Willkommen zum Training

anderScore
trust in competence

Apache Kafka:
Einführung in Event Streaming mit Java

Vorstellung

Daniel Krämer (M.Sc. Informatik)

- Senior Software Engineer
- Schwerpunkte
  - Software-Architektur
  - Integration und Migration
  - Clean Code
  - Testautomatisierung
  - DevOps
  - Trainings, Vorträge, Artikel
- Java, Spring, Kafka, Microservices, …

Vorstellung anderScore GmbH

Individuelle Anwendungsentwicklung - Java Enterprise, Web, Mobile

- seit 2005 ♦ in Köln ♦ für alle Branchen ♦ Goldschmiede anderScore
- nach Aufwand & zum Festpreis
- ✓ Digitalisierung / Prozesse / Integration
- ✓ Migration
- ✓ Neuentwicklung
- ✓ Notfall / kritische Situation
- ➔ pragmatisch, zielgerichtet, zuverlässig

Kompletter SW Life Cycle

- Projektmanagement / agile Methodik
- Anforderungsanalyse
- Architektur & SW-Design
- Implementierung & Testautomation
- Studien & Seminare

Einführung neues Versicherungsprodukt

Onlinebanking

Naturschutz: Kartierung bedrohter Arten

µservices Automotive

Stabilisierung Retail

Migration Energieversorger

Schnittstellen Gesundheitswesen

Adressverwaltung Logistik

Nutzerservices Energieversorger

Börsenhandel Wertpapiere

Security Härtung

EAM & Refactoring Leasing

... und für Sie? Sprechen Sie uns an!

Apache Kafka
Einführung in Event Streaming mit Java

10.12.2021

Daniel Krämer

Agenda

Begrüßung

1. Einführung
2. Paradigmen und Funktionsweise
3. Komponenten
4. Implementierung von Clients
5. Best Practices
6. Ausblick: Microservices, Apache Avro

Lektion 1 - Einführung

Inhalt und Ziel

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 (z.B. IntelliJ)

Zeitplan

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

Remote Training

- Video-Konferenz über Zoom
  - Bildschirmfreigabe für Folien
  - Breakout Rooms für Übungen
  - Lautsprecher + Mikrophon benötigt, Kamera empfehlenswert
- Entwicklung
  - Eigener Rechner
- Material auf GitHub

  https://github.com/anderscore-gmbh/kafka-21.12-2

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

Material

- 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

Vorstellung

Jetzt sind Sie dran!

- Name
- Vorwissen
- Erwartungen
- Themenwünsche

Ihre Umgebung

- Kafkacat
- JDK 8+
- Maven
- Entwicklungsumgebung (z.B. IntelliJ)
- Docker (optional)
- Docker Compose (optional)

Aufgabe 0 - Hello World mit 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: `sudo apt install kafkacat`
- Nachricht senden:

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

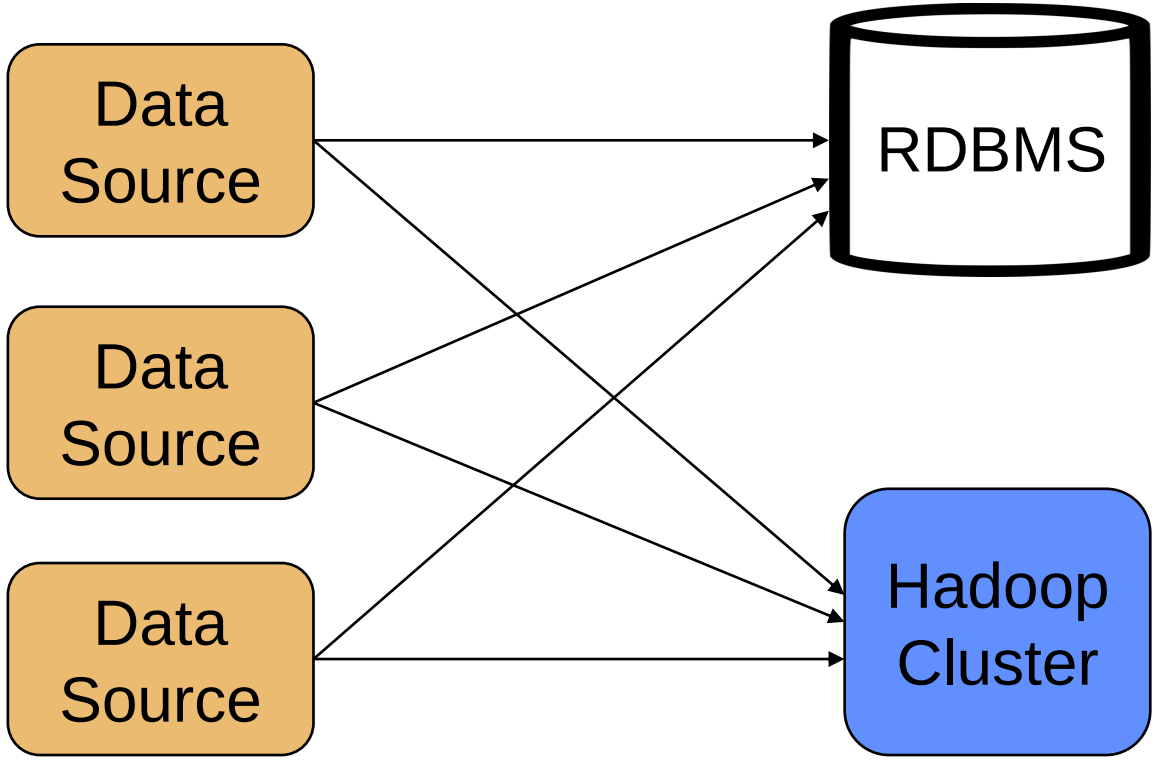
- Nachricht konsumieren:

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

Lektion 2 - Paradigmen und Funktionsweise von Kafka

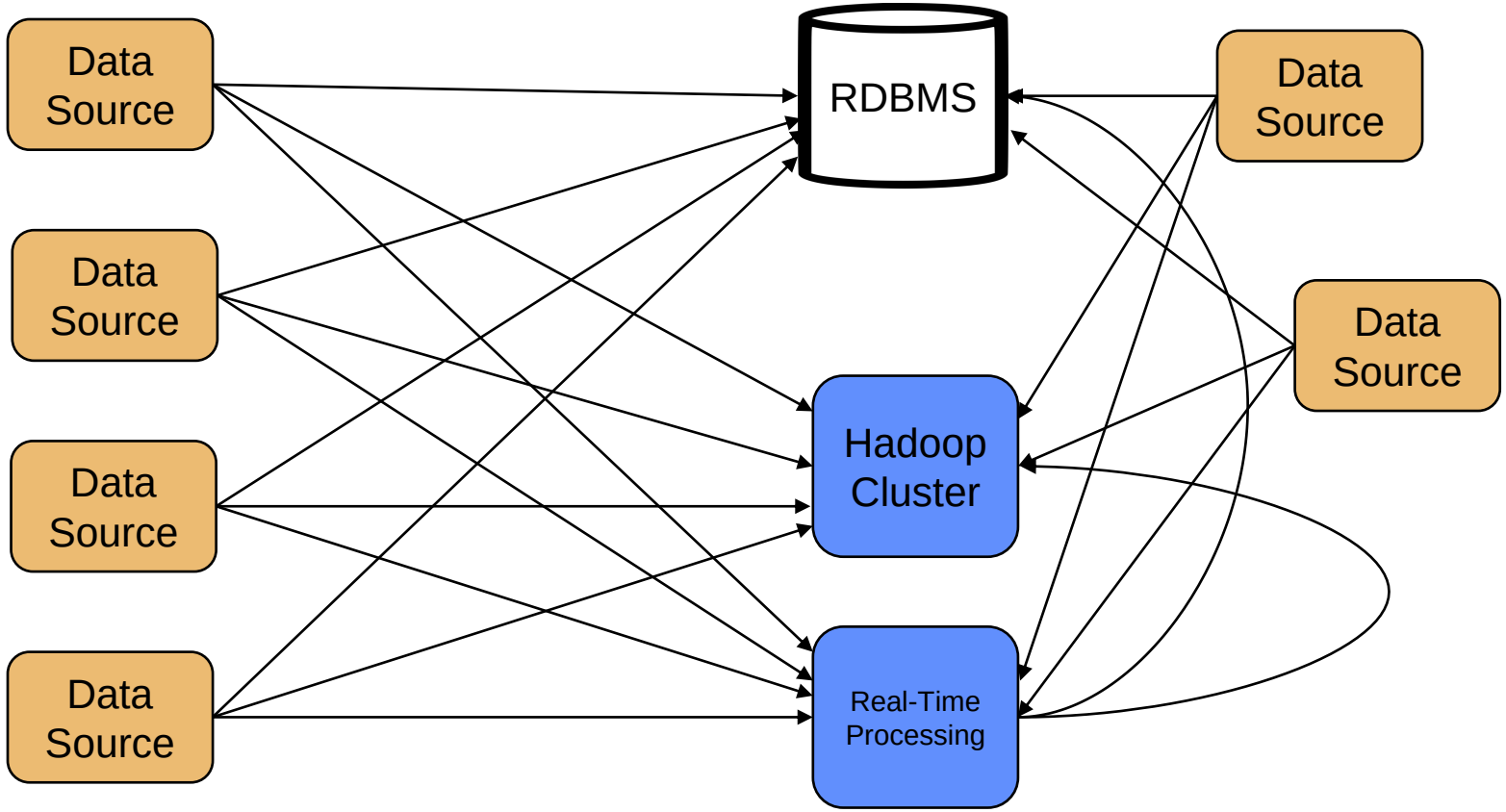
Was ist Kafka?

Einfache Datenverteilung:



Was ist Kafka?

Komplexe Datenverteilung:



Motivation für Kafka

Motivation

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

Ansatz

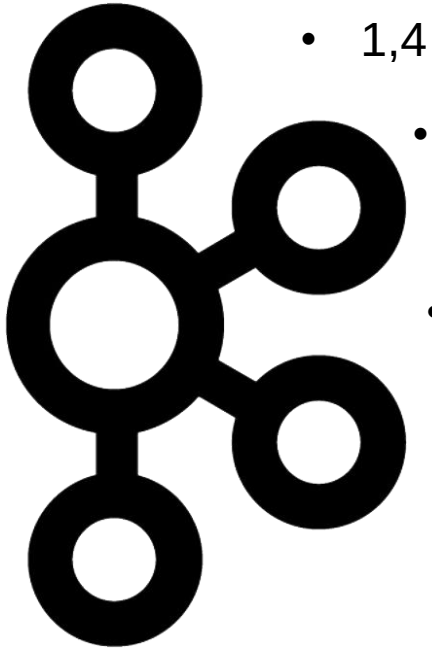
- Middleware für persistente Logs / Streams
- Ähnlich zu MQTT und Message Queueing

Ziele von Kafka

- Pipelines vereinfachen
- Data Stream Handling

Anstatt Batch Processing -> **Stream Processing**

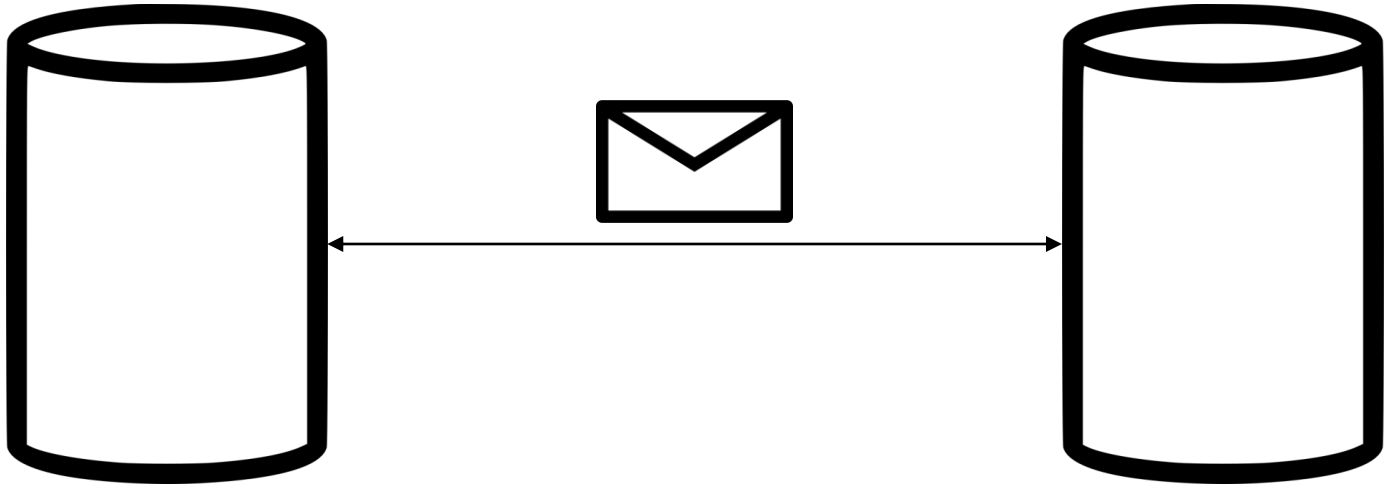
Apache Kafka



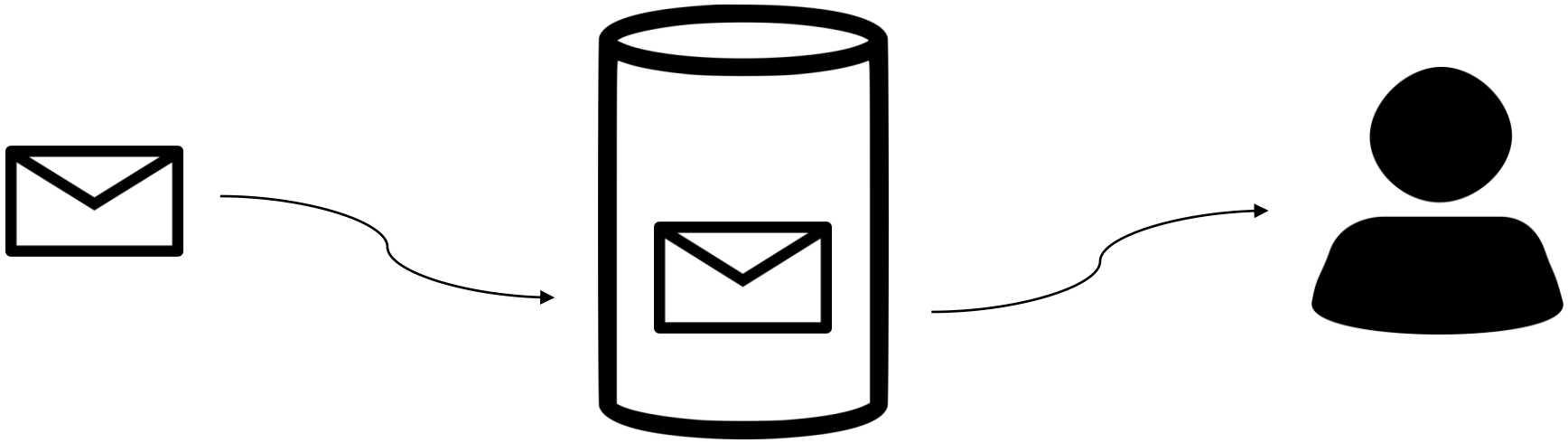
- LinkedIn (2010)
  - Teil der Core-Architektur
  - 1,4 Milliarden Nachrichten pro Tag
- Genutzt von:
  - IBM, Spotify, Uber, Hotel.com, Twitter…
- Use Cases:
  - Event Verarbeitung (*quasi* realtime)
  - Log Aggregation
  - Metriken & Analyse
  - Messaging / µService Kommunikation

- Keine Realtime- bzw. Echtzeitverarbeitung (Werbeversprechen)

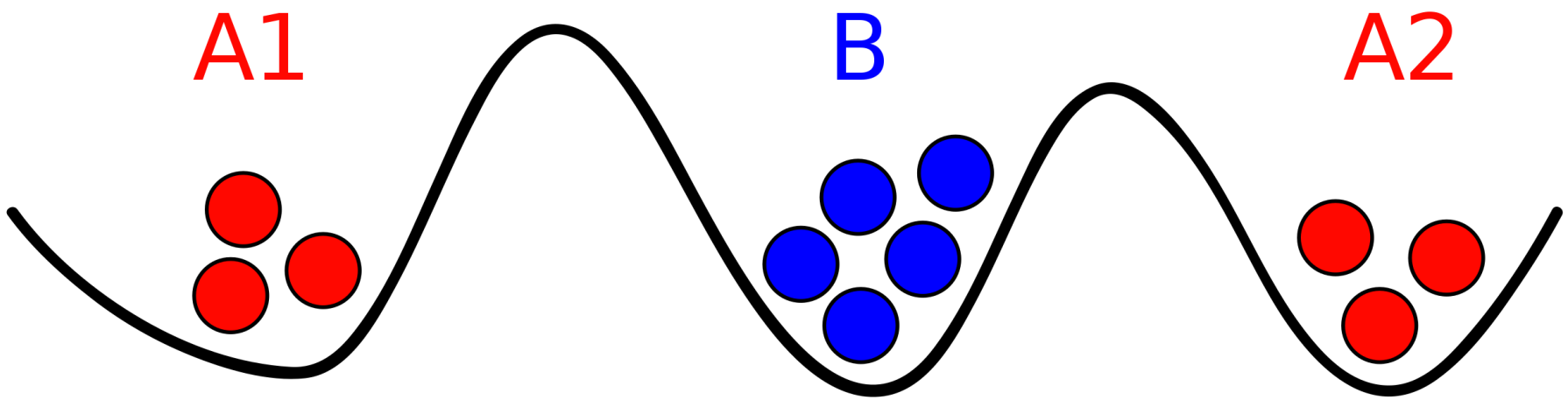
Synchrone Kommunikation



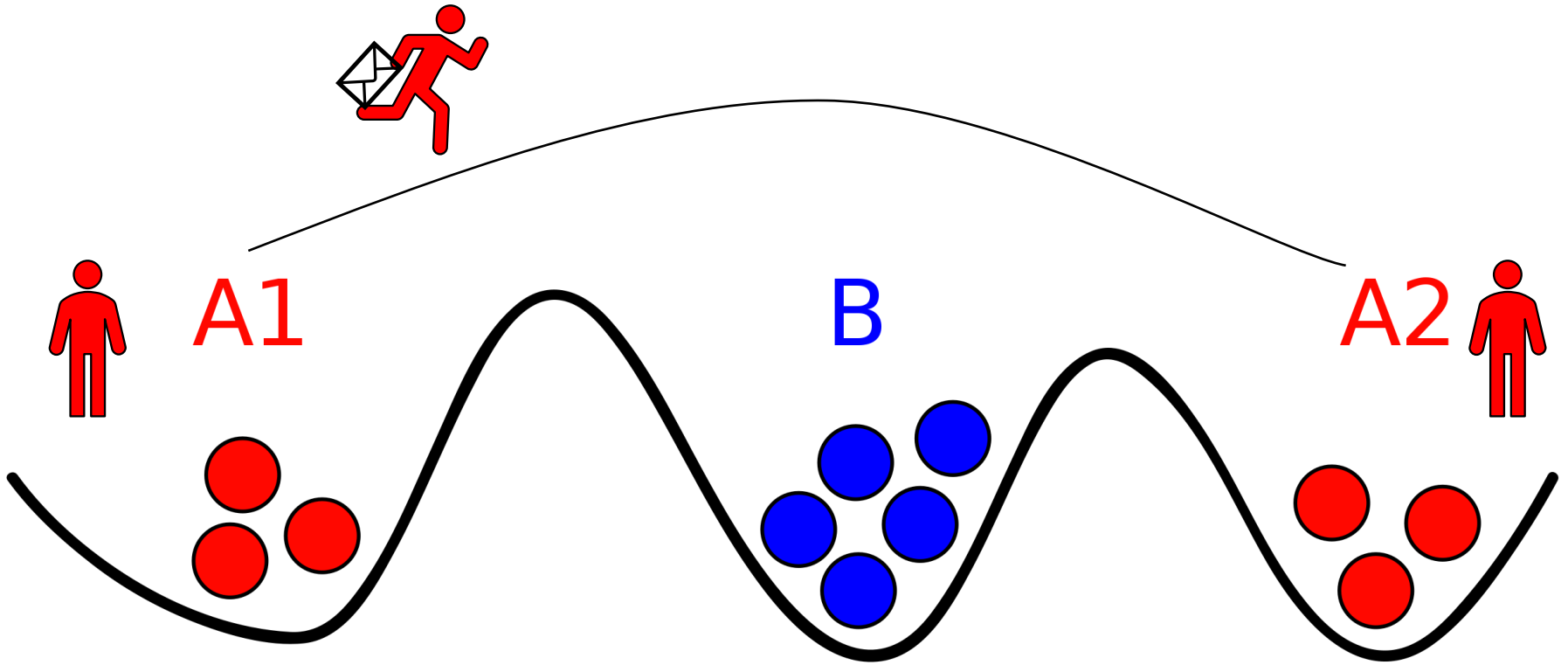
Asynchrone Kommunikation



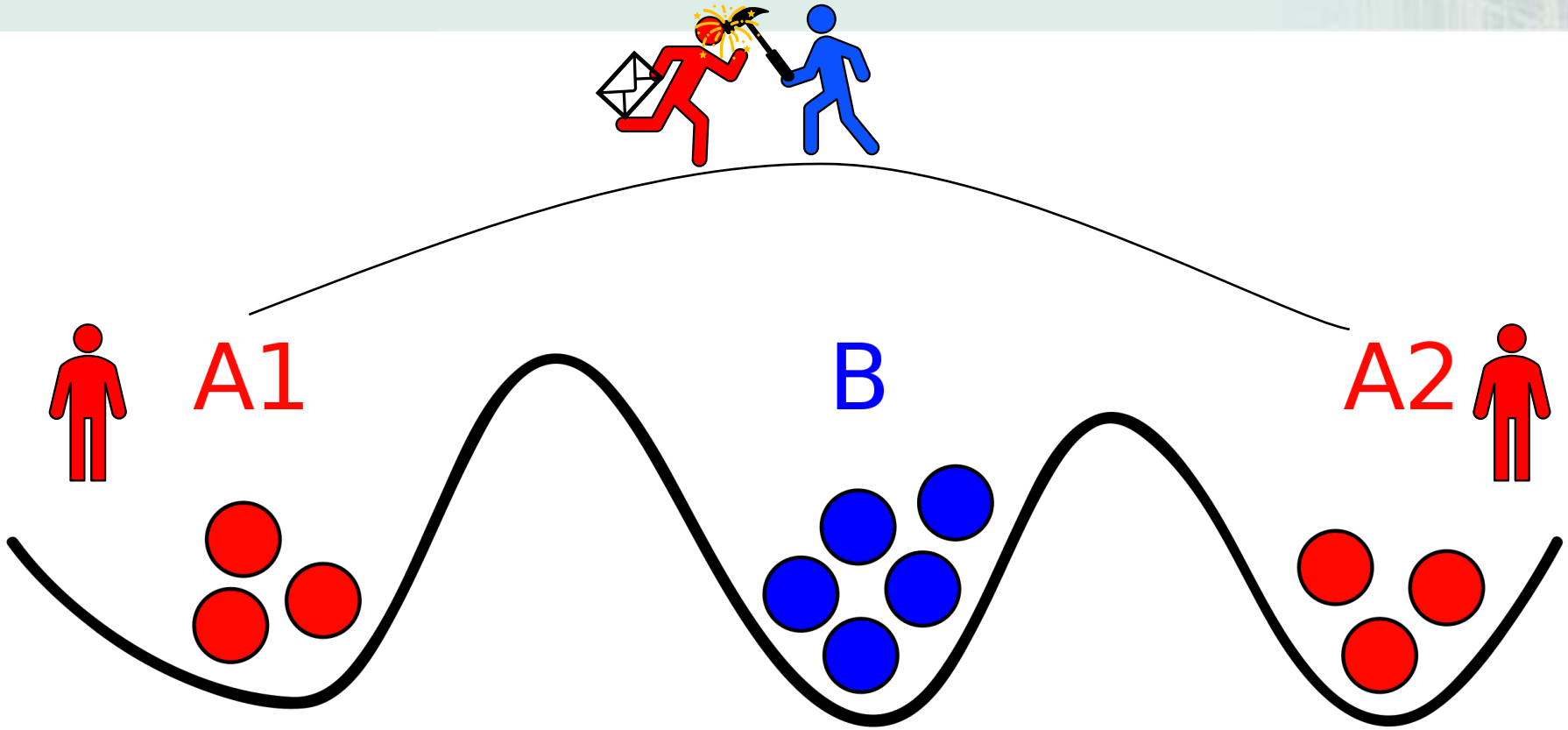
Two Generals Problem



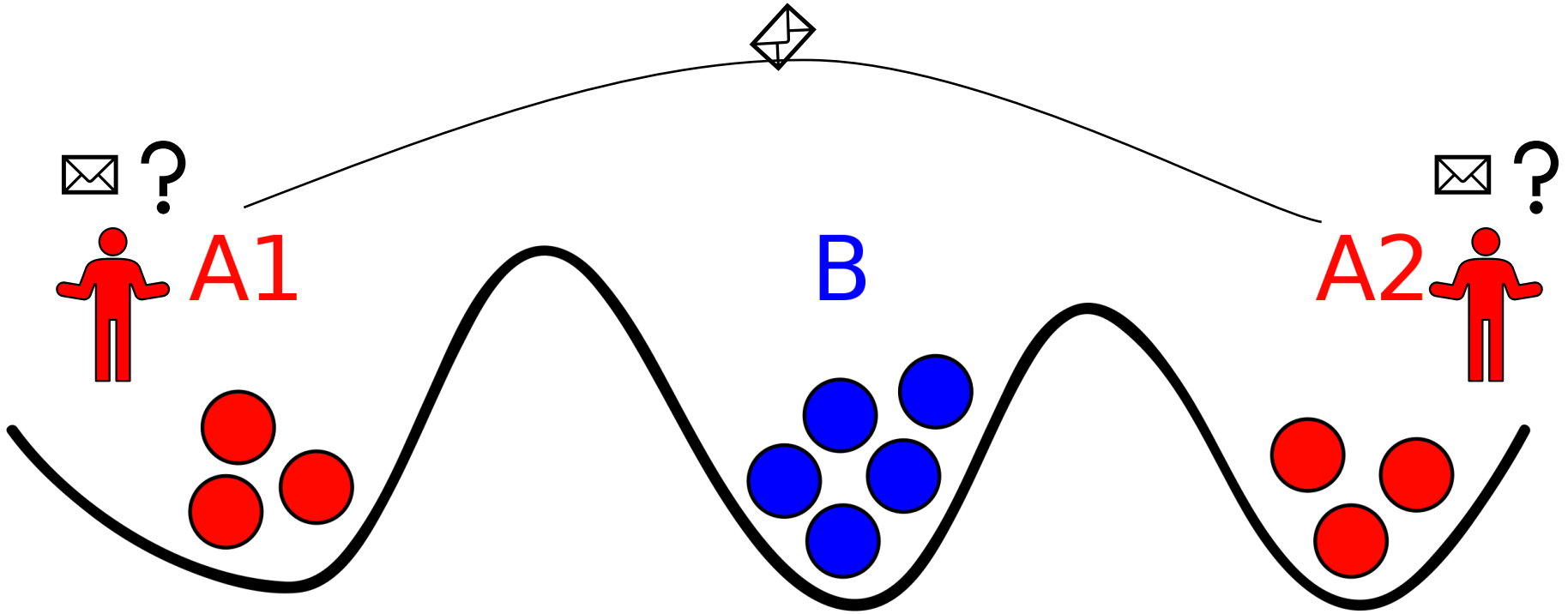
Two Generals Problem



Two Generals Problem



Two Generals Problem



Idempotenz

Definition:

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

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

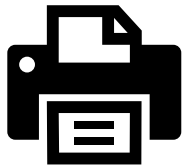
Semantik beim Aufruf

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 Beispiel: Print Server

At Least Once

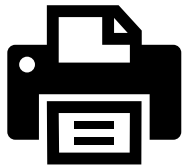


Situation: Druckfehler / Toner leer: Ausdruck zu hell!

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

At Most Once Beispiel: Print Server

At Most Once

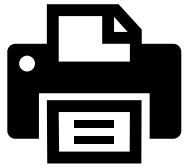


Situation: Druckfehler / Toner leer: Ausdruck zu hell!

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

Exactly Once Beispiel: Print Server

Exactly Once



Situation: Druckfehler / Toner leer: Ausdruck zu hell

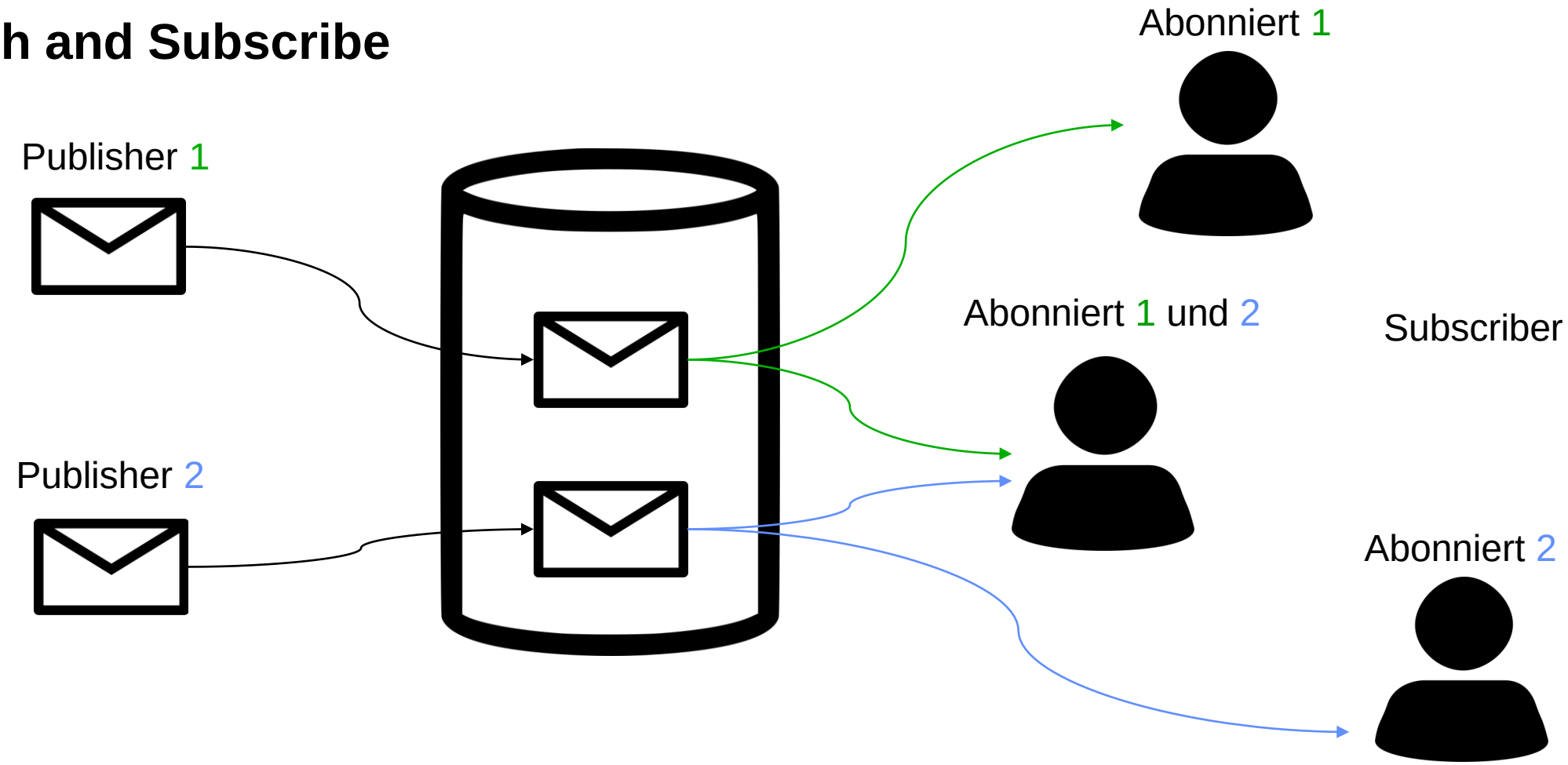
1. Fehler wird behoben
2. Genau die zu hellen Seiten werden gedruckt
3. … Perfekt ☺

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

https://www.confluent.io/de-de/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

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

Aufgabe 1 – Hello Kafka World

- **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

Aufgabe 1.1 - Hinweise Quickstart

anderScore trust in competence

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
```

Aufgabe 1.1 - Hinweise Quickstart

anderScore trust in competence

Topic erzeugen:

```
$ bin/kafka-topics.sh --create --topic HelloWorld --partitions 1 --replication-factor 1 --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
```

Aufgabe 1.1 - Hinweise Quickstart

anderScore trust in competence

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
```

Aufgabe 1.2 - Docker

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

Aufgabe 1.2 - Hinweise Docker

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 den Wert zu: 172.17.0.1
> vi docker-compose.yml
> docker-compose up -d

# Optional: Scalen Sie das Cluster, indem Sie mehr Broker hinzufügen
> docker-compose scale kafka=3

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

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

Aufgabe 1.2 - Hinweise Docker

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 1 --bootstrap-server `broker-list.sh`

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

- Hinweis: Eventuell muss *broker-list.sh* im Container der Kafka Shell erst noch angelegt werden. Sie finden die Datei im *kafka-docker* Git Repository.

Aufgabe 1.2 - Hinweise Docker

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'
```

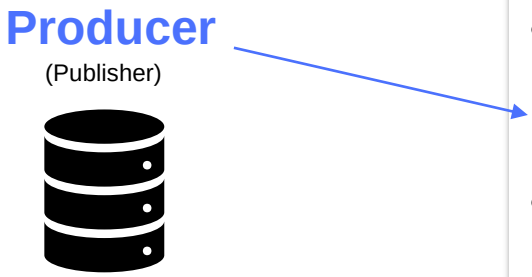
Hello Consumer

- 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

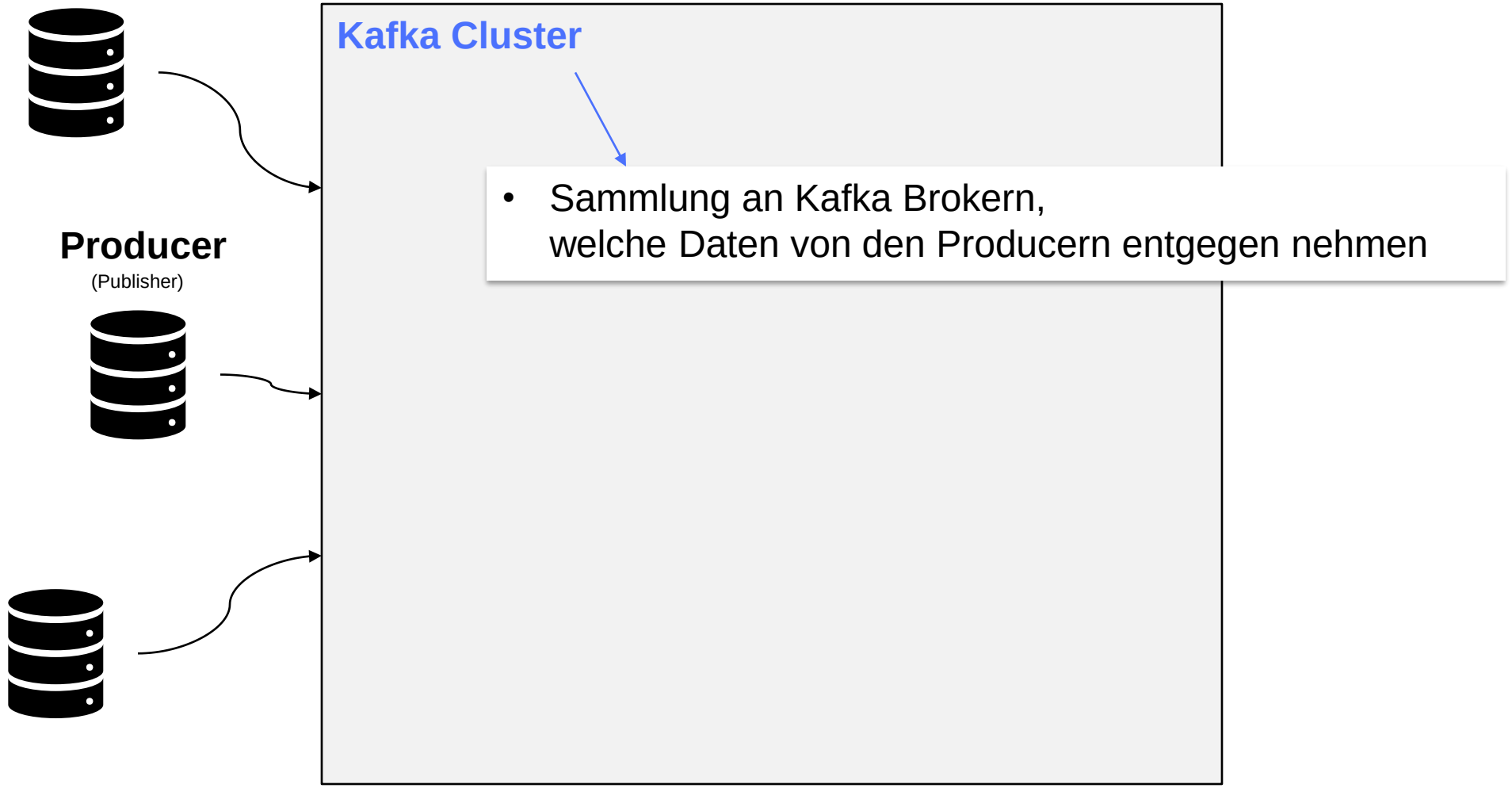
Kafka Komponenten: Producer



- Teil einer Anwendung
- Senden Daten an den Kafka Cluster
- Sharding der Messages (Partitionierung)
  - Über Hash Key oder Round Robin
  - Alternativ auch über eigene Strategie
- Für Load Balancing und Semantic Partitioning

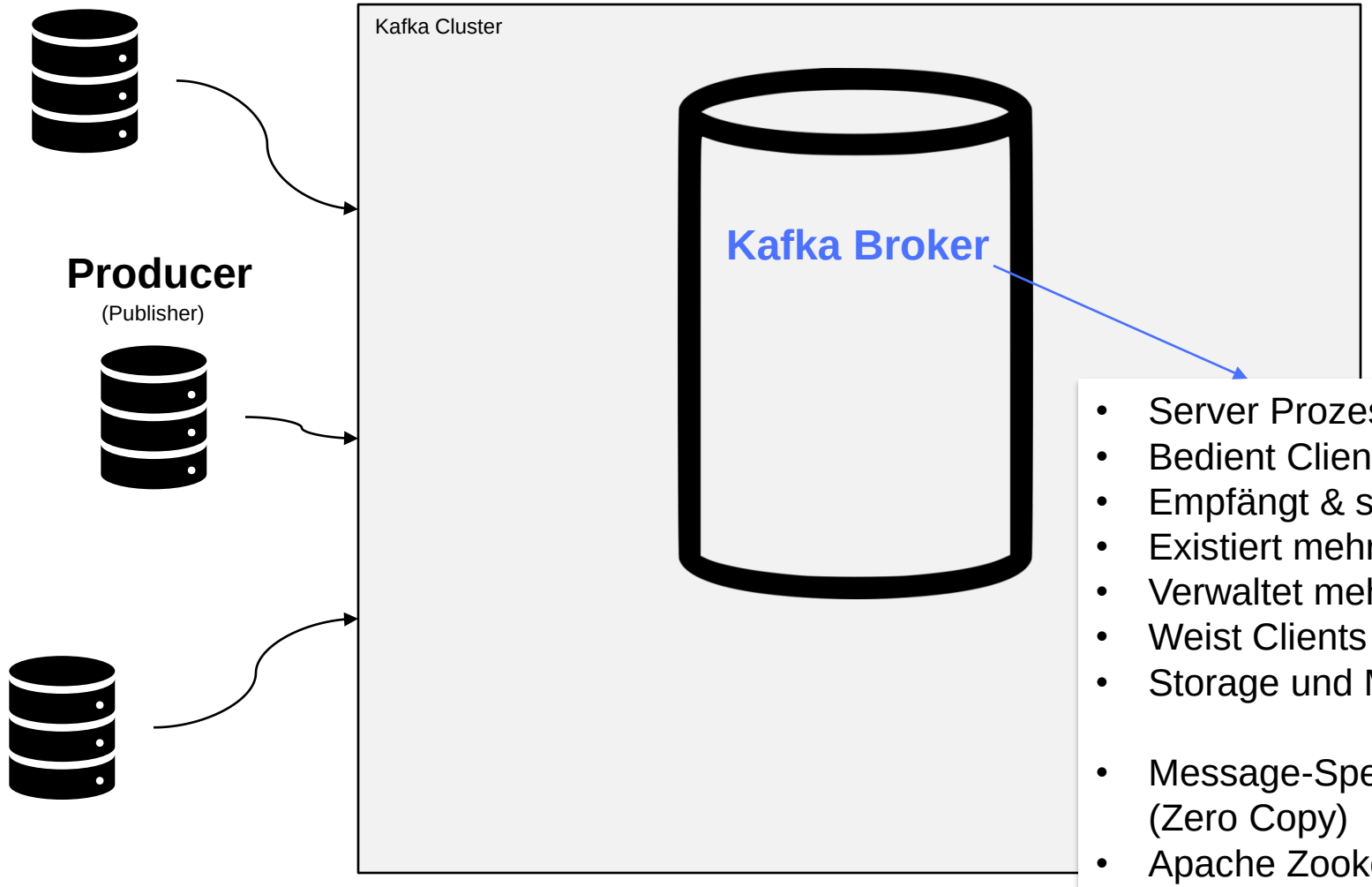
- Anbindung über:
  - Nativ: Java, C/C++, Python, Go, .Net, JMS
  - Rest (Confluent)
- Zusätzlich existieren Implementierungen für viele andere Sprachen

Kafka Komponenten: Cluster



- Sammlung an Kafka Brokern,
  welche Daten von den Producern entgegen nehmen

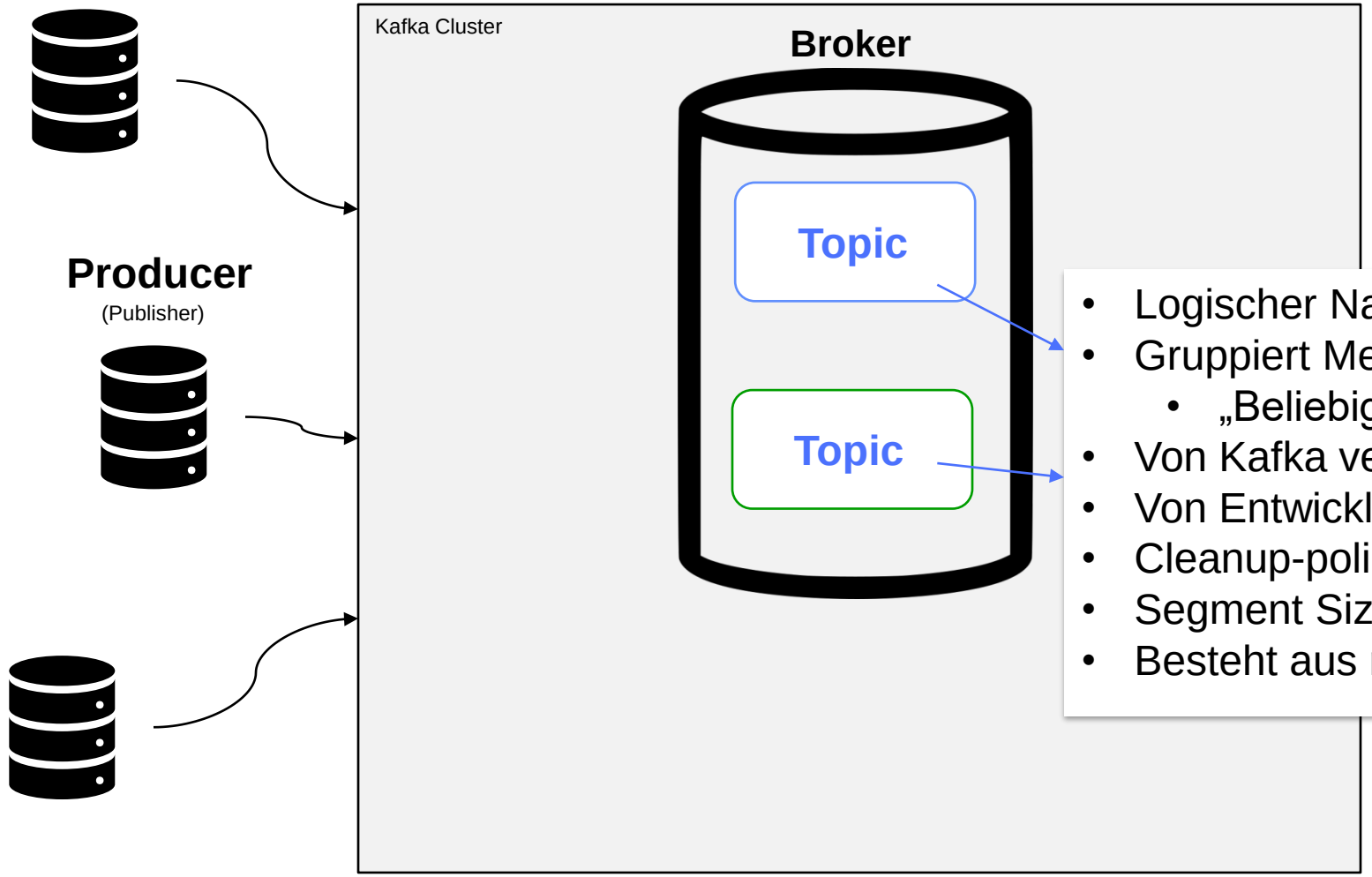
Kafka Komponenten: Broker



- Server Prozess (aktiv)
- Bedient Client Prozesse
- Empfängt & speichert Nachrichten
- Existiert mehrfach pro Cluster
- Verwaltet mehrere Topics & Partitionen
- Weist Clients individuelle Offsets zu
- Storage und Messaging Komponente

- Message-Speicher: Direkt auf Hard Disk (Zero Copy)
- Apache Zookeeper hält Quorum

Kafka Komponenten: Topic

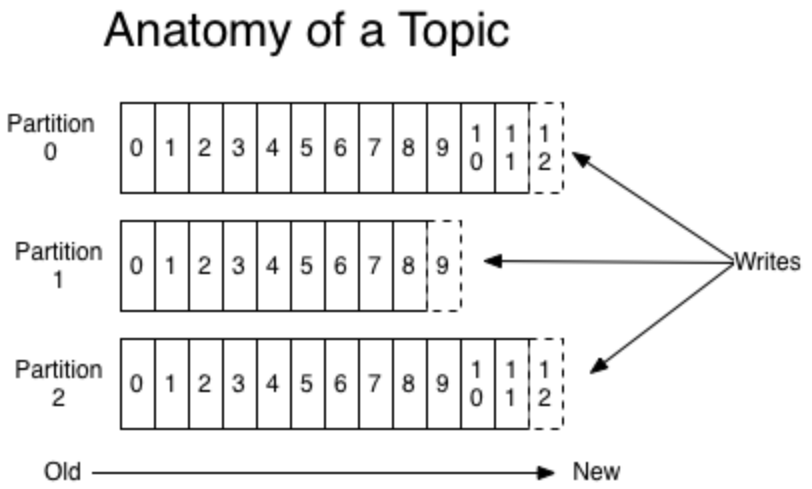


- Logischer Name eines Streams
- Gruppiert Messages
  - „Beliebig“ viele pro Cluster
- Von Kafka verwaltet
- Von Entwickler(in) administriert
- Cleanup-policy: compact vs. delete
- Segment Size: head vs. tail
- Besteht aus mehreren Partitions

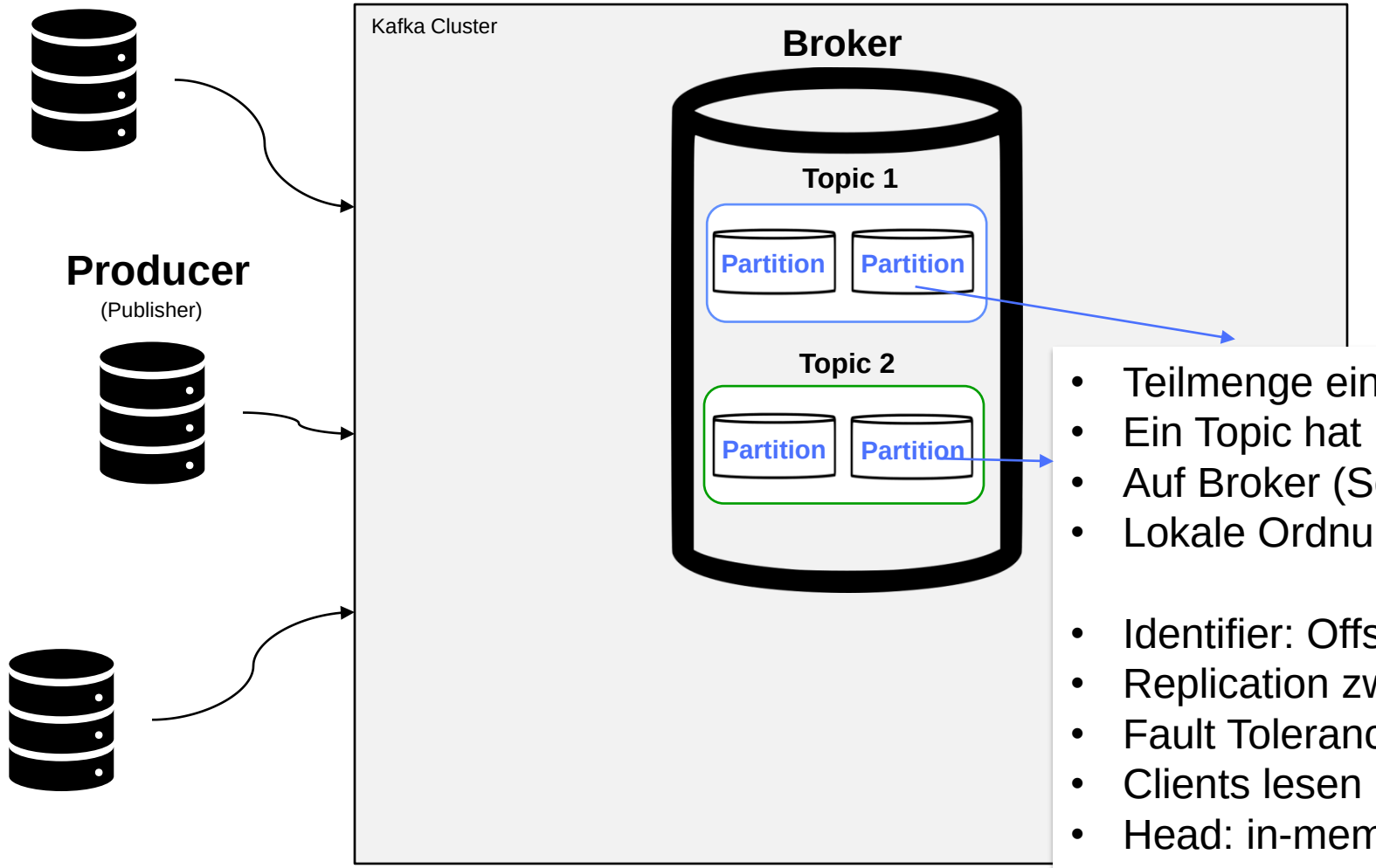
Anatomie eines Topics

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



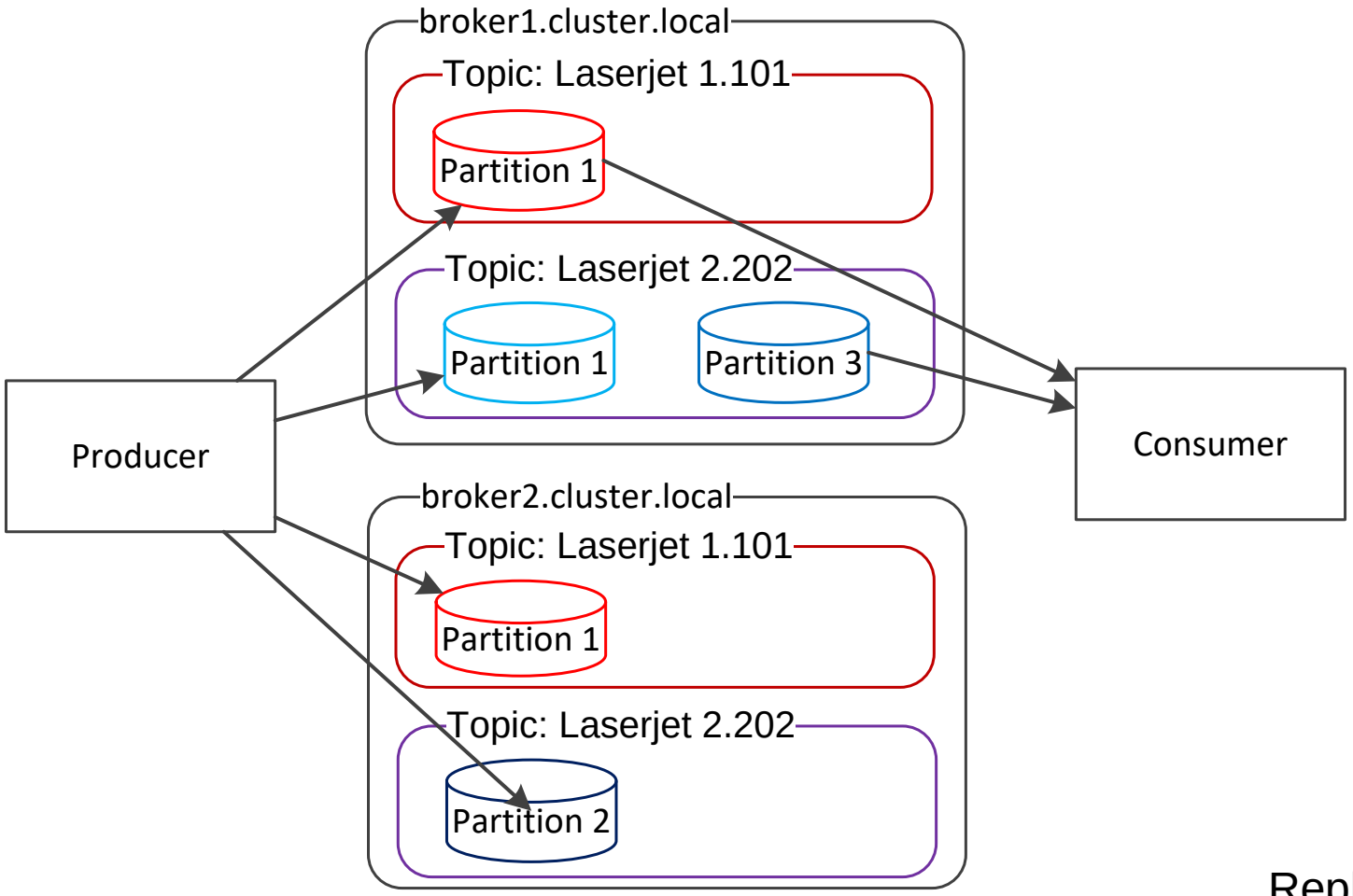
Kafka Komponenten: Partition



- Teilmenge eines Topics
- Ein Topic hat 1-n Partitions
- Auf Broker (Server) verteilt und repliziert
- Lokale Ordnung, append-only

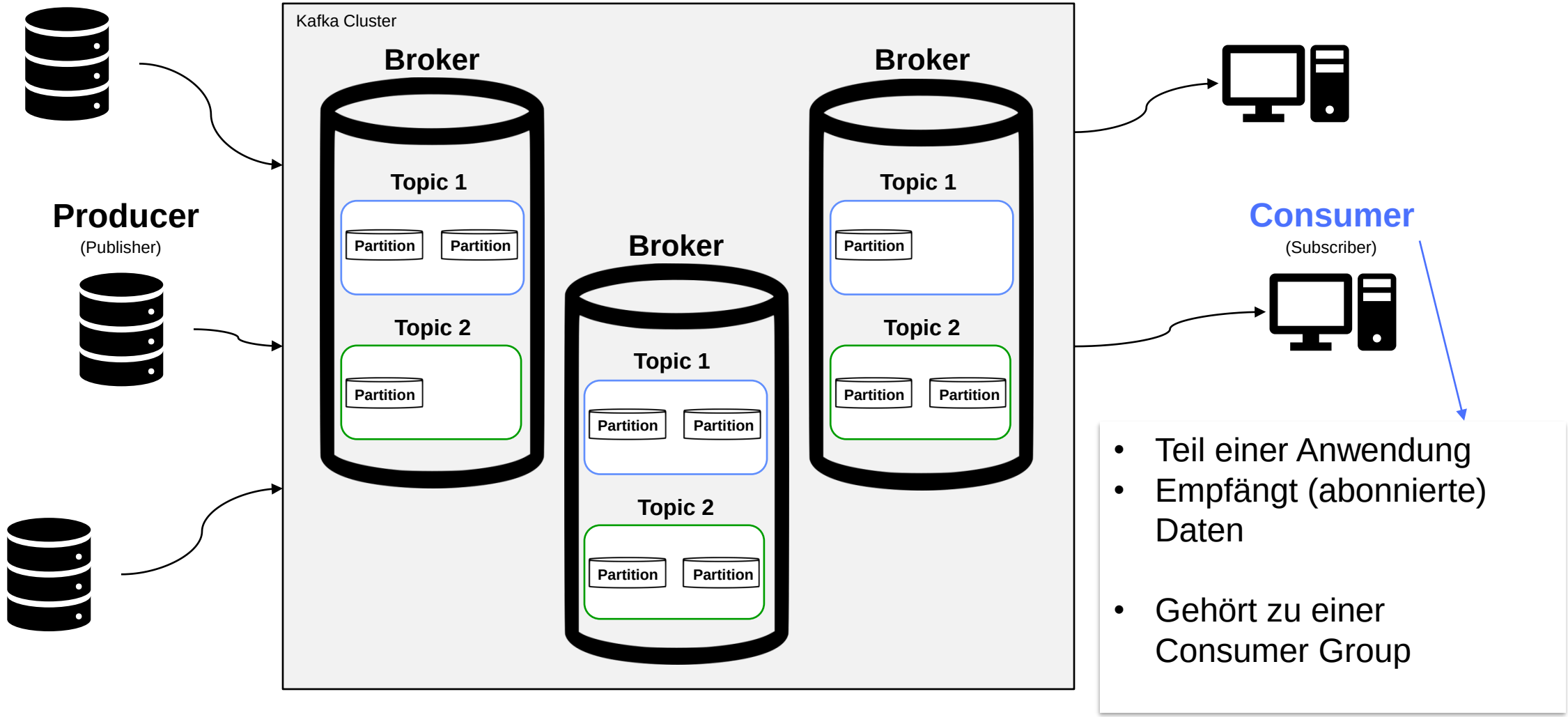
- Identifier: Offset einer Message, wachsend
- Replication zwischen Brokern möglich
- Fault Tolerance
- Clients lesen nur vom Leader
- Head: in-memory
- Tail: persistent (Deletion & Compaction)

Partitionen: Printer Beispiel



Repliziert: Laserjet 1.101
Sharding: Laserjet 2.202

Kafka Komponenten: Consumer

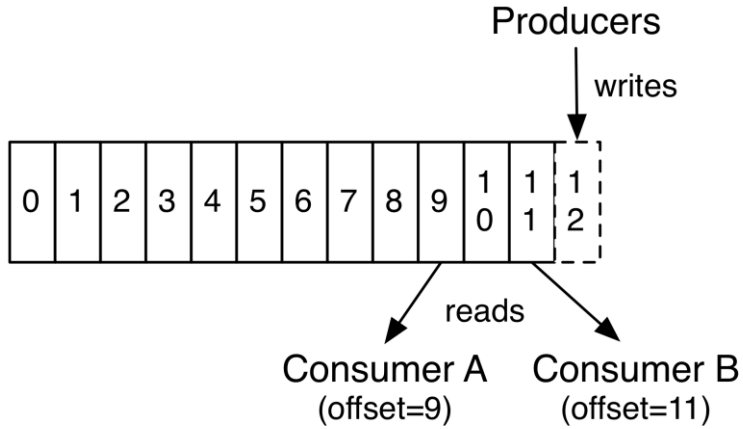


- Teil einer Anwendung
- Empfängt (abonnierte) Daten
- Gehört zu einer Consumer Group

Kafka Komponenten: Consumer

- Abruf von Messages via **pull, single-threaded**

- **Consumer Offset**
  - *Nächste* zu lesende Nachricht
  - Pro Consumer und Partition
  - Speicher: spezielles internes 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 (lokale Transaktion)

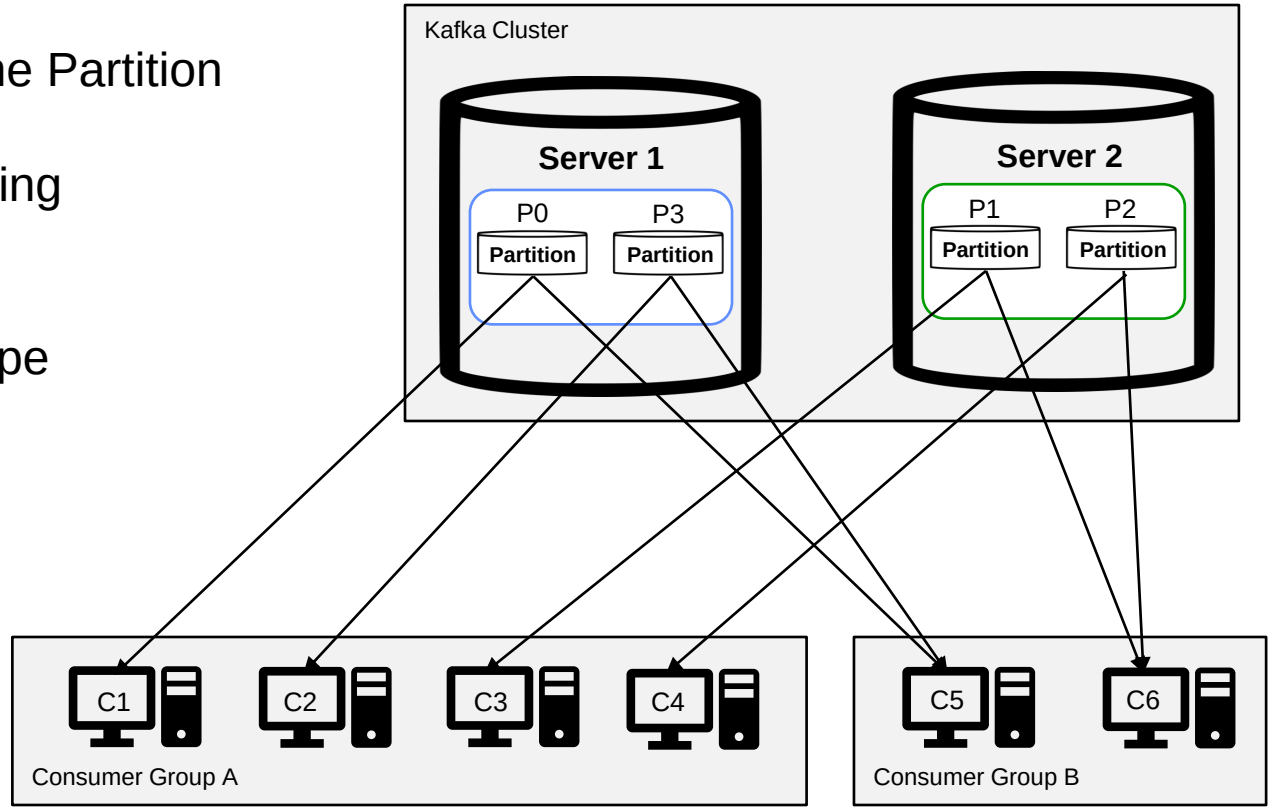
- **Verschiedene Consumer**
  - Gleichzeitiges Lesen möglich
  - Default: Alle Nachrichten im Topic an alle Consumer
  - Spezialfall: *Consumer Group*

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

Kafka Komponenten: Consumer Groups

- Gruppierung: **Consumer Group**
  - Mehrere Consumer zusammenfassen -> 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)



Zookeeper

Was ist **Zookeeper**?

- Open Source von Apache
- Ermöglicht verteilte Koordination
- Kümmert 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

Zookeeper in Kafka

Kafka Broker nutzen **Zookeeper** für:

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

Aufgabe 2 – Topics und Partitionen

anderScore trust in competence

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?

Aufgabe 2 - Hinweise

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
```

Aufgabe 2 - Hinweise

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 – Implementierung von Clients

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

Java Producer API

anderScore trust in competence

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();
```

Producer Eigenschaften

- **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 Leader geschrieben wurde
  - **acks = all**: Producer wartet, bis alle in-sync Replikate das Erhalten der Datensätze bestätigt haben

Nachricht an Kafka senden

- 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
```

Buffer beim Versenden

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
```

Callbacks

anderScore trust in competence

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());
                        }
                    }
                });
```

Java Consumer API

anderScore trust in competence

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());
}
```

Consumer Eigenschaften

- **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);
```

Nachricht aus Kafka lesen

- **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
```

Aufgabe 3 – Producer und Consumer

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

Aufgabe 3 - Hinweise

anderScore trust in competence

- 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>3.0.0</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
```

Lektion 5 – Best Practices

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 (erfordert Idempotenz!)
- Mehrere Bootstrap Server angeben (Fehlertoleranz)

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

Best Practices

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 6 – Ausblick

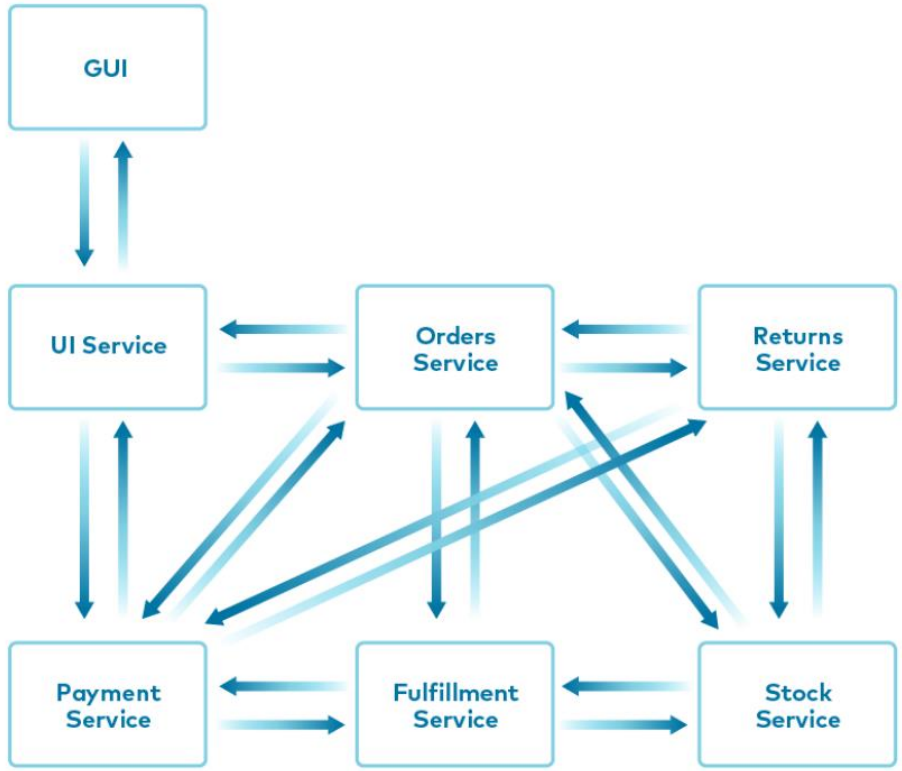
Microservice Architektur

Microservices:

- Fachlicher Schnitt (Bounded Contexts)
- Weniger Verantwortlichkeiten -> leichter austauschbar
- Potential für bessere Skalierbarkeit und Fehlertoleranz
- Lose Kopplung zwischen Services

Beispiel:

- Simples Business System in Microservice Architektur
- Asynchrone Kommunikation über Messages
- Eventbasiert



Quelle: WP Microservices in the Apache Kafka Ecosystem Confluent 2017

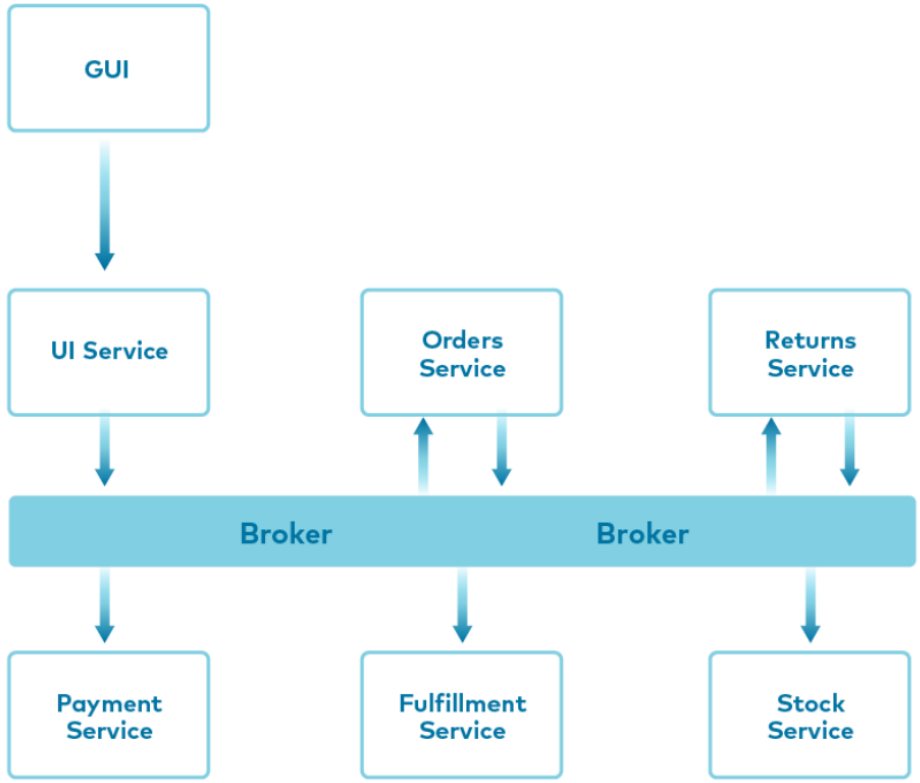
Microservices mit Kafka

Microservice Environment

- Eventbasiert
- Request / Response basiert
- **Hybrider Ansatz:**
  - Service Discovery
  - Synchrone Vorgänge
  - Asynchrone, Eventbasierte 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 Touple versenden



Quelle: WP Microservices in the Apache Kafka Ecosystem Confluent 2017

Apache Avro - Serialization

Apache Avro

- Data Serialization System (& RPC, Container)
- Vergleichbar mit: Thrift, Protocol Buffers
- Aktuell: 1.10.2 (März 2021)
- Bindings: Java, Pyhton, 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

Apache Avro – Beispiel für Schemata

```
{
  "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

Apache Avro – Integration mit Kafka

- 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