



Apache Kafka



Inhalts- verzeichnis



Einführung



Programmieren mit Kafka



Realisierung eines skalierenden Systems



Administration und Überwachung



Aufsetzen eines Kafka-Systems



Kafka Erweiterungen



Arbeitsweise von Kafka



Topics



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
- Trend zu Services- und Microservices-Architekturen
 - Eine Fachanwendung besteht heute aus einer Vielzahl unabhängiger Prozesse



Allgemeine Problemstellung

- Jedes dieser Systeme produziert eine ganze Menge von (Log-)Informationen
 - 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 ist notwendig
- Ebenso kommunizieren die Fach-Prozesse intensiv miteinander
 - Mit unterschiedlichen Garantien der Nachrichten-Übermittlung
 - „Fire and Forget“ für Heartbeats
 - „At Least Once“ für Aktualisierungsnachrichten
 - „Exactly Once“ zur konsistenten Datenhaltung



Etablierte Lösungen existieren

- Zentrales Logging
 - ElasticSearch
 - Splunk
 - ...
- Kommunikation über Messaging-Systeme oder Event-Bus
 - WebSphere MQ
 - ActiveMQ
 - ...



Warum dann noch Apache Kafka?

- Eine berechtigte Frage!
 - „Kafka löst keine neuartigen Probleme“
 - „Kafka ist nicht konkurrenzlos“
 - „Statt eines neuen Systems setzen wir auf bewährte Lösungen“
- Kafka ist aber definitiv ein gutes Produkt
 - Moderne, skalierbare Cluster-Architektur mit geringem Administrationsaufwand
 - Verfügbare und ausfallsichere Datenhaltung auch im Hochlastbereich
 - Einfaches Programmiermodell mit Treibern für die gängigen Programmiersprachen
 - Mit Kafka Connect und Kafka Streams stehen noch weitere mächtige Bibliotheken zur Integration und Datenanalyse zur Verfügung



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



Realisierung eines skalierenden Systems



Cluster-Ansätze



Kafka broker



Cluster-Ansätze

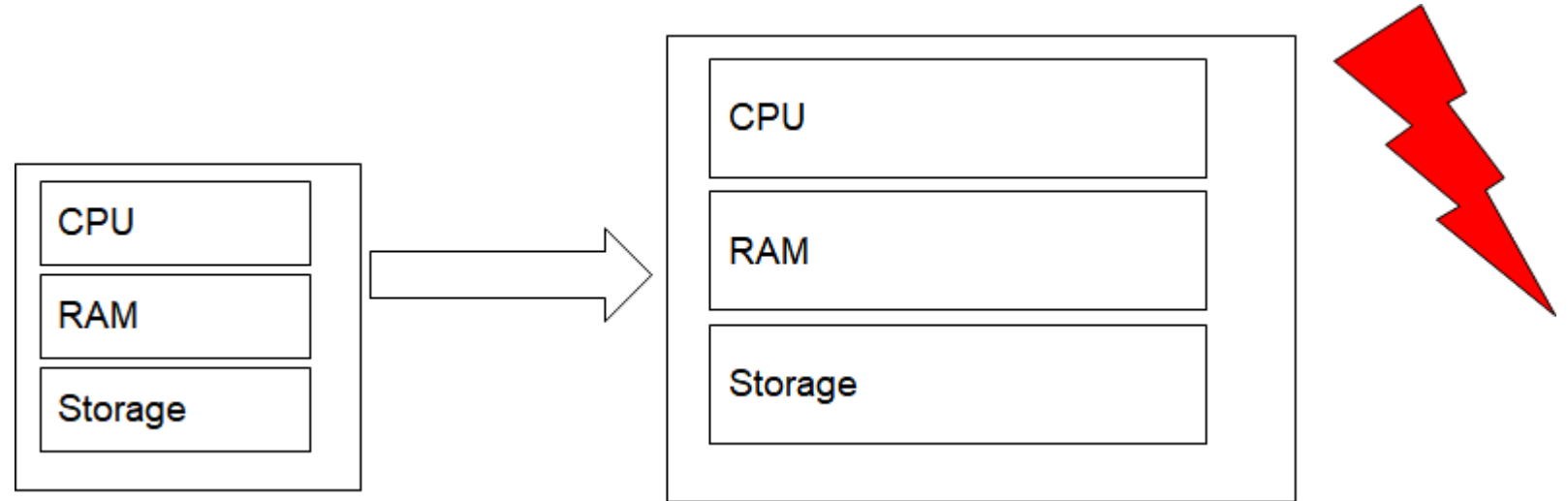


Erste Möglichkeit: Vertikale Skalierung

- Limits in CPU/RAM/Storage werden durch die Migration auf ein größeres System behoben
- Konsistente Datenhaltung ist relativ zu erreichen



Vertikale Skalierung



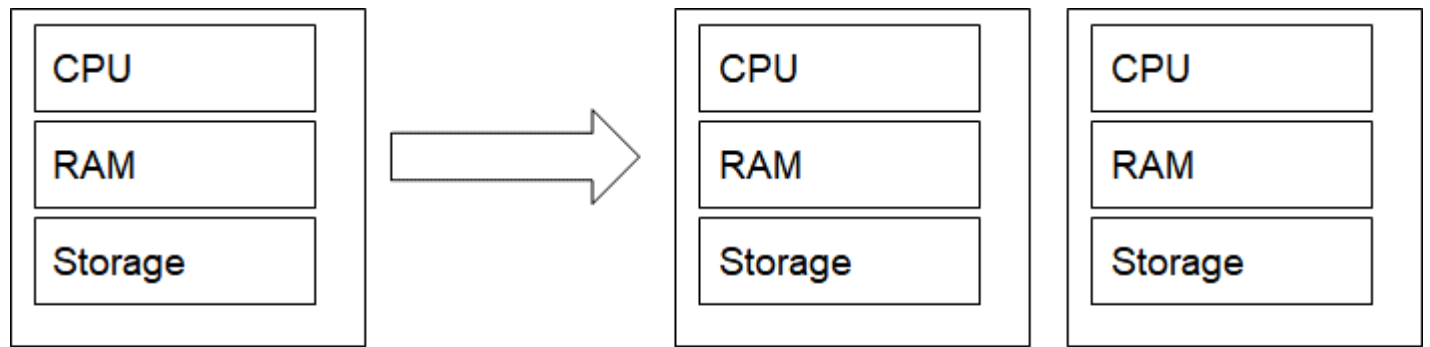


Zweite Möglichkeit: Horizontale Skalierung

- Skalierung durch dynamisches Hinzufügen (up-scaling) oder Entfernen (down-scaling) von Brokern
- Datenkonsistenz ist hier aufwändiger zu realisieren



Zweite Möglichkeit: Horizontale Skalierung



Kafka ist ein horizontal skalierendes System

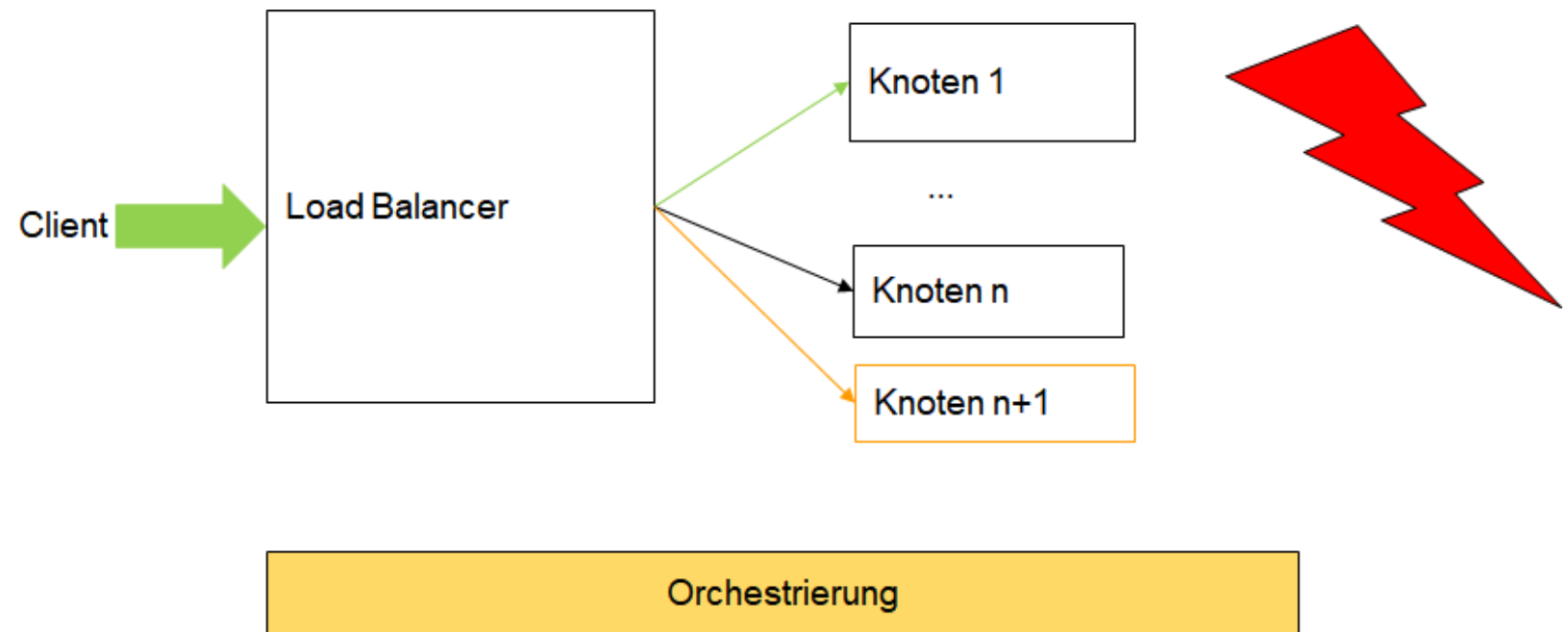


Mögliche Realisierung

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

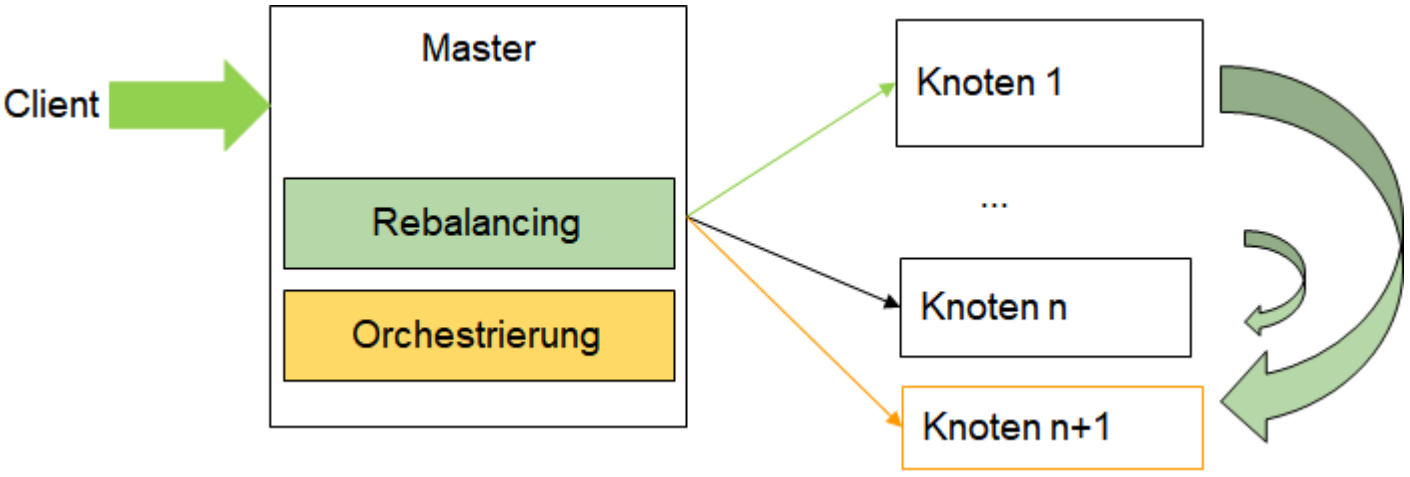


Stateless Broker



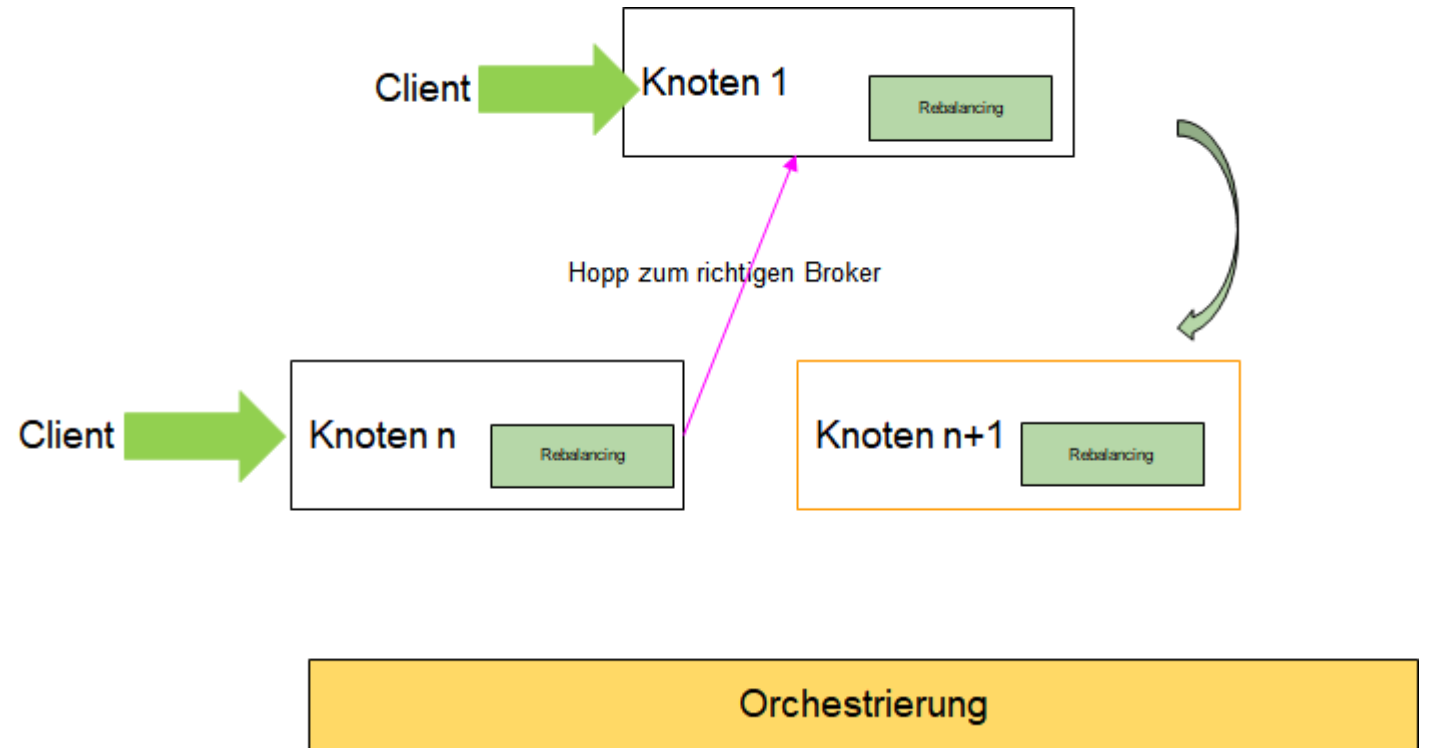


Master-Broker





Masterless Broker





Bewertung der Ansätze

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



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



Kafka broker



Realisierung in Kafka

- Master
 - Apache Zookeeper
 - Ablage des Cluster-States
 - Broker-State
 - Nicht die eigentliche Datenhaltung!
 - Locks
 - Weitere Features des Zookeepers werden von Kafka nicht genutzt
- Kafka-Broker kommunizieren auch in einer Ring-Architektur
 - Weiterleiten von Client-Requests an den “richtigen” Broker
 - ...



Warum beide Varianten in einem Produkt?

- Master-Broker ist einfacher zu realisieren als ein Ring, sobald Datenkonsistenz gefordert ist
- Masterless-Ring ist aber deutlich einfacher in der Administration
 - “admin-less”
- Kafka “work in progress”
 - Die ersten Kafka-Versionen waren klar Master-Broker
 - Mit jedem Release wurde die Benutzung des Zookeepers zurückgefahren
 - Geplant ist Kafka als kompletter Ring-Cluster



Kafka-Ring

- Controller
 - Ein bevorzugter Broker, der den gesamten Ring kontrolliert
 - Wird von allen Brokern “gewählt”
- Leader
 - Die aktive Datensenke/Datenquelle



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

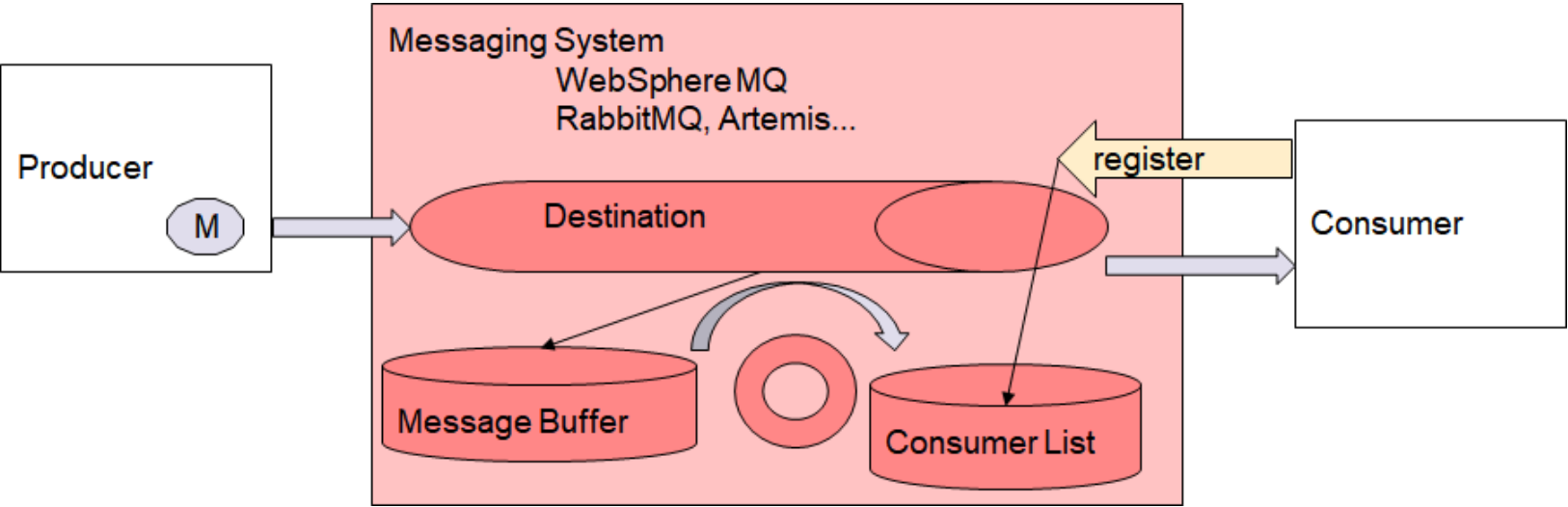


Ein laufendes Kafka-system



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

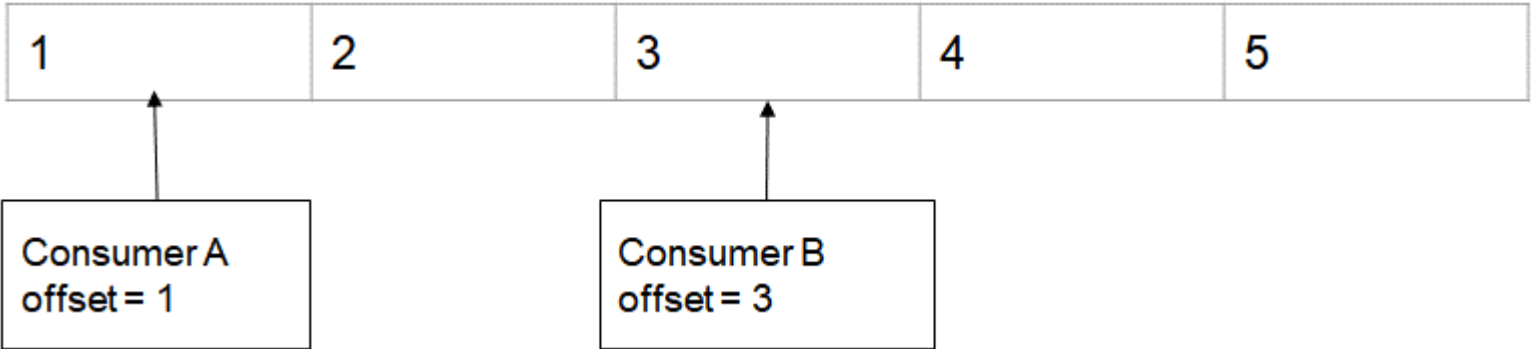


Kafka

- “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

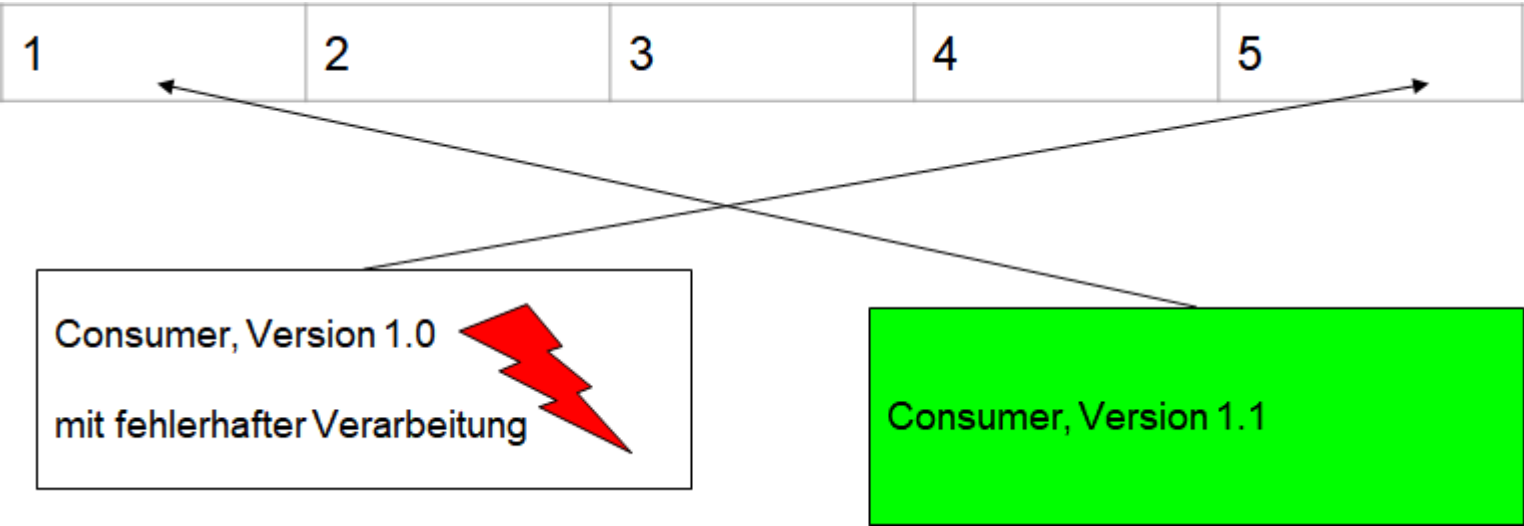


Unabhängigkeit der Consumer





Beispiel: BugFix eines Consumers





Skalierung

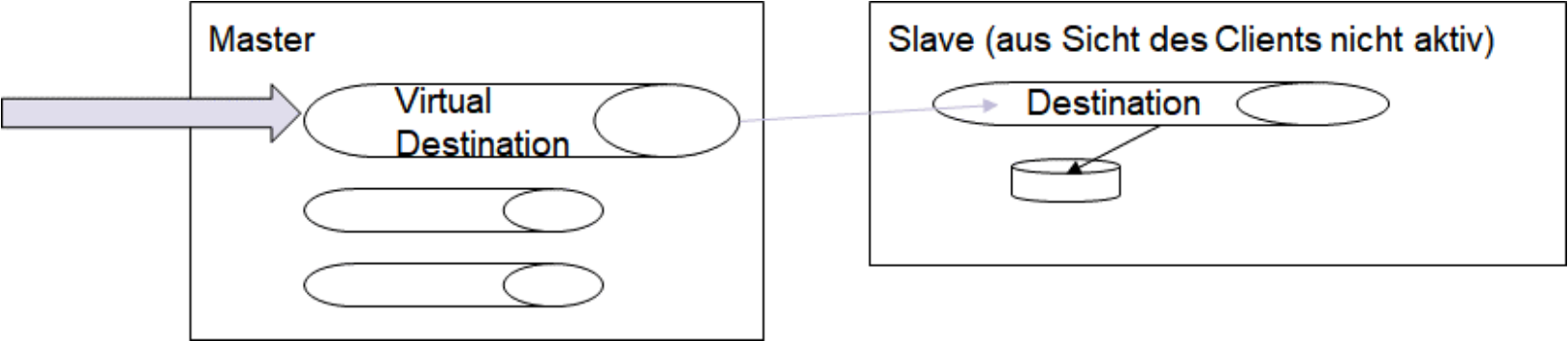


Cluster-Lösung bei Messaging-Systemen

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

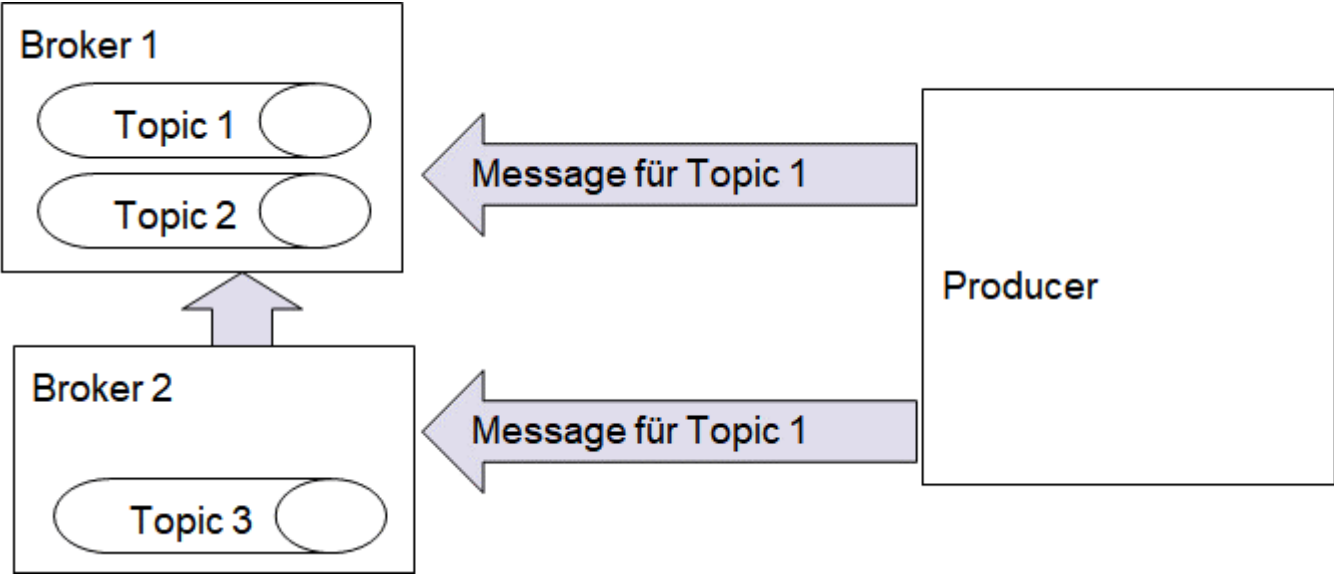


Cluster-Lösung bei Messaging-Systemen





Kafka

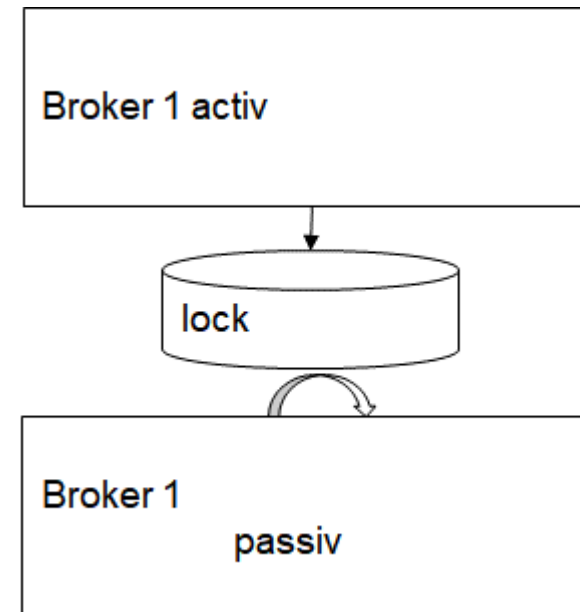




Ausfallsicherheit

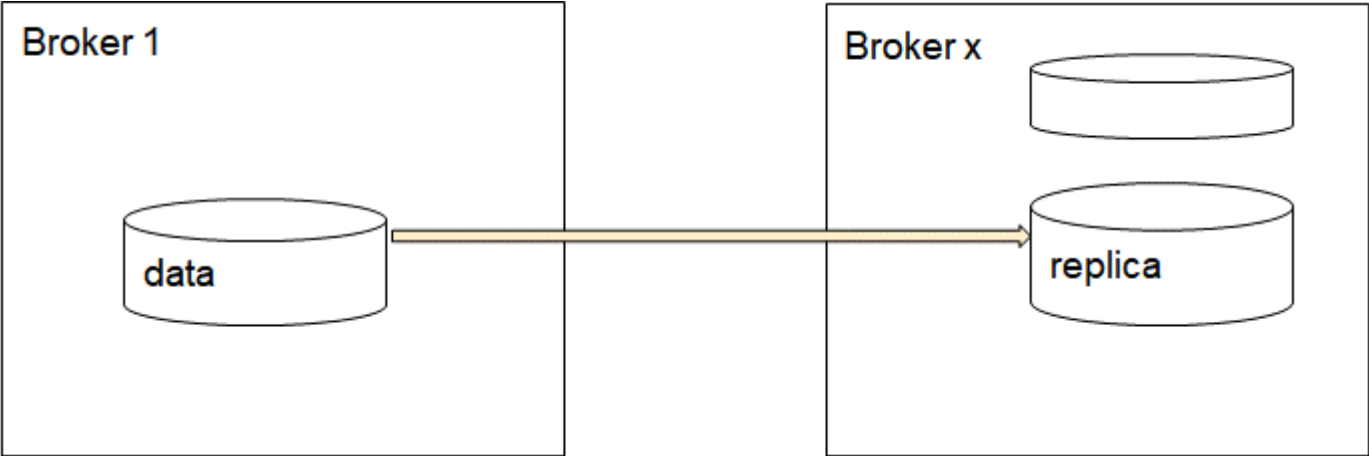


Ausfallsicherheit: Aktiv - Passiv





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 eine 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



Ein laufendes Kafka-system

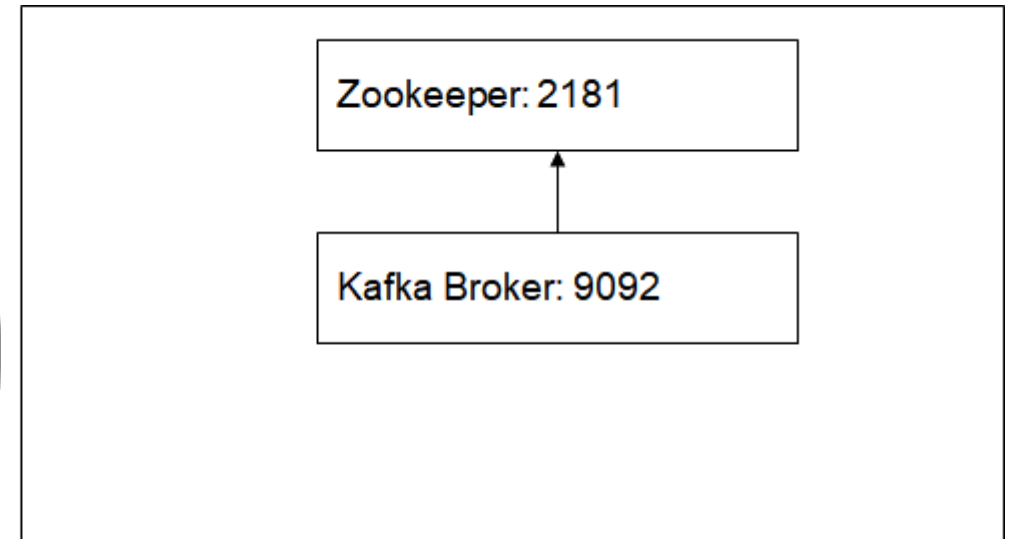
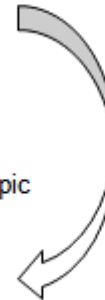


Einfache lokale Kafka-Umgebung

```
kafka-console-producer.sh|bat  
--broker-list localhost:9092 --topic demo
```

Eingabe-Prompt
> Hello <Enter>

```
kafka-console-consumer.sh|bat  
--bootstrap-server localhost:9092 --topic  
demo  
Blockierende Console  
Hello
```



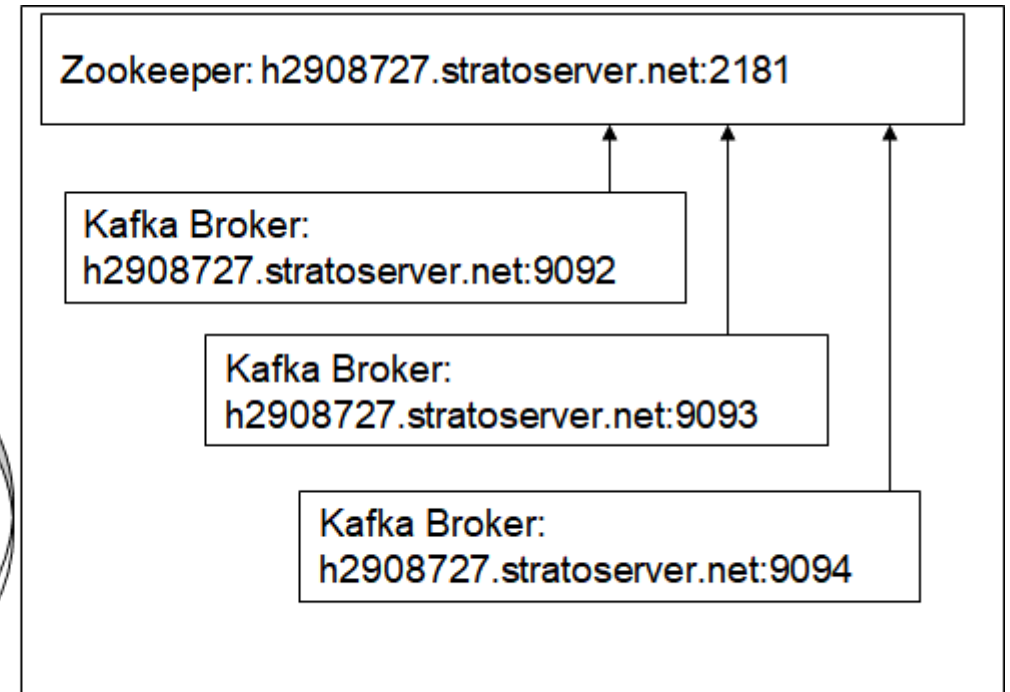


Ein remote Kafka-Cluster

```
kafka-console-producer.sh|bat  
--broker-list h2908727.stratoserver.net:9092  
--topic demo<namenskürzel>
```

Eingabe-Prompt
> Hello <Enter>

```
kafka-console-consumer.sh|bat  
--bootstrap-server  
h2908727.stratoserver.net:9092 --topic  
demo<namenskürzel>  
Blockierende Console  
Hello
```





Topics



Begriffe



Detaillierte Arbeitsweise



Garantien der
Nachrichtenübermittlung



Kafka und Services in der Cloud



Begriffe



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



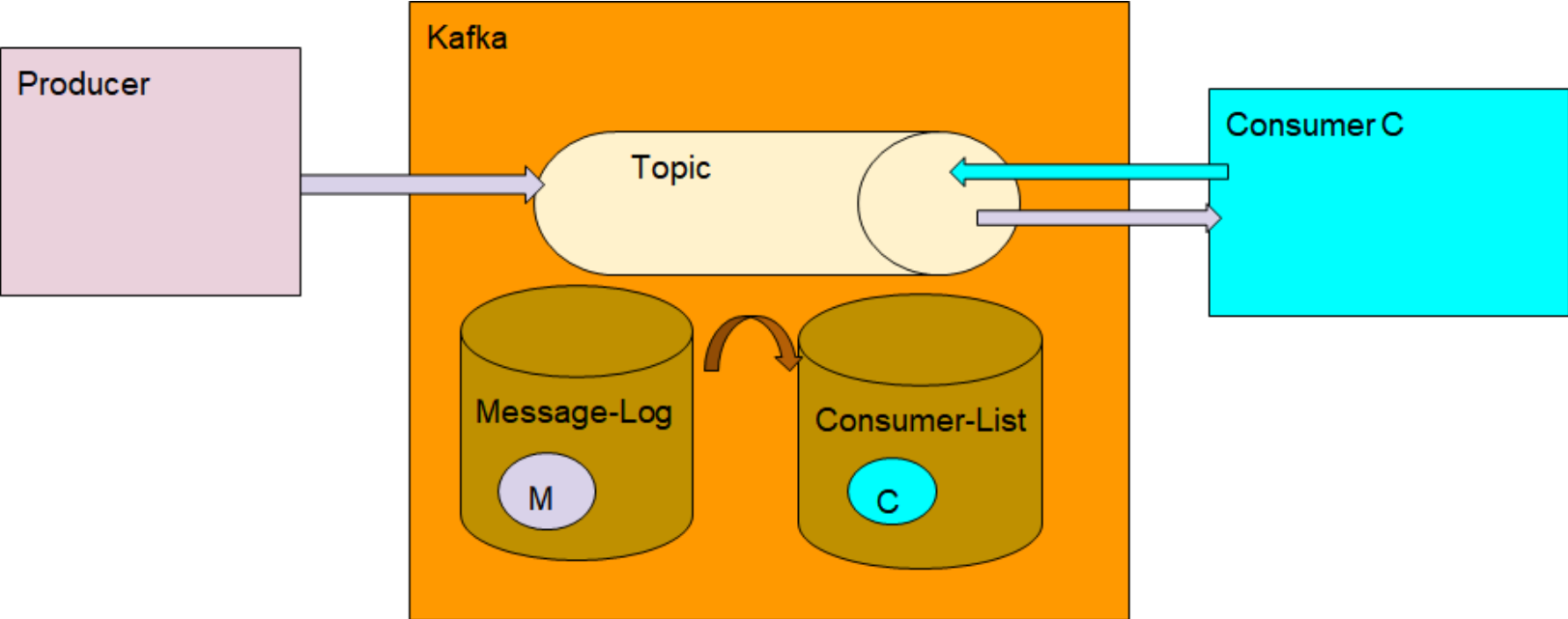
Consumer Groups

- 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



Detaillierte Arbeitsweise

Kafka-Messaging





Message Log

- “Kafka löscht nicht”
- Das Message Log ist append-only
- Schreiben einer Message in das Log ist extrem schnell
- Der Stand der bereits ausgelieferten Nachrichten wird im so genannten “commit-log” festgehalten
 - Speichert den Stand “welche Nachrichten wurden ausgeliefert”
 - Jeder Consumer kann relativ oder absolut einen offset angeben und so auch bereits ausgelieferte Nachrichten konsumieren



Kafka arbeitet als Datenbank-System

- Genauer: Ein Key-Value-Store
 - producer “insert into”
 - consumer “select * from where offset = ”



Im Detail: Retention

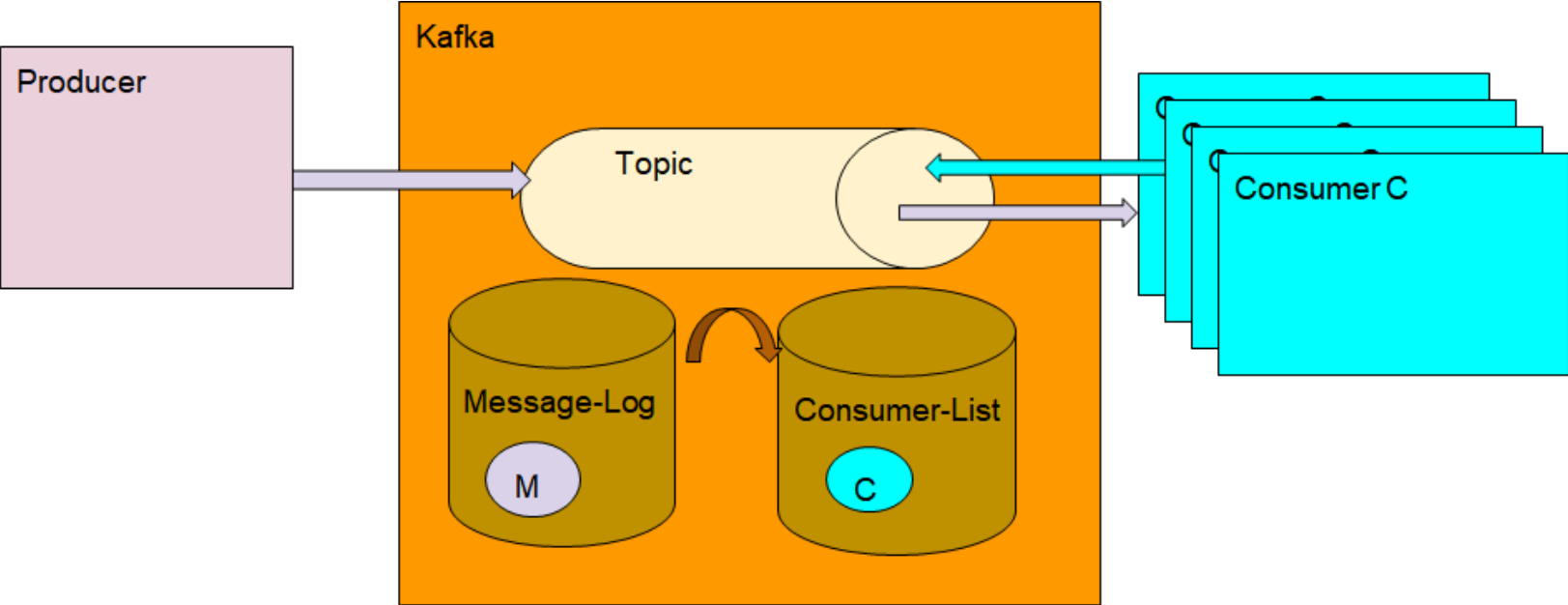
- Die Daten des Message Logs werden bereinigt
 - Sie haben eine Verweildauer
- Konfiguration
 - Gültigkeit
 - pro Broker
 - pro Topic
 - Angabe
 - Größe des Message Logs
 - Dauer
- Kriterium zum Löschen ist die Vereinigung



Konzeption der Topics

- Definition von Kommunikations-Kanälen
- Parallelisierung von Consumer-Prozessen
 - Jedes Topic übermittelt seine Nachrichten an potenziell mehrere Consumer
 - Jeder registrierte Consumer verwaltet sein eigenes commit

Kafka-Messaging: Parallelisierung der Consumer





Skalierung

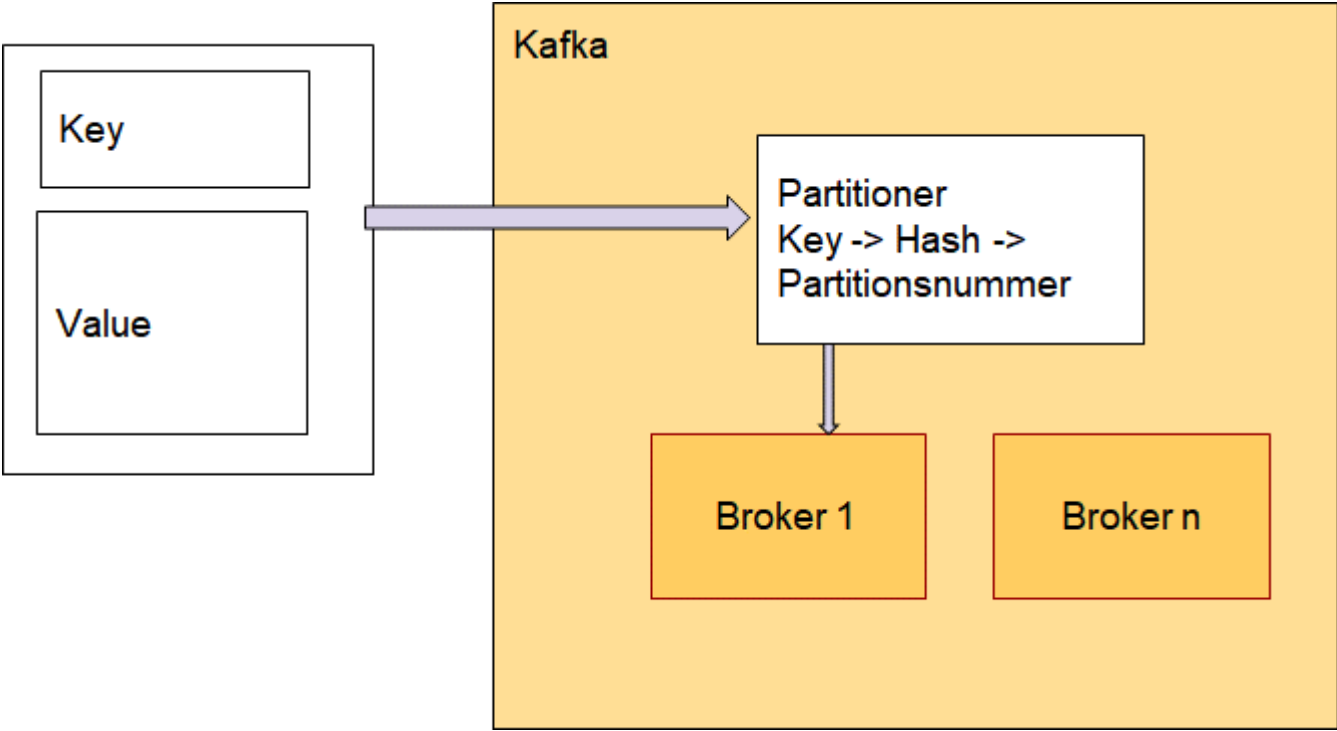
- Aus Sicht der Consumer
 - Klassisches Messaging: Queue
 - Jede Nachricht erreicht exakt einen Consumer
 - Mehr als ein Consumer registriert
 - Lastverteilung der Queue über Round Robin oder First Available
 - “Point-to-Point”-Messaging wird von Kafka nicht unterstützt
- Aus Sicht der Producer
- Aus Sicht des Brokers
 - Verteilte Datenhaltung



Kafka-Lösung der Skalierung

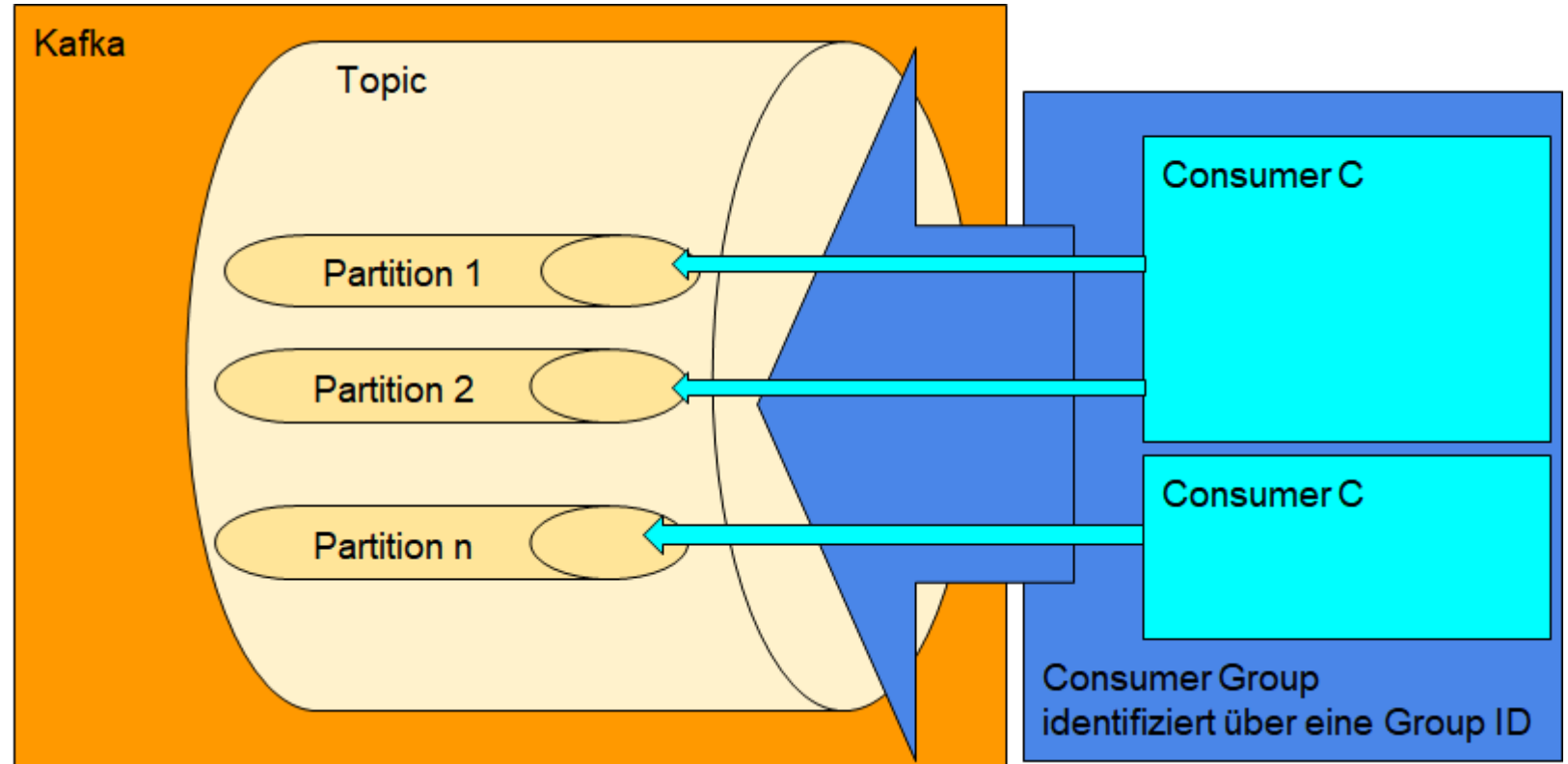
- Partitionierung der Topics
- Jeder Broker hält “nur” eine Partition des Topics
 - Damit enthält jedes Topic mindestens eine Partition
- Vorsicht: Eine Veränderung der Anzahl der Partitionen eines Topics ist möglich aber aufwändig
 - Anzahl der Partitionen sollte im Vorfeld festgelegt werden
 - Ist nicht wirklich dynamisch
- Producer schreiben die Daten in den Cluster “unter Angabe der Partition”
 - unter Angabe eines Keys

Messages in Kafka

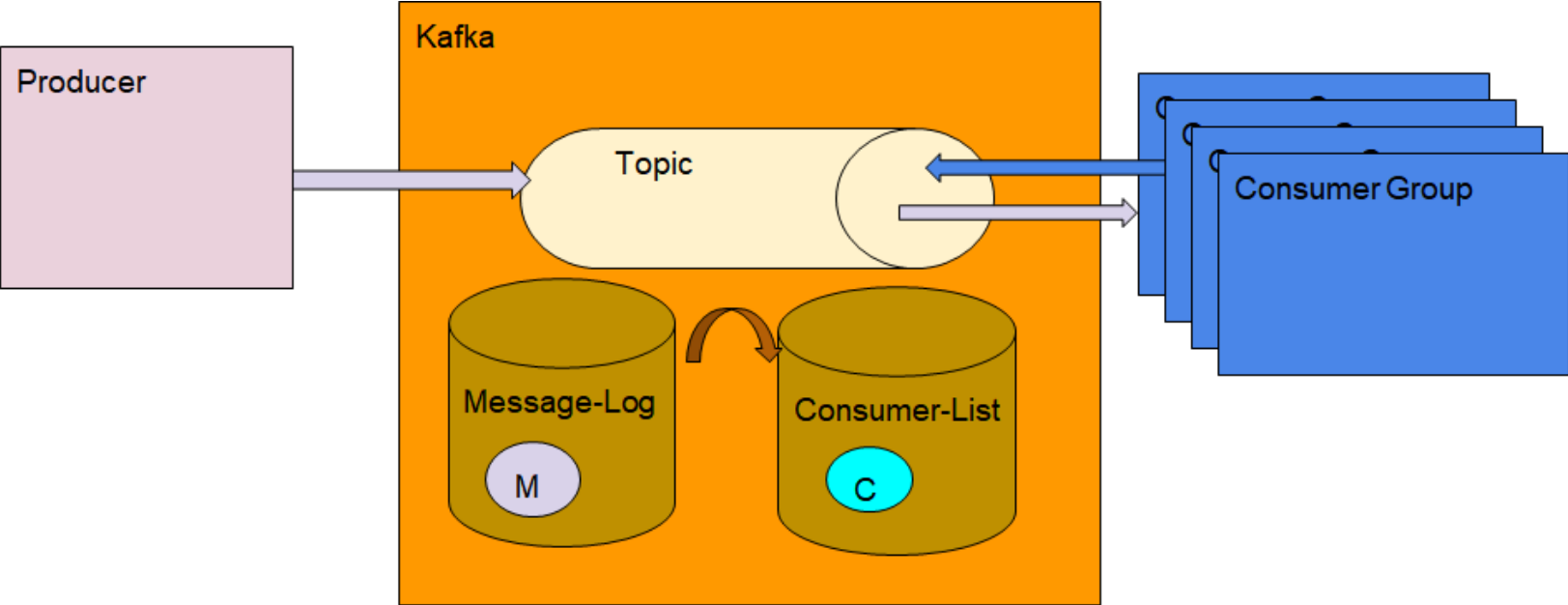




Consumer-Skalierung



Parallelisierung mit Consumer Groups





Garantien der Nachrichten- übermittlung



Garantien der Nachrichten-Übermittlung

- Andere Systeme
 - Klassische Datenbank-Systeme nutzen ACID-Transaktionen
 - Atomic, Consistent, Isolated, Durable
 - Messaging-Systeme
 - Point-to-Point
 - Jede Nachricht wird garantiert von einem einzigen Consumer empfangen
 - Publish-Subscribe
 - Jede Nachricht wird garantiert an alle aktiven Consumer gesendet
 - Publish-Durable Subscription
 - Jede Nachricht wird garantiert an alle registrierten Consumer gesendet



Garantien der Nachrichten-Übermittlung

- Kafka
 - Als Datenbank-System ist Kafka eine Realisierung einer BASE-Architektur
 - Basically Available, Soft State, Eventual (“im Endeffekt”, nicht “eventuell”) Consistency
 - Nachrichten
 - Anything...
 - At most once
 - Jede Nachricht wird maximal einmal verarbeitet
 - beinhaltet: 0
 - At least once
 - Jede Nachricht wird minimal einmal verarbeitet
 - beinhaltet Duplikate
 - Exactly once
 - Jede Nachricht wird exakt einmal verarbeitet



Konsistenz im Ausfall-Fall

- Der Leader einer Partition garantiert für sich selber sehr wohl Konsistenz
- `producer.send(...)`
 - Batch-Modus, potenziell Asynchron
- Retry-Mechanismen
 - Netzwerkproblem ist ein Kriterium für einen Retry
 - Duplikate sind möglich
- Acknowledge
 - 0, 1, all
- Transactional
 - Garantierte Nachrichtenübermittlung ohne Duplikate
 - Im Cluster wird nun zusätzlich ein Transaction-Log geschrieben
 - Producer-ID führt dazu, dass der Broker Duplikate erkennen kann



Verarbeitung durch den Consumer

- Aktuell: “Auto-Commit”
 - Hintergrund-Prozess, der in einem konfigurierbaren Intervall committed
 - Beim Ausfall des Consumers
 - Offset der Partition ist nicht hochgezählt
 - Duplikate
- Ohne AutoCommit
 - Synchrones Commit
 - Asynchrones Commit
- Auch der Consumer kann den Commit des Nachrichtenempfangs transaktionell durchführen



Wie mache ich was?

- Any
 - Kafka Default
 - Reasonable Default für den Einsatz von Kafka zur losen Kopplung von Microservices
- At Most Once
 - Producer sendet synchron mit ack=all
 - Consumer
 - Disable Auto Commit
 - Synchrones Commit
 - Streng: Transaktionelles Commit
- At Least Once
 - Producer sendet synchron mit ack=all
 - Consumer
 - Auto Commit
- Exactly Once
 - Transaktioneller Consumer und Producer



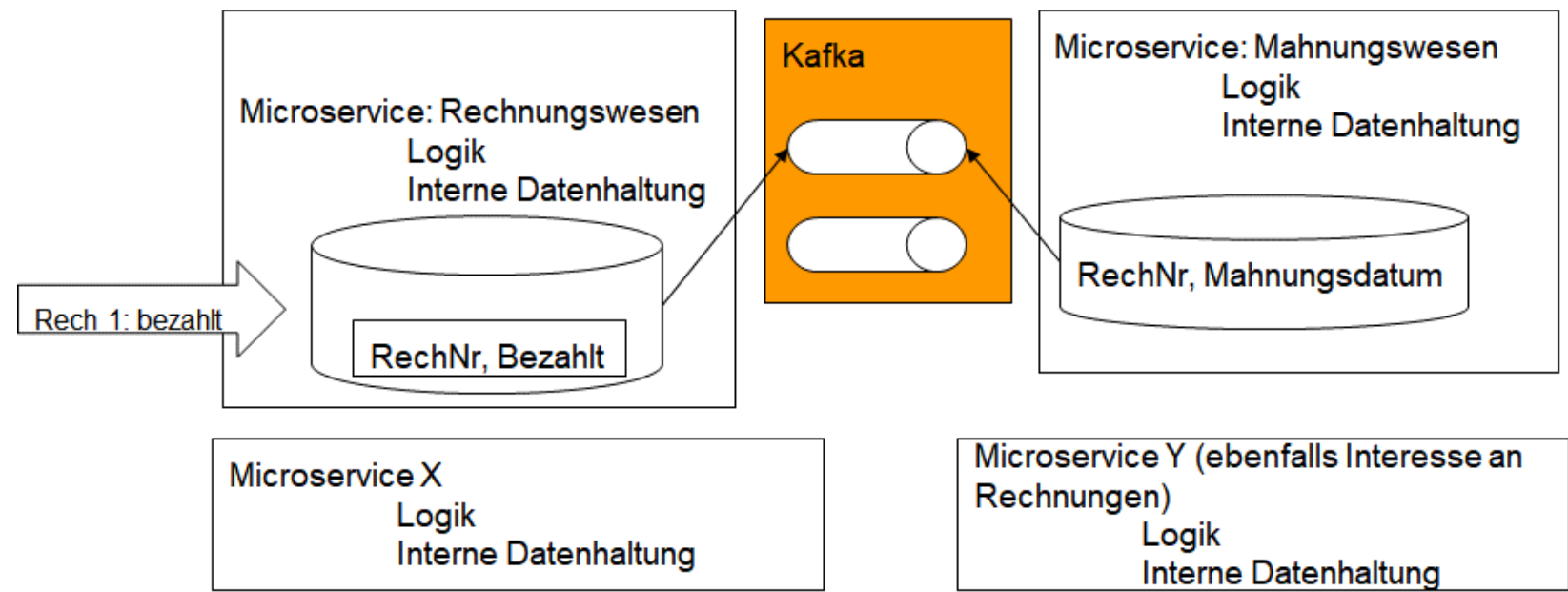
Beispiel

- Transaktionell
 - <https://github.com/eugenp/tutorials/blob/master/libraries-6/src/main/java/com/baeldung/kafka/TransactionalWordCount.java>
 - <https://www.baeldung.com/kafka-exactly-once>
- Garantien zur Nachrichtenverarbeitung in Kafka
 - <https://medium.com/@andy.bryant/processing-guarantees-in-kafka-12dd2e30be0e>

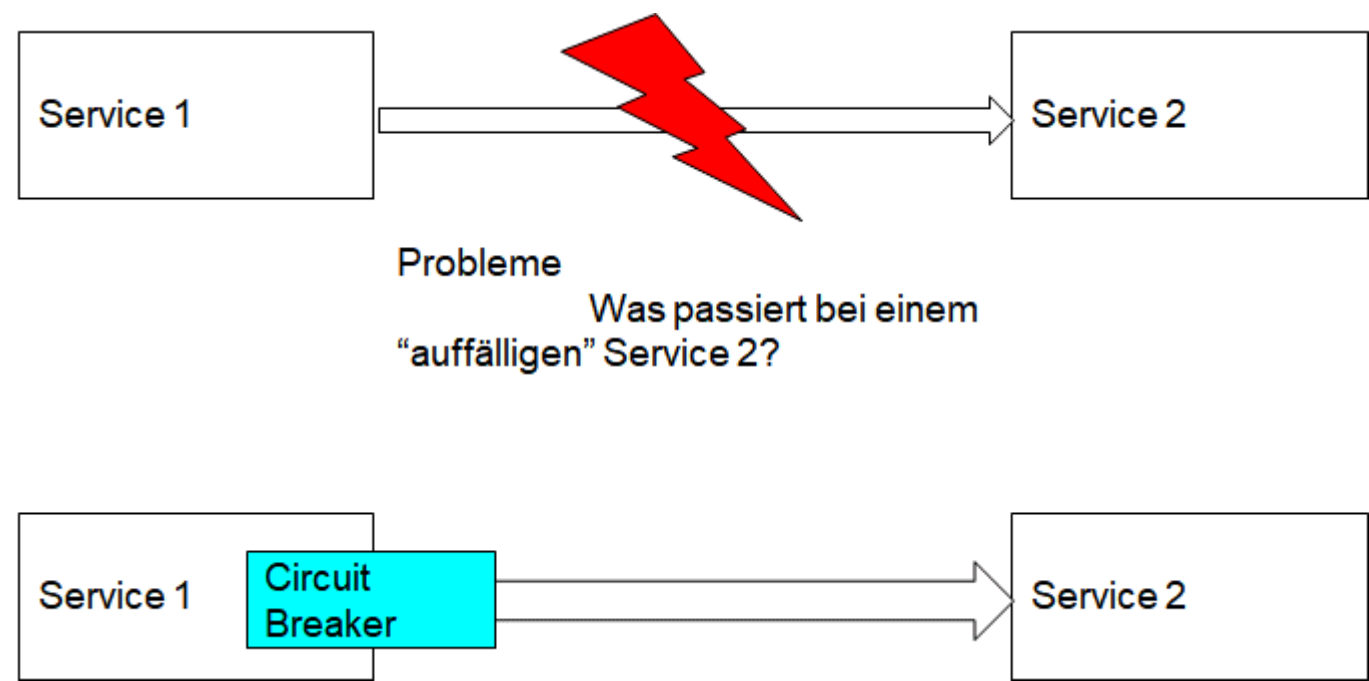


Kafka und Services in der Cloud

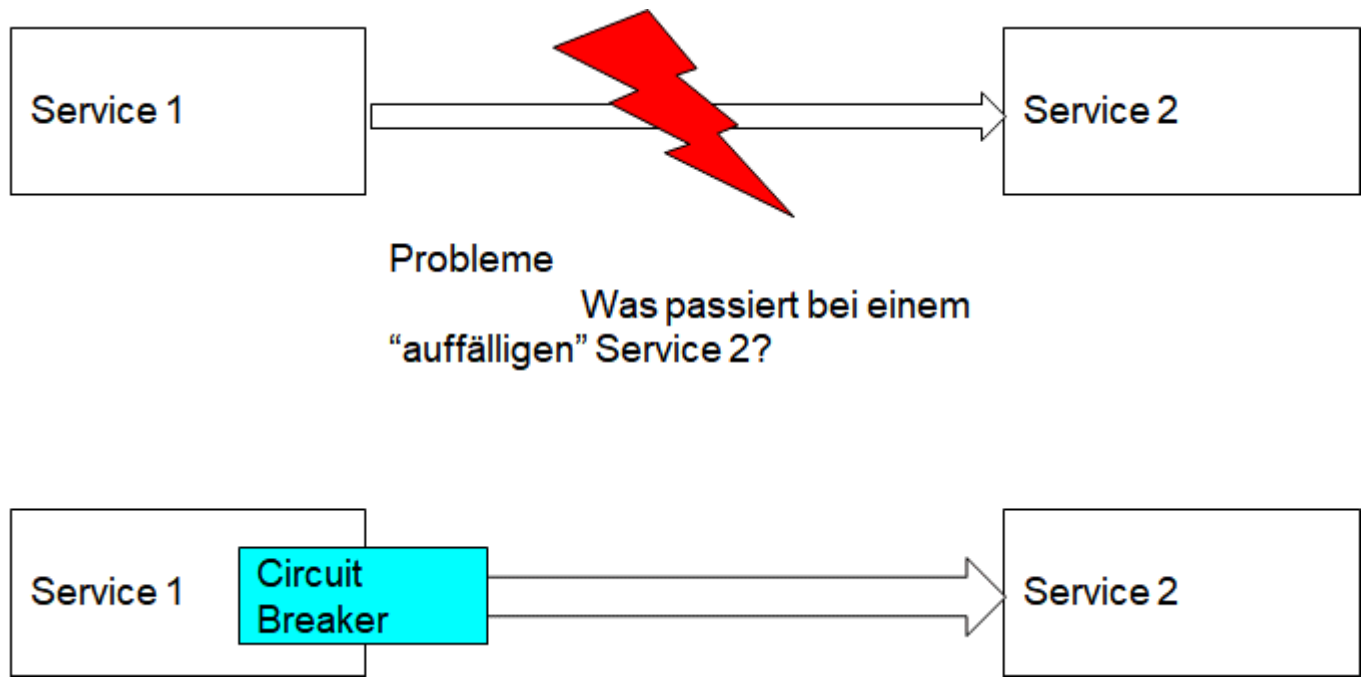
Garantien zur Nachrichtenüber- mittlung in der Cloud



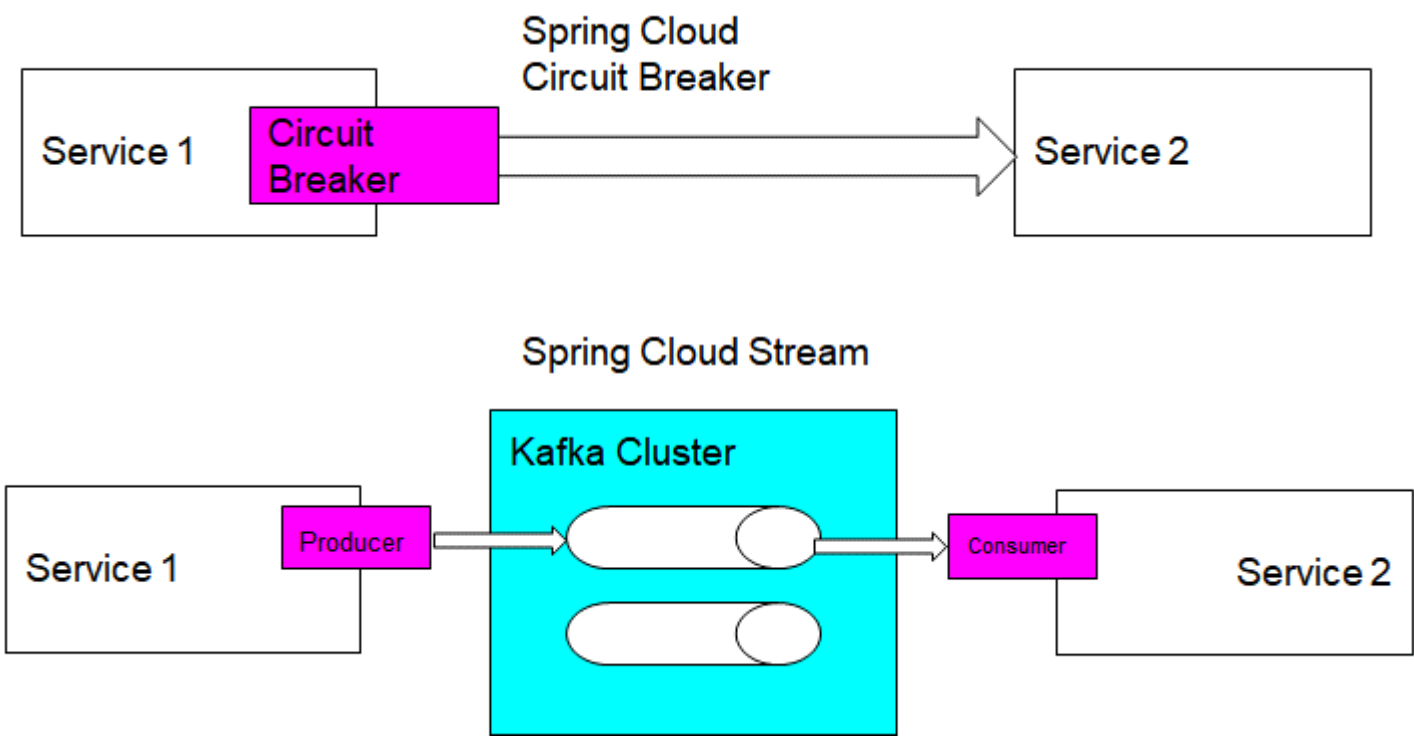
Services in der Cloud: Direkte Kommunikation



Services in der Cloud: Lose Kommunikation



Services in der Spring Cloud





Programmieren mit Kafka



Übersicht



Ein erstes Programm



Serialisierung und
Deserialisierung



Partitioner



Übersicht



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



Ein erstes Programm



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

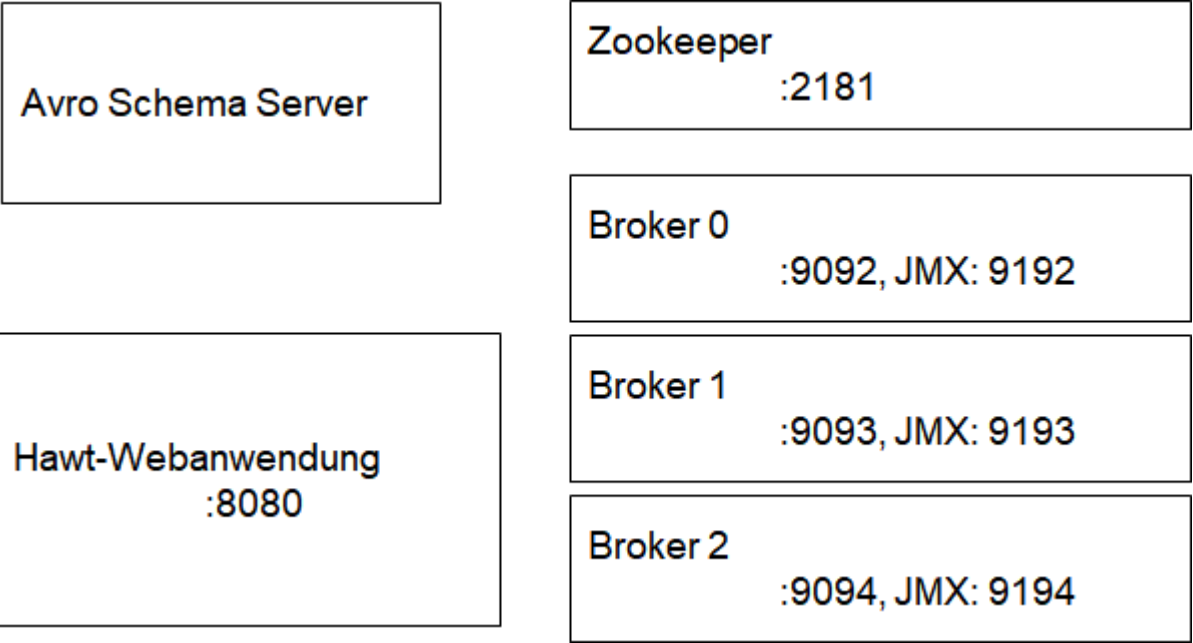


Unterstützte Datentypen

- 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



Administration und Überwachung



JMX



Ein überwachter Cluster



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



Empfehlung: Jolokia

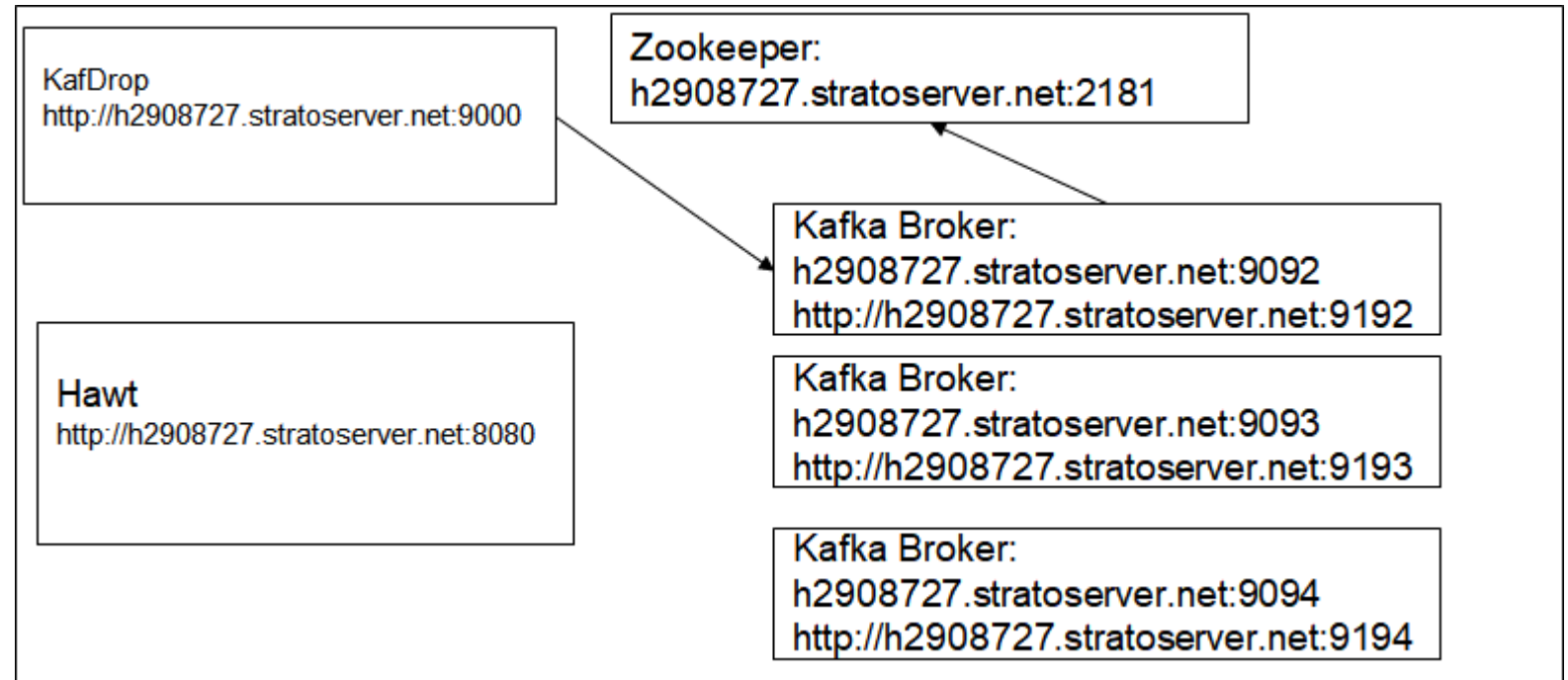
- Benutzen Sie nicht die jconsole und entfernen Sie die Standard-JMX-Konfiguration im Aufruf-Skript
 - Faktisch ein Security Problem
- Ermöglichen Sie den Zugriff auf JMX mit Jolokia
 - <https://java.integrata-cegos.de/jolokia-simples-management-von-java-anwendungen/>
- Hawtio



Ein überwachter Cluster



Server-seitige Kafka- Umgebung mit Überwachung





Kafka Erweiterungen



Kafka Connect

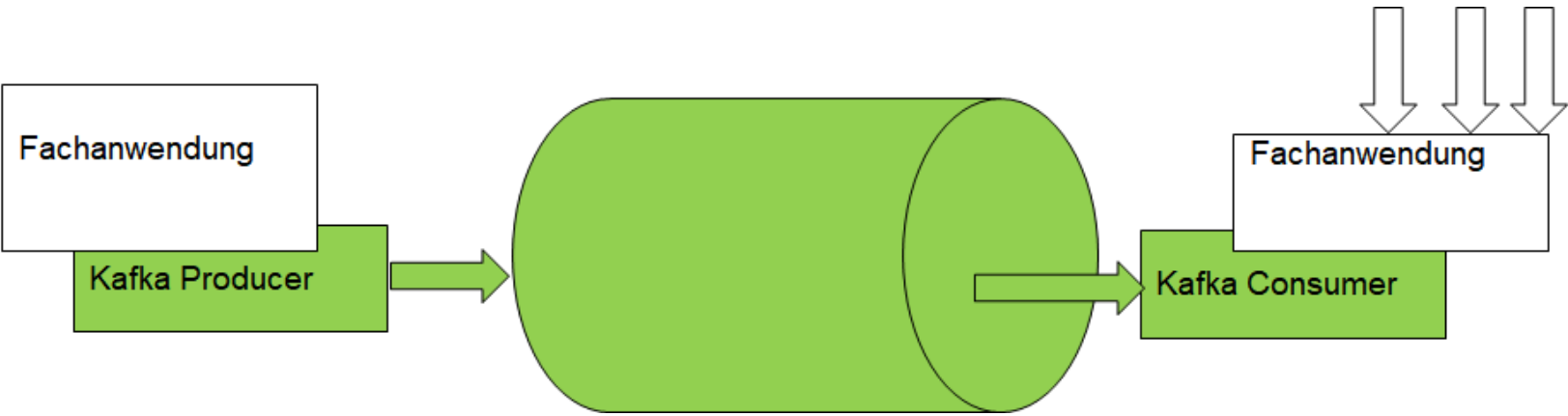


Kafka Stream

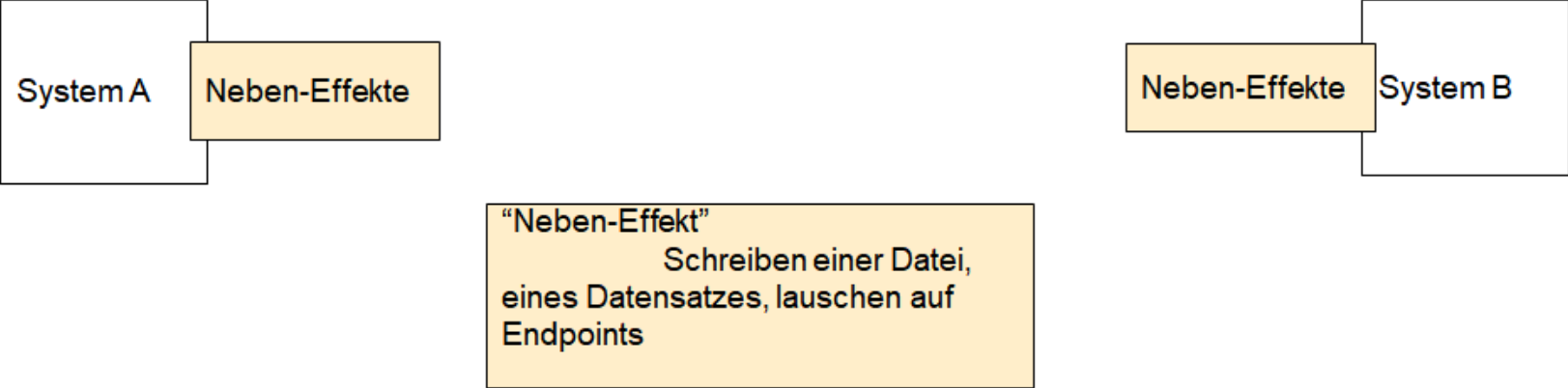


Kafka Connect

Fachanwendungen mit Kafka-Anbindung

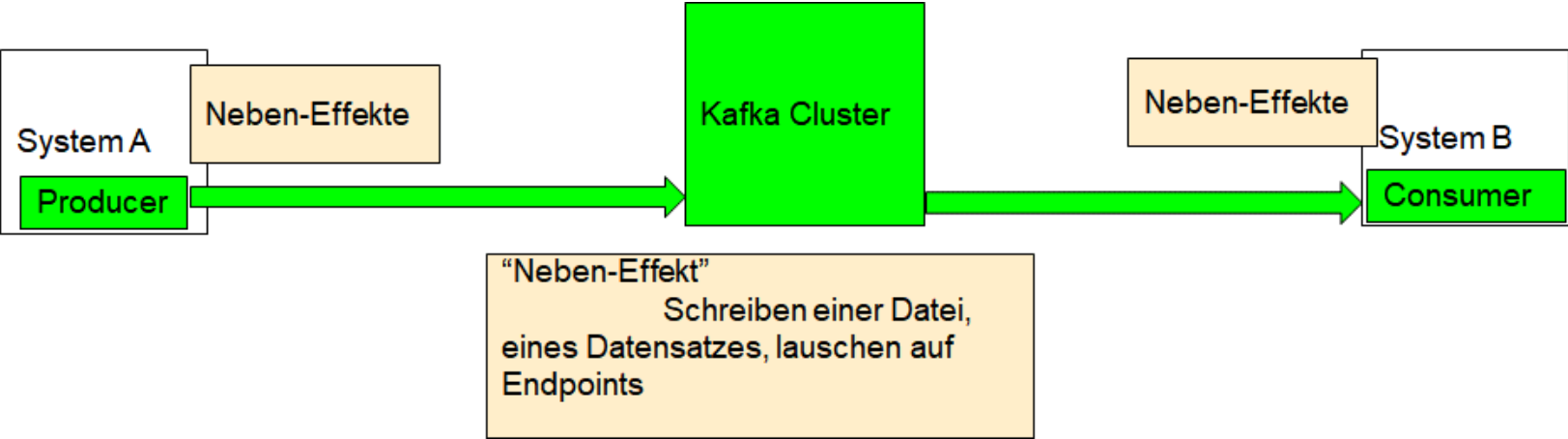


Getrennte Systeme

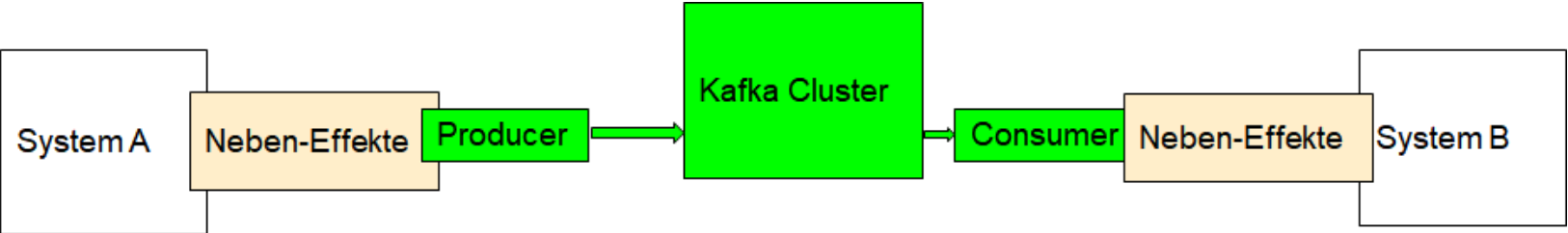




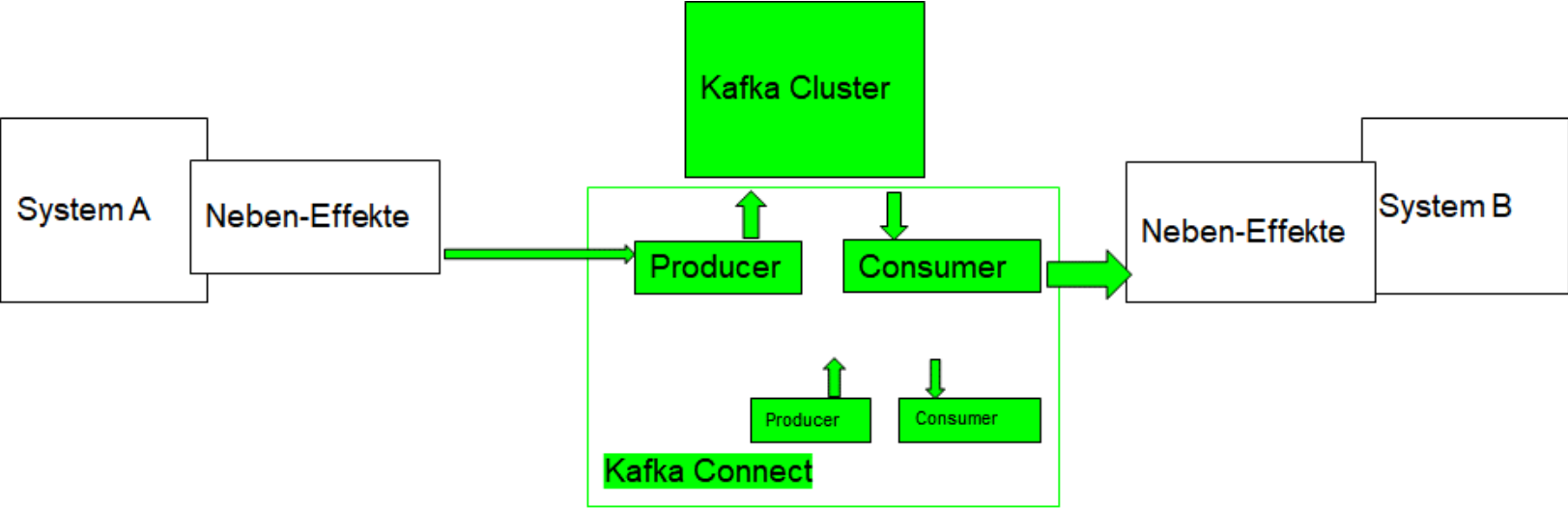
Systemintegration durch Einbau von Kafka



Enterprise Application Integration (EAI)



Systemintegration mit Kafka Connect





Kafka Connect

- Producer und Consumer
 - Separate Anwendungen
 - unabhängig vom Kafka Cluster zu betreiben
 - Anbindung an den Cluster: konfigurativ
 - Standalone
 - mit einem simplen integrierten Kafka Broker
- Start mit
 - `bin\connect-distributed.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'}}`



Alternativen zu Connect

- Apache Camel
 - EAI-Framework
 - Messaging-System ist ein interner Bus
- Spring Integration



Kafka Stream



Anwendungsfall: Extract - Transform - Load

- Spezialfall eines Streaming-Prozesses
- Datenverarbeitungslogik
 - Datenquelle (hat kein definiertes Ende)
 - Verarbeitungs-Pipeline
 - filter
 - transform
 - 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



Alternativen

- Flume oder Spark
 - interne Kommunikation und Datenhaltung kann in Kafka erfolgen