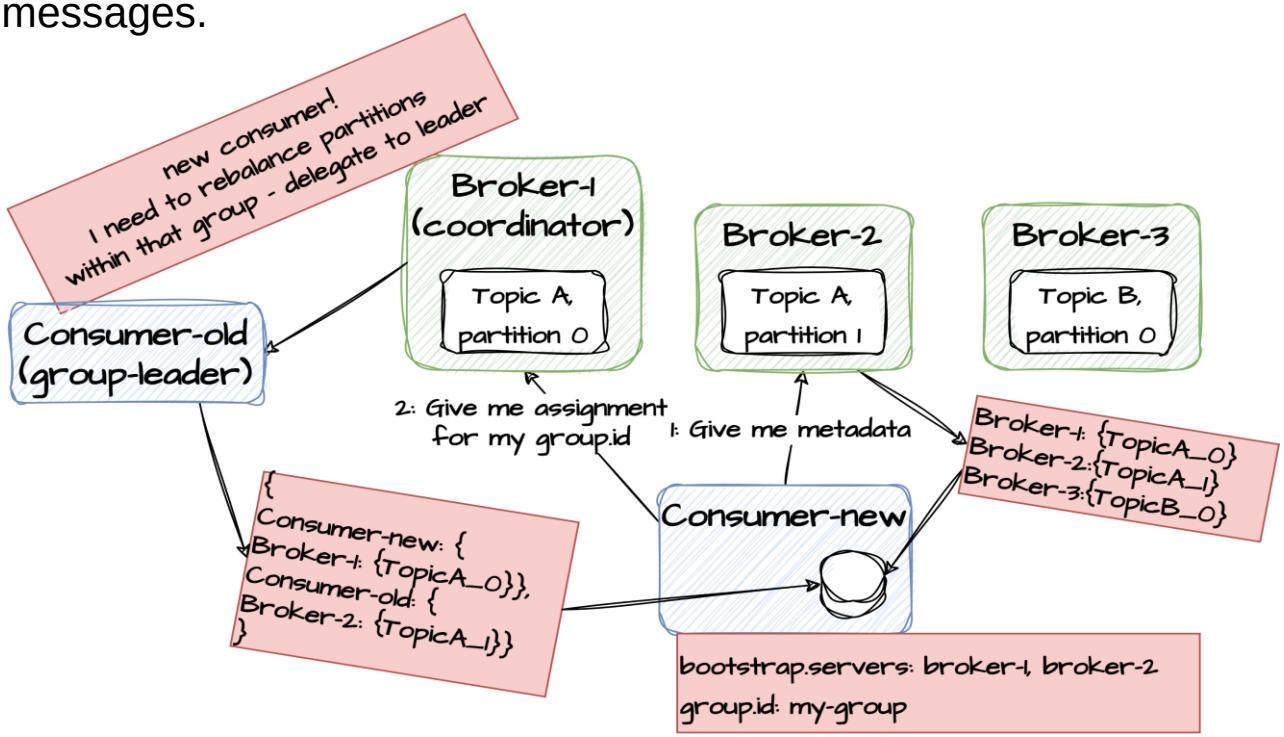
Consumer

Bootstrapping

Lund&Bendsen
developing developers

During initialization, the consumer must specify a bootstrap URL – a list of brokers in the cluster and potentially a consumer id, group id along with a deserialization class. One of the brokers in the bootstrap list is contacted and returns information of how partitions, leaders etc is distributed in the cluster. The consumer contacts the coordinator of the topics that it is interested in. The group coordinator+leader initiates a rebalancing process to distribute partitions evenly among the consumers. The consumer is informed about the partitions that it is assigned and what offset to consume from. This information is cached in the consumer and only renewed if a repartitioning event happens. With this bootstrap information in place, the consumer can target the right brokers when consuming messages.

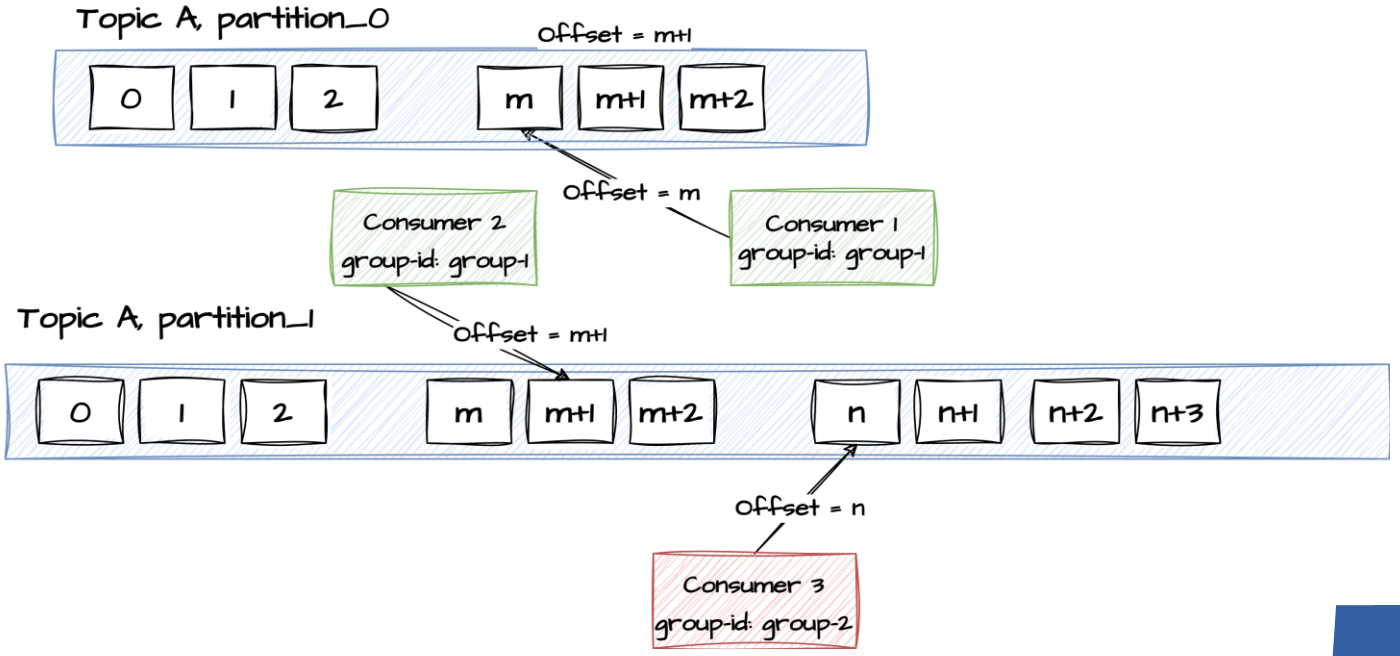


Consuming

Lund&Bendsen
developing developers

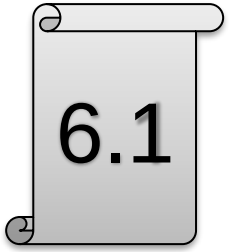
After the bootstrapping process has completed, the consumer is now ready to consume from the partitions that it has been assigned to. If no offset was provided, the consumer starts from the tail of the log (The begin strategy can be controlled in the initialization property `auto.offset.reset`).

The consumer goes into an endless loop of polling the partitions it has been assigned for new records. When the consumer receives a batch of records it is told the last offset it has now consumed. This is the offset from where it will request the next batch.



Exercise

Lund&Bendsen
developing developers



Create a consumer

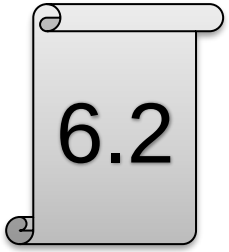
- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --topic my-topic**
- Open a new PowerShell and navigate to <project-folder>/docker
- Start a consumer and type string messages.
- type **.\kafka-producer.bat --topic my-topic**
- Go back to the consumer terminal and see that they were echoed there

Exercise

Lund&Bendsen
developing developers

Create another consumer

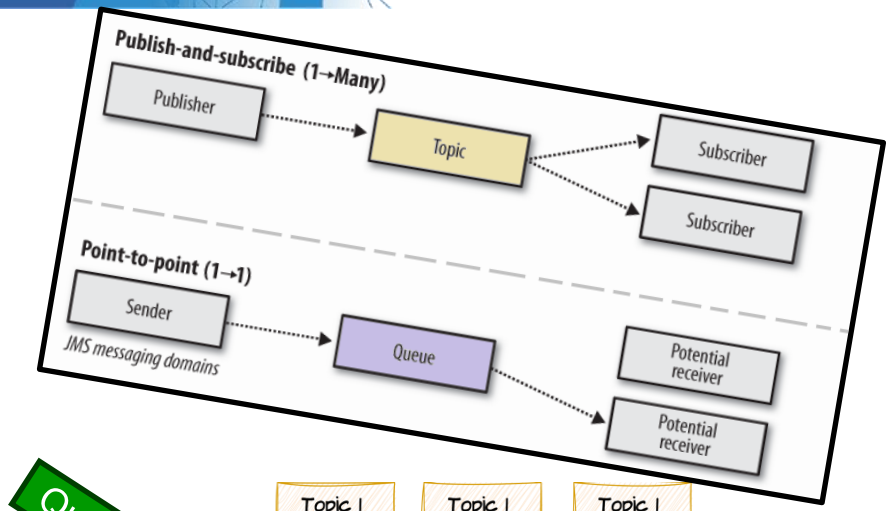
- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --topic my-topic –from-beginning**
- This will start a new consumer for the same topic.
- The consumer starts consumption from the beginning of the topic
- In the producer terminal enter a few more messages and see that they are echoed in both producer terminal windows

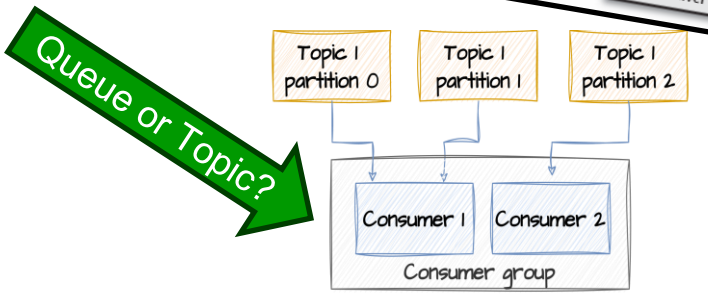


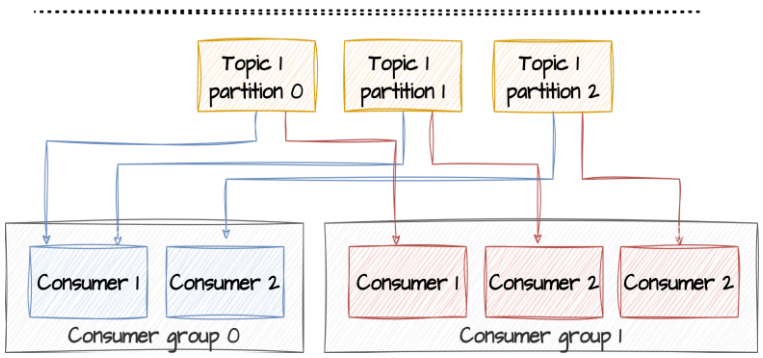
Consumer groups

Lund&Bendsen
developing developers

Consumer groups allow you to parallelize the consumption of data, dividing the workload among multiple consumer instances. Each consumer within a group reads from a unique subset of partitions within a Kafka topic. This division ensures that each piece of data in a topic is processed by only one consumer within the group, maintaining consistency and load balancing. Consumers can join a group by using the same group.id.







- Kafka assigns partitions to the consumers in the same group.
- Each partition is consumed by exactly one consumer in the group.
- Each consumer within a group reads from exclusive partitions.
- One consumer can consume multiple partitions.
- It cannot have more consumers than partitions. Otherwise, some consumers will be inactive state.

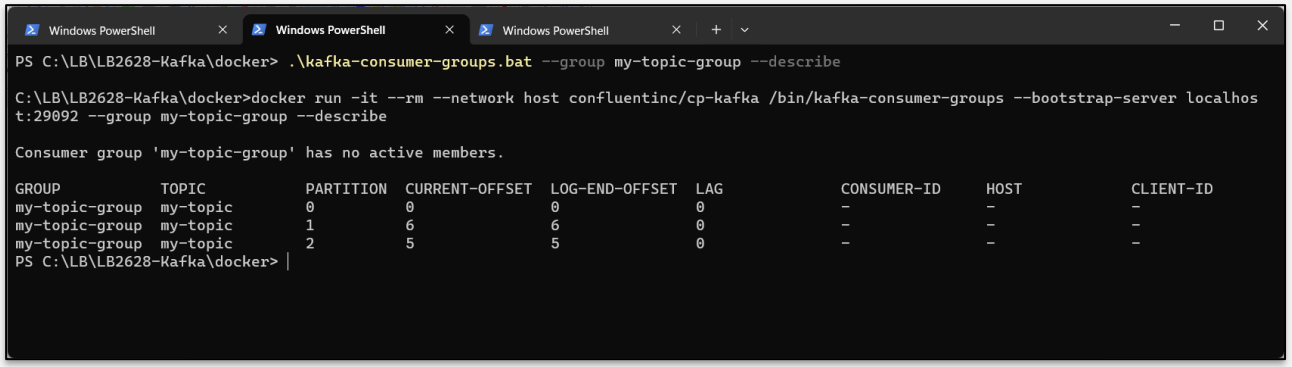
Exercise

Lund&Bendsen
developing developers

6.3

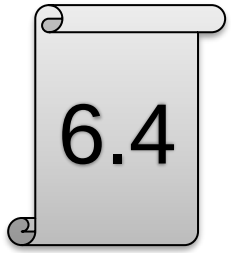
Create consumer group

- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --topic my-topic --group my-group**
- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --topic my-topic --group my-group**
- Open PowerShell and navigate to <project-folder>/docker
- type **.\kafka-producer.bat --topic my-topic**)
- Go back to the consumer terminal and see that they were echoed there. Note that messages are only processed by one consumer in the same group
- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer-groups.bat –list** to see all consumer groups
- Type **.\kafka-consumer-groups.bat --group my-group –describe**
- You should see two consumers in that group



Exercise

Lund&Bendsen
developing developers



Reset consumer group offsets

- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --group my-group --topic my-topic --from-beginning**
- A consumer with a consumer group is created
- Open another PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-producer-perf-test.bat --topic my-topic --num-records 100 --record-size 10 --throughput 10**
- A producer is producing a number of records to the topic
- Open another PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer-groups.bat --group my-group –describe**
- You should see all the topics + partitions and its offsets for that group
- Stop the consumer (else you get a warning when resetting)
- Type **.\kafka-consumer-groups.bat --group my-group --to-earliest --reset-offsets --execute --all-topics**
- This will reset all offsets in the consumer group back to earliest
- Run the consumer group describe command again to see that offsets have been reset
- Type **.\kafka-consumer-groups.bat --group my-group --reset-offsets --shift-by 1 --execute --all-topics**
- Run the consumer group describe command again to see that offsets have been incremented
- This is where new consumers in that group will pick up from when started

Consumer offset

Lund&Bendsen
developing developers

Offset management in Apache Kafka allows consumers to keep track of which messages they have already consumed and ensures that they can resume processing from the correct position in the event of a restart or failure.

An offset is a unique identifier assigned to each message within a Kafka partition. It represents the position or sequence number of a message within the partition. Offsets start from 0 for the first message and increment by one for each subsequent message.

Consumer Offset:

- Each Kafka consumer maintains its own offset for each partition it consumes. This offset indicates the next message that the consumer intends to read from a specific partition.
- It is the responsibility of the consumer to keep track of and manage its own offsets.

Check it out in the topics

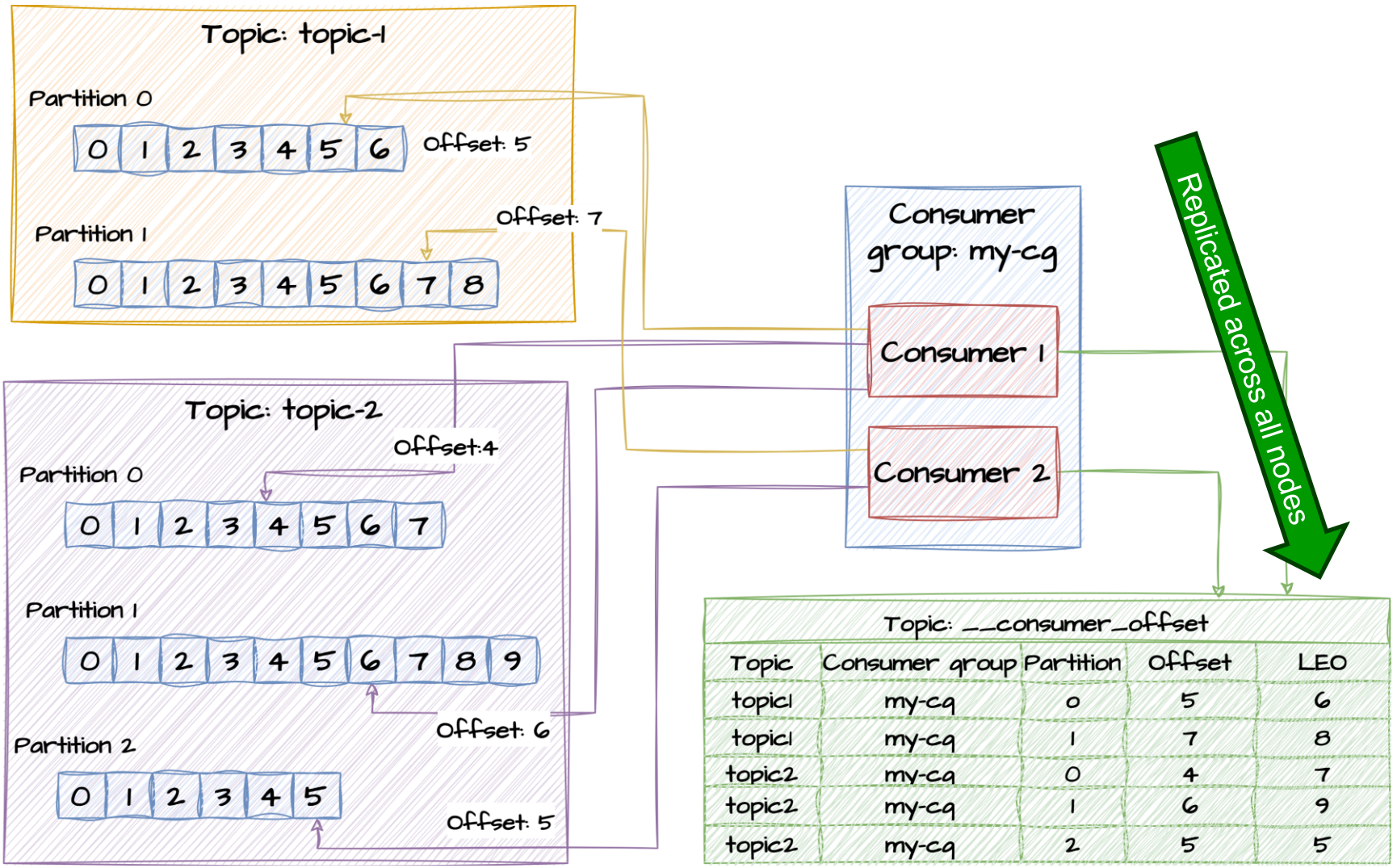
Offset Committing:

- Kafka consumers can commit their current offset to a Kafka topic (__consumer_offsets) or an external storage system. Committing an offset signifies that the consumer has successfully processed messages up to and including that offset.
- Offset committing is essential for enabling consumers to resume processing from the last successfully consumed message, even if the consumer is restarted or replaced.
- Offsets can be committed automatically or manually

We don't have to use Kafkas offset management

Consumer offset – using Kafka topic

Lund&Bendsen
developing developers



Topic: __consumer_offset

Topic	Consumer group	Partition	Offset	LEO
topic1	my-cg	0	5	6
topic1	my-cg	1	7	8
topic2	my-cg	0	4	7
topic2	my-cg	1	6	9
topic2	my-cg	2	5	5

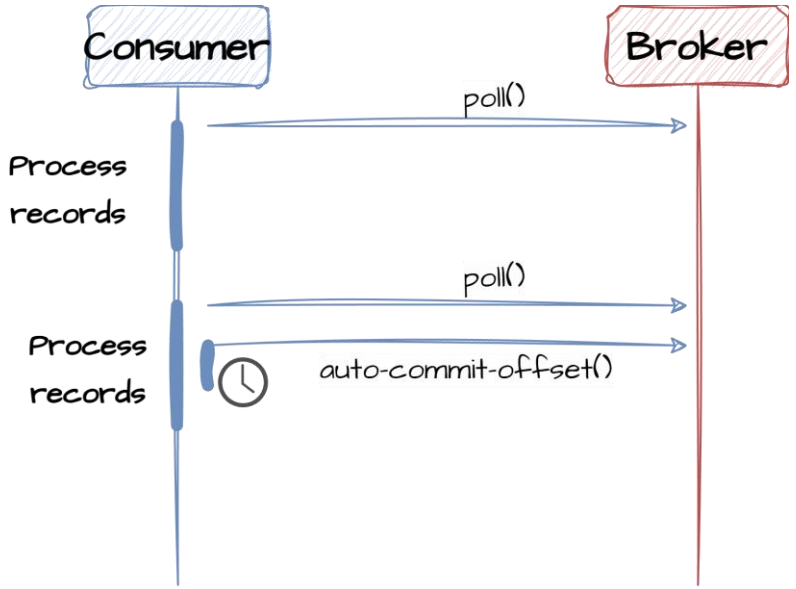
Auto-commit offsets

Lund&Bendsen
developing developers

Auto-commit hands over the responsibility of committing offsets to the consumer. The consumer is configured to do auto committing using these properties:

`enable.auto.commit`, set to true to enable auto commit

`auto.commit.interval.ms`, defines the commit interval



When auto commit is enabled, the producer will commit the last consumed offset with a fixed frequency independent of the processing of the records.

The problem with this approach is that if the application fails to process some of the messages or crashes during processing, the auto-commit may already have committed the latest offset. This could cause a potential data loss. Not recommended in production

Consumer - auto commit

Lund&Bendsen
developing developers

```
Properties props = new Properties();
props.put("enable.auto.commit", ”true");
props.put("auto.commit.interval.ms", ”5000");

try (KafkaConsumer<String, Supplier> consumer = new KafkaConsumer<>(props)) {
   consumer.subscribe(Arrays.asList(topicName));
   while (true) {
      ConsumerRecords<String, Supplier> records = consumer.poll(100);
      for (ConsumerRecord<String, Supplier> record : records) {
         ...
      }
}
} catch (Exception ex) {
} finally {
   consumer.close();
}
```

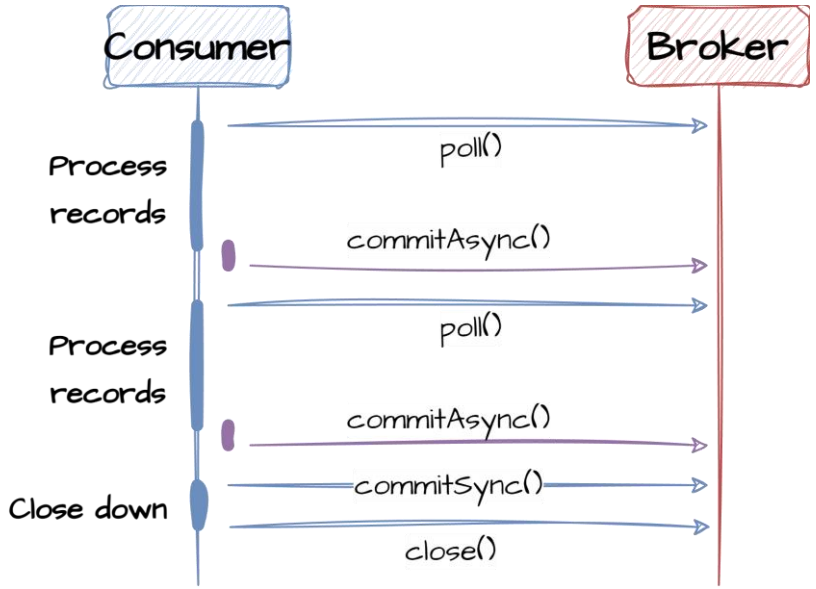
Auto-commit every 5 sec

Manually commit offsets

Lund&Bendsen
developing developers

Manually commit of offsets allows the application to control what offsets are communicated back to Kafka as committed. By setting `enable.auto.commit` to false, the consumer stops the periodic auto-commit process. There are two ways to do manual commit.

One is a synchronous call that blocks until the broker confirms that the offset has been committed and the other is an async call. The async has very little performance impact on the application but offers no guarantee of success/failure whereas the sync version offers high guarantee of success/failure with some performance impacts.



Typically, you would use a combination of the two when designing the offset commit strategy.

Be careful with the sync frequency (long processing latency) – the group coordinator may think the consumer is not responding!

Why is that a problem?

Consumer – manual commit

Lund&Bendsen
developing developers

```
Properties props = new Properties();
props.put("enable.auto.commit", "false");

try (KafkaConsumer<String, Supplier> consumer = new KafkaConsumer<>(props)) {
   consumer.subscribe(Arrays.asList(topicName));
   while (true) {
      ConsumerRecords<String, Supplier> records = consumer.poll(100);
      for (ConsumerRecord<String, Supplier> record : records) {
         ...
      }
      consumer.commitAsync();
   }
} catch (Exception ex) {
} finally {
   consumer.commitSync();
   consumer.close();
}
```

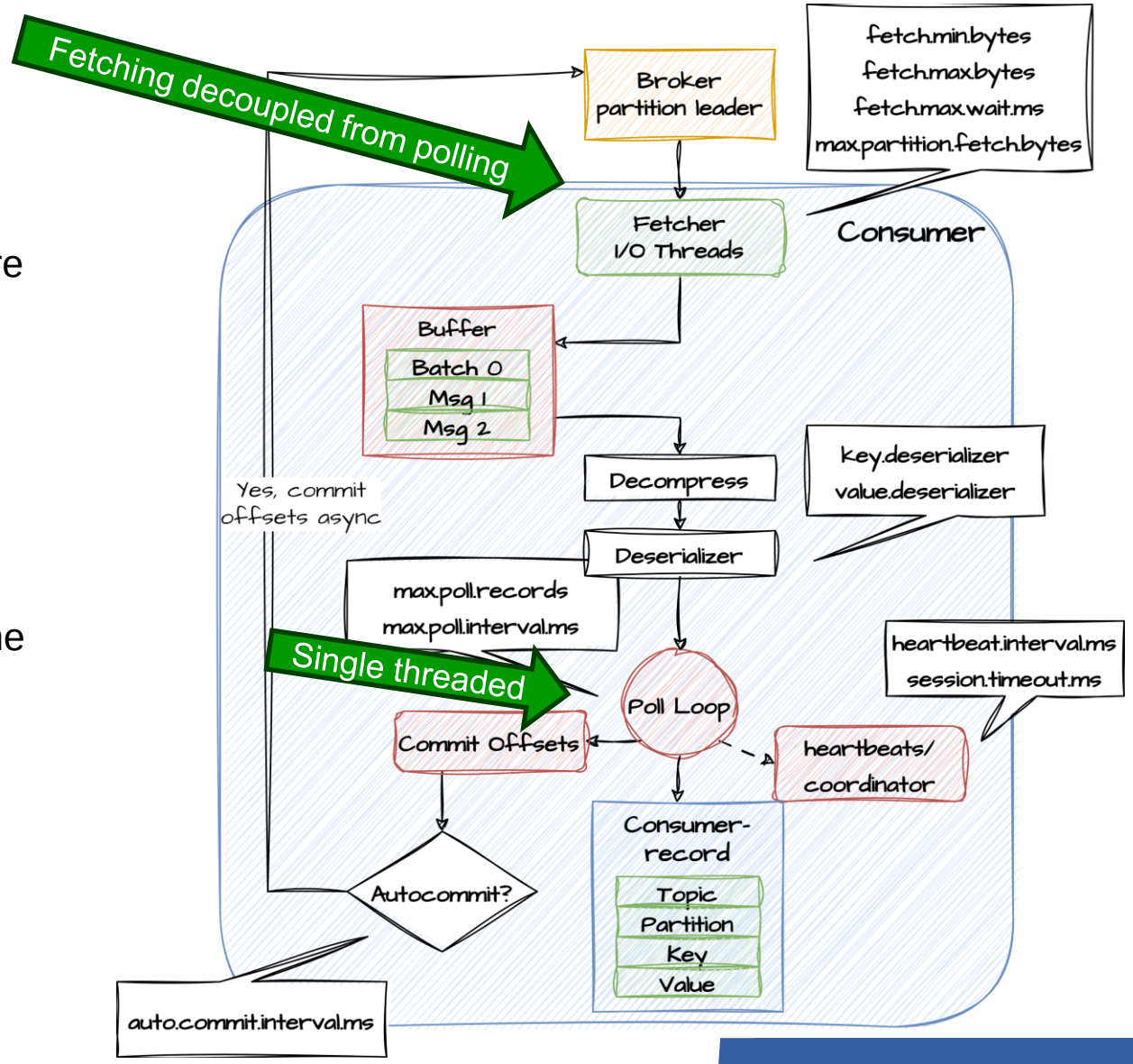
Non-blocking. No commit guarantee

Blocking. Guaranteed commit

Consumer internal

Lund&Bendsen
developing developers

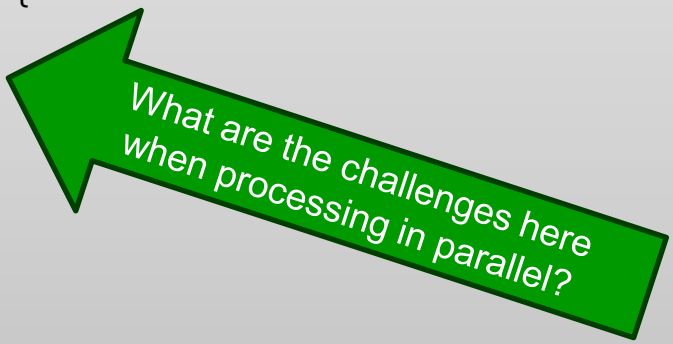
- Fetcher loads batches from the partition leaders and stores them in an in-memory buffer.
- Poll loop requests a number of batches from the buffer. Batches are decompressed and deserialized
- Poll loop checks if auto commit is enabled - if so, is it time to do an auto-commit call?
- Heart beats are regularly sent to partition coordinator
- Consumer records are passed to the consuming application.



Consumer – Parallel processing

Lund&Bendsen
developing developers

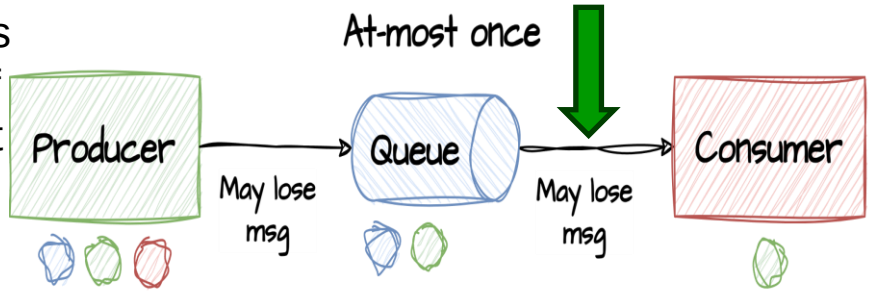
```
ExecutorService executorService = Executors.newFixedThreadPool(5);

try (KafkaConsumer<String, Supplier> consumer = new KafkaConsumer<>(props)) {
   consumer.subscribe(Arrays.asList(topicName));
   while (true) {
      ConsumerRecords<String, Supplier> records = consumer.poll(100);
      for (ConsumerRecord<String, Supplier> record : records) {
         executorService.submit(() -> {
            processRecord(record);
         }
      }
   }
}
} catch (Exception ex) {
} finally {}
```



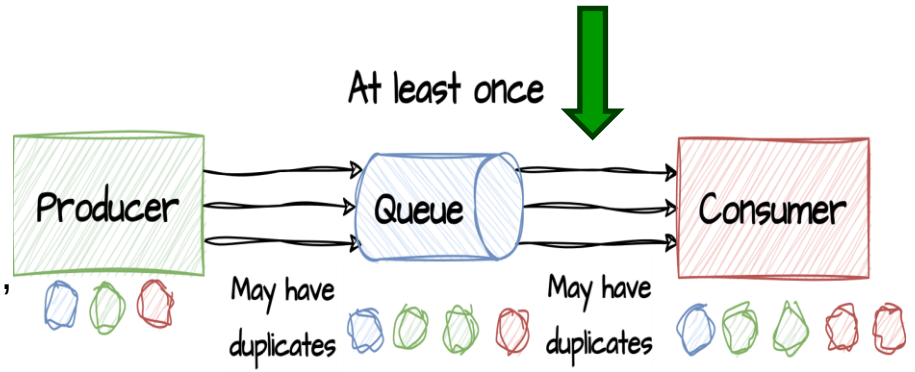
Consumer guarantee

Lund&Bendsen
developing developers

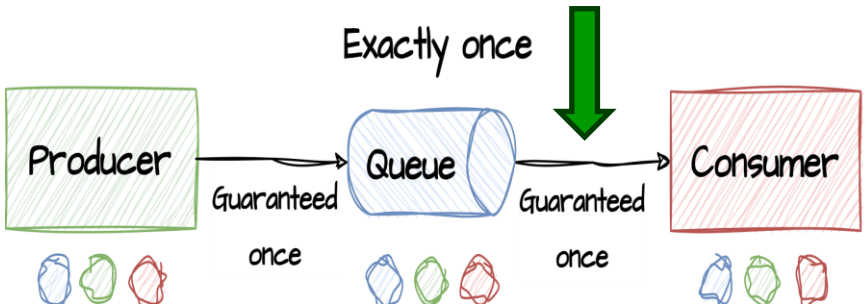
At most once - Consumer reads a set of messages, saves its position in the log, and then processes the messages. If the consumer process crashes after saving its position, but before saving the output of its message processing, the consumer that takes over processing would start at the saved position and the messages prior to that position would not be processed.



At least once - This means a consumer reads a set of messages, processes them, and then saves its position. If the consumer process crashes after processing messages, but before saving its position, the new process that takes over may process some messages a second time.



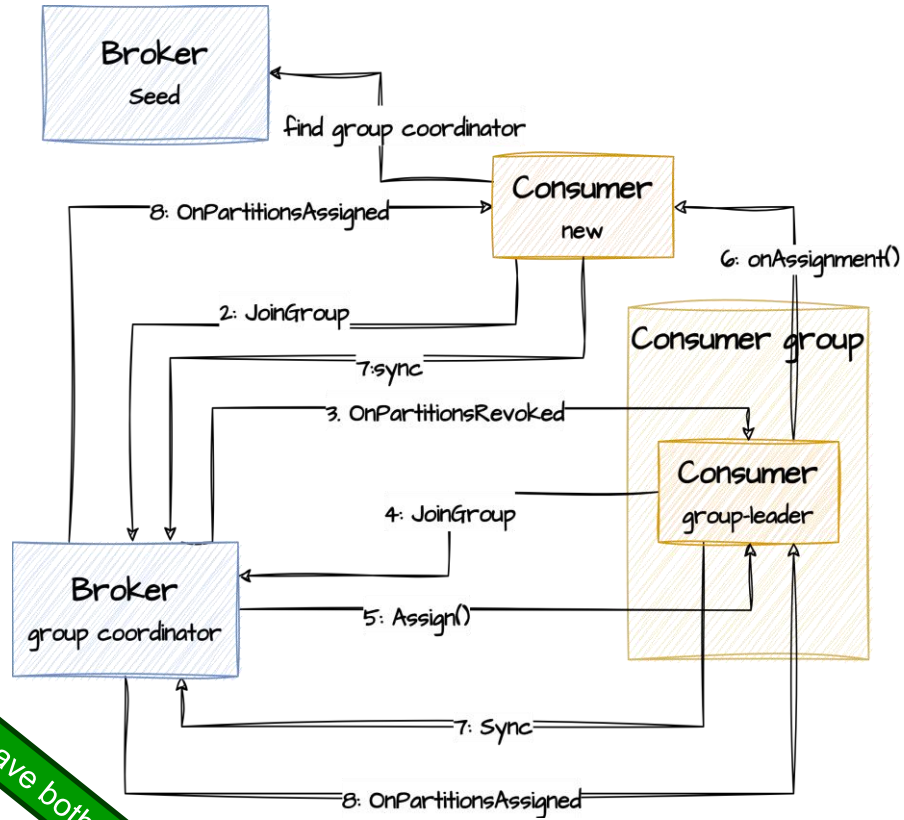
Exactly once - When consuming from a Kafka topic and producing to another topic, Kafka leverages transactional producer capabilities. The consumer's position is stored as a message in a topic, so offset data in written to Kafka in the same transaction as when processed data is written to the output topics.



Rebalancing partitions

Lund&Bendsen
developing developers

- Reasons for a rebalancing partitions between consumers can be consumers leaving/joining the group or consumers stop sending heartbeats. It could also be new brokers leaving/joining the cluster.
- The group coordinator is the broker which receives heartbeats(or polling for message) from all consumers within the consumer group and decides when to initiate rebalancing.
- The group coordinator decides on a group leader (consumer) to do the partition assignment. Assignments get communicated to all other members in the group through the coordinator
- Internally in the consumer, partition assignment is handled by a class implementing the `PartitionAssignor` interface
- A callback interface `ConsumerRebalanceListener` is available for the application to be notified about the different steps in the repartitioning process



Why have both a coordinator and a leader?

Assigning partitions

Lund&Bendsen
developing developers

The `PartitionAssignor` interface is used for internal communication between consumers when decisions on how partitions should be distributed must be made. This interface has callback methods for regular members and for the group leader that makes the assignment decision

Kafka provides several built-in partition assignors, such as the `RangeAssignor` and `RoundRobinAssignor`, which determine the rules for assigning partitions to consumers.

```
public interface PartitionAssignor {

   // Callback on any member to get Subscription for all topics the consumer is interested in
   Subscription subscription(Set<String> topics);

   // Invoked on the group leader to decide partitinon assignment pr member
   Map<String, Assignment> assign(Cluster metadata, Map<String, Subscription> subscriptions);

   // Callback on all members to communicate what partitions have been assigned
   void onAssignment(Assignment assignment);

   // Unique name of the consumer
   String name();
}
```

Rebalancing callback

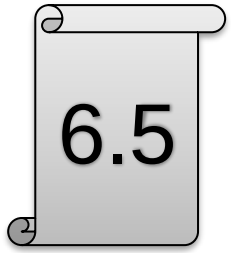
Lund&Bendsen
developing developers

The `ConsumerRebalanceListener` interface is used as a callback to the application. This allows the application to respond to rebalancing events before and after rebalancing happens. The below code persists its offsets to external storage and retrieves it again later

```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
   private Consumer<?,?> consumer;
   public SaveOffsetsOnRebalance(Consumer<?,?> consumer) {
      this.consumer = consumer;
   }
   public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
      // save the offsets in an external store using some custom code not described here
      for(TopicPartition partition: partitions) {
         saveOffsetInExternalStore(consumer.position(partition));
      }
   }
   public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
      // read the offsets from an external store using some custom code not described here
      for(TopicPartition partition: partitions) {
         consumer.seek(partition, readOffsetFromExternalStore(partition));
      }
   }
}
```

Exercise

Lund&Bendsen
developing developers



Partition reassignment

- Open PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --group my-group --topic my-topic --from-beginning**
- Open another PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer-groups.bat --group my-group –describe**
- See what partitions have been assigned to what clients
- Open another PowerShell and navigate to <project-folder>/docker
- Type **.\kafka-consumer.bat --group my-group --topic my-topic --from-beginning**
- Type **.\kafka-consumer-groups.bat --group my-group –describe**
- See that partitions have been reassigned

Consumer configuration

Lund&Bendsen
developing developers

fetch.min.bytes

The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. The default setting of 1 byte means that fetch requests are answered as soon as that many byte(s) of data is available or the fetch request times out waiting for data to arrive.

group.id

A unique string that identifies the consumer group this consumer belongs to. This property is required if the consumer uses either the group management functionality by using subscribe(topic) or the Kafka-based offset management strategy.

session.timeout.ms

The timeout used to detect client failures when using Kafka's group management facility. The client sends periodic heartbeats to indicate its liveness to the broker. If no heartbeats are received by the broker before the expiration of this session timeout, then the broker will remove this client from the group and initiate a rebalance. Note that the value must be in the allowable range as configured in the broker configuration by group.min.session.timeout.ms and group.max.session.timeout.ms.

Consumer configuration

Lund&Bendsen
developing developers

allow.auto.create.topics

Allow automatic topic creation on the broker when subscribing to or assigning a topic. A topic being subscribed to will be automatically created only if the broker allows for it using auto.create.topics.enable broker configuration.

auto.offset.reset

What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted):

- earliest: automatically reset the offset to the earliest offset
- latest: automatically reset the offset to the latest offset
- none: throw exception to the consumer if no previous offset is found for the consumer's group anything else: throw exception to the consumer.

enable.auto.commit

If true the consumer's offset will be periodically committed in the background.

Consumer configuration

Lund&Bendsen
developing developers

fetch.max.bytes

The maximum amount of data the server should return for a fetch request. Records are fetched in batches by the consumer, and if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that the consumer can make progress. As such, this is not a absolute maximum. The maximum record batch size accepted by the broker is defined via message.max.bytes (broker config) or max.message.bytes (topic config). Note that the consumer performs multiple fetches in parallel.

max.poll.interval.ms

The maximum delay between invocations of poll() when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If poll() is not called before expiration of this timeout, then the consumer is considered failed and the group will rebalance in order to reassign the partitions to another member. For consumers using a non-null group.instance.id which reach this timeout, partitions will not be immediately reassigned. Instead, the consumer will stop sending heartbeats and partitions will be reassigned after expiration of session.timeout.ms. This mirrors the behavior of a static consumer which has shutdown.

Consumer configuration

Lund&Bendsen
developing developers

max.poll.records

The maximum number of records returned in a single call to poll(). Note, that max.poll.records does not impact the underlying fetching behavior. The consumer will cache the records from each fetch request and returns them incrementally from each poll.

partition.assignment.strategy

A list of class names or class types, ordered by preference, of supported partition assignment strategies that the client will use to distribute partition ownership amongst consumer instances when group management is used. Available options are:

- **RangeAssignor**: Assigns partitions on a per-topic basis.
- **RoundRobinAssignor**: Assigns partitions to consumers in a round-robin fashion.
- **StickyAssignor**: Guarantees an assignment that is maximally balanced while preserving as many existing partition assignments as possible.
- **CooperativeStickyAssignor**: Follows the same StickyAssignor logic, but allows for cooperative rebalancing.