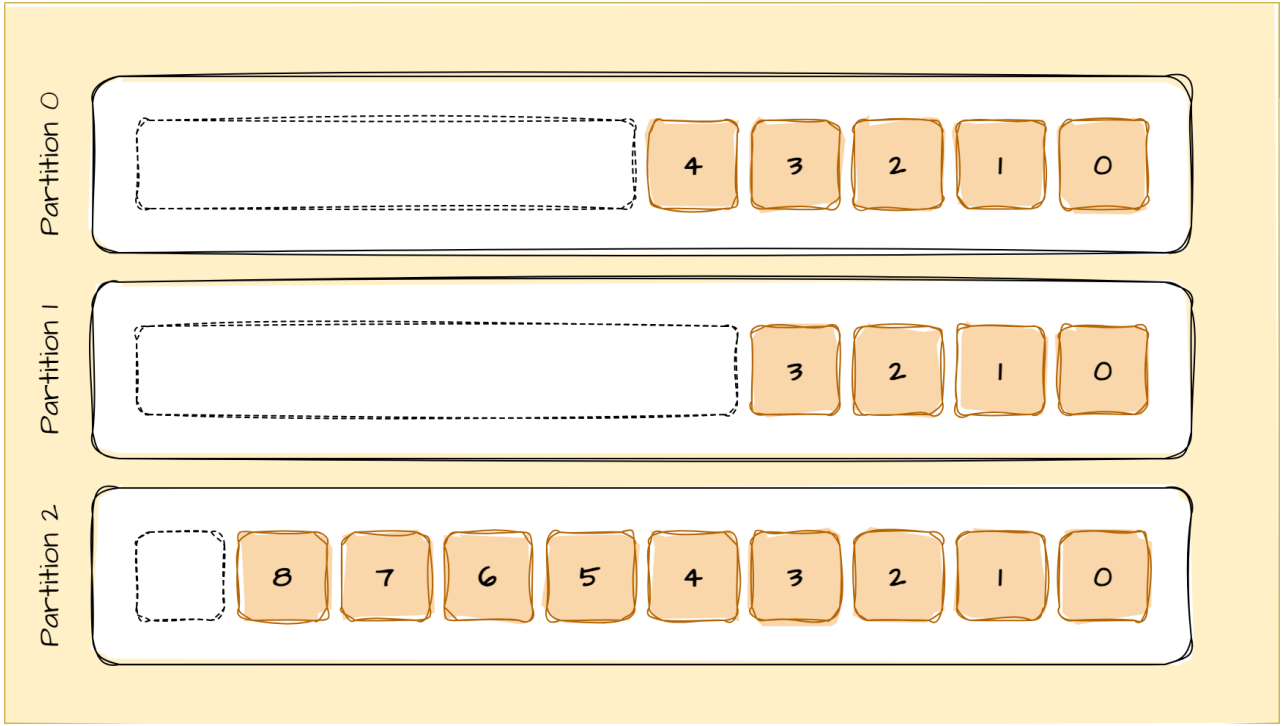
The technical impact of these innovations is that Kafka can write messages to a file system. The files are written sequentially as messages are produced, and they are read sequentially as messages are consumed. These design decisions mean that nonsequential reading or writing of files by a Kafka message broker is very, very rare, and that lets Kafka handle messages at very high speeds.

Kafka uses a persistent log. Producers append to it, and consumers read from it sequentially.

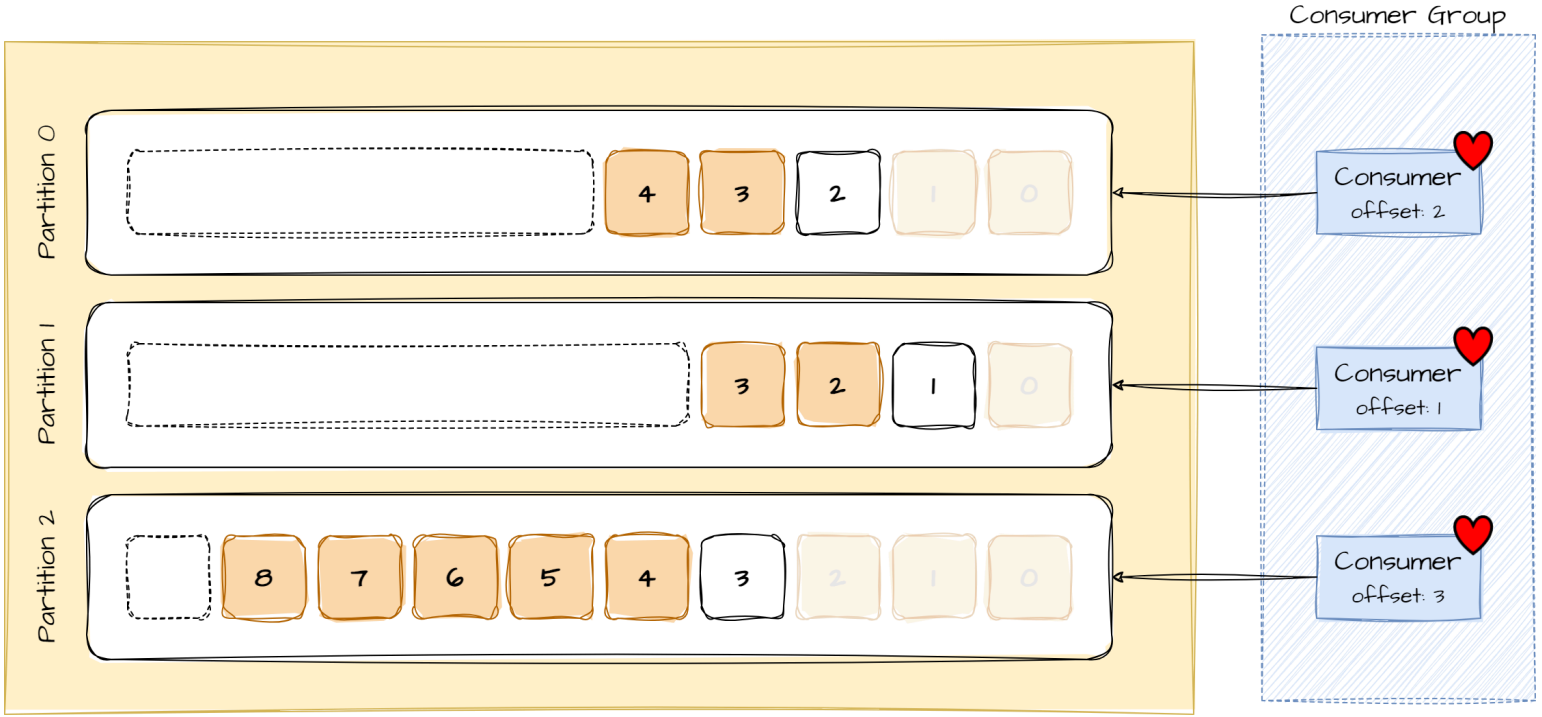



Segment Files
topic/34477849968.kafka
Message 34477849968
Message 34477850175
Message 35551591806
Message 35551592051
Configurable Log Size
topic/82796232652.kafka
Message 82796232652
Message 82796233065
Message 83613823481
Message 83700325311
Active Segment List
Deletes
Reads
Appends
Consistent Views
34477849968 - 35551592051
35551592052 - 36625333894
36625333895 - 37699075830
80648748861 - 81722490796
81722490797 - 82796232651
82796232652 - 83869974631

A Kafka topic is comprised of at least one partition.

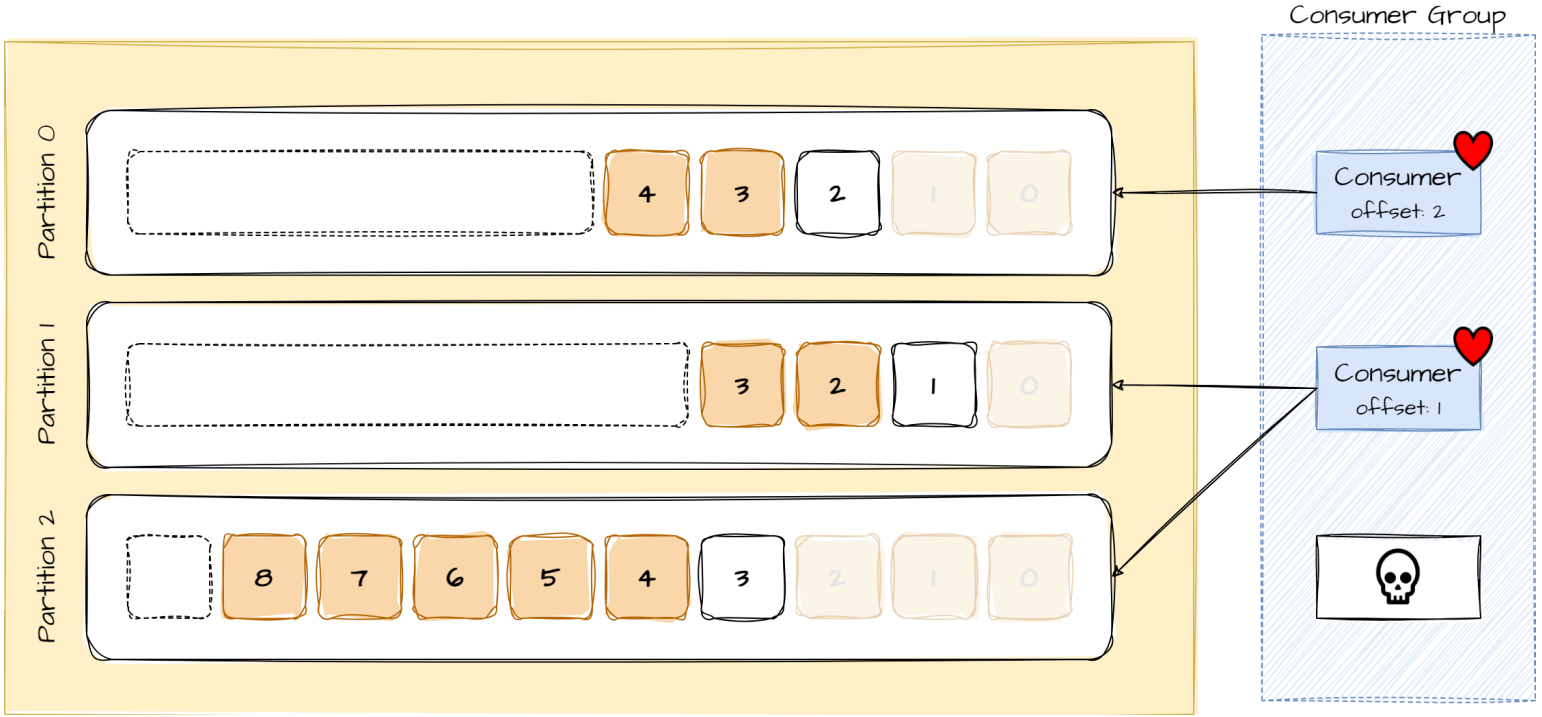


Consumers that participate in the same consumer group share the read workload on a topic.

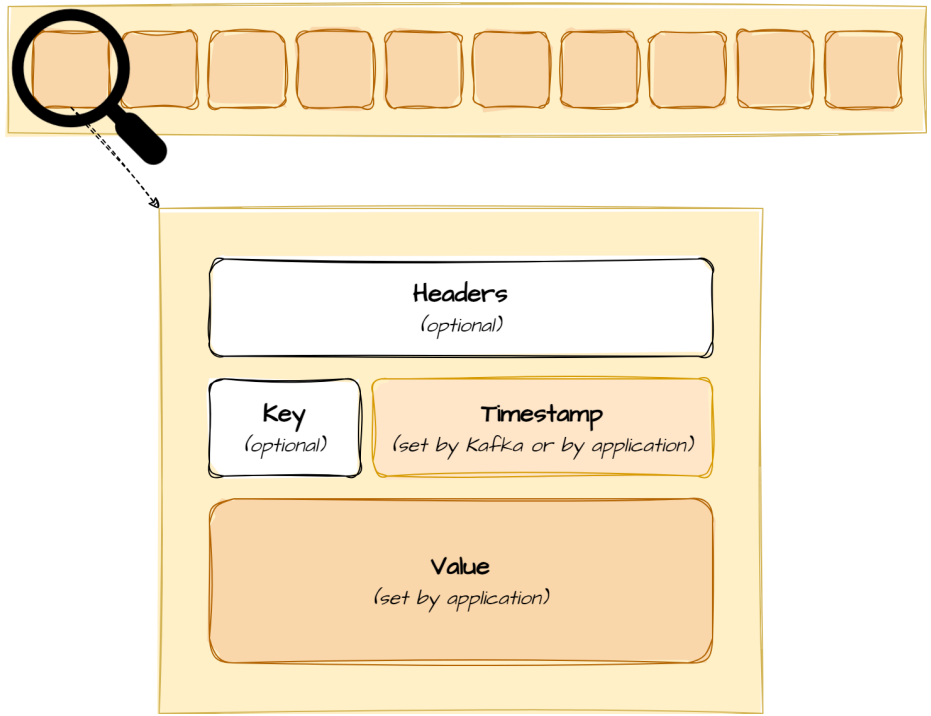


Recall: Messages are acknowledged in order. Since a producer only appends to the log and at most one consumer can consume messages from a topic-partition, this guarantees that we read all messages from a specific topic-partition in the same order in which they have been published.

Kafka redistributes work if a consumer fails and is no longer able to process messages.



A message (or record, or event) contains metadata alongside the message payload.



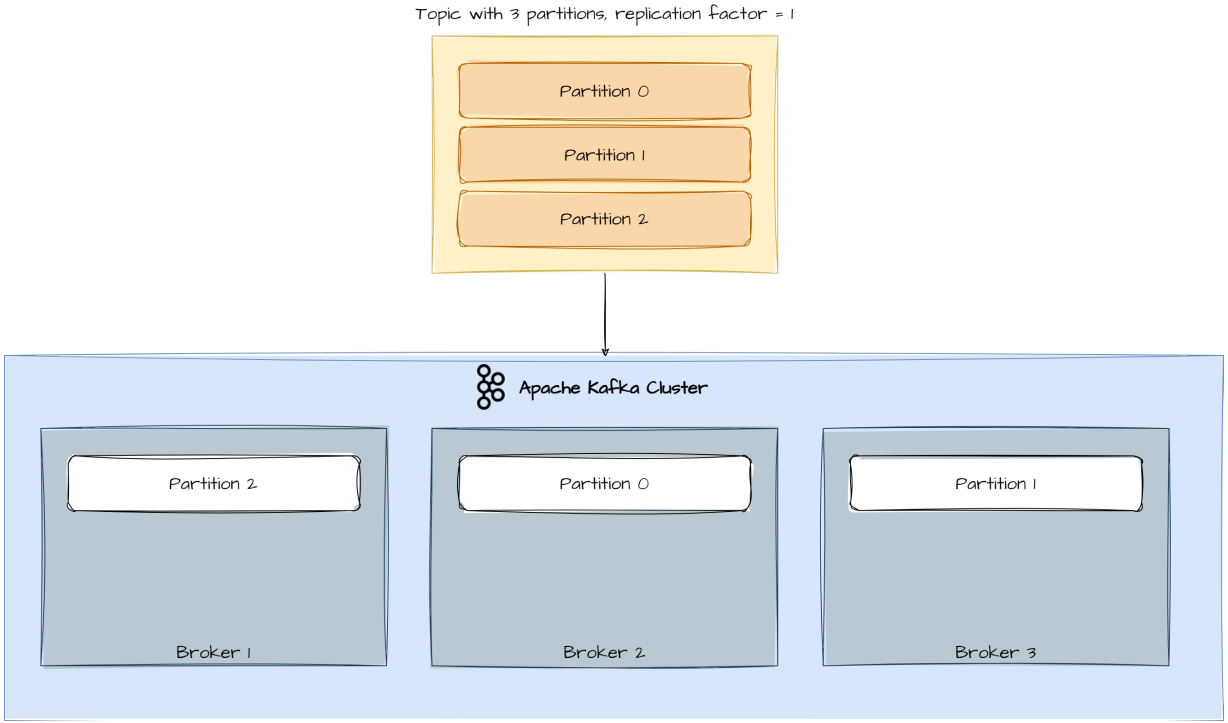
- Key-based partitioning can be used to ensure that all events that target a certain aggregate are written (and read) in-order into a topic-partition.

- Keys are hashed using a Murmur-based hashing algorithm (language agnostic).

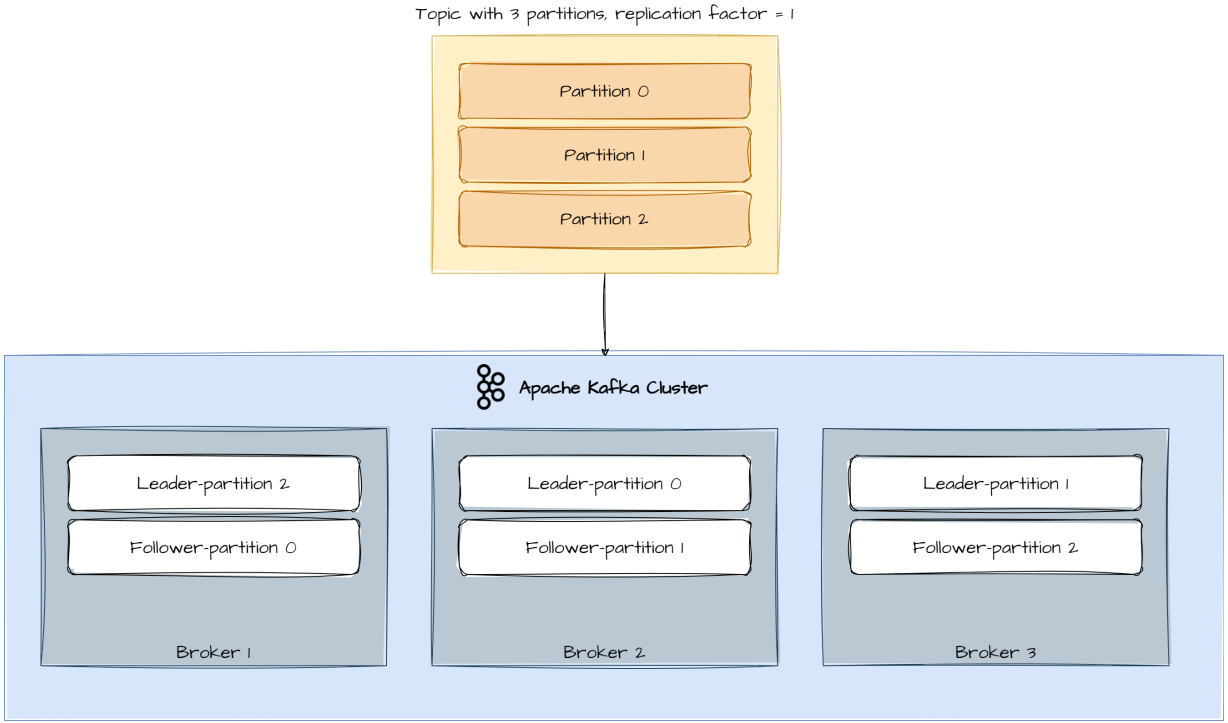
- It is possible to provide a custom partitioning strategy.

- It is possible to use the event time as timestamp. This requires the implementation of the `TimestampExtractor` interface. The event time is part of the message payload and thus must be extracted before it can be injected into the timestamp field.

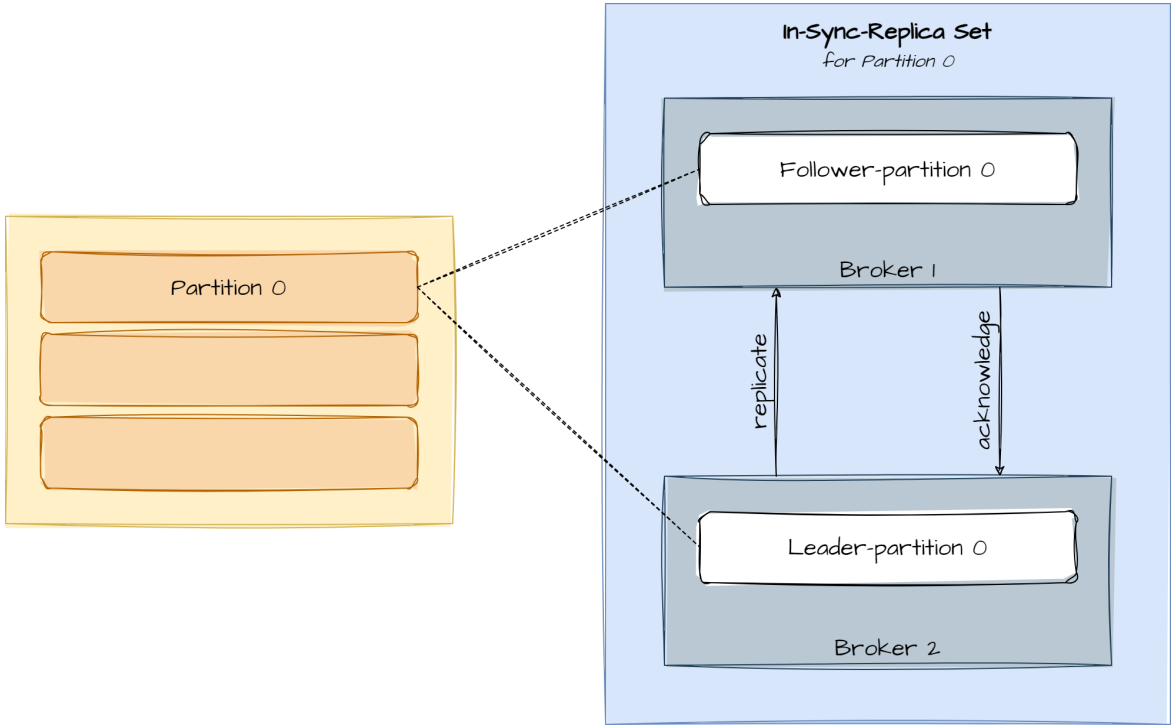
Partitions are spread across brokers and span multiple machines in a Kafka cluster.



Partitions are spread across brokers and span multiple machines in a Kafka cluster. (cont.)



The In-Sync-Replica set contains brokers that are either a leader or follower of a partition.



Let's talk briefly about Kafka versions that you encounter in the wild.

- In the wild you'll find a huge spread of Kafka versions
- Kafka Clients are usually (somewhat) compatible
- The configuration defaults are not
- Be **very** careful to use the same version in dev as in other stages

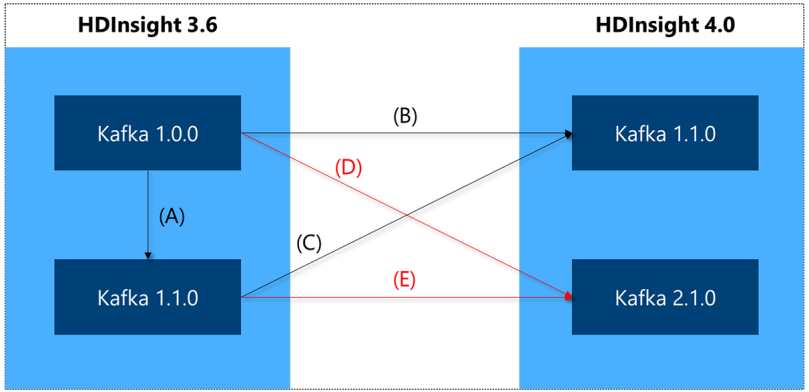
What's the situation on AWS?

Kafka version	MSK release date	End of support date
1.1.1	--	2024-06-05
2.1.0	--	2024-06-05
2.2.1	2019-07-31	2024-06-08
2.3.1	2019-12-19	2024-06-08
2.4.1	2020-04-02	2024-06-08
2.4.1.1	2020-09-09	2024-06-08
2.5.1	2020-09-30	2024-06-08
2.6.0	2020-10-21	2024-09-11
2.6.1	2021-01-19	2024-09-11
2.6.2	2021-04-29	2024-09-11
2.6.3	2021-12-21	2024-09-11
2.7.0	2020-12-29	2024-09-11
2.7.1	2021-05-25	2024-09-11
2.7.2	2021-12-21	2024-09-11
2.8.0	--	2024-09-11
2.8.1	2022-10-28	2024-09-11
2.8.2-tiered	2022-10-28	--
3.1.1	2022-06-22	2024-09-11
3.2.0	2022-06-22	2024-09-11
3.3.1	2022-10-26	2024-09-11
3.3.2	2023-03-02	2024-09-11
3.4.0	2023-05-04	--

What's the situation on Azure?

> **Note**
>
> Event Hubs for Kafka Ecosystems supports Apache Kafka version 1.0 and later.

Component	HDInsight 5.1	HDInsight 5.0
Apache Spark	3.3.1	3.1.3
Apache Hive	3.1.2	3.1.2
Apache Kafka	3.2.0	2.4.1
Apache Hadoop	3.3.4	3.1.1



Questions?

Apache Kafka

for Java Developers

Producing Records

Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

The Apache Kafka Clients SDK

A Producer's Perspective

Clients for Apache Kafka come in different sizes and shapes.

- Built-in CLI
- Java-based clients
  - Apache Kafka Client library
  - Apache Kafka for Spring
  - SmallRye Reactive Messaging with Kafka
- `librdkafka`-based clients
  - C, Python, ...

The official Apache Kafka Client library is only a single dependency away.

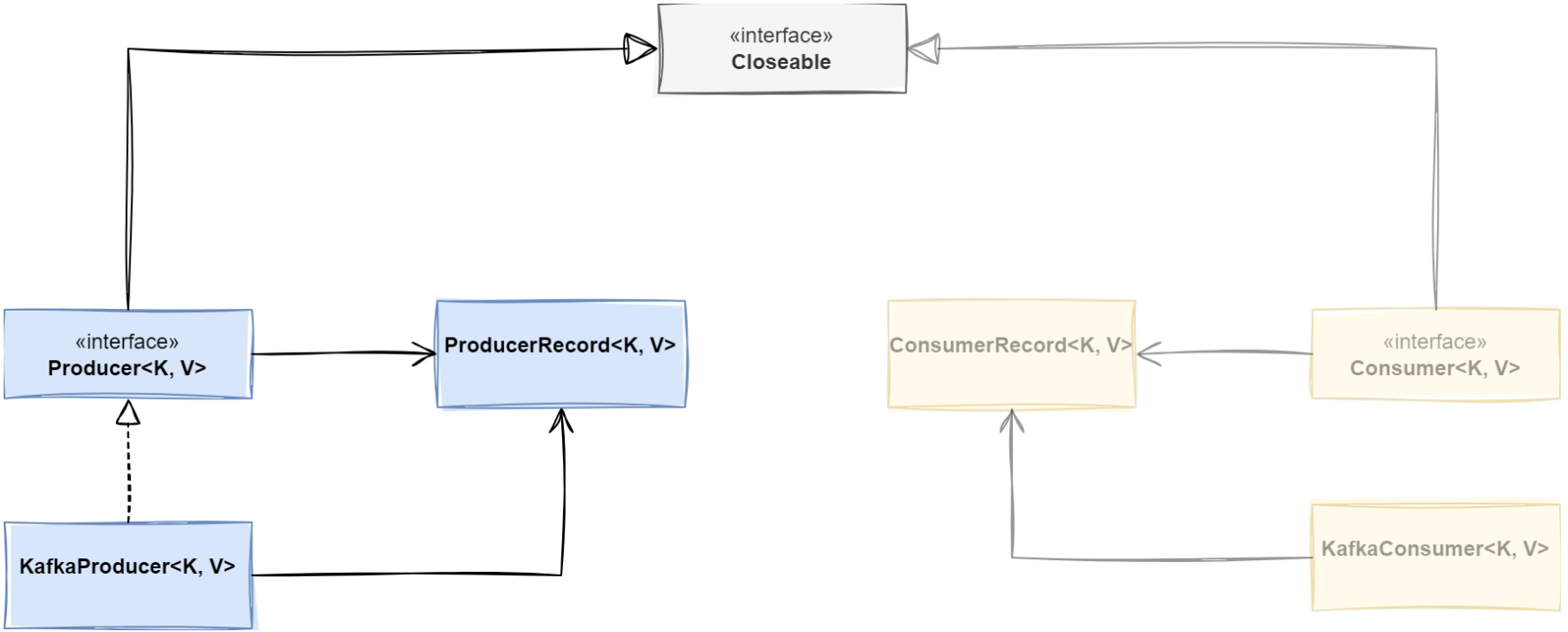
Maven

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</groupId>
  <version>3.6.1</version>
</dependency>
```

Gradle

```
dependencies {
  implementation "org.apache.kafka:kafka-clients:3.6.1"
}
```

We are able to write a basic producer by using just these few classes on the left.



- `Producer<K,V>`: This is the public interface of the producer client. It comprises the necessary methods for publishing records and using transactions.
- `KafkaProducer<K,V>`: This is the implementation of the `Producer` interface. Whereas the interface is barely documented, the implementation class offers rich documentation for the class itself as well as for every method of its public API.
- `ProducerRecord<K,V>`: This is a data structure that comprises all attributes of a Kafka record as perceived by a producer. To be more precise, this is the representation of a record **before** it has been published to a partition. It contains topic name, partition number, an optional set of headers, key, value, and a timestamp.

The parametric types `K` and `V` stand for the key resp. the value of of the record.

A Producer<K, V> has methods for publishing records and managing transactions.

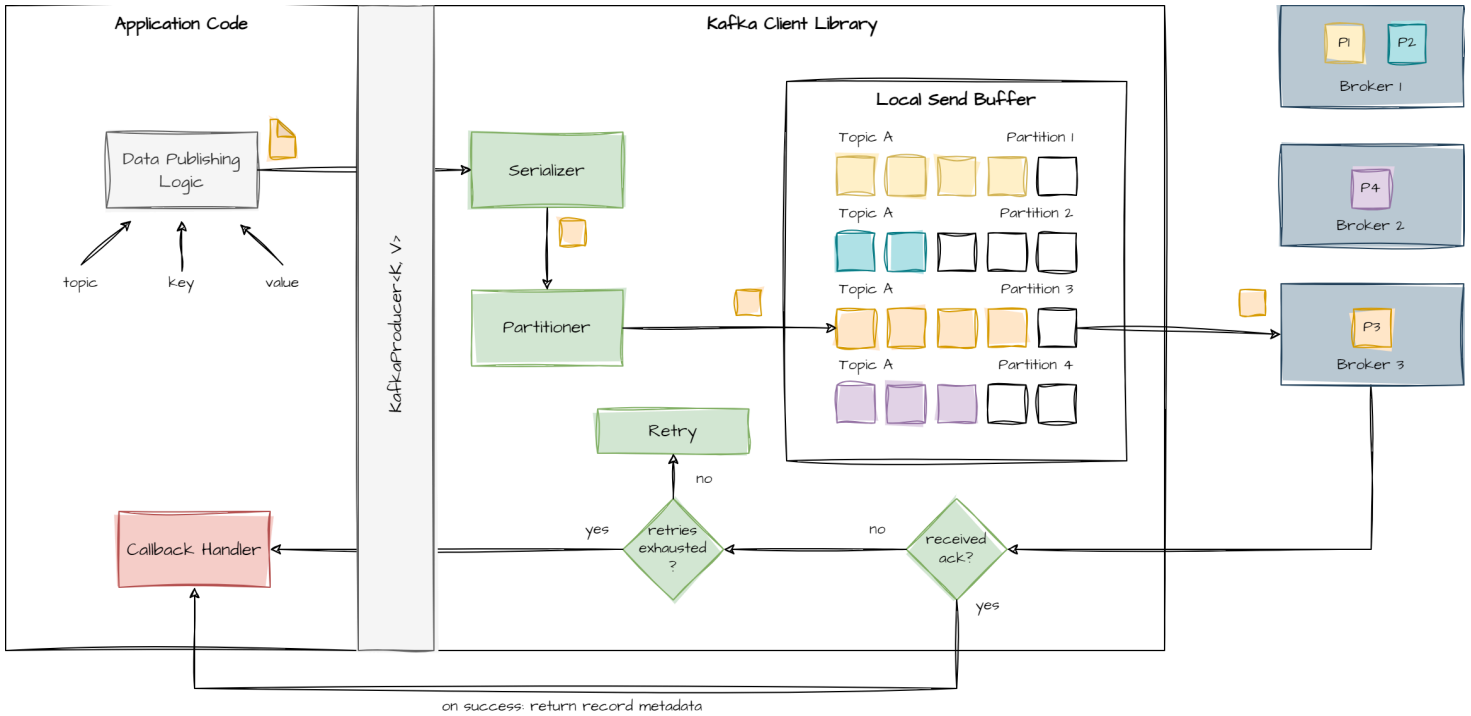
```
public interface Producer<K, V> extends Closeable {

  // methods for publishing records
  Future<RecordMetadata> send(ProducerRecord<K, V> record);

  Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback);

  // methods for managing transactional publishing
  void initTransactions();

  void beginTransaction() throws ProducerFencedException;

  void commitTransaction() throws ProducerFencedException;

  void abortTransaction() throws ProducerFencedException;

  // ... a couple of more methods omitted for brevity ...
}
```

The `send()` methods operate asynchronously, meaning they complete immediately after the record is serialized and placed into a local buffer. The process of actually transmitting the record occurs in the background, managed by a dedicated I/O thread. If the application needs to wait for the outcome, it should call the `get()` method on the `Future` object provided.

With this knowledge in mind, we are able to write a first, yet simple, producer!

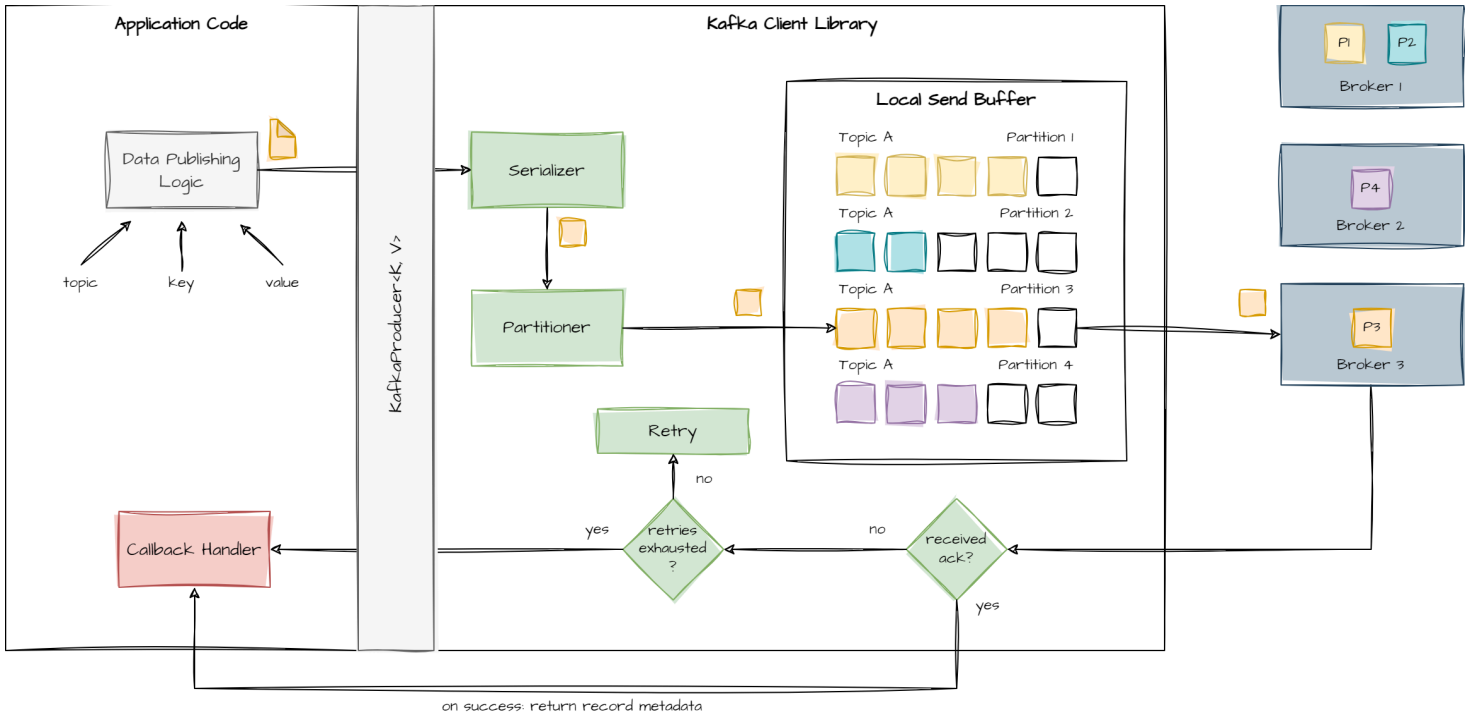
```
public class BasicProducer {
  public static void main(String[] args) throws Exception {
    var topic = "getting-started";
    Map<String, Object> config = Map.of(
      ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092",
      ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(),
      ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName()
    );
    try (var producer = new KafkaProducer<String, String>(config)) {
      var key = "my-key";
      var value = new Date().toString();
      Callback callback = (metadata, exception) -> {
        System.out.println("Published with metadata: %s, error: %s%n",
          metadata, exception);
      };
      producer.send(new ProducerRecord<>(topic, key, value), callback);
    }
  }
}
```

But what happens after we call producer.send?



Serialization

A `Serializer` encodes data of type T into `byte[]`.



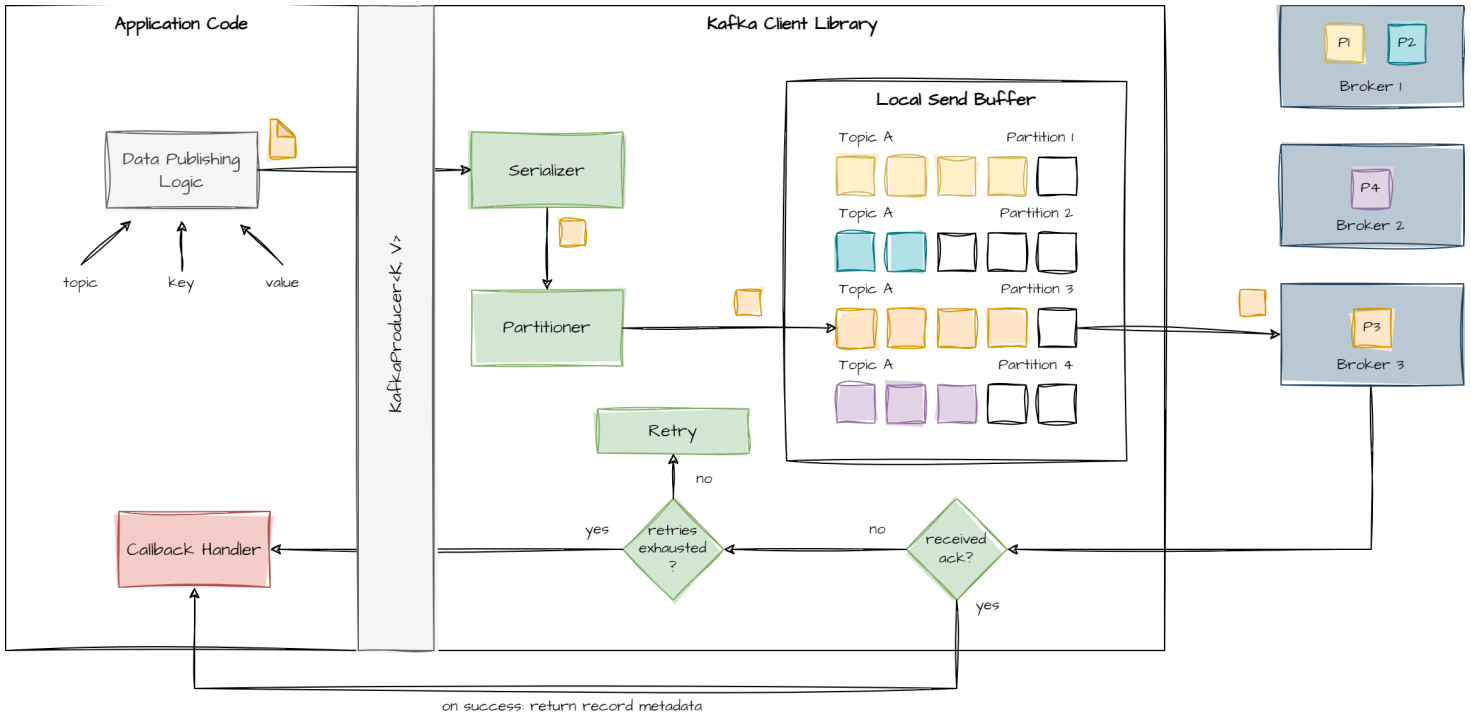
A `Serializer` encodes data of type T into `byte[]`. (cont.)

```
public interface Serializer<T> extends Closeable {

  byte[] serialize(String topic, T data);

  default byte[] serialize(String topic, Headers headers, T data) {
    return serialize(topic, data);
  }

  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

A `Serializer` encodes the given data into `byte[]`. This is the target format for Apache Kafka. Kafka does not interpret the payload of a record. It only deals with `byte[]`. Thus, serialization is a concern of the application.

Partition Assignment

After serialization, a partitioner sorts data into buffers for their resp. topic-partition.



Partition assignment factors in different strategies.

1. Use the partition of the `ProducerRecord<K,V>` (if assigned)
2. Use a custom `Partitioner` (if configured)
3. Try to calculate the partition based on record key (if not null)
    1. uses a hash function over the key
    2. default for Java-based clients is `murmur2`
    3. default for `librdkafka` is `crc32`
4. If there is no key or key should be ignored, use built-in partitioning logic

The `murmur2`-hash is calculated like this:

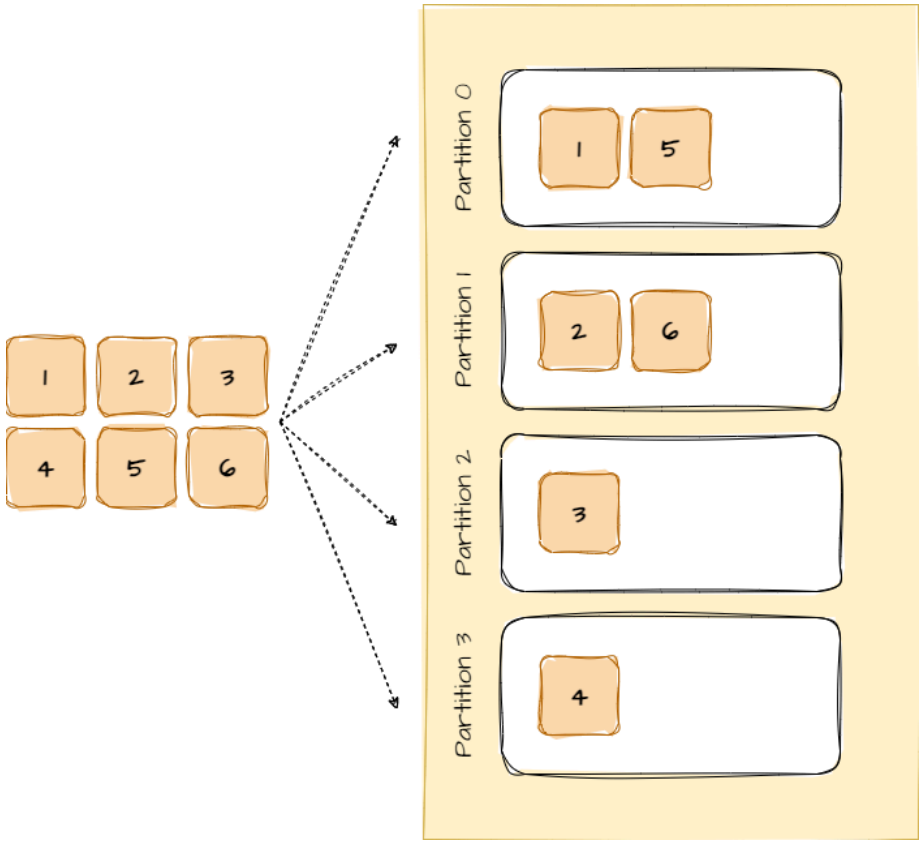
```
int targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
```

Moving to the **Sticky Partitioner** is not just about the fact that the previously used round-robin strategy led to an imbalanced workload between topic-partitions (cf. KAFKA-9965), but also results in a significant increase in performance (cf. KIP-480: Sticky Partitioner).

For Kafka 2.3 and below, the default partitioner uses a round-robin strategy.

Round Robin Partitioner

- one batch per partition
  - more batches
  - smaller batches
- leads to
  - more requests
  - higher latency
- bug: leads to uneven distribution (2.4+)

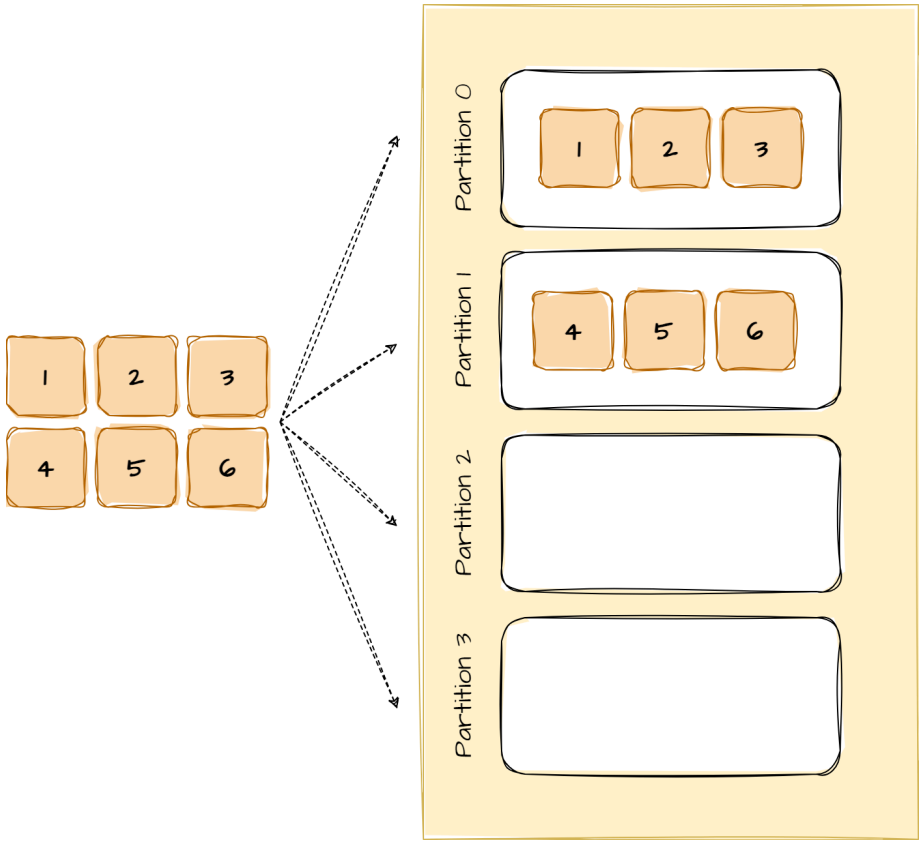


There is an bug that leads to an uneven distribution if the `RoundRobinPartitioner` is used in Kafka 2.4+. See https://issues.apache.org/jira/browse/KAFKA-9965 for details.

For Kafka 2.4. and up to Kafka 3.2, the default partitioner uses batch stickiness.

Sticky Partitioner

- stick to a partition
  - until batch is full
  - `linger.ms` has elapsed
- leads to
  - larger batches
  - reduced latency
- bug: uneven distribution (slow brokers)



This will lead to larger batches and reduced latency (due to larger requests, `batch.size` is more likely to be reached). Over time, the records are still spread *almost evenly* across partitions, so the balance of the cluster is not affected.

For Kafka 3.3 and higher, there are two new strategies available.

Uniform Sticky Batch Size

- don't switch unless `batch.size` bytes got produced to partition
- uniform throughput and data distribution
  - adapts well to higher latency brokers
- might slow down producer rate due to filling local buffer if brokers are constantly lagging behind

For Kafka 3.3 and higher, there are two new strategies available.

Uniform Sticky Batch Size

- don't switch unless `batch.size` bytes got produced to partition
- uniform throughput and data distribution
  - adapts well to higher latency brokers
- might slow down producer rate due to filling local buffer if brokers are constantly lagging behind

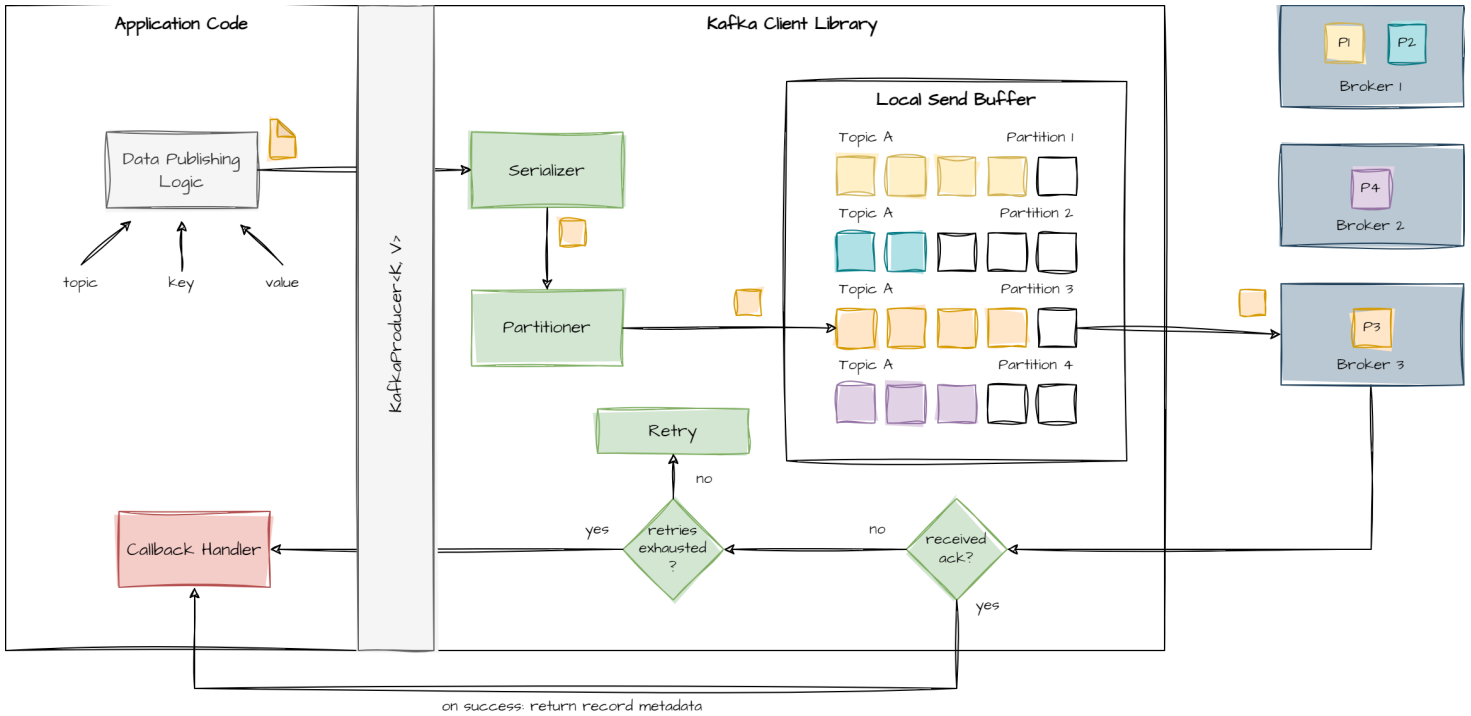
Adaptive Partition Switching (default)

- adapts to broker load
  - queue size of unsent batches is indicator
  - probability of choosing a partition is proportional to the inverse queue size
  - partitions with longer queues are less likely to be chosen
- `partitioner.availability.timeout.ms`
  > 0 to indicate a failed batch if the producer is unable to produce data within timeout

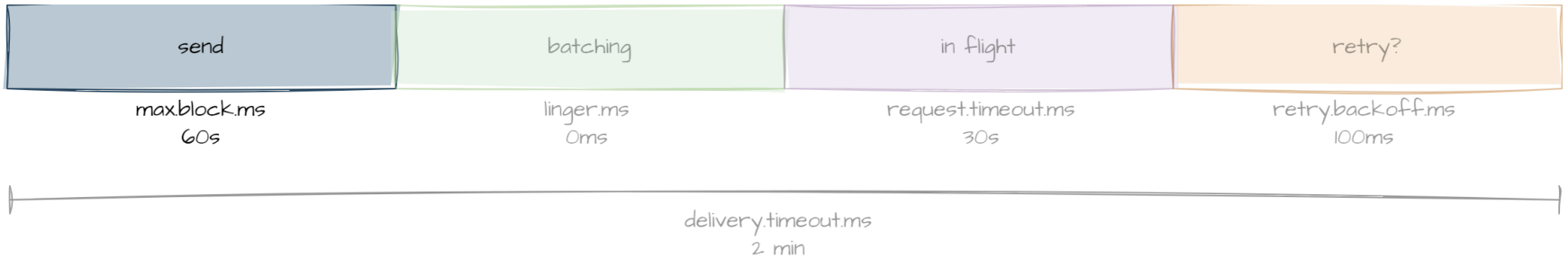
Set `partitioner.adaptive.partitioning.enable` to `false` to use **Uniform Sticky Batch Size**. If set to `true` (default), **Adaptive Partition Switching** is used.

Timeouts

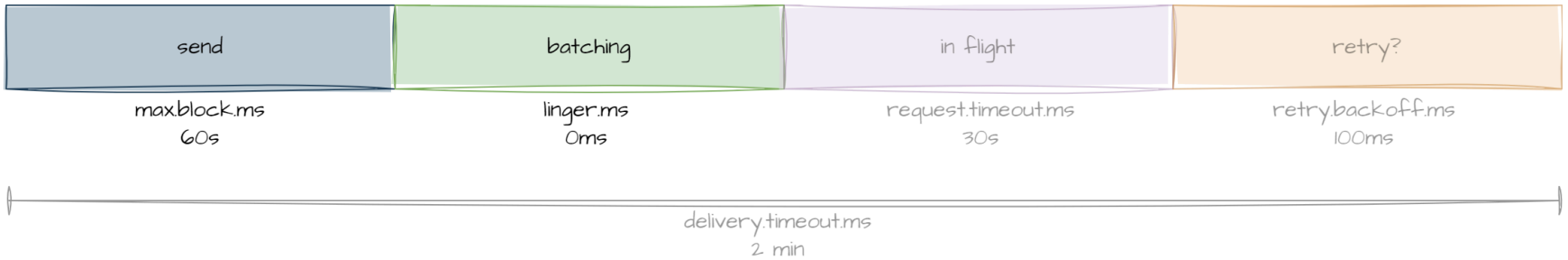
After partitioning, data is moved into a local buffer for the target topic-partition.



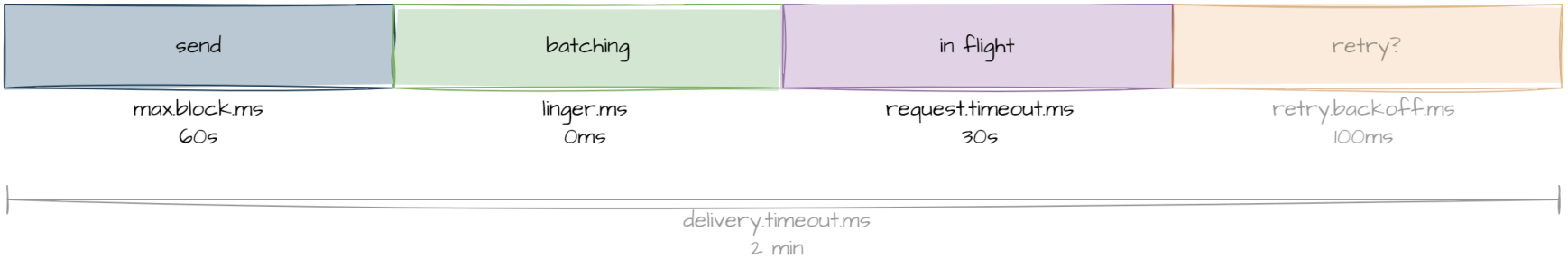
The send method of the producer waits a maximum of `max.block.ms`.



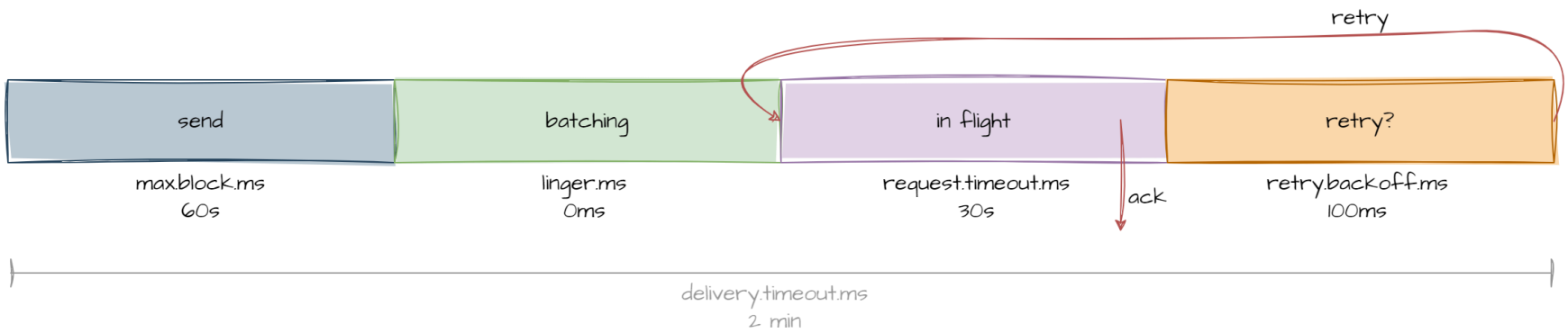
The producer waits a maximum of `linger.ms` for messages to include into a batch.



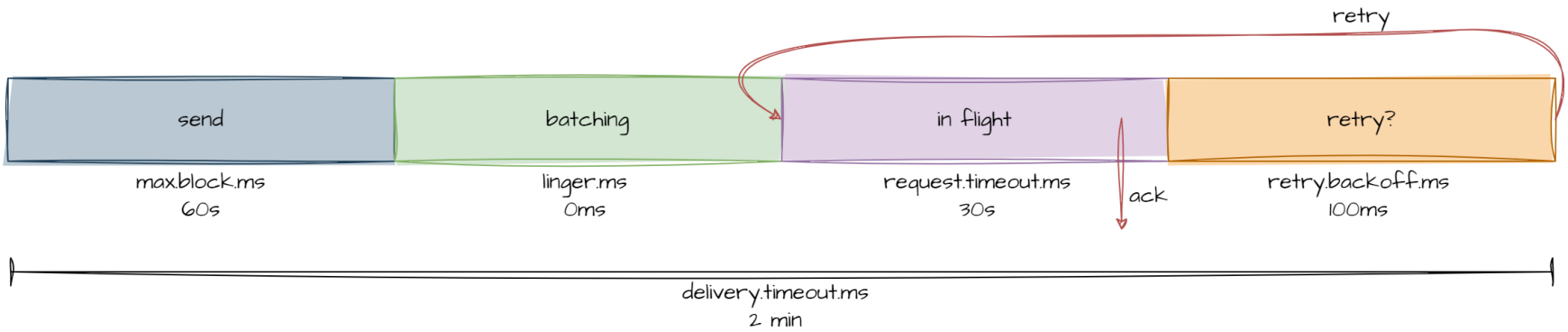
The producer waits for `request.timeout.ms` for an acknowledgement.



If the producer receives no ack within a certain time, its retry mechanism fires.



After a maximum of `delivery.timeout.ms` the attempt to publish will be aborted.



Acknowledgements

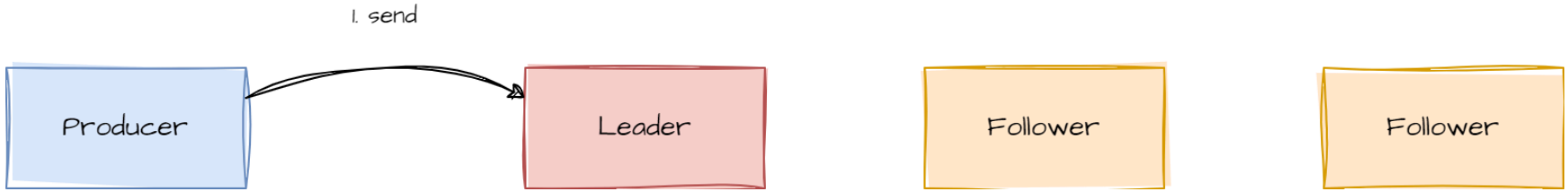
Depending on your `acks` setting, the producer waits for an acknowledgment (or not).

General

- Producer is able to submit data anew in case of missing ack
- If the error is persistent, the producer will generate an exception

> ***Acknowledgements*** *are not only used to signal that data has been received, but also play a part in Kafka's* ***replication strategy****.*

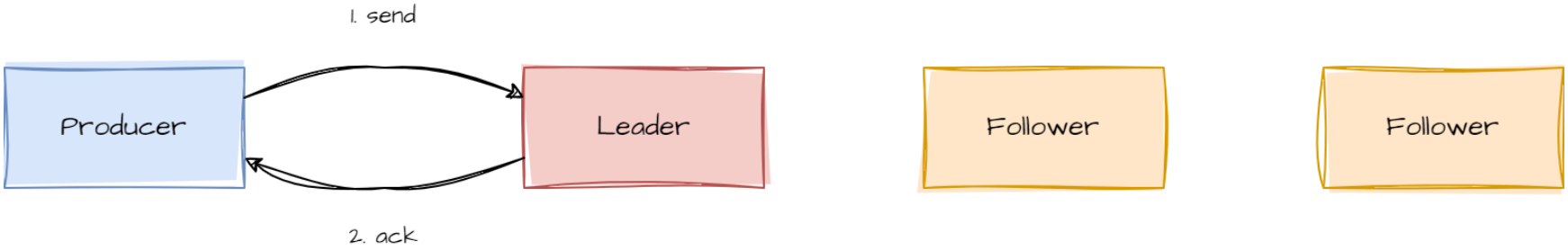
The producer does not wait for an acknowledgment if `acks=0`.



Traits

- Fire-and-forget
- Networking analogy: UDP
- Best performance, if data loss is tolerable

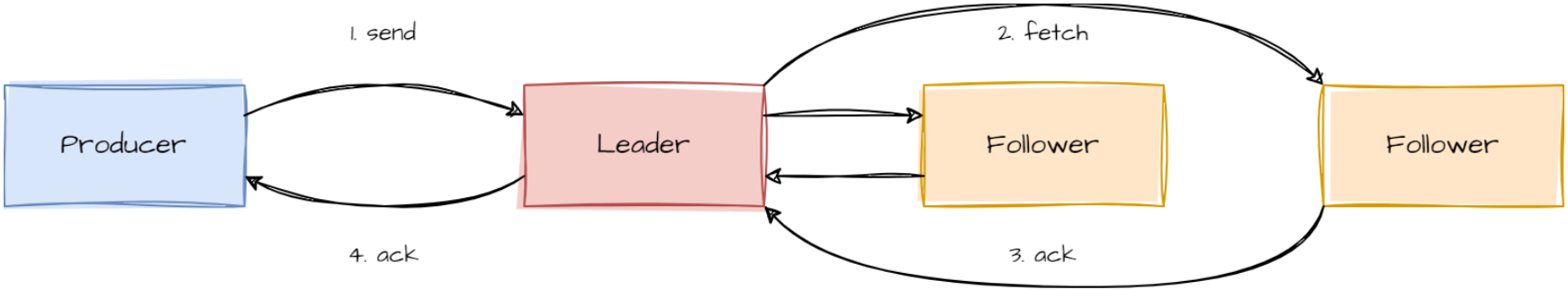
The leader acknowledges directly after receiving the record if `acks=1`.



Traits

- Does not wait for the replication result to followers
- Networking analogy: TCP
- Default configuration up until Apache Kafka 3.0

The leader replicates to all followers before it sends the acknowledgment if `acks=all`.



Traits

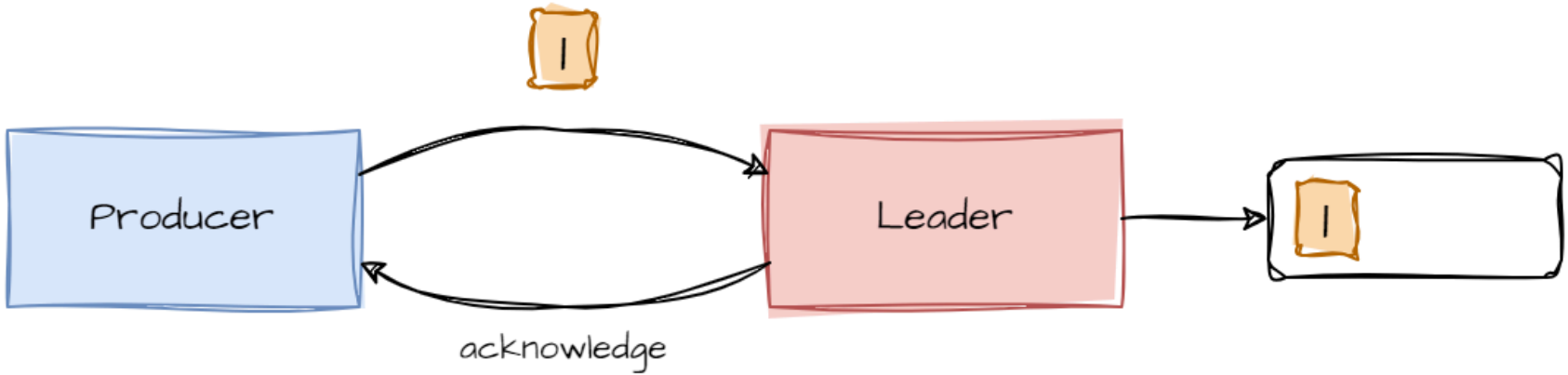
- Replicas are considered in-sync if they received latest data within 30 s
- Best consistency guarantees
- `min.insync.replicas` controls how many brokers must be in-sync

- Best consistency guarantees, but for the lack of a higher throughput

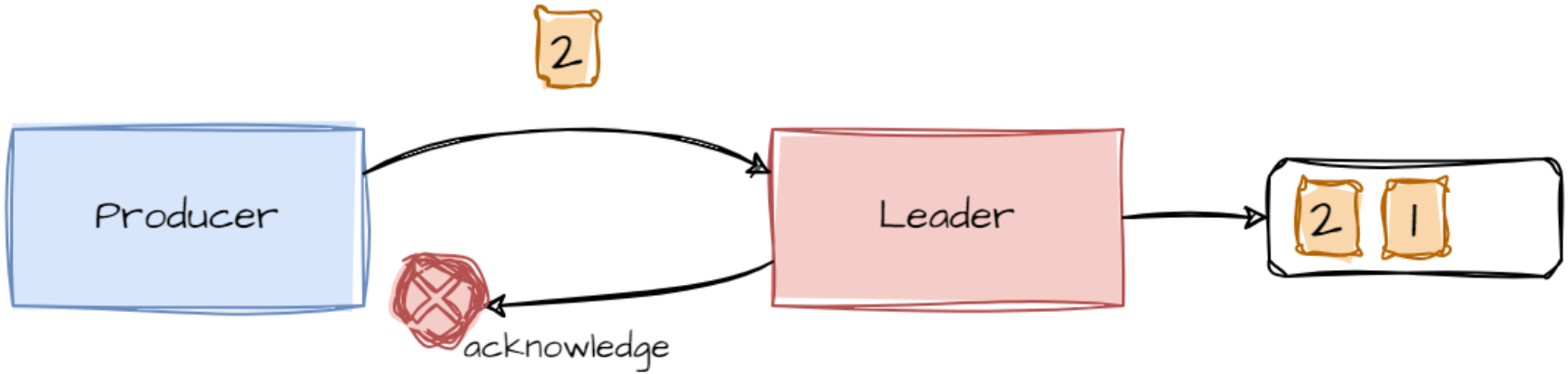
- If there are not enough `min.insync.replicas`, the broker sends an error back to the producer, which will raise a `NotEnoughReplicasException`

Delivery Guarantees

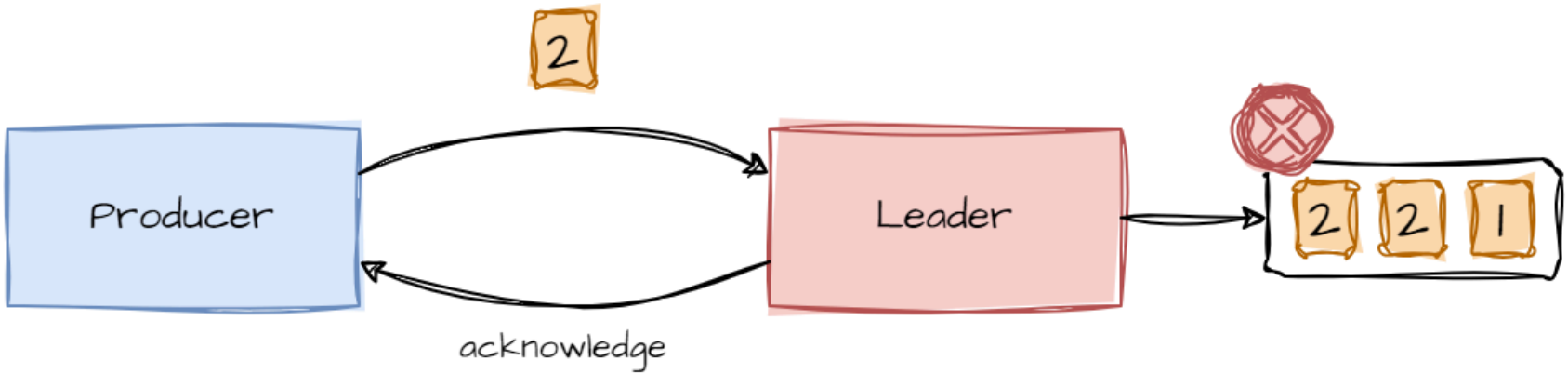
Lost acknowledgments may lead to duplicated records in the log.



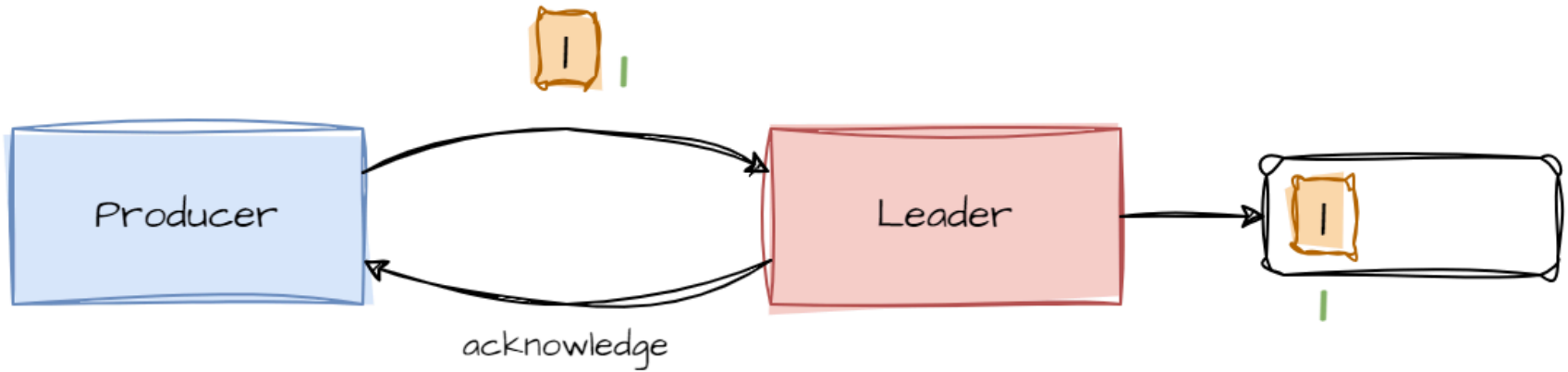
Lost acknowledgments may lead to duplicated records in the log. (cont.)



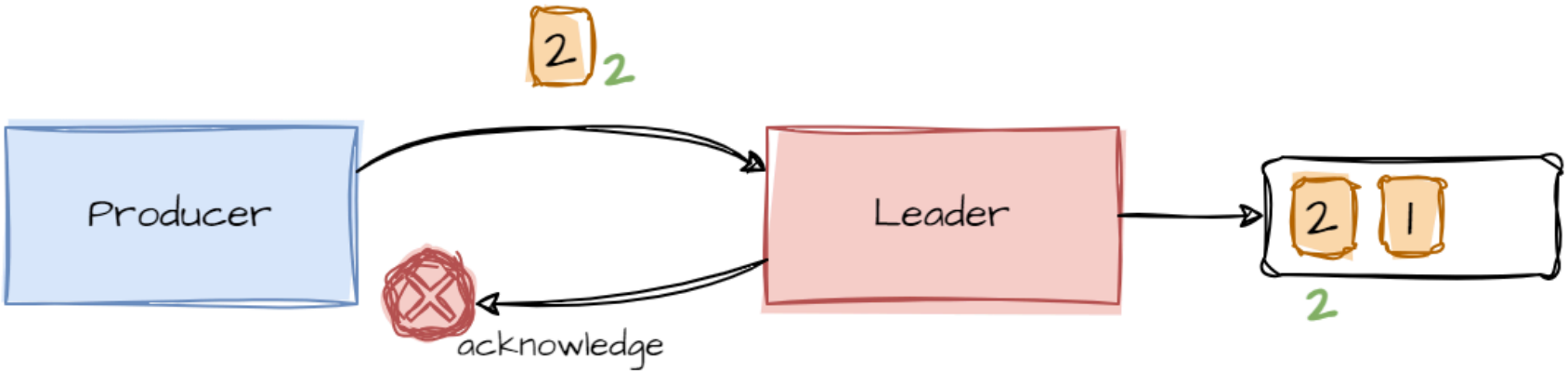
Lost acknowledgments may lead to duplicated records in the log. (cont.)



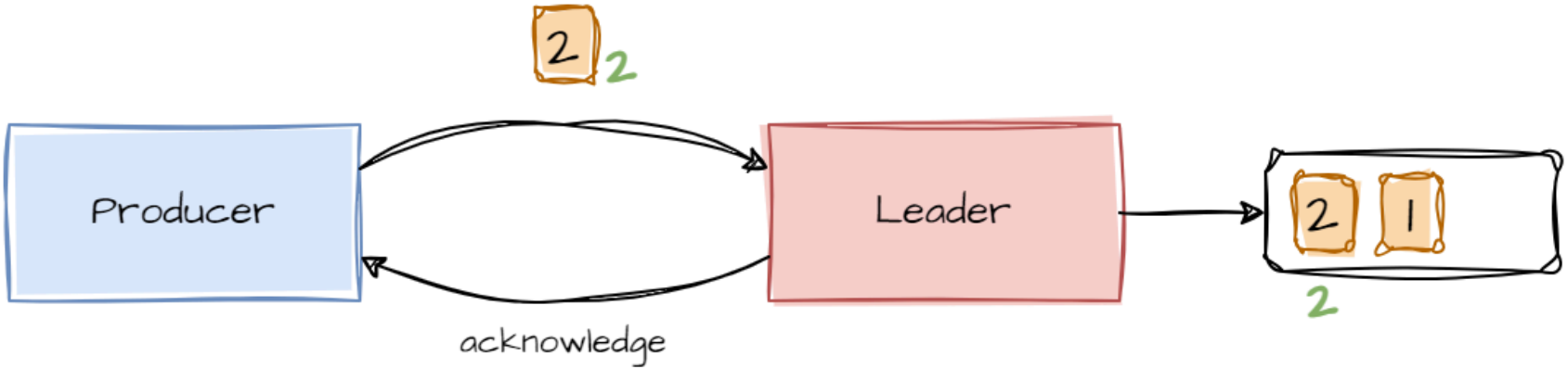
Setting `acks=all` and `enable.idempotence=true` prevents duplicates.



Setting `acks=all` and `enable.idempotence=true` prevents duplicates. (cont.)



Setting `acks=all` and `enable.idempotence=true` prevents duplicates. (cont.)



We distinguish between three different delivery guarantees.

At-most-once

> *Guarantees that a record will be delivered **at most one time**. There can be **no duplicates**. There is **no guarantee** that the record will be received from the broker.*

- This is the case for `acks=0`

We distinguish between three different delivery guarantees. (cont.)

At-least-once

> *Guarantees that a record **will be received** by the broker, but **possibly multiple times**. Hence, there **may be duplicates**.*

- This the case for
  - `acks=all`
  - `min.insync.replicas` to a sensible value

We distinguish between three different delivery guarantees. (cont.)

Exactly-once

> *Guarantees that a record **will be written to the log** exactly **one time** (idempotency).*

- This is the case for
  - `acks=all`
  - `enable.idempotence=true`
- Default setting since Apache Kafka 3.0

Summary

What did we learn?

- Client SDK Essentials
- Partition Assignment
- Timeouts
- Retries
- Error Handling
- Acknowledgements
- Delivery Guarantees

Summary

What did we learn?

- Client SDK Essentials
- Partition Assignment
- Timeouts
- Retries
- Error Handling
- Acknowledgements
- Delivery Guarantees

What's to follow?

- *Serialization*
- Interceptors
- Broker Internals
- Producer Designs
- Transactions

Questions?

Apache Kafka

for Java Developers

Consuming Records

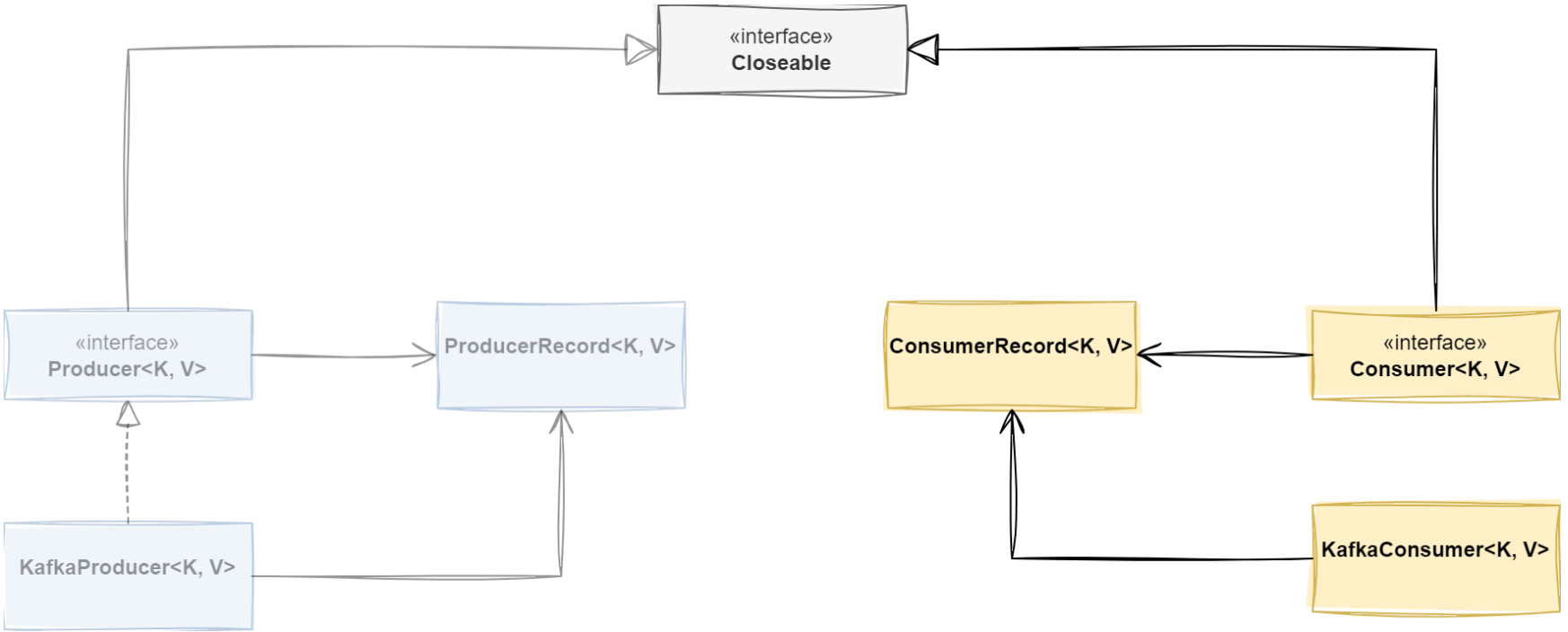
Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

The Apache Kafka Clients SDK

A Consumer's Perspective

We are able to write a basic consumer by using just these few classes on the right.



- `Consumer<K,V>`: This is the public interface for a consuming client. It contains message signatures for controlling subscriptions and topic/partition assignment, fetching records, committing offsets, ...
- `KafkaConsumer<K,V>`: This is the implementation of the `Consumer` interface. Like its counterpart, it features rich documentation of its public API.
- `ConsumerRecord<K,V>`: This is a data structure that comprises all attributes of a Kafka record as perceived by a consumer. It is essentially a superset of the `ProducerRecord<K,V>`, as it contains additional metadata such as the record offset, a checksum, and some other attributes.

The parametric types `K` and `V` stand for the key resp. the value of of the record.

A Consumer<K,V> offers methods for managing subscriptions and consuming records.

```
public interface Consumer<K, V> extends Closeable {
  // subscription management
  void subscribe(Collection<String> topics);
  void subscribe(Collection<String> topics, ConsumerRebalanceListener callback);
  void subscribe(Pattern pattern);
  void subscribe(Pattern pattern, ConsumerRebalanceListener callback);
  void unsubscribe();
  void assign(Collection<TopicPartition> partitions);
  Set<TopicPartition> assignment();
  Set<String> subscription();

  // record consumption
  ConsumerRecords<K, V> poll(long timeout);
  ConsumerRecords<K, V> poll(Duration timeout);

  // offset committing (not showing overloaded methods)
  void commitSync();
  void commitAsync();

  // manage consumption order (not showing overloaded methods)
  void seek(TopicPartition partition, long offset);
  void seekToBeginning(Collection<TopicPartition> partitions);
  void seekToEnd(Collection<TopicPartition> partitions);
  // ... and a lot more ...
}
```

With this knowledge in mind, we are able to write a first, yet simple, consumer!

```
public class BasicConsumer {
  public static void main(String[] args) {
    var topic = "getting-started";
    Map<String, Object> config = Map.of(
      ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092",
      ConsumerConfig.KEY_DESERIALIZER_CONFIG, StringDeserializer.class.getName(),
      ConsumerConfig.VALUE_DESERIALIZER_CONFIG, StringDeserializer.class.getName(),
      ConsumerConfig.GROUP_ID_CONFIG, "basic-consumer-group",
      ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest",
      ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
    try (var consumer = new KafkaConsumer<String, String>(config)) {
      consumer.subscribe(Set.of(topic));
      while (true) {
        var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
          System.out.println("Received record with value %s%n", record.value());
        }
        consumer.commitAsync();
      }
    }
  }
}
```

Subscribing and Consuming

A `KafkaConsumer<K,V>` offers two ways for subscribing to the topics.

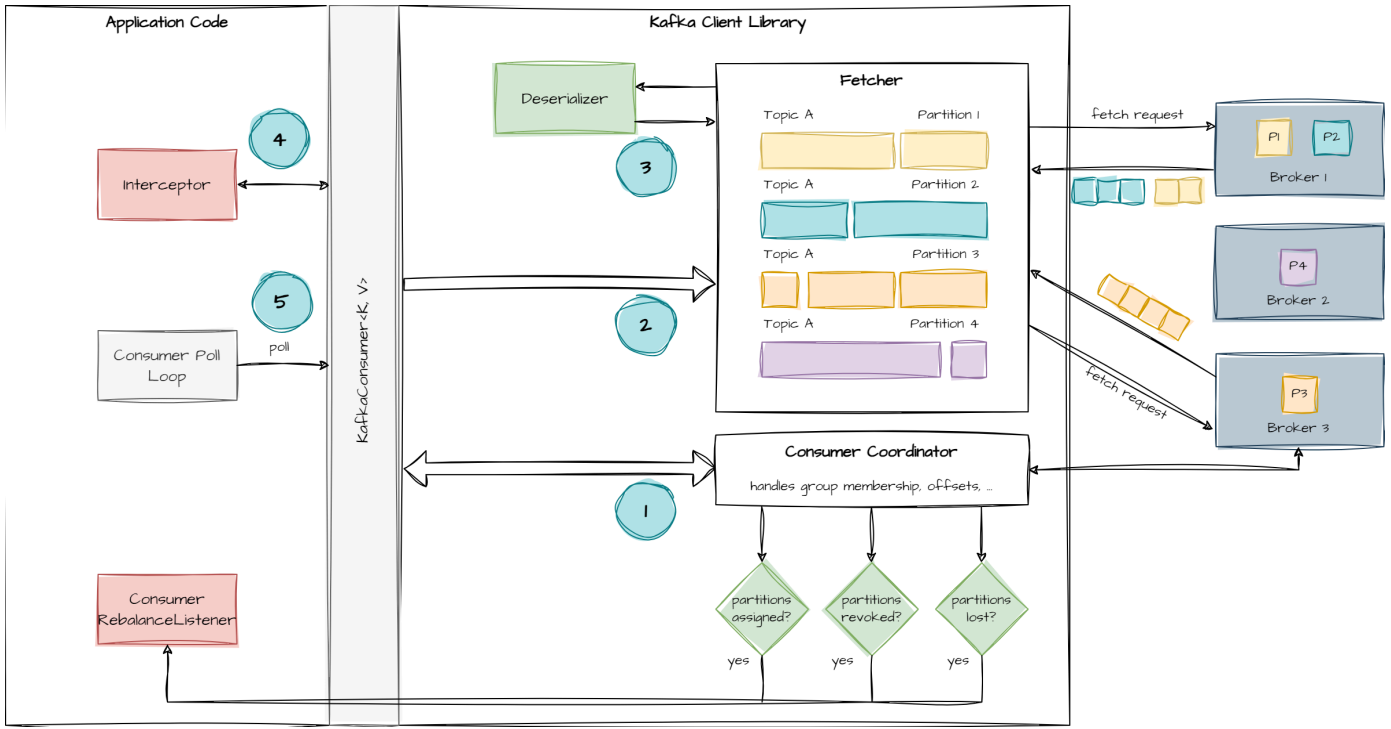
via `subscribe`

- subscribe to list of topics (regexp possible)
- group membership with failure detection
  - client-side
  - server-side
- dynamic partition assignment
- automatic or manual offsets management
- single consumer per partition

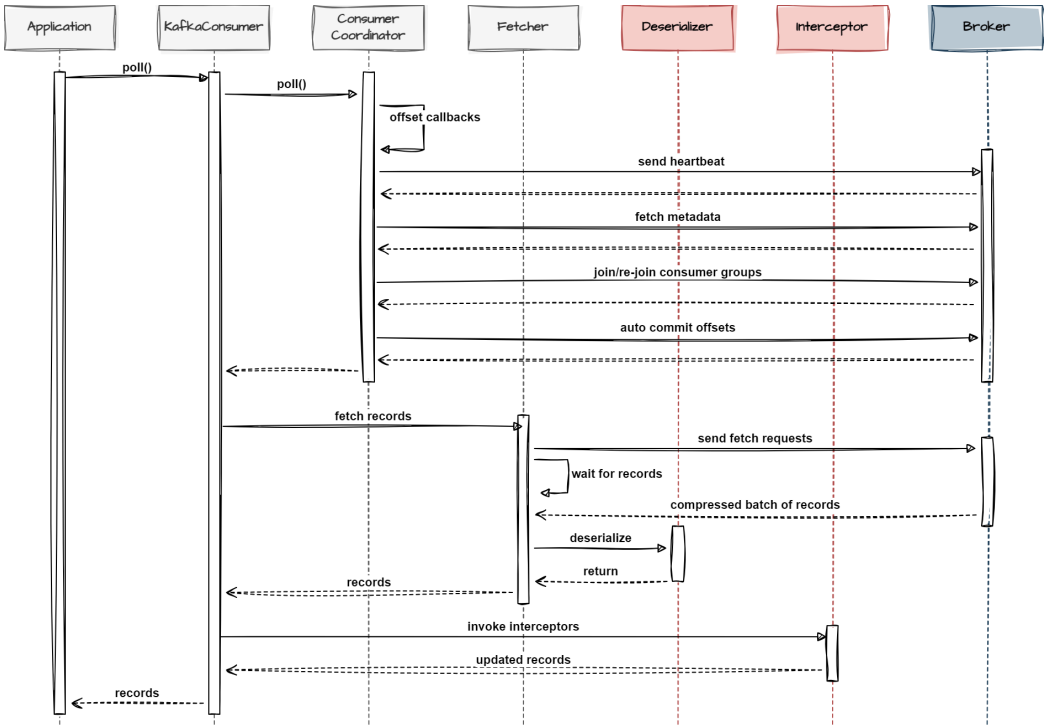
via `assign`

- finer control with topic-partition subscription
- automatic or manual offsets management
- supports multiple consumers per partition

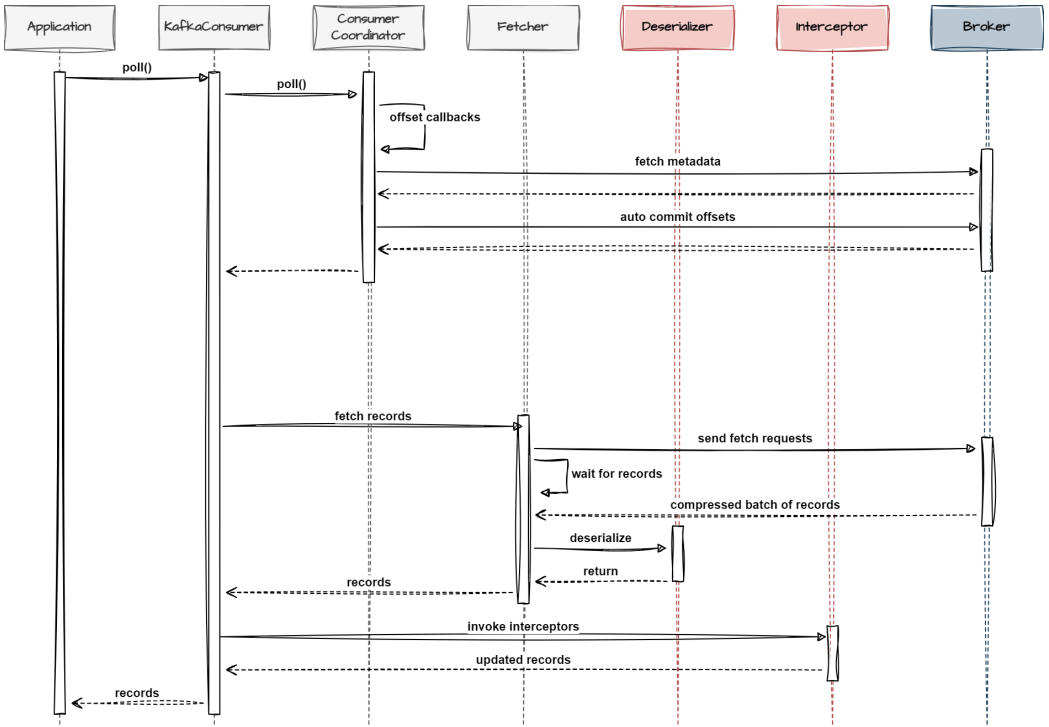
But what happens after subscribing and entering the poll-loop?



With `subscribe`, the consumer won't read records until it joins the consumer group.



With `assign`, the consumer won't invoke group membership functionality.



Offset Management

Partition offsets are part of the local consumer state, unless shared with the Kafka cluster.

- Use consumer offsets as **resumption point**
- Upon re-assigning a consumer, we need to know where to continue
- Skip over any records that have already been processed

Persisting the consumer state to the Kafka cluster is called committing an offset.

- Commit `offset of last record + 1`
  1. New consumer joins the consumer group and takes over.
  2. Starts off at the offset of the **last record** that has not been read
  3. This is `offset of last record + 1`

- Kafka employs a **recursive strategy** when managing offsets
  - Utilizes itself to persist and track offsets
  - cf. topic `__consumer_offsets`

The contents of the `__consumer_offsets` are compacted regularly to reduce the data progressively to only the last known committed offsets for every combination of `(consumer group, topic-partition)`.

Controlling when an offset is committed provides flexibility wrt. delivery guarantees.

- Move between *at-most-once* to *at-least-once* simply
  - by committing offsets **before** record processing (at-most-once)
  - by committing offsets **after** record processing completes (at-least-once)

> *A committed offset implies that the record **one below that offset** and **all prior records** have been processed by the consumer.*

- Last offset of a batch of records acknowledges the whole batch
- Built your error handling strategy around that fact

By default, a Kafka consumer will automatically commit offsets every five seconds.

- Adjusting frequency is done by setting `auto.commit.interval.ms`
- Setting `enable.auto.commit` to `false` disables this behavior
- API offers **synchronous** and **asynchronous** commit operations

```
public interface Consumer<K,V> {
  // only showing commit* methods
  // synchronous commits
  void commitSync();
  void commitSync(Duration timeout);
  void commitSync(Map<TopicPartition, OffsetAndMetadata> offsets);
  void commitSync(Map<TopicPartition, OffsetAndMetadata> offsets, Duration timeout);
  // asynchronous commits
  void commitAsync();
  void commitAsync(OffsetCommitCallback callback);
  void commitAsync(Map<TopicPartition, OffsetAndMetadata> offsets, OffsetCommitCallback callback);
}
```

- Setting `enable.auto.commit` to `false` disables the auto-commit-behavior. This means that you have to call one of the `commit*`-methods yourself. Make sure that this happens somewhere as part of your poll-and-process-records-loop.

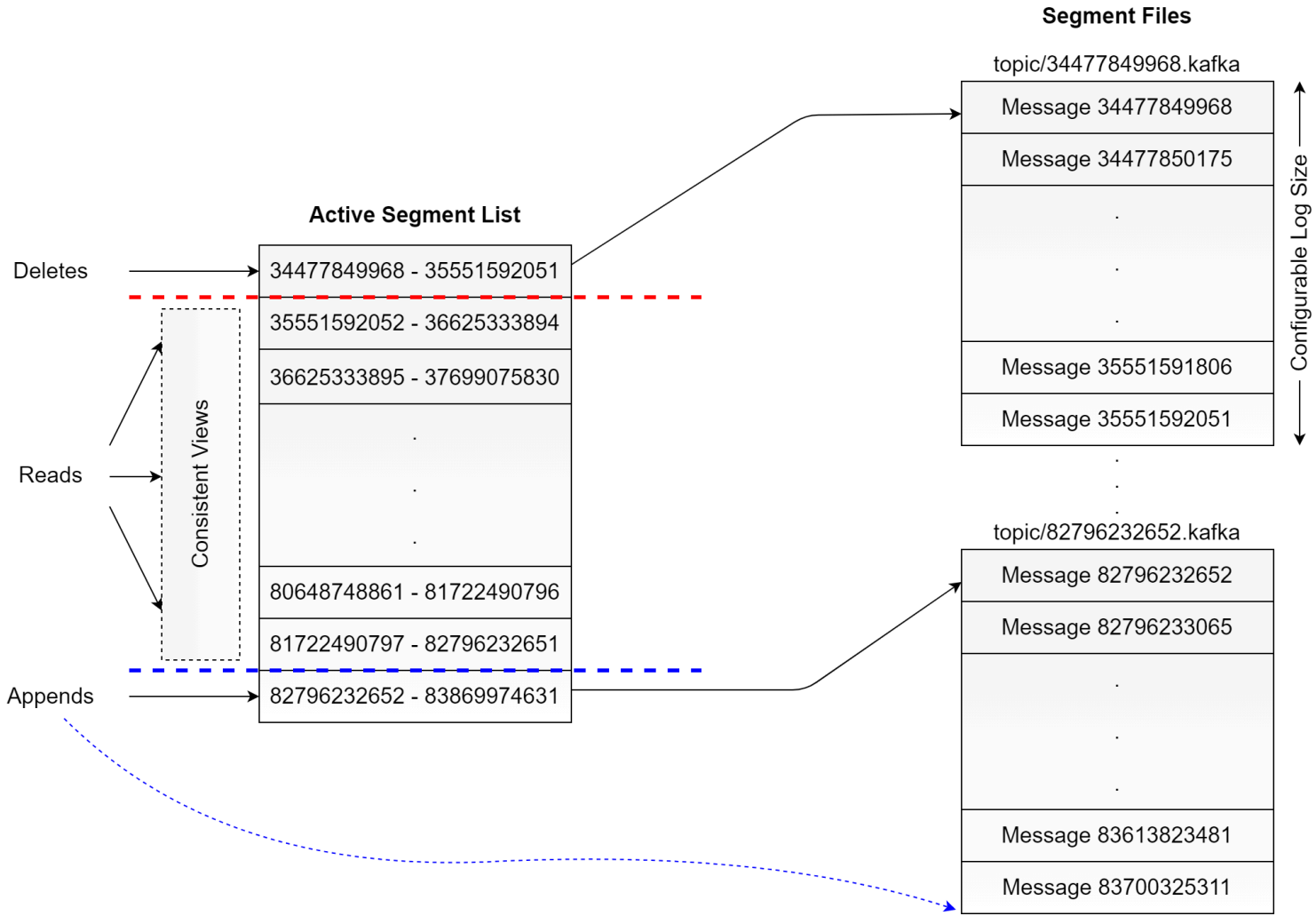
In case of no persisted offsets, `auto.offset.reset` controls where the consumer starts.

- Offers three different options
  - `earliest`: reset offset to the earliest offset (low watermark)
  - `latest`: reset offset to latest offset (high watermark)
  - none: throw exception if there are no offsets present

- `earliest`: Reset offset to the earliest offset. Consume from the beginning of the topic partition. This is the earliest offset of the *consistent view* onto a topic-partition and represents the **low watermark** (LWM) of the topic-partition.

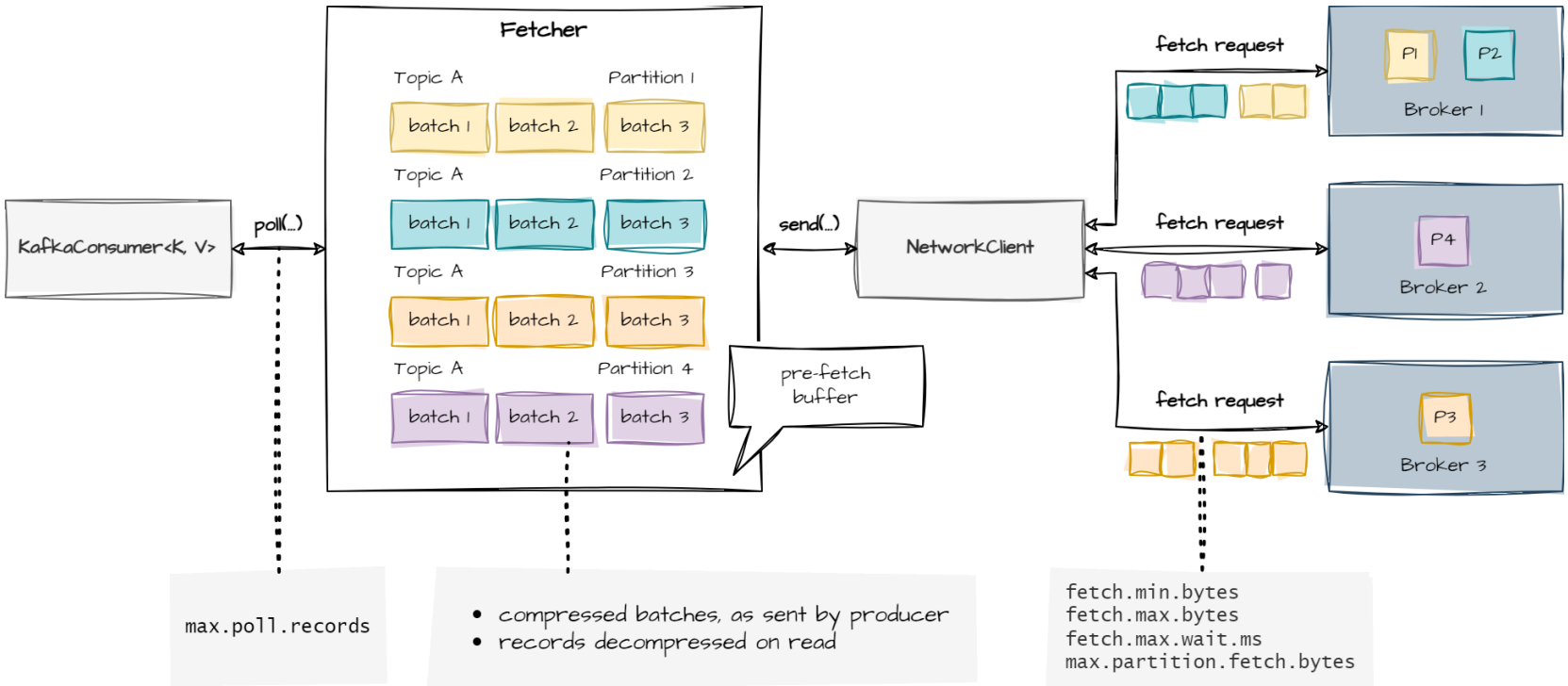
- `latest` (default): Reset offset to the latest offset. Consume from the end of the topic. This is the latest offset of the *consistent view* onto a topic-partition and represents the **high watermark** (HWM) of the topic-partition.

- `none`: Throw an exception if no offset is present for the consumer group.


Segment Files
topic/34477849968.kafka
Message 34477849968
Message 34477850175
Message 35551591806
Message 35551592051
Configurable Log Size
topic/82796232652.kafka
Message 82796232652
Message 82796233065
Message 83613823481
Message 83700325311
Active Segment List
Deletes
Reads
Appends
Consistent Views
34477849968 - 35551592051
35551592052 - 36625333894
36625333895 - 37699075830
80648748861 - 81722490796
81722490797 - 82796232651
82796232652 - 83869974631

Fetcher

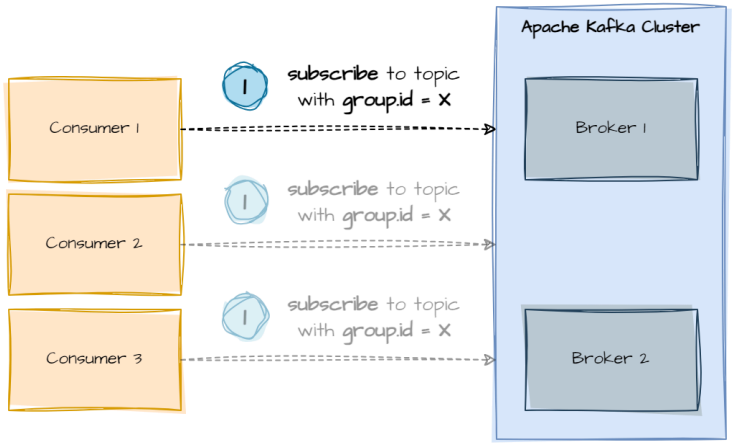
The Kafka consumer uses `Fetcher` as a buffer that retrieves batches from brokers.



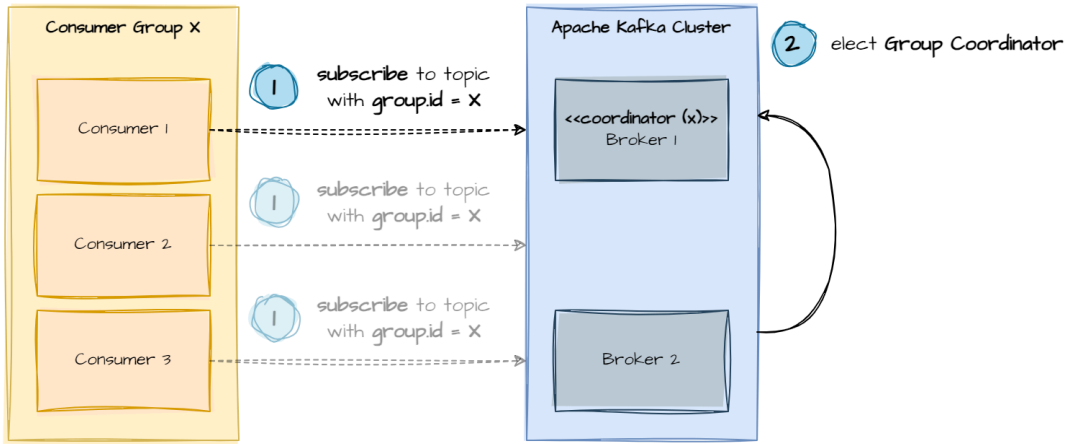
- `Fetcher` keeps compressed batches in-memory (as they are sent by the producer)
- `Fetcher` decompresses records on `poll()`
- Once a batch is completely consumed, it will be discarded from the buffer
- Key configurations:
  - `fetch.min.bytes`
  - `fetch.max.wait.ms`
- Increasing fetch size and wait time optimizes for **increased throughput**.
- Decreasing fetch size and wait time optimizes for **reduced latency**.

Consumer Groups

Consumers with the same `group.id` form a consumer group to cooperate.



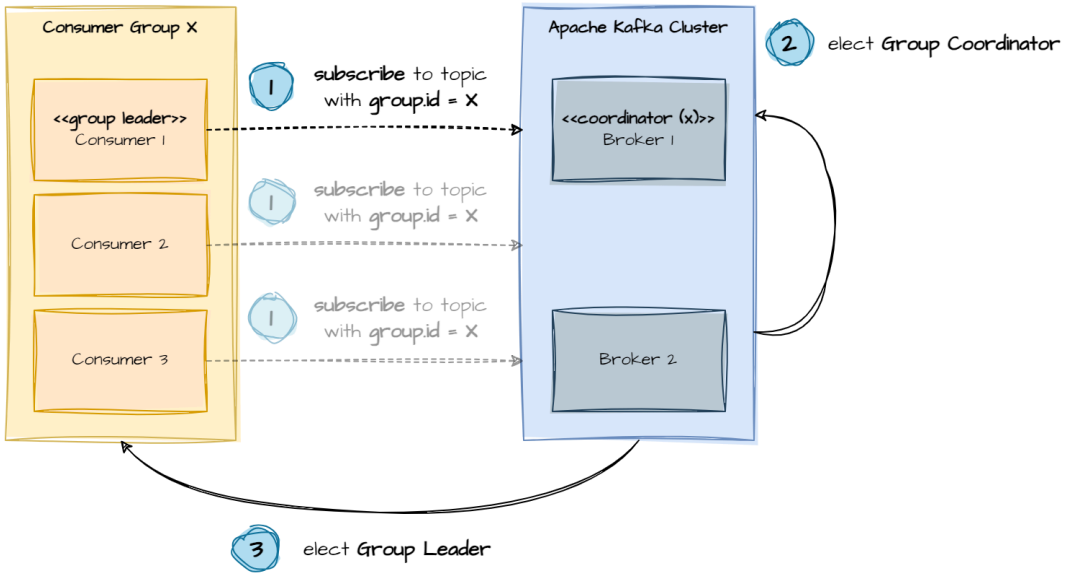
The Kafka cluster elects one of the brokers as *Group Coordinator*.



The *Group Coordinator* is responsible for

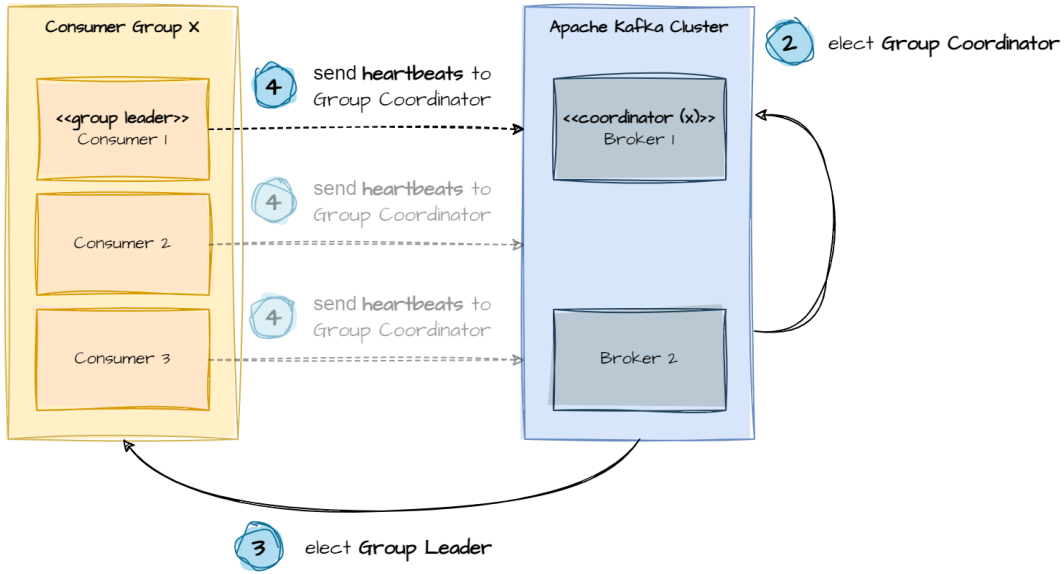
- managing group membership
- receiving heartbeats
- triggering rebalances
- ...

The *Group Coordinator* elects one consumer as *Group Leader*.

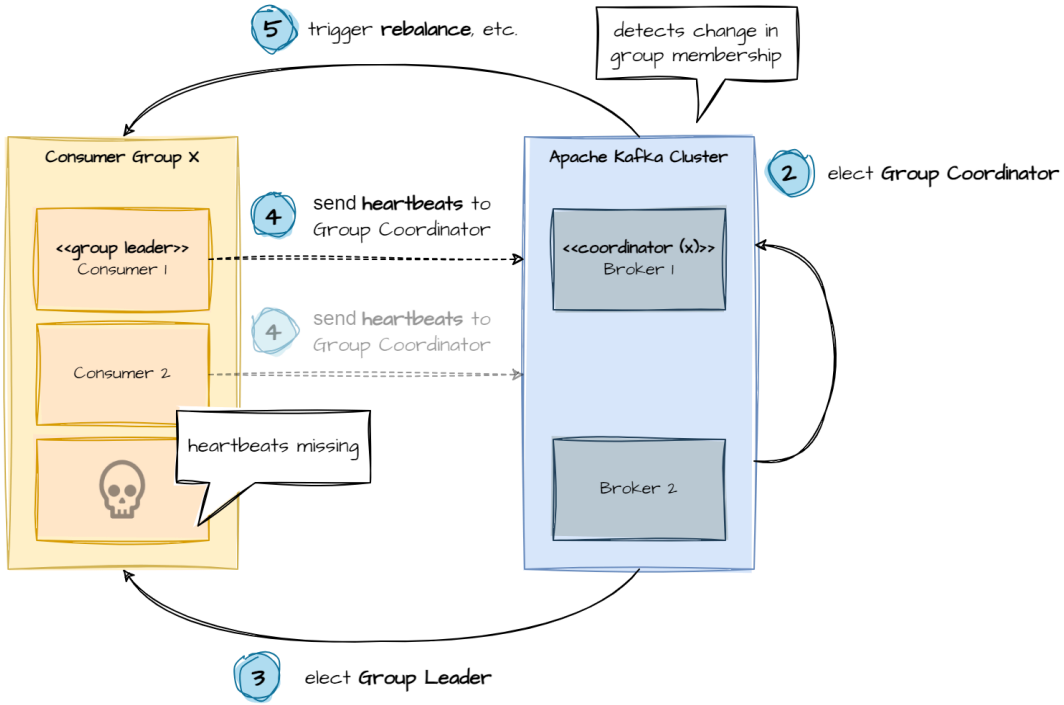


The *Group Leader* is responsible for partition assignments across consumers in the same group. Each partition will have only one consumer assigned.

Every consumer of the group sends regular heartbeats to the *Group Coordinator*.



The *Group Coordinator* detects a change in membership due to missing heartbeats.



Any change in group membership triggers consumer group rebalances.

Rebalance is triggered when

- consumer joins the group
- consumer leaves the group
- client-side failure detected via `max.poll.interval.ms`
- server-side failure detected via `session.timeout.ms`

There are many probable causes for rebalancing.

- Service is scaling up or down
- `poll()` and long message processing occur in the same thread
- Heartbeats do not reach the *Group Coordinator*
- Long JVM garbage collection pauses (stop-the-world)
- Kubernetes pods become CPU-throttled
- Pod evictions due to Kubernetes cluster upgrades
- Networking issues (latency, packet drop, ...)
- ...

The *Group Leader* uses a configurable partition assignment strategy.

- **Range** (default)
  - stop-the-world strategy
  - works on a per-topic basis
  - can generate imbalanced assignments
- **Round Robin**
  - stop-the-world strategy
  - uniformly distributes partitions

The strategy can be set using parameter `partition.assignment.strategy`.

Range (`org.apache.kafka.clients.consumer.RangeAssignor`)

For each topic, we lay out the available partitions in numeric order and the consumers in lexicographic order. We then divide the number of partitions by the total number of consumers to determine the number of partitions to assign to each consumer. If it does not evenly divide, then the first few consumers will have one extra partition.

Round Robin (`org.apache.kafka.clients.consumer.RoundRobinAssignor`)

The round robin assignor lays out all the available partitions and all the available consumers. It then proceeds to do a round robin assignment from partition to consumer. If the subscriptions of all consumer instances are identical, then the partitions will be uniformly distributed.

The *Group Leader* uses a configurable partition assignment strategy.

- **Range** (default)
  - stop-the-world strategy
  - works on a per-topic basis
  - can generate imbalanced assignments
- **Round Robin**
  - stop-the-world strategy
  - uniformly distributes partitions
- **Sticky**
  - stop-the-world strategy
  - initial distribution close to *Round Robin*
  - tries to minimize effect of a rebalance
  - can generate imbalanced assignments
- **Cooperative Sticky** (prefer for newer clusters)
  - incremental rebalance
  - does not stop consumption
  - same logic as *Sticky*

The strategy can be set using parameter `partition.assignment.strategy`.

Sticky (`org.apache.kafka.clients.consumer.StickyAssignor`)

Guarantees an assignment that is as balanced as possible, meaning that the numbers of topic partitions assigned to consumers differ by at most one and each consumer that has 2+ fewer topic partitions than some other consumer cannot get any of those topic partitions transferred to it. It preserves as many existing assignments as possible when a reassignment occurs, thus reducing overhead processing when topic partitions move from one consumer to another.

Cooperative Sticky (`org.apache.kafka.clients.consumer.CooperativeStickyAssignor`)

Follows the same logic as *Sticky*, but allows for cooperative rebalancing while *Sticky* follows the eager rebalancing protocol. Users should prefer this strategy for newer clusters. To enable it properly, all consumers must be configured using this strategy. Important: If you upgrade from 2.3 or earlier, please make sure to follow the Confluent recommendations on switching to this strategy.

A `ConsumerRebalanceListener` enables us to react on altered partition assignments.

```
public interface ConsumerRebalanceListener {

  void onPartitionsRevoked(Collection<TopicPartition> partitions);

  void onPartitionsAssigned(Collection<TopicPartition> partitions);

  default onPartitionsLost(Collection<TopicPartition> partitions) {
    onPartitionsRevoked(partitions);
  }
}
```

- Use it to save / restore offsets to / from external storage
- Use it to make sure that outstanding offsets are committed
- Instance is passed to `subscribe` when subscribing to a topic
- **Important: Different semantics for eager and incremental assignors!**

Summary

What did we learn?

- Client SDK Essentials
- Subscriptions vs. Assignments
- Offset Management
- The `poll()` loop
- Pre-Fetch Buffer
- Consumer Group
- Partition Assignment & Re-Balancing

Summary

What did we learn?

- Client SDK Essentials
- Subscriptions vs. Assignments
- Offset Management
- The `poll()` loop
- Pre-Fetch Buffer
- Consumer Group
- Partition Assignment & Re-Balancing

What's to follow?

- *Deserialization*
- Interceptors
- Consumer Coordinator
- Consumer Designs
- Transactions

Questions?

assignment is available at

bit.ly/kafka-workshop-exchanging-records

Apache Kafka

for Java Developers

Serialization Strategies

Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

Agenda

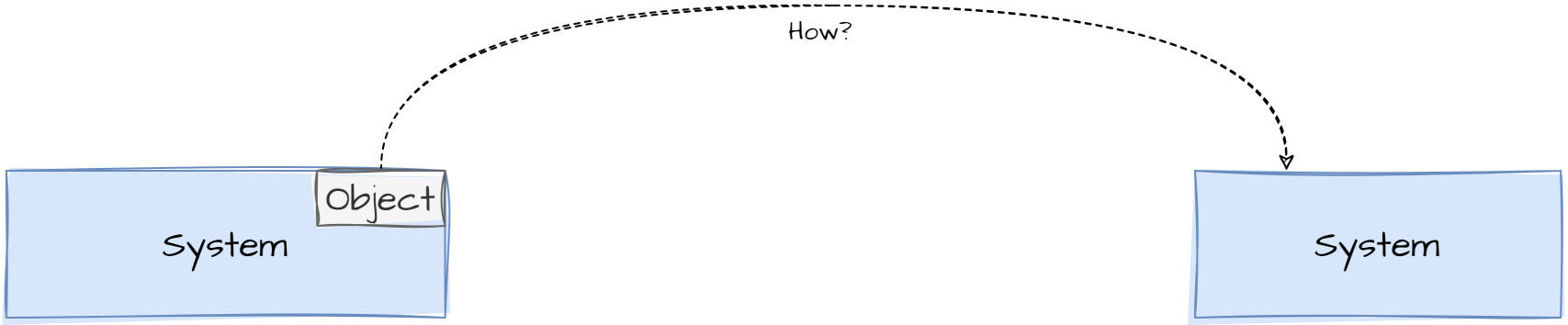
We'll talk about

- Definition, purpose, challenges
- (Basic) Serialization mechanisms and facilities in Kafka
- Common options with pro's and con's
- Solutions for schema management

Data (De)Serialization

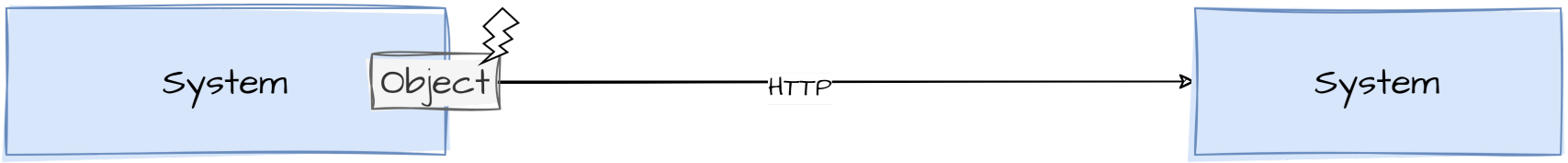
Definition, Purpose, Challenges

Why do we need to serialize data anyways?



- How can systems exchange data?
- We need a way to ...
  - connect systems via network
  - present data in a way that both systems understand

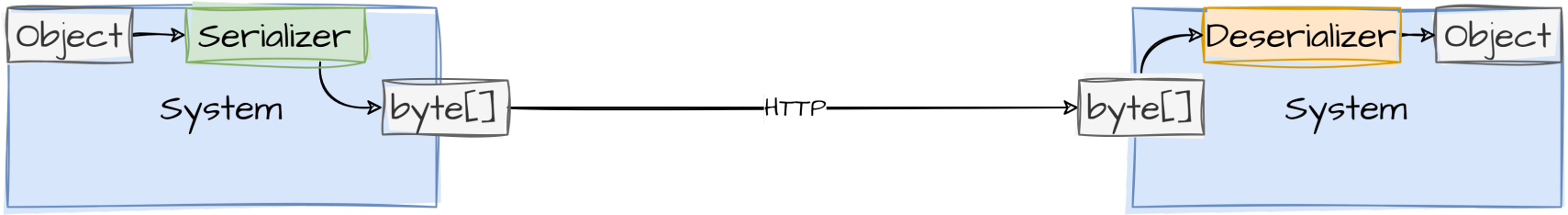
Why do we need to serialize data anyways? (cont.)



- A connection alone is not enough
- Objects are in-memory structures that can't be transferred directly
- Even if you **could** magically share memory over the wire
  - different in-memory representations
  - security nightmare

Why do we need to serialize data anyways? (cont.)

The solution: a common *language* that can be transferred via network



Definition

Serialization

is the process of converting a data object into a series of bytes that saves the state of the object in an easily transmittable form

Deserialization

is the process of reconstructing a data structure or object from a series of bytes or a string in order to instantiate the object for consumption.

Purpose

- Enables efficient, ordered, and reliable data streaming across distributed systems
- Maintains the integrity and consistency of data across systems
- Acts as a communication contract between 1 to n systems
- Allows data to be
  - monitored easily while *in-flight*
  - mocked for testing / development purposes
  - saved and analyzed / debugged later on

Some common challenges

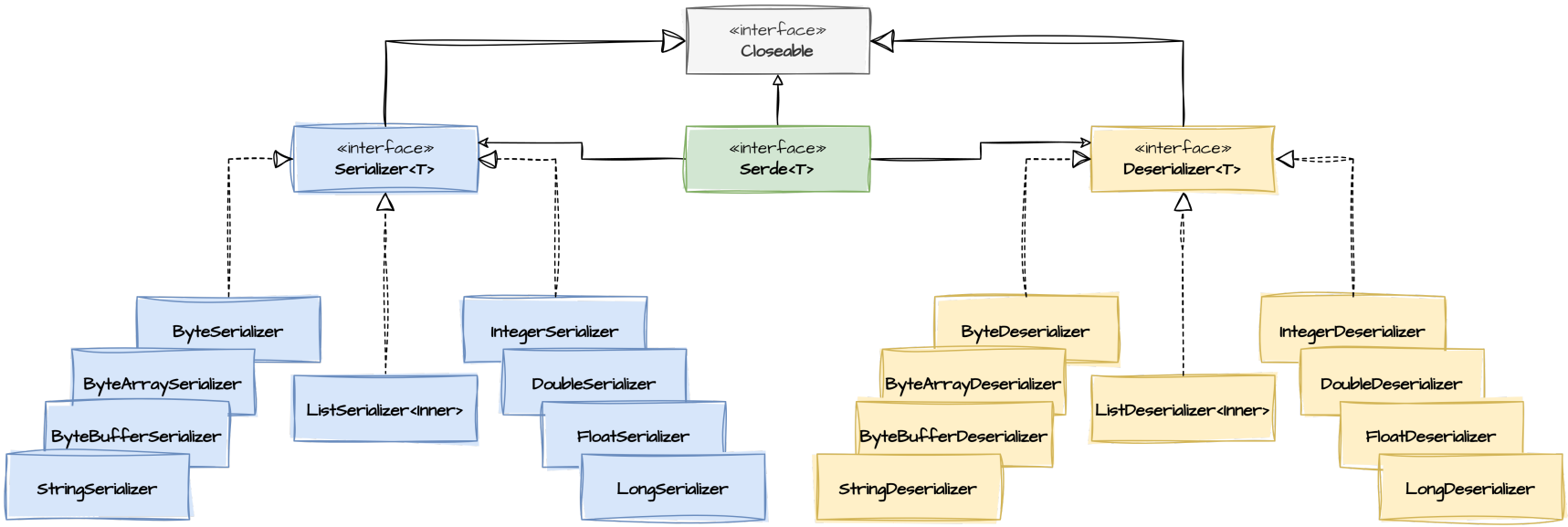
- Structures change / evolve
  - How can we keep that in sync on a multitude of systems?
- Transferring large datasets
  - Is the *language* and medium up to the task?
  - Is compression feasible?
- Computational overhead
  - (De)Serialization costs CPU time
  - **Especially** when size matters

Serialization Mechanisms in Kafka

Kafka and (De)Serialization

- Kafka ...
  - acts as a transport medium for data between *producer* & *consumer*
  - doesn't care about the data contract (unless you use a schema registry)
  - is agnostic regarding the structure of the data
- But: Kafka provides ...
  - mechanisms how applications can handle these issues
  - a set of default implementations for primitive data types

Kafka (De)Serializer interfaces and built-in support



The `Serializer` interface

```
public interface Serializer<T> extends Closeable {

  byte[] serialize(String topic, T data);

  default byte[] serialize(String topic, Headers headers, T data) {
    return serialize(topic, data);
  }

  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

The Deserializer interface

```
public interface Deserializer<T> extends Closeable {

    default void configure(Map<String, ?> configs, boolean isKey) {
        // intentionally left blank
    }

    default T deserialize(String topic, Headers headers, byte[] data) {
        return deserialize(topic, data);
    }

    default T deserialize(String topic, Headers headers, ByteBuffer data) {
        return deserialize(topic, headers, Utils.toNullableArray(data));
    }

    @Override
    default void close() {
        // intentionally left blank
    }
}
```

Configuring the (De)Serializer

```
public class ConfigurationExample {

  public void basicProducerConfiguration() {
    Map<String, Object> config = Map.of(
      ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(),
      ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName()
    );
    // ...
  }

  public void basicConsumerConfiguration() {
    Map<String, Object> config = Map.of(
      ConsumerConfig.KEY_DESERIALIZER_CONFIG, StringDeserializer.class.getName(),
      ConsumerConfig.VALUE_DESERIALIZER_CONFIG, StringDeserializer.class.getName()
    );
  }

}
```

The Kafka Clients facilities are pretty bare metal

- What's with complex objects?
- What happens when a (De)Serializer fails?
- What about schema management / validation?

With only the Kafka Client library you have to solve these issues on your own, e.g. with a custom (De)Serializer.

Advanced Use-Cases (cont.)

- Using JSON Objects is very common
- Easy to integrate with e.g. Jackson / Gson
- Spring Kafka has more convenience
  - Provides a `JsonSerializer` / `JsonDeserializer`
  - Provides a `ErrorHandlingDeserializer` that can handle faulty data
  - But is very opinionated (as usual) and might cause problems with other clients

We will build our own `JsonSerializer` and `JsonDeserializer` as part of the lab

Data Serialization Options

Schema Management in Kafka SerDes

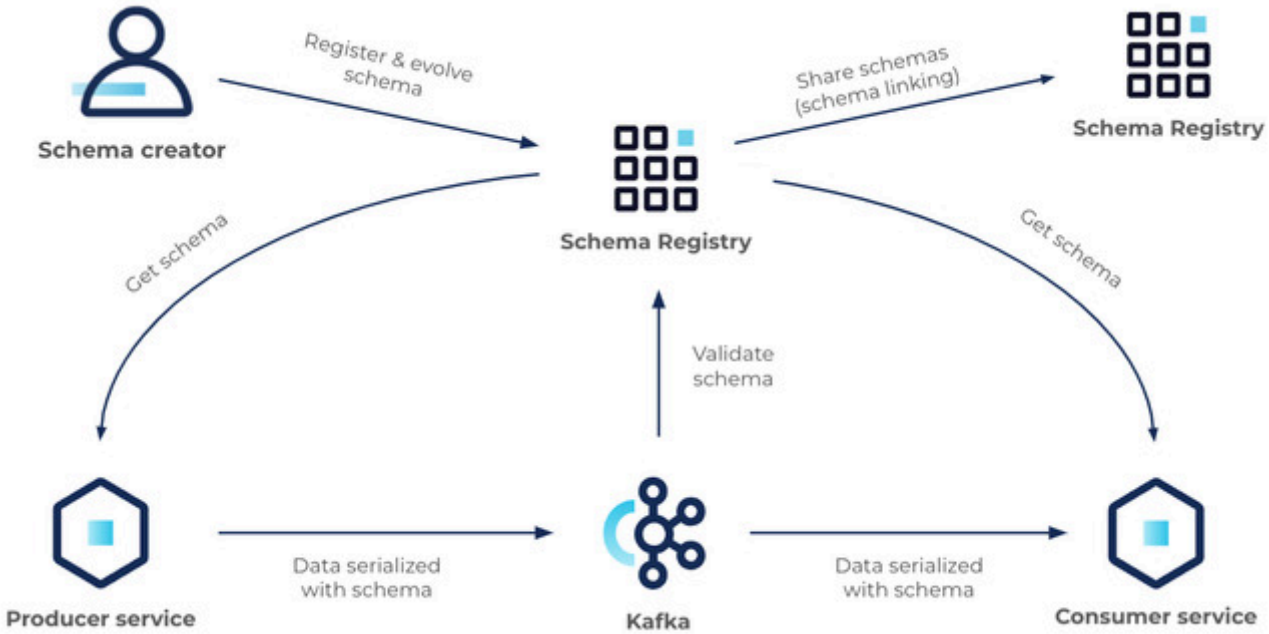
- Ensures compatibility across different versions of data producers and consumers.
- **Key Options**
    1. Confluent Schema Registry
    2. Apache Avro without a Registry
    3. Protobuf with or without a Registry
    4. JSON Schema

> *Choosing the right schema management tool depends on needs for compatibility, performance, and ease of use.*

Confluent Schema Registry

- A centralized service that provides runtime schema enforcement.
- **Pros**
  - Centralizes schema management.
  - Supports schema evolution with full compatibility checks.
  - Reduces payload size as schema is not included in each message.
- **Cons**
  - Introduces a single point of failure in the architecture.
  - Dependency on external service increases system complexity.
  - Primarily supports Avro, limited support for other formats.

Confluent Schema Registry (cont.)



Apache Avro without a Registry

- Using Avro for serialization without a centralized schema registry
- Typically means embedding the schema within each message payload
- **Pros**
  - Schema is self-contained within each message, ensuring the consumer can always deserialize data
  - Removes the dependency on an external schema registry service
  - Flexible and simple to implement in small-scale systems

Apache Avro without a Registry (cont.)

- **Cons**
  - Increases message size as schema is included in every message
  - No centralized schema management, which can lead to inconsistencies
  - Lacks automatic compatibility checks, increasing the risk of runtime errors

Protobuf with or without a Registry

- Binary serialization format by Google
- **Pros**
  - Highly efficient binary format reduces message size and improves performance
  - Strongly typed, which helps in catching errors during development
  - Schema registry integration is optional but recommended for large scale deployments
- **Cons**
  - Steeper learning curve due to its binary nature and tooling
  - Less human-readable than JSON or Avro
  - Managing schemas without a registry can be challenging in large environments

JSON Schema

- Defines the structure of JSON data for validation and documentation
- **Pros**
  - Human-readable and easy to understand, making it popular for web applications
  - Flexible and easy to integrate with modern web technologies
  - Does not require a centralized registry, simplifying deployment
- **Cons**
  - Less efficient in terms of payload size compared to binary formats like Protobuf
  - Lacks built-in support for schema versioning, evolution and has weak schema enforcement

Overview

	Serialization libraries			Popular formats	
	Avro	Thrift	Protobuf	JSON	XML
Binary representation	+	+	+	BSON	EXI
Generic data types	+	+/-	-	+	+
Schema-based	+	+	+	-	+
Supports schema evolution	+	+	+	-	-
Specific encoding	+	+	+	-	+
Browser support	+	+	-	++	+
Date types	+	-	+	-	+

Summary

Summary

Data serialization is a complex topic that will have a major impact on your system landscape in the long run!

- Choosing the right **serialization strategy** is very important as the wrong choice can be hard to fix later on. As usual: **there is no silver bullet solution**
- Lots of choices, all have their benefits and drawbacks
- A perfect fit might not exist, but there are some key questions you can as! yourself to make a good decision

Key questions

- *What are my performance requirements?*
  - Do I need to optimize for high throughput or low latency?
  - How does the serialization format impact the performance of my system?
- *What is the nature of the data being serialized?*
  - Is the data highly structured or schema-less?
  - Does the data format need to support complex types and hierarchies?

Key questions (cont.)

- *How important is schema evolution to my application?*
  - Will the data structure change over time?
  - Do I need backward and forward compatibility between different versions of the schema?
- *What are the system integration requirements?*
  - Which programming languages and frameworks are being used?
  - Do these technologies have native support or robust libraries for the serialization format I'm considering?

Key questions (cont.)

- *What are the operational considerations?*
  - Do I have the resources to manage a schema registry?
  - What are the implications of adding a schema registry in terms of setup, maintenance, and overhead?
- *How does the choice of serialization impact data security and compliance?*
  - Does the serialization format or schema registry offer features that enhance data security, such as encryption or access control?
  - Are there compliance requirements for data storage or transmission that could influence the choice of serialization?

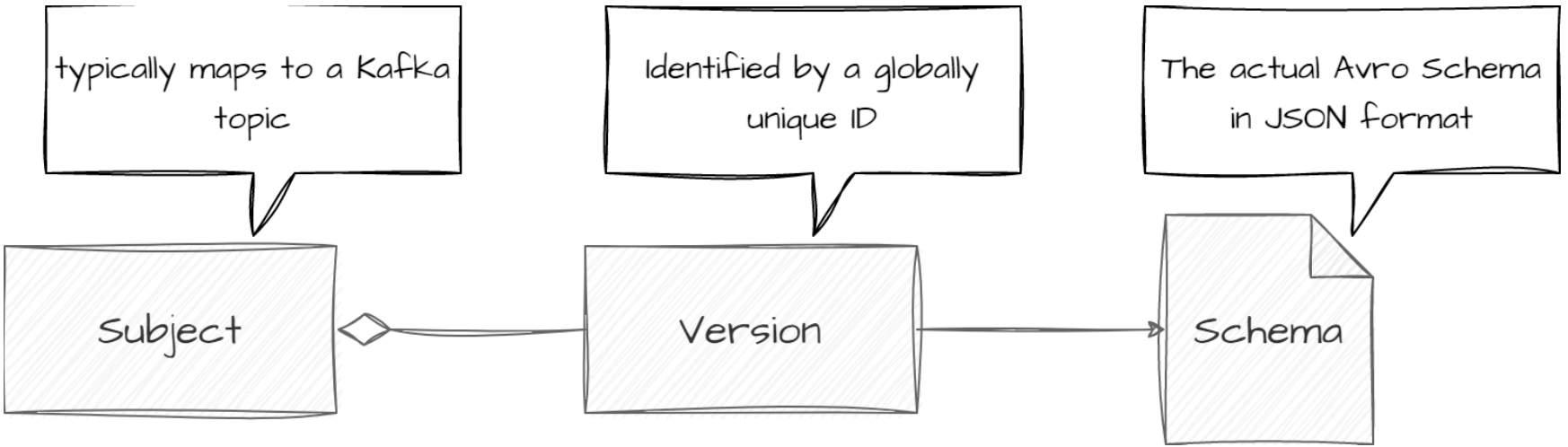
Questions?

assignment is available at

bit.ly/kafka-workshop-serialization-strategies

Confluent Schema Registry

The data model of Confluent's Schema Registry is quite simple.



What is a subject?

A subject

- is typically associated with a topic
  - `{topic-name}-key`
  - `{topic-name}-value`
- Binding between subject and topic *is not strict*
  - the ID of a version is unique across all subjects
  - possible to use the same schema for multiple topics

Confluent Schema Registry also provides a REST API with endpoints to manage schemas.

- **`GET /subjects`**: Get a list of all subjects.
- **`GET /subjects/{subject}/versions`**: Fetch all versions of the schema registered under the specified subject.
- **`GET /subjects/{subject}/versions/{version}`**: Fetch a specific version of the schema registered under the specified subject.
- **`POST /subjects/{subject}/versions`**: Register a new version of the schema under the specified subject.
- **`DELETE /subjects/{subject}/versions/{version}`**: Delete a specific version of the schema registered under the subject.

Apache Kafka

for Java Developers

Reference Architecture and Examples

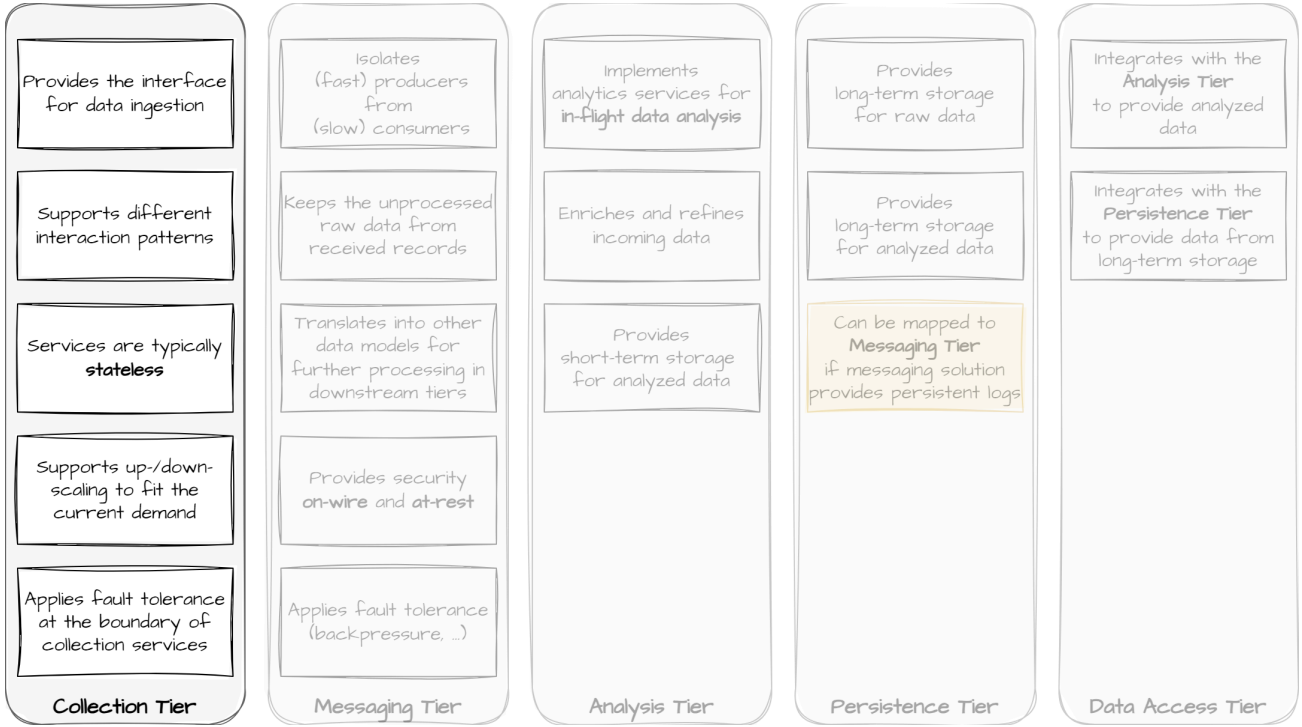
Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

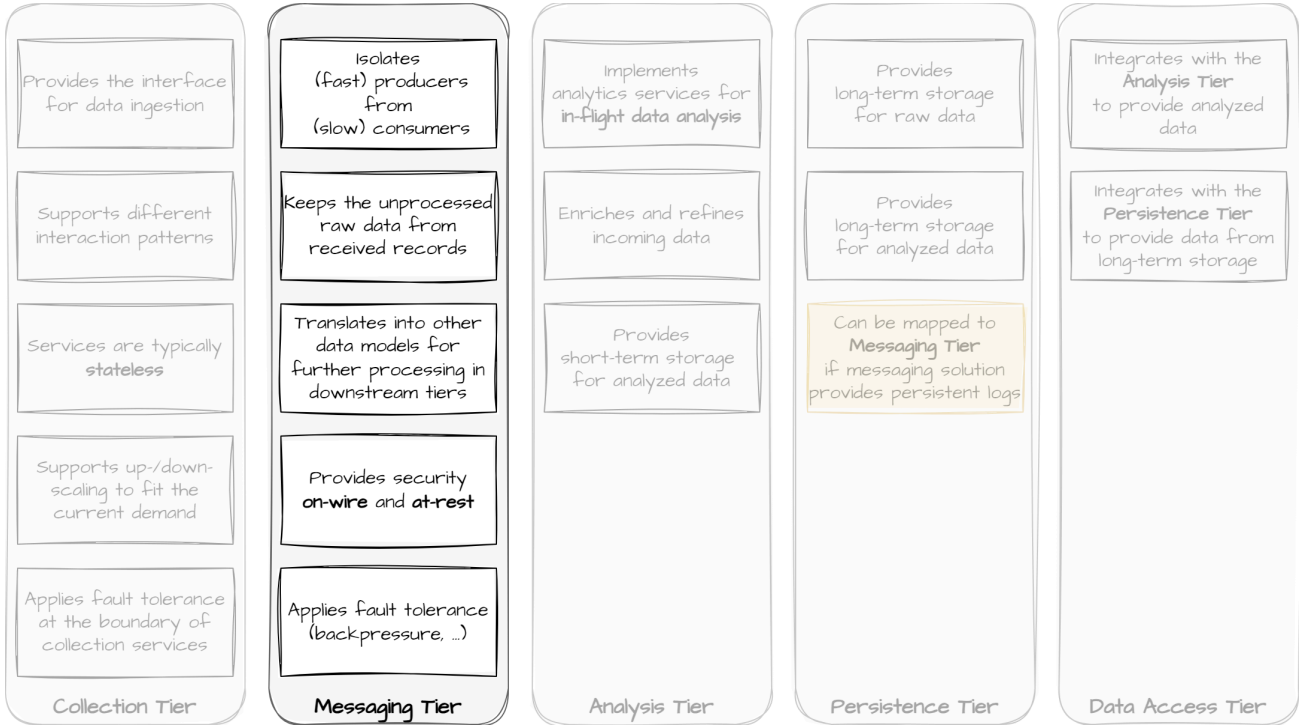
Reference Architecture

for Data Streaming Applications

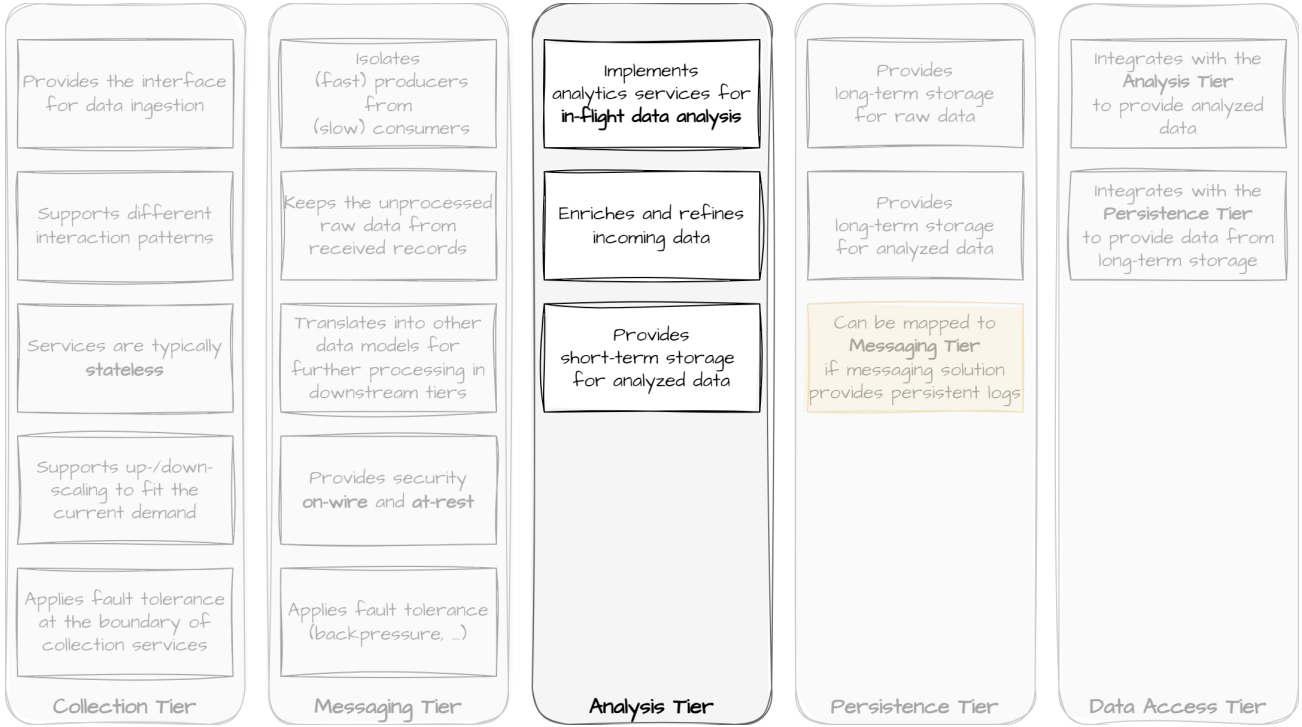
The *Collection Tier* is the entry point for bringing data into our streaming system.



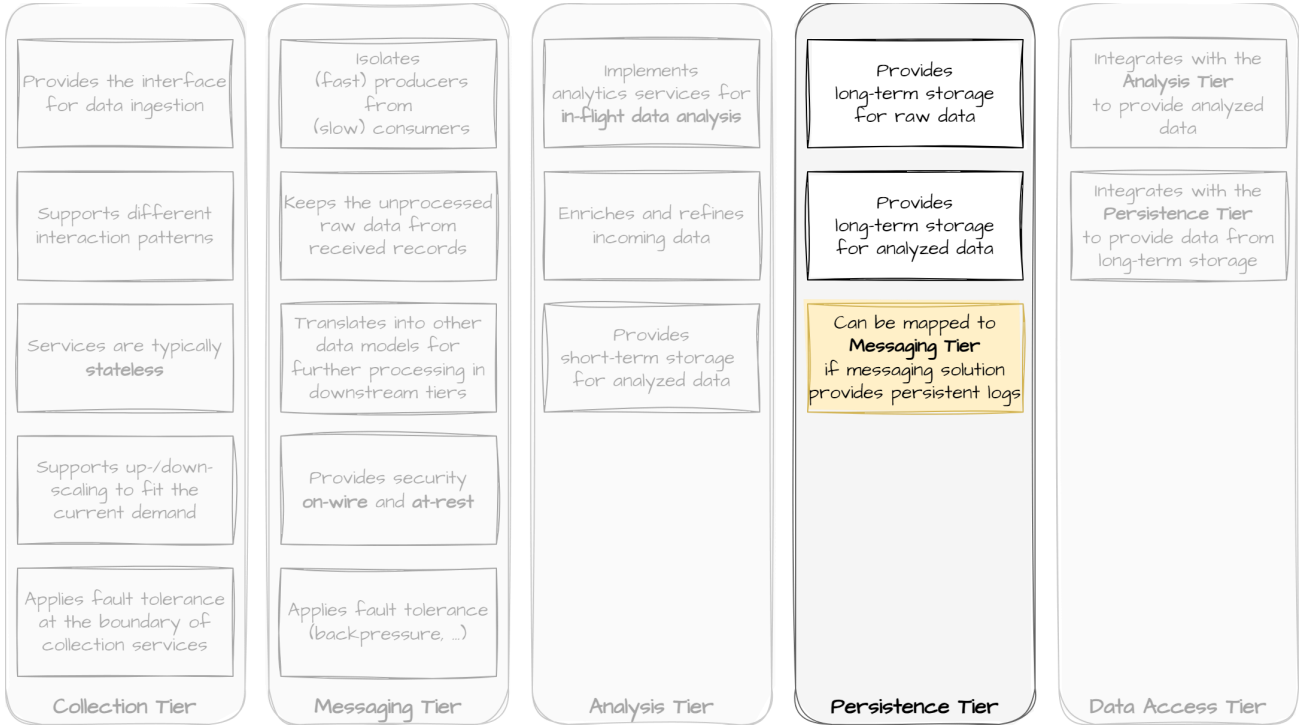
The *Messaging Tier* transports data from the *Collection Tier* to the rest of the pipeline.



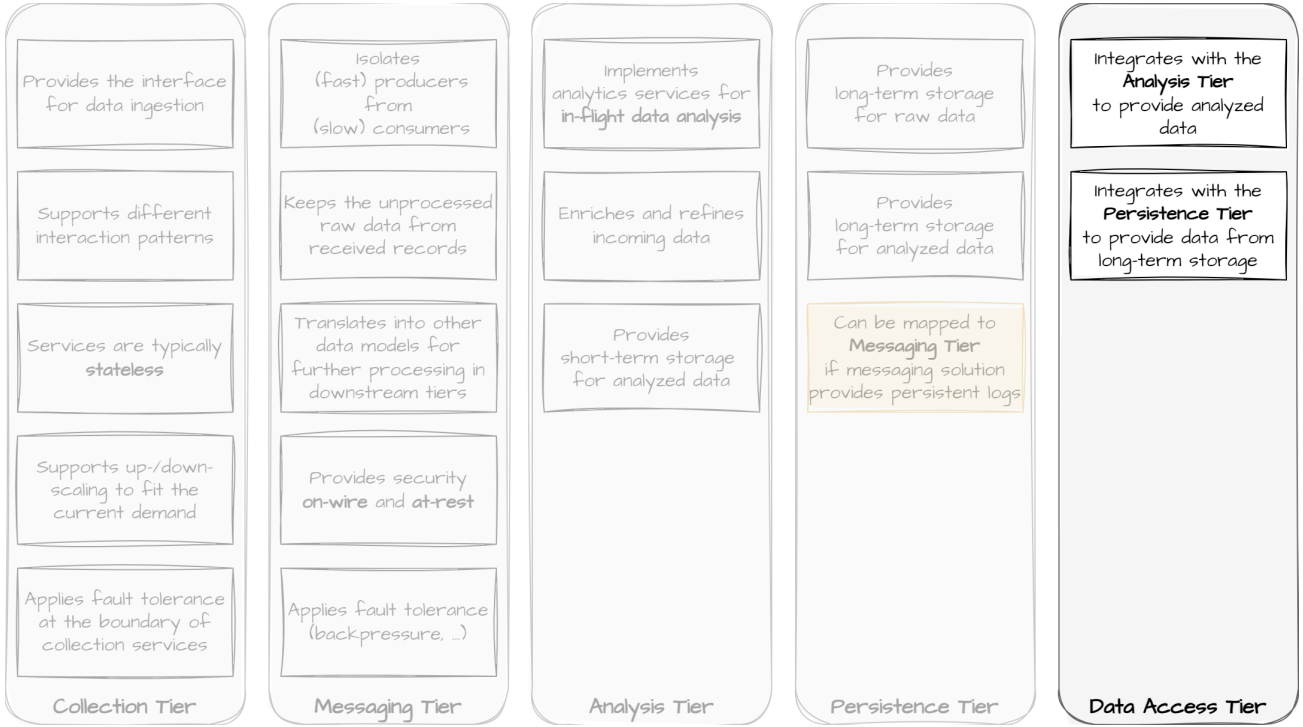
The *Analysis Tier* provides the capability of in-flight data analysis.



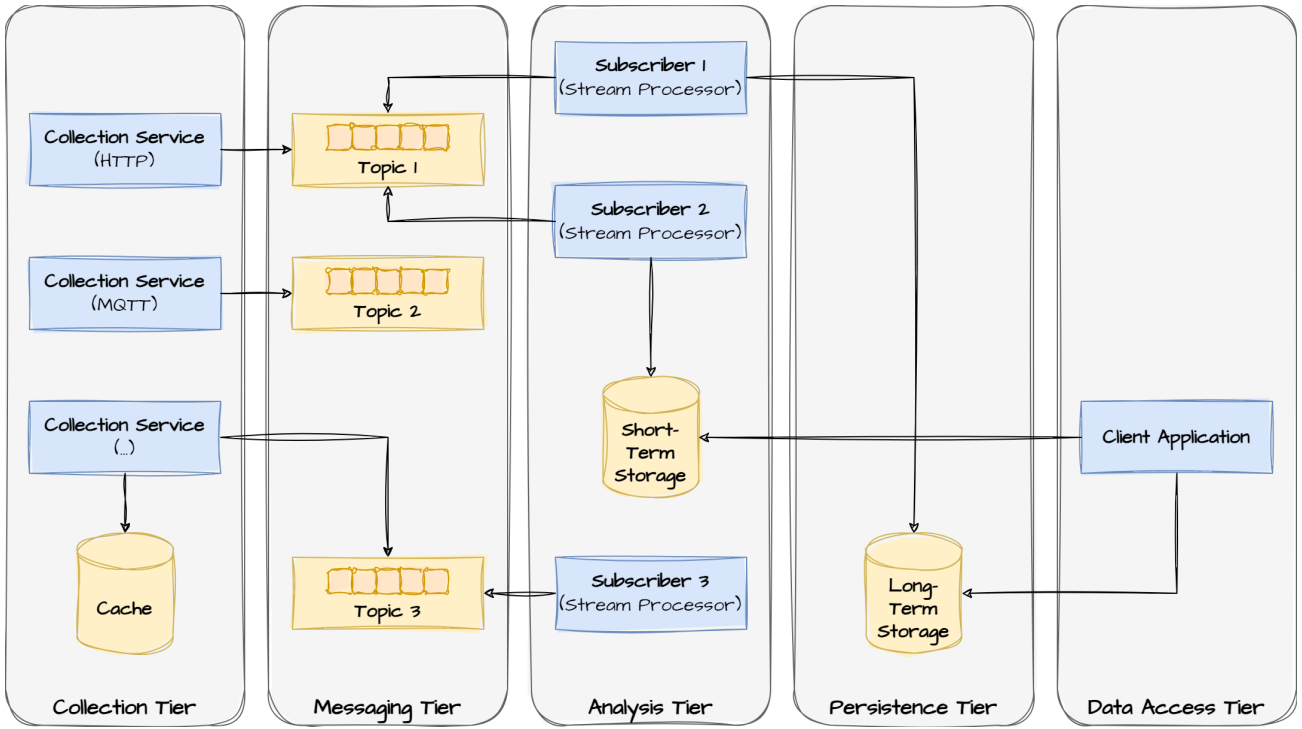
The *Persistence Tier* allows us to store raw, refined, and enriched data for later usage.



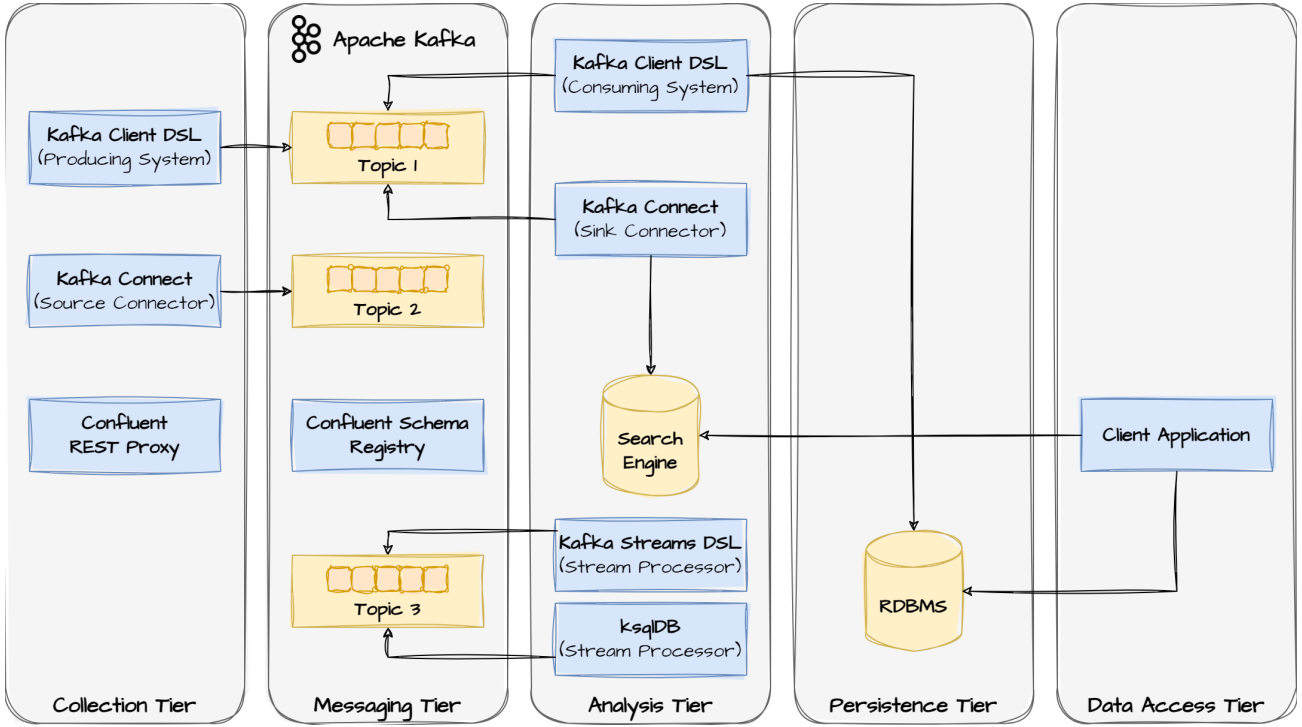
The *Data Access Tier* integrates with *Analysis* and *Persistence Tier* to make data available.



A reference architecture helps to reason about tiers and (non-)functional requirements.



Kafka features a rich ecosystem of services that fit into the tiers of a streaming architecture.



Real-World Examples

Example #1: Efficient persisting and querying of audit trail data and sensor events.

Status

- Data origins at
  - Backend service (HTTP)
  - Sensors in the field (MQTT)
- RDBMS for write- and read-side
  - Optimized for writes, not reads
  - Moderate write requests: ~235 req. / s

Example #1: Efficient persisting and querying of audit trail data and sensor events.

Status

- Data origins at
  - Backend service (HTTP)
  - Sensors in the field (MQTT)
- RDBMS for write- and read-side
  - Optimized for writes, not reads
  - Moderate write requests: ~235 req. / s

Analysis

- Writes introduce heavy load on the DB
- (Non-optimized) Reads perform abysmal

Example #1: Efficient persisting and querying of audit trail data and sensor events.

Status

- Data origins at
  - Backend service (HTTP)
  - Sensors in the field (MQTT)
- RDBMS for write- and read-side
  - Optimized for writes, not reads
  - Moderate write requests: ~235 req. / s

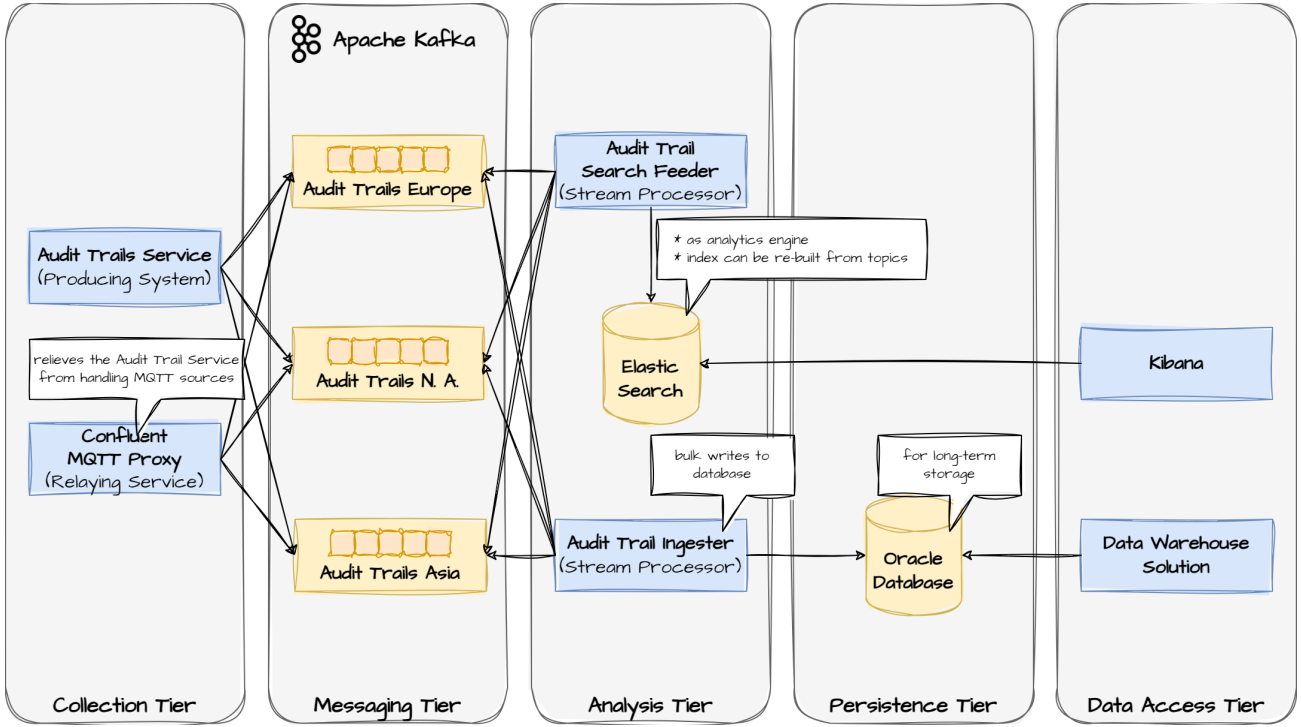
Analysis

- Writes introduce heavy load on the DB
- (Non-optimized) Reads perform abysmal

Proposal and implementation

- Decouple write- from read-side
- Use optimized store for reads

Example #1: Efficient persisting and querying of audit trail data and sensor events.



Example #2: Collecting and analyzing sensor data for alerting in near real-time

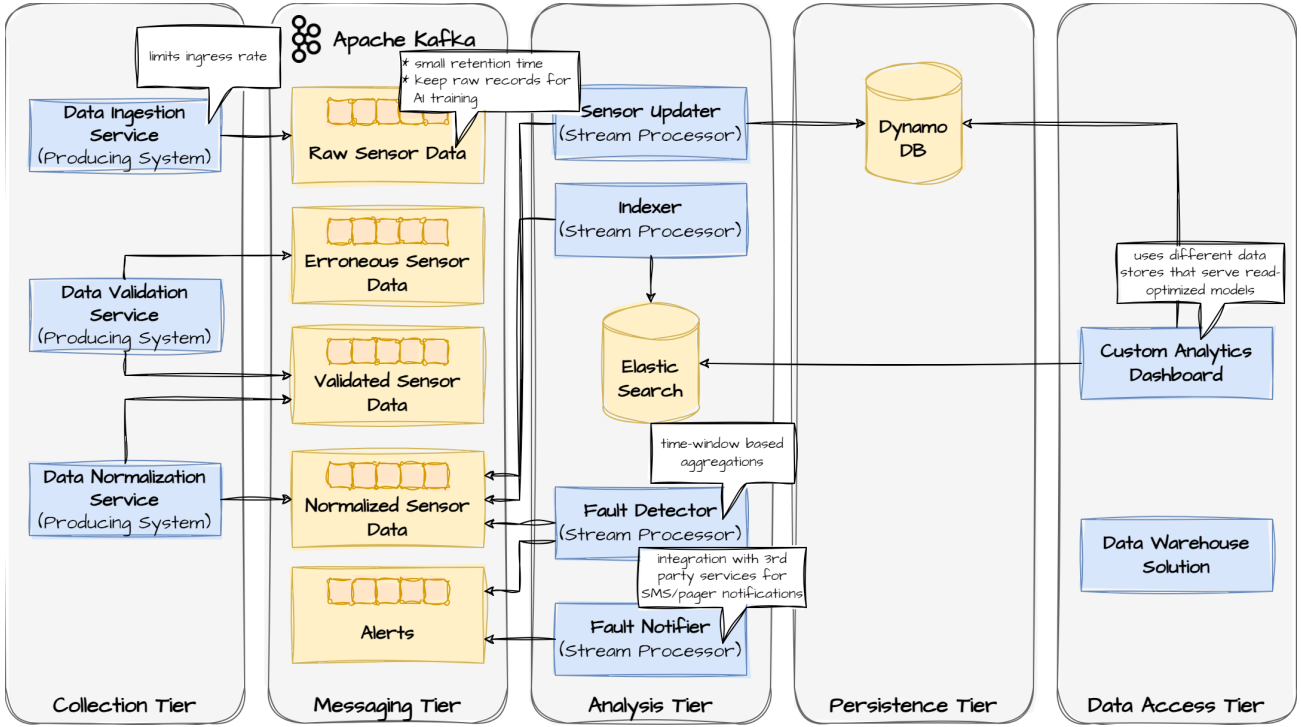
Project

- Sensors periodically push health data
- Single machine houses 10-50+ sensors
  - i machines per site
  - j sites per tenant
  - k tenants that need supervision
- Near real-time processing to detect errors
- Manufacturers use different data models

Status

- Existing system does not scale
  - due to used broker
  - due to software design
- Services / tiers intermixed
  - hard to optimize for non-functional requirements
- **Proposal: Redesign core messaging**

Example #2: Collecting and analyzing sensor data for alerting

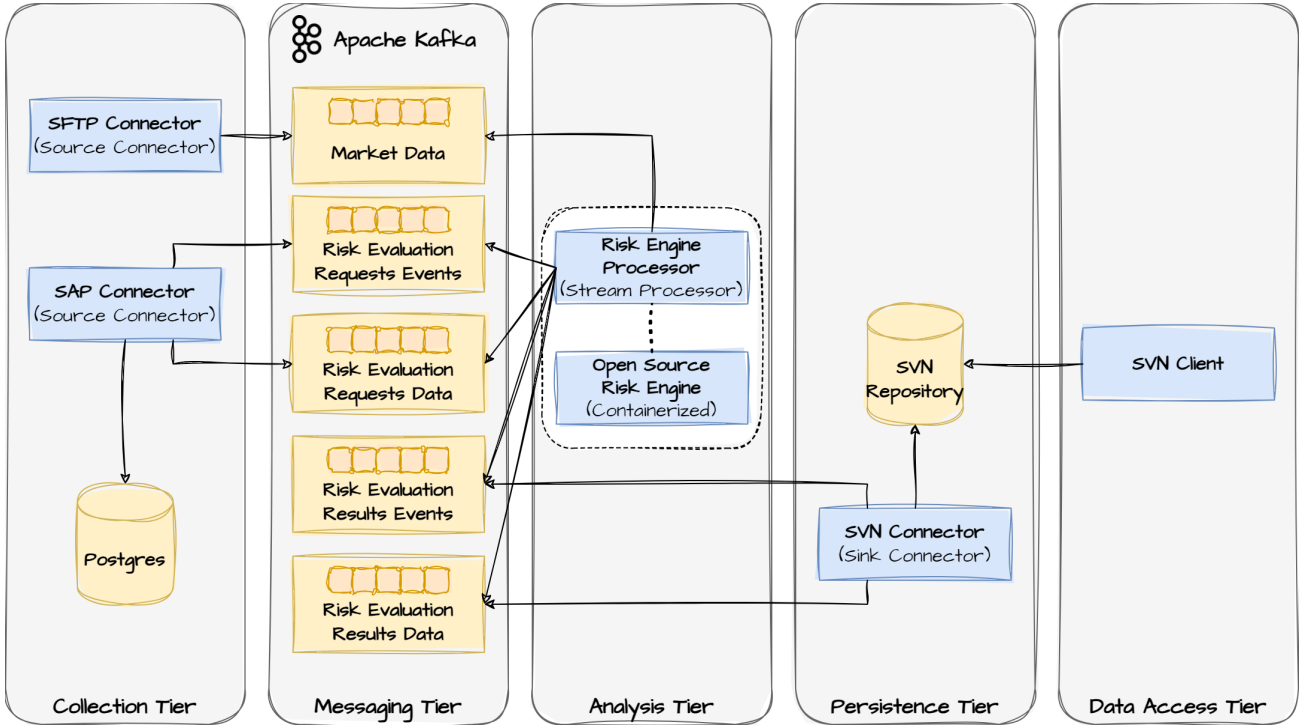


Example #3: On-the-fly risk estimation on mortgages (greenfield)

Project

- Financial risk assesors evaluate the risk on mortgages using software tools
  - given current market data
  - given current the property in question
  - given the investor's / house builder's financial situation
- Necessary data channels do not have streaming capabilities (SAP, ...)
- Risk evaluation software is a C++ application with manual feed-in of data

Example #3: On-the-fly risk estimation on mortgages (greenfield)



Questions?

Apache Kafka

for Java Developers

What's next?

Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

You're right at the beginning of your Kafka journey...

If you want to learn more, we offer deep-dive courses on several topics.

Kafka Workshops (*planned*)

- Apache Kafka for Java Developers
- Introduction to Kafka Connect
- Mastering Kafka Streams

If you want to learn more, we offer deep-dive courses on several topics.

Kafka Workshops (*planned*)

- Apache Kafka for Java Developers
- Introduction to Kafka Connect
- Mastering Kafka Streams

Other Workshops

- Reactive Application Development with Quarkus
  - 3-day workshop
- Spring (Boot) Deep Dive
  - 2-3 day workshop
  - various specializations

Thank you!