

Apache Kafka

for Java Developers

Consuming Records

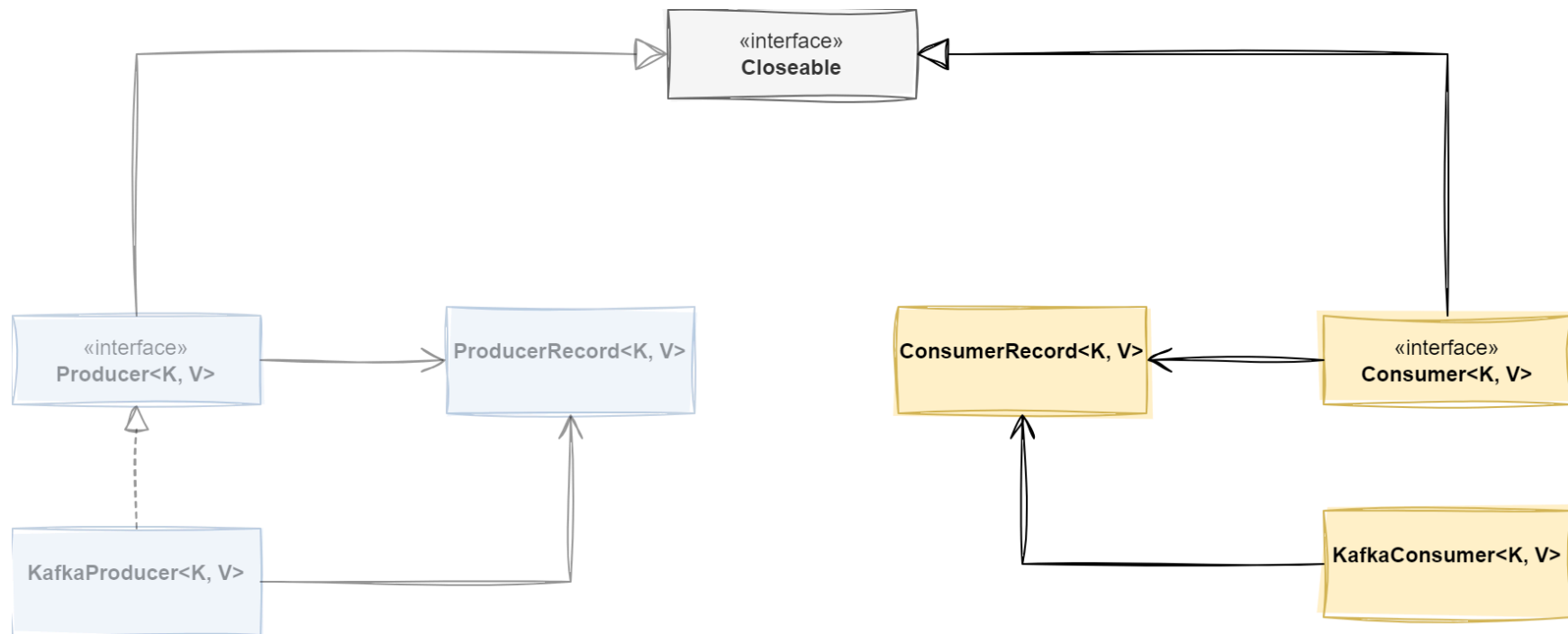
Boris Fresow, Markus Günther

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.



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.

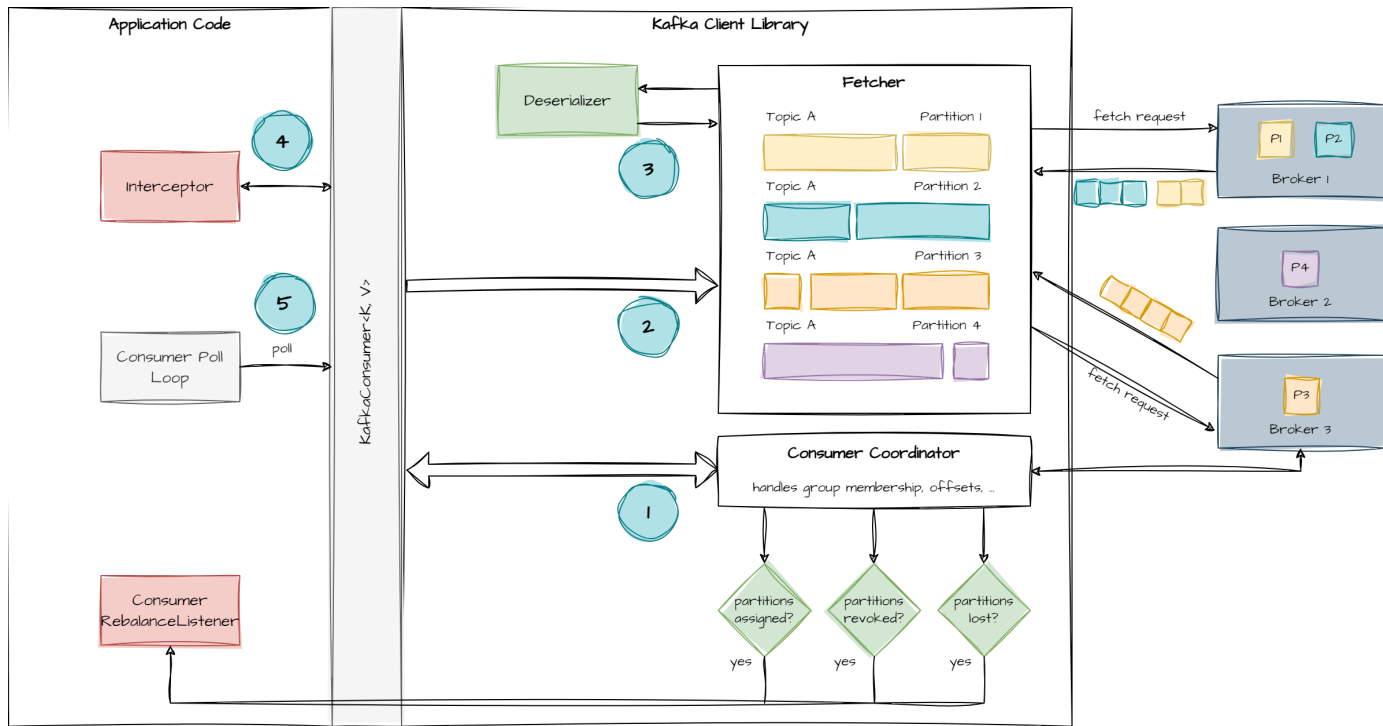
via `subscribe`

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

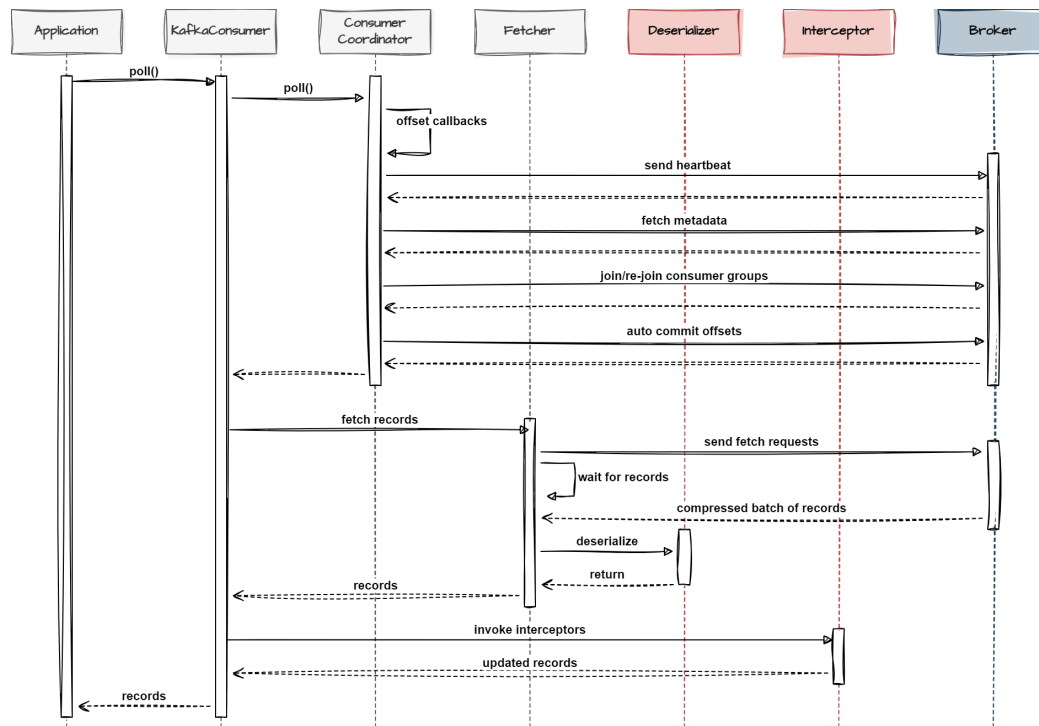
via `assign`

- finer control with topic-partition subscription
- manual offset management
- supports multiple consumers per partition

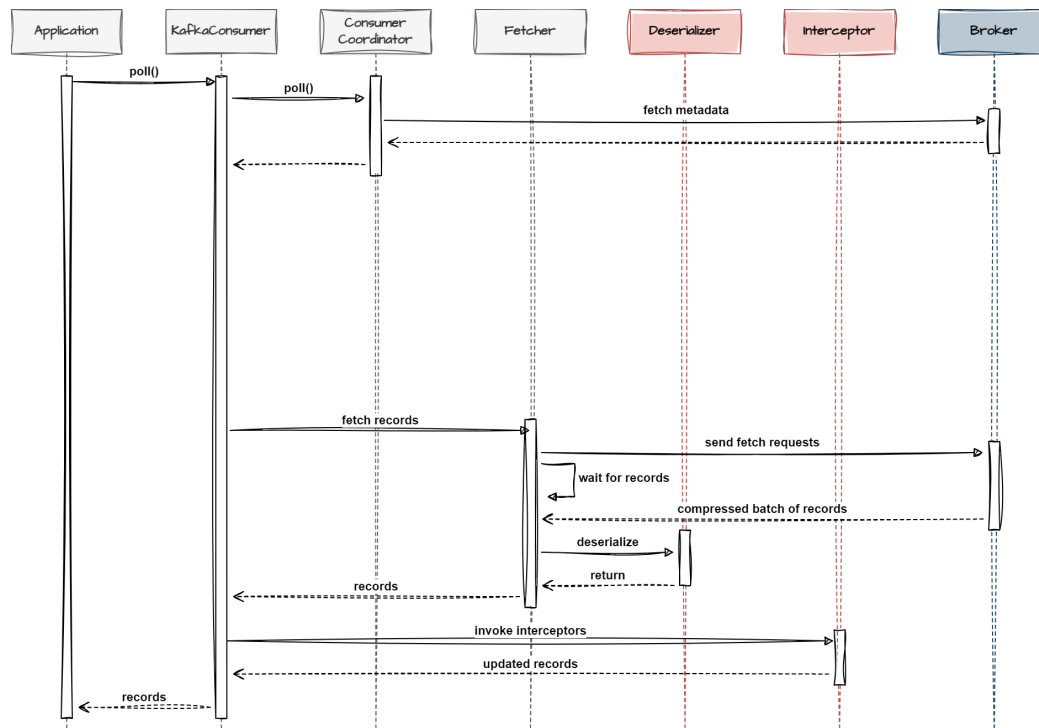
But what happens after subscribing and entering the poll-loop?



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`

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

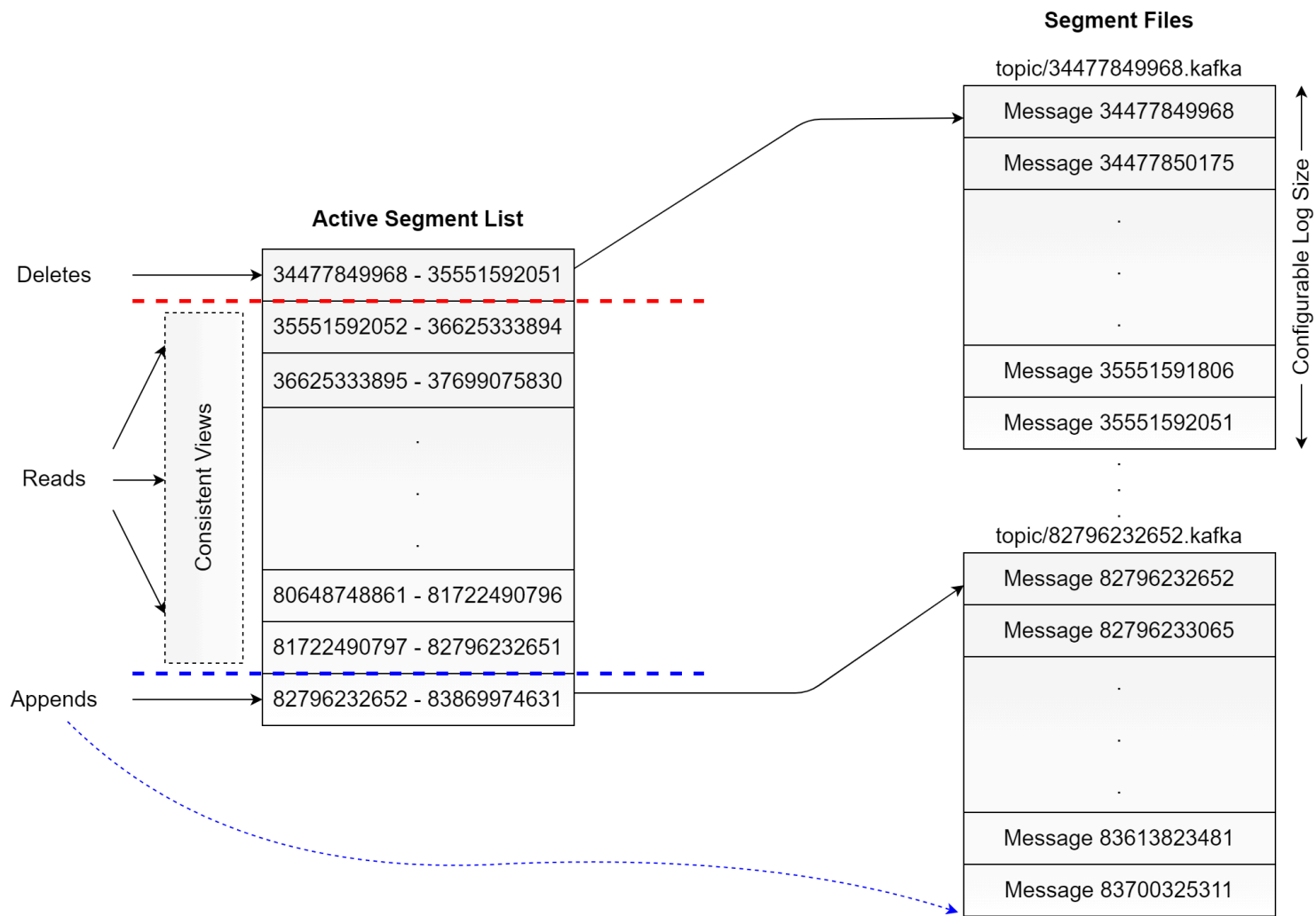
By default, a Kafka consumer will automatically commit offsets every five seconds.

- 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);  
}
```

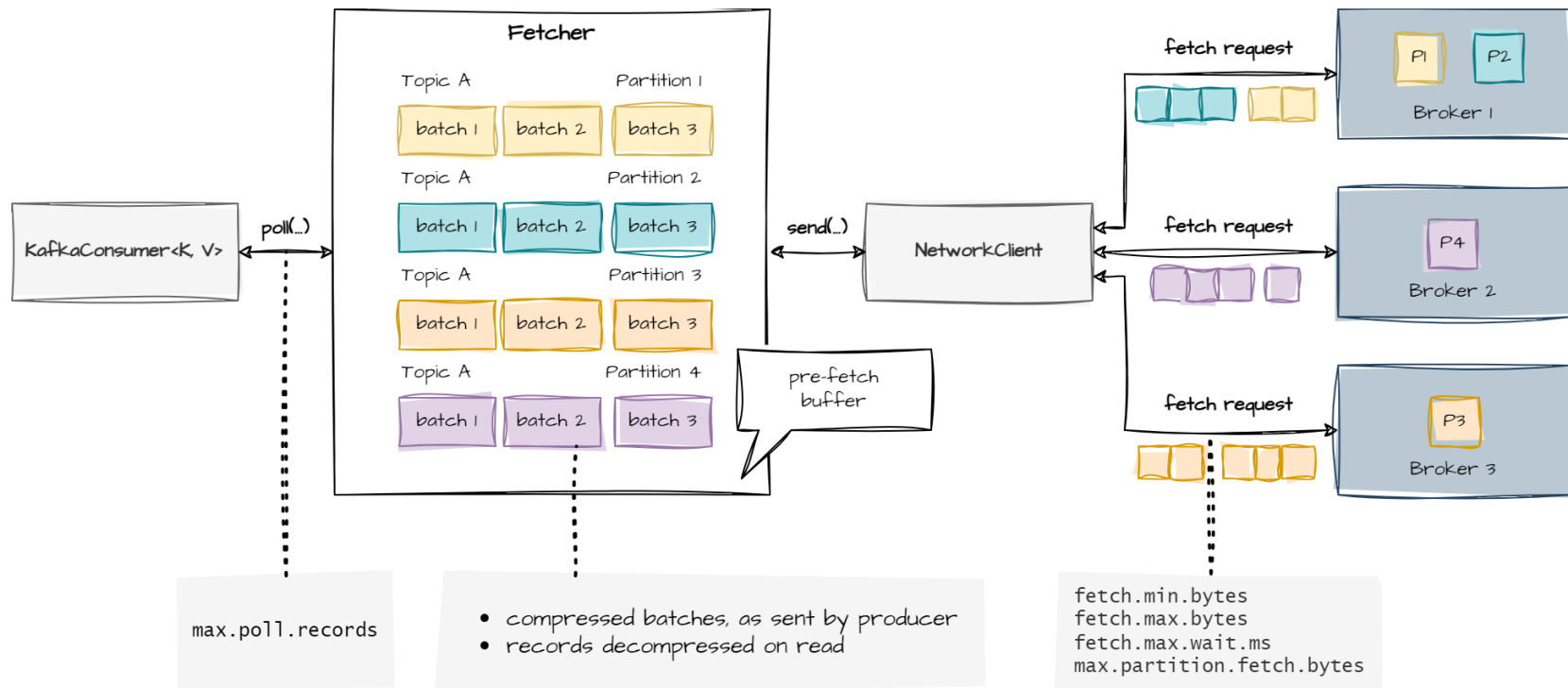
In case of no persisted offsets, `auto.offset.reset` controls where the consumer starts.

- 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



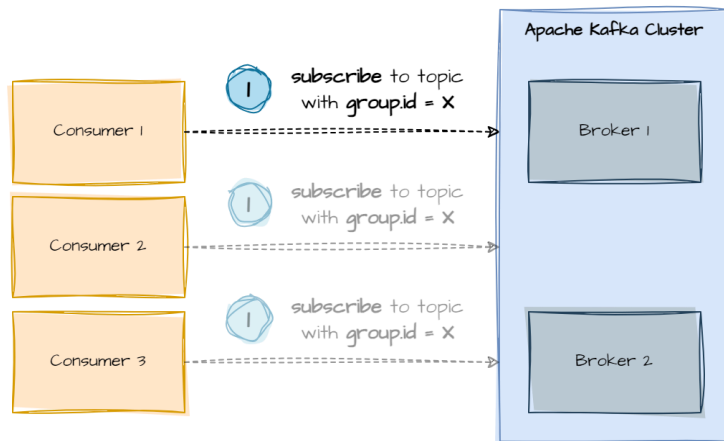
Fetcher

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

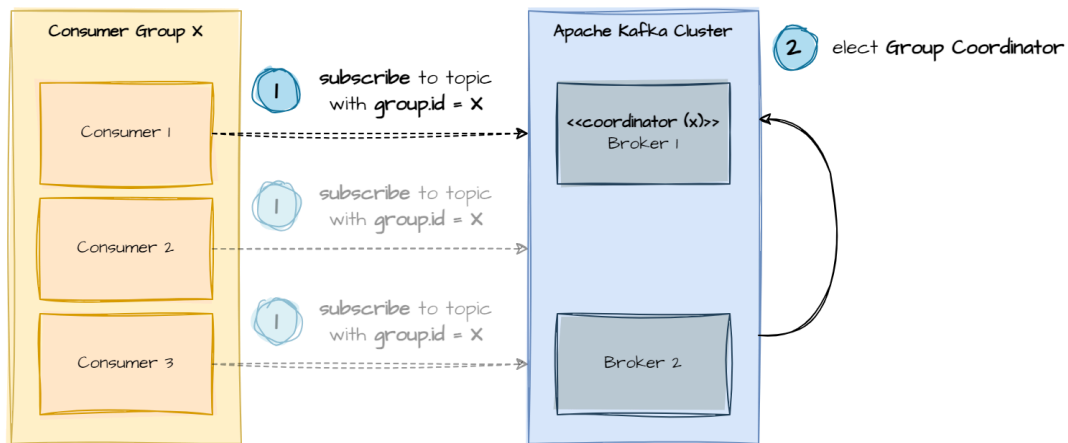


Consumer Groups

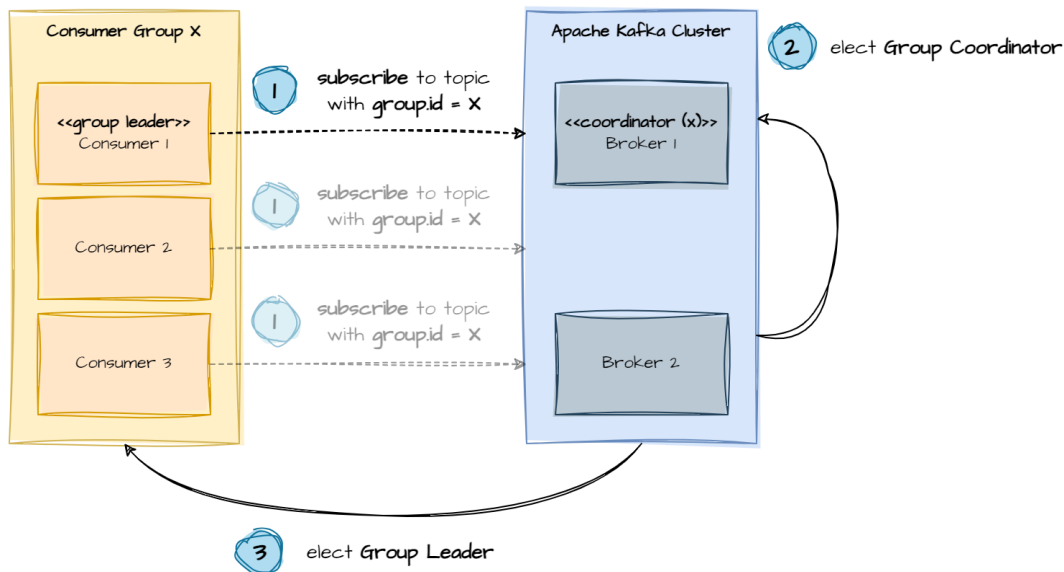
Consumers with the same `group.id` form a consumer group to cooperate.



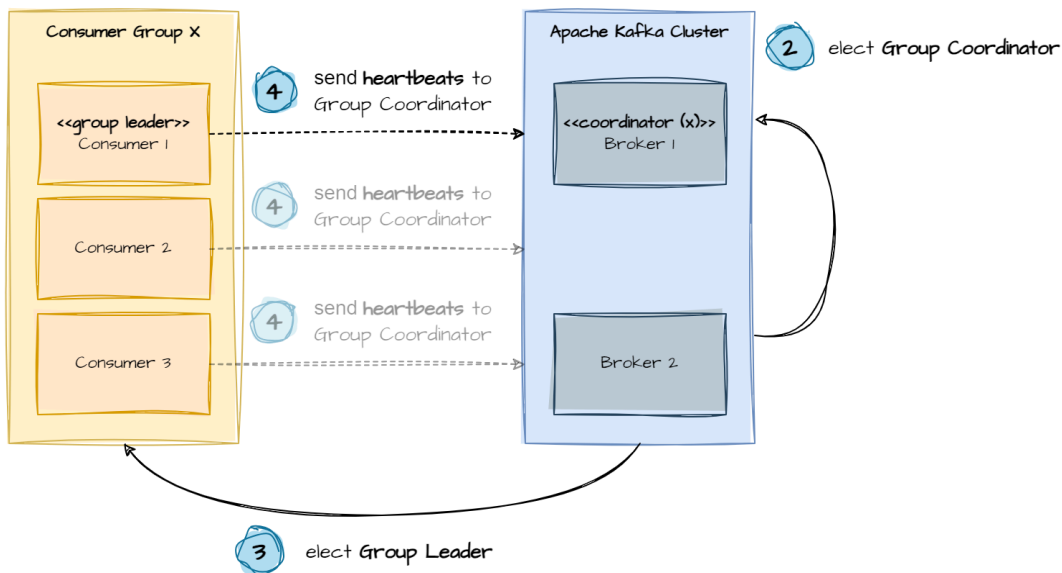
The Kafka cluster elects one of the brokers as *Group Coordinator*.



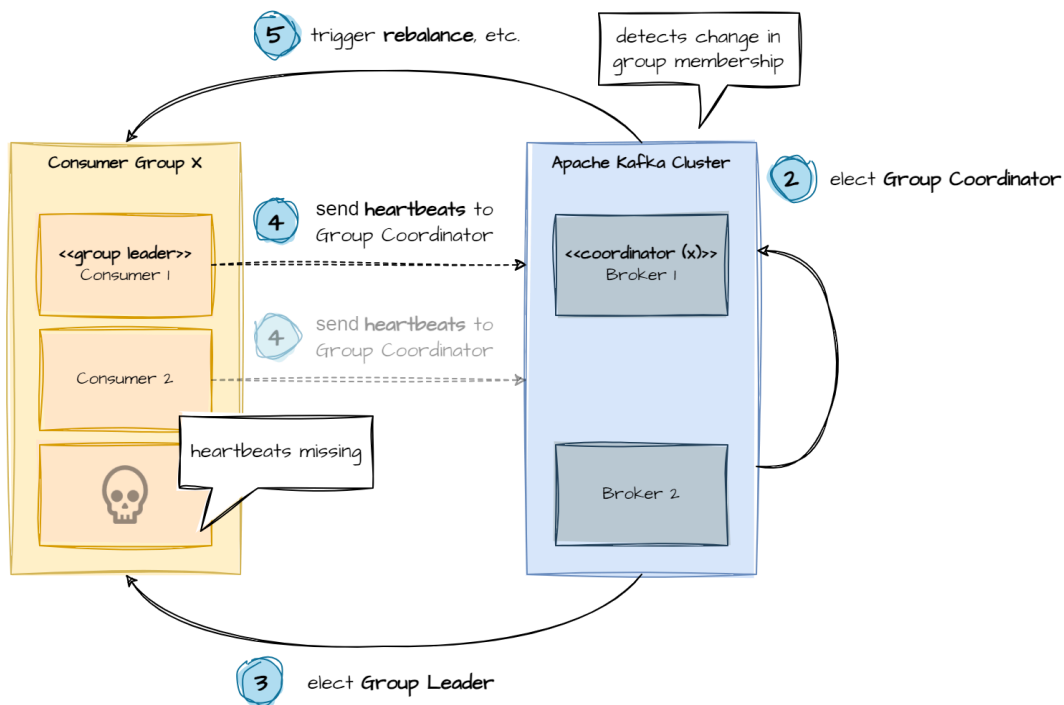
The *Group Coordinator* elects one consumer as *Group Leader*.



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 *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*

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!**

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 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