Apache Kafka

integrata cegos

INSPIRE · QUALIFY · CHANGE

Inhalts-verzeichnis

Einführung

- Ist-Situation
- Features von Apache Kafka

Ist-Situation

System-Landschaft

- Host-Rechner, auf denen die verschiedensten Produkte/Server laufen
  - Datenbank-Systeme
    - Relationale Datenbanken
    - auch NoSQL-Umfeld
  - Server
    - insbesondere Web Server
- Netzwerk
  - TCP/IP
  - Ausrichtung auf http/https, Stichwort RESTful Web Services
- Lokale Ressourcen auf jedem Host
  - Dateisystem

Allgemeine Problemstellung

- Jedes dieser Systeme produziert eine ganze Menge von Daten
  - Verteilung an die jeweiligen Ziel-Systeme über wohl-definierte Kommunikationskanäle
  - Zugriff auf die lokalen Ressourcen der Host-Rechner ist häufig sehr aufwändig
- Zentrale gemeinsame Steuerung/Überwachung

Etablierte Lösungen existieren

- Kommunikation über
  - Messaging Systeme
  - Event-getriebene Systeme, Event-Bus
- Gemeinsame zentrale Datenablage für Log-Informationen
  - z.B. Splunk

Features von Apache Kafka

Datensenke

- Hauptfokus ist das Weg-Schreiben von Informationen
  - Potenziell unendlich groß
    - Skalierbarkeit muss gewährleistet sein
    - Dynamisch, keine Wartungsfenster etc. notwendig
  - Potenziell beliebig “breitbandig”
    - Limit des Datendurchsatzes ist das Netzwerk
- Hier verhält sich Kafka wie ein ganz normales Datenbank-System

Datenquelle

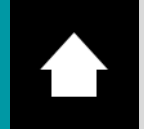
- Verteilung der Daten erfolgt aktiv an registrierte Konsumenten
  - Verwaltungslogik durch Algorithmen
    - Kafka benutzt und benötigt CPU-Ressourcen
- Kafka verhält sich hier wie ein Messaging-System

Datenverarbeitung

- Streaming in Kafka
  - Filtern
  - Transformation
  - Aggregieren
- Kafka verhält sich hier wie ein Event-Bus

Der Broker

- Realisierung eines skalierenden Systems
- Apache Kafka

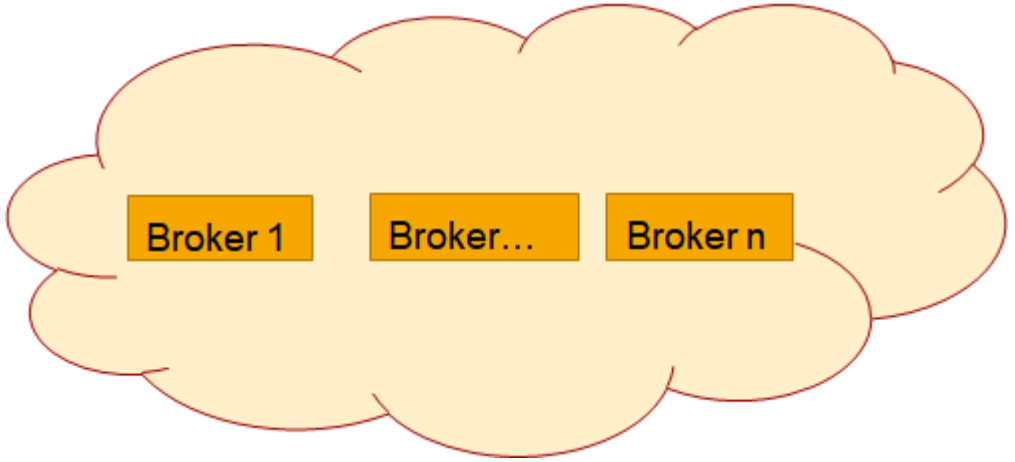


Realisierung eines skalierenden Systems

Horizontal skalierender Cluster

- Ein Kafka-Cluster besteht prinzipiell aus mehreren Brokern
  - Diese werden „irgendwie“ zu einer logischen Einheit gruppiert

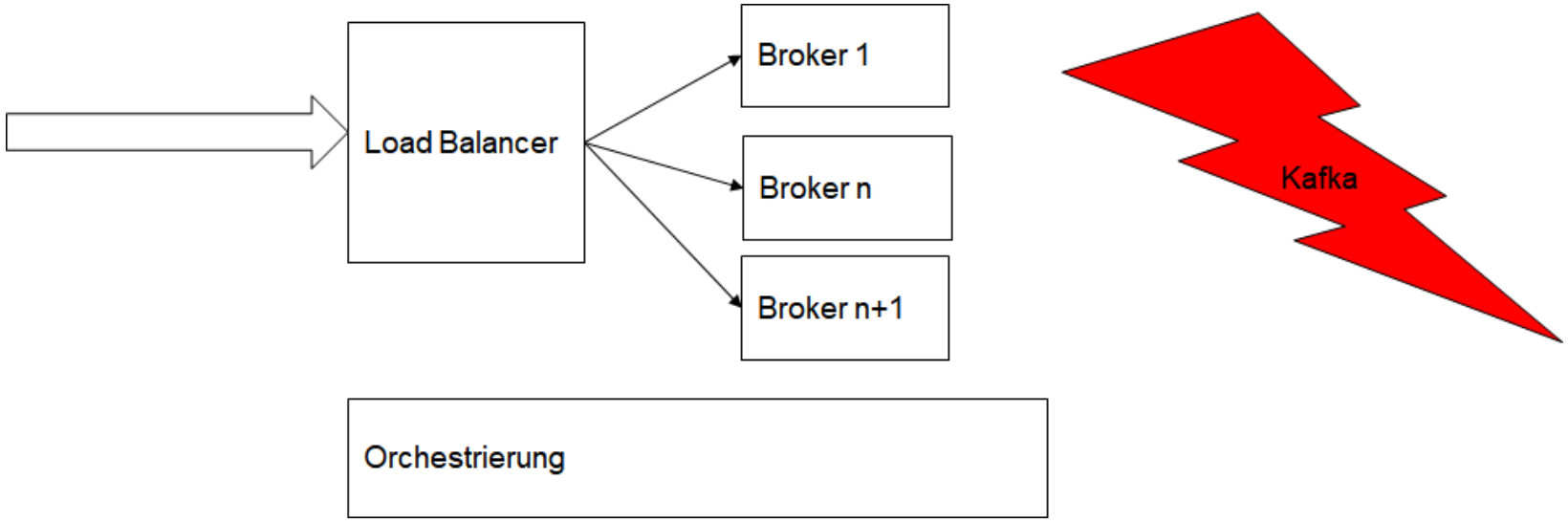
Horizontal skalierender Cluster



Mögliche Realisierungen

- Orchestrierung von Stateless Brokern
- Cluster mit Master
- Masterless Ring Cluster

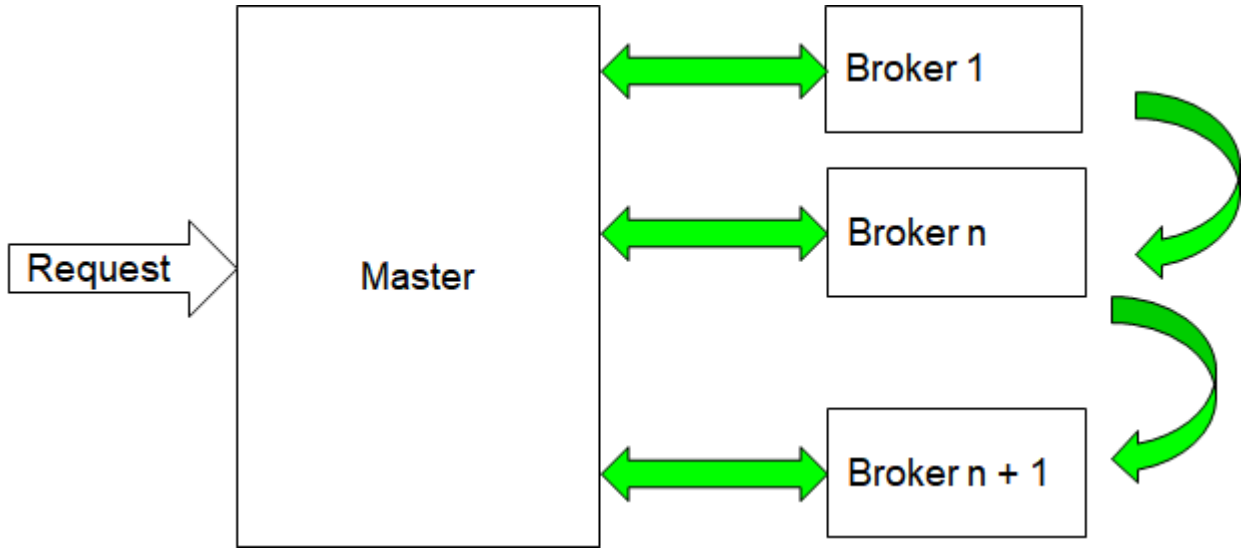
Orchestrierung von Stateless Brokern



Cluster mit Master

- Aktive Verwaltung des Clusters über den Master
- Dynamische Änderung der Broker-Anzahl erfordert ein Rebalancing über den Master

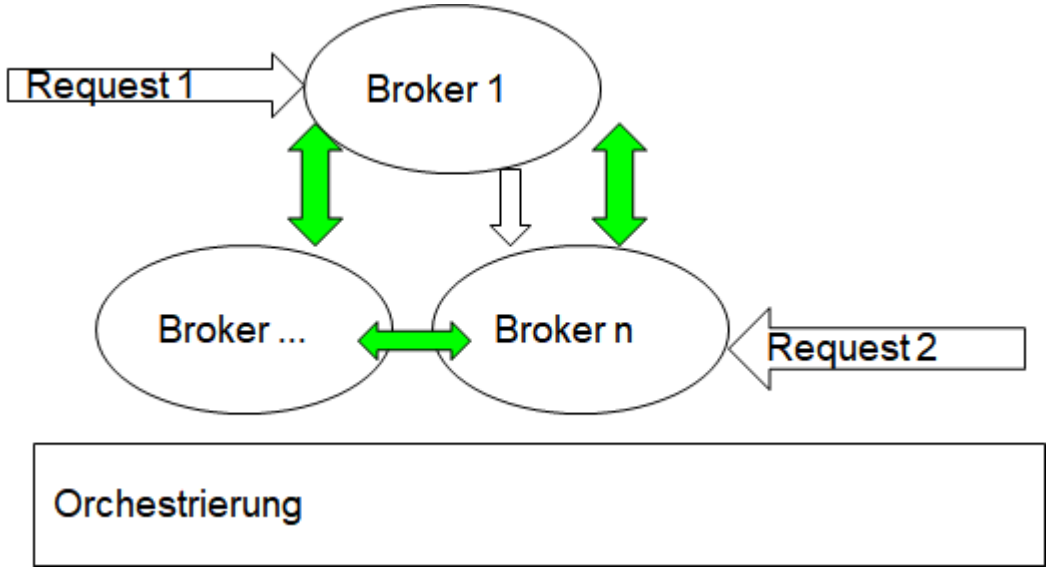
Cluster mit Master



“Masterless” Ring Cluster

- Broker verwalten sich intern selber
  - Insbesondere organisieren sie das Rebalancing selber
  - Eine Orchestrierung fügt dynamisch Broker hinzu oder stoppt überflüssige

“Masterless” Ring Cluster

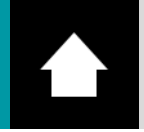


Beispiele

- Load Balancing und Orchestrierung
  - Jede Microservices-Architektur
- Master-basierter Cluster
  - Datenbank-Systeme
    - Postgres
    - Mongo DB
- Ring Cluster
  - Datenbank-Systeme
    - Apache Cassandra
    - Couchbase
    - Apache Ignite

Bewertung der Ansätze

- Aus betrieblicher Sicht einfach zu realisieren:
  - Load Balancer in Kombination mit Orchestrierung
  - Masterless Ring Cluster
- Ein Master erleichtert die Implementierung aus Sich des Broker-Herstellers
  - Daten-Konsistenz



Apache Kafka

Apache Kafka

- Master
  - Apache Zookeeper
    - Sammelt Informationen über den Status der einzelnen Broker-Instanzen
      - Neu gestartete oder stoppende Broker kommunizieren mit dem Zookeeper
    - Jeder Broker kann im Zookeeper Informationen ablegen
      - Zookeeper ist etwas vereinfacht ein hierarchisches Dateisystem
    - Als Single-Point of Failure wird in Zookeeper im Real-Betrieb als Ensemble betrieben
- Auch die Arbeitsweise des Ring-Clusters ist sichtbar
  - Ein Client-Programm kann sich für bestimmte Aktionen an einen Broker-Knoten wenden
  - Rebalancing der Knoten wird vom Zookeeper angestoßen, aber von den Knoten realisiert

Warum arbeitet Kafka so?

- Genauer: “Warum arbeitet Kafka **aktuell** so?”
- Work in Progress
  - Ursprünglich war der Zookeeper ein echter Master
    - jegliche Client-Kommunikation lief darüber
  - Refactoring von Kafka hin zu einem reinen Ring Cluster
    - “Zookeeper will be removed in future versions...”

Aufsetzen eines Kafka-Systems

- System-Voraussetzungen
- Download
- Download

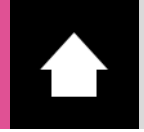
System-Voraussetzungen

Anforderungen an das Host-System

- Java Runtime
  - damit Apache Kafka Plattform-unabhängig
  - Java 8 oder größer
- Mindestens Java-Prozesse notwendig:
  - Zookeeper
  - Mindestens ein Kafka Broker
- Hardware-Anforderungen sind moderat
  - Zookeeper: 256MB RAM
  - Kafka-Broker 1GB RAM (512MB geht auch)

Kafka im Container

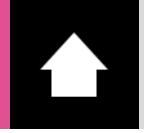
- Docker-Images mit Kafka sind möglich
- Apache Kafka Community stellt kein fertiges Docker-Image zur Verfügung
  - Wurstmeister
  - Bitnami,
  - Spotify
  - Confluent
    - Anbieter für kommerzielle Kafka-Lösungen



Download

Kafka-Distribution

- Plattform-unabhängiger Download von den Apache Seiten
- `bin`
  - Shell-Skripte und Windows Bat
- `config`
  - Plattform-Unabhängig
- Vorsicht
  - Neben den essenziellen Dateien auch viele Beispiele etc
  - Stripped Down
    - Start-Skripte für Zookeeper und Kafka Broker
    - Konfiguration `zookeeper.properties` und `server.properties`



Download

Start Schritt für Schritt

- Zookeeper
  - `bin/zookeeper-server-start.sh|bat /config/zookeeper.properties`
    - Ab jetzt: Java-Prozess lauschend auf Port 2181
- Kafka Broker
  - `bin/kafka-server-start.sh|bat /config/server.properties`
    - Ab jetzt: Java-Prozess lauschend auf Port 9092
- Potenzielle Probleme
  - “Java not found”
    - Java installiert? CHECK java -version
  - “Kann Pfad nicht finden”
    - CHECK: Pfad zur Java-Runtime mit Leerzeichen unter Windows?
  - “Pfad zu lang” oder so ähnlich
    - Workaround für Windows:

Test

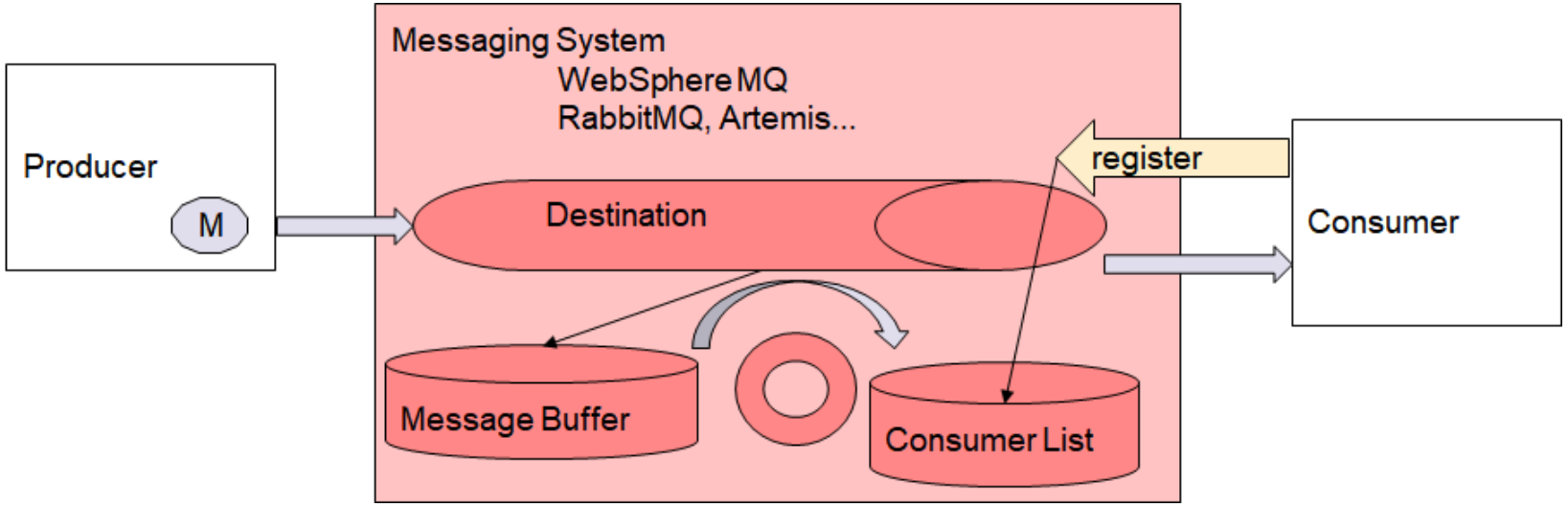
- Linux
  - Producer
    - `kafka-console-producer.sh --broker-list localhost:9092 --topic demo`
  - Consumer
    - `kafka-console-consumer.sh--bootstrap-server localhost:9092 --topic demo`
- Windows
  - Korrespondierende Skripte unter *`bin\windows`*
  - Alternativ: Git Bash oder Ähnliches

Arbeitsweise von Kafka

- Analogie zu Messaging-systemen
- Kafka als Messaging System
- Skalierung
- Ausfallsicherheit
- Topics

Analogie zu Messaging-systemen

Klassische Messaging

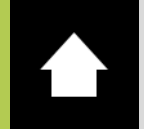


Technische Umsetzung

- Message Buffer
  - befüllt beim Eintreffen einer Message
  - Nach einer erfolgreichen Auslieferung ist die Nachricht wieder zu löschen
- Ablage der Message als Datei im Dateisystem oder in einer Datenbank
- Aus Sicht des Producers heraus wird das Messaging System den Erhalt der Nachricht quittieren, sobald diese im Message Buffer gespeichert wurde

Consumer beim klassischen Messaging

- Strategien zur Auslieferung
  - “Publish/Subscribe”
    - Die Message wird an alle registrierten und aktiven Consumer ausgeliefert
    - und anschließend aus dem Message Buffer gelöscht
  - “Point-to-Point”
    - Die Message wird so lange vorgehalten, bis sie exakt ein aktiver Consumer bekommt
    - und anschließend aus dem Message Buffer gelöscht
  - “Publish/Durable Subscriptions”
    - Die Message wird an alle registrierten Consumer ausgeliefert, deshalb so lange vorgehalten, bis der letzte registrierte Consumer wieder aktiv wurde
    - und anschließend aus dem Message Buffer gelöscht

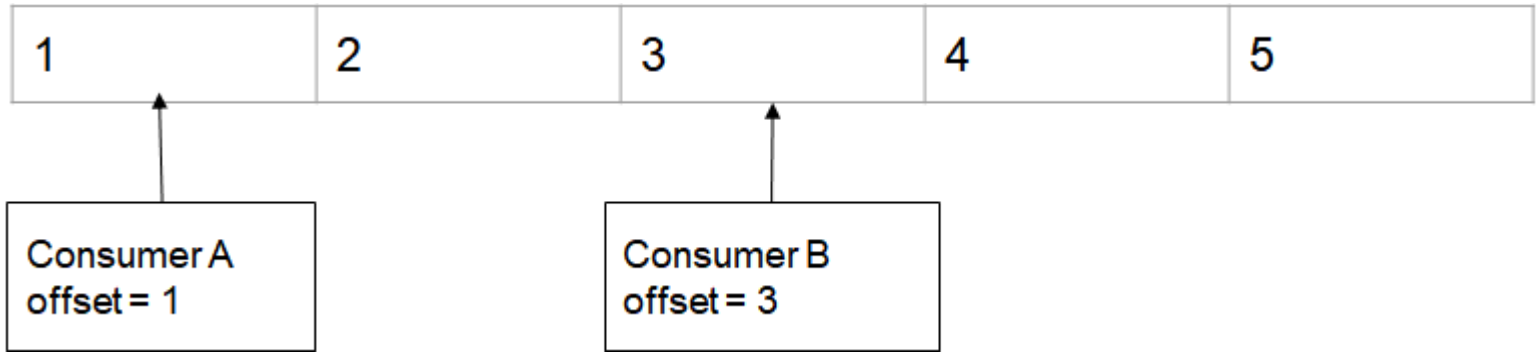


Kafka als Messaging System

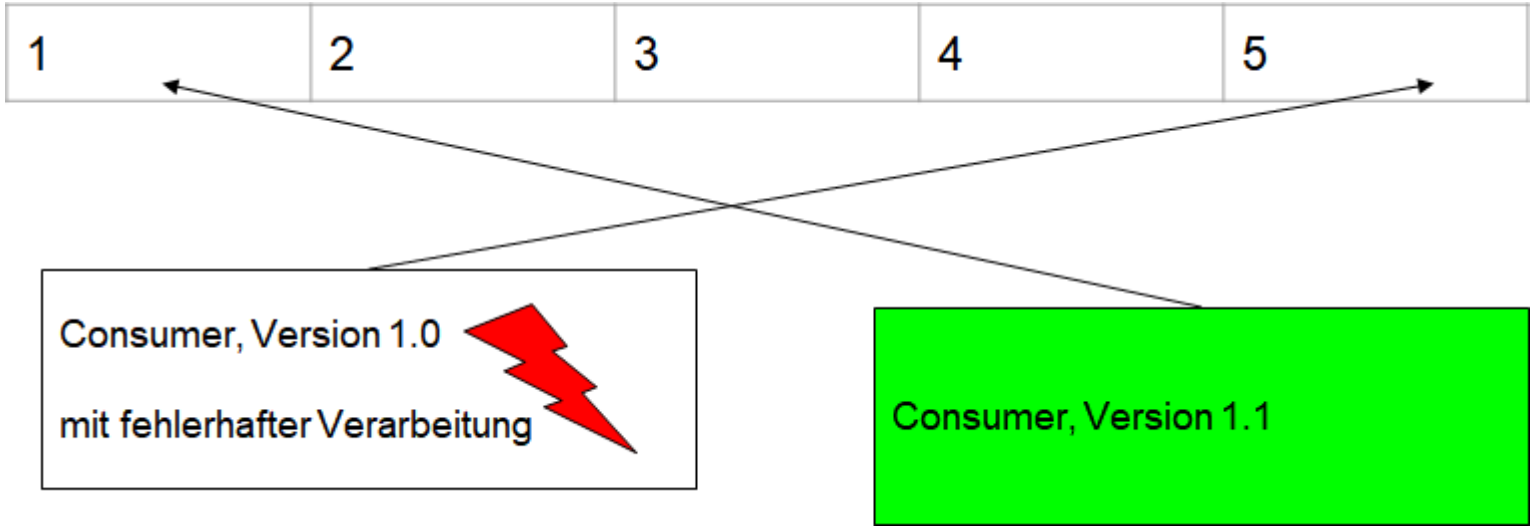
Kafka

- “Kafka löscht gar nicht”
  - Message Buffer ist ein “Append-only Log”
  - Automatisches Löschen durch Timeout (z.B. 30 Tage) oder Größe (z.B. 500 GByte)
- Damit verhält sich Kafka wie eine Datenbank (!)
- Ein Consumer kann sich jederzeit auch bereits gelesene/ausgelieferte Nachrichten anschauen
  - Für jeden Consumer-Prozess muss Kafka einen “State” halten
  - “Offset” pro Consumer-Session

Unabhängigkeit der Consumer



Beispiel: BugFix eines Consumers

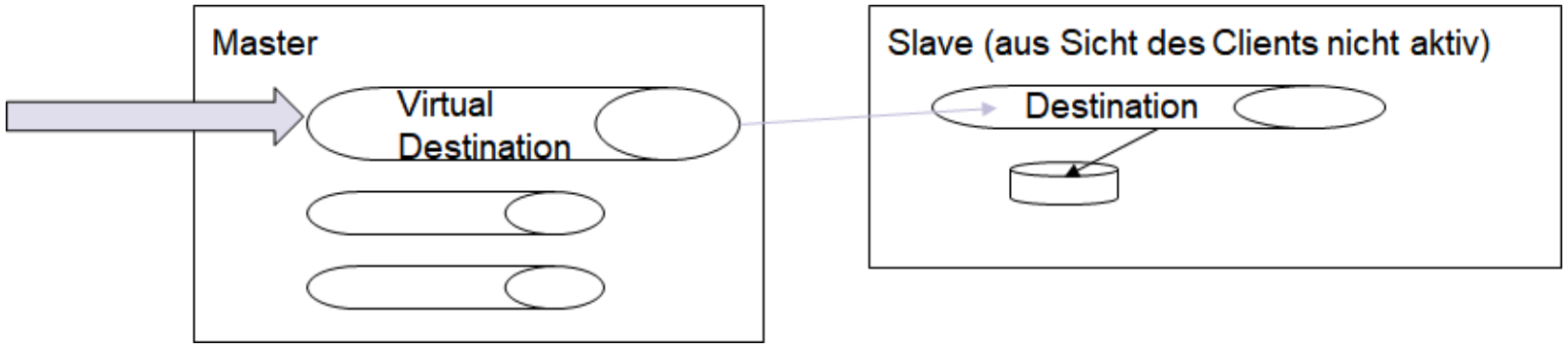


Skalierung

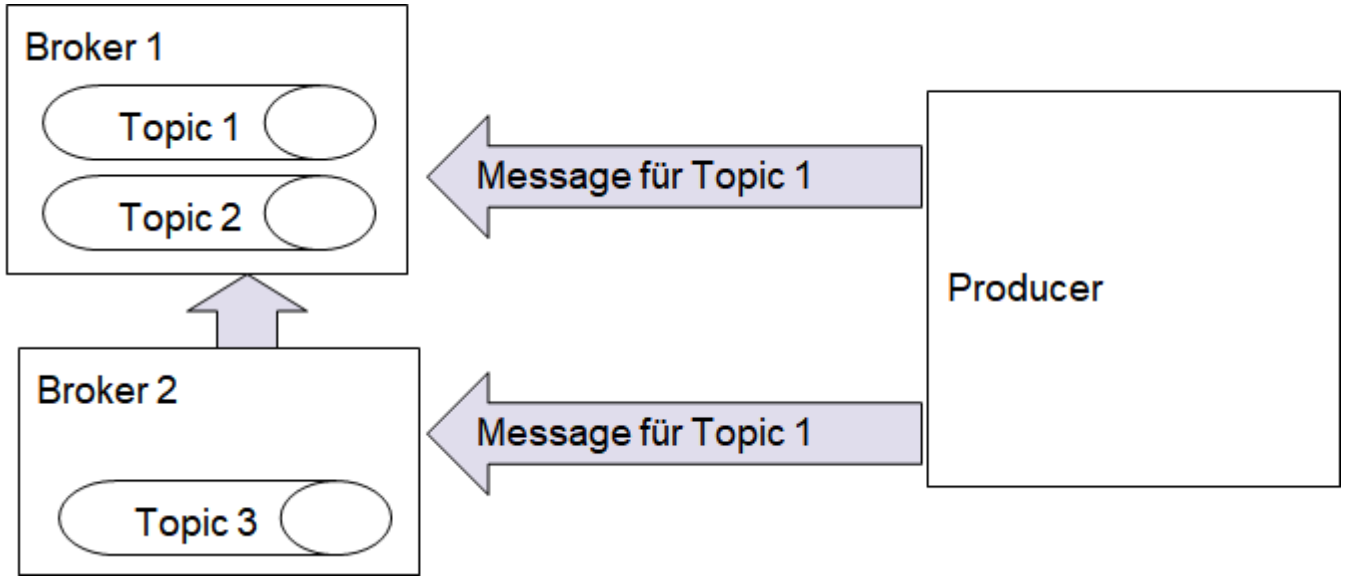
Cluster-Lösung bei Messaging-Systemen

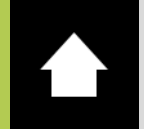
- Auslieferungs-Garantien sind im Cluster sehr schwierig umzusetzen
  - ein wirklich großes Problem
- In der Praxis ist deshalb eigentlich nur ein Broker der aktive Master

Cluster-Lösung bei Messaging-Systemen



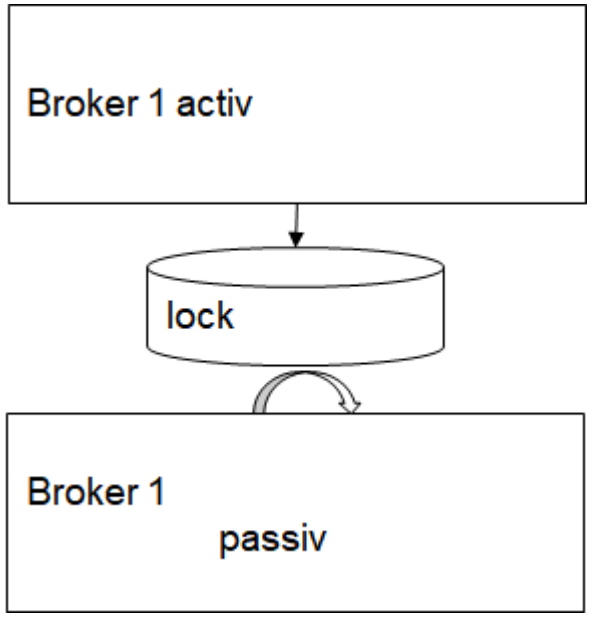
Kafka


Broker 1
Topic 1
Topic 2
Broker 2
Topic 3
Message für Topic 1
Message für Topic 1
Producer

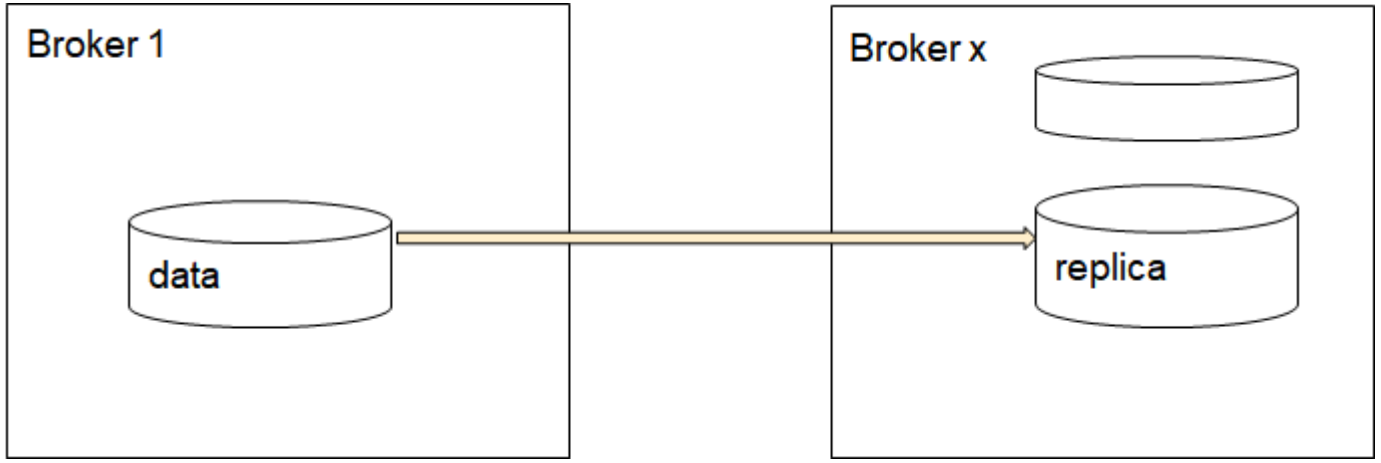


Ausfallsicherheit

Ausfallsicherheit: Aktiv - Passiv



Ausfallsicherheit: Replikation



Kafka: Replikation

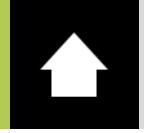
- Replikation auf Topic-Ebene
- Der aktive Broker eines Topics heißt “Leader”
- Hinweis
  - Komplette Ausfallsicherheit kann nicht ohne Aufwand gewährleistet werden
    - Wie viele Replikationsserver müssen beim Eintreffen einer neuen Nachricht synchronisiert werden, bis der Producer einen erfolgreiche Nachrichtenübermittlung signalisiert bekommt?

Die Rolle des Zookeepers

- Kennt alle aktiven Broker
- Kennt auch alle aktuellen Leader
  - Für jedes Topic ein “Locking”
- Speichern der aktuellen Offsets der Consumer
- Vorsicht:
  - Work in Progress
  - Neue Kafka-Releases vermeiden immer mehr die Datenhaltung im Zookeeper

Exkurs: Daten-Replikation und Daten-Konsistenz

- Problematik, die alle Datenbank-Systeme trifft
- CAP-Theorem
  - Consistency
  - Availability
  - Partition Tolerance (Können Ausfälle im Netzwerk zwischen Broker-Knoten toleriert werden?)
  - “Nur zwei aus drei möglich”
- Kafka muss dem CAP-Theorem folgen
  - In Sync Replication (ISR)
  - Out of sync Replication



Topics

Topics und Partitionen

- Jedes Topic besteht aus zumindest einer Partition
- Jede Partition wird von exakt einem Broker verwaltet
- Consumer lesen Nachrichten aus einer Partition
- Beim Anlegen eines Topics kann die Anzahl der Partitionen sowie die Anzahl der Replikationen gewählt werden
- `kafka-topics.bat --zookeeper h2908727.stratoserver.net:2181 --create --topic <my_topic> --partitions 3 --replication-factor 3`

Partitionen

- Skalierbarkeit der Daten aus Sicht des Producers
- Aus Sicht der Consumer-Prozesse resultiert ebenfalls Skalierbarkeit
- Vorsicht:
  - Einführen neuer Partitionen ist durchaus aufwändig (!)
  - Ebenso ein Rebalancing der Consumer innerhalb einer Gruppe

Replikations-Faktor

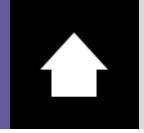
- Resultiert in erhöhter Ausfallsicherheit
- Replication Factor = 1
  - Zeitfenster, in dem produzierte Daten verloren gehen können
    - Broker schreibt in seine Partition
    - und stürzt ab
- Replication Factor = 3
  - Zeitfenster, in dem produzierte Daten verloren gehen können
    - Broker schreibt in seine Partition
    - Broker schreibt in die Replikationsserver
    - und stürzt ab
  - Property “Anzahl der Acknowledgements”
    - ack = 1
    - ack = 3

Consumer Groups

- Jeder Consumer ist einer Consumer Group zugeordnet
  - Identifikation über die `group.id`
- Das Commit eines Offsets ist pro Consumer Group
- Eine Partition eines Topics wird garantiert nur von einem Consumer einer Consumer Group gelesen

Administration und Überwachung

- JMX
- Jolokia



JMX

Java Virtual Machine

- Ein Kafka-Broker ist ein normaler Java-Prozess
  - Heap-Speicher
    - `-Xms, -Xmx`
      - Initialer und maximaler Heap-Speicher der JVM
    - Normale Java Garbage Collection
  - Eigentlich direkter Widerspruch zu den Anforderungen an Kafka
    - Stop-the World Effekt durch Speicherbereinigung, insbesondere: Defragmentierung teilweise unter Benutzung nur einer einzigen CPU
      - Proportional zur Maximalgröße des Speichers
      - Dauer: Von Millisekunden bis hin zu Sekunden
  - Notwendig: Mitschreiben aller Garbage Collections in eine Log-Datei
    - `java -Xloggc:file`

Fehlersituation, die aus der Garbage Collection begründet sein können

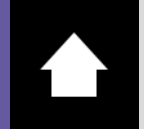
- “Unerklärliche” Einbrüche im Durchsatz
- Zookeeper-Probleme
  - Häufige Leader-Elections
  - Out of Sync/In Sync

Schreiben und Auswerten des GC-Logs

- Im Aufruf des Java-Prozesses muss die Option -Xloggc:file angegeben sein
- Zur Auswertung dieser Datei
  - Jede Garbage Collection ist als Zeile in dieser Datei vorhanden
    - Zeitstempel, Minor/Major, Speicher vorher - nachher, Dauer
  - Editor, Excel, gcviewer oder ähnliches

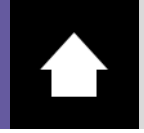
Hinweis zur Implementierung von Kafka

- Intern benutzt Kafka das so genannte “Off Heap-Memory”, “native Memory”
  - So etwas wie eine “RAM-Disk”
  - Hier läuft keine Garbage Collection
  - Zugriffszeit auf Heap ist deutlich schneller



Überwachung ist JMX-basiert

- Standard JVM-Metriken
  - Memory, CPU etc.
    - `java.lang:type=Memory`
- Broker-Metriken
  - messages pro Sekunde
  - Bytes pro Sekunde
  - ...
- Producer und Consumer-Metriken
  - für Java-Clients



Jolokia

Empfehlung

- Benutzen Sie nicht die jconsole und entfernen Sie die Standard-JMX-Konfiguration im Aufruf-Skript
  - Faktisch ein Security Problem
- Ermöglichen Sie den Zugriff auf JMX mit Jolokia
  - https://java.integrata-cegos.de/jolokia-simples-management-von-java-anwendungen/
- Hawtio

Programmieren mit Kafka

- Übersicht
- Trainingsumgebung
- Ein erstes Programm
- Serialisierung und Deserialisierung
- Partitioner
- Kafka Connect und Kafka Stream

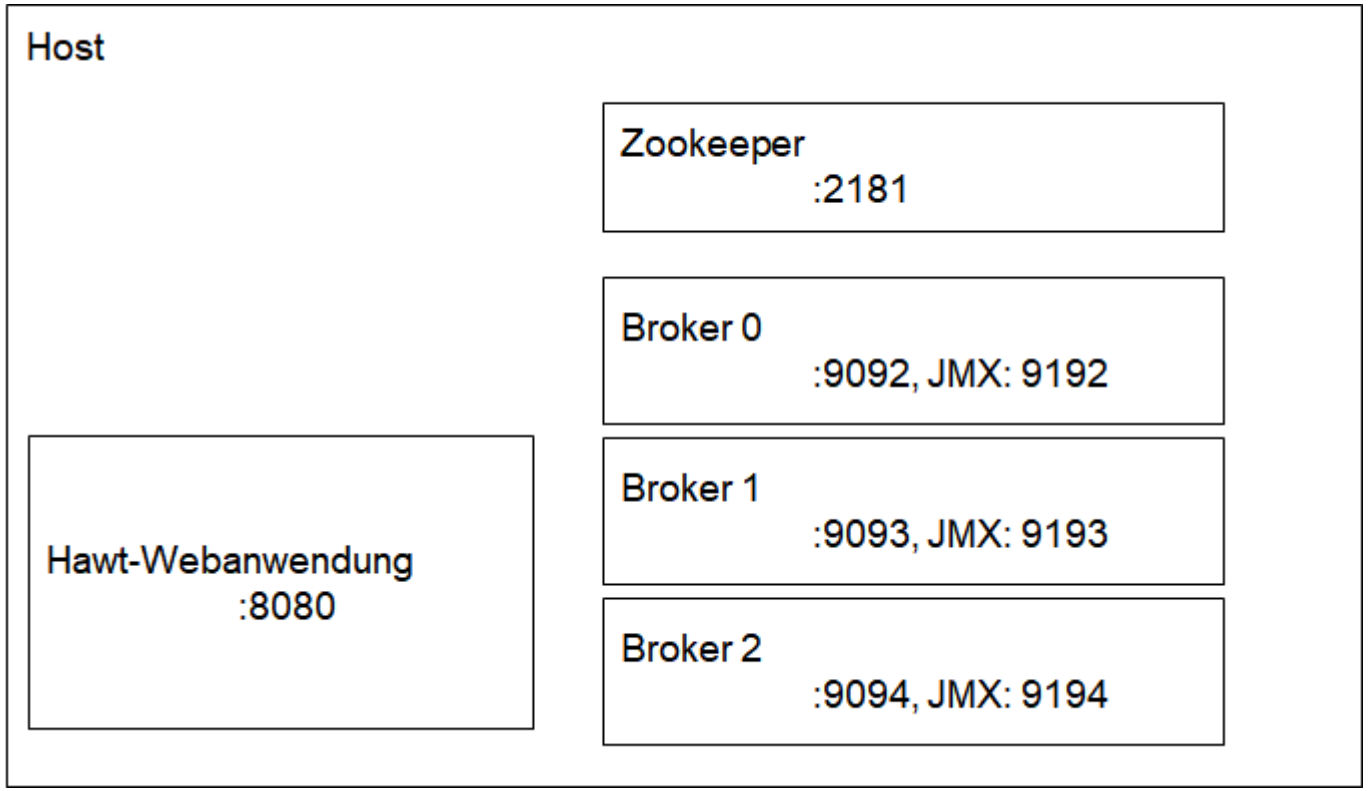
Übersicht

Zugriff auf Kafka

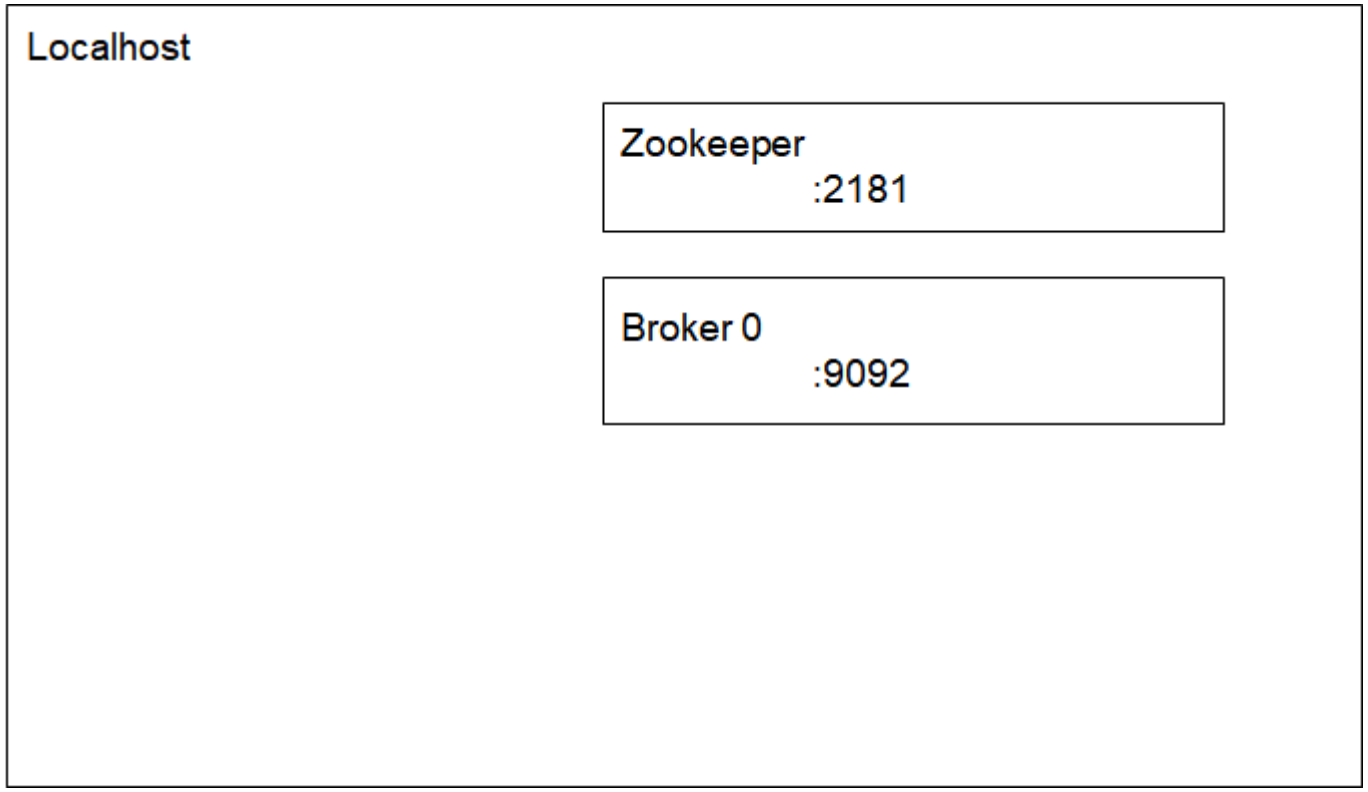
- Klassische Treiber-Software für verschiedene Programmiersprachen
  - z.B. Java-Treiber
  - C#
  - …
- REST-API
  - Interoperabel
- Konkretes Beispiel
  - Maven-basiertes Projekt mit Zugriff auf Kafka

Trainingsumgebung

Eine komplette Kafka-Umgebung



Eine stark vereinfachte Developer-Umgebung



Mehrere Broker auf einer Maschine

- Schritt 1: Kopieren der server.properties z.B. auf *server0.properties, server1…*
- 3 Stellen müssen angepasst werden:
  - Zeile 21: eindeutige Broker-ID
  - Zeile 60: Eindeutiger Name für die Log-Datei, `kafka-log-0, …`
  - Zeile 31: Kommentar entfernen, `listeners = PLAINTEXT://localhost:9092`
- Starten der Broker mit kafka-server-start unter Angabe der jeweiligen *server<n>.properties*

Ein erstes Programm

Kafka aus Sicht einer Anwendung

- Besteht aus einer beliebigen Anzahl so genannter Topics
  - Jedes Topic definiert einen Kommunikationskanal
- Development-Umgebung wird Topics bei Bedarf selber erzeugen
  - In der Realität ist diese selbstverständlich ein administrativer Akt

Eine erste Kafka-Anwendung

- Simpler Producer und eine simpler Consumer werden über ein gemeinsam bekanntes Topic “verbunden”
  - https://github.com/Javacream/org.javacream.training.kafka/tree/d23e6b80c992988b39c03172a221191841b31ac8

Der Producer-Client

- Treiber-Bibliotheken realisieren eine Thick Client
  - Arbeitet mit einer initialen Broker-Liste
    - Erste antwortende Broker sendet eine Meta-Information zurück, welche Broker aktuell vorhanden sind
  - Automatischer Retry
  - Batch-Modus
    - Ein Sendevorgang wird nicht direkt ausgeführt, sondern es existiert eine Queue
  - Sende-Verfahren
    - “Fire and Forget”
    - Synchron oder Asynchron
- Konfigurationseinstellungen
  - z.B. Batch-Size, “Sammel-Zeit” für den Batch, ..

Der Consumer Client

- Fetch-Size
- Commit
  - Der Offset ist der “Merker”, der mitteilt, welche Messages bereits einem Consumer übermittelt wurden

Produce revisited

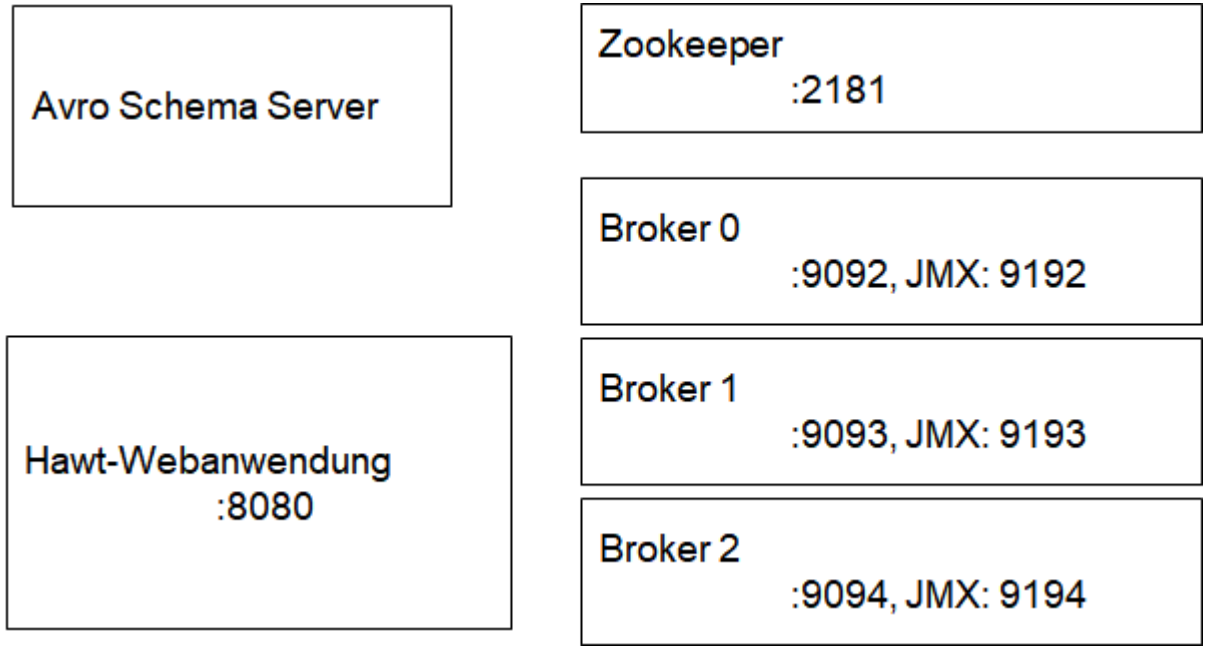
- Bisher:
  - `topic, value`
- Nun
  - `topic, key, value`
  - `key` bestimmt die Partition (!)
  - *`kafka-run-class.bat kafka.tools.GetOffsetShell --broker-list … --topic --time -1`*

Serialisierung und Deserialisierung

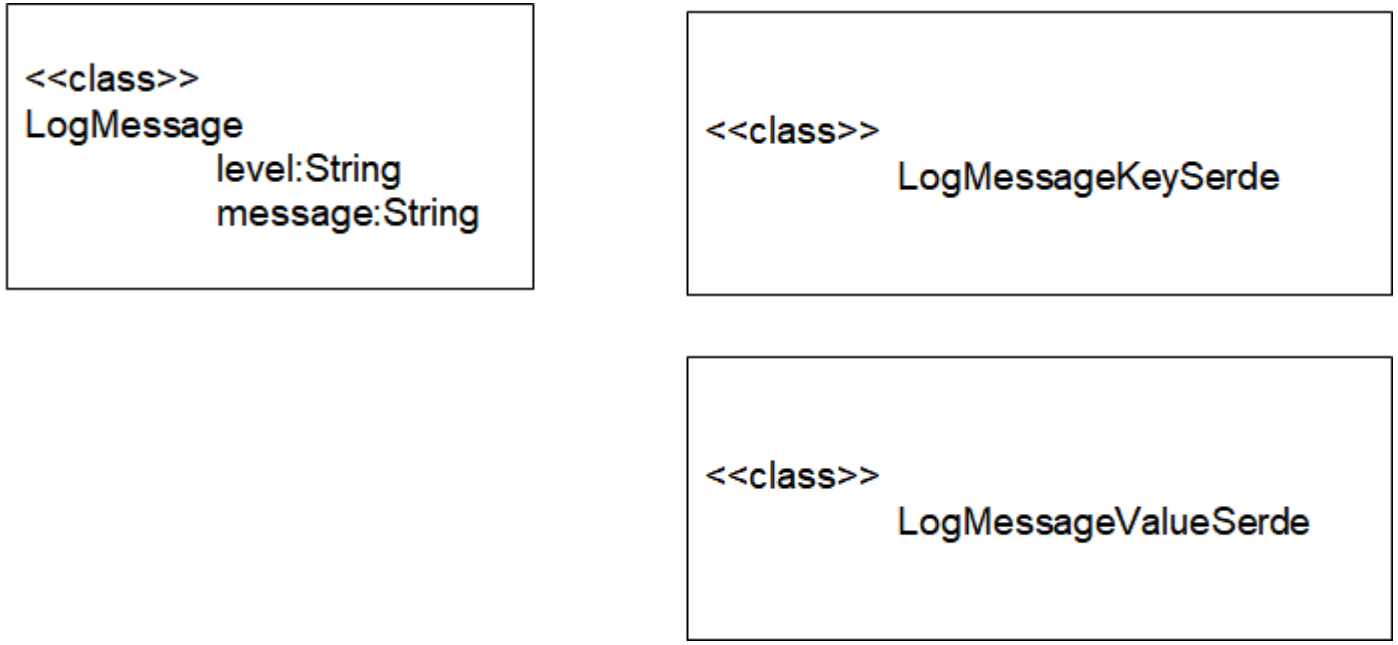
Unterstützte Datentypen

- Standard-Java-Datentypen
  - `String, int, …`
- Für die Values interessant: JSON-Serializer
- Avro
  - Serialisierung und Deserialisierung ihrer Java-Objekte in ein Schema-behaftetes Dokument
- Custom-Datentypen
  - Zwei Schnittstellen im Kafka-Paket
    - `Serializer<T>`
    - `Deserializer<T>`
  - Typische Implementierung implementiert beide Schnittstellen
    - “Serde”

System mit Avro



Beispiel


<<class>>
LogMessage
level:String
message:String
<<class>>
LogMessageKeySerde
<<class>>
LogMessageValueSerde

Partitioner

Partitioner

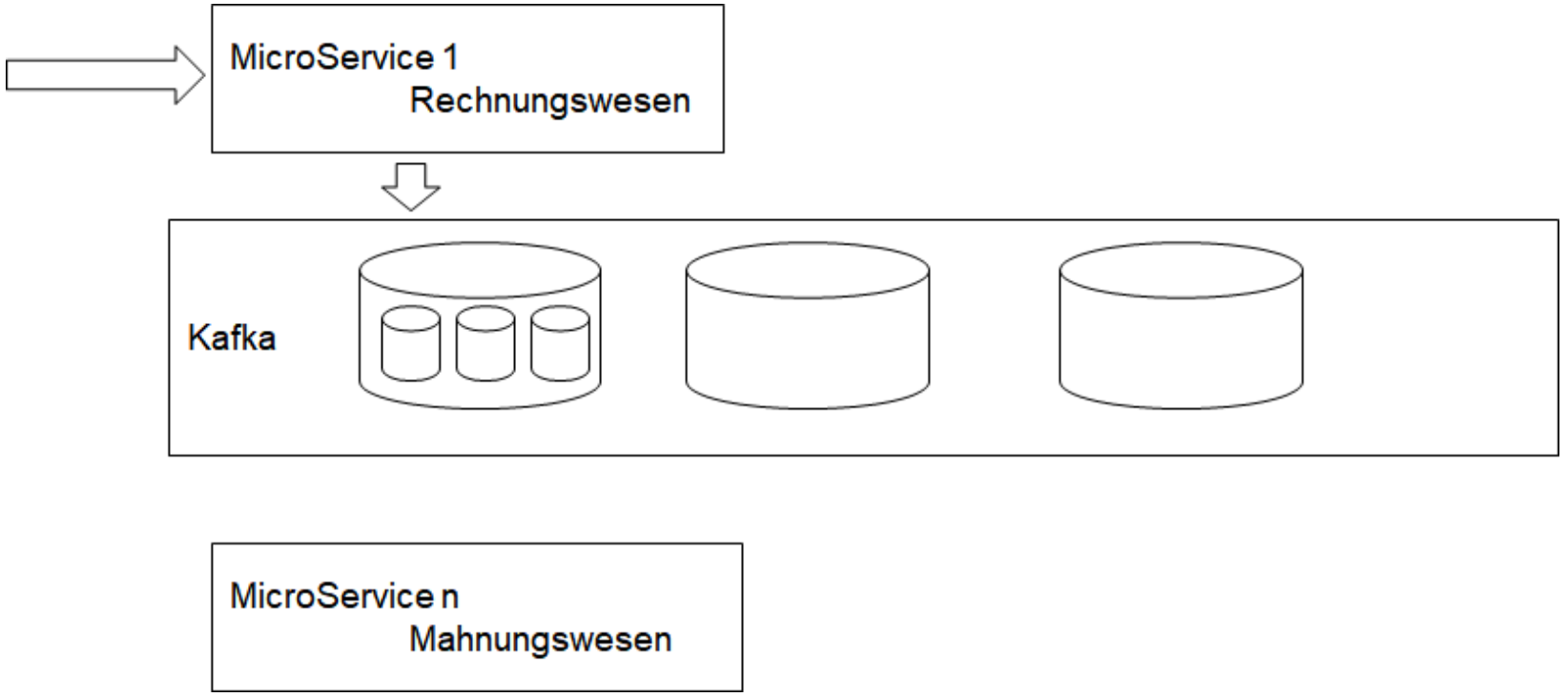
- Bestimmt aus dem Key die zugehörige zu benutzende Partition

```
interface Partitioner{
     int partition(String topic, Object keyObject, byte[] keyRaw,
            Object valueObject, byte[] valueRaw,
            Cluster kafkaCluster){
     partitionInfos = kafkaCluster.availablePartitionsForTopic(topic);
 }
```

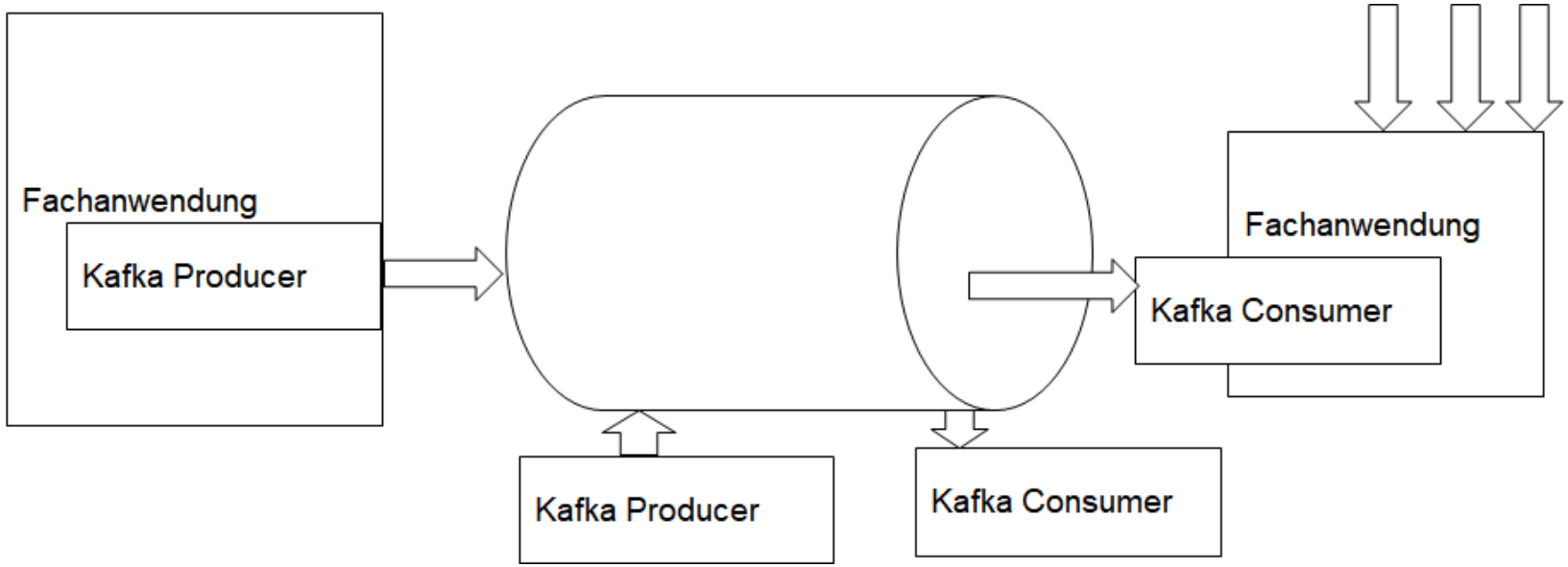
- Rückgabewert: Nummer der für diesen Key zu benutzenden Partition
- Bei der Consumer-Registrierung können Topic und Partitionsnummer berücksichtigt werden

Kafka Connect und Kafka Stream

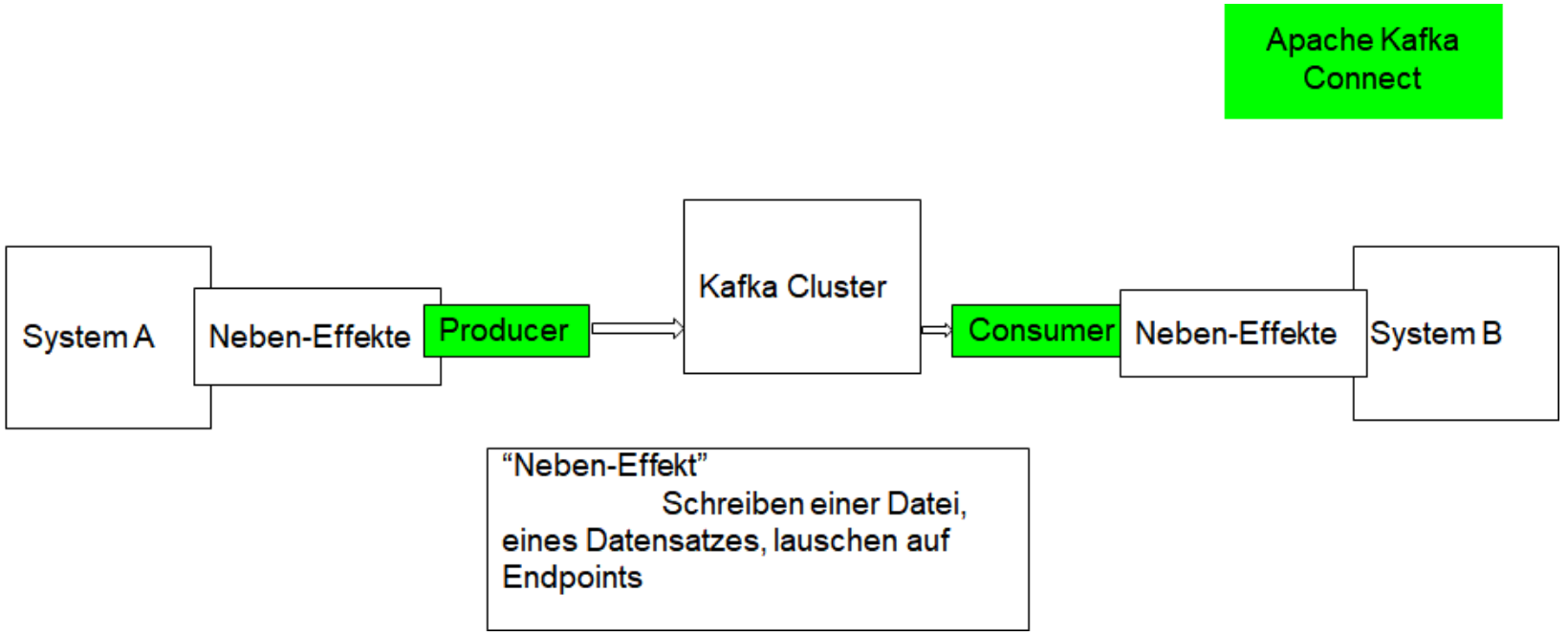
Kafka in einer Microservices Landschaft



Bisheriger Stand



Enterprise Application Integration (EAI)



Kafka Connect

- Producer und Consumer
  - Separate Anwendungen
    - unabhängig vom Kafka Cluster zu betrieben
    - Anbindung an den Cluster: konfigurativ
  - Standalone
    - mit einem simplen integrierten Kafka Broker
- Start mit
  - `bin\connect-distributet.sh config\connect-distributed.properties`
  - Ab jetzt: REST-Endpoint auf Port `8083`

Producer und Consumer mit Connect

- rein konfigurativ
- JSON-Dokument beschreibt z.B. einen Producer

```
{'name':'demo-producer', config: {'connector-class':
'...FileSource'}}
```

```
{'name':'demo-consumer', config: {'connector-class':
'...FileSink'}}
```

Anwendungsfall: Extract - Transform - Load

- Spezialfall eines Streaming-Prozesses
- Datenverarbeitungslogik
  - Datenquelle (hat kein definiertes Ende)
  - Verarbeitungs-Pipeline
    - `filter`
    - `transform/map`
    - `aggregate`
    - `reduce`
  - Ohne `aggregate/reduce`: Weiterleitung in eine Datensenke

Kafka Streams

- KStream bietet ein Streaming-API
- Operiert auf Topics/Partitions
  - permanenter Datenstrom
- filter und map-Operationen sind unkritisch
  - Das sind “pure functions”: Parameter - Function - Result
- Aggregat-Funktionen, z.B. Maximalwert oder Minimalwert einer Zahlenkolonne? Mittelwert?
  - Im Extremfall benötigt die Aggregation Zugriff auf alle Daten
  - Kafka ist ab jetzt ein Applikationsserver