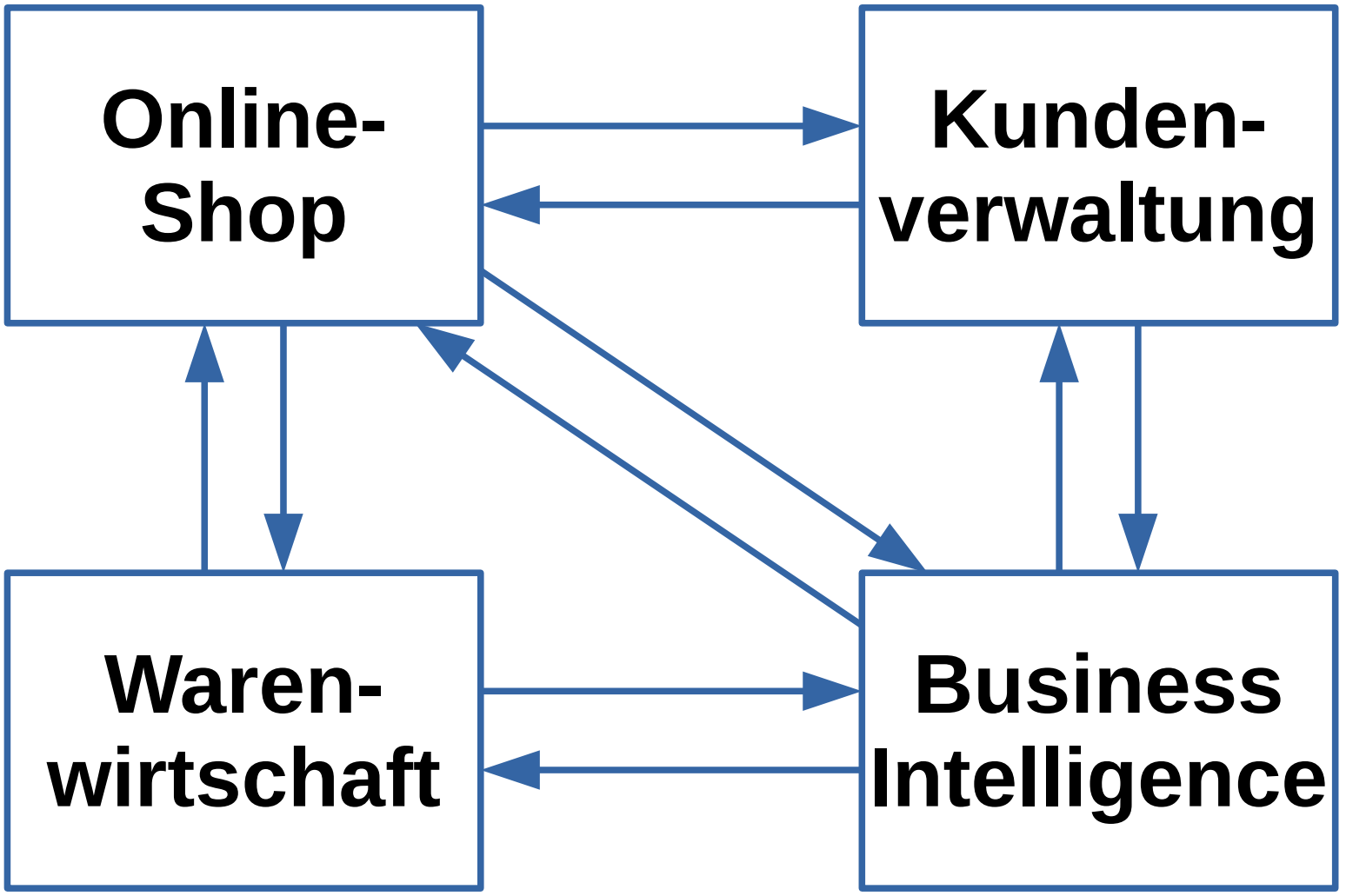
Big-Data-Technologien

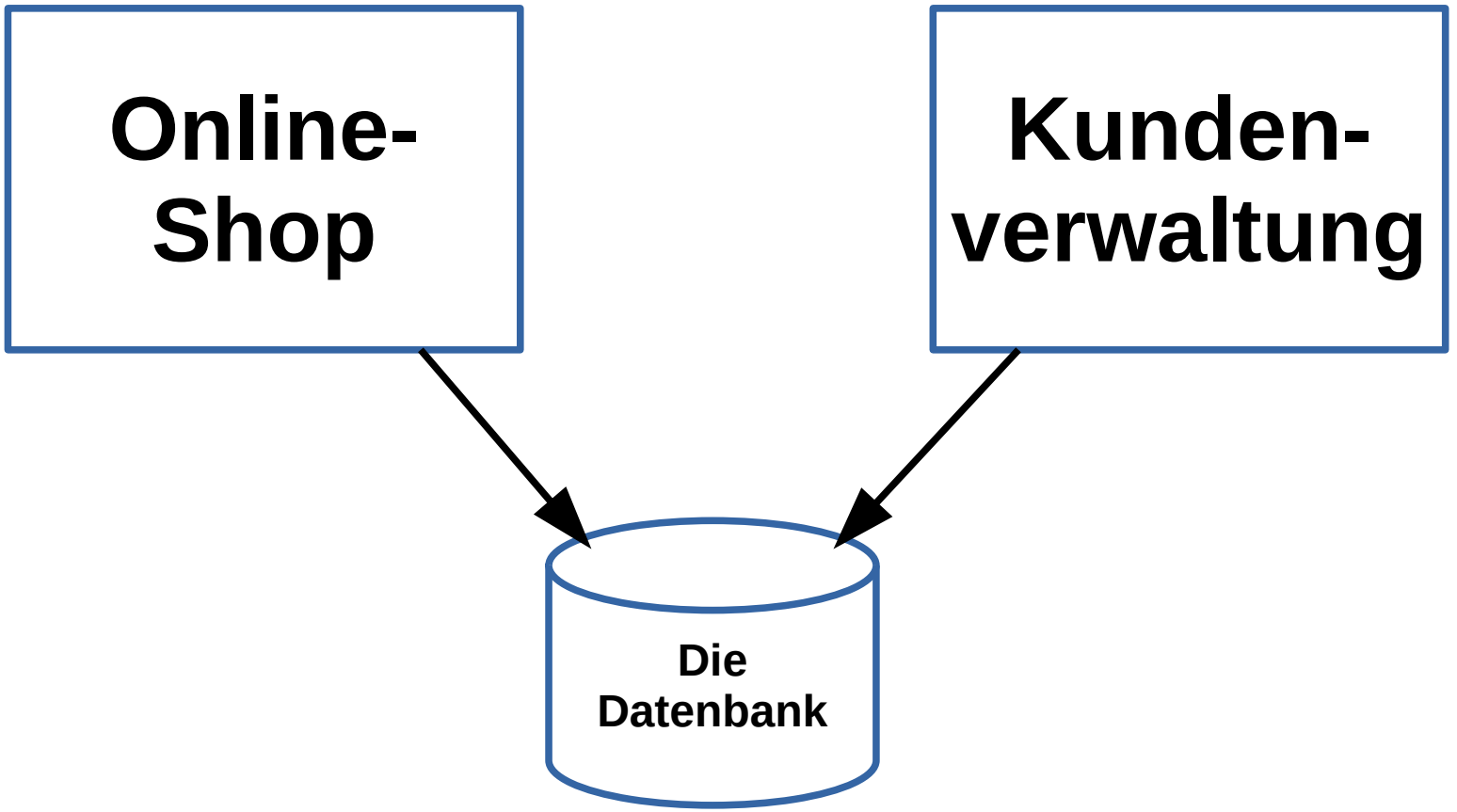
Kapitel 14: Messaging

Hochschule Trier
Prof. Dr. Christoph Schmitz

Enterprise Application Integration

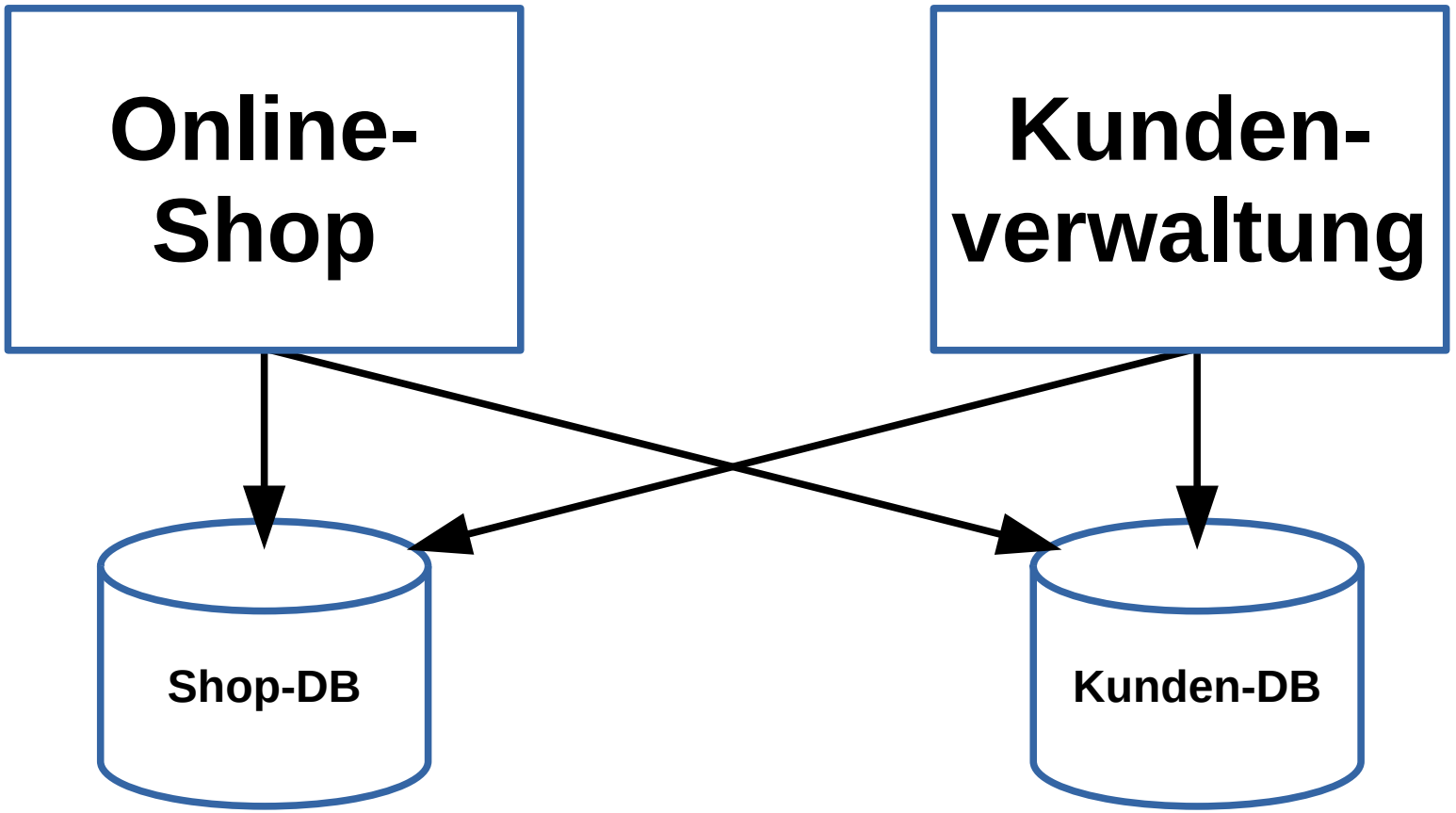


Enterprise Application Integration



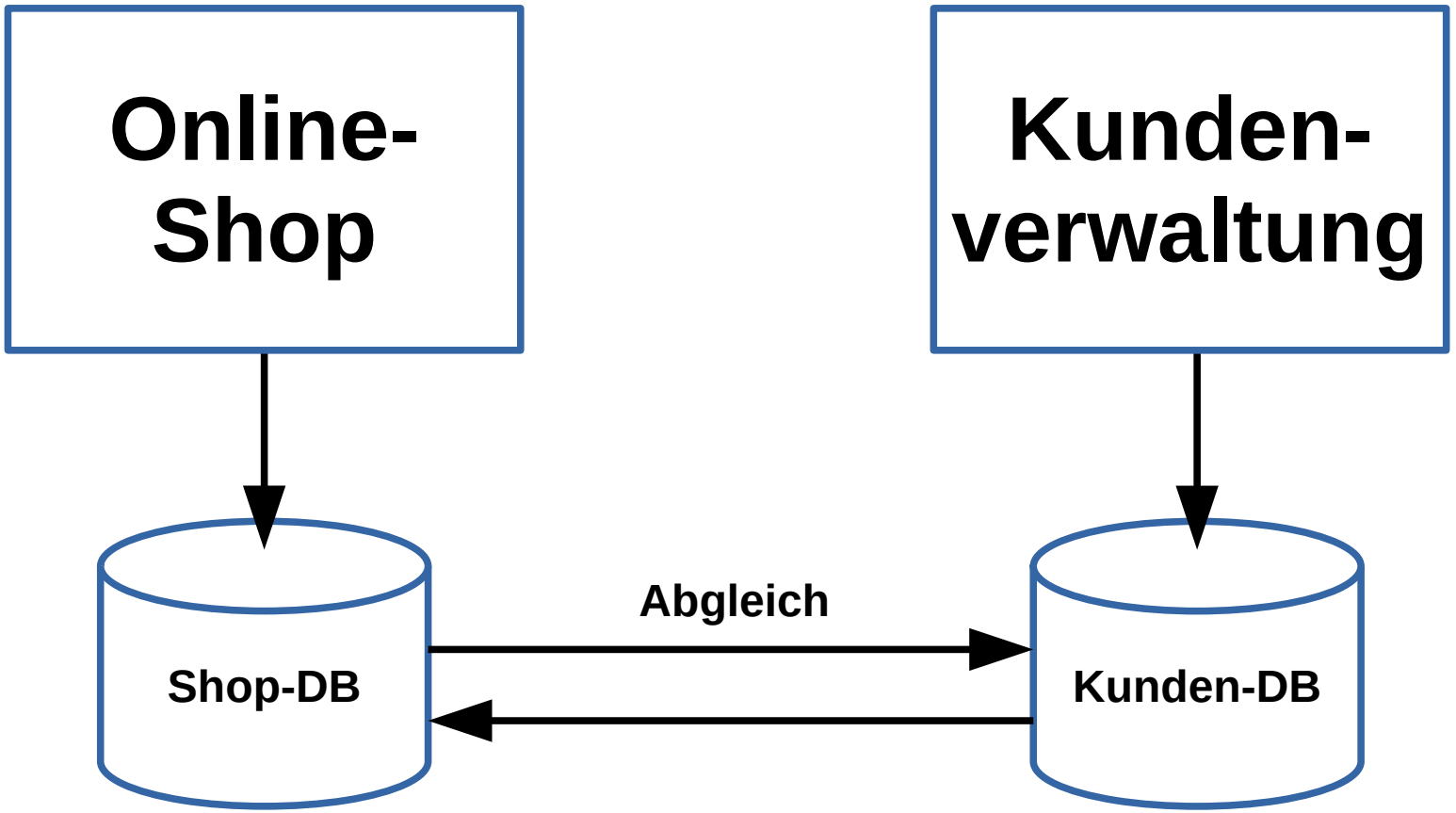
"Die Datenbank als Schnittstelle"

Enterprise Application Integration

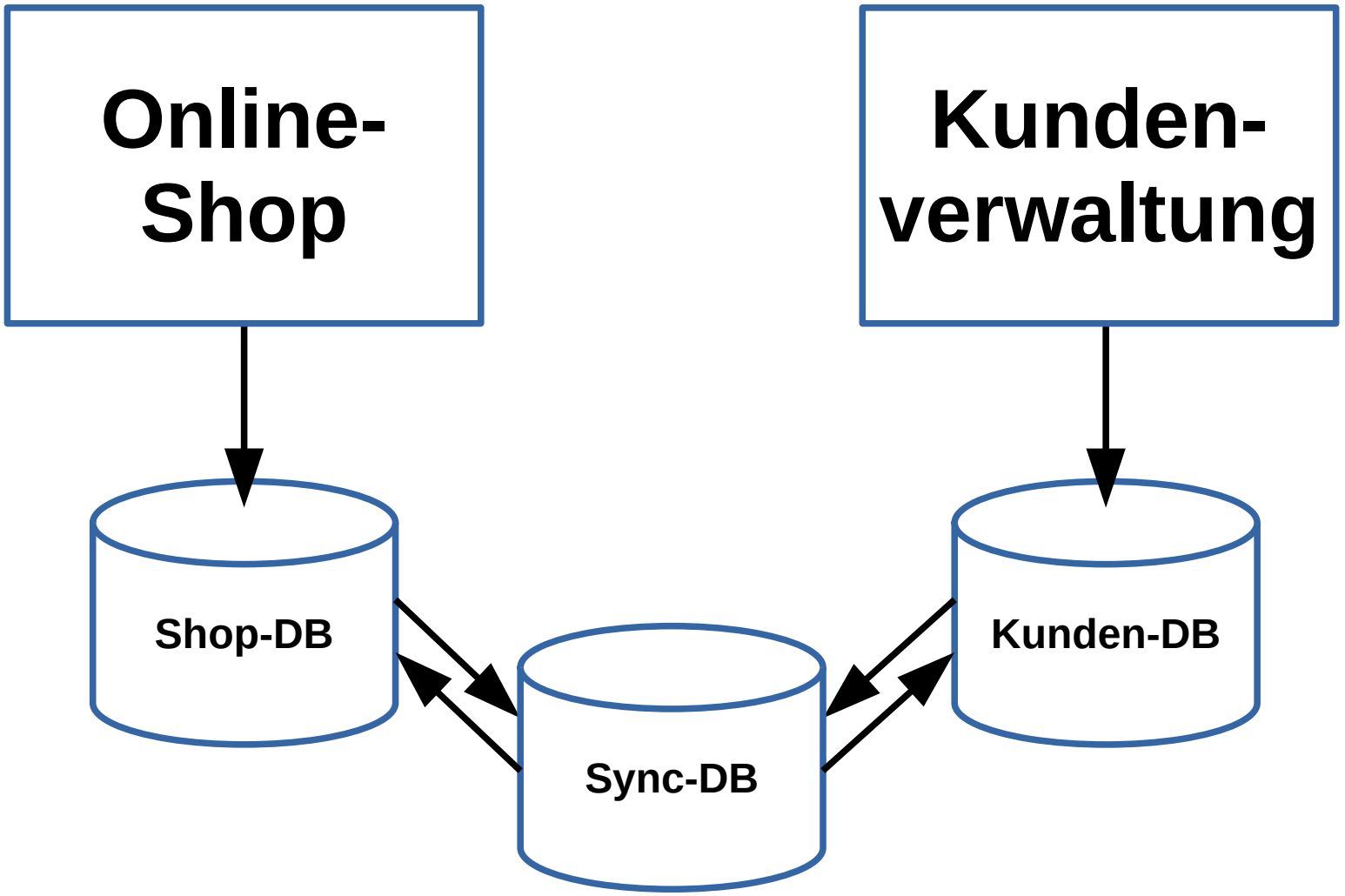


"Die Datenbank als Schnittstelle"

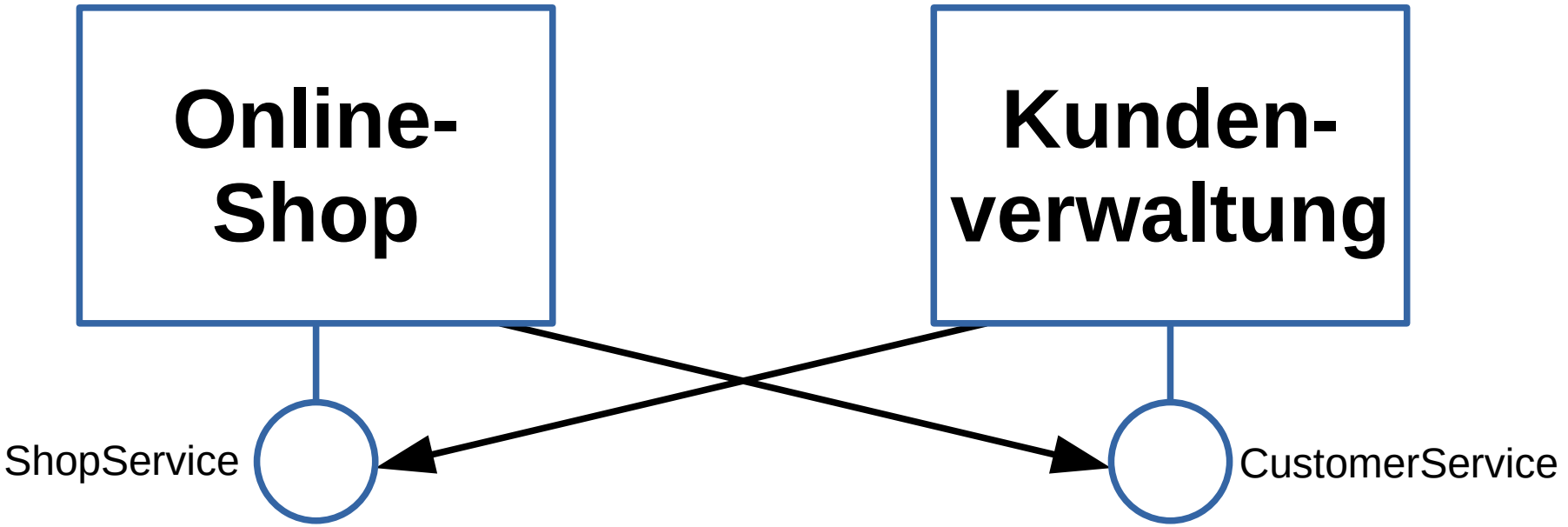
Enterprise Application Integration



Enterprise Application Integration

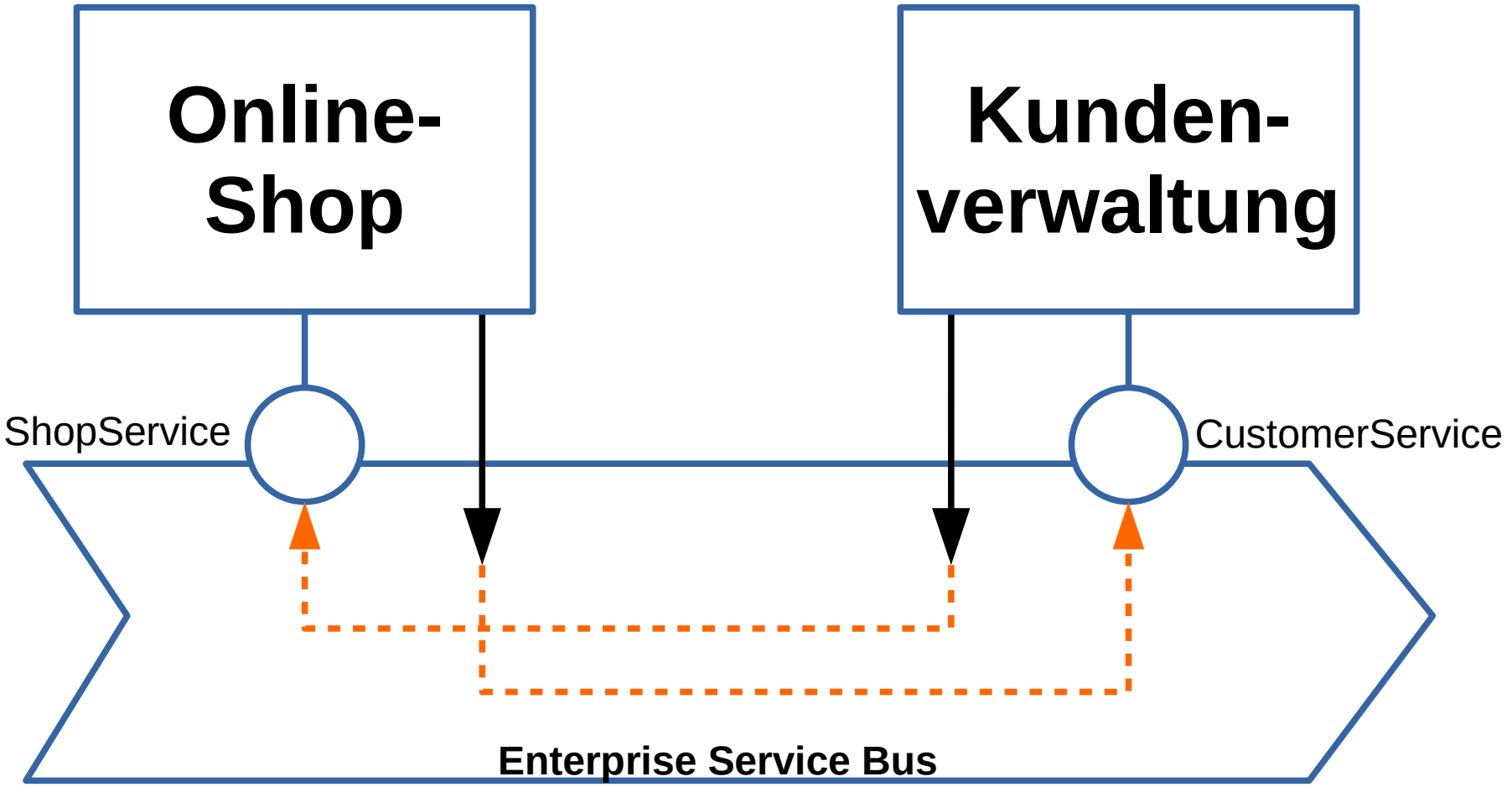


Neue Idee: Services

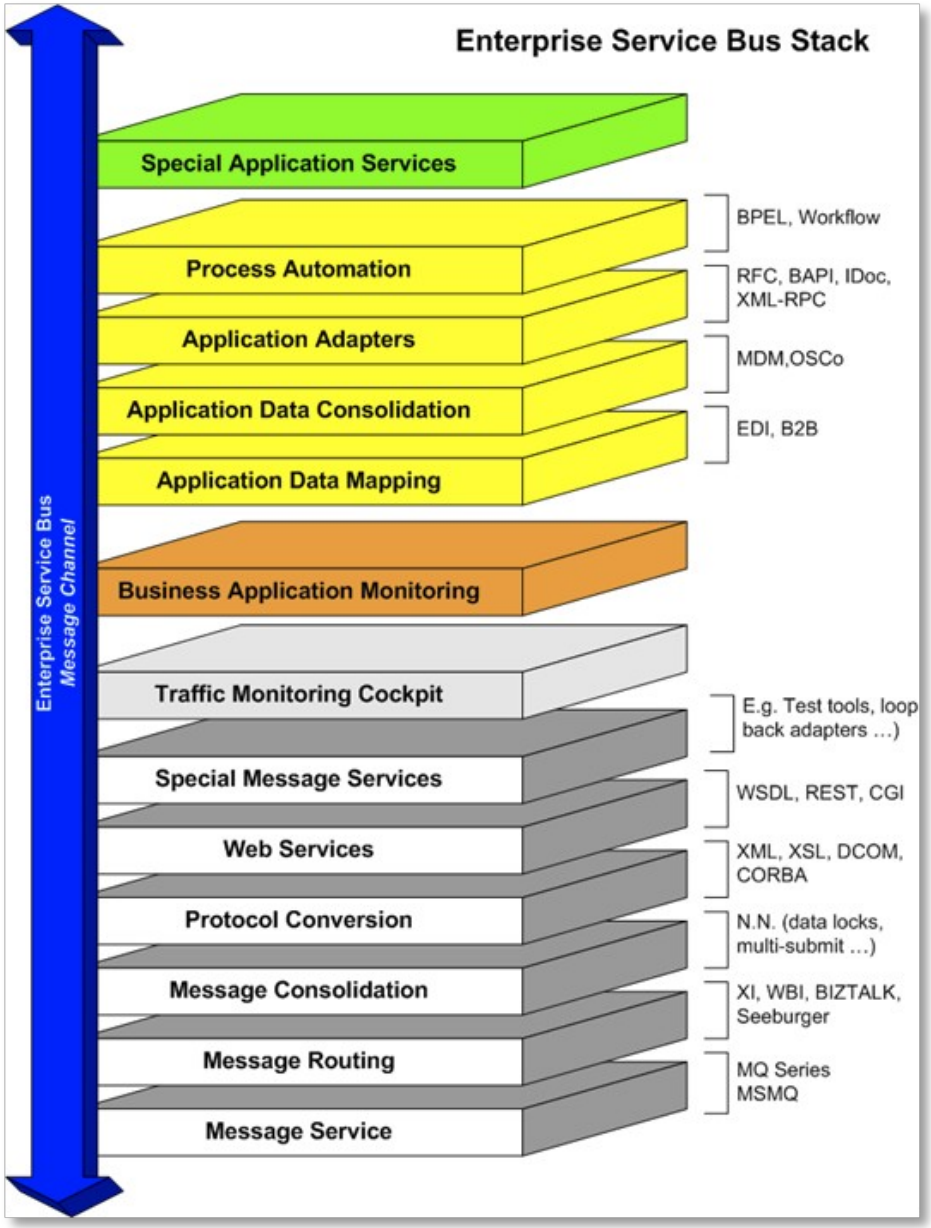


SOA: Service-Oriented Architecture

Enterprise Service Bus (ca. 2002)

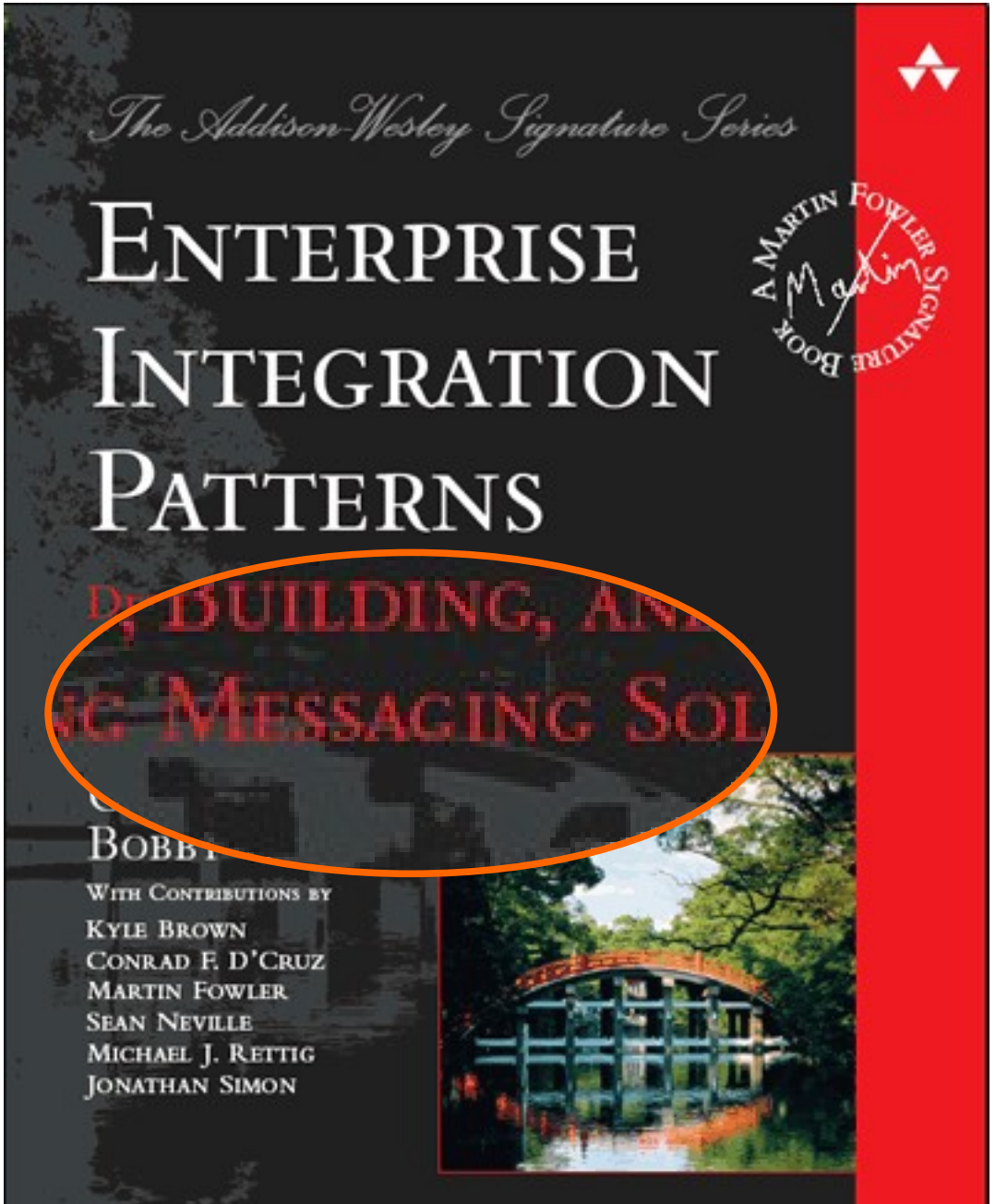


Doch dann…



Quelle: Wikipedia (Axelangeli - Own work, CC BY 3.0)

Messaging


The Addison-Wesley Signature Series
ENTERPRISE INTEGRATION PATTERNS
A MARTIN FOWLER SIGNATURE BOOK
Martin
BUILDING, AN
MESSAGING SOL
BOBB
WITH CONTRIBUTIONS BY
KYLE BROWN
CONRAD F. D'CRUZ
MARTIN FOWLER
SEAN NEVILLE
MICHAEL J. RETTIG
JONATHAN SIMON

Messaging

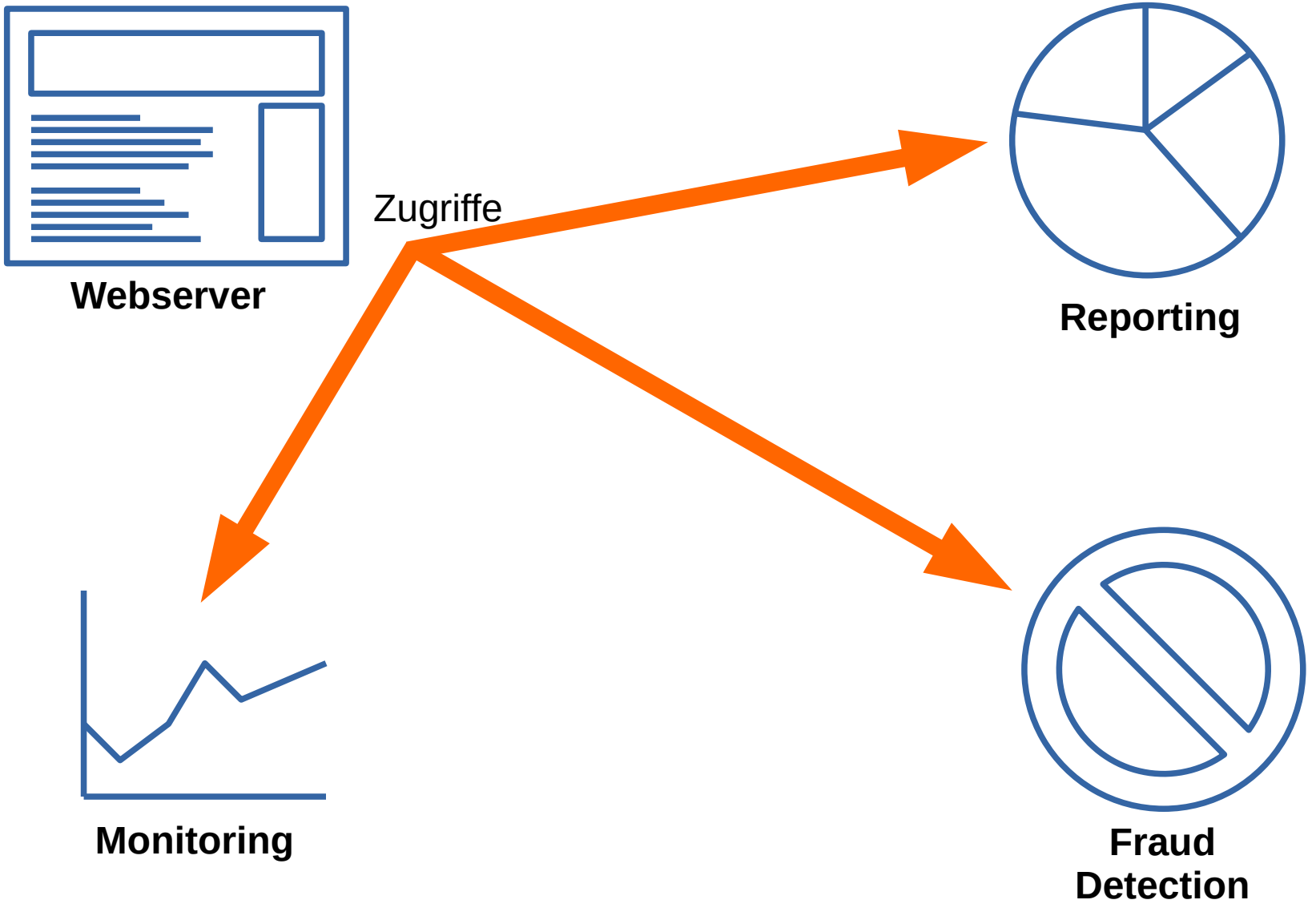
- Meist **asynchrone** Nachrichten
- Systeme werden **entkoppelt**
- Gesamtsystem wird **robuster**
- **Geringe Latenz** statt Datenbank-Abzüge

Messaging und Big Data

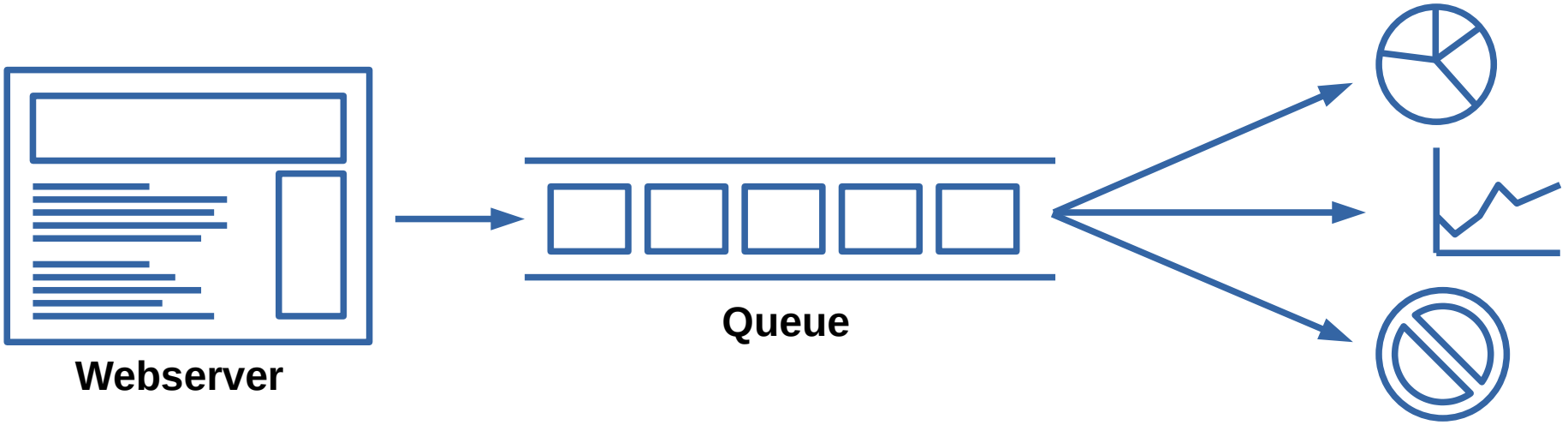
- Bekannte Anforderungen
  - geringe **Latenz**
  - hoher **Durchsatz**

- Beispiele
  - **Bewegungsdaten** (Webseiten, Mobilfunk, …)
    - z. B. Twitter "Firehose": **500 Mio. Tweets/Tag (~ 6.000/s im Mittel, 140.000/s Peak)**
  - **Sensordaten**
  - **Monitoring**
  - **Finanzdaten**
    - "A **1-millisecond** advantage in trading applications can be worth **$100 million a year** to a major brokerage firm, by one estimate." (informationweek.com)

Beispiel


Webserver
Zugriffe
Reporting
Monitoring
Fraud
Detection

Beispiel



Messaging für Big Data: Kafka

- Hochskalierbare **Message ~~Queue~~ Broker**
- Inspiriert von **Transaktions-Logs**
- **Publish/Subscribe**: Themen abonnieren
- Ausgelegt für große, **verteilte Systeme**

- **LinkedIn** (2012; heute: Confluent)

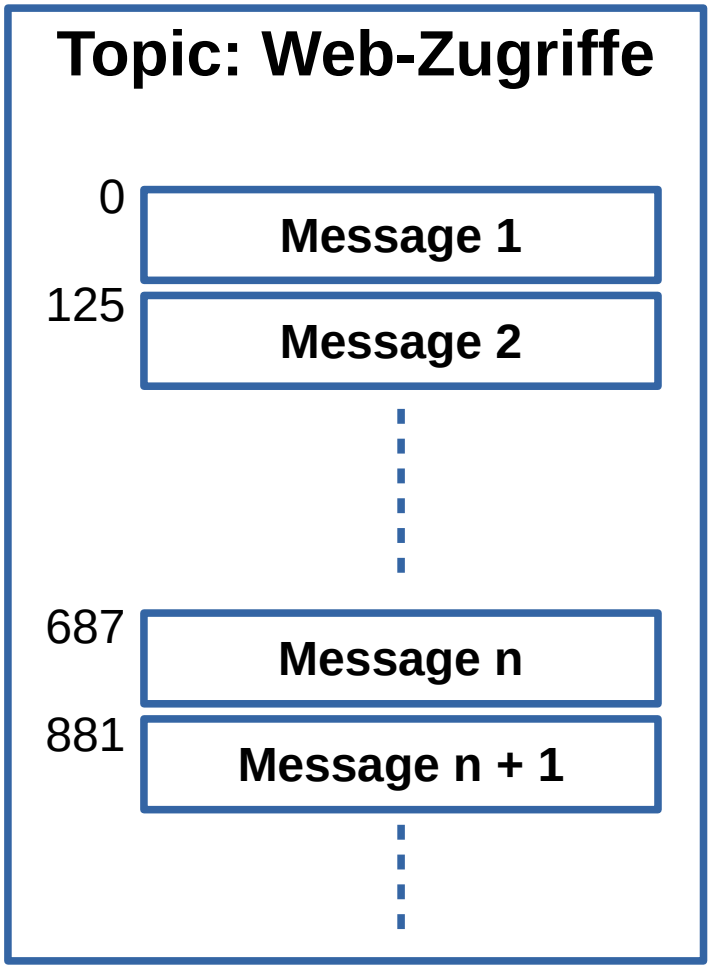
APACHE kafka™
A distributed streaming platform

Bild: kafka.apache.org

Grundideen

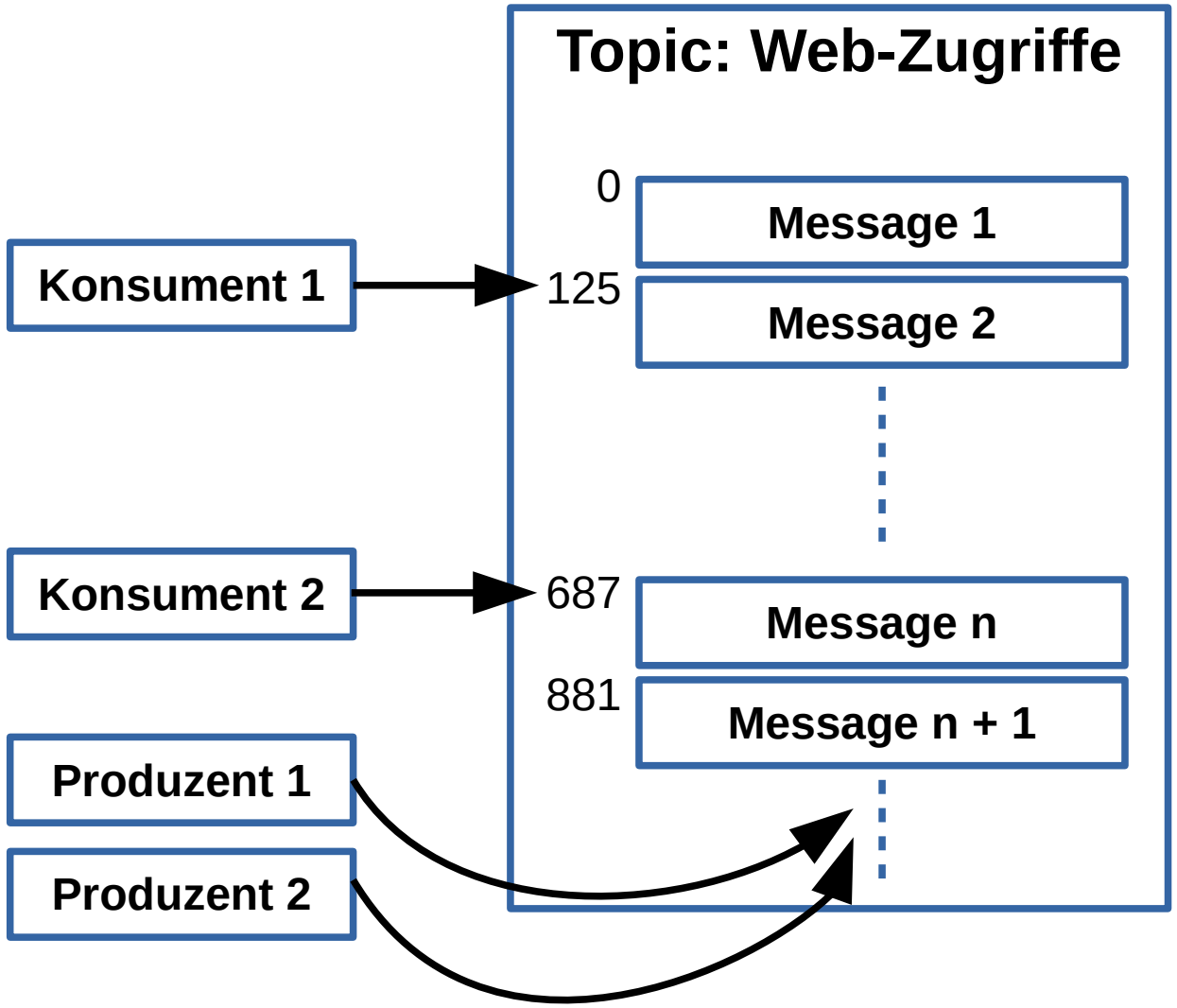
- Daten organisiert in **Topics**
- Topic ≙ **Log**
- **Serielles Schreiben**: nur Anfügen
- **Serielles Lesen** (pro Konsument)

  → Effizient auch auf **HDD**



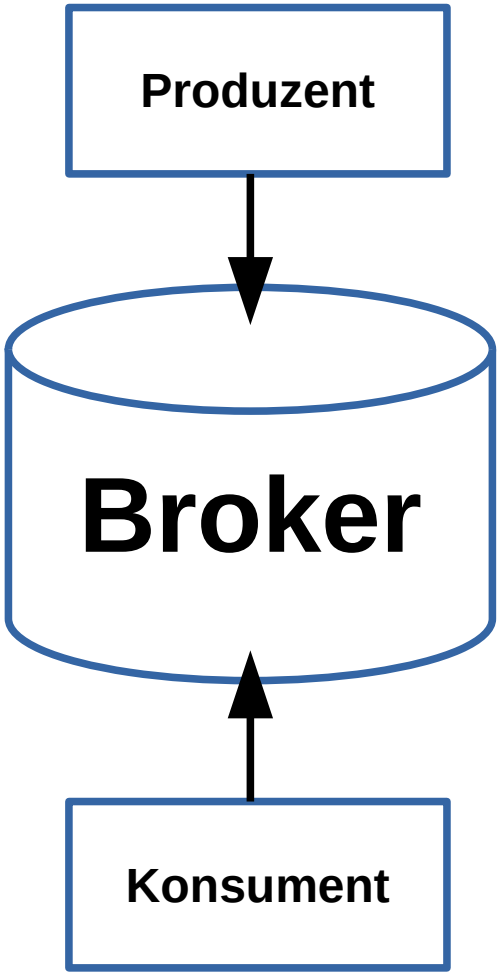
Grundideen

- **Produzenten** fügen nur am Ende an
- **Konsumenten** können unterschiedliche Stände haben
- **Pull**: Konsumenten fordern Daten an

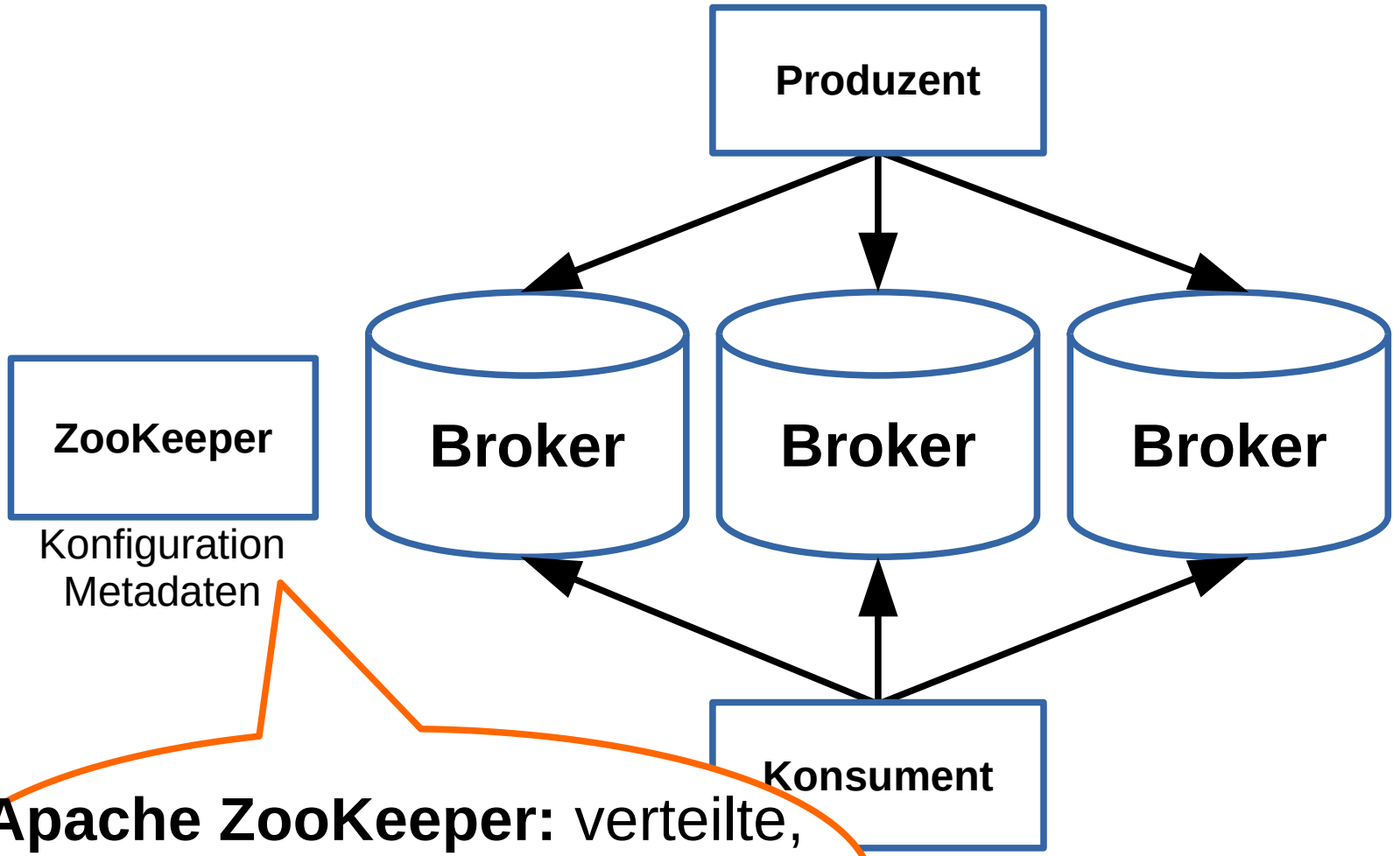


→ keine Organisation der Lesezugriffe durch den Broker

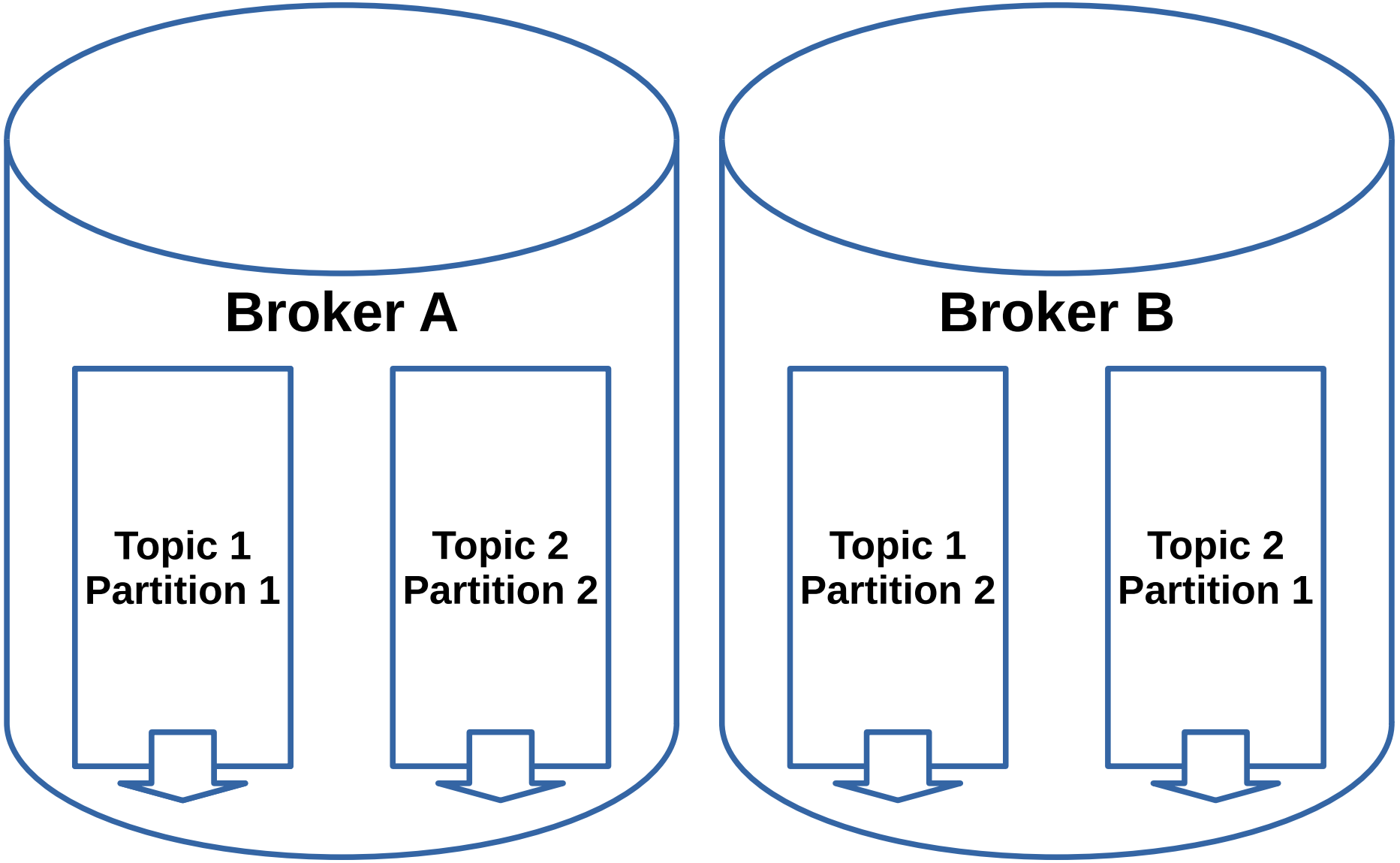
Kafka: Architektur



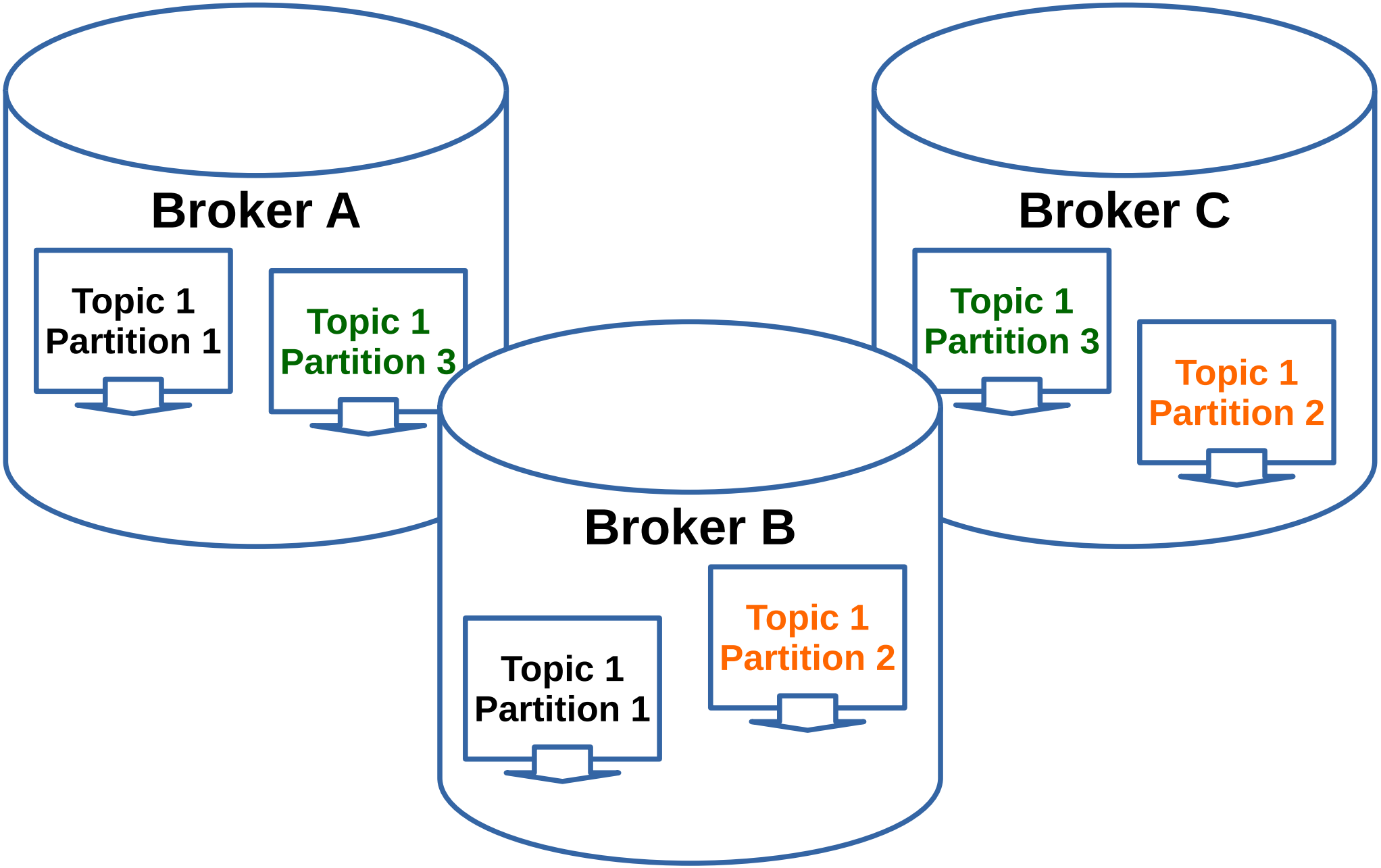
Kafka: Architektur



Verteilung von Topics

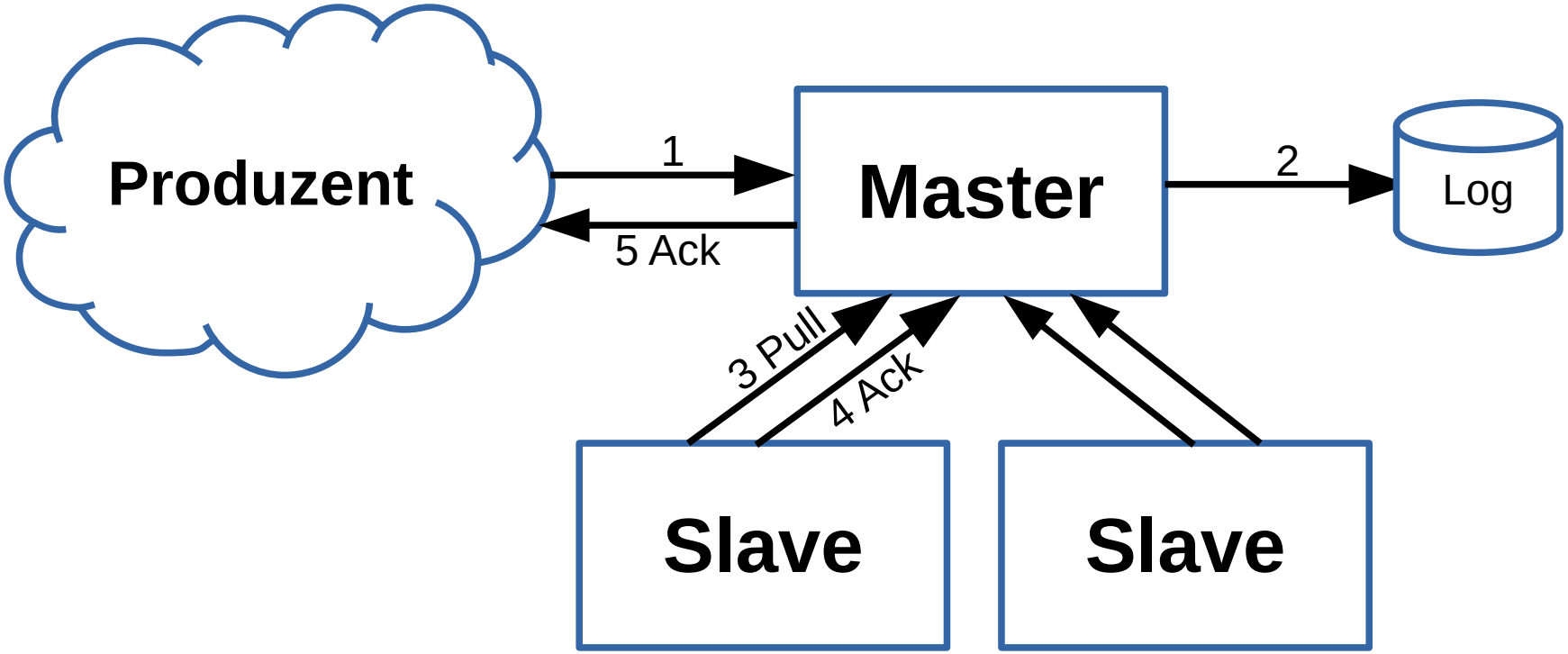


Replikation von Partitionen



Replikation

- **Master-Slave-Replikation** pro Partition
- **Verschiedene Master** pro Partition → Metadaten



Was bedeutet das?

- **Partitionen**
  - **Lastverteilung** über die Broker
  - Erhöhung des **Durchsatzes**
  - Einheit für **Parallelisierung**
- **Replikas**
  - **Robustheit** gegen Ausfälle
  - **Lastverteilung**

Kafka: Konsumenten

- Funktionsweise: Lesen aus Topic
  - zu lesende **Partitionen bestimmen**
  - pro Partition
    - letzten **Offset bestimmen**
    - **Broker** für Partition auswählen
    - **Daten** ab Offset anfordern
    - **Offset quittieren**

- Speichern der **Offsets**
  - alt: in **ZooKeeper**
  - neu: in **Kafka** selbst

Beispiel

- Kafka Console Producer

```
echo "Hallo Welt" | \
/usr/hdp/current/kafka-broker/bin/kafka-console-producer.sh \
   --broker-list infbdt04.fh-trier.de:6667 \
   --topic bigdata200-1 \
   --security-protocol SASL_PLAINTEXT
```

Beispiel

- Kafka Console Consumer

```
/usr/hdp/current/kafka-broker/bin/kafka-console-consumer.sh \
   --bootstrap-server infbdt03.fh-trier.de:6667 \
   --zookeeper infbdt03.fh-trier.de:2181 \
   --topic bigdata200-1 \
   --from-beginning \
   --new-consumer \
   --security-protocol SASL_PLAINTEXT

Hallo Welt
```

Speicherung der Offsets

- **Alt:** in Zookeeper
- **Neu:** in Kafka-Topic **`__consumer_offsets`**

```
[console-consumer-20259,bigdata200-1,2]::
[OffsetMetadata[25298,NO_METADATA],CommitTime 1496840338641,ExpirationTime
2385873042641]
[console-consumer-20259,bigdata200-1,1]::
[OffsetMetadata[23303,NO_METADATA],CommitTime 1496840338641,ExpirationTime
2385873042641]
[console-consumer-20259,bigdata200-1,0]::
[OffsetMetadata[26205,NO_METADATA],CommitTime 1496840338641,ExpirationTime
2385873042641]
```

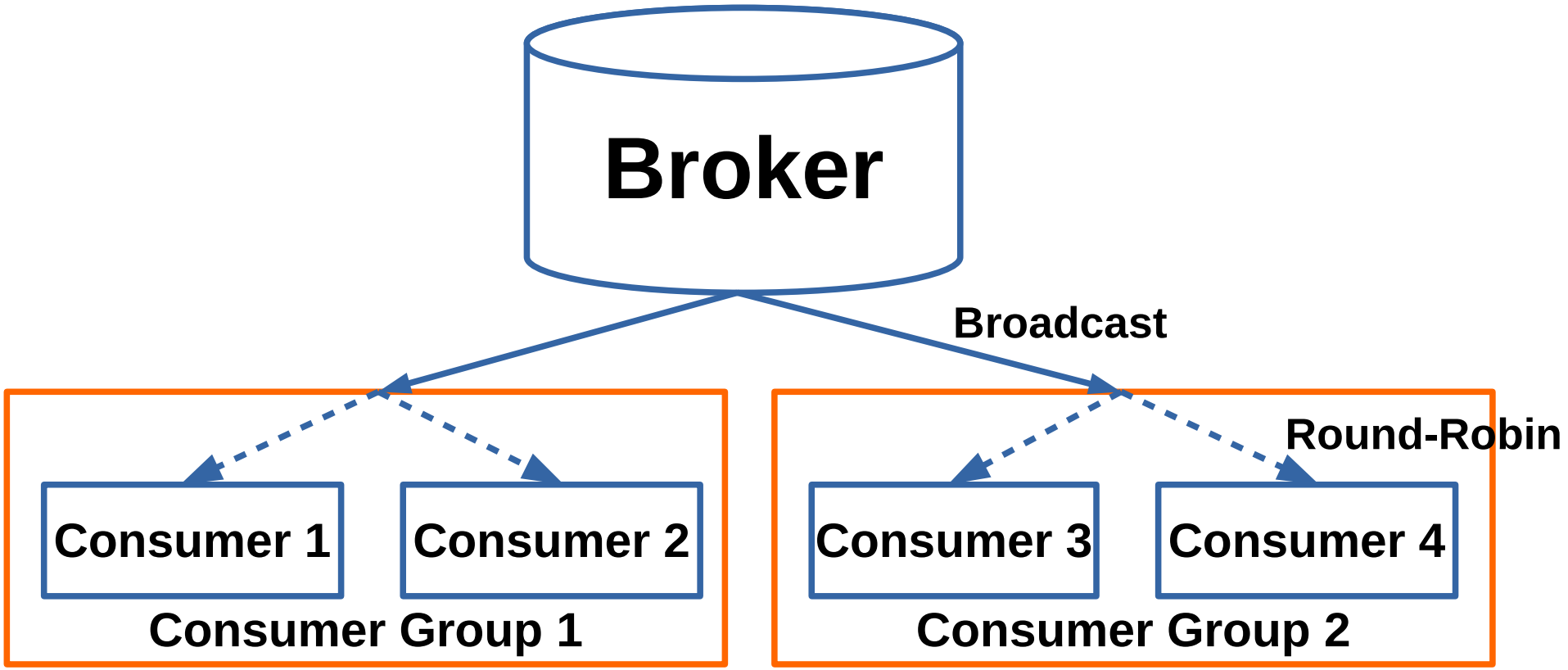
Auslieferungssemantik (I)

- Jede Nachricht an **jeden Konsumenten**
  - Beispiel: Status-Updates, Chat
  - **"Publish/Subscribe"**
- Jede Nachricht an **genau einen Konsumenten**
  - Beispiel: Arbeitsaufträge, Buchungen
  - **"Round Robin"**
- **Mischformen**

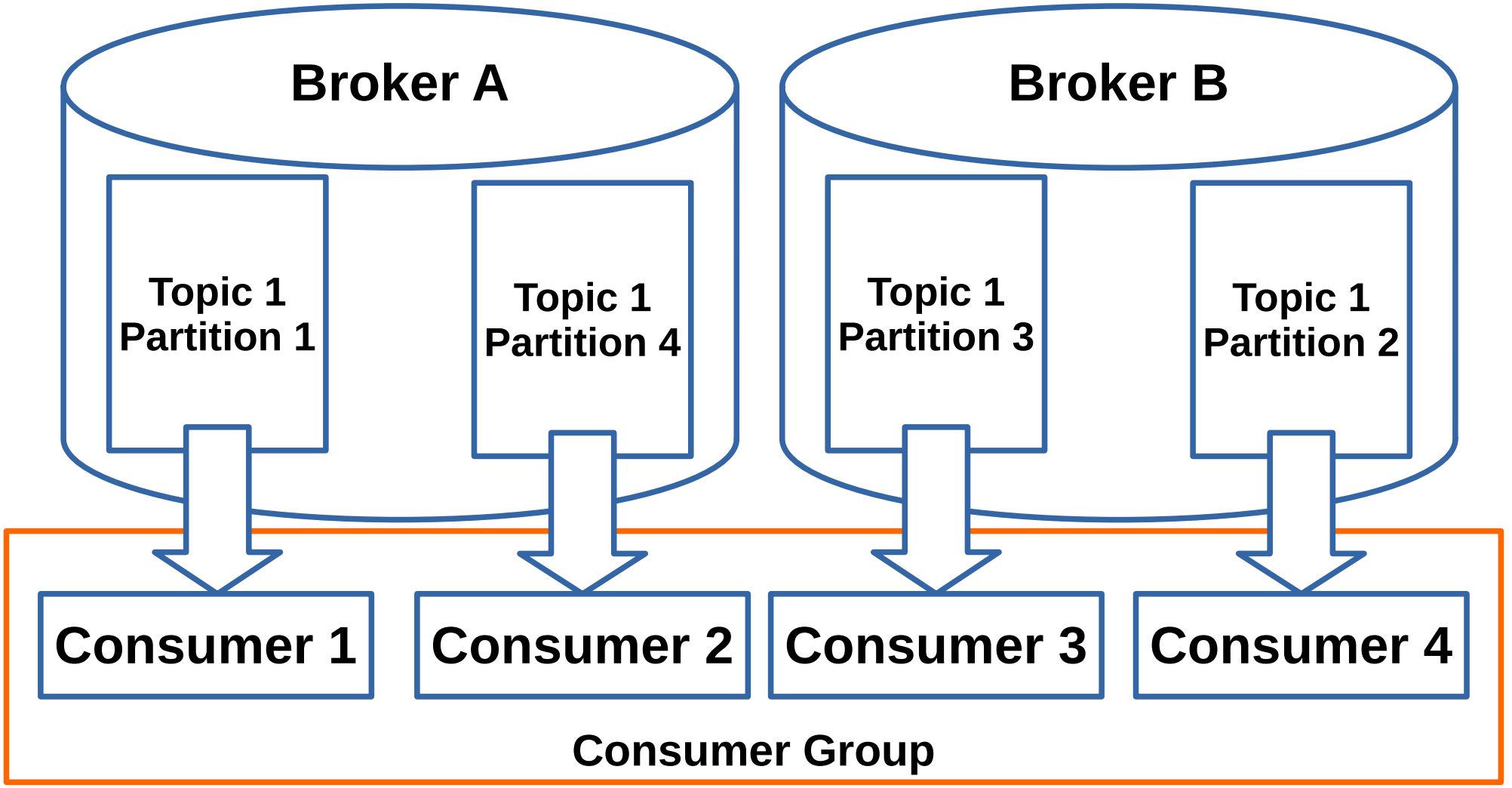
Consumer Groups

- Consumer Group = Menge von Konsumenten
  - … mit gemeinsamen Metadaten
- Auslieferung an **alle** Consumer Groups
- Auslieferung an **einen Konsumenten pro** Consumer Group

Consumer Groups



Parallelisierung



Auslieferungssemantik (II)

- Unterschiedliche Garantien
  - **At Least Once:**
    Jede Nachricht kommt **mindestens** ein Mal an
  - **At Most Once:**
    Jede Nachricht kommt **höchstens** ein Mal an
  - **Exactly Once:**
    Jese Nachricht kommt **genau** ein Mal an

Auslieferungssemantik (II)

- Garantien in Kafka:
  - **Kommt auf das Verhalten des Konsumenten an!**
- **At most once** → Autocommit einschalten
- **At least once** → Nachrichten erst nach Verarbeitung quittieren
- **Exactly once** → Verarbeitung jeder einzelnen Nachricht lokal atomar speichern

Beispiel: Java-Consumer

```
Properties props = new Properties();
props.put("bootstrap.servers", "infbdt03.fh-trier.de:6667");
props.put("group.id", "consumer-tutorial");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("mytopic"));
try {
    while (running) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record : records)
            System.out.println(record.offset() + ": " + record.value());
    }
} finally {
    consumer.close();
}
```

Beispiel: Java-Producer

```
Properties props = new Properties();
props.put("bootstrap.servers");
props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

producer.send(new ProducerRecord<>("mytopic", "key", "value"));

producer.close();
```

Beispiel: Java-Consumer

```
Properties props = new Properties();
props.put("bootstrap.servers", "infbdt03.fh-trier.de:6667");
props.put("group.id", "consumer-tutorial");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("mytopic"));
try {
    while (running) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record : records)
            System.out.println(record.offset() + ": " + record.value());
    }
} finally {
    consumer.close();
}
```

Zusammenfassung

- **Messaging**
  - Applikationsintegration
  - lose Kopplung, wenig Abhängigkeiten
  - Asynchronität
- **Kafka**
  - "Message Broker" für Big-Data-Anwendungen
  - realisiert als verteiltes Transaktions-Log
  - nur serielles Schreiben → hocheffizient
  - Publish/Subscribe und Broadcast möglich
  - Zwischenformen über Consumer Groups