Apache Kafka:

Tips and Tricks I wish I had known before.

Arnaud Esteve
arnaud.esteve@gmail.com


twitter.com/arnaudesteve


github.com/aesteve


linkedin.com/in/arnaud-esteve-74283745

Legend

- 🚩 Don't
- ⚠️ Be careful
- 💡 Tip
- 📏 Rule (of thumb)
- 🧐 Keep in mind
- 🤔 Why would I?

Context

*Past exp. w/ Kafka at **Decathlon***

*+ **Confluent** customers' Support + FAQs*

= Lots of stories merged into a "**we**"

Heard about Kafka?

Ever tried Kafka?

Deployed in production?

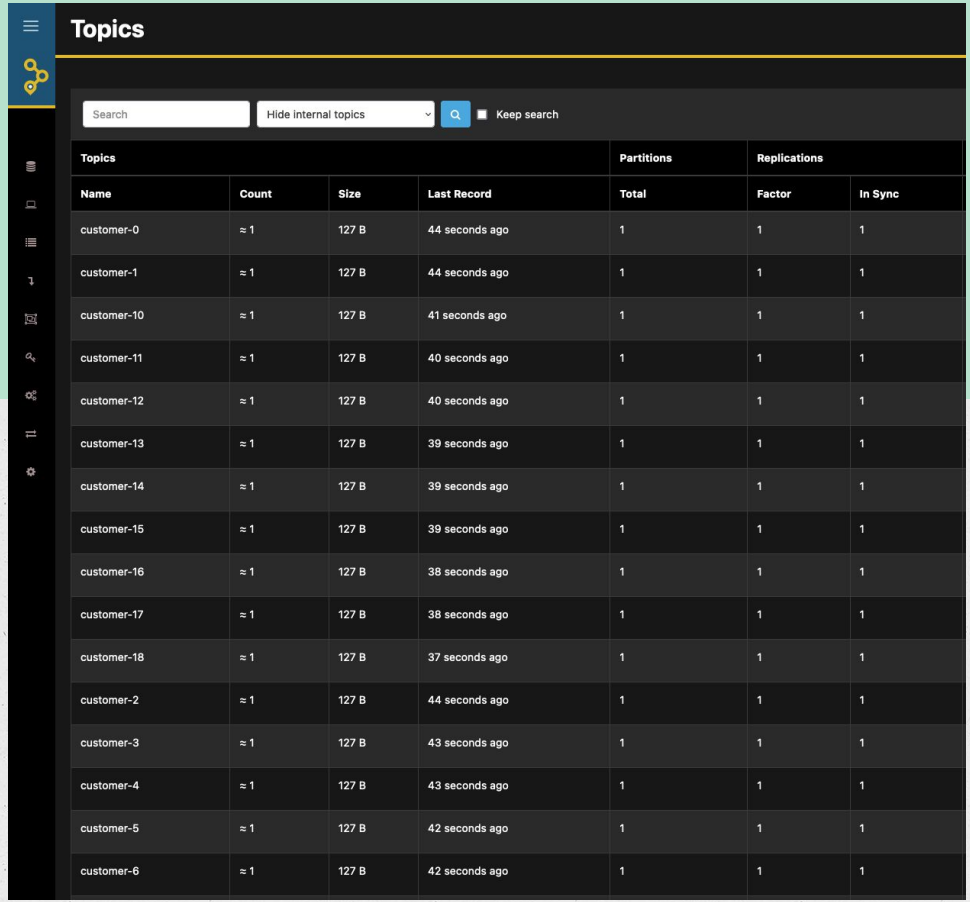
server.properties

```
…
auto.topics.create.enable=?
…
```

Can you spot the problem?

```
producer = KafkaProducer(value_serializer=lambda m: json.dumps(m).encode('ascii'))
for customer in customers:
   producer.send("customer-{}".format(customer),
                 {"amount": 1234, "date": "2023-01-05T22:02:00.000Z"},
                 b'orders'
                 )
```

What did you expect?



Topics				Partitions	Replications	
Name	Count	Size	Last Record	Total	Factor	In Sync
customer-0	≈ 1	127 B	44 seconds ago	1	1	1
customer-1	≈ 1	127 B	44 seconds ago	1	1	1
customer-10	≈ 1	127 B	41 seconds ago	1	1	1
customer-11	≈ 1	127 B	40 seconds ago	1	1	1
customer-12	≈ 1	127 B	40 seconds ago	1	1	1
customer-13	≈ 1	127 B	39 seconds ago	1	1	1
customer-14	≈ 1	127 B	39 seconds ago	1	1	1
customer-15	≈ 1	127 B	39 seconds ago	1	1	1
customer-16	≈ 1	127 B	38 seconds ago	1	1	1
customer-17	≈ 1	127 B	38 seconds ago	1	1	1
customer-18	≈ 1	127 B	37 seconds ago	1	1	1
customer-2	≈ 1	127 B	44 seconds ago	1	1	1
customer-3	≈ 1	127 B	43 seconds ago	1	1	1
customer-4	≈ 1	127 B	43 seconds ago	1	1	1
customer-5	≈ 1	127 B	42 seconds ago	1	1	1
customer-6	≈ 1	127 B	42 seconds ago	1	1	1

What happened

```
for customer in customers:
    producer = KafkaProducer(value_serializer=lambda m: json.dumps(m).encode('ascii'))
    producer.send("customer-{}".format(customer),
                  {"amount": 1234, "date": "2023-01-05T22:02:00.000Z" },
                  b'orders'
                  )
```

🚩 **No** ⇒ `auto.topics.create.enable=`**`false`**

(and yes, connectors do create topics…)

🚩 Human-created? Not necessarily better

💡 **Automate everything** ⇒ integrate in CI/CD

- 📏 PRs, reviews, etc. ⇒ Usual IaC workflow
- 💡 **Static code checks**:
  - retention policy / period
  - nb. of partitions
  - naming

How should we name our topics?

Can you spot the problem?

```
try (var producer = new KafkaProducer<String, String>(props)) {
   // legacy topic, before conventions
   producer.send(new ProducerRecord<>("preprod_customer_orders", "key", "value"));
   // new topic
   producer.send(new ProducerRecord<>("preprod.customer.orders", "key", "value"));
}
```

🚩 **Avoid** dynamic parts:

Environment variables,

System properties,

Computed fields (date, etc.)

⚠️ Dots & Underscores collide!

💡 Pick a **convention** & stick to it

⚠️ with schema registry conventions

💡 Think **Wildcards** for ACLs (`preprod.customers.*`)

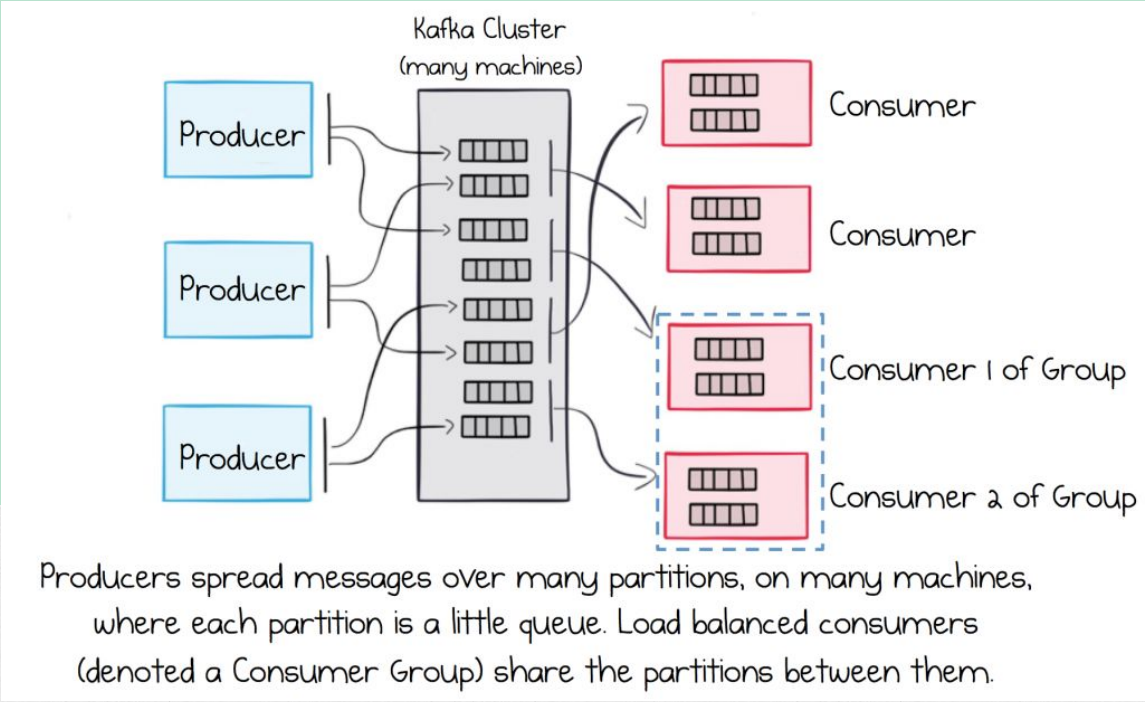
💡 **DOCUMENT** everything

+ automated checks (CI/CD + regex, ...)

👉 https://medium.com/@kiranprabhu/kafka-topic-naming-conventions-best-practices-6b6b332769a3

👉 https://devshawn.com/blog/apache-kafka-topic-naming-conventions/

What should I use as a key?



- 💡 The producer sets the partition! (=> `Partitioner`)
- 💡 By default: `hash(key) % partitionCount`
- 💡 **Same key ⇒ Same partition**

Test.java

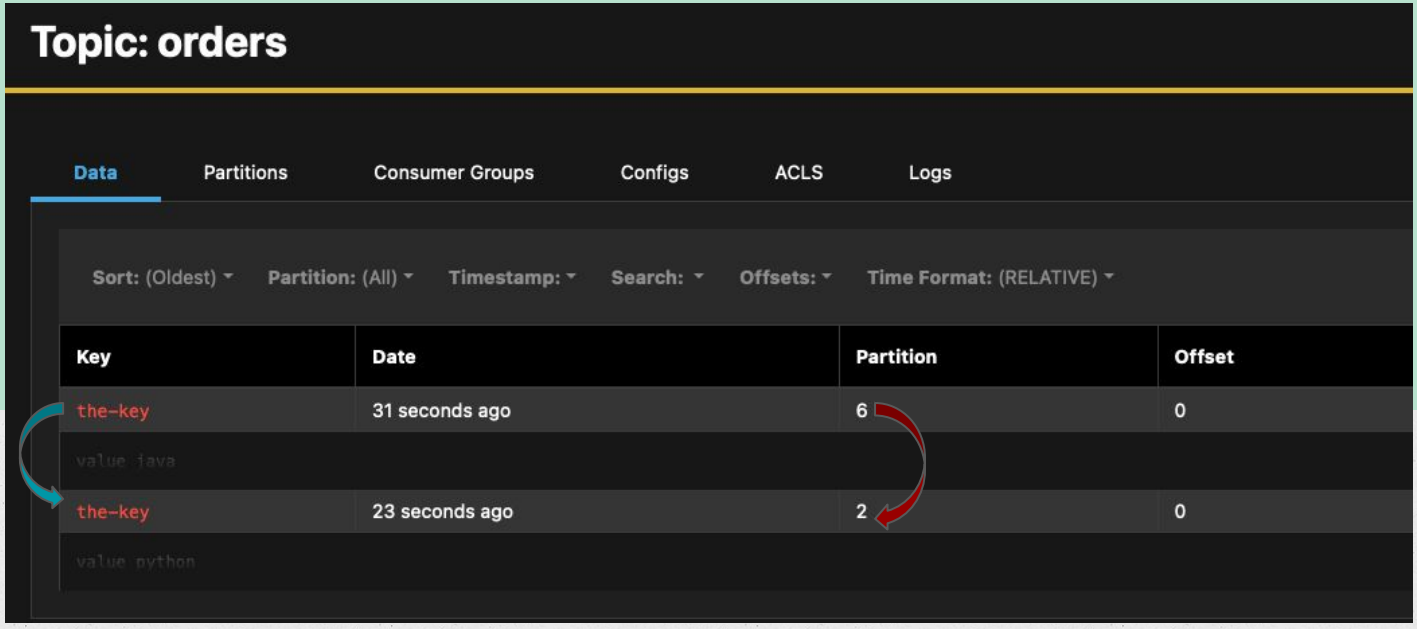
```java
var topicName = "orders";
try (var admin = AdminClient.create(props)) {
    var topic = new NewTopic(topicName, 15, (short) 1); // 15 partitions
    admin.createTopics(Collections.singleton(topic)).all().get(); // wait for completion
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

    try (var producer = new KafkaProducer<String, String>(props)) {
        producer.send(new ProducerRecord<>(topicName, "the-key", "value java"));
    }
}
```

test.py

```python
producer = Producer({'bootstrap.servers': 'localhost:9092'})
producer.produce(topic='orders', key='the-key', value='value python')
producer.flush()
```

Wait what?



librdkafka / CONFIGURATION.md ↑ Top

Preview | Code | Blame | 183 lines (178 loc) · 54.9 KB | Raw

					Type: boolean
partitioner	P		consistent_random	high	Partitioner: `random` - random distribution, `consistent` - CRC32 hash of key (Empty and NULL keys are mapped to single partition), `consistent_random` - CRC32 hash of key (Empty and NULL keys are randomly partitioned), `murmur2` - Java Producer compatible Murmur2 hash of key (NULL keys are mapped to single partition), `murmur2_random` - Java Producer compatible Murmur2 hash of key (NULL keys are randomly partitioned. This is functionally equivalent to the default partitioner in the Java Producer.), `fnv1a` - FNV-1a hash of key (NULL keys are mapped to single partition), `fnv1a_random` - FNV-1a hash of key (NULL keys are randomly partitioned). *Type: string*

🚩 **AVOID** custom partitioners
(unless you really have to)

🚩 **WATCH OUT** for skew:

"One customer makes up for 90% of the traffic"

🚩 **AVOID** producing from many languages

⚠️ The producer's responsibility

⚠️ Keyless can be an option
(if no ordering needed + skew)

💡 Partitioning => ordering => causality

💡 Determinism for consumers

=> Same key = **causal** to one another

What are cleanup policies?

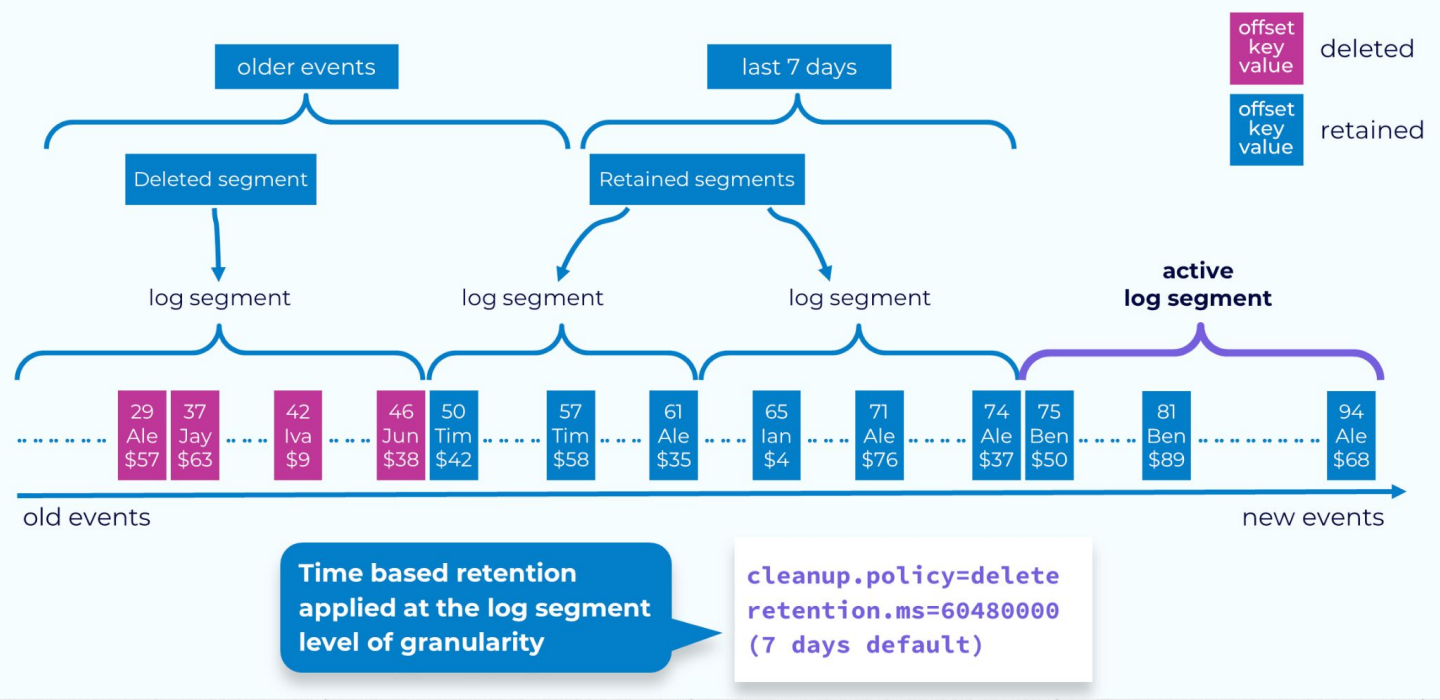
What is going to happen to yesterday's records?

```
try (var admin = AdminClient.create(props)) {
   var topic = new NewTopic("orders", 6, (short) 3);
   topic.configs(Map.of(
           TopicConfig.CLEANUP_POLICY_CONFIG,"compact",
           TopicConfig.RETENTION_MS_CONFIG, "86400000"
   ));
   admin.createTopics(Collections.singleton(topic));
}
```

COMPACT + DELETE

```
try (var admin = AdminClient.create(props)) {
   var topic = new NewTopic("orders", 6, (short) 3);
   topic.configs(Map.of(
           TopicConfig.CLEANUP_POLICY_CONFIG,"compact,delete",
           TopicConfig.RETENTION_MS_CONFIG, "86400000"
   ));
   admin.createTopics(Collections.singleton(topic));
}
```

But still, watch out!



Let’s tune the log cleaner, then!

```
try (var admin = AdminClient.create(props)) {
    // 1 partition (easier to observe)
    var topic = new NewTopic(topicName, 1, (short) 1);
    topic.configs(Map.of(
            TopicConfig.CLEANUP_POLICY_CONFIG, "compact,delete",
            TopicConfig.RETENTION_MS_CONFIG, "86400",
            TopicConfig.SEGMENT_MS_CONFIG, "100" // What a good idea 🙄
    ));
    admin.createTopics(Collections.singleton(topic)).all().get();
}

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
try (var producer = new KafkaProducer<String, String>(props)) {
    var i = 0;
    while (++i < 1_000) {
        producer.send(new ProducerRecord<>(topicName, "order-id-" + i, "the order"));
        Thread.sleep(100);
    }
}
```

Yikes...

```
[root@broker appuser]# lsof | wc -l
22606
[root@broker appuser]# lsof | wc -l
31119
[root@broker appuser]# lsof | wc -l
32905
[root@broker appuser]# lsof | wc -l
41119
[root@broker appuser]# lsof | wc -l
43152
[root@broker appuser]# lsof | wc -l
46143
[root@broker appuser]# lsof | wc -l
47977
[root@broker appuser]# lsof | wc -l
56076
```

```
[root@broker appuser]# ls /tmp/kraft-combined-logs/orders-0/
00000000000000001998.index
00000000000000002031.timeindex
00000000000000001998.log
00000000000000002032.index
00000000000000001998.snapshot
00000000000000002032.log
00000000000000001998.timeindex
00000000000000002032.snapshot
00000000000000001999.index
00000000000000002032.timeindex
00000000000000001999.log
00000000000000002033.index
00000000000000001999.snapshot
00000000000000002033.log
00000000000000001999.timeindex
00000000000000002033.snapshot
00000000000000002000.index
00000000000000002033.timeindex
…
```

🚩 **DO NOT**

overtune the `log.cleaner`

🚩 **WATCH OUT** for anti-patterns:

"We must not have any duplicated data"

"We want to reduce the consumption time to … "

"We want to reduce topic size to …"

⚠️ Pick the right `cleanup.policy`

⚠️ Remember the active segment

⚠️ Choose the right `retention.period`

💡 `cleanup.policy`:

`compact` / `delete` or **BOTH**!

💡 `retention.ms`:

It's in the name!

💡 Compaction is an optimistic mechanism

💡 Use a **StateStore** or a **cache** if need be

How many partitions?

📏 Unit of scalability:

more partitions ⇒ more parallelism

📏 Signals to look for:

batch reprocessing “from the origin of time”

very slow / highly parallelizable consumption:

```
forEach(record) { doSomethingSlow() }
```

🧐 Tradeoffs:

more distributed ⇒ more leaders ⇒ longer rebalance

Also, **less-efficient batching**!

💡 **Highly composite** numbers:

6, 12, 24, 48, 60, 120, 180, 240, 360

→ divide workload more evenly

👉 https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster

How can I know upfront?

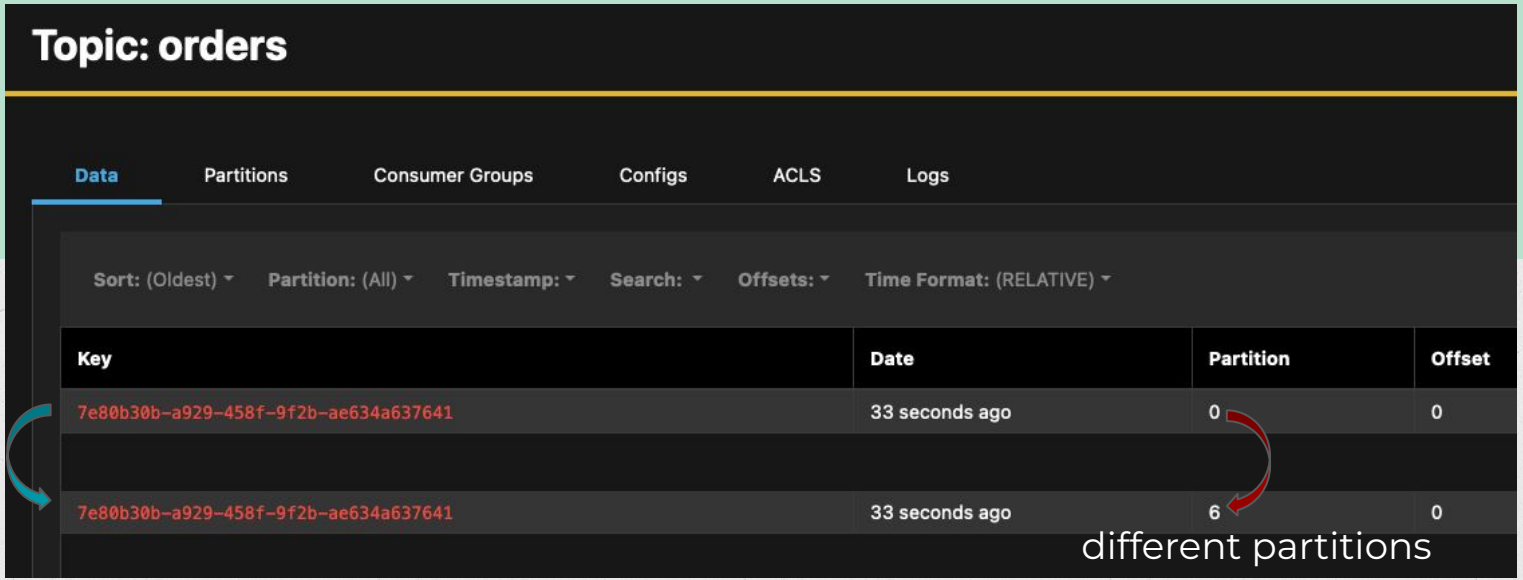
And what if I mess up?

No problem!

```
var topicName = "orders";
var key = UUID.randomUUID().toString();
try (var admin = AdminClient.create(props)) {
   // 3 partitions originally
   var topic = new NewTopic(topicName, 3, (short) 1);
   admin.createTopics(Collections.singleton(topic)).all().get(); // wait for completion

   props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
   props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
   try (var producer = new KafkaProducer<String, Long>(props)) {
       producer.send(new ProducerRecord<>(topicName, key, 1L));
   }
   admin.createPartitions(Map.of(topicName, NewPartitions.increaseTo(10))).all().get();
   try (var producer = new KafkaProducer<String, Long>(props)) {
       producer.send(new ProducerRecord<>(topicName, key, 2L));
   }
}
```

No problem, really?



🚩 **DO NOT** increase the nb. of partitions

Unless you know what you're doing:

=> you don't care about ordering

=> no stateful consumers

💡 Create a **new topic** with more partitions instead

=> replicate data to new topic

=> deploy the use-case requiring more partitions

=> migrate consumers progressively

=> migrate producers

CLIENTS

Can you spot the error?

```
var props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("ack", "all"); // I need the strongest delivery guarantee!!
props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());
var topicName = "orders";
try (var producer = new KafkaProducer<String, String>(props)) {
   producer.send(new ProducerRecord<>(topicName, "the-key", "some value"), (metadata, ex) -> {
       LOG.info(
               "Successfully published to all replicas of partition {} (Offset is {}),"
               metadata.partition(),
               metadata.offset()
       );
   });
}
```

Here it is!

acks

```
var props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("ack", "all");
props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());
var topicName = "orders";
try (var producer = new KafkaProducer<String, String>(props)) {
   producer.send(new ProducerRecord<>(topicName, "the-key", "some value"), (metadata, ex) -> {
       LOG.info(
               "Successfully published to all replicas of partition {} (Offset is {}),"
               metadata.partition(),
               metadata.offset()
       );
   });
}
```

Safer this way

```
var props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.ACKS_CONFIG, "all");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
var topicName = "orders";
try (var producer = new KafkaProducer<String, String>(props)) {
   producer.send(new ProducerRecord<>(topicName, "the-key", "some value"), (metadata, ex) -> {
       LOG.info(
               "Successfully published to all replicas of partition {} (Offset is {})",
               metadata.partition(),
               metadata.offset()
       );
   });
}
```

Constants, constants everywhere

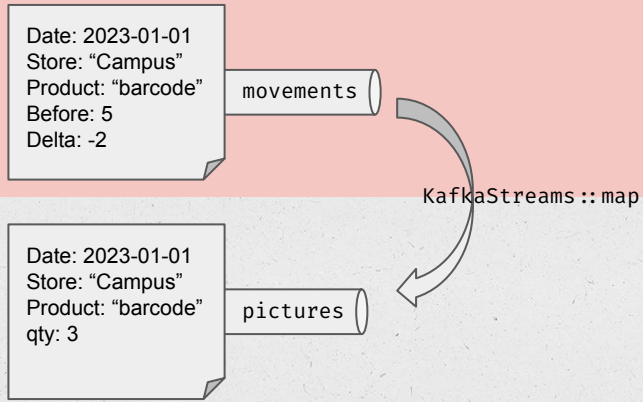
- 📏 Don't waste time with typos 🙏
- 💡 Both for keys and values
  (`EXACTLY_ONCE_V2` , `SASL_PLAIN` , etc.)

`AdminClientConfig`
`SecurityConfig`
`ProducerConfig`
`CommonClientConfig`
`StreamsConfig`
etc.

👉 https://kafka.apache.org/35/javadoc/constant-values.html

The Ops strike back

```
var keySerde = Serdes.String();
var movementSerde = new StockMovementSerde();
var pictureSerde = new StockPictureSerde();
var props = new Properties();
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.APPLICATION_ID_CONFIG, System.getenv("POD_NAME_VERSION"));
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
var builder = new StreamsBuilder();
builder.stream("movements", Consumed.with(keySerde, movementSerde))
        .map((key, movement) -> {
            var qtyLeft = movement.before() + movement.delta(); // trivial computation
            return new KeyValue<>(
                    key
                    new StockPicture(movement.date(), movement.store(), movement.product(), qtyLeft)
            );
        })
        .to("pictures", Produced.with(keySerde, pictureSerde));
try (var app = new KafkaStreams(builder.build(), props)) {
   app.start();
}
```



Guess what...

```
var keySerde = Serdes.String();
var movementSerde = new StockMovementSerde();
var pictureSerde = new StockPicturesSerde();
var props = new Properties();
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.APPLICATION_ID_CONFIG, System.getenv("POD_NAME_VERSION"));
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
var builder = new StreamsBuilder();
builder.stream("stock-movements", Consumed.with(keySerde, movementSerde))
        .map((key, movement) -> {
            var qtyLeft = movement.before() + movement.delta();
            return new KeyValue<>(
                    key
                    new StockPicture(movement.date(), movement.store(), movement.product(), qtyLeft)
            );
        })
        .to("stock-pictures", Produced.with(keySerde, pictureSerde));
try (var app = new KafkaStreams(builder.build(), props)) {
    app.start();
}
```

Changing this

Restarts from here

Mmh this product hasn't been sold since 2020???

🚩 **Avoid** dynamic parts:

Environment variables,

System properties,

Computed fields (date, etc.)

⚠️ Consumer Id, Streams Id

⚠️ Producers if transactional Have an id too

⚠️ `client.id` != `group.id`

💡 Deterministic identifiers for clients

💡 Constant in code, versioned in VCS

💡 Potentially extracted at build time

=> the **code drives identifiers**

Wtf why are these records duplicated?

🚩 **Avoid**

Assuming everything is EO

Setting EOS by default

🧐 Understand delivery semantics

"Know what you're doing"

⚠️ Tradeoff durability / perf.

💡 Set **idempotency** on producers

💡 Connectors => check documentation

💡 Kafka Streams, set:

`StreamsConfig.PROCESSING_GUARANTEE_CONFIG`

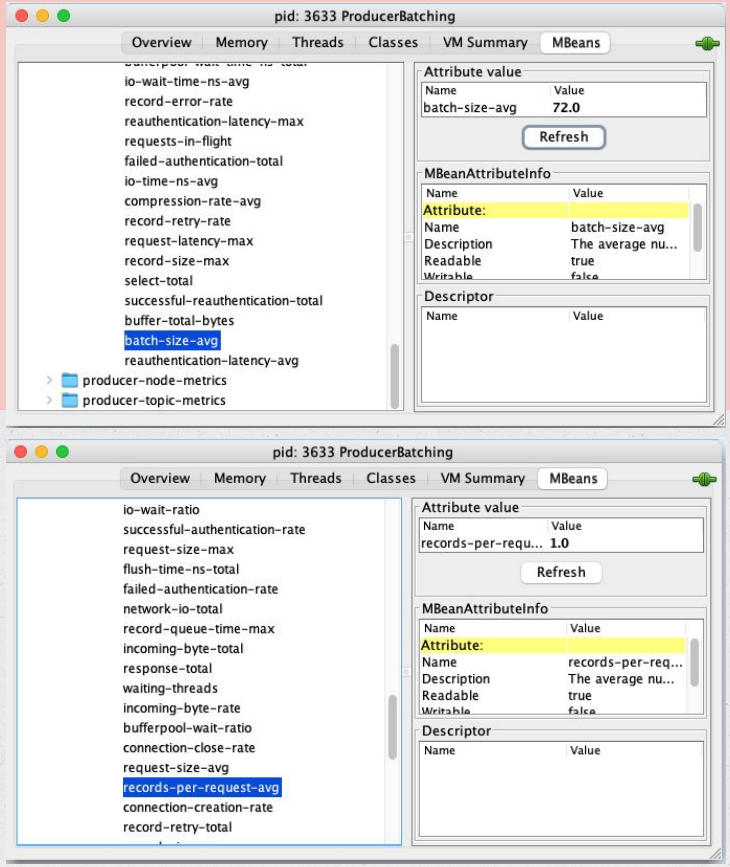
📏 EOS is no magic, only for Kafka-to-Kafka

⇒ design around **At Least Once** where possible

👉 https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-processing-guarantees

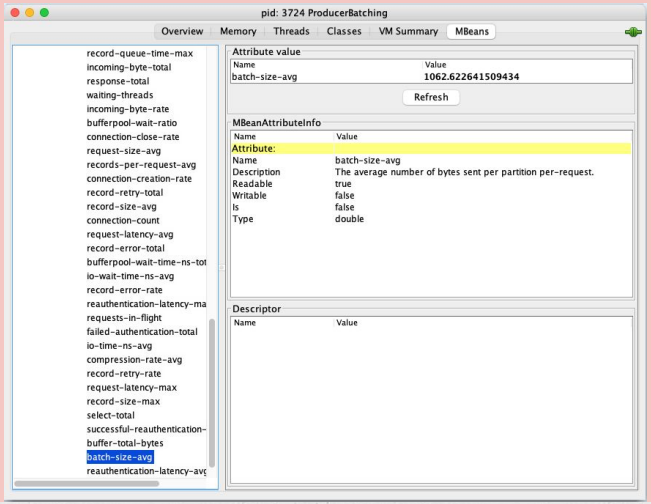
I’m just trying to batch…

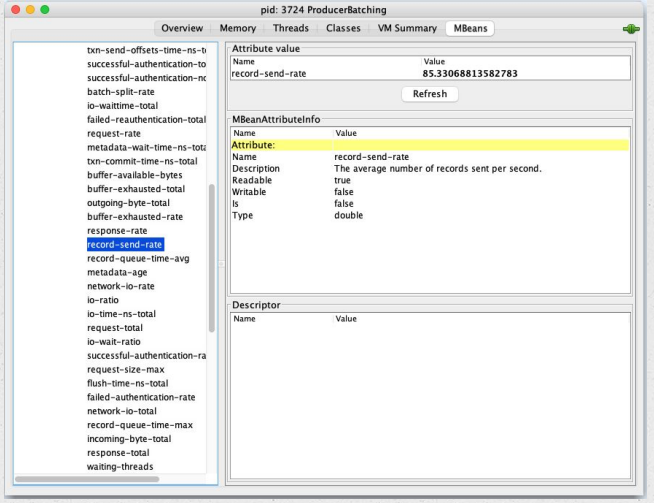
```
var props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.BATCH_SIZE_CONFIG, "10000");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
var topic = "some-topic";
try (var producer = new KafkaProducer<String, String>(props)){
   var i = 0;
   while (true) {
       producer.send(new ProducerRecord<>(topic, Integer.toString(++i)));
       Thread.sleep(10);
   }
} catch (InterruptedException e) {
   throw new RuntimeException(e);
}
```



I’m just trying to batch…

```
var props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.BATCH_SIZE_CONFIG, "10000");
props.put(ProducerConfig.LINGER_MS_CONFIG, "1000");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
var topic = "some-topic";
try (var producer = new KafkaProducer<String, String>(props)){
   var i = 0;
   while (true) {
       producer.send(new ProducerRecord<>(topic, Integer.toString(++i)));
       Thread.sleep(10);
   }
} catch (InterruptedException e) {
   throw new RuntimeException(e);
}
```





Producer batching + compression

💡 Use JMX Metrics to understand what's going on

💡 Increase your chances of **batching**!!!

`linger.ms` > 0

💡 THEN, configure **compression** (compression acts on batches)

➕ saves space on disk

📏 but compress on the producer ⇒ `compression.type = producer`

➕ saves bandwidth!

➕ saves CPU on brokers (~= 0-copy)

Producers: misc

🧐 Record Timestamp (configurable at topic level)

- 💡 by default: CreateTime (== producer time)
- 🧐 can be set to LogAppendTime (== broker time)

💡 Use **record headers**!!!

- OpenTracing
- Metadata (like PII, etc.)
- Lineage

Which format should I use?


It's on YOU!

Format = YOUR choice

📏 What about the rest of your ecosystem? (Protobuf ⇔ Google services?)

📏 Well-supported in your ecosystem?

- ⚠️ Dynamic languages + Avro?
- ⚠️ Maturity of the library?

📏 JSON Schema can be considered the “default goto”

- ⚠️ But isn’t compressed

💡 Avro & Protobuf => generate code from schemas

Avro-specific

💡 Specific records, almost always

- ⚠️ Consider generic records as a smell (but...)

😒 Sure... Sure... Really useful...

What matters is the **Schema**!

🙏 Use & Manage Schemas 🥺

e.g. by using the Schema Registry

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
BACKWARD	• Delete fields • Add optional fields	Last version	Consumers
BACKWARD_TRANSITIVE	• Delete fields • Add optional fields	All previous versions	Consumers
FORWARD	• Add fields • Delete optional fields	Last version	Producers
FORWARD_TRANSITIVE	• Add fields • Delete optional fields	All previous versions	Producers
FULL	• Add optional fields • Delete optional fields	Last version	Any order
FULL_TRANSITIVE	• Add optional fields • Delete optional fields	All previous versions	Any order
NONE	• All changes are accepted	Compatibility checking disabled	Depends

Compatibility modes:

- 💡 f**OR**ward ⇒ upgrade p**RO**ducer first
- 💡 ba**C**kward ⇒ upgrade **C**onsumers first

🚩 **Avoid** the non transitive ones (esp. **If big retention period**)

- 💡 As conservative as possible
- 💡 Default setting at SR level, be drastic:
  - set to **FULL_TRANSITIVE**
  - then override per topic if really needed

🤨 Why is Backward the default?

Important

The Confluent Schema Registry default compatibility type is `BACKWARD`, not `BACKWARD_TRANSITIVE`.

The main reason that `BACKWARD` compatibility mode is the default, and preferred for Kafka, is so that you can rewind consumers to the beginning of the topic. With `FORWARD` compatibility mode, you aren't guaranteed the ability to read old messages.

Also `FORWARD` compatibility mode is harder to work with. In a sense, you need to anticipate all future changes. For example, in `FORWARD` compatibility mode with Protobuf, you cannot add new message types to a schema.

Can I test it?

📏 Unit Tests: think Property-Based Testing

`deserializeWithV2(serializeWithV1(record))` ⇐ checks ba**C**kwards compatibility

`deserializeWithV1(serializeWithV2(record))` ⇐ checks f**OR**ward compatibility

Inverse property: `deserialize(serialize(record)) == record`

💡 Use: io.confluent.kafka.schemaregistry.CompatibilityChecker (schema-registry lib)

```
78    // visible for testing ←
79    public List<String> isCompatible(
80        ParsedSchema newSchema, List<? extends ParsedSchema> previousSchemas
81    ) {
82      List<? extends ParsedSchema> previousSchemasCopy = new ArrayList<>(previousSchemas);
83      // Validator checks in list order, but checks should occur in reverse chronological order
84      Collections.reverse(previousSchemasCopy);
85      return validator.validate(newSchema, previousSchemasCopy);
86    }
```

Integ. Testing / Schema Promotion

📏 ⇒ leverage the Schema Registry REST API, or maven plugin

😏 Speaking of testing...

Testing

🚩 **Avoid** integ. testing only

🚩 **Avoid** using the same topics lacrosse tests

Integ. Testing: `docker forAll(testSuite)` (in CI):

⚠️ topic-naming (UUID) strategy per test

⚠️ **FLUSH** if test involves a real producer!

Simple, unit testing

💡 `MockProducer` / `MockConsumer`
(+ whatever your framework provides)

💡 **`TopologyTestDriver`** for streams
⇐ really awesome

💡 librdkafka <= `rdkafka-mock`

Advanced, Integ. Testing

💡 `EmbeddedKafkaCluster`

💡 TestContainers
kafka, redpanda, or docker-compose!
(if you need the whole stack)

💡 KRaft (KIP-500) is out!! (zookeeper less)

I need to test performances

Let me tell you a story about 

Perf. testing

🚩 **DO NOT** use custom scripts to create load

🚩 **DO NOT** test blindly

🚩 **DO NOT** set random expectations, please

⚠️ Performances in Kafka is a tradeoff between:

- Latency
- Bandwidth

HEAVILY depends on your workload

📏 **MEASURE**, don't guess

Use proper tooling

📏 JMeter, Gatling, Trogdor

Testing a cluster?

- `kafka-producer-perf-test.sh`
- `kafka-consumer-perf-test.sh`

💡 TestContainers

- kafka, redpanda, or docker-compose! (if you need the whole stack)
- 💡 KRaft (KIP-500) is out!! (zookeeper less)

💡 **MIRROR** your traffic as much as you can

- 💡 eg: MirrorMaker 2
- 📏 Same:
  - Traffic **S**hape
  - **R**epartition (`topic*partitions`)
  - **M**essage format

Chaos Testing

- **AVOID** the rabbit hole
- **DO NOT** set unrealistic expectations

- Trogdor for Kafka workloads with faults
- Testcontainers + Toxiproxy
- Also: Docker + Pumba

Keep it Simple

We must not lose any message

We must recover in time

Check exception handling!!

ex1: check the consumer lag shape ⇒ does it match the outage?

ex2: just look for unexpected behaviour:

10s broker downtime leads to 99% CPU usage on consumers for 10min

Any architectural advice?

Kafka Connect

Remember it's a library!

- 💡 Drives development
  - 🤔 *What do you do in case of failure?*
  - 🤔 *Where do you restart from (source)*
  - 🤔 *How do you know if the record is written? (sink) When do you ACK?*
  - 📏 Start asking yourself those questions?
    - 💡 Using Connect as a lib?

- **What does it bring?**
  - ➕ Deployment model (worker / tasks)
  - ➕ Metrics & all
  - ➕ Benefit from an existing infra? (connect instance running already?)
  - ➕ SerDe agnostic!!!
  - ➕ SMTs!!

- **Examples?**
  - Polling from an external service (HTTP, example: OpenSky API)
  - Protocol-Bridging (websockets to Kafka, signalR to Kafka, ...)

Questions?

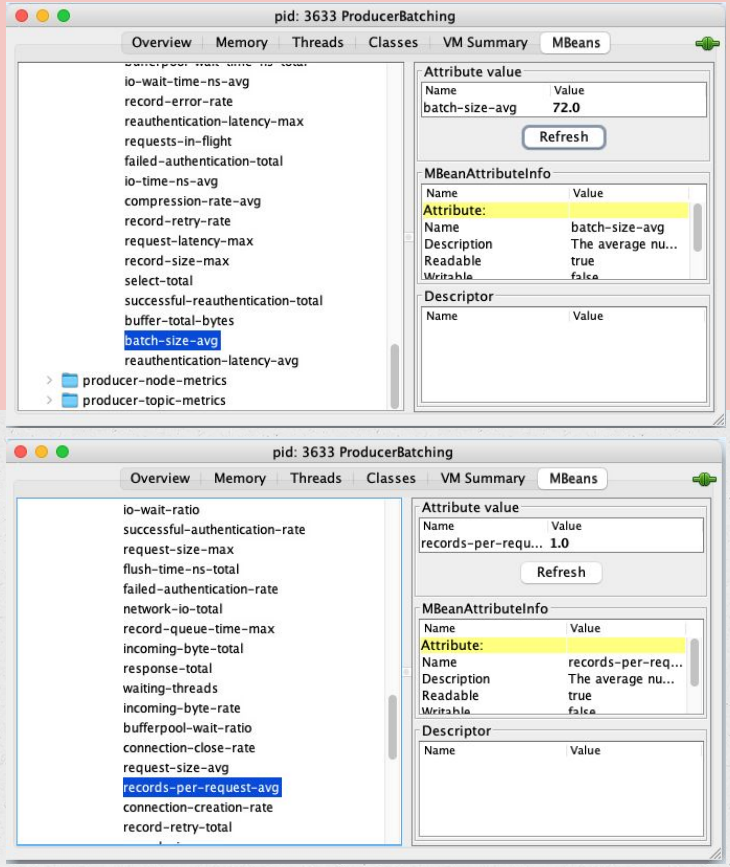
In the director's cut: (90+ min)

- 💡 **linger.ms** & batching
- 💡 A fun (🙄) story about **performance testing** and 🐠
- 💡 Ramblings re. **Idempotency** and Exactly Once Semantics
- 💡 Dealing w/ configuration

Arnaud Esteve
arnaud.esteve@gmail.com

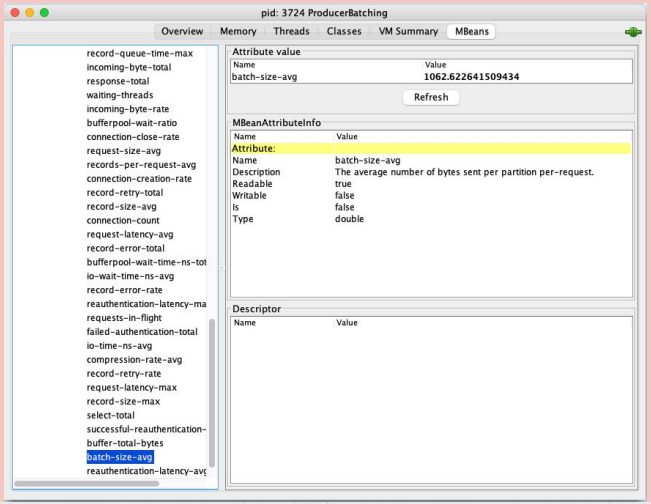
I’m just trying to batch…

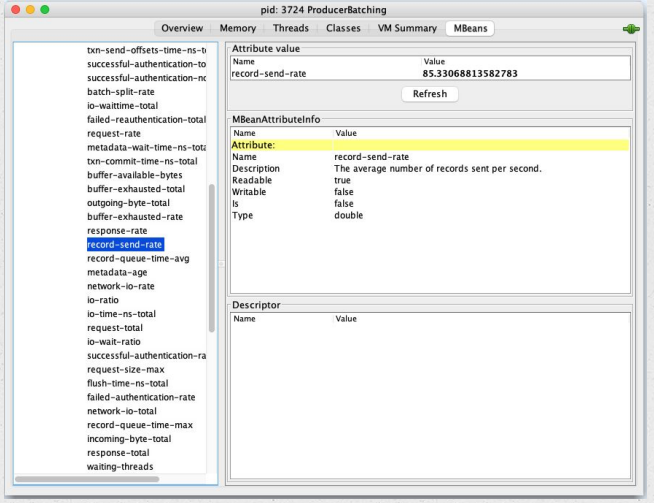
```
var props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.BATCH_SIZE_CONFIG, "10000");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
var topic = "some-topic";
try (var producer = new KafkaProducer<String, String>(props)){
   var i = 0;
   while (true) {
       producer.send(new ProducerRecord<>(topic, Integer.toString(++i)));
       Thread.sleep(10);
   }
} catch (InterruptedException e) {
   throw new RuntimeException(e);
}
```



I’m just trying to batch…

```
var props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.BATCH_SIZE_CONFIG, "10000");
props.put(ProducerConfig.LINGER_MS_CONFIG, "1000");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
var topic = "some-topic";
try (var producer = new KafkaProducer<String, String>(props)){
   var i = 0;
   while (true) {
       producer.send(new ProducerRecord<>(topic, Integer.toString(++i)));
       Thread.sleep(10);
   }
} catch (InterruptedException e) {
   throw new RuntimeException(e);
}
```





Producer batching + compression

💡 Use JMX Metrics to understand what's going on

💡 Increase your chances of **batching**!!!

`linger.ms` > 0

💡 THEN, configure **compression** (compression acts on batches)

➕ saves space on disk

📏 but compress on the producer ⇒ `compression.type = producer`

➕ saves bandwidth!

➕ saves CPU on brokers (~= 0-copy)

Questions?

In the director's cut

- 💡 **linger.ms** & batching
- 💡 Storing configuration
- 💡 A story about **performance testing** and 🐠
- 💡 Ramblings re. Idempotency and Exactly Once Semantics

Arnaud Esteve
arnaud.esteve@gmail.com

I need to test performances

Let me tell you a story about 

Perf. testing

🚩 **DO NOT** use custom scripts to create load

🚩 **DO NOT** test blindly

🚩 **DO NOT** set random expectations, please

⚠️ Performances in Kafka is a tradeoff between:

- Latency
- Bandwidth

HEAVILY depends on your workload

📏 **MEASURE**, don't guess

Use proper tooling

📏 JMeter, Gatling, Trogdor

Testing a cluster?

- `kafka-producer-perf-test.sh`
- `kafka-consumer-perf-test.sh`

💡 TestContainers

- kafka, redpanda, or docker-compose! (if you need the whole stack)
- 💡 KRaft (KIP-500) is out!! (zookeeper less)

💡 **MIRROR** your traffic as much as you can

- 💡 eg: MirrorMaker 2
- 📏 Same:
  - Traffic **S**hape
  - **R**epartition (`topic*partitions`)
  - **M**essage format

Chaos Testing

- **AVOID** the rabbit hole
- **DO NOT** set unrealistic expectations

- Trogdor for Kafka workloads with faults
- Testcontainers + Toxiproxy
- Also: Docker + Pumba

Keep it Simple

We must not lose any message

We must recover in time

Check exception handling!!

ex1: check the consumer lag shape ⇒ does it match the outage?

ex2: just look for unexpected behaviour:

10s broker downtime leads to 99% CPU usage on consumers for 10min

Questions?

In the director's cut

- 💡 **linger.ms** & batching
- 💡 Storing configuration
- 💡 A story about **performance testing** and 🐠
- 💡 Ramblings re. Idempotency and Exactly Once Semantics

Arnaud Esteve
arnaud.esteve@gmail.com

Wtf why are these records duplicated?

🚩 **Avoid**

Assuming everything is EO

Setting EOS by default

🧐 Understand delivery semantics

"Know what you're doing"

⚠️ Tradeoff durability / perf.

💡 Set **idempotency** on producers

💡 Connectors => check documentation

💡 Kafka Streams, set:

`StreamsConfig.PROCESSING_GUARANTEE_CONFIG`

📏 EOS is no magic, only for Kafka-to-Kafka

⇒ design around **At Least Once** where possible

👉 https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-processing-guarantees

Questions?

In the director's cut

- 💡 **linger.ms** & batching
- 💡 Storing configuration
- 💡 A story about **performance testing** and 🐠
- 💡 Ramblings re. Idempotency and Exactly Once Semantics

Arnaud Esteve
arnaud.esteve@gmail.com