

Willkommen zum Seminar



© 2021 anderScore GmbH

Apache Kafka - Event Streaming mit Java

Jan Lühr (M.Sc. Computer Science)

- Senior Software Engineer
- Schwerpunkte
 - Pragmatic Architect
 - Integration und Migration
 - Web / Mobile Engineering
 - Clean Code
 - Trainings, Artikel, Vorträge
 - Network- and Security-Techniques
 - IT-Trainer
- Java, Spring, JEE, Kafka, Android, Microservices,...



- Standort: Köln (mit Rheinblick...)
- Individuelle Softwareentwicklung
- Confluent Partner
- Consulting und Festpreis
- Gesamter Application Life Cycle
- Konferenzen und Artikel
- Öffentliche Trainings



- Technologien
 - JEE, Spring
 - Vaadin, Wicket, Angular, React, Vue
 - Docker, Kubernetes, Apache Kafka
 - ...
- Goldschmiede@anderScore



Apache Kafka – Event Streaming mit Java

02.12. – 03.12.21

Jan Lühr



Begrüßung

1. Einführung
2. Paradigmen und Funktionsweise
3. Komponenten
4. Client-Implementierung
5. Analyse und Transformation
6. Broker Operations
7. Best Practices
8. Ausblick

Lektion 1 - Einführung

Workshop

- Einführung in grundlegende Kafka-Konzepte
- Überwindung von Einstiegshürden
- Funktionen und Features mit Aufgaben erarbeiten

Zielgruppe

- Softwareentwickler, Architekten
und DevOps (m/w/x) mit guten Java-Kenntnissen

Voraussetzungen

- Gute Java Kenntnisse
- Sicherheit im Umgang mit einer IDE (hier: IntelliJ Community Edition)

Beginn	09:00 Uhr
Kaffeepause	ca. 10:30 Uhr
Mittagspause	12:00 bis 13:00 Uhr
Ende	16:00 Uhr

- Video-Konferenz über Zoom
 - Bildschirmfreigabe für Folien
 - Breakout Rooms für Übungen
 - Lautsprecher + Mikrofon benötigt, Kamera empfehlenswert
- Entwicklung: Remote Desktop Protocol (RDP) zur GFU
 - Praktische Übungen
 - Ubuntu VM
 - Aufschaltung über Zoom möglich
- Material auf GitHub

<https://github.com/anderscore-gmbh/kafka-21.12>

Vereinbarungen

- Pausen
 - Gemeinsam zu vorgegebenen Zeiten
 - Individuell während der Übungen
- Erreichbarkeit Dozent
 - Zoom (Chat, Mikrophon)
 - E-Mail
 - Kamera aus: gerade nicht anwesend bzw. ansprechbar
- Regeln
 - Mikrophon möglichst aus (Hintergrundgeräusche)
 - Bei Fragen: "Hand heben" oder Chat
 - Wenn Übung fertig, selbst in Hauptsession zurückkehren

- Kafka Quickstart
 - <https://kafka.apache.org/quickstart>
- Kafka Cheat Sheet
 - <https://github.com/lensesio/kafka-cheat-sheet>
- Kafka E-Book
 - <https://www.confluent.io/resources/kafka-the-definitive-guide>

Jetzt sind Sie dran!

- Name
- Vorwissen
- Erwartungen
- Themenwünsche



- OpenJDK 11
- IntelliJ Community Edition
- Docker
- Docker Compose
- Maven
- Spring
- Kafkacat

Aufgabenstellung:

- Installieren Sie `kafkacat` auf Ihrem System
- Senden Sie eine Nachricht an `broker-1.k.anderscore.com:9092` – Topic: `HelloWorld`
- Konsumieren Sie alle Nachrichten des Topics und geben Sie diese aus

Hinweise:

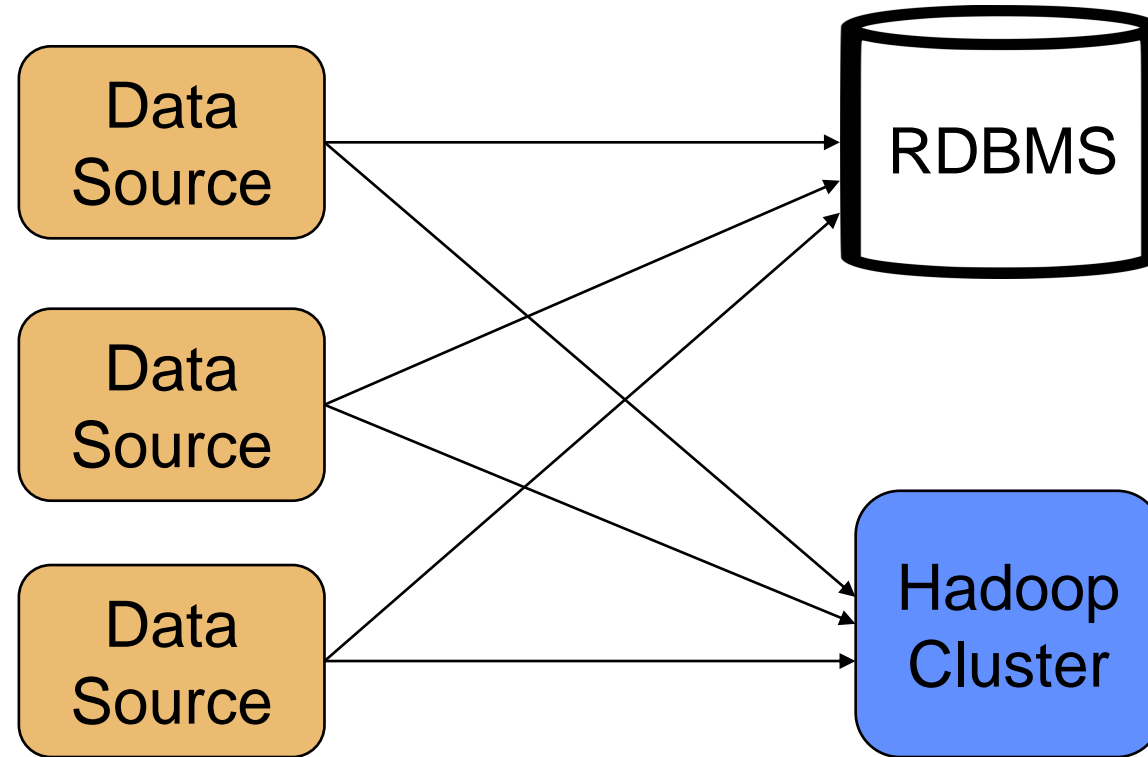
- Ubuntu / Debian (vorinstalliert): `apt-get install kafkacat`
- Nachricht senden:

```
echo "Hallo Welt" | kafkacat -b broker-1.k.anderscore.com -t HelloWorld
```
- Nachricht konsumieren:

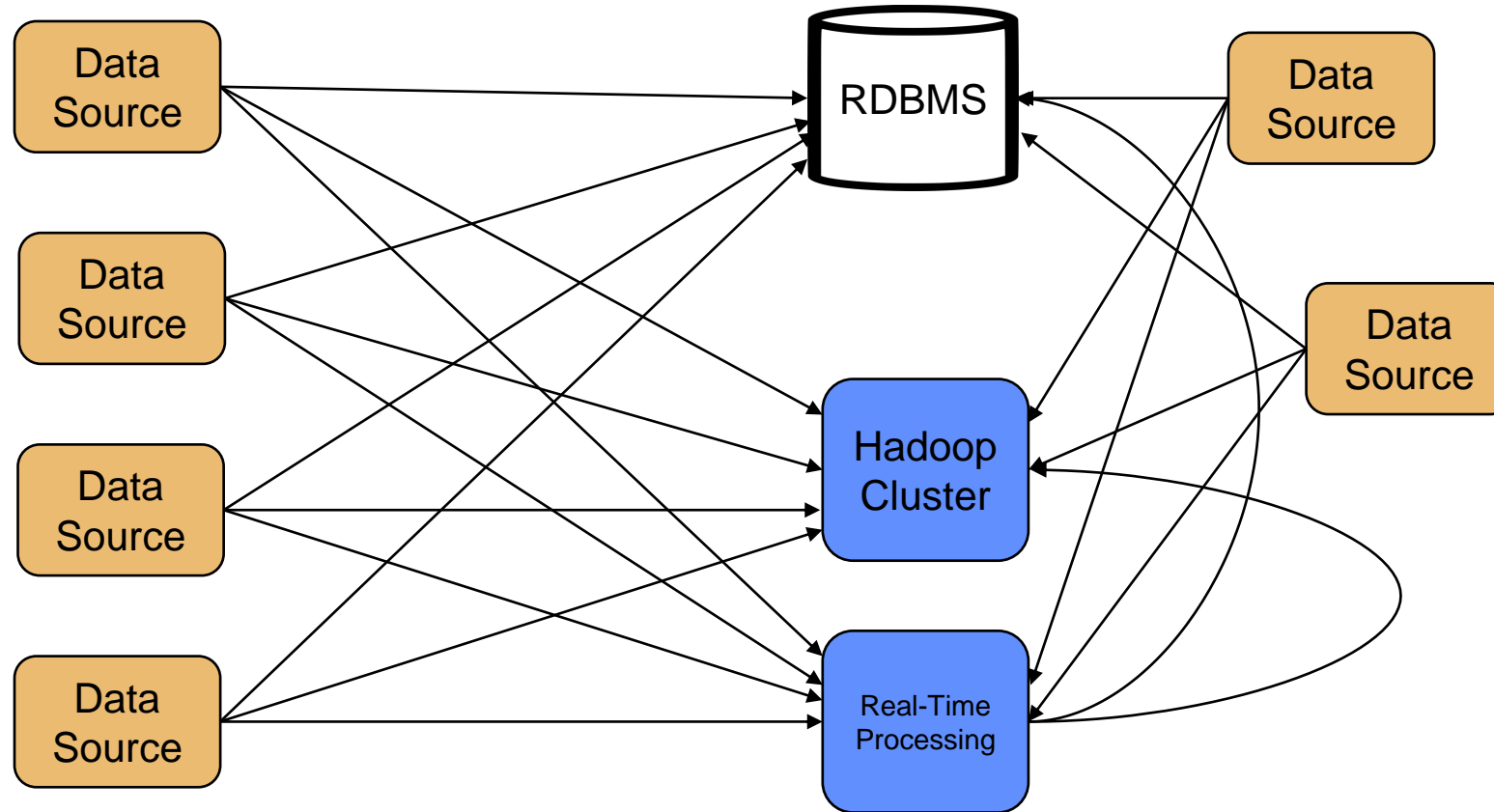
```
kafkacat -b broker-1.k.anderscore.com -t HelloWorld
```

Lektion 2 - Paradigmen und Funktionsweise von Kafka

Einfache Datenverteilung:



Komplexe Datenverteilung:



Motivation

- Komplexe Datenverteilung bewältigen
- Batch-Verarbeitungsprozess verbessern
- Zeitnahe Verarbeitung ermöglichen

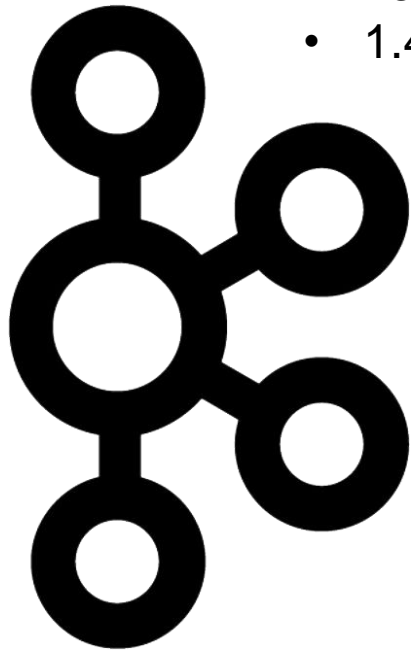
Ansatz

- Middleware für persistente Logs / Streams
- Ähnlich zu MQTT und Message Queueing

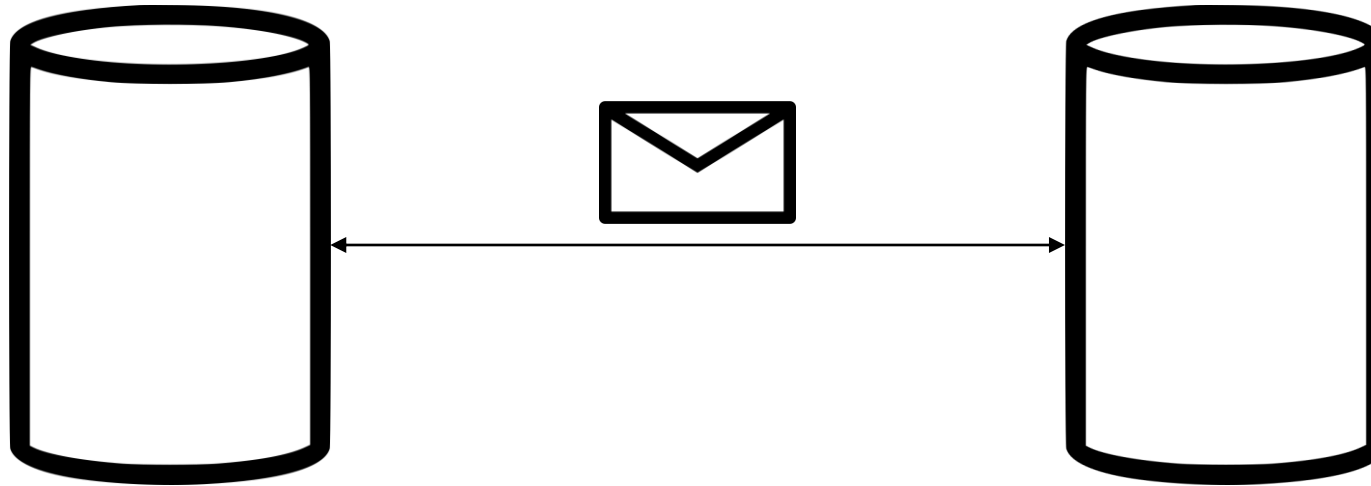
Ziele von Kafka

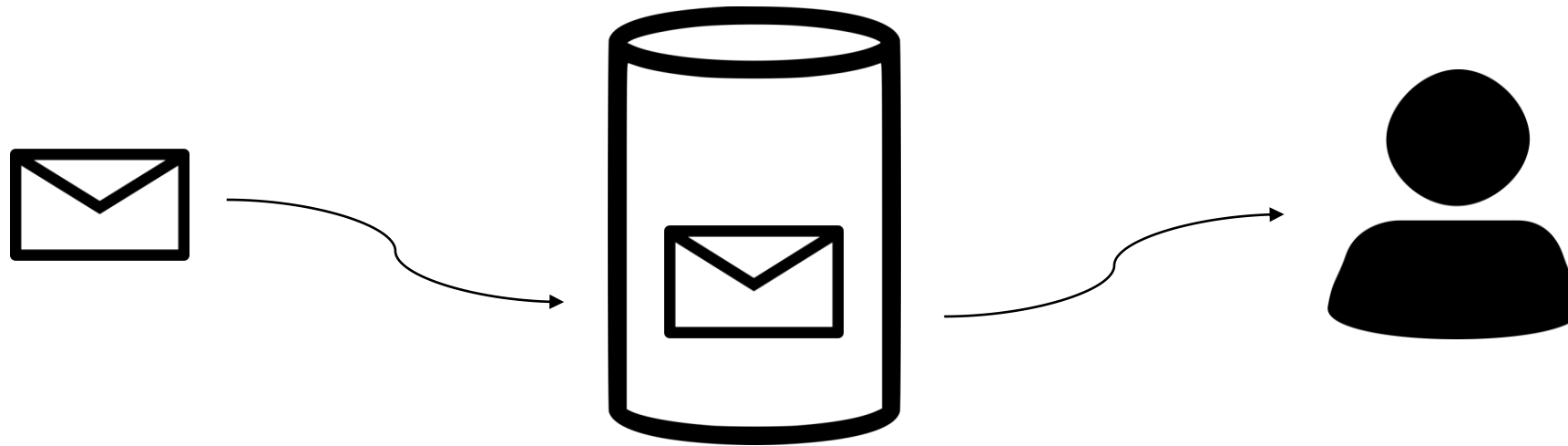
- Pipelines vereinfachen
- Data Stream Handling

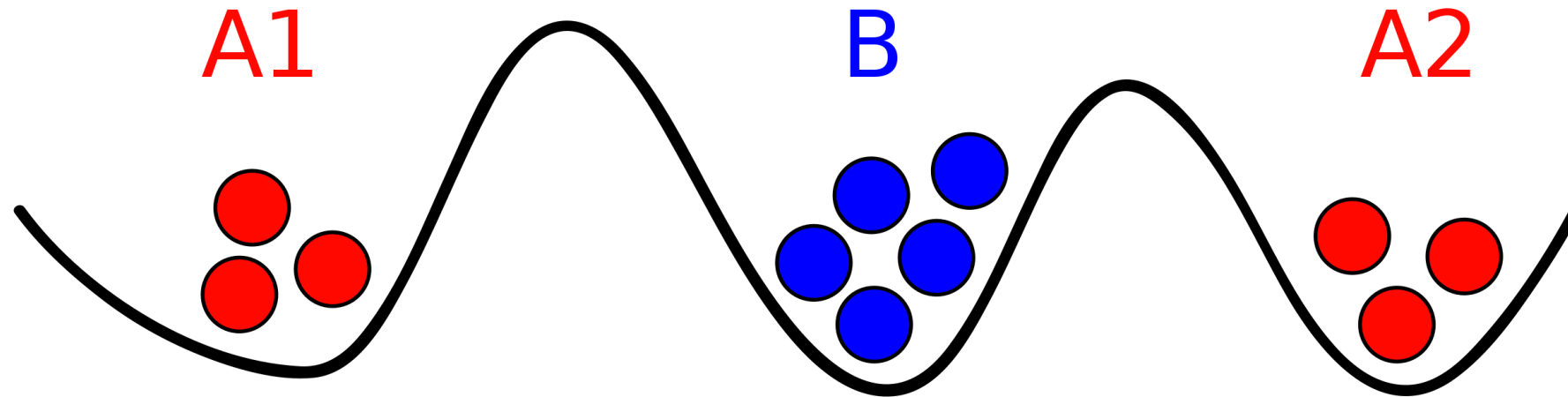
Anstatt Batch Processing -> **Stream Processing**



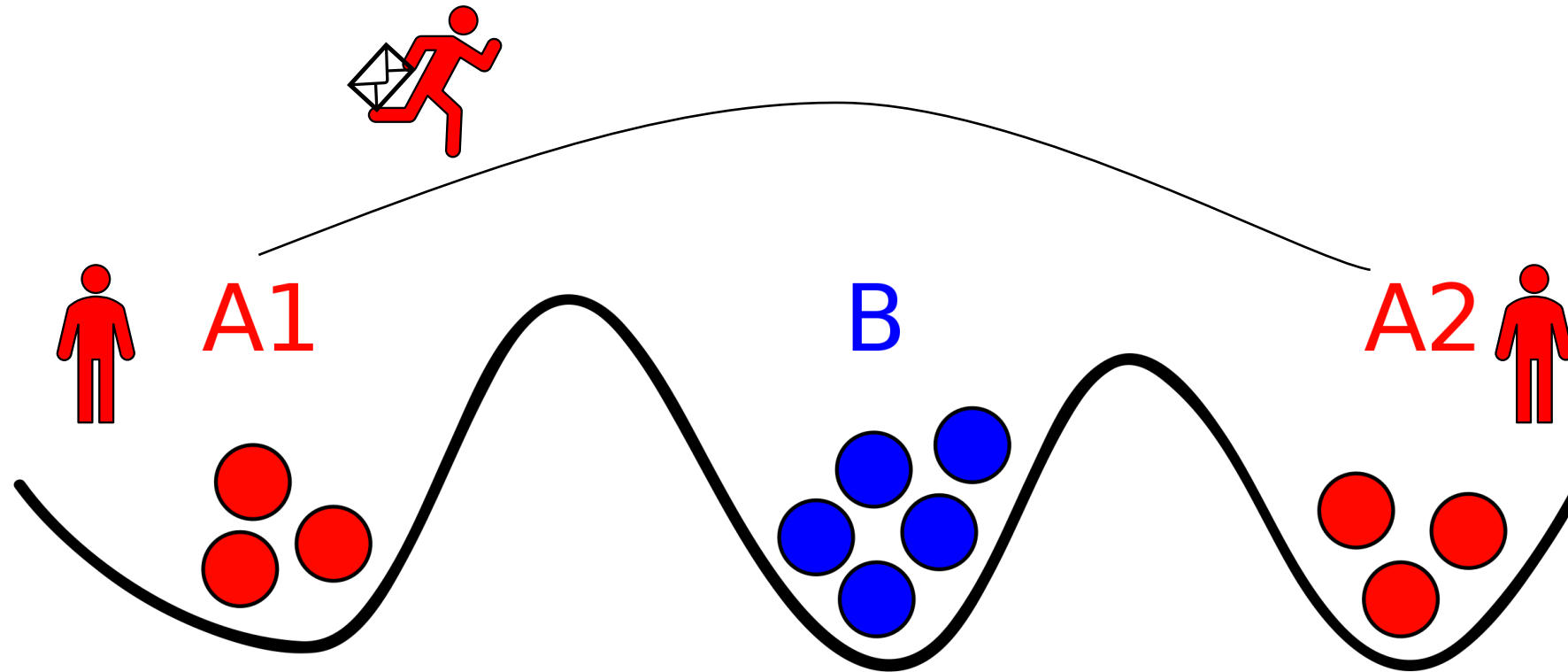
- LinkedIn (2010)
 - Teil der Core-Architektur
 - 1.4 Milliarden Nachrichten pro Tag
- Genutzt von:
 - IBM, Spotify, Uber, Hotel.com, Twitter...
- Use Cases:
 - Event Verarbeitung (*quasi realtime*)
 - Log Aggregation
 - Metriken & Analyse
 - Messaging / μ Service Kommunikation
- Keine Realtime- bzw. Echtzeitverarbeitung
(Werbeversprechen; im Sinne von Reordering Queues)



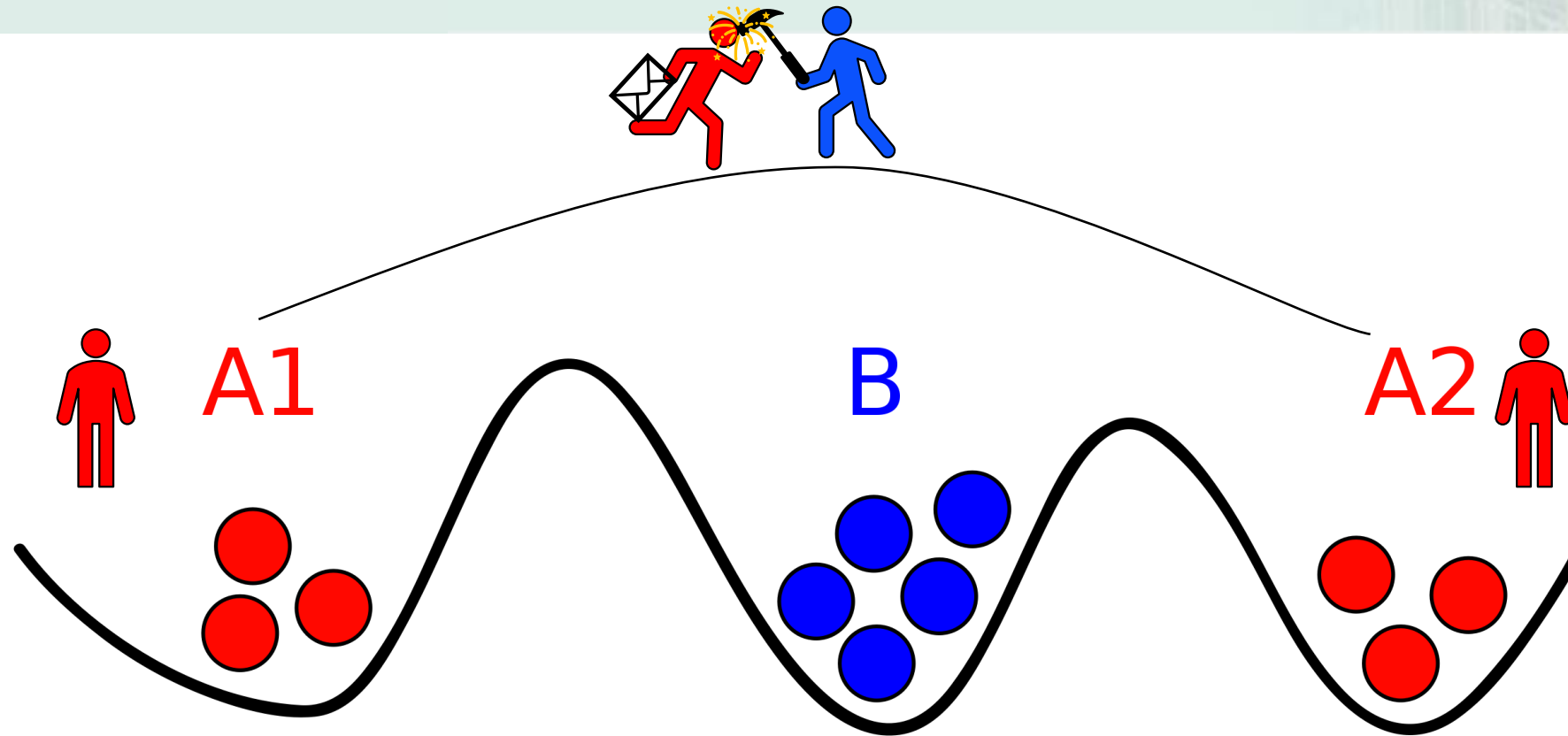


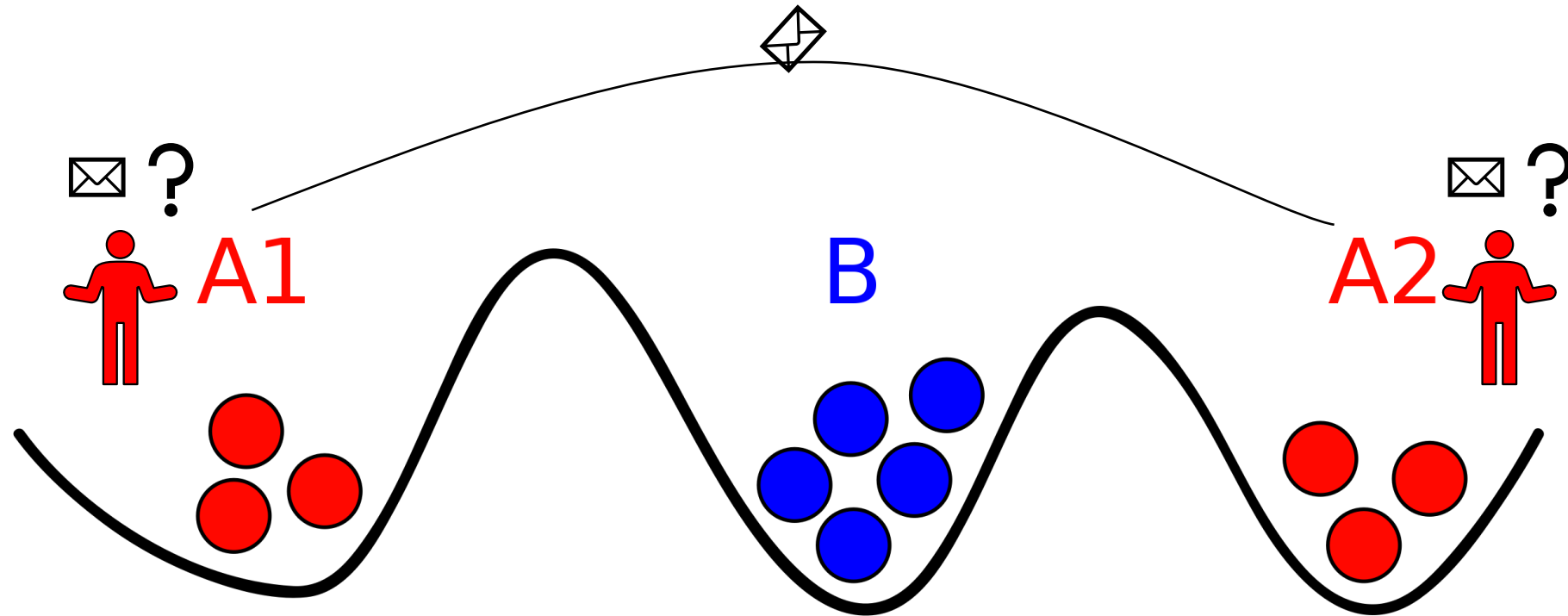


Two Generals Problem



Two Generals Problem





Definition:

Eine Operation, welche mehrfach hintereinander ausgeführt das gleiche Ergebnis wie bei einer einzigen Ausführung liefert.

<https://de.wikipedia.org/wiki/Idempotenz>

Was passiert bei einem Fehler?

- At least once
 - Mindestens ein Mal: Risiko von Duplikaten
- At most once
 - Maximal ein Mal: kein Neuversuch beim Fehlschlag
- Exactly once
 - Genau ein Mal: in Praxis schwer zu erreichen

At Least Once



Situation: Druckfehler / Toner leer: Ausdruck zu hell!

1. Fehler wird behoben
2. Alle Seiten in der Warteschlange werden gedruckt
3. ... sehr viel Papier

At Most Once



Situation: Druckfehler / Toner leer: Ausdruck zu hell!

1. Fehler wird behoben
2. Keine Seite in der Warteschlange wird gedruckt
3. ... hin und her laufen.

Exactly Once



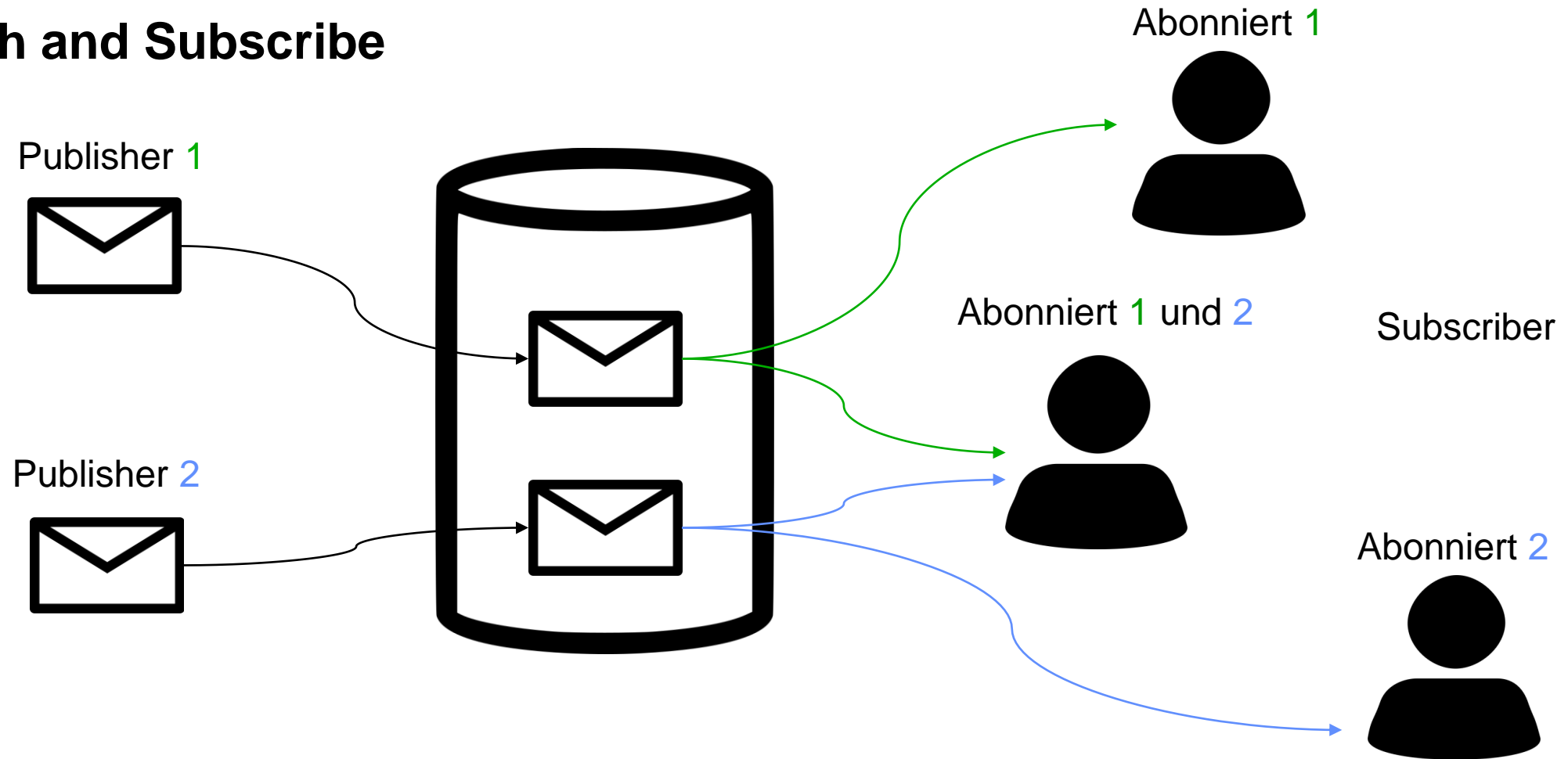
Situation: Druckfehler / Toner leer: Ausdruck zu hell

1. Fehler wird behoben
2. Genau die zu hellen Seiten werden gedruckt
3. ... Perfekt 😊

Woher weiß der Drucker, welche Seiten zu hell sind?

- Idee: Seitennummer am Bedienfeld eingeben
- Im Allgemeinen: schwer umsetzbar
- Idee *auch bei Kafka*: Stand der Verarbeitung im Ergebnis speichern

Publish and Subscribe



Active Polling

Konzept:

- Es wird nicht automatisch an alle Subscriber gesendet (vgl. Observer Pattern)
- Subscriber senden zyklisch Anfragen
- Existiert neuer Inhalt, wird er als Antwort zurückgeliefert

Event Sourcing

Konzept:

- Veränderung eines Zustandes = Event
- Nach Empfang neuer Daten werden alte nicht gelöscht
- Neue Events werden kontinuierlich im Event Store an alte angehängt
- Kafka: Retention Policy

Zweck:

- Kein Informationsverlust
- Analysemuster
- Macht Kafka zu einem Hybrid aus Datenbank und Messaging System

- **Kafka Quickstart** oder
- **Docker Broker**

Aufgabenstellung Quickstart:

- Laden Sie das aktuelle Kafka Release aus:
<http://kafka.apache.org/quickstart>
- Starten Sie das Kafka Environment in der Linux Shell
- Legen Sie einen Topic an
- Schreiben Sie einige Nachrichten in das Topic
- Lesen Sie die Nachrichten aus

Hinweise:

- Download Link Kafka:

https://dlcdn.apache.org/kafka/3.0.0/kafka_2.13-3.0.0.tgz

```
$ tar -xzf kafka_2.13-3.0.0.tgz  
$ cd kafka_2.13-3.0.0
```

- Zookeeper starten:

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

- Kafka Broker starten:

```
$ bin/kafka-server-start.sh config/server.properties
```

Topic erzeugen:

```
$ bin/kafka-topics.sh --create --topic HelloWorld --bootstrap-server localhost:9092
```

Topic prüfen:

```
$ bin/kafka-topics.sh --describe --topic HelloWorld --bootstrap-server localhost:9092  
Topic: HelloWorld PartitionCount:1 ReplicationFactor:1 Configs:  
Topic: HelloWorld Partition: 0 Leader: 0 Replicas: 0 Isr: 0
```

Nachricht in das Topic schreiben:

```
$ bin/kafka-console-producer.sh --topic HelloWorld --bootstrap-server localhost:9092  
This is my first event  
This is my second event
```

Nachricht aus Topic auslesen:

```
$ bin/kafka-console-consumer.sh --topic HelloWorld --from-beginning --bootstrap-server localhost:9092  
This is my first event  
This is my second event
```

Aufgabenstellung Docker:

- Setzen Sie einen eigenen Kafka Broker mit Docker gemäß folgendem Tutorial auf:
 - <https://medium.com/big-data-engineering/hello-kafka-world-the-complete-guide-to-kafka-with-docker-and-python-f788e2588cfc>
 - Starten Sie die Kafka Shell
- Legen Sie einen Topic an
- Initialisieren Sie einen Producer, der eine „Hello World“ Nachricht in den Topic schreibt
- Initialisieren Sie einen Consumer von einem anderen Kafka Terminal, welcher Nachrichten aus dem Topic liest

Hinweise:

- Klonen Sie das *kafka-docker* Projekt und initialisieren Sie die Umgebung mit docker-compose:

<https://github.com/wurstmeister/kafka-docker>

```
> git clone https://github.com/wurstmeister/kafka-docker.git
> cd kafka-docker

# Updaten Sie KAFKA_ADVERTISED_HOST_NAME in 'docker-compose.yml',
# Ändern Sie es zum Beispiel zu: 172.17.0.1
> sudo vim docker-compose.yml
> sudo docker-compose up -d

# Optional: Scalen Sie das Cluster indem Sie mehr Broker hinzufügen (Wird eine zookeeper Instanz starten)
> sudo docker-compose scale kafka=3

# Sie können die laufenden Prozesse mittels folgendem Befehl checken:
> sudo docker-compose ps

# Zerstören sie das Cluster wenn Sie damit fertig sind:
> sudo docker-compose stop
```

Kafka Shell

- Mit folgendem Befehl hochfahren

```
> ./start-kafka-shell.sh <DOCKER_HOST_IP/KAFKA_ADVERTISED_HOST_NAME>
# Wie im Beispiel:
> ./start-kafka-shell.sh 172.17.0.1
```

Einen 'Hello' Topic anlegen

- In der Kafka Shell:

```
> $KAFKA_HOME/bin/kafka-topics.sh --create --topic test \
--partitions 4 --replication-factor 2 \
--bootstrap-server `broker-list.sh`

> $KAFKA_HOME/bin/kafka-topics.sh --describe --topic test \
--bootstrap-server `broker-list.sh`
```


Hello Producer

- Producer initialisieren und eine Nachricht in den Topic schreiben:

```
> $KAFKA_HOME/bin/kafka-console-producer.sh --topic=test \  
--broker-list=`broker-list.sh`  
>> Hello World!  
>> I'm a Producer writing to 'hello-topic'
```

Hello Consumer

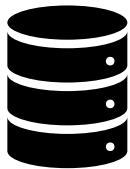
- Consumer von einem anderen Kafka Terminal initialisieren, welcher Nachrichten vom Topic liest:

```
> $KAFKA_HOME/bin/kafka-console-consumer.sh --topic=test \  
--from-beginning --bootstrap-server `broker-list.sh`
```

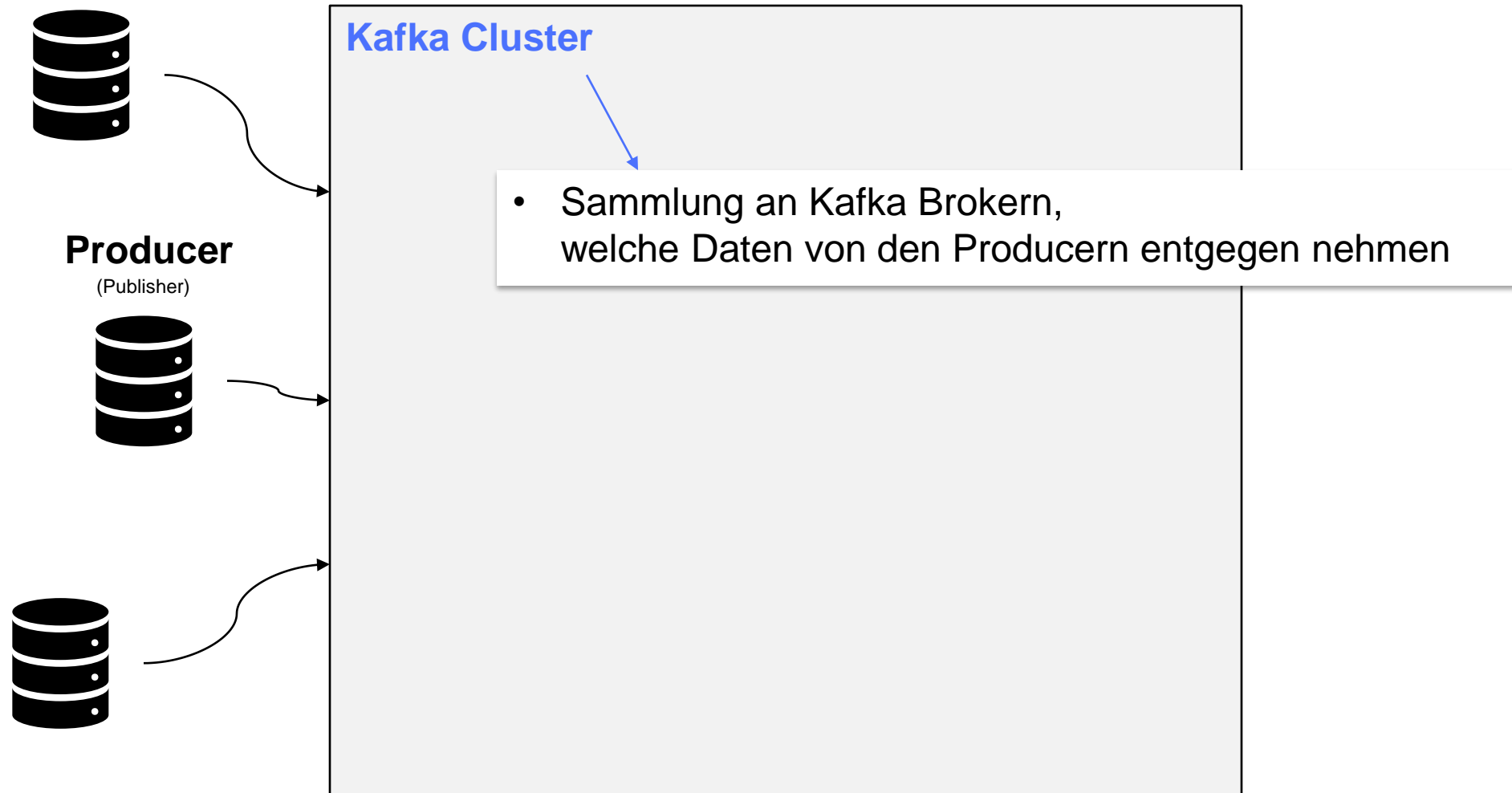
Lektion 3 – Komponenten

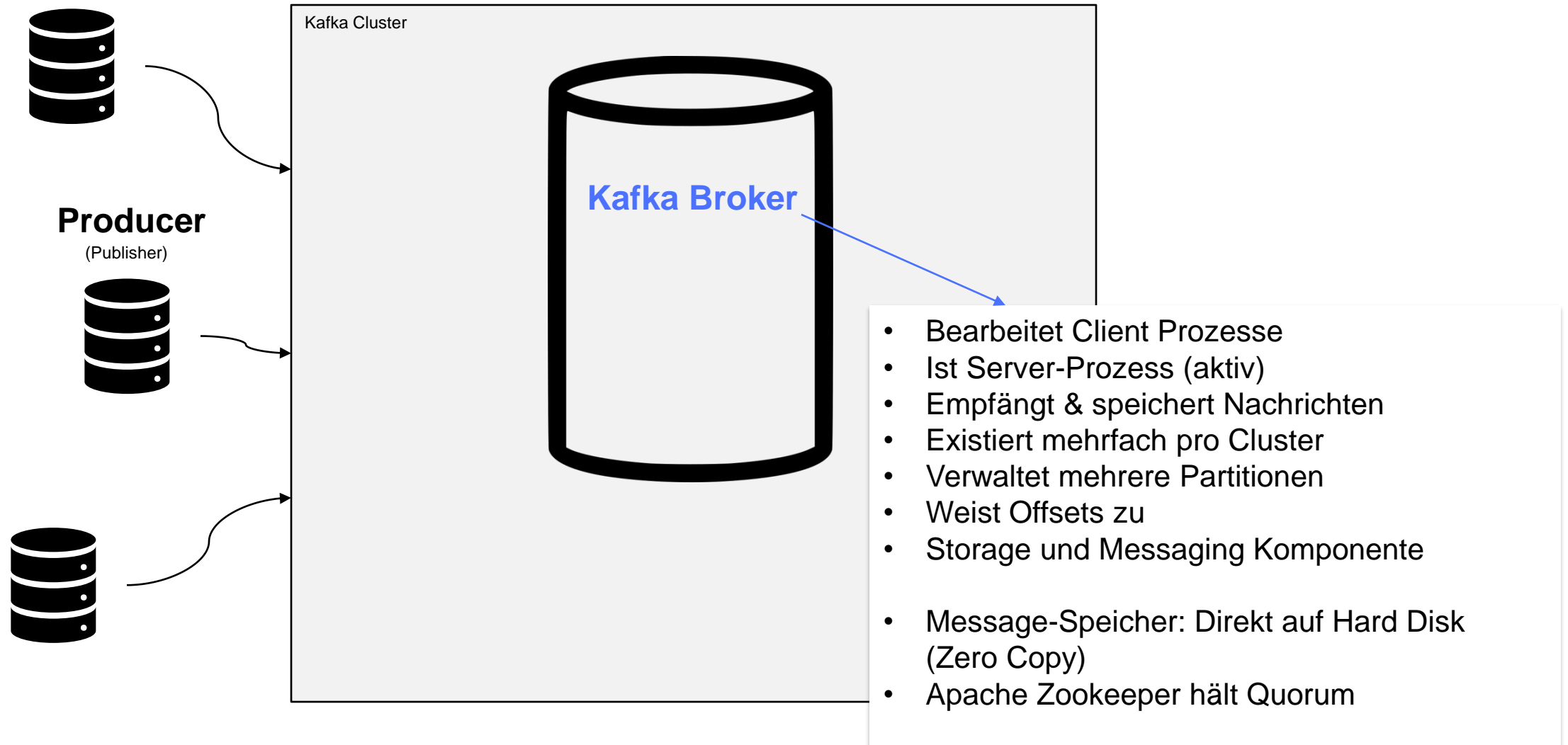
Producer

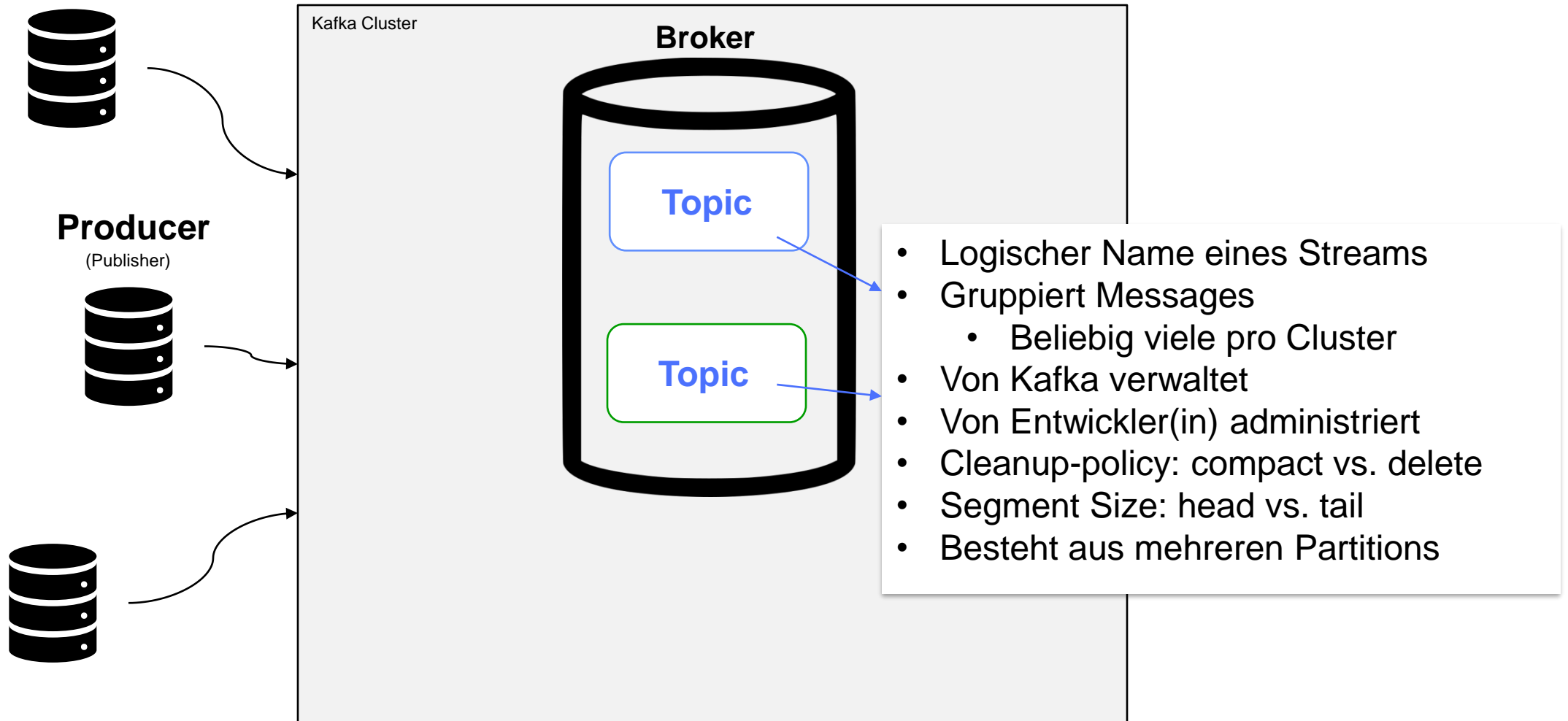
(Publisher)



- Teil einer Anwendung
- Senden Daten an den Kafka Cluster
- Sharding der Messages (Partitionieren)
 - Über Hash Key oder Round Robin
 - Alternativ auch über eigene Strategie
- Für Load Balancing und Semantic Partitioning
- Anbindung über:
 - Nativ: Java, C/C++, Python, Go, .Net, JMS
 - Rest (Confluent)
- Weiterhin existieren Implementierungen für viele andere Sprachen



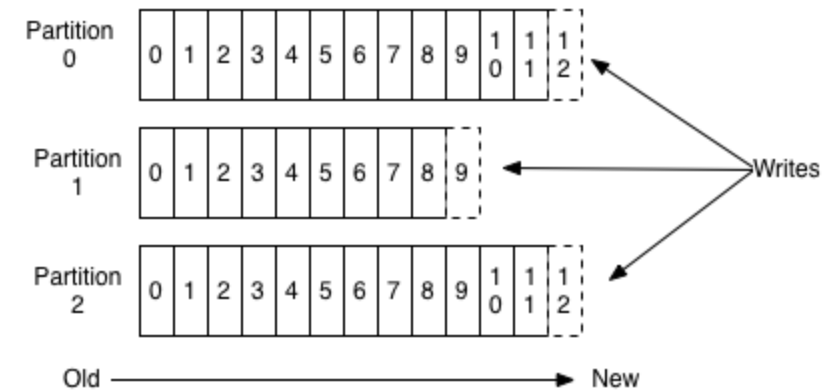


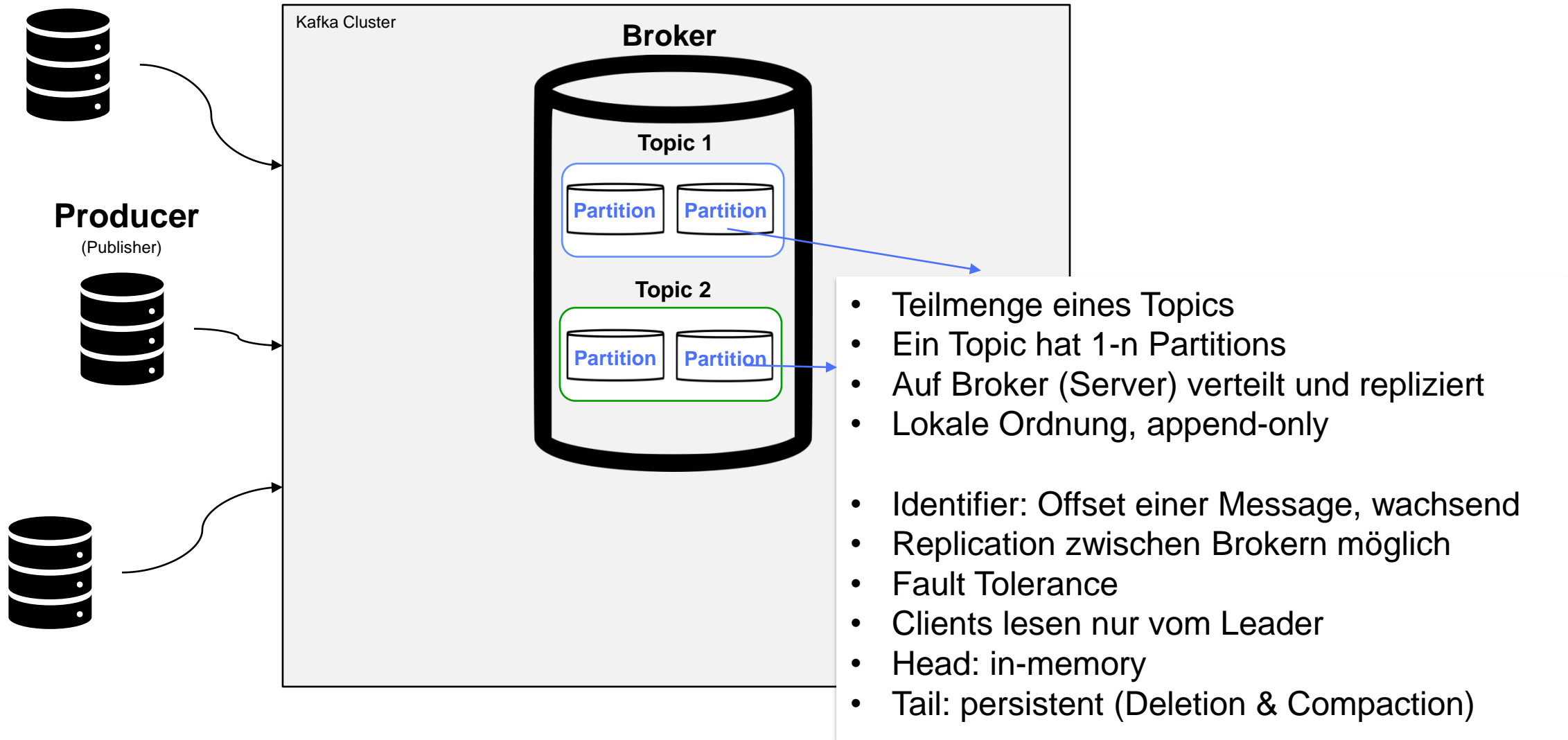


Aufbau:

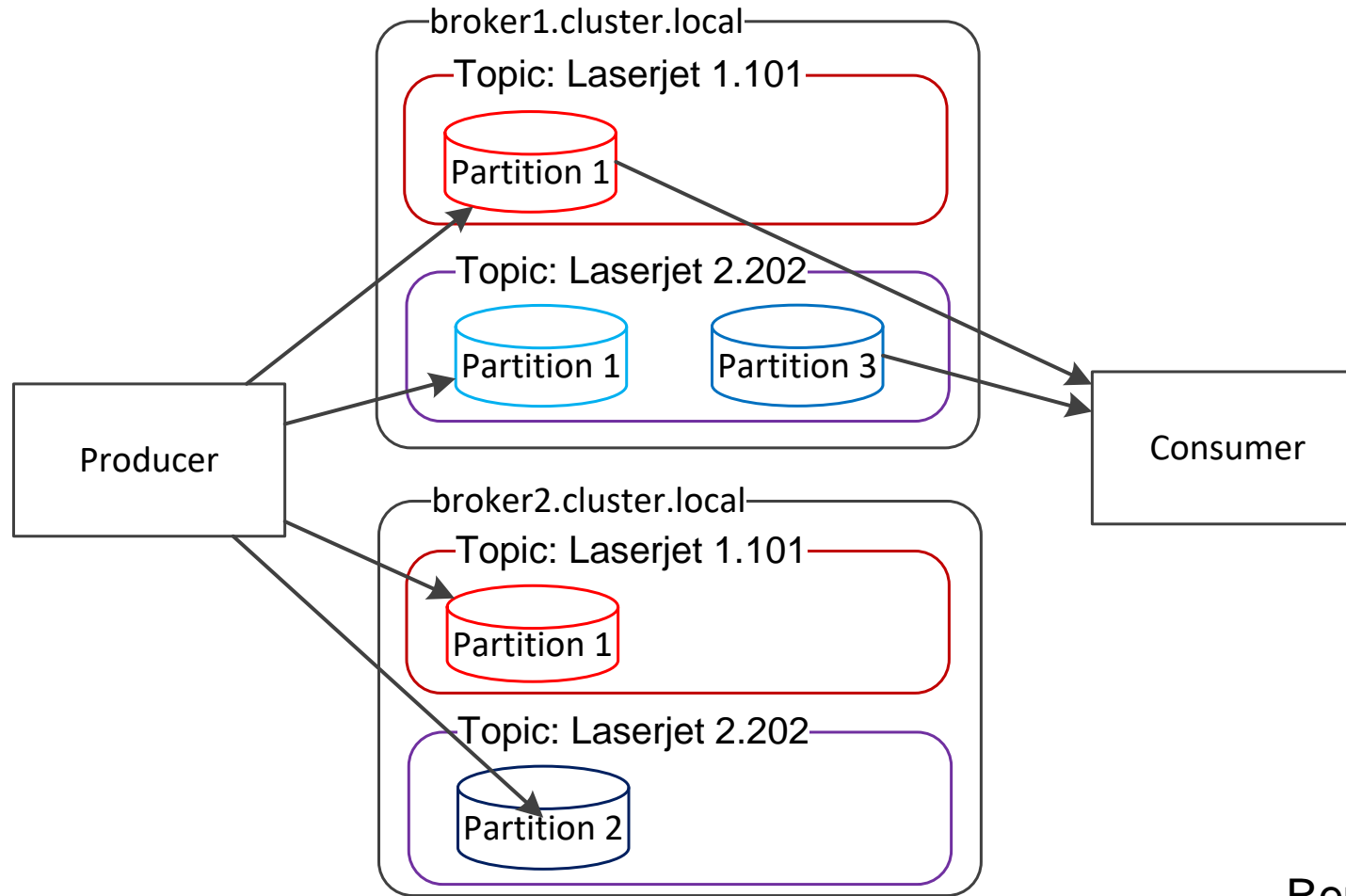
- Cluster besteht aus 1-n Brokern
- Broker haben Topics
- Topics haben 1-m Partitions
 - Clients lesen nur vom Leader
 - Drift konfigurierbar
- Partitions haben eine wachsende Anzahl an Offsets
- Lokale Ordnung innerhalb einer Partition

Anatomy of a Topic

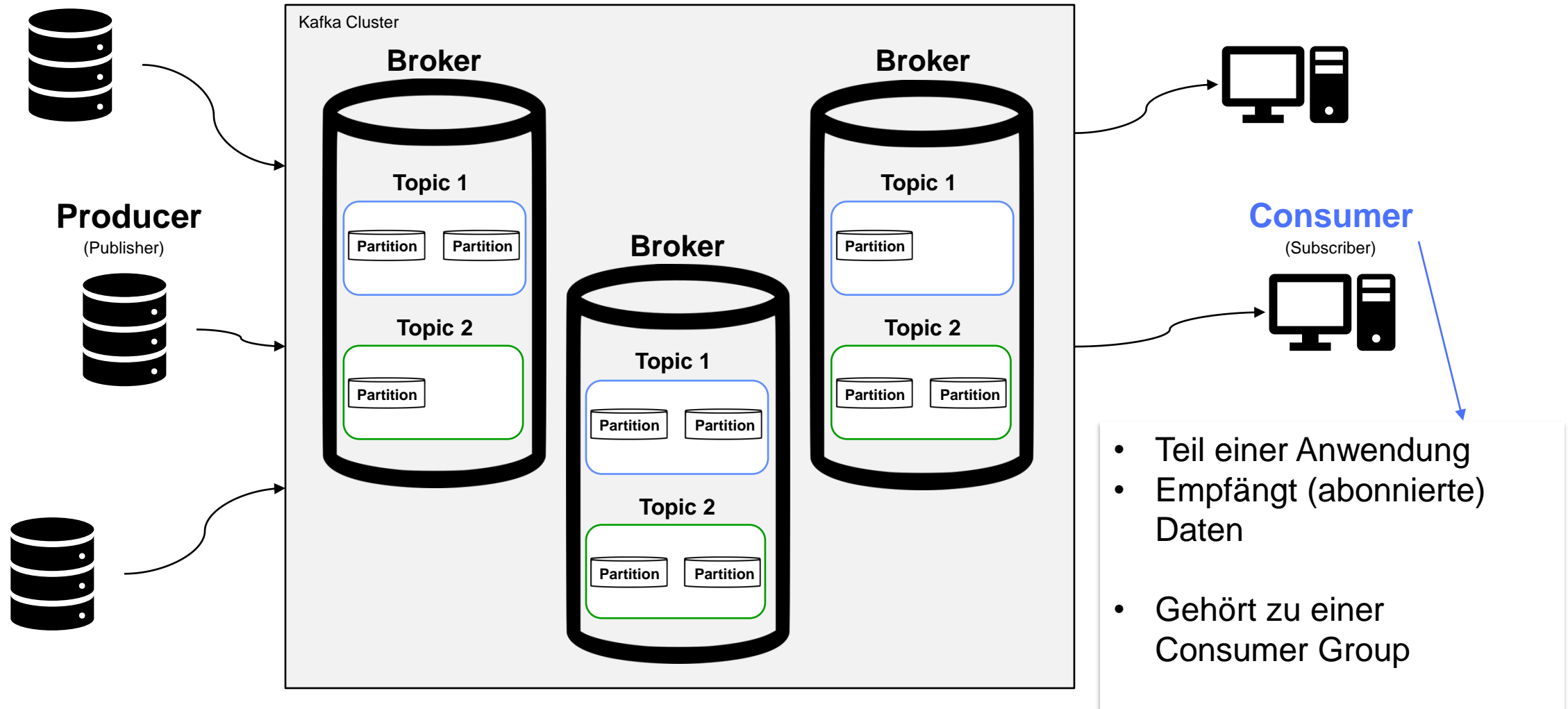




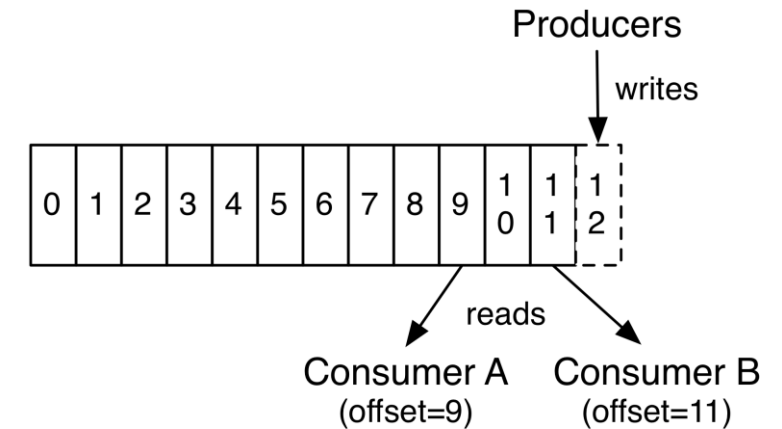
Partitionen: Printer Beispiel



Repliziert: Laserjet 1.101
Sharding: Laserjet 2.202

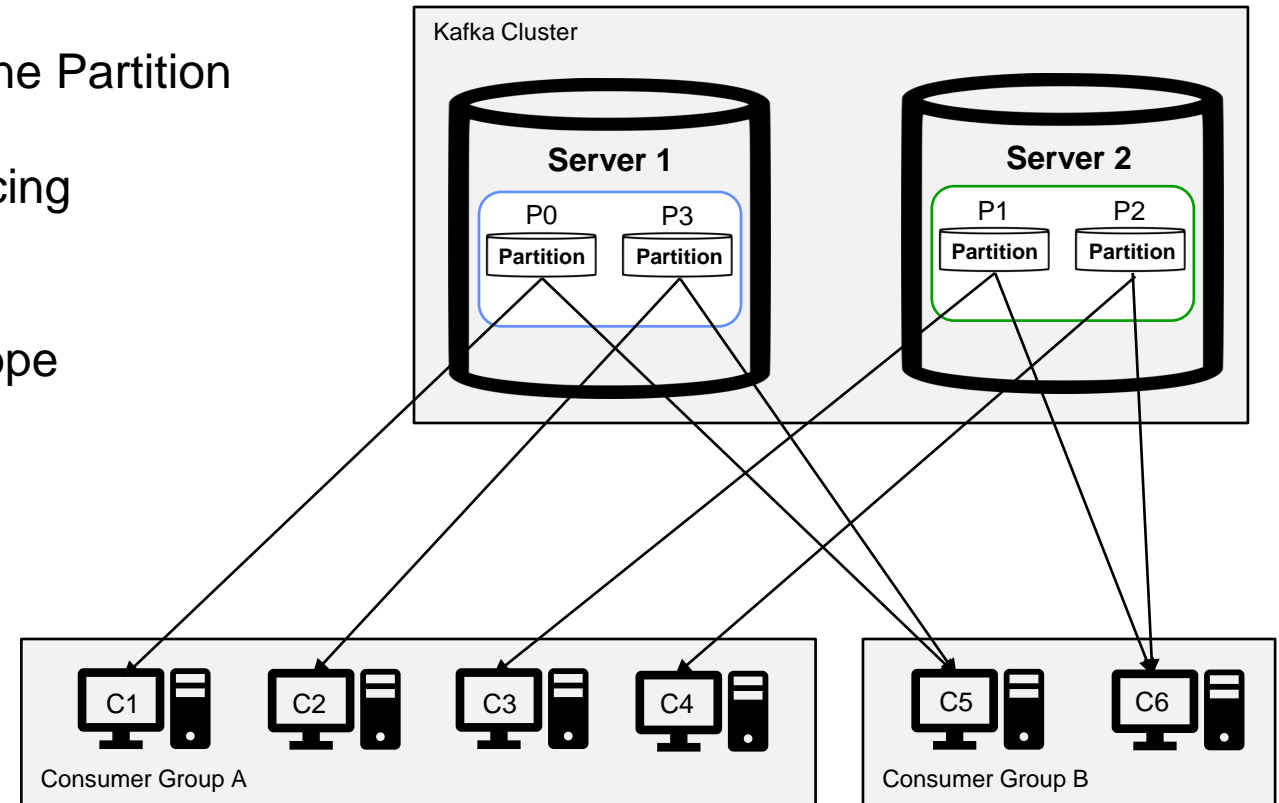


- Abruf von Messages via **pull, single-threaded**
- **Consumer Offset**
 - *Nächste zu lesende* Nachricht
 - Pro Partition
 - Speicher: Spezielles Topic (oder extern)
 - Commit: automatisch (default: 5 sec – Obacht!) oder manuell
- **Semantik:**
 - At least once: Nachricht bearbeiten *danach* Commit
 - At most once: Commit, *danach* Nachricht bearbeiten
 - Exactly once: Offset im Zielsystem speichern
- **Verschiedene Consumer**
 - Gleichzeitiges Lesen möglich
 - Default: Alle Nachrichten im Topic an alle Consumer
 - Spezialfall: *Consumer Group*



<https://docs.confluent.io/platform/current/clients/consumer.html>

- Gruppierung: **Consumer Group**
 - Mehrere Consumer kombinieren -> Consumer Group
 - Jeder Consumer bearbeitet nur **Subset** der Partitionen
 - Eindeutige Group ID
 - Jeder Consumer in einer Group braucht eigene Partition
 - Ein Subset von Partitionen
 - Automatisches Error-Handling & Load-Balancing
- Scaling
 - Maximal ein Consumer pro Partition pro Gruppe
 - **#Consumer ≤ #Partitions**
 - #Partition ändern: Schwer möglich
 - Besser: Neues Topic bei Release-Wechsel (API-Versionierung)



Was ist **Zookeeper**?

- Open Source von Apache
- Ermöglicht verteilte Koordination
- Kümmt sich um Konfigurationsinformationen
- Bietet verteilte Synchronisation

Besteht aus drei oder fünf Servern im Quorum

- Quorum:
 - Eine replizierte Gruppe von Servern in der gleichen Applikation nennt man Quorum
 - Im replicated mode haben alle Server im Quorum eine Kopie der gleichen Config Datei

Kafka Broker nutzen **Zookeeper** für:

- Cluster Management
- Fehlerfindung und Wiederherstellung
- Speicherung von Access Control Lists (ACL)

Aufgabenstellung:

- Erstellen Sie ein Topic mit 2 Partitionen auf dem Broker, welchen Sie in Aufgabe 1 eingerichtet haben
 - Alternativer Broker: `broker-1.k.anderscore.com`
- Welche Funktion und Auswirkung haben die Parameter:
 - a) `segment.ms` **und** `segment.bytes`
 - b) `cleanup.policy = delete` (**oder** `compact`)
 - c) `retention.ms` **oder** `retention.bytes`
 - d) `min.cleanable.dirty.ratio`
- Schreiben Sie die Zahlen 0...42 in das Topic und lesen Sie sie daraus.
Wie sind die Zahlen geordnet?

Hinweise:

- Verwenden Sie die Kafka Command Line Tools um das Topic anzulegen

Topic erzeugen:

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create \  
--topic myTopic --partitions 2 --replication-factor 1
```


Nachricht senden:

```
for zahl in `seq 0 42`; do echo $zahl | bin/kafka-console-producer.sh \
--broker-list localhost:9092 --topic myTopic; done.
```

Hinweis: Schneller geht's auch mit `kafkacat` statt `kafka-console-producer.sh`, da kein JVM-Process pro Iteration erzeugt wird – `kafka-console-producer.sh` bringt jedoch mehr Optionen

Nachricht Lesen:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
--from-beginning --topic myTopic
```

Lektion 4 – Client Implementierung

Java APIs für Kafka

- **Java Producer API**
 - Kafka Client, welcher Datensätze zum Kafka Cluster published
 - Thread Safe
 - Dependency Injection Scope: Singleton
 - Pufferung bei Verbindungsverlust
 - Automatische Retries möglich (Vorsicht: Reordering!)
- **Java Consumer API**
 - Kafka Client, welcher Datensätze aus einem Kafka Cluster konsumiert
 - Transparenz bei Fehlern von Brokern
 - Passt sich an Migrationen von Partitionen im Cluster an
 - Interagiert mit Broker und erlaubt Zugriff auf dessen Consumer Groups

Erstellung eines Producers:

Klasse

```
public class KafkaProducer<K,V>
```

Wichtige Eigenschaften und Senden

```
Properties props = new Properties();  
props.put("bootstrap.servers", "localhost:9092");  
props.put("acks", "all");  
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
  
Producer<String, String> producer = new KafkaProducer<>(props);  
for (int i = 0; i < 100; i++)  
    producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toString(i)));  
producer.close();
```

- **bootstrap.servers**
 - Liste an Broker host/port Paaren für eine initiale Verbindung zum Cluster
- **key.serializer / value.serializer**
 - Klassen zur Serialisierung von Keys bzw. Values
 - Müssen Serializer Interface implementieren
- **acks**
 - Anzahl an Bestätigungen (Acknowledgements), welche der Producer benötigt, bevor der Request fertig ist
 - **acks = 0**: Producer wartet nicht auf Bestätigungen vom Server
 - **acks = 1**: Producer wartet, bis der Datensatz auf den lokalen Log geschrieben wurde
 - **acks = all**: Producer wartet, bis alle in-sync Replikate das Erhalten der Datensätze bestätigt haben

- Die `send()` Methode ist non-blocking
 - Gibt Datensatz in einen Buffer von wartenden Datensätzen über und gibt sofort ein Return zurück
 - Effizienz: Batched einzelne Datensätze zusammen
 - Falls nötig kann mit `.send(record).get()` ein block geforced werden

```
ProducerRecord<String, String> record = new ProducerRecord<String, String>("my-topic", "myKey", "myValue");  
  
Future<RecordMetadata> metadata = producer.send(record);  
producer.close();
```

- **Retries:**
 - Wie oft ein Send bei einem Fehler wiederholt wird
 - Kann zu Änderung in der Reihenfolge der Nachrichten führen!
 - Anzahl der Connections kann angepasst werden

```
retry.backoff.ms=100  
retries=600  
  
# default ist 5  
set max.in.flight.requests.per.connection=1
```

Default Linger:

- Buffer sendet sofort, auch bei ungenutztem Space
- Um die Anzahl an Requests zu verringern, kann Wartezeit konfiguriert werden
- Erhöht Effizienz bei minimaler Latenz

```
# größer als 0  
linger.ms = 1
```

Buffer Größe:

- Gesamtmenge an Speicher, welcher dem Producer für den Buffer zu Verfügung gestellt wird
- Wenn der Buffer voll ist, werden Send Requests geblockt (TimeoutException)

```
buffer.memory  
max.block.ms
```

Reaktion auf Fehler:

- Fehler: Metadata ist `null`
- Kein Fehler: Exception ist `null`

```
ProducerRecord<byte[],byte[]> record = new ProducerRecord<String,String>("the-topic", key, value);
producer.send(record,
    new Callback() {
        public void onCompletion(RecordMetadata metadata, Exception e) {
            if(e != null) {
                e.printStackTrace();
            } else {
                System.out.println("The offset of the record we just sent is: " + metadata.offset());
            }
        }
    });
```


Erstellung eines Consumers:

Klasse

```
public class KafkaConsumer<K,V>
```

Wichtige Eigenschaften und Polling

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");
props.setProperty("group.id", "test");
props.setProperty("enable.auto.commit", "true");
props.setProperty("auto.commit.interval.ms", "1000");
props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(),
record.value());
}
```

- **bootstrap.servers**
 - Liste aus Broker hosts/ports Paaren für eine initiale Verbindung zum Cluster
- **key.deserializer / value.deserializer**
 - Klassen zur Deserialisierung von Keys bzw. Values
 - Müssen Deserializer Interface implementieren
- **group.id**
 - Zeigt an, zu welcher Consumer Group der Consumer gehört
- **enable.auto.commit**
 - Bei true triggered der Consumer offset commits

```
Properties props = new Properties();  
props.setProperty(setting, value);
```

- Die `poll()` Methode gibt alle verfügbaren Nachrichten zurück
 - Bis zu der maximalen Größe per Partition

```
# default 1048576 bytes  
max.partition.fetch.bytes
```

- Eine zu hohe Anzahl an Partitionen kann extreme Mengen an Daten zurückgeben
 - Gesamtmenge von Datensätzen in einem einzelnen Poll kann reduziert werden (Chunking)

```
max.poll.records
```

Aufgabenstellung:

- Erzeugen Sie ein neues Java Projekt und binden Sie den Kafka Client ein
- Verbinden Sie sich mit Ihrem Kafka Broker
- Implementieren Sie einen Consumer: Lesen Sie alle Nachrichten aus dem Topic „HelloWorld“ aus
- Geben Sie die Nachrichten auf der Konsole aus
- Implementieren Sie einen Producer

- Im einfachsten Fall können Sie Kafka-Events ohne weitere Frameworks konsumieren.
- Hierzu müssen die entsprechenden Bibliotheken in das Projekt angebunden werden.

```
# Maven dependencies
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.0.0</version>
</dependency>
```

- Erstellen Sie ein neues Maven Projekt z.B. mit:

```
mvn archetype:generate -DgroupId=gs -DartifactId=kafka \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DinteractiveMode=false
```

- Fügen Sie dann die Maven Dependency der Datei pom.xml hinzu
- Nutzen Sie die folgenden Klassen:

```
org.apache.kafka.clients.consumer.KafkaConsumer
org.apache.kafka.clients.producer.KafkaProducer
```

Spring for Kafka

- Version 2.7 seit 2016
 - Basiert auf Kernkonzepten von Spring (z.B. DI, Annotationen, Templates)
 - Wird verwendet, um Kafka basierte Messaging Lösungen zu entwickeln
 - Bietet Template für High-Level Abstraktion für das Senden von Messages
 - Bietet Support für Message-driven POJOs
-
- Es kann plain Java zum Versenden und Empfangen von Messages genutzt werden
 - Java mit Konfiguration
 - Oder am simpelsten mit **Spring Boot**

Dependencies

- spring-kafka JAR und alle seine Dependencies
- Am einfachsten mit Maven

```
<dependency>  
  <groupId>org.springframework.kafka</groupId>  
  <artifactId>spring-kafka</artifactId>  
  <version>2.6.5</version>  
</dependency>
```

Spring Boot Beispiel:

- Anwendung sendet 3 Nachrichten an ein Topic
- Nachrichten werden empfangen
- Anwendung wird beendet

Voreinstellungen

- Es wird Group Management benutzt, um Topics und Partitions Consumern zuzuordnen
- Hierfür muss eine Gruppe erstellt werden:

```
spring.kafka.consumer.group-id=foo
```

- Der Container könnte nach dem Verstand der Nachrichten starten
- Es muss ein Offset eingestellt werden, um sicher zu gehen, dass die neue Consumer Group die Nachrichten auch bekommt:

```
spring.kafka.consumer.auto-offset-reset=earliest
```



```
@SpringBootApplication
public class Application implements CommandLineRunner {

    public static Logger logger = LoggerFactory.getLogger(Application.class);

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args).close();
    }

    @Autowired private KafkaTemplate<String, String> template;
    private final CountDownLatch latch = new CountDownLatch(3);

    @Override
    public void run(String... args) throws Exception {
        this.template.send("myTopic", "foo1");
        this.template.send("myTopic", "foo2");
        this.template.send("myTopic", "foo3");
        latch.await(60, TimeUnit.SECONDS);
        logger.info("All received");
    }

    @KafkaListener(topics = "myTopic")
    public void listen(ConsumerRecord<?, ?> cr) throws Exception {
        logger.info(cr.toString());
        latch.countDown();
    }
}
```

Aufgabenstellung:

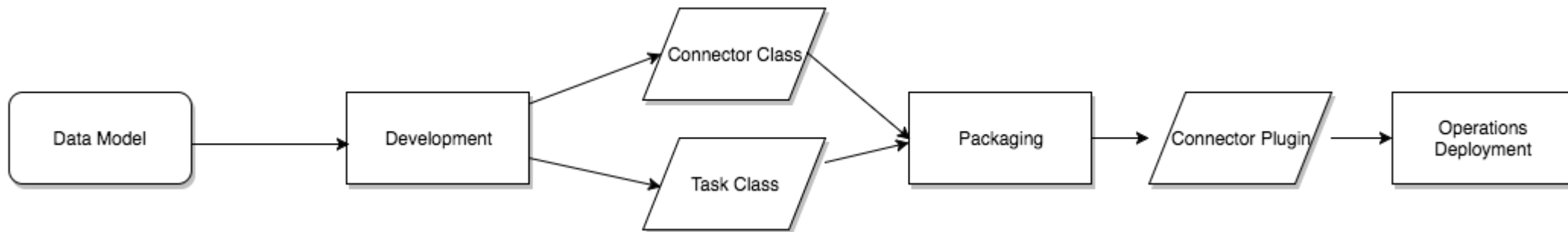
- Erstellen Sie ein Spring Boot Projekt
- Binden Sie die Kafka Bibliotheken ein und konfigurieren Ihren Broker auf Port 9092
- Konsumieren Sie das Topic „Hallo Welt“ und geben Sie es auf der Konsole aus
- Senden Sie eine Nachricht an das Topic

- **Properties:** `group-id` und `auto-offset-reset` anpassen
- Wie ein **neues Spring-Boot-Projekt** anzulegen ist, wird hier beschrieben:
 - <https://spring.io/guides/gs/spring-boot/>
 - <https://start.spring.io>

Lektion 5 – Analyse und Transformation

Anbindung von externen Systemen an Kafka

- Beispiel: Oracle RDB
- Verschiedene Konnektoren in Confluent Platform enthalten
- **Connector:** Job bzw. Plugin für Kafka Cluster
- **Task:** Logik zur Durchführung des Datenaustausches (stateless)
- **Workers:** Laufende Prozesse für Connectors und Tasks (standalone vs. distributed)
- **Converters:** Konverter zwischen Kafka und Datenquelle bzw. Datensenke (z.B. JsonConverter)
- **Transforms:** Anpassung von Nachrichten von oder zu einem Connector (z.B. Filter)
- **Dead Letter Queue:** Behandlung von Connector Fehlern (*auch für Consumer relevant!*)



Quelle: <https://docs.confluent.io/platform/current/connect/concepts.html#:~:text=handles%20connector%20errors-,Connectors,defined%20in%20a%20connector%20plugin.>

Kafka ist eine Streaming Plattform

- Stream = kontinuierliches, ununterbrochenes, updatendes Set
- Streams von Datensätzen publishen und abonnieren
 - ähnelt einer Message Queue
 - oder: einem Enterprise Messaging System
- Speichern von Datensatz-Streams
 - Fehlertolerante Weise
- Verarbeiten von Datensätzen
 - sobald diese auftreten

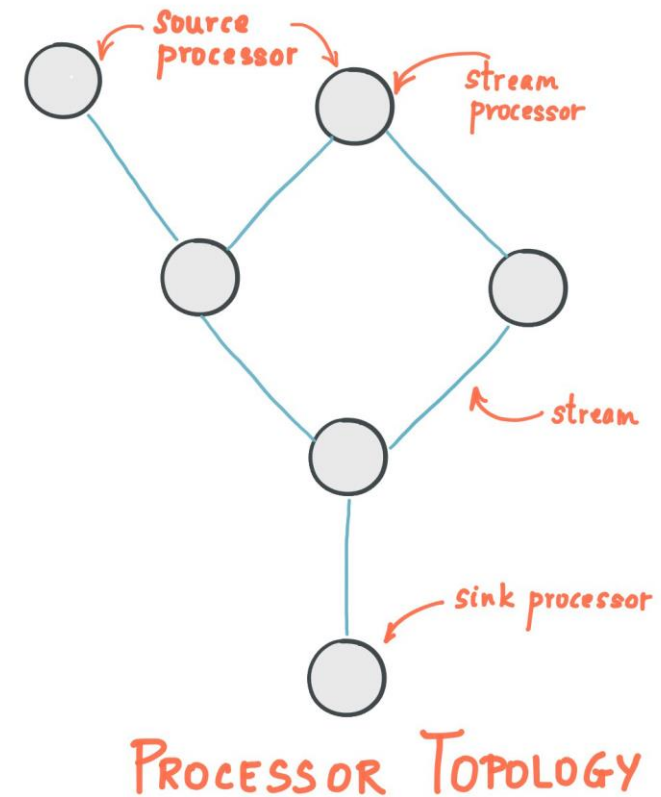
Anwendung von Streams

- Real Time Streaming Datenpipelines
 - Zuverlässiger Austausch von Daten zwischen Systemen oder Anwendungen
- Real Time Streaming Anwendungen
 - Auf Daten Streams reagieren oder diese transformieren

- Einfache Anwendung durch Java Library
- Ähnlich zu Spark Streaming, Apache Storm und Co.
- KafkaStreams DSL (map, flatMap, count, ...) vs. low-level Processor API
- Abstraktion: KStream, KTable, ksqlDB

```
public class WordCountApplication {  
  
    public static void main(final String[] args) throws Exception {  
        Properties props = new Properties();  
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application");  
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");  
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());  
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());  
  
        StreamsBuilder builder = new StreamsBuilder();  
        KStream<String, String> textLines = builder.stream("TextLinesTopic");  
        KTable<String, Long> wordCounts = textLines  
            .flatMapValues(textLine -> Arrays.asList(textLine.toLowerCase().split("\\W+")))  
            .groupBy((key, word) -> word)  
            .count(Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as("counts-store"));  
        wordCounts.toStream().to("WordsWithCountsTopic", Produced.with(Serdes.String(), Serdes.Long()));  
  
        KafkaStreams streams = new KafkaStreams(builder.build(), props);  
        streams.start();  
    }  
}
```

Quelle: <https://kafka.apache.org/documentation/streams/>



Quelle: <https://kafka.apache.org/20/documentation/streams/core-concepts>

- KStream
 - Abstraktion des Record Streams
 - Alle Messages werden als Insert betrachtet
- KTable
 - Abstraktion des Changelog Streams
 - Messages werden als Insert bzw. Update (sog. „Upsert“) betrachtet
- Beispiel: Summation
 - Nachricht 1: `{id: 'apple', value: 1}`
 - Nachricht 2: `{id: 'apple', value: 2}`
 - Ergebnis:
 - KStream: 3 (Summe der Records)
 - KTable: 2 (Update für Nachricht mit ID `'apple'`)

Beschreibung

- Kafka-Streams erlauben die „Echtzeit“-Auswertung und Transformation von Daten, die auf Kafka-Topics veröffentlicht werden
- In der Dokumentation wird die Verwendung zur Zählung der Worte im Stream verwendet:
<https://kafka.apache.org/documentation/streams/>

Aufgabenstellung:

- Starten Sie das Beispiel aus der Dokumentation
- Zählen Sie die Wörter im Topic „Hello World“ auf dem Broker `broker-1.k.anderscore.com` (Port 9092)

Hinweise:

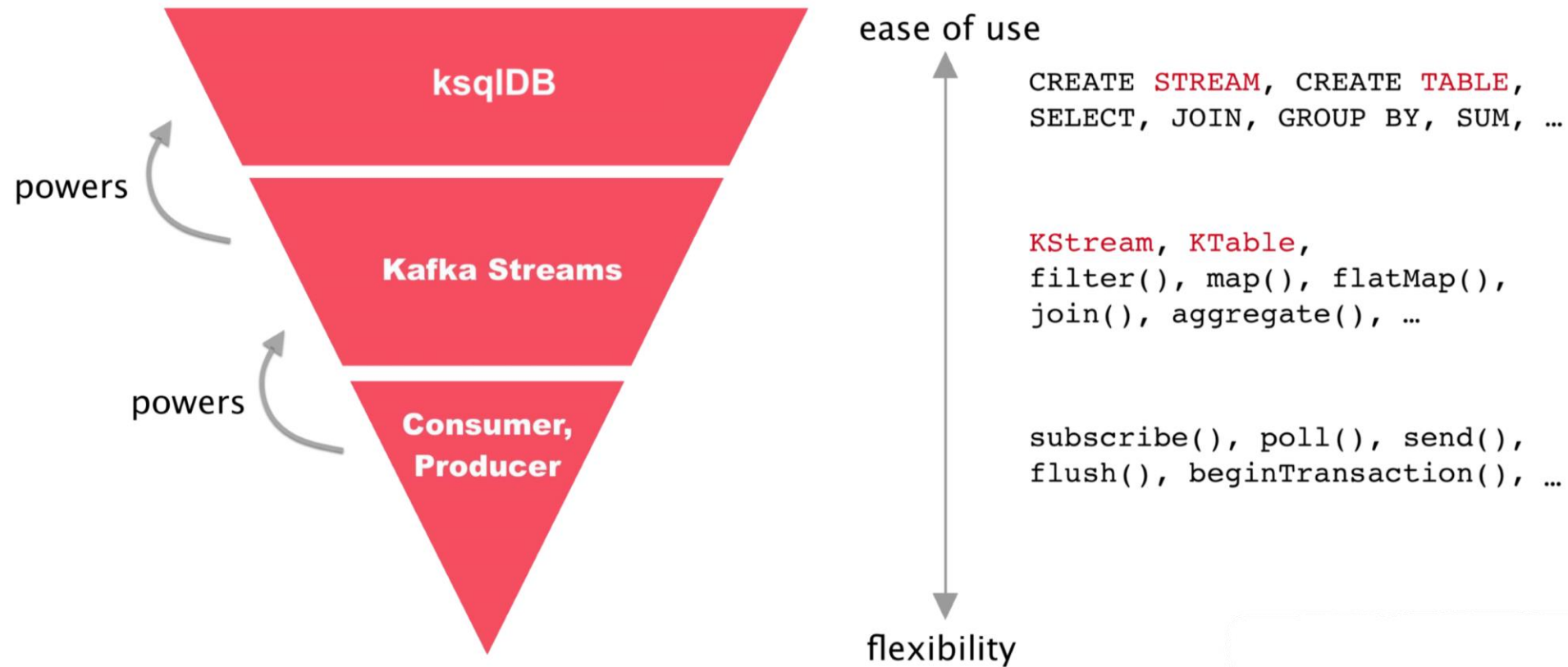
- Erzeugen Sie die Projektstruktur wie in Aufgabe 4
- Für Kafka-Streams wird eine weitere Maven Dependency benötigt:

```
<dependency>  
<groupId>org.apache.kafka</groupId>  
<artifactId>kafka-streams</artifactId>  
<version>2.0.0</version>  
</dependency>
```

- Die Streams aktualisieren sich, wenn Sie Nachrichten senden.
- Das Codebeispiel schreibt das Ergebnis in den Stream `WordsWithCountsTopic`, den Sie konsumieren müssen.
- Die Ergebnisse erscheinen innerhalb des eingestellten Intervalls – per default müssen Sie ein paar Sekunden warten.

Event Streaming Database

- Hilft beim Erstellen komplexer Stream Processing Anwendungen mit Kafka
- Abfragen, Lesen, Schreiben und Verarbeiten von Daten
- Deklarative Definition von Tabellen, Streams und Konnektoren
- Dynamische Joins
- Lightweight SQL Syntax
- SQL-like Interface
- Runtime: ksqlDB Engine
- Community Component von Confluent
- Alternative / Aufbau zu Kafkas Stream API
- Besser für:
 - Streaming ETL Pipelines
 - Reaktion auf kontinuierliche Real-Time Business Requests
 - Anomalien erkennen



Quelle: <https://docs.ksqldb.io/en/latest/concepts/ksqldb-and-kafka-streams>

ksqlDB:

```
CREATE STREAM fraudulent_payments AS  
SELECT fraudProbability(data) FROM payments WHERE fraudProbability(data) > 0.8  
EMIT CHANGES;
```

Kafka Streams:

```
public class FraudFilteringApplication {  
    StreamsBuilder builder = new StreamsBuilder();  
    KStream<String, Payment> fraudulentPayments = builder  
        .stream("payments-topic")  
        .filter((customer, payment) -> payment.getFraudProbability() > 0.8);  
    .to("fraudulent-payments-topic");  
  
    Properties config = new Properties();  
    config.put(StreamsConfig.APPLICATION_ID_CONFIG, "fraud-filtering-app");  
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");  
  
    new KafkaStreams(builder.build(), config).start();  
}
```

Nach: <https://docs.ksqldb.io/en/latest/concepts/ksqldb-and-kafka-streams>

Lektion 6 – Broker Operations

Config Files

- Jeder Broker hat eine Config Datei
 - `.property` file extension
 - Kann in der Datei oder programmatisch angepasst werden
 - Nutzt Key / Value Paare

Beispiel:

- `/usr/local/kafka/config/server.properties`

broker.id

- ID des Servers
- Bekommt ID automatisch, wenn nicht explizit gesetzt
- Vermeidung von Konflikten zwischen automatisch und explizit gesetzten IDs:
 - `reserved.broker.max.id + 1`

log.dirs

- Verzeichnis der log Daten
- Wenn nicht genutzt, wird stattdessen das Verzeichnis in `log.dir` benutzt

zookeeper.connect

- Definiert den Zookeeper connection string
 - `hostname:port`
- Um Verbindung zu anderen Zookeeper Nodes zu ermöglichen, falls eine down ist:
 - `hostname1:port1, hostname2:port2...`

- advertised.host.name
 - advertised.listeners
 - advertised.port
 - auto.create.topics.enable
 - auto.leader.rebalance.enable
 - background.threads
 - compression.type
 - control.plane.listener.name
 - delete.topic.enable
 - host.name
 - leader.imbalance.check.interval.seconds
 - leader.imbalance.per.broker.percentage
 - listeners
 - log.flush.interval.ms
 - log.flush.offset.checkpoint.interval.ms
 - log.flush.start.offset.checkpoint.interval.ms
 - log.retention.bytes/hours/minutes/ms
 - log.roll.hours/ms
 - log.roll.jitter.hours /ms
 - log.segment.bytes
 - log.segment.delete.delay.ms
 - message.max.bytes
 - min.insync.replicas
 - num.io.threads
 - num.network.threads
 - num.recovery.threads.per.data.dir
 - num.replica.alter.log.dirs.threads
 - offsets.metadata.max.bytes
 - offsets.commit.required.acks
 - offsets.topic.segment.bytes
 - port
 - queued.max.requests
 - quota.consumer.default
 - quota.producer.defaults
 - replica.fetch.min.bytes
 - request.timeout.ms
 - socket.receive.buffer.bytes
 - socket.request.max.bytes
 - transaction.max.timeout.ms
 - unclean.leader.election.enable
 - zookeeper.connection.timeout.ms
 - zookeeper.max.in.flight.requests
 - zookeeper.session.timeout.ms
- Gesamte Liste: <https://kafka.apache.org/documentation/#brokerconfigs>

Live Update

- Einige Configs können ohne Broker Restart geändert werden
 - ready-only: nur mit Neustart
 - per-broker: dynamisch für jeden Broker einzeln
 - cluster-wide: dynamisch gesamtes Cluster oder einzelne Broker

Beispiel Live Update:

- Broker id 0
- Anzahl der Log Cleaner Threads

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --alter --add-config log.cleaner.threads=2
```

Overrides können auch später mit `add-config` konfiguriert werden

- Im folgenden Beispiel wird die `max.message.bytes` upgedated:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name my-topic  
--alter --add-config max.message.bytes=128000
```

- Overrides überprüfen mit `--describe`:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name my-topic --describe
```

- Override entfernen mit `--alter --delete-config`:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name my-topic  
--alter --delete-config max.message.bytes
```

Topic Level Einstellungen: Beispiele

- cleanup.policy
 - compression.type
 - delete.retention.ms
 - file.delete.delay.ms
 - flush.messages
 - flush.ms
 - follower.replication.throttled.replicas
 - index.interval.bytes
 - leader.replication.throttled.replicas
 - min./max.compaction.lag.ms
 - max.message.bytes
 - message.format.version
 - message.timestamp.difference.max.ms
 - message.timestamp.type
 - min.cleanable.dirty.ratio
 - min.insync.replicas
 - preallocate
 - retention.bytes
 - retention.ms
 - segmentet.bytes
 - segment.index.bytes
 - segment.jitter.ms
 - segment.ms
 - unclean.leader.election.enable
 - message.downconversion.enable
-
- Komplet: <https://kafka.apache.org/documentation/#topicconfigs>

Topic Konfiguration

- Relevante Konfiguration
 - Server Default
 - Per-Topic Override

Der Override kann beim Erstellen eines Topics in der Kafka Shell eingestellt werden

- `--config`
- Eine oder mehrere Einstellungen

```
> bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-topic --partitions 1 \  
--replication-factor 1 --config max.message.bytes=64000 --config flush.messages=1
```

Lektion 7 – Best Practices

Producer

- Automatische Retries möglichst abschalten (Reordering)
- acks = all für “Konsistenz über Verfügbarkeit” (vgl. CAP Theorem)
- Mehrere Bootstrap Server angeben (Fehlertoleranz)

Consumer

- Nicht zu viele Daten auf einmal pollen (Netzwerklast, OutOfMemoryError)
- Auto-Commit beachten und möglichst abschalten
- Topics mit Consumer Offsets richtig konfigurieren (Compaction und Deletion!)
- At-least-once-Semantik meist beste Wahl
- Mehrere Bootstrap Server angeben (Fehlertoleranz)

Nach: https://media.ccc.de/v/froscon2018-2213-apache_kafka_lessons_learned

Broker

- Defaults setzen und bei Bedarf überschreiben (Fehlervermeidung)

Topics

- Deletion vs. Compaction **fachlich** entscheiden
- retention.ms wirkt nur bei Deletion, max.cleanable.dirty.ratio nur bei Compaction
- Automatisiert anlegen (z.B. Skript)
- Richtwert: ~ 10 bis 30 Partitionen pro Topic

Streams

- Dead Letter Queue vorsehen (z.B. Topic)

Nach: https://media.ccc.de/v/froscon2018-2213-apache_kafka_lessons_learned

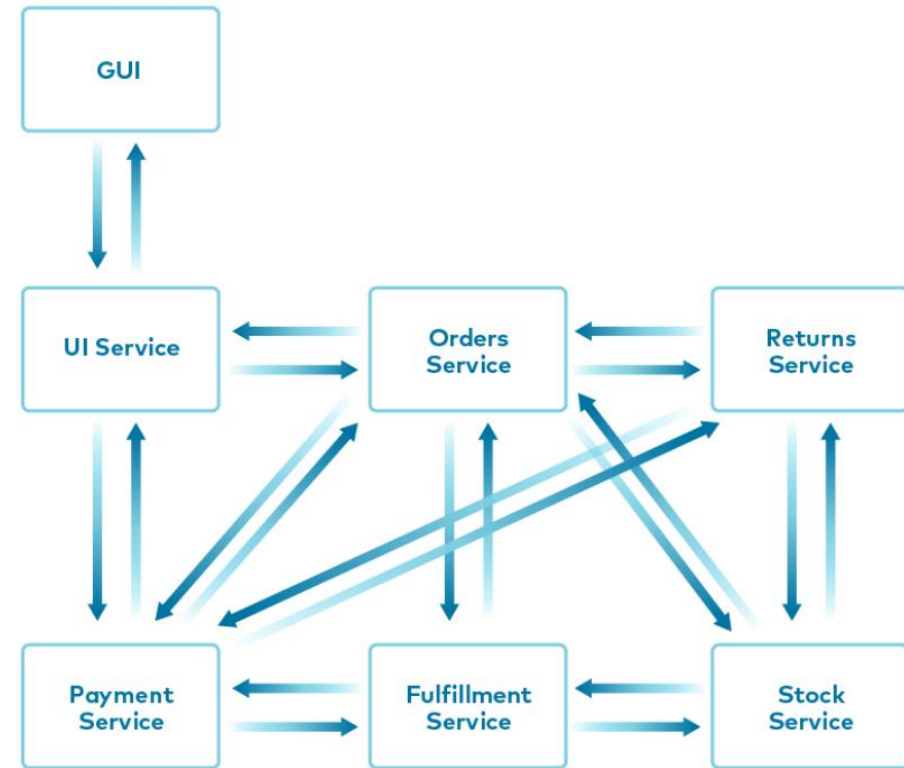
Lektion 8 – Ausblick

Microservices:

- Bequeme Abstraktion für Wiederbenutzbarkeit
- Kleine Services sind einfach austauschbar
- Potential für bessere Skalierbarkeit und Fehlertoleranz
- Komplette Unabhängigkeit von Services

Beispiel:

- Simple Business System in Microservice Architektur
- Event-basiert



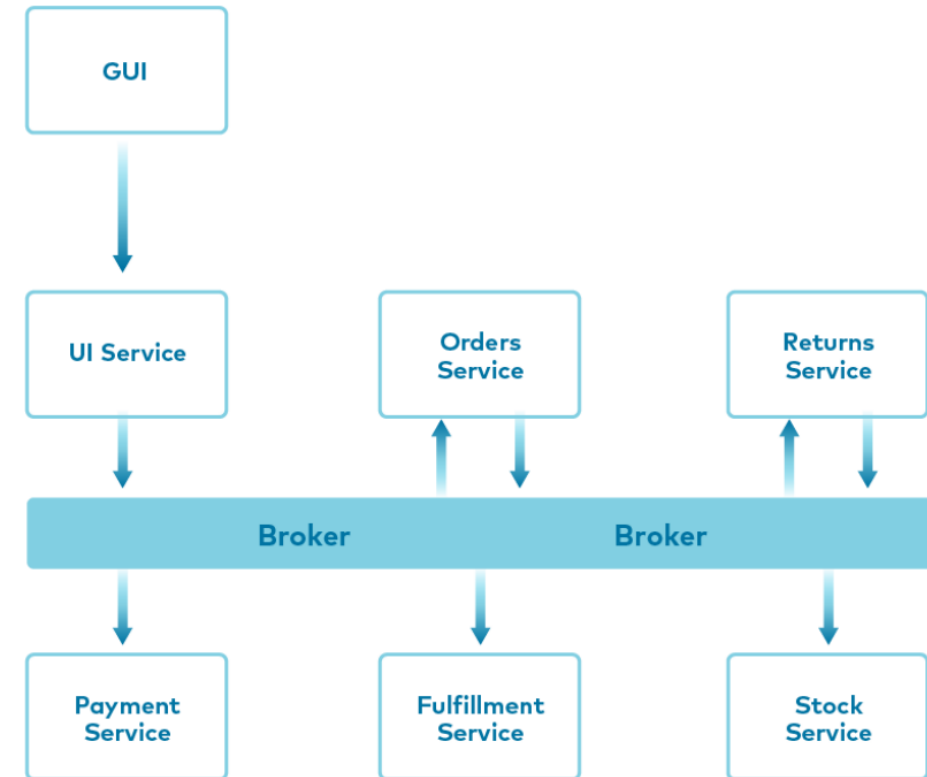
Quelle: WP Microservices in the Apache Kafka Ecosystem Confluent 2017

Microservice Environment

- Event-basiert
- Request / Response basiert
- **Hybrider Ansatz:**
 - Service Discovery
 - Synchrone Vorgänge
 - Asynchrone, Event-basierte Flows
 - Aber auch:
 - Anpassbarkeit
 - Skalierbarkeit

Praxisbeispiel:

- Asynchroner eMail Service mit Kafka Streams
- Order und Payment Stream joinen
- Ergebnis zu einem Lookup Table von Kunden joinen
- E-Mail zu jedem resultierenden Tuple versenden



Quelle: WP Microservices in the Apache Kafka Ecosystem Confluent 2017

Apache Avro

- Data Serialization System (& RPC, Container)
- Vergleichbar mit: Thrift, Protocol Buffers
- Aktuell: 1.10.2 (März 2021)
- Bindings: Java, Python, C, C++, C#, JavaScript, ... (3rd party)

Teil des Hadoop Projekts

- Serialization Format für persistente Daten
- Wire Format für Nodes und Clients

Schema für Daten in JSON

- Getrennt von den Nutzdaten
- Code-Generatoren (code-first, contract-first)
- Gemeinsames Schema für verschiedene µServices



```
{  
  "namespace": "example.avro",  
  "type": "record",  
  "name": "User",  
  "fields": [  
    {"name": "name", "type": "string"},  
    {"name": "favorite_number", "type": ["int", "null"]},  
    {"name": "favorite_color", "type": ["string", "null"]}  
  ]  
}
```

Quelle: https://en.wikipedia.org/wiki/Apache_Avro



- Confluent Schema Registry (OpenSource)
 - Historisierung, Java & RESTful API, Migration
 - Kein Transport mit den Daten (Performance)
 - Persistenz: Kafka-Topic
 - Command Line Client
- Avro als wire-format?
 - Einheitliches Schema für Daten zwischen Systemen
 - Pro:
 - Integration (confluent platform)
 - Contra:
 - XML & JSON deutlich weiter verbreitet

Beschreibung:

- Austausch von Nachrichten zwischen zwei Services über Kafka
- Serialisierung und Deserialisierung über Avro mit JSON-Schemata
- Generierung von Java Klassen aus Metadaten
- Die Confluent Platform enthält eine Avro Schema Registry, mit der Schemas unternehmensweit verwaltet werden können.
- Die OpenSource-Ausgabe der Confluent-Platform ist auf `broker-2.k.anderscore.com` installiert.
- Weitere Infos: <https://docs.confluent.io/current/schema-registry/docs>

Aufgabenstellung:

- Unter folgendem Link liegt ein Producer, welcher Personendaten (SteuerID, Name, Vorname) sendet:
<https://github.com/anderscore-gmbh/kafka-21.12/tree/master/Aufgabe6-Producer>
- Implementieren Sie einen Consumer, welcher die Daten de-serialisiert und auf der Konsole ausgibt.

Schemata:

<http://broker-2.k.anderscore.com:8081/subjects>

<http://broker-2.k.anderscore.com:8081/schemas/ids/1>

<http://broker-2.k.anderscore.com:8081/schemas/ids/2>

Hinweise:

- Verwenden Sie den Consumer aus Aufgabe 4 als Vorlage
- Die Schemata können Sie aus dem Producer übernehmen oder aus der Registry laden.
- Beachten Sie die pom.xml des Producers zu Abhängigkeit und Generator-Konfiguration
- Simulieren Sie einen Tippfehler: Benennen Sie das Feld firstName in fristName um und generieren Sie den Avro-Code für den Consumer erneut. Was passiert beim Start?
- Tipp: Ändern Sie die Consumer Group bei Bedarf
- Ein Beispiel für den Consumer finden Sie unter:

<https://github.com/anderscore-gmbh/kafka-21.12/tree/master/Aufgabe6-Producer>