Tutorial on Kafka Streaming Platform

Cloudurable provides Cassandra and Kafka Support on AWS/EC2

Kafka Tutorial

CLOUDURABLE

Kafka growing

Why Kafka?

Kafka adoption is on the rise but why

What is Kafka?

Kafka growth exploding

- Kafka growth exploding
- 1/3 of all Fortune 500 companies
- Top ten travel companies, 7 of ten top banks, 8 of ten top insurance companies, 9 of ten top telecom companies
- LinkedIn, Microsoft and Netflix process 4 comma message a day with Kafka (1,000,000,000,000)
- Real-time streams of data, used to collect big data or to do real time analysis (or both)

Why Kafka is Needed?

- Real time streaming data processed for real time analytics
  - Service calls, track every call, IOT sensors
- Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system
- Kafka is often used instead of JMS, RabbitMQ and AMQP
  - higher throughput, reliability and replication

Why is Kafka needed? 2

- Kafka can works in combination with
  - Flume/Flafka, Spark Streaming, Storm, HBase and Spark for real-time analysis and processing of streaming data
    - Feed your data lakes with data streams
  - Kafka brokers support massive message streams for follow-up analysis in Hadoop or Spark
  - Kafka Streaming (subproject) can be used for real-time analytics

Kafka Use Cases

- Stream Processing
- Website Activity Tracking
- Metrics Collection and Monitoring
- Log Aggregation
- Real time analytics
- Capture and ingest data into Spark / Hadoop
- CRQS, replay, error recovery
- Guaranteed distributed commit log for in-memory computing

Who uses Kafka?

- ***LinkedIn***: Activity data and operational metrics
- ***Twitter***: Uses it as part of Storm – stream processing infrastructure
- ***Square***: Kafka as bus to move all system events to various Square data centers (logs, custom events, metrics, an so on). Outputs to Splunk, Graphite, Esper-like alerting systems
- Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, DataDog, LucidWorks, MailChimp, NetFlix, etc.

Why is Kafka Popular?

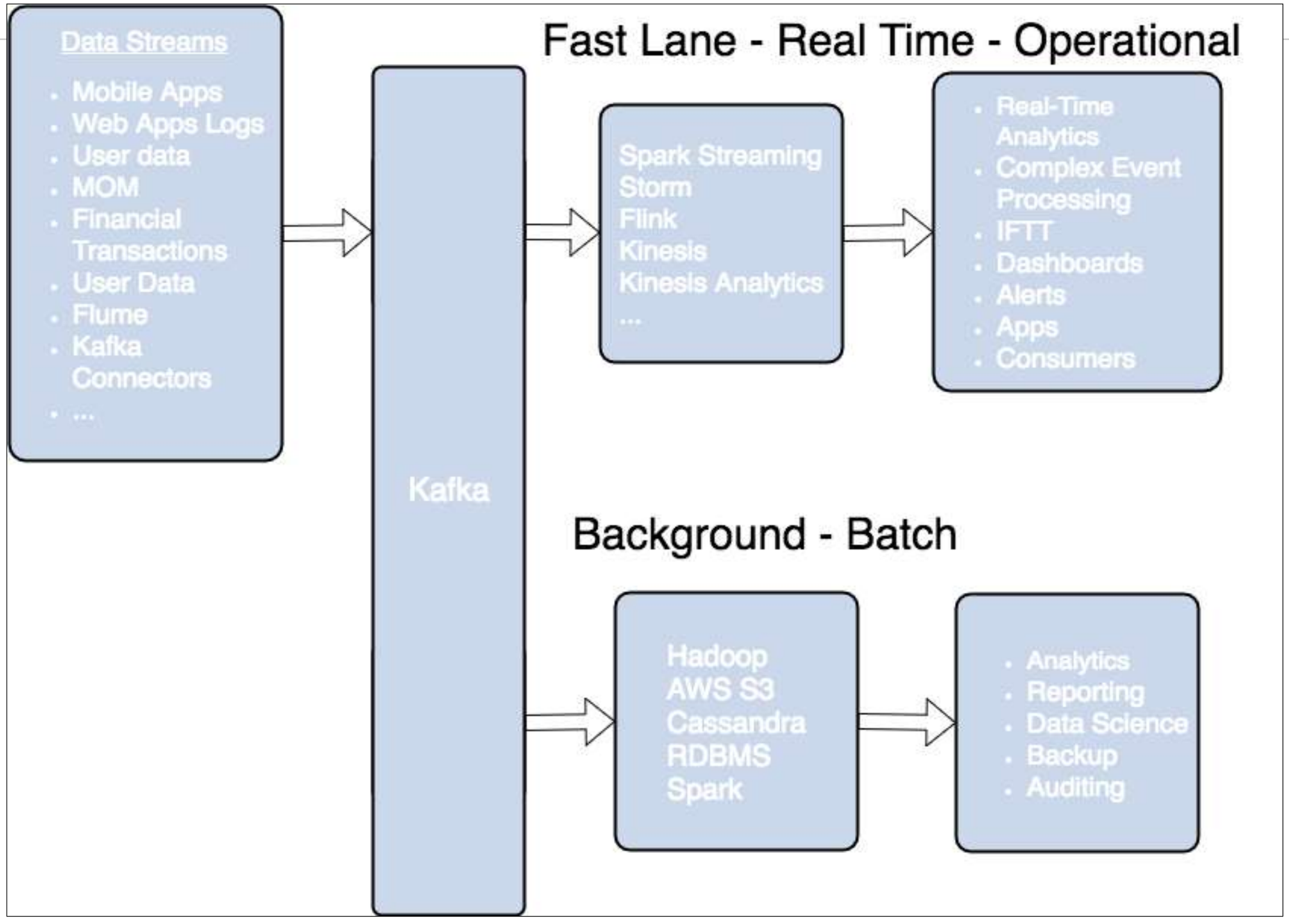
- ***Great performance***
- Operational Simplicity, easy to setup and use, easy to reason
- Stable, Reliable Durability,
- Flexible Publish-subscribe/queue (scales with N-number of consumer groups),
- Robust Replication,
- Producer Tunable Consistency Guarantees,
- Ordering Preserved at shard level (Topic Partition)
- Works well with systems that have data streams to process, aggregate, transform & load into other stores

Most important reason: ***Kafka's great performance***: throughput, latency, obtained through great engineering

Why is Kafka so fast?

- **Zero Copy** - calls the OS kernel direct rather to move data fast
- **Batch Data in Chunks** - Batches data into chunks
  - end to end from Producer to file system to Consumer
  - Provides More efficient data compression. Reduces I/O latency
- **Sequential Disk Writes** - Avoids Random Disk Access
  - writes to immutable commit log. No slow disk seeking. No random I/O operations. Disk accessed in sequential manner
- **Horizontal Scale** - uses 100s to thousands of partitions for a single topic
  - spread out to thousands of servers
  - handle massive load

Kafka Streaming Architecture



Why Kafka Review

- Why is Kafka so fast?
- How fast is Kafka usage growing?
- How is Kafka getting used?
- Where does Kafka fit in the Big Data Architecture?
- How does Kafka relate to real-time analytics?
- Who uses Kafka?

Cassandra / Kafka Support in EC2/AWS

What is Kafka?

Kafka messaging
Kafka Overview

What is Kafka?

- Distributed Streaming Platform
  - Publish and Subscribe to streams of records
  - Fault tolerant storage
    - Replicates Topic Log Partitions to multiple servers
  - Process records as they occur
  - Fast, efficient IO, batching, compression, and more
- Used to decouple data streams

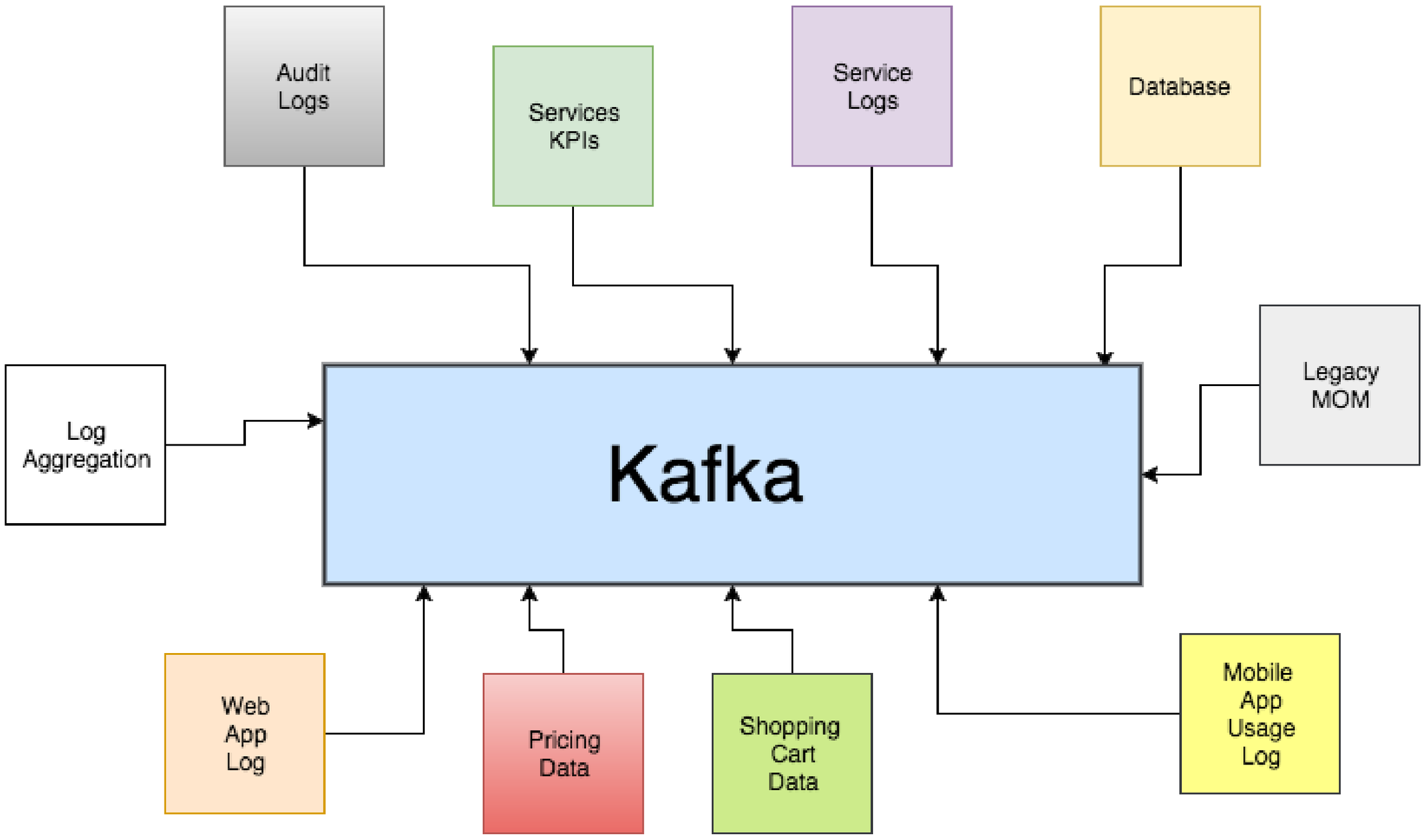
Kafka helps decouple data streams

- Kafka decouple data streams
- producers don’t know about consumers
- Flexible message consumption
  - Kafka broker delegates log partition offset (location) to Consumers (clients)

Kafka messaging allows

- Feeding of high-latency daily or hourly data analysis into Spark, Hadoop, etc.
- Feeding microservices real-time messages
- Sending events to CEP system
- Feeding data to do real-time analytic systems
- Up to date dashboards and summaries
- At same time

Kafka Decoupling Data Streams



Kafka Polyglot clients / Wire protocol

- Kafka communication from clients and servers wire protocol over TCP protocol
- Protocol versioned
- Maintains backwards compatibility
- Many languages supported
- Kafka REST proxy allows easy integration (not part of core)
- Also provides Avro/Schema registry support via Kafka ecosystem (not part of core)

Kafka Usage

- Build real-time streaming applications that react to streams
  - Real-time data analytics
  - Transform, react, aggregate, join real-time data flows
    - Feed events to CEP for complex event processing
    - Feed data lakes
- Build real-time streaming data pipe-lines
  - Enable in-memory microservices (actors, Akka, Vert.x, Qbit, RxJava)

Kafka Use Cases

- Metrics / KPIs gathering
  - Aggregate statistics from many sources
- Event Sourcing
  - Used with microservices (in-memory) and actor systems
- Commit Log
  - External commit log for distributed systems. Replicated data between nodes, re-sync for nodes to restore state
- Real-time data analytics, Stream Processing, Log Aggregation, Messaging, Click-stream tracking, Audit trail, etc.

Kafka Record Retention

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based - configurable based on size
    - Compaction - keeps latest record
- Retention policy of three days or two weeks or a month
- It is available for consumption until discarded by time, size or compaction
- Consumption speed not impacted by size

Kafka scalable message storage

- Kafka acts as a good storage system for records/messages
- Records written to Kafka topics are persisted to disk and replicated to other servers for fault-tolerance
- Kafka Producers can wait on acknowledgment
  - Write not complete until fully replicated
- Kafka disk structures scales well
  - Writing in large streaming batches is fast
- Clients/Consumers can control read position (offset)
  - Kafka acts like high-speed file system for commit log storage, replication

Kafka Review

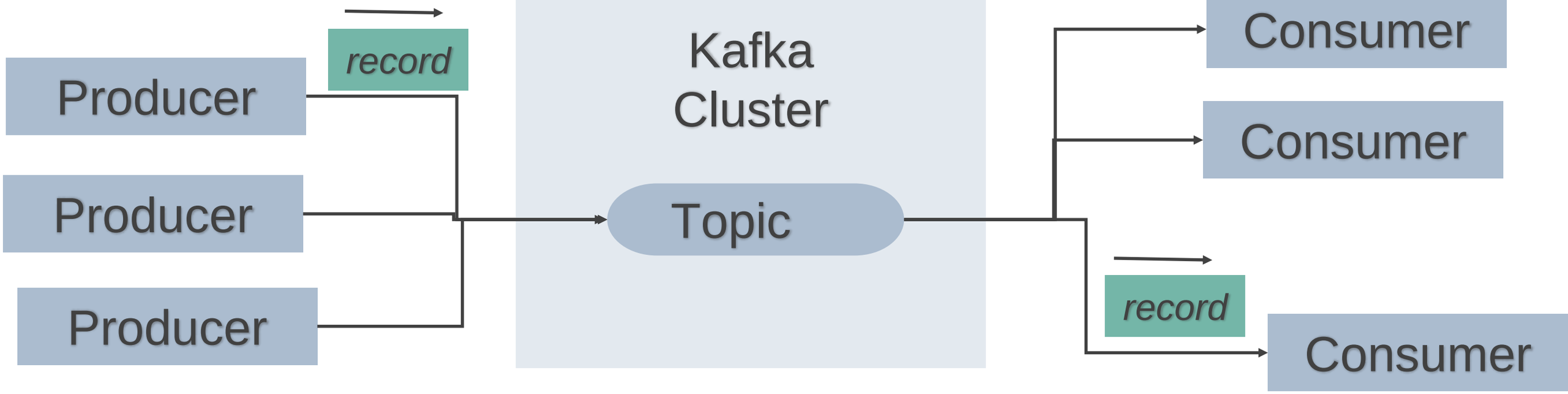
- How does Kafka decouple streams of data?
- What are some use cases for Kafka where you work?
- What are some common use cases for Kafka?
- How is Kafka like a distributed message storage system?
- How does Kafka know when to delete old messages?
- Which programming languages does Kafka support?

Kafka Architecture

Kafka Fundamentals

- ***Records*** have a ***key (optional)***, ***value*** and ***timestamp; Immutable***
- ***Topic*** a stream of records (“/orders”, “/user-signups”), feed name
  - ***Log*** topic storage on disk
  - ***Partition*** / Segments (parts of Topic Log)
- ***Producer*** API to produce a streams or records
- ***Consumer*** API to consume a stream of records
- ***Broker***: Kafka server that runs in a Kafka Cluster. Brokers form a cluster. Cluster consists on many Kafka Brokers on many servers.
- ***ZooKeeper***: Does coordination of brokers/cluster topology. Consistent file system for configuration information and leadership election for Broker Topic Partition Leaders

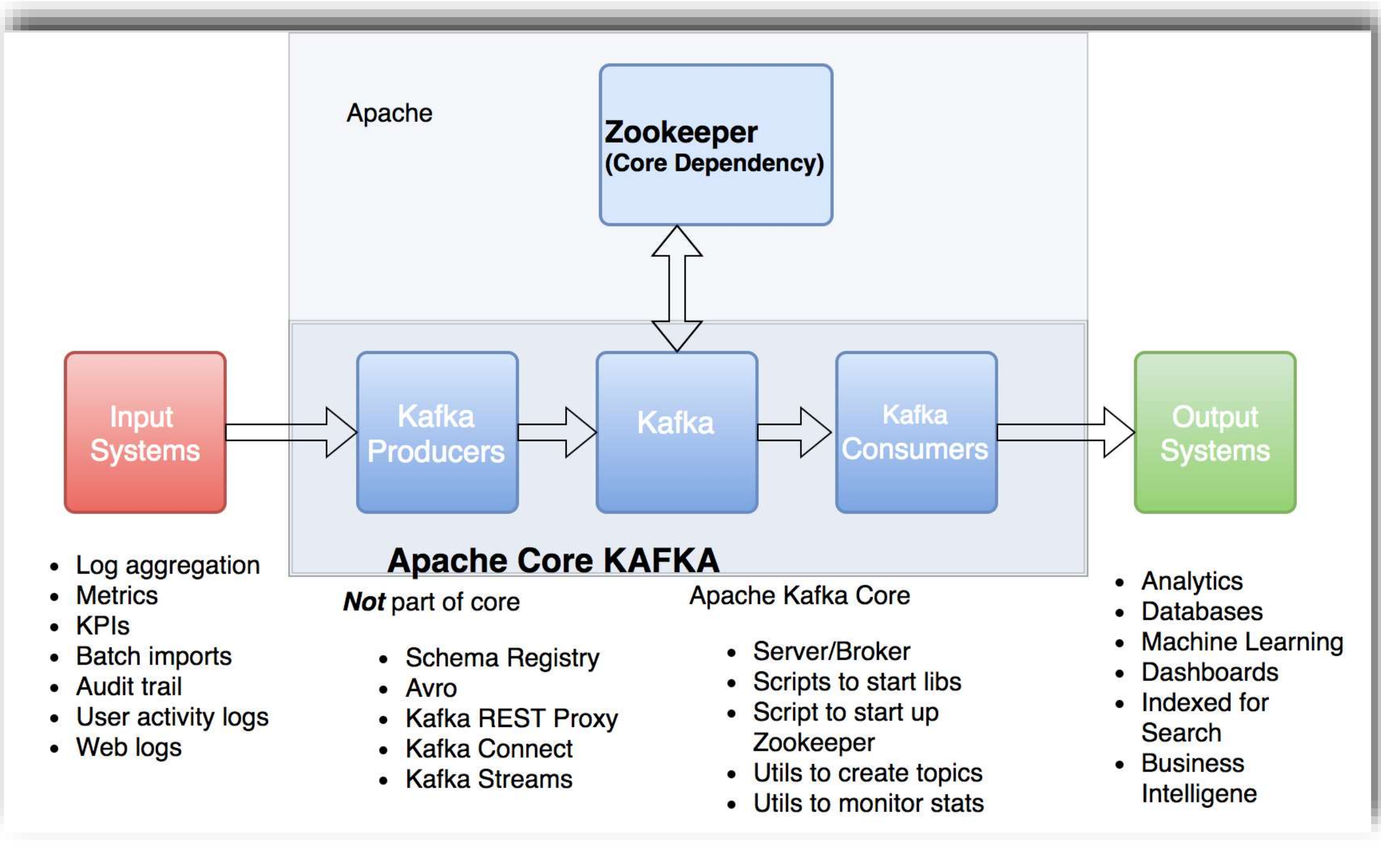
Kafka: Topics, Producers, and Consumers



Apache Kafka - Core Kafka

- Kafka gets conflated with Kafka ecosystem
- Apache Core Kafka consists of Kafka Broker, startup scripts for ZooKeeper, and client APIs for Kafka
- Apache Core Kafka does ***not*** include
  - Confluent Schema Registry (not an Apache project)
  - Kafka REST Proxy (not an Apache project)
  - Kafka Connect (not an Apache project)
  - Kafka Streams (not an Apache project)

Apache Kafka



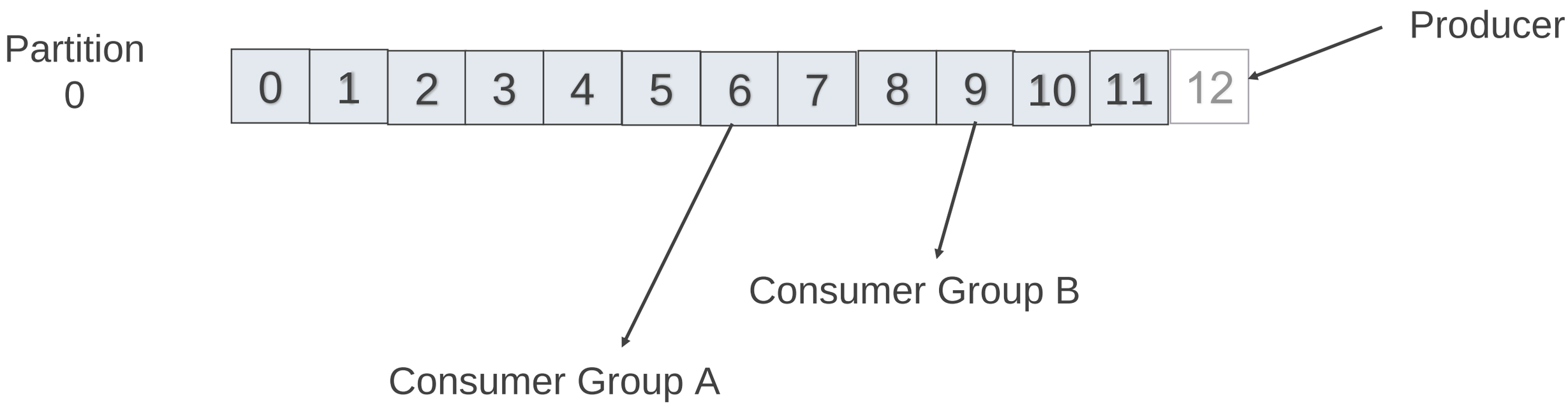
Kafka needs Zookeeper

- Zookeeper helps with leadership election of Kafka Broker and Topic Partition pairs
- Zookeeper manages service discovery for Kafka Brokers that form the cluster
- Zookeeper sends changes to Kafka
  - New Broker join, Broker died, etc.
  - Topic removed, Topic added, etc.
- Zookeeper provides in-sync view of Kafka Cluster configuration

Kafka Producer/Consumer Details

- ***Producers*** write to and ***Consumers*** read from ***Topic(s)***
- ***Topic*** associated with a log which is data structure on disk
- ***Producer***(s) append ***Records*** at end of Topic log
- Topic ***Log*** consist of Partitions -
  - Spread to multiple files on multiple nodes
- ***Consumers*** read from Kafka at their own cadence
  - Each Consumer (Consumer Group) tracks offset from where they left off reading
- ***Partitions*** can be distributed on different machines in a cluster
  - high performance with horizontal scalability and failover with replication

Kafka Topic Partition, Consumers, Producers



Consumer groups remember offset where they left off.
Consumers groups each have their own offset.

Producer writing to offset 12 of Partition 0 while…
- Consumer Group A is reading from offset 6.
- Consumer Group B is reading from offset 9.

Kafka Scale and Speed

- How can Kafka scale if multiple producers and consumers read/write to same Kafka Topic log?
- Writes fast: Sequential writes to filesystem are ***fast*** (700 MB or more a second)
- Scales writes and reads by ***sharding:***
  - Topic logs into ***Partitions*** (parts of a Topic log)
  - Topics logs can be split into multiple Partitions ***different machines/different disks***
  - Multiple Producers can write to different Partitions of the same Topic
  - Multiple Consumers Groups can read from different partitions efficiently

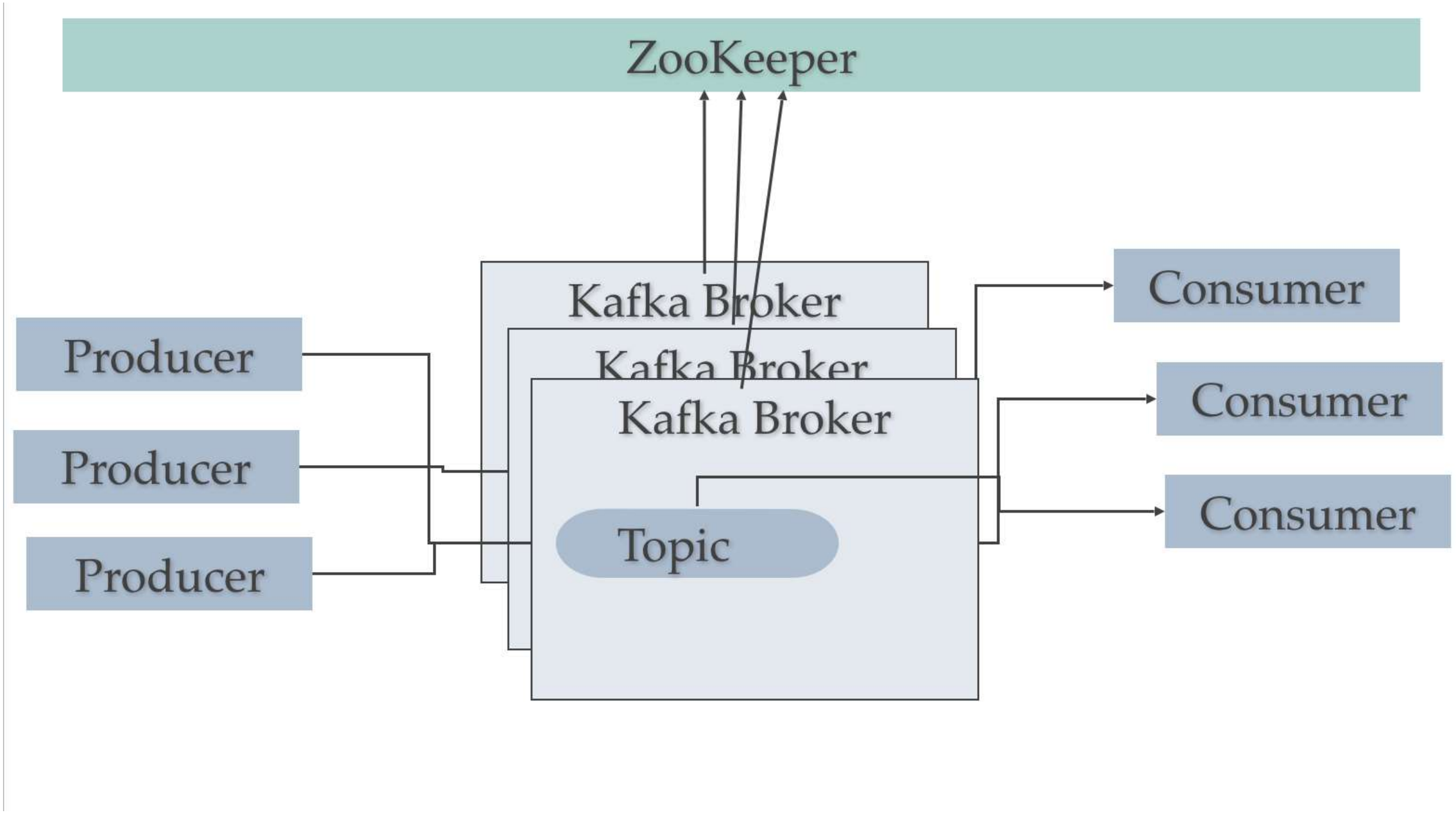
Kafka Brokers

- Kafka Cluster is made up of multiple Kafka Brokers
- Each Broker has an ID (number)
- Brokers contain topic log partitions
- Connecting to one broker bootstraps client to entire cluster
- Start with at least three brokers, cluster can have, 10, 100, 1000 brokers if needed

Kafka Cluster, Failover, ISRs

- Topic ***Partitions*** can be ***replicated***
  - across ***multiple nodes*** for failover
- Topic should have a replication factor greater than 1
  - (2, or 3)
- ***Failover***
  - if one Kafka Broker goes down then Kafka Broker with ISR (in-sync replica) can serve data

ZooKeeper does coordination for Kafka Cluster



Failover vs. Disaster Recovery

- Replication of Kafka Topic Log partitions allows for failure of a rack or AWS availability zone
  - You need a replication factor of at least 3
- ***Kafka Replication*** is for ***Failover***
- ***Mirror Maker*** is used for ***Disaster Recovery***
- Mirror Maker replicates a Kafka cluster to another data-center or AWS region
  - Called mirroring since replication happens within a cluster

Kafka Review

- How does Kafka decouple streams of data?
- What are some use cases for Kafka where you work?
- What are some common use cases for Kafka?
- What is a Topic?
- What is a Broker?
- What is a Partition? Offset?
- Can Kafka run without Zookeeper?
- How do implement failover in Kafka?
- How do you implement disaster recovery in Kafka?

Kafka versus

Kafka vs JMS, SQS, RabbitMQ Messaging

- Is Kafka a Queue or a Pub/Sub/Topic?
  - Yes
- Kafka is like a Queue per consumer group
  - Kafka is a queue system per consumer in consumer group so load balancing like JMS, RabbitMQ queue
- Kafka is like Topics in JMS, RabbitMQ, MOM
  - Topic/pub/sub by offering Consumer Groups which act like subscriptions
  - Broadcast to multiple consumer groups
- MOM = JMS, ActiveMQ, RabbitMQ, IBM MQ Series, Tibco, etc.

Kafka vs MOM

- By design, Kafka is better suited for scale than traditional MOM systems due to partition topic log
  - Load divided among Consumers for read by partition
  - Handles parallel consumers better than traditional MOM
- Also by moving location (partition offset) in log to client/consumer side of equation instead of the broker, less tracking required by Broker and more flexible consumers
- Kafka written with mechanical sympathy, modern hardware, cloud in mind
  - Disks are faster
  - Servers have tons of system memory
  - Easier to spin up servers for scale out

Kinesis and Kafka are similar

- Kinesis Streams is like Kafka Core
- Kinesis Analytics is like Kafka Streams
- Kinesis Shard is like Kafka Partition
- Similar and get used in similar use cases
- In Kinesis, data is stored in shards. In Kafka, data is stored in partitions
- Kinesis Analytics allows you to perform SQL like queries on data streams
- Kafka Streaming allows you to perform functional aggregations and mutations
- Kafka integrates well with Spark and Flink which allows SQL like queries on streams

Kafka vs. Kinesis

- Data is stored in Kinesis for default 24 hours, and you can increase that up to 7 days.
- Kafka records default stored for 7 days
  - can increase until you run out of disk space.
  - Decide by the size of data or by date.
  - Can use compaction with Kafka so it only stores the latest timestamp per key per record in the log
- With Kinesis data can be analyzed by lambda before it gets sent to S3 or RedShift
- With Kinesis you pay for use, by buying read and write units.
- Kafka is more flexible than Kinesis but you have to manage your own clusters, and requires some dedicated DevOps resources to keep it going
- Kinesis is sold as a service and does not require a DevOps team to keep it going (can be more expensive and less flexible, but much easier to setup and run)

Kafka Topics

Kafka Topics Architecture

Replication
Failover
Parallel processing
Kafka Topic Architecture

Topics, Logs, Partitions

- Kafka ***Topic*** is a stream of records
- ***Topics*** stored in log
- ***Log*** broken up into ***partitions*** and ***segments***
- ***Topic*** is a category or stream name or feed
- Topics are pub/sub
  - Can have zero or many subscribers - consumer groups
- ***Topics*** are broken up and spread by partitions for speed and size

Topic Partitions

- ***Topics*** are broken up into ***partitions***
- ***Partitions*** decided usually by key of record
  - Key of record determines which partition
- ***Partitions*** are used to scale Kafka across many servers
  - Record sent to correct partition by key
- ***Partitions*** are used to facilitate parallel consumers
  - Records are consumed in parallel up to the number of partitions
- Order guaranteed per partition
- Partitions can be ***replicated*** to multiple brokers

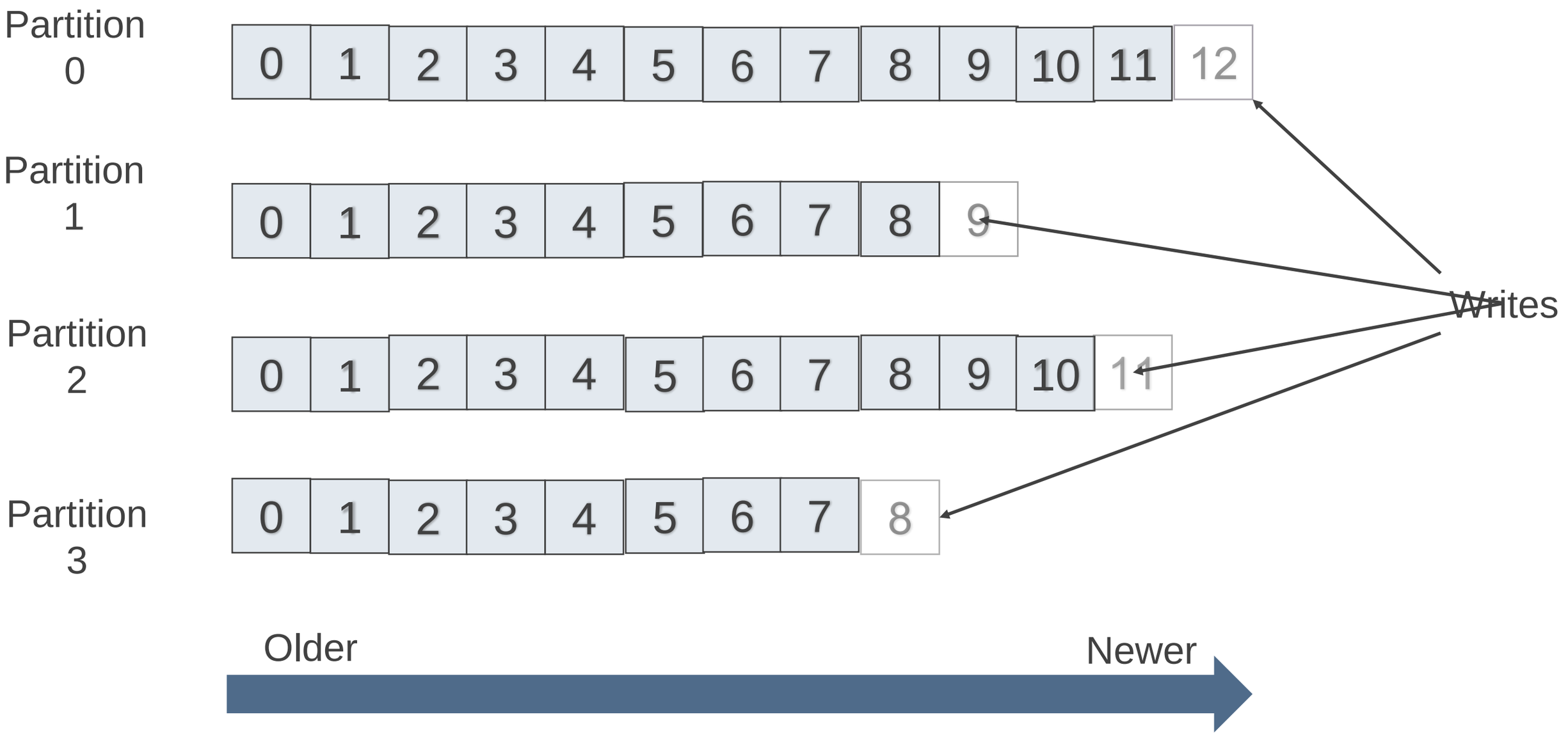
Topic Partition Log

- ***Order*** is maintained only in a single ***partition***
  - ***Partition*** is ordered, immutable sequence of records that is continually appended to—a structured commit ***log***
- ***Records*** in partitions are assigned ***sequential id*** number called the *offset*
- ***Offset*** identifies each record within the partition
- ***Topic Partitions*** allow Kafka log to scale beyond a size that will fit on a single server
  - Topic partition must fit on servers that host it
  - topic can span many partitions hosted on many servers

Topic Parallelism and Consumers

- Topic Partitions are unit of ***parallelism*** - a partition can only be used by one consumer in group at a time
- Consumers can run in their own process or their own thread
- If a consumer stops, Kafka spreads partitions across remaining consumer in group
- #of Consumers you can run per Consumer Group limited by #of Partitions
- Consumers getting assigned partition aids in efficient message consumption tracking

Kafka Topic Partitions Layout



Replication: Kafka Partition Distribution

- Each partition has ***leader server*** and zero or more ***follower servers***
  - ***Leader*** handles all read and write requests for partition
  - ***Followers*** replicate leader, and take over if leader dies
  - Used for parallel consumer handling within a group
- Partitions of log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of partitions
- Each partition can be replicated across a configurable number of Kafka servers - Used for fault tolerance

Replication: Kafka Partition Leader

- One node/partition's replicas is chosen as ***leader***
- Leader handles all reads and writes of Records for partition
- Writes to partition are ***replicated*** to ***followers*** (node/partition pair)
- An ***follower*** that is ***in-sync*** is called an ***ISR (in-sync replica)***
- If a partition leader fails, one ISR is chosen as new leader

Kafka Replication to Partition 0

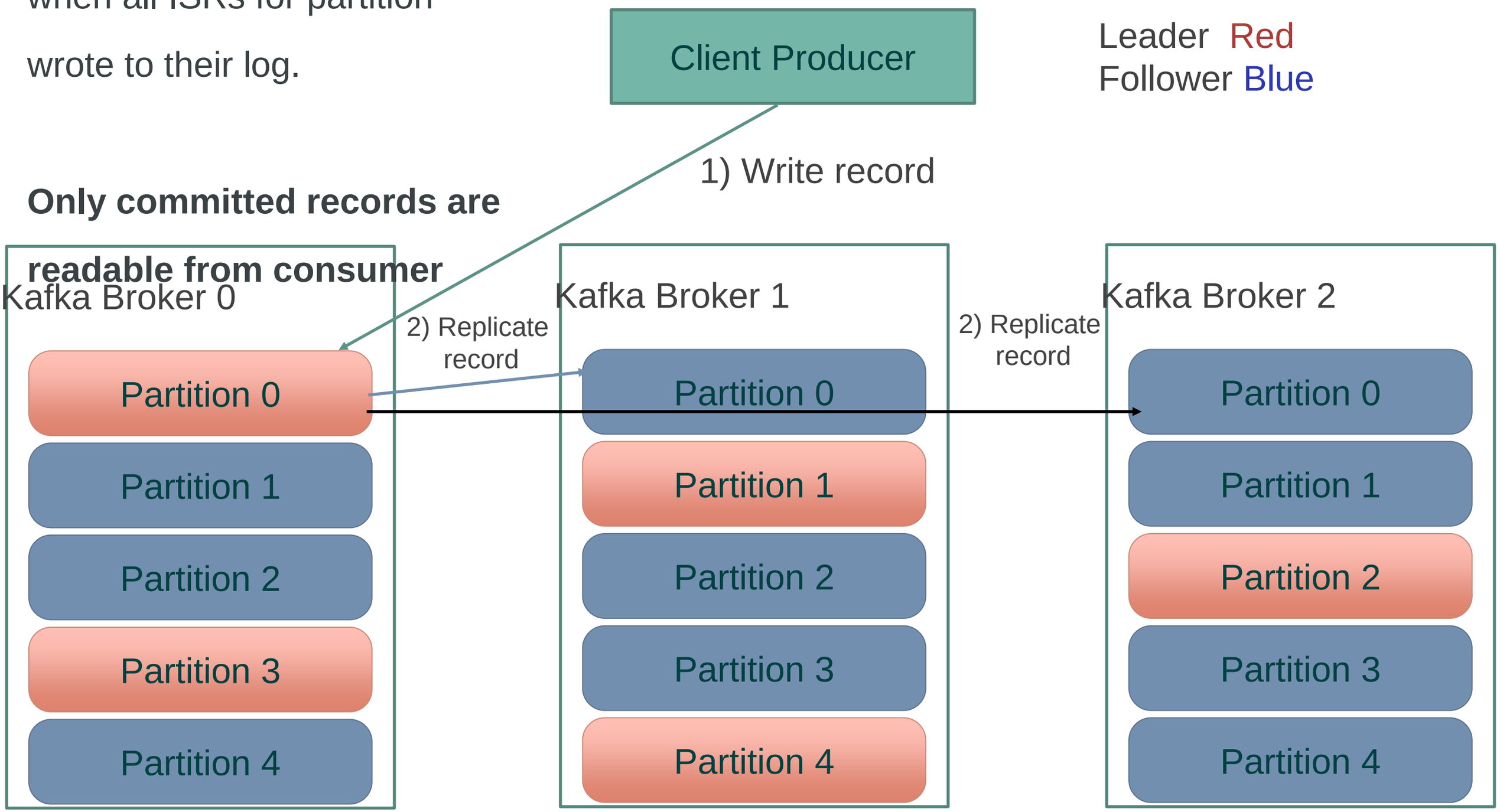
Record is considered "committed"

when all ISRs for partition

wrote to their log.

Only committed records are readable from consumer

Leader Red
Follower Blue



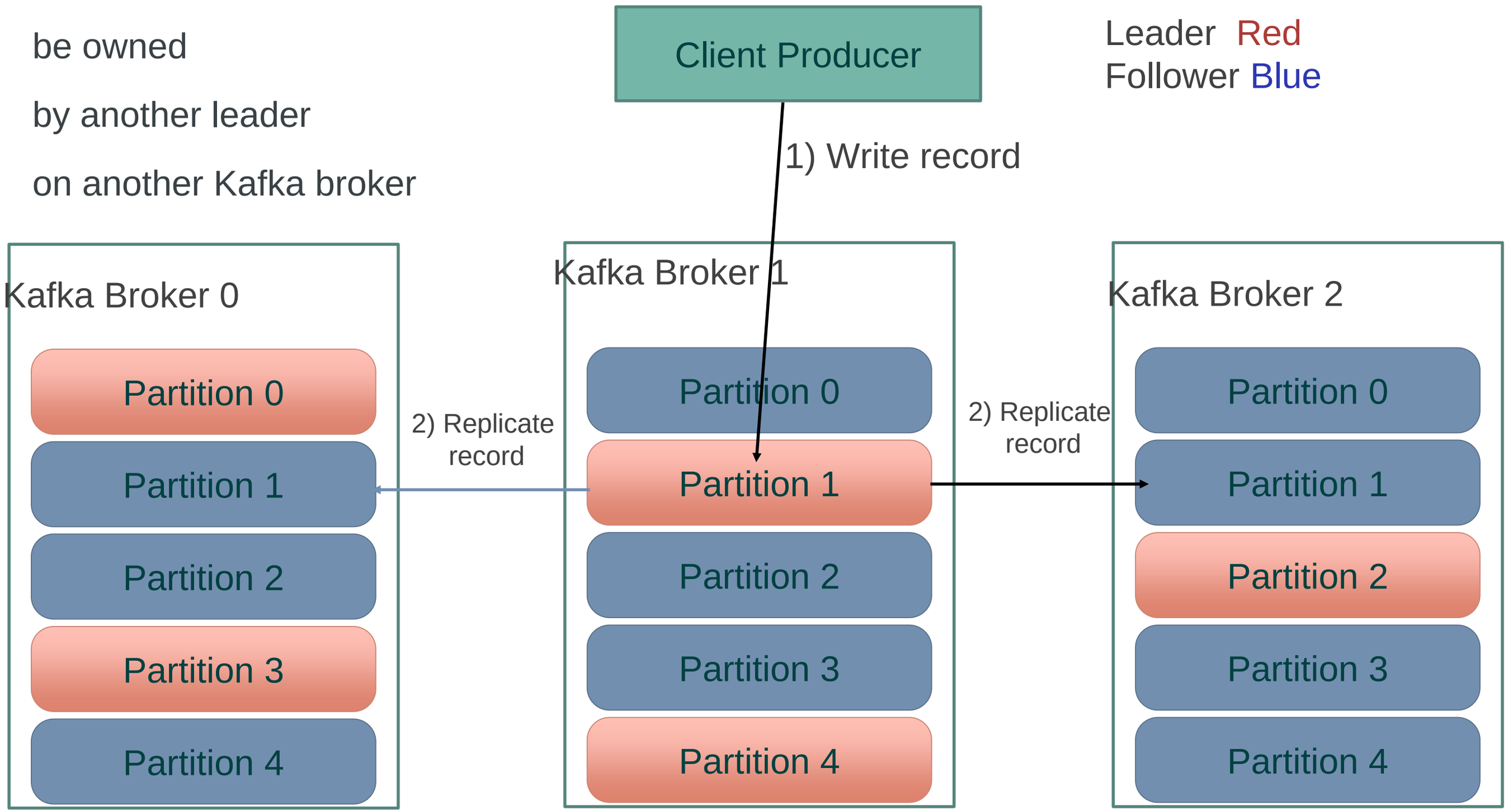
Kafka Replication to Partitions 1

Another partition can

be owned

by another leader

on another Kafka broker



Topic Review

- What is an ISR?
- How does Kafka scale Consumers?
- What are leaders? followers?
- How does Kafka perform failover for Consumers?
- How does Kafka perform failover for Brokers?

Kafka Producers

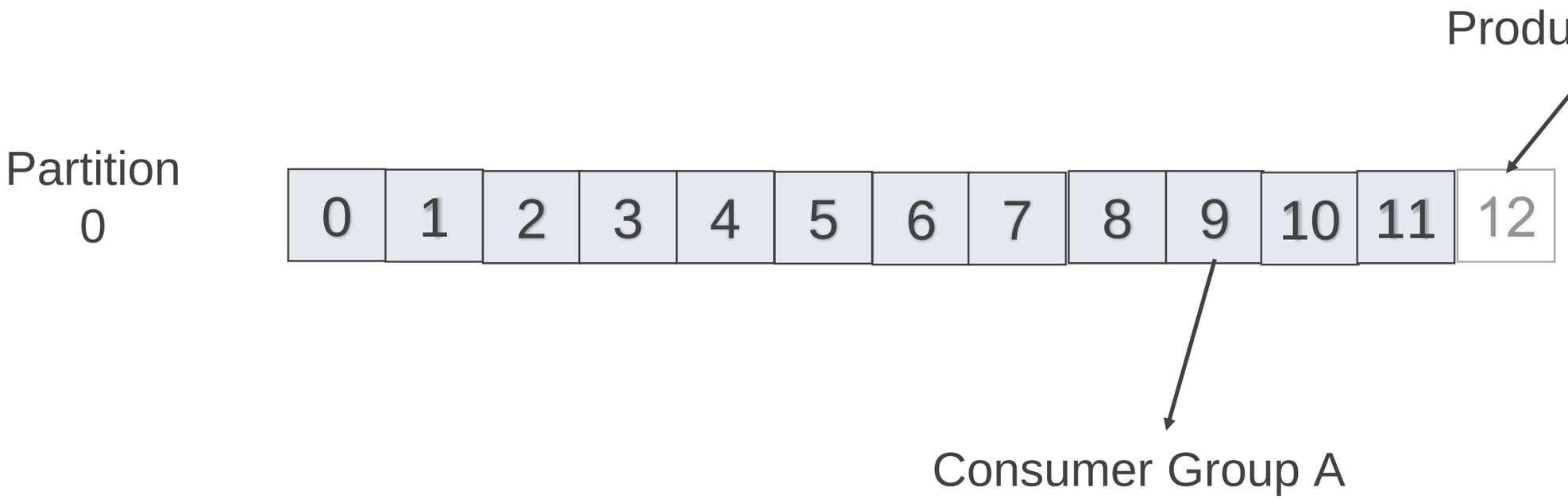
Kafka Producers

Partition selection
Durability
Kafka Producer Architecture

Kafka Producers

- ***Producers*** send records to topics
- ***Producer*** picks which partition to send record to per topic
  - Can be done in a ***round-robin***
  - Can be based on priority
  - Typically based on ***key*** of ***record***
  - Kafka ***default partitioner*** for Java uses hash of keys to choose partitions, or a round-robin strategy if no key
- Important: *Producer picks partition*

Kafka Producers and Consumers



Producers are writing at Offset 12

Consumer Group A is Reading from Offset 9.

Kafka Producers

- ***Producers*** write at their own cadence so order of Records cannot be guaranteed across partitions
- ***Producer*** configures consistency level (ack=0, ack=all, ack=1)
- ***Producers*** pick the ***partition*** such that Record/messages goes to a given same partition based on the data
  - Example have all the events of a certain 'employeeId' go to same partition
  - If order within a partition is not needed, a 'Round Robin' partition strategy can be used so Records are evenly distributed across partitions.

Producer Review

- Can Producers occasionally write faster than consumers?
- What is the default partition strategy for Producers without using a key?
- What is the default partition strategy for Producers using a key?
- What picks which partition a record is sent to?

Kafka Consumers

Load balancing consumers
Failover for consumers

Offset management per consumer group

Kafka Consumer Architecture

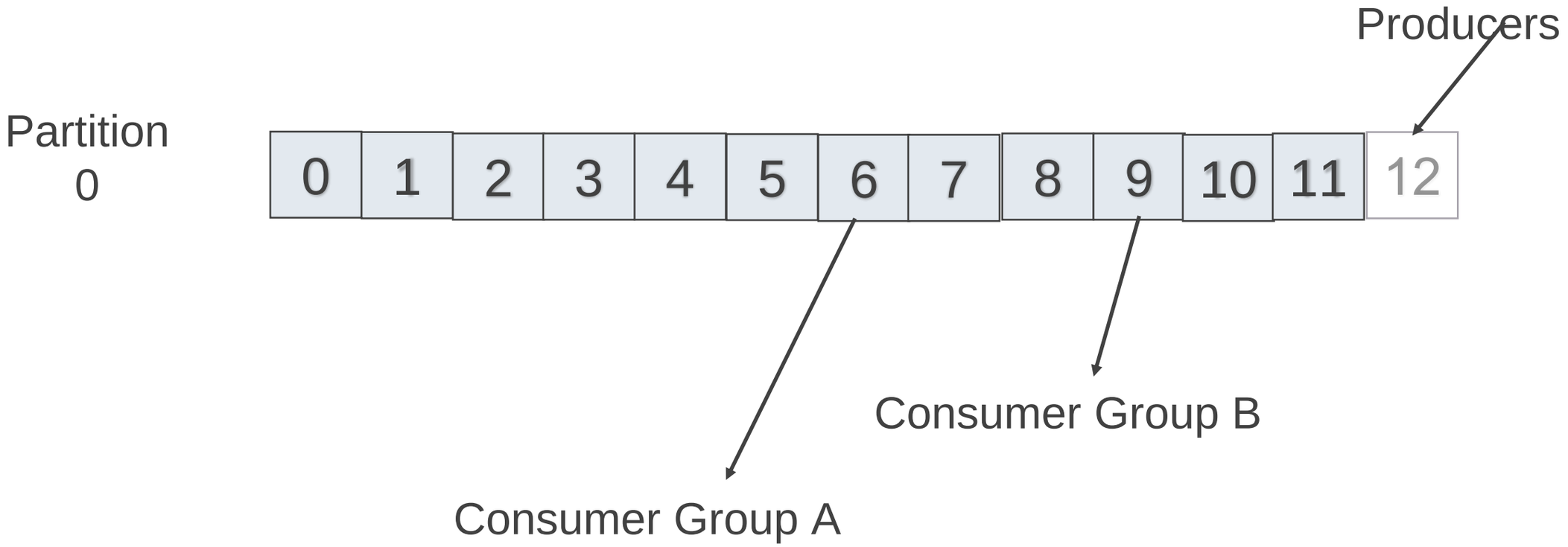
Kafka Consumer Groups

- Consumers are grouped into a ***Consumer Group***
  - ***Consumer group*** has a unique id
  - Each ***consumer group*** is a subscriber
  - Each ***consumer group*** maintains its own offset
    - Multiple subscribers = multiple consumer groups
    - Each has different function: one might delivering records to microservices while another is streaming records to Hadoop
- ***A Record*** is delivered to one ***Consumer*** in a ***Consumer Group***
- Each consumer in consumer groups takes records and only one consumer in group gets same record
- Consumers in Consumer Group ***load balance*** record consumption

Kafka Consumer Load Share

- Kafka ***Consumer*** consumption ***divides*** partitions over consumers in a Consumer Group
- Each ***Consumer*** is exclusive consumer of a ***"fair share" of partitions***
- This is Load Balancing
- ***Consumer*** membership in ***Consumer Group*** is handled by the Kafka protocol dynamically
- If new Consumers ***join*** Consumer group, it gets a share of partitions
- If Consumer ***dies***, its partitions are split among remaining live Consumers in Consumer Group

Kafka Consumer Groups



Consumers remember offset where they left off.

Consumers groups each have their own offset per partition.

Kafka Consumer Groups Processing

- How does Kafka divide up topic so multiple ***Consumers*** in a ***Consumer Group*** can process a topic?
- You group consumers into consumers group with a group id
- ***Consumers*** with same id belong in same ***Consumer Group***
- One ***Kafka broker*** becomes ***group coordinator*** for Consumer Group
  - assigns partitions when new members arrive (older clients would talk direct to ZooKeeper now broker does coordination)
  - or reassign partitions when group members leave or topic changes (config / meta-data change
- When ***Consumer group*** is created, offset set according to reset policy of topic

Kafka Consumer Failover

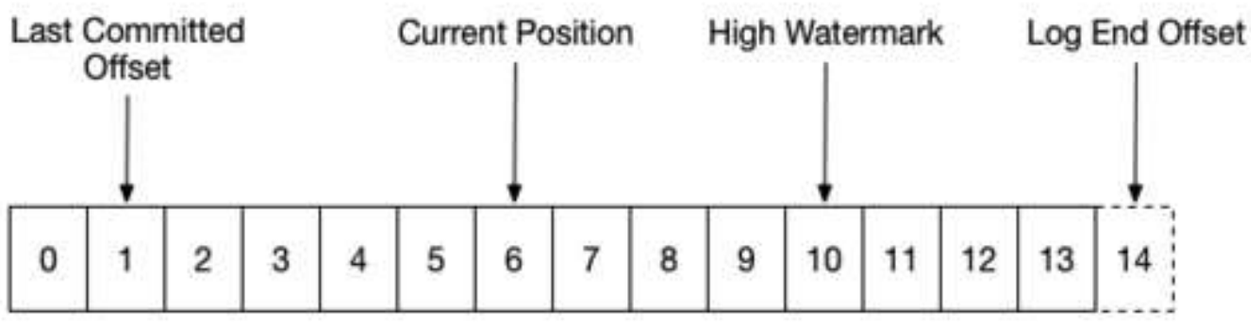
- ***Consumers*** notify broker when it successfully processed a record
  - advances offset
- If ***Consumer*** fails before sending commit offset to Kafka broker,
  - different ***Consumer*** can continue from the last committed offset
  - some Kafka records could be reprocessed
  - ***at least once behavior***
  - ***messages should be idempotent***

Kafka Consumer Offsets and Recovery

- Kafka stores offsets in topic called “__consumer_offset”
  - Uses Topic Log Compaction
- When a consumer has processed data, it should commit offsets
- If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in “__consumer_offset”

Kafka Consumer: What can be consumed?

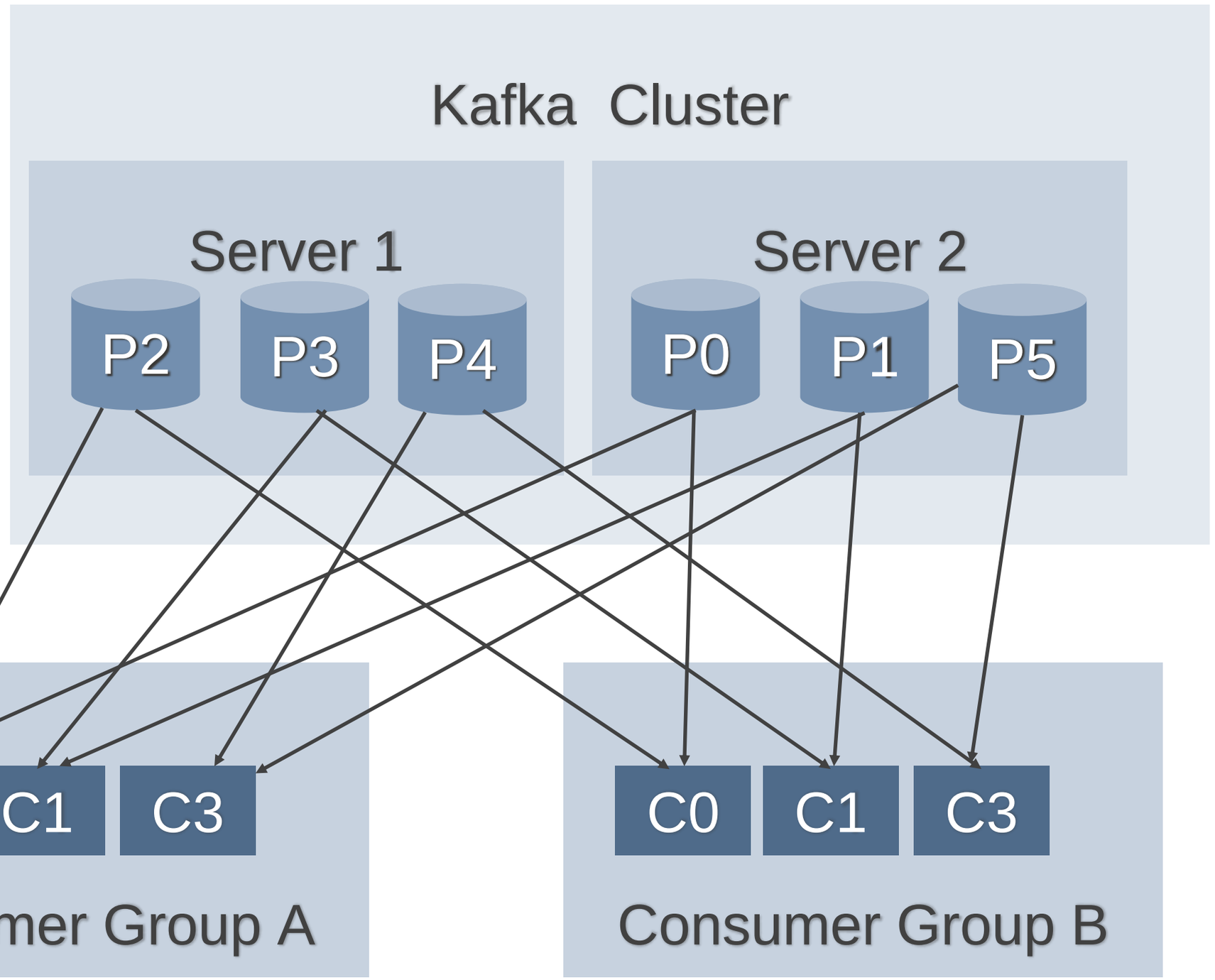
- ***"Log end offset"*** is offset of last record written to log partition and where ***Producers*** write to next
- ***"High watermark"*** is offset of last record successfully replicated to all partitions followers
- ***Consumer*** only reads up to “high watermark”. ***Consumer can’t read un-replicated data***



Consumer to Partition Cardinality

- Only a single ***Consumer*** from the same ***Consumer Group*** can access a single ***Partition***
- If ***Consumer Group*** count ***exceeds*** Partition count:
  - Extra Consumers remain idle; can be used for failover
- If more Partitions than Consumer Group instances,
  - Some Consumers will read from more than one partition

2 server Kafka cluster hosting 4 partitions (P0-P5)



Multi-threaded Consumers

- You can run more than one Consumer in a JVM process
- If processing records takes a while, a single Consumer can run multiple threads to process records
  - Harder to manage offset for each Thread/Task
  - One Consumer runs multiple threads
  - 2 messages on same partitions being processed by two different threads
  - Hard to guarantee order without threads coordination
- ***PREFER***: Multiple Consumers can run each processing record batches in their own thread
  - Easier to manage offset
  - Each Consumer runs in its thread
  - Easier to mange failover (each process runs X num of Consumer threads)

Consumer Review

- What is a consumer group?
- Does each consumer have its own offset?
- When can a consumer see a record?
- What happens if there are more consumers than partitions?
- What happens if you run multiple consumers in many thread in the same JVM?

Using Kafka Single Node

Using Kafka Single Node

Run ZooKeeper, Kafka
Create a topic
Send messages from command line
Read messages from command line
Tutorial Using Kafka Single Node

Run Kafka

- Run ZooKeeper start up script
- Run Kafka Server/Broker start up script
- Create Kafka Topic from command line
- Run producer from command line
- Run consumer from command line

Run ZooKeeper

run-zookeeper.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh \
   kafka/config/zookeeper.properties

```

```
$ ./run-zookeeper.sh
[2017-05-13 13:34:52,489] INFO Reading configuration from: kafka/config/zookeeper.properties (org.
apache.zookeeper.server.quorum.QuorumPeerConfig)
[2017-05-13 13:34:52,491] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.Dat
adirCleanupManager)
[2017-05-13 13:34:52,491] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.Datad
irCleanupManager)
[2017-05-13 13:34:52,491] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCl
eanupManager)
[2017-05-13 13:34:52,491] WARN Either no config or no quorum defined in config, running  in standa
lone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2017-05-13 13:34:52,504] INFO Reading configuration from: kafka/config/zookeeper.properties (org.
apache.zookeeper.server.quorum.QuorumPeerConfig)
[2017-05-13 13:34:52,504] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
[2017-05-13 13:34:57,609] INFO Server environment:zookeeper.version=3.4.9-1757313, built on 08/23/
2016 06:50 GMT (org.apache.zookeeper.server.ZooKeeperServer)
[2017-05-13 13:34:57,609] INFO Server environment:host.name=10.0.0.115 (org.apache.zookeeper.serve
```

Run Kafka Server

run-kafka.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

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

```
$ ./run-kafka.sh
[2017-05-13 13:47:01,497] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
        auto.create.topics.enable = true
        auto.leader.rebalance.enable = true
        background.threads = 10
        broker.id = 0
        broker.id.generation.enable = true
        broker.rack = null
        compression.type = producer
        connections.max.idle.ms = 600000
        controlled.shutdown.enable = true
        controlled.shutdown.max.retries = 3
        controlled.shutdown.retry.backoff.ms = 5000
        controller.socket.timeout.ms = 30000
```

Create Kafka Topic

create-topic.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# Create a topic
kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
--replication-factor 1 --partitions 13 --topic my-topic
```

```
$ ./create-topic.sh
Created topic "my-topic".
```

List Topics

list-topics.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --list \
    --zookeeper localhost:2181

```

```
~/kafka-training/lab1/solution
$ ./list-topics.sh
__consumer_offsets
_schemas
my-example-topic
my-example-topic2
my-topic
new-employees
```

Run Kafka Producer

start-producer-console.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-producer.sh --broker-list \
localhost:9092 --topic my-topic
```

Run Kafka Consumer

start-consumer-console.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

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

Running Kafka Producer and Consumer

```
new-employees
~/kafka-training/lab1/solution
$ ./start-producer-console.sh
This is message 1
This is message 2
This is message 3
Message 4
Message 5
Message 6
Message 7
```

```
Last login: Sat May 13 13:57:09 on ttys004
~/kafka-training/lab1/solution
$ ./start-consumer-console.sh
Message 4
This is message 2
This is message 1
This is message 3
Message 5
Message 6
Message 7
```

?Kafka Single Node Review

- What server do you run first?
- What tool do you use to create a topic?
- What tool do you use to see topics?
- What tool did we use to send messages on the command line?
- What tool did we use to view messages in a topic?
- Why were the messages coming out of order?
- How could we get the messages to come in order from the consumer?

Use Kafka to send and receive messages

Lab Use Kafka

Use single server version of Kafka.
Setup single node.
Single ZooKeeper.
Create a topic.
Produce and consume messages from the command line.

CLOUDURABLE ™

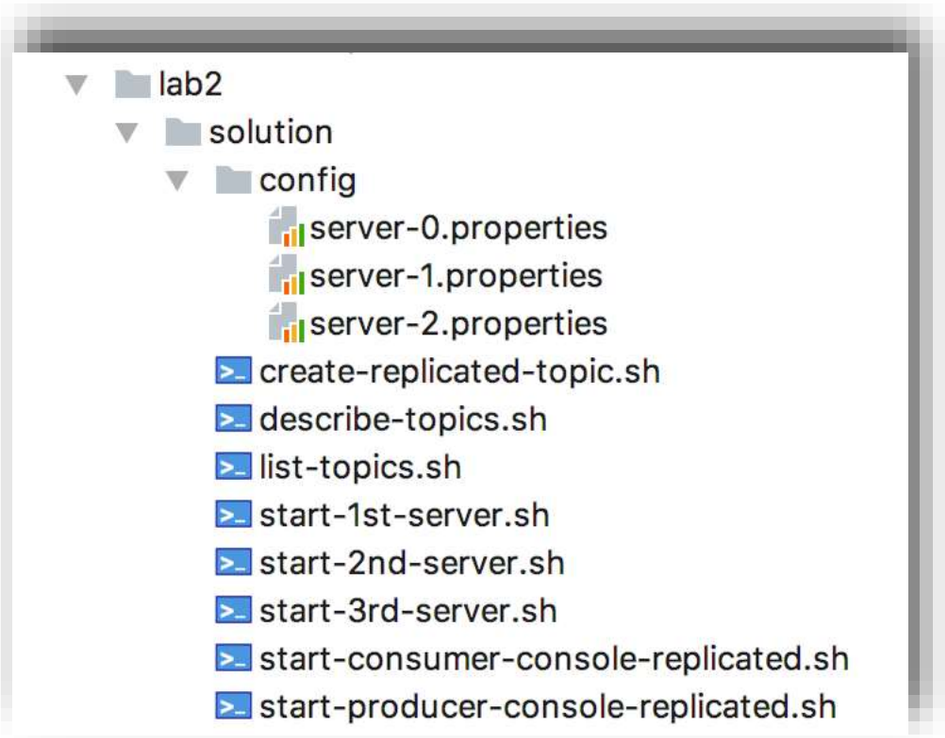
Cassandra / Kafka Support in EC2/AWS. *Kafka Training*, *Kafka Consulting*

Using Kafka Cluster and Failover

Demonstrate Kafka Cluster
Create topic with replication
Show consumer failover
Show broker failover
Kafka Tutorial Cluster and Failover

Objectives

- Run many Kafka Brokers
- Create a replicated topic
- Demonstrate Pub / Sub
- Demonstrate load balancing consumers
- Demonstrate consumer failover
- Demonstrate broker failover



Running many nodes

- If not already running, start up ZooKeeper
  - Shutdown Kafka from first lab

```
~/kafka-training
$ ./run-zookeeper.sh
```

- Copy server properties for three brokers
  - Modify properties files, Change port, Change Kafka log location
- Start up many Kafka server instances
- Create Replicated Topic
- Use the replicated topic

Create three new server-n.properties files

- Copy existing ***server.properties*** to ***server-0.properties, server-1.properties***, ***server-2.properties***
- Change ***server-1.properties*** to use ***log.dirs*** "./***logs/kafka-logs-0***"
- Change ***server-1.properties*** to use ***port 9093***, ***broker id 1***, and ***log.dirs*** "./***logs/kafka-logs-1***"
- Change ***server-2.properties*** to use ***port 9094***, ***broker id 2***, and ***log.dirs*** "./***logs/kafka-logs-2***"

Modify server-x.properties

server-0.properties

```
broker.id=0
port=9092
log.dirs=./logs/kafka-0
```

server-1.properties

```
broker.id=1
port=9093
log.dirs=./logs/kafka-1
```

server-2.properties

```
broker.id=2
port=9094
log.dirs=./logs/kafka-2
```

- Each have different ***broker.id***
- Each have different ***log.dirs***
- Each had different ***port***

Create Startup scripts for three Kafka servers

start-1st-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config

cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"
```

start-2nd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

start-3rd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-2.properties"
```

start-2nd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training

## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

- Passing properties files from last step

Run Servers

```
$ ./start-1st-server.sh
[2017-05-15 11:18:00,168] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
```

```
$ ./start-2nd-server.sh
[2017-05-15 11:18:24,980] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
```

```
~/kafka-training/lab2/solution
$ ./start-3rd-server.sh
[2017-05-15 11:19:04,129] INFO KafkaConfig v
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
```

Create Kafka replicated topic my-failsafe-topic

create-replicated-topic.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 13 \
    --topic my-failsafe-topic

```

- ***Replication Factor*** is set to 3
- Topic name is ***my-failsafe-topic***
- ***Partitions*** is 13

```
$ ./create-replicated-topic.sh
Created topic "my-failsafe-topic".
```

Start Kafka Consumer

start-consumer-console-replicated.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9094,localhost:9092 \
    --topic my-failsafe-topic \
    --from-beginning

```

- Pass list of Kafka servers to bootstrap-server
- We pass two of the three
- Only one needed, it learns about the rest

Start Kafka Producer

start-producer-console-replicated.sh

```
1  #!/usr/bin/env bash
2  cd ~/kafka-training
3
4  kafka/bin/kafka-console-producer.sh \
5  --broker-list localhost:9092,localhost:9093 \
6  --topic my-failsafe-topic
7
```

- Start producer
- Pass list of Kafka Brokers

Kafka 1 consumer and 1 producer running

```
Last login: Mon May 15 11:25:19 on ttys007
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
```

solution — java ◂ sta

```
Last login: Mon May 15 11:19:27 on ttys006
~/kafka-training/lab2/solution
$ ls
config                                  start-2
create-replicated-topic.sh              start-3
list-topics.sh                          start-c
start-1st-server.sh                     start-p
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
```

Start a second and third consumer

```
$ ./start-producer-console-replicated.sh
Hi mom
How are you?
How are things going?
Good!
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:28:21 on ttys011
~/kafka-training/lab2/solution
$ ./start-co
Good!
How are thin
How are you?
Hi mom
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:35:19 on ttys007
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

```
Last login: Mon May 15 11:35:35 on ttys011
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
Good!
How are things going?
How are you?
Hi mom
message 1
message 2
message 3
```

- Acts like pub/sub
- Each consumer in its own group
- Message goes to each
- How do we load share?

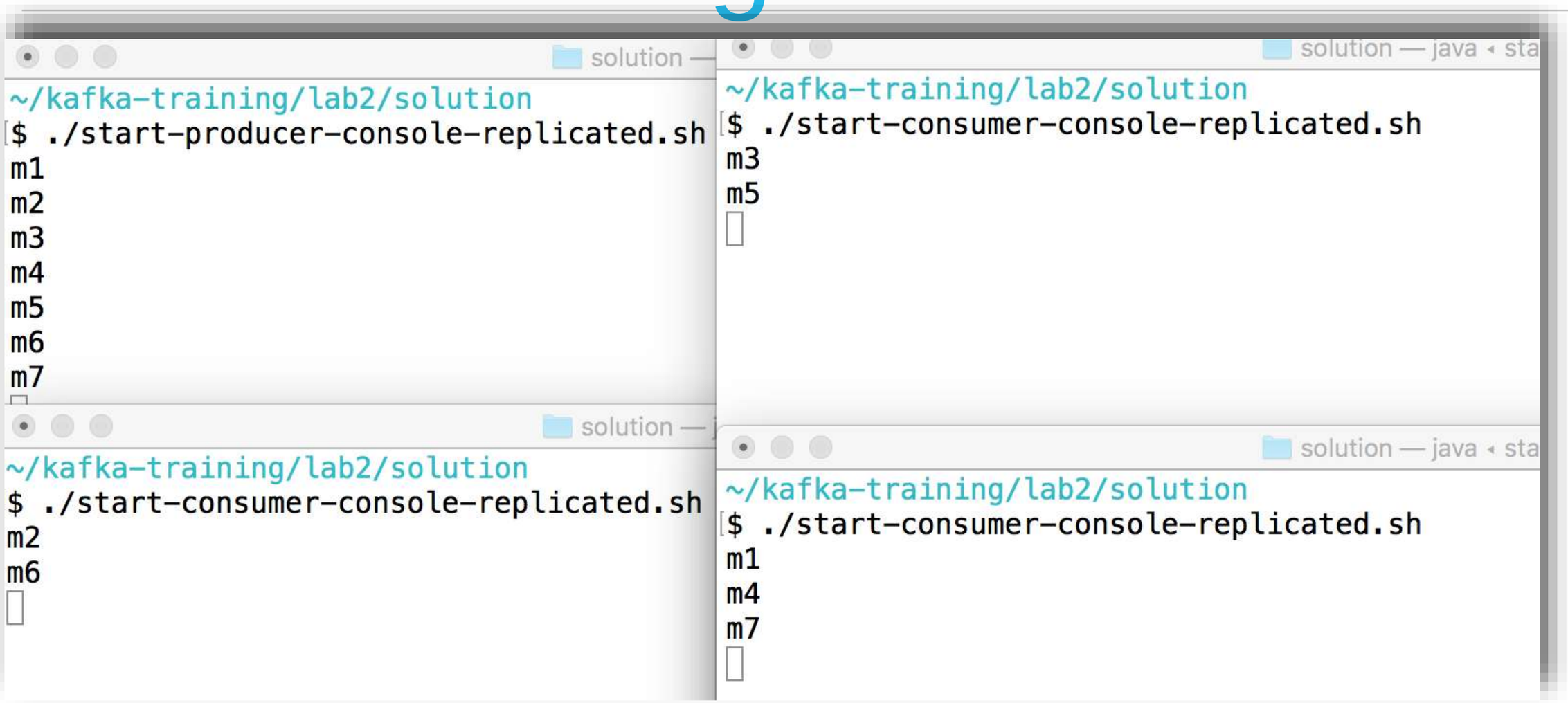
Running consumers in same group

start-consumer-console-replicated.sh

```
1  #!/usr/bin/env bash
2  cd ~/kafka-training
3
4  kafka/bin/kafka-console-consumer.sh \
5      --bootstrap-server localhost:9094,localhost:9092 \
6      --topic my-failsafe-topic \
7  --consumer-property group.id=mygroup
8  --from-beginning
9
```

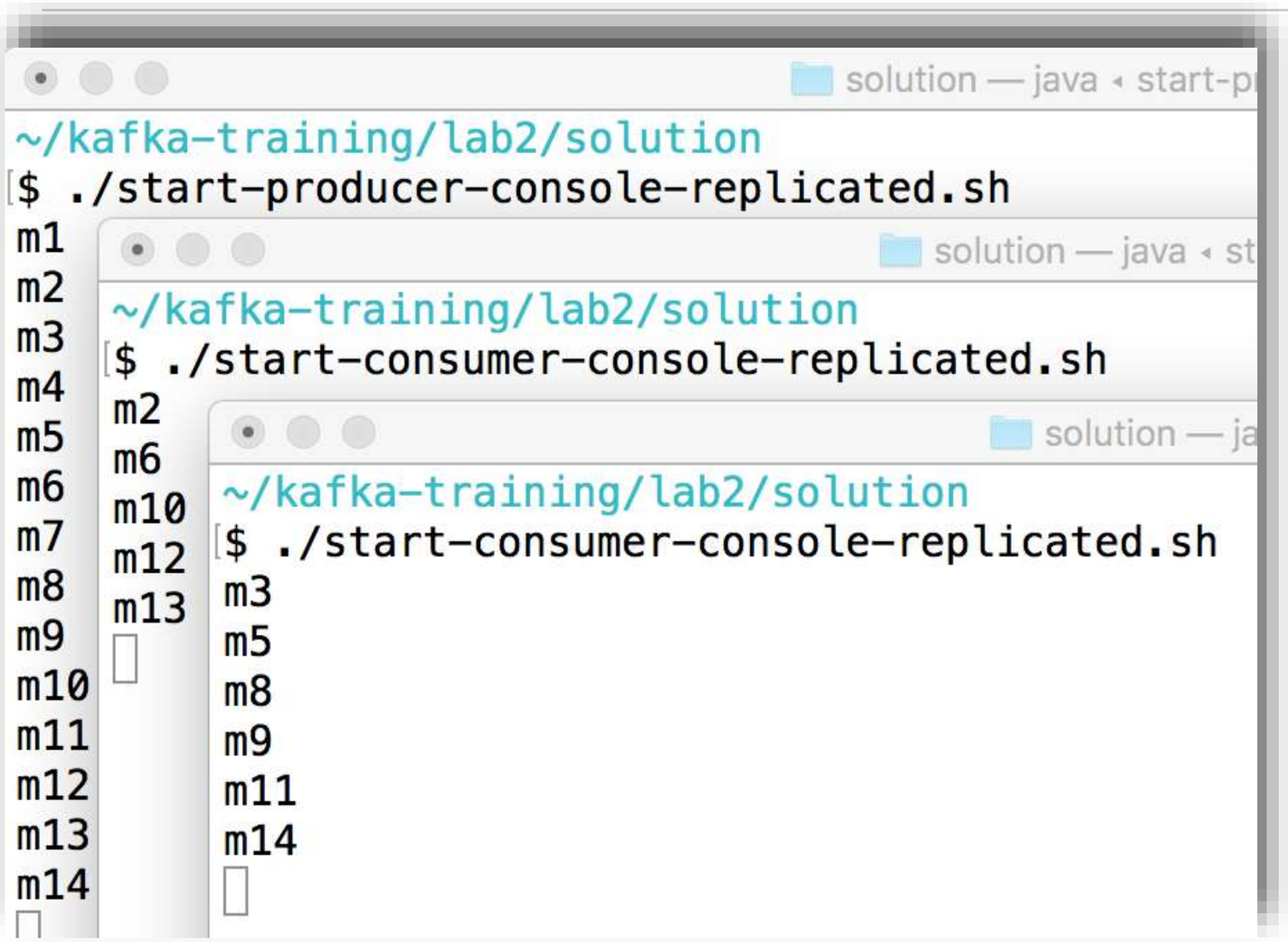
- Modify start consumer script
- Add the consumers to a group called mygroup
- Now they will share load

Start up three consumers again



- Start up producer and three consumers
- Send 7 messages
- Notice how messages are spread among 3 consumers

Consumer Failover



- Kill one consumer
- Send seven more messages
- Load is spread to remaining consumers
- Failover WORK!

Create Kafka Describe Topic

describe-topics.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

# List existing topics
kafka/bin/kafka-topics.sh --describe \
    --topic my-failsafe-topic \
    --zookeeper localhost:2181

```

- —describe will show list partitions, ISRs, and partition leadership

Use Describe Topics

```
$ ./describe-topics.sh
Topic:my-failsafe-topic PartitionCount:13       ReplicationFactor:3     Configs:
        Topic: my-failsafe-topic        Partition: 0    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
        Topic: my-failsafe-topic        Partition: 1    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: my-failsafe-topic        Partition: 2    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: my-failsafe-topic        Partition: 3    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
        Topic: my-failsafe-topic        Partition: 4    Leader: 0       Replicas: 0,2,1 Isr: 0,2,1
        Topic: my-failsafe-topic        Partition: 5    Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
        Topic: my-failsafe-topic        Partition: 6    Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
        Topic: my-failsafe-topic        Partition: 7    Leader: 0       Replicas: 0,1,2 Isr: 0,1,2
        Topic: my-failsafe-topic        Partition: 8    Leader: 1       Replicas: 1,2,0 Isr: 1,2,0
        Topic: my-failsafe-topic        Partition: 9    Leader: 2       Replicas: 2,1,0 Isr: 2,1,0
        Topic: my-failsafe-topic        Partition: 10   Leader: 0       Replicas: 0,2,1 Isr: 0,2,1
        Topic: my-failsafe-topic        Partition: 11   Leader: 1       Replicas: 1,0,2 Isr: 1,0,2
        Topic: my-failsafe-topic        Partition: 12   Leader: 2       Replicas: 2,0,1 Isr: 2,0,1
```

- Lists which broker owns (leader of) which partition
- Lists Replicas and ISR (replicas that are up to date)
- Notice there are 13 topics

Test Broker Failover: Kill 1st server

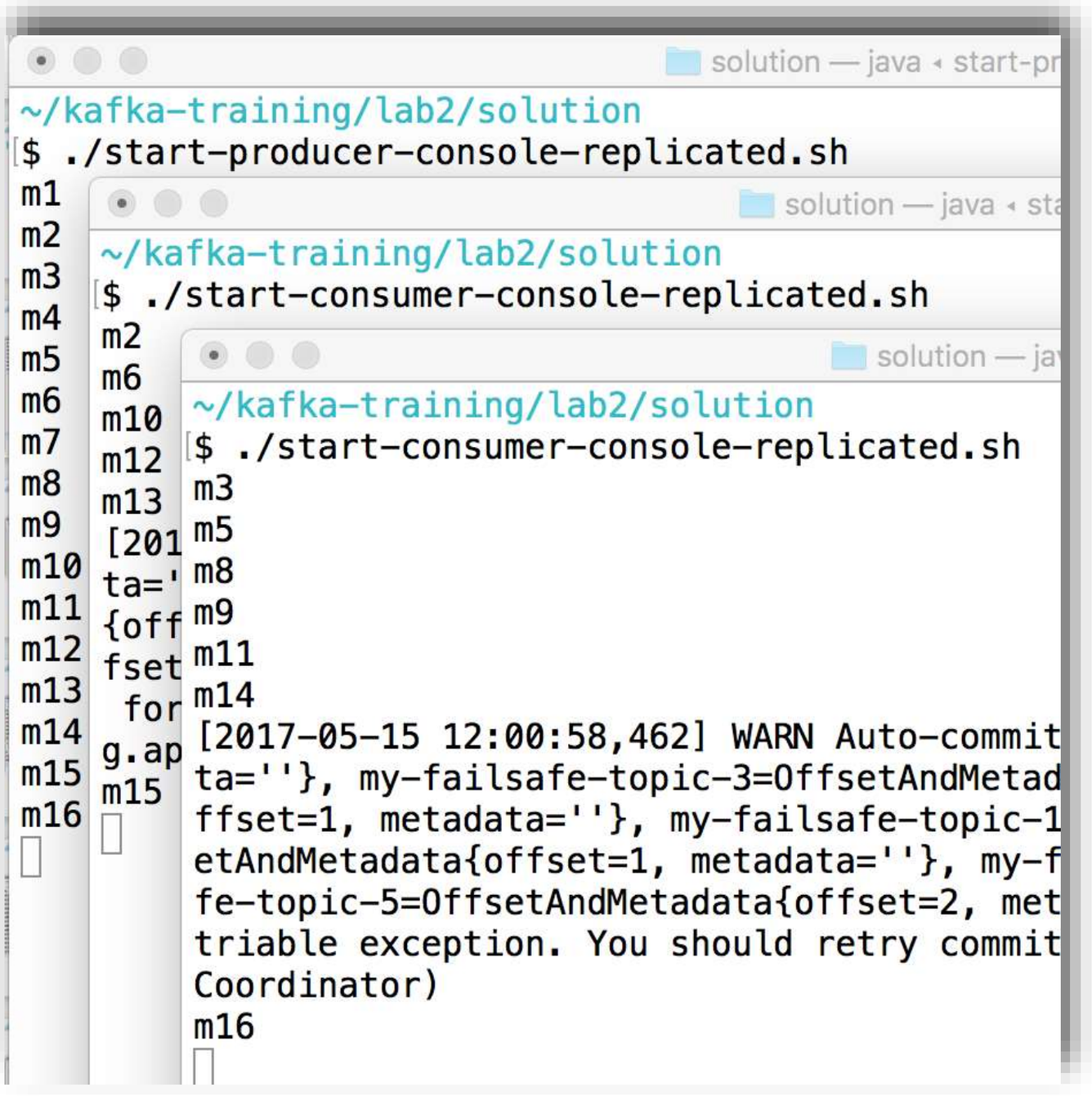
Kill the first server

```
~/kafka-training/lab2/solution
$ kill `ps aux | grep java | grep server-0.properties | tr -s " " | cut -d " " -f2`
```

se Kafka topic describe to see that a new leader was elected!

```
$ ./describe-topics.sh
Topic:my-failsafe-topic PartitionCount:13       ReplicationFactor:3     Configs:
        Topic: my-failsafe-topic        Partition: 0    Leader: 2       Replicas: 2,0,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 1    Leader: 1       Replicas: 0,1,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 2    Leader: 1       Replicas: 1,2,0 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 3    Leader: 2       Replicas: 2,1,0 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 4    Leader: 2       Replicas: 0,2,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 5    Leader: 1       Replicas: 1,0,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 6    Leader: 2       Replicas: 2,0,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 7    Leader: 1       Replicas: 0,1,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 8    Leader: 1       Replicas: 1,2,0 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 9    Leader: 2       Replicas: 2,1,0 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 10   Leader: 2       Replicas: 0,2,1 Isr: 2,1
        Topic: my-failsafe-topic        Partition: 11   Leader: 1       Replicas: 1,0,2 Isr: 1,2
        Topic: my-failsafe-topic        Partition: 12   Leader: 2       Replicas: 2,0,1 Isr: 2,1
```

Show Broker Failover Worked



- Send two more messages from the producer
- Notice that the consumer gets the messages
- Broker Failover WORKS!

Kafka Cluster Review

- Why did the three consumers not load share the messages at first?
- How did we demonstrate failover for consumers?
- How did we demonstrate failover for producers?
- What tool and option did we use to show ownership of partitions and the ISRs?

Use Kafka to send and receive messages

Lab 2 Use Kafka multiple nodes

Use a Kafka Cluster to replicate a Kafka topic log

Kafka Ecosystem

Kafka Connect
Kafka Streaming
Kafka Schema Registry
Kafka REST

Kafka Universe

- ***Ecosystem is Apache Kafka Core plus these (and community Kafka Connectors)***
- ***Kafka Streams***
  - ***Streams*** API to transform, aggregate, process records from a stream and produce derivative streams
- ***Kafka Connect***
  - ***Connector*** API reusable producers and consumers
  - (e.g., stream of changes from DynamoDB)
- ***Kafka REST Proxy***
  - Producers and Consumers over REST (HTTP)
- ***Schema Registry*** - Manages schemas using Avro for Kafka Records
- ***Kafka MirrorMaker*** - Replicate cluster data to another cluster

What comes in Apache Kafka Core?

Apache Kafka Core Includes:

- ZooKeeper and startup scripts
- Kafka Server (Kafka Broker), Kafka Clustering
- Utilities to monitor, create topics, inspect topics, replicated (mirror) data to another datacenter
- Producer APIs, Consumer APIs
- Part of Apache Foundation
- Packages / Distributions are free do download with no registry

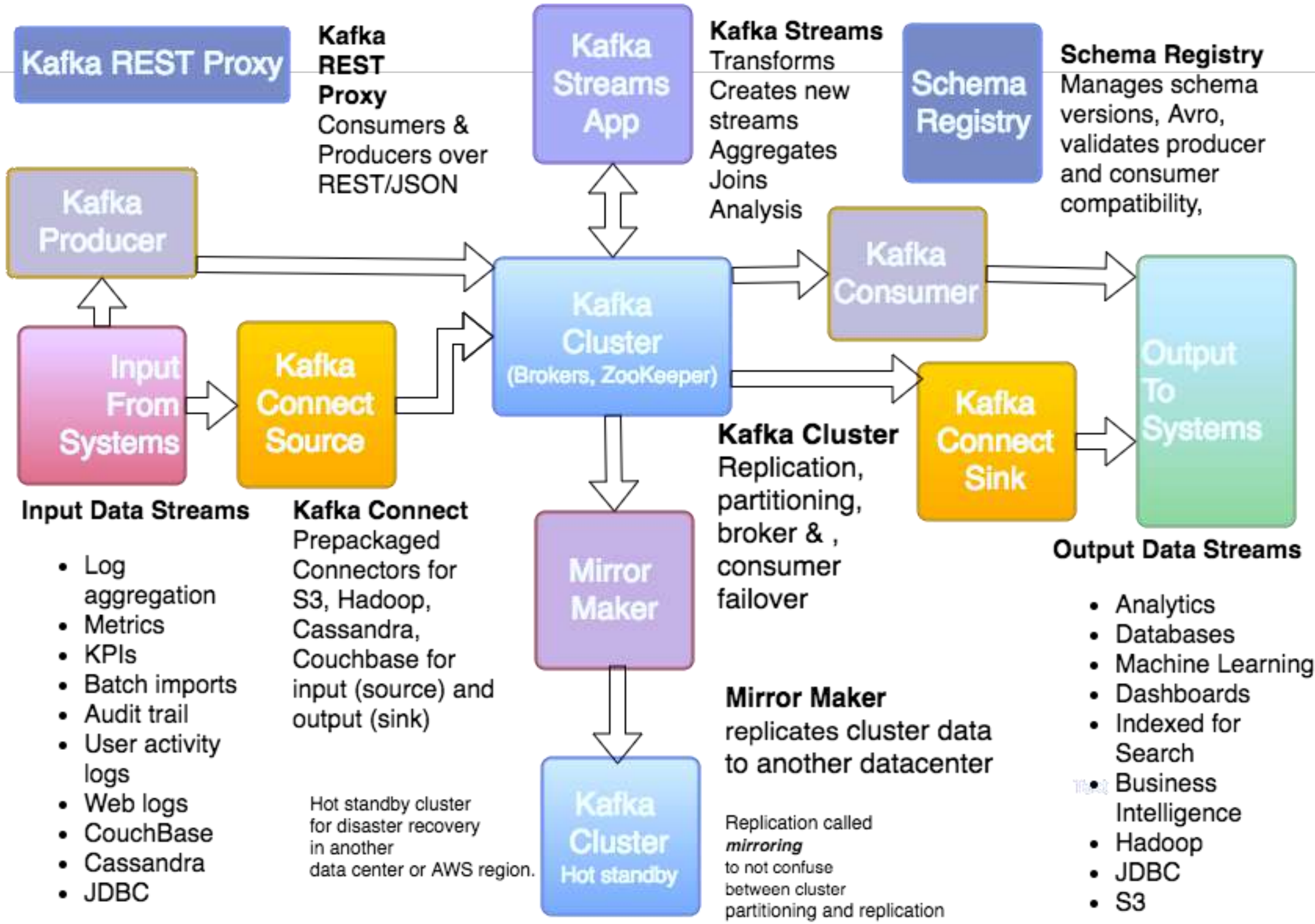
What comes in Kafka Extensions?

Confluent.io:

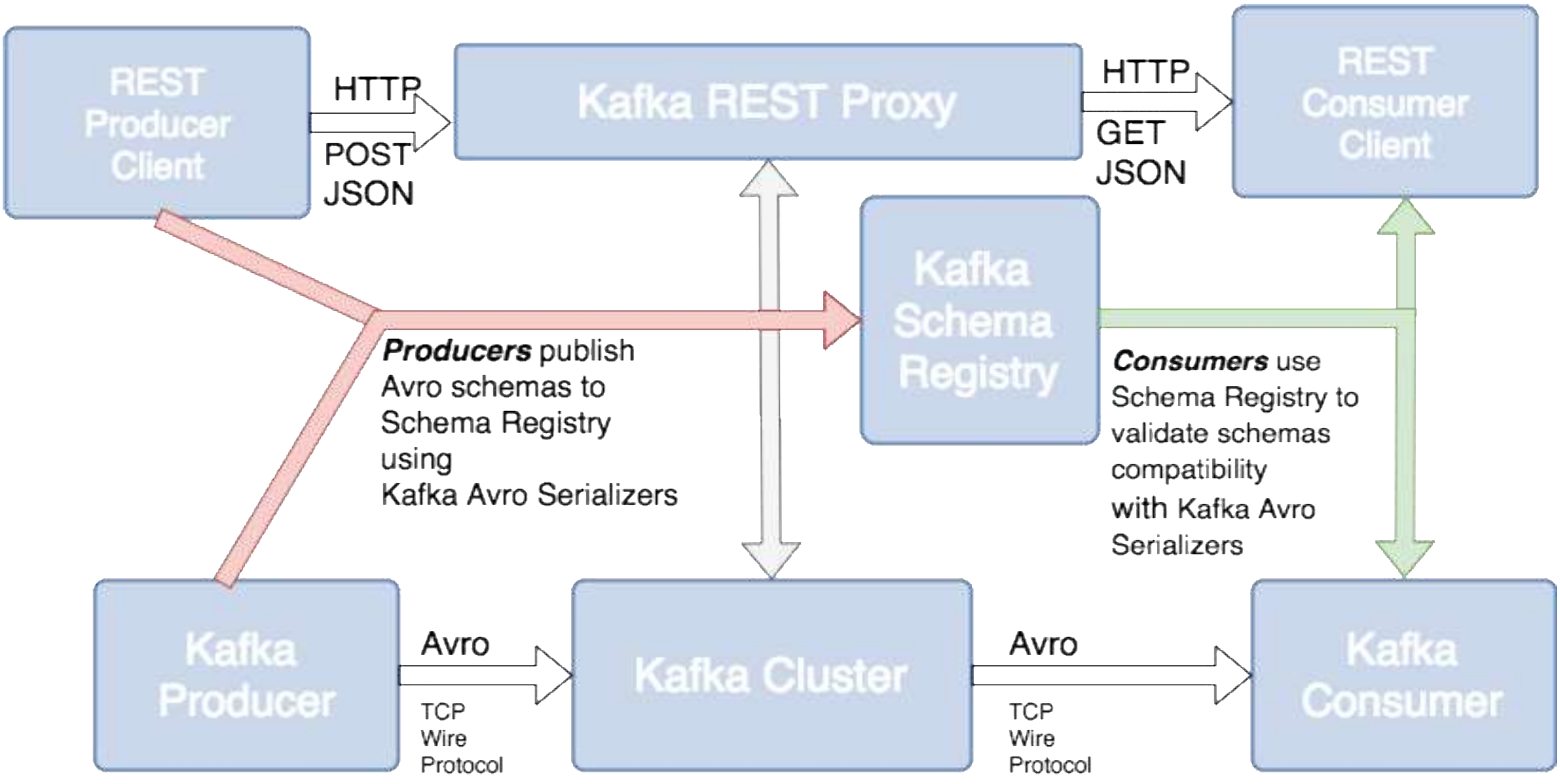
- All of Kafka Core
- ***Schema Registry*** (schema versioning and compatibility checks) (Confluent project)
- ***Kafka REST Proxy (Confluent project)***
- ***Kafka Streams*** (aggregation, joining streams, mutating streams, creating new streams by combining other streams) (Confluent project)
- ***Not*** Part of Apache Foundation controlled by Confluent.io
- Code hosted on GitHub
- Packages / Distributions are free do download ***but you must register with*** Confluent

Community of Kafka Connectors from 3rd parties and Confluent

Kafka Universe



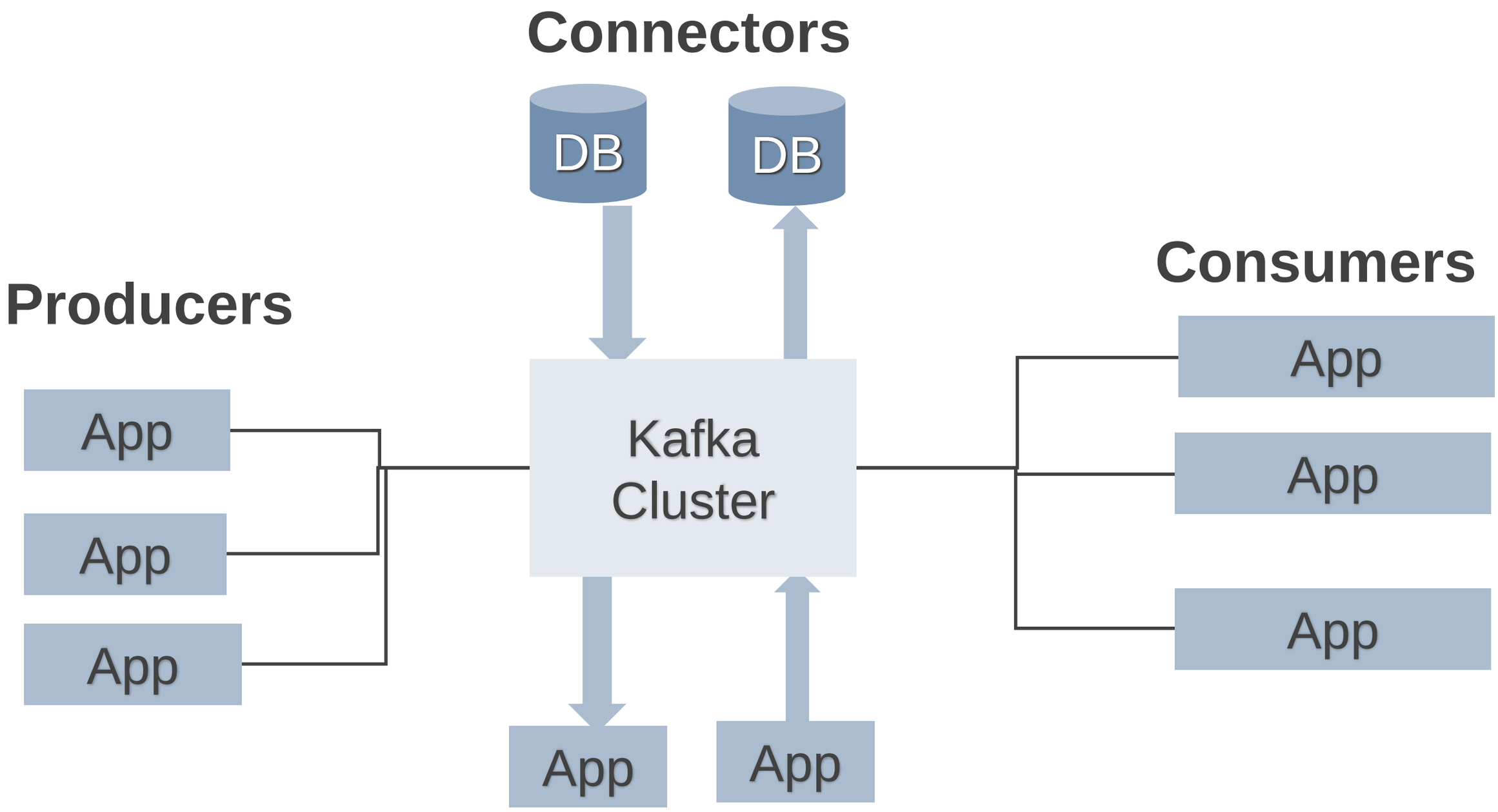
Kafka REST Proxy and Schema Registry



Kafka Stream : Stream Processing

- ***Kafka Streams*** for Stream Processing
  - Kafka enable ***real-time*** processing of streams.
- Kafka Streams supports ***Stream Processor***
  - processing, transformation, aggregation, and produces 1 to * output streams
- Example: video player app sends events videos watched, videos paused
  - output a new stream of user preferences
  - can gear new video recommendations based on recent user activity
  - can aggregate activity of many users to see what new videos are hot
- Solves hard problems: out of order records, aggregating/joining across streams, stateful computations, and more

Kafka Connectors and Streams



? Kafka Ecosystem review

- What is Kafka Streams?
- What is Kafka Connect?
- What is the Schema Registry?
- What is Kafka Mirror Maker?
- When might you use Kafka REST Proxy?

References

- **Learning Apache Kafka**, Second Edition 2nd Edition by Nishant Garg (Author), 2015, ISBN 978-1784393090, Packet Press
- ***Apache Kafka Cookbook***, 1st Edition, Kindle Edition by Saurabh Minni (Author), 2015, ISBN 978-1785882449, Packet Press
- Kafka Streams for Stream processing: A few words about how Kafka works, Serban Balamaci, 2017, ***Blog: Plain Ol' Java***
- Kafka official documentation, 2017
- Why we need Kafka? Quora
- Why is Kafka Popular? Quora
- Why is Kafka so Fast? Stackoverflow
- Kafka growth exploding (Tech Republic)
- Apache Kafka Series - Learning Apache Kafka for Beginners - great introduction to using Kafka on Udemy by Stephane Maarek

Working with Kafka Consumers – Java Example Producer

Kafka Producer Introduction Java Examples

Working with producers in Java

Step by step first example

Creating a Kafka Producer in Java

Objectives Create Producer

- Create simple example that creates a ***Kafka Producer***
- Create a new replicated ***Kafka topic***
- ***Create Producer*** that uses topic to send records
- ***Send records*** with ***Kafka Producer***
- ***Send records asynchronously.***
- ***Send records synchronously***

Create Replicated Kafka Topic

create-topic.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

## Create topics
kafka/bin/kafka-topics.sh --create \
    --replication-factor 3 \
    --partitions 13 \
    --topic my-example-topic \
    --zookeeper  localhost:2181


## List created topics
kafka/bin/kafka-topics.sh --list \
    --zookeeper localhost:2181
```

```
$ ./create-topic.sh
Created topic "my-example-topic".
EXAMPLE_TOPIC
__consumer_offsets
kafkatopic
my-example-topic
my-failsafe-topic
my-topic
```

Gradle Build script

kafka-training

```
group 'cloudurable-kafka'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.apache.kafka:kafka-clients:0.10.2.0'
    compile 'ch.qos.logback:logback-classic:1.2.2'
}
```

Create Kafka Producer to send records

- Specify bootstrap servers
- Specify client.id
- Specify Record Key serializer
- Specify Record Value serializer

Common Kafka imports and constants

KafkaProducerExample.java

KafkaProducerExample

```
1   package com.cloudurable.kafka;
2
3   import org.apache.kafka.clients.producer.*;
4   import org.apache.kafka.common.serialization.LongSerializer;
5   import org.apache.kafka.common.serialization.StringSerializer;
6
7   import java.util.Properties;
8
9   public class KafkaProducerExample {
10
11      private final static String TOPIC = "my-example-topic";
12      private final static String BOOTSTRAP_SERVERS =
13              "localhost:9092,localhost:9093,localhost:9094";
```

Create Kafka Producer to send records

KafkaProducerExample.java

```
KafkaProducerExample

14
15      private static Producer<Long, String> createProducer() {
16          Properties props = new Properties();
17          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
18                                                  BOOTSTRAP_SERVERS);
19          props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
20          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
21                                                  LongSerializer.class.getName());
22          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
23                                                  StringSerializer.class.getName());
24          return new KafkaProducer<>(props);
25      }
26
```

Send sync records with Kafka Producer

KafkaProducerExample.java

KafkaProducerExample | runProducer()

```
27
28      static void runProducer(final int sendMessageCount) throws Exception {
29          final Producer<Long, String> producer = createProducer();
30          long time = System.currentTimeMillis();
31
32          try {
33              for (long index = time; index < time + sendMessageCount; index++) {
34                  final ProducerRecord<Long, String> record =
35                          new ProducerRecord<>(TOPIC, index,
36                                  value: "Hello Mom " + index);
37
38                  RecordMetadata metadata = producer.send(record).get();
39
40                  long elapsedTime = System.currentTimeMillis() - time;
41                  System.out.printf("sent record(key=%s value=%s) " +
42                                  "meta(partition=%d, offset=%d) time=%d\n",
43                          record.key(), record.value(), metadata.partition(),
44                          metadata.offset(), elapsedTime);
45
46              }
47          } finally {
48              producer.flush();
49              producer.close();
50          }
51      }
```

Running the Producer

```
public static void main(String... args) throws Exception {
    if (args.length == 0) {
        runProducer( sendMessageCount: 5);
    } else {
        runProducer(Integer.parseInt(args[0]));
    }
}
```

```
Run  KafkaExample
/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
sent record(key=1492463982402 value=Hello Mom 1492463982402) meta(partition=0, offset=380) time=139
sent record(key=1492463982403 value=Hello Mom 1492463982403) meta(partition=0, offset=381) time=141
sent record(key=1492463982404 value=Hello Mom 1492463982404) meta(partition=0, offset=382) time=141
sent record(key=1492463982405 value=Hello Mom 1492463982405) meta(partition=0, offset=383) time=141
sent record(key=1492463982406 value=Hello Mom 1492463982406) meta(partition=0, offset=384) time=141
Got Record: (1492463982402, Hello Mom 1492463982402) at offset 380
Got Record: (1492463982403, Hello Mom 1492463982403) at offset 381
Got Record: (1492463982404, Hello Mom 1492463982404) at offset 382
Got Record: (1492463982405, Hello Mom 1492463982405) at offset 383
Got Record: (1492463982406, Hello Mom 1492463982406) at offset 384
DONE
```

Send async records with Kafka Producer

```
static void runProducer(final int sendMessageCount) throws InterruptedException {
    final Producer<Long, String> producer = createProducer();
    long time = System.currentTimeMillis();
    final CountDownLatch countDownLatch = new CountDownLatch(sendMessageCount);

    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            final ProducerRecord<Long, String> record =
                    new ProducerRecord<>(TOPIC, index, value: "Hello Mom " + index);
            producer.send(record, (metadata, exception) -> {
                long elapsedTime = System.currentTimeMillis() - time;
                if (metadata != null) {
                    System.out.printf("sent record(key=%s value=%s) " +
                                    "meta(partition=%d, offset=%d) time=%d\n",
                            record.key(), record.value(), metadata.partition(),
                            metadata.offset(), elapsedTime);
                } else {
                    exception.printStackTrace();
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await( timeout: 25, TimeUnit.SECONDS);
    }finally {
        producer.flush();
        producer.close();
```

Async Interface Callback

org.apache.kafka.clients.producer

Interface Callback

`public interface` **`Callback`**

A callback interface that the user can implement to allow code to execute when the request is complete. This callback will generally execute in the background I/O thread so it should be fast.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`onCompletion`**`(`**`RecordMetadata`** `metadata,` **`Exception`** `exception)` A callback method the user can implement to provide asynchronous handling of request completion.

Async Send Method

send

```
public Future<RecordMetadata> send(ProducerRecord<K,V> record,
                                   Callback callback)
```

Asynchronously send a record to a topic and invoke the provided callback when the send has been acknowledged.

The send is asynchronous and this method will return immediately once the record has been stored in the buffer of records waiting to be sent. This allows sending many records in parallel without blocking to wait for the response after each one.

- Used to send a record to a topic
- provided callback gets called when the send is acknowledged
- Send is asynchronous, and method will return immediately
  - once the record gets stored in the buffer of records waiting to post to the Kafka broker
- Allows sending many records in parallel without blocking

Checking that replication is working

```
$ kafka/bin/kafka-replica-verification.sh --broker-list localhost:9092 --topic-white-list my-example-topic
2017-05-17 14:06:46,446: verification process is started.
2017-05-17 14:07:16,416: max lag is 0 for partition [my-example-topic,12] at offset 197 among 13 partitions
2017-05-17 14:07:46,417: max lag is 0 for partition [my-example-topic,12] at offset 201 among 13 partitions
```

- Verify that replication is working with
  - kafka-replica-verification
  - Utility that ships with Kafka
  - If lag or outage you will see it as follows:

```
2017-05-17 14:36:47,497: max lag is 11 for partition [my-example-topic,5] at offset 272 among 13 partitions
2017-05-17 14:37:19,408: max lag is 15 for partition [my-example-topic,5] at offset 272 among 13 partitions
…
2017-05-17 14:38:49,607: max lag is 0 for partition [my-example-topic,12] at offset 272 among 13 partitions
```

Java Kafka Simple Producer recap

- Created simple example that creates a ***Kafka Producer***
- Created a new replicated ***Kafka topic***
- ***Created Producer*** that uses topic to send records
- ***Sent records*** with ***Kafka Producer using async and sync send***

? Kafka Producer Review

- What does the Callback lambda do?
- What will happen if the first server is down in the bootstrap list? Can the producer still connect to the other Kafka brokers in the cluster?
- When would you use Kafka async send vs. sync send?
- Why do you need two serializers for a Kafka record?

Working with Kafka Consumer – Java Example

Kafka Consumer Introduction Java Examples

Working with consumers

Step by step first example

Creating a Kafka Java Consumer

Objectives Create a Consumer

- Create simple example that creates a ***Kafka Consumer***
  - that consumes messages form the ***Kafka Producer*** we wrote
- ***Create Consumer*** that uses topic from first example to receive messages
- ***Process messages*** from Kafka with ***Consumer***
- Demonstrate how Consumer Groups work

Create Consumer using Topic to Receive Records

- Specify bootstrap servers
- Specify Consumer Group
- Specify Record Key deserializer
- Specify Record Value deserializer
- Subscribe to Topic from last session

Common Kafka imports and constants

KafkaConsumerExample.java

KafkaConsumerExample

```
1   package com.cloudurable.kafka;
2   import org.apache.kafka.clients.consumer.*;
3   import org.apache.kafka.clients.consumer.Consumer;
4   import org.apache.kafka.common.serialization.LongDeserializer;
5   import org.apache.kafka.common.serialization.StringDeserializer;
6
7   import java.util.Collections;
8   import java.util.Properties;
9
10  public class KafkaConsumerExample {
11
12      private final static String TOPIC = "my-example-topic";
13      private final static String BOOTSTRAP_SERVERS =
14              "localhost:9092,localhost:9093,localhost:9094";
15
```

Create Consumer using Topic to Receive Records

KafkaConsumerExample.java

KafkaConsumerExample | createConsumer()

```
16      private static Consumer<Long, String> createConsumer() {
17          final Properties props = new Properties();
18          props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
19                                    BOOTSTRAP_SERVERS);
20          props.put(ConsumerConfig.GROUP_ID_CONFIG,
21                                    "KafkaExampleConsumer");
22          props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
23                  LongDeserializer.class.getName());
24          props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
25                  StringDeserializer.class.getName());
26
27          // Create the consumer using props.
28          final Consumer<Long, String> consumer =
29                                    new KafkaConsumer<>(props);
30
31          // Subscribe to the topic.
32          consumer.subscribe(Collections.singletonList(TOPIC));
33          return consumer;
34      }
```

Process messages from Kafka with Consumer

KafkaConsumerExample.java

KafkaConsumerExample

```
40      static void runConsumer() throws InterruptedException {
41          final Consumer<Long, String> consumer = createConsumer();
42
43          final int giveUp = 100;   int noRecordsCount = 0;
44
45          while (true) {
46              final ConsumerRecords<Long, String> consumerRecords =
47                      consumer.poll( timeout: 1000);
48
49              if (consumerRecords.count()==0) {
50                  noRecordsCount++;
51                  if (noRecordsCount > giveUp) break;
52                  else continue;
53              }
54
55              consumerRecords.forEach(record -> {
56                  System.out.printf("Consumer Record:(%d, %s, %d, %d)\n",
57                          record.key(), record.value(),
58                          record.partition(), record.offset());
59              });
60
61              consumer.commitAsync();
62          }
63          consumer.close();
64          System.out.println("DONE");
```

Consumer poll

- poll() method returns fetched records based on current partition offset
- Blocking method waiting for specified time if no records available
- When/If records available, method returns straight away
- Control the maximum records returned by the poll() with `props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);`
- poll() is not meant to be called from multiple threads

Running both Consumer then Producer

```
68      public static void main(String... args) throws Exception {
69          runConsumer();
70      }
71
```

```
Run  KafkaConsumerExample
    ssl.provider = null
    ssl.secure.random.implementation = null
    ssl.trustmanager.algorithm = PKIX
    ssl.truststore.location = null
    ssl.truststore.password = null
    ssl.truststore.type = JKS
    value.deserializer = class org.apache.kafka.common.serialization.StringDeserializer

15:17:35.267 [main] INFO  o.a.kafka.common.utils.AppInfoParser - Kafka version : 0.10.2.0
15:17:35.267 [main] INFO  o.a.kafka.common.utils.AppInfoParser - Kafka commitId : 576d93a8dc0cf421
15:17:35.384 [main] INFO  o.a.k.c.c.i.AbstractCoordinator - Discovered coordinator 10.0.0.115:9093
15:17:35.391 [main] INFO  o.a.k.c.c.i.ConsumerCoordinator - Revoking previously assigned partitions
15:17:35.391 [main] INFO  o.a.k.c.c.i.AbstractCoordinator - (Re-)joining group KafkaExampleConsumer
15:17:42.257 [main] INFO  o.a.k.c.c.i.AbstractCoordinator - Successfully joined group KafkaExampleC
15:17:42.259 [main] INFO  o.a.k.c.c.i.ConsumerCoordinator - Setting newly assigned partitions [my-e
Consumer Record:(1494973064716, Hello Mom 1494973064716, 6, 4)
Consumer Record:(1494973064719, Hello Mom 1494973064719, 10, 6)
Consumer Record:(1494973064718, Hello Mom 1494973064718, 9, 9)
Consumer Record:(1494973064717, Hello Mom 1494973064717, 12, 9)
Consumer Record:(1494973064720, Hello Mom 1494973064720, 4, 8)
```

Logging

logback.xml

```
<configuration>
    <appender name="STDOUT"
              class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
                %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <logger name="org.apache.kafka" level="INFO"/>
    <logger name="org.apache.kafka.common.metrics" level="INFO"/>

    <root level="debug">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

- Kafka uses sl4j
- Set level to DEBUG to see what is going on

Try this: Consumers in Same Group

- Three consumers and one producer sending 25 records
- Run three consumers processes
- Change Producer to send 25 records instead of 5
- Run one producer
- What happens?

Outcome 3 Consumers Load Share

Consumer 0 (key, value, partition, offset)

```
Consumer Record:(1495042369488, Hello Mom 1495042369488, 0, 9)
Consumer Record:(1495042369490, Hello Mom 1495042369490, 3, 9)
Consumer Record:(1495042369498, Hello Mom 1495042369498, 3, 10)
Consumer Record:(1495042369504, Hello Mom 1495042369504, 3, 11)
Consumer Record:(1495042369508, Hello Mom 1495042369508, 3, 12)
Consumer Record:(1495042369491, Hello Mom 1495042369491, 4, 9)
Consumer Record:(1495042369503, Hello Mom 1495042369503, 4, 10)
Consumer Record:(1495042369505, Hello Mom 1495042369505, 4, 11)
Consumer Record:(1495042369494, Hello Mom 1495042369494, 2, 9)
Consumer Record:(1495042369499, Hello Mom 1495042369499, 2, 10)
```

Consumer 1 (key, value, partition, offset)

```
Consumer Record:(1495042369486, Hello Mom 1495042369486, 12, 10)
Consumer Record:(1495042369493, Hello Mom 1495042369493, 12, 11)
Consumer Record:(1495042369507, Hello Mom 1495042369507, 12, 12)
Consumer Record:(1495042369487, Hello Mom 1495042369487, 9, 12)
Consumer Record:(1495042369492, Hello Mom 1495042369492, 10, 7)
Consumer Record:(1495042369495, Hello Mom 1495042369495, 10, 8)
Consumer Record:(1495042369501, Hello Mom 1495042369501, 11, 8)
Consumer Record:(1495042369506, Hello Mom 1495042369506, 11, 9)
```

Consumer 2 (key, value, partition, offset)

```
Consumer Record:(1495042369509, Hello Mom 1495042369509, 6, 6)
Consumer Record:(1495042369497, Hello Mom 1495042369497, 7, 11)
Consumer Record:(1495042369500, Hello Mom 1495042369500, 7, 12)
Consumer Record:(1495042369496, Hello Mom 1495042369496, 5, 7)
Consumer Record:(1495042369510, Hello Mom 1495042369510, 5, 8)
Consumer Record:(1495042369489, Hello Mom 1495042369489, 8, 9)
Consumer Record:(1495042369502, Hello Mom 1495042369502, 8, 10)
```

Producer

```
ey=1495042369509 value=Hello Mom 1495042369509) meta(partition=6, offset=6) t
ey=1495042369487 value=Hello Mom 1495042369487) meta(partition=9, offset=12)
ey=1495042369486 value=Hello Mom 1495042369486) meta(partition=12, offset=10)
ey=1495042369493 value=Hello Mom 1495042369493) meta(partition=12, offset=11)
ey=1495042369507 value=Hello Mom 1495042369507) meta(partition=12, offset=12)
ey=1495042369488 value=Hello Mom 1495042369488) meta(partition=0, offset=9) t
ey=1495042369490 value=Hello Mom 1495042369490) meta(partition=3, offset=9) t
ey=1495042369498 value=Hello Mom 1495042369498) meta(partition=3, offset=10)
ey=1495042369504 value=Hello Mom 1495042369504) meta(partition=3, offset=11)
ey=1495042369508 value=Hello Mom 1495042369508) meta(partition=3, offset=12)
ey=1495042369497 value=Hello Mom 1495042369497) meta(partition=7, offset=11)
ey=1495042369500 value=Hello Mom 1495042369500) meta(partition=7, offset=12)
sent record(key=1495042369492 value=Hello Mom 1495042369492) meta(partition=10, offset=7)
sent record(key=1495042369495 value=Hello Mom 1495042369495) meta(partition=10, offset=8)
sent record(key=1495042369491 value=Hello Mom 1495042369491) meta(partition=4, offset=9) t
sent record(key=1495042369503 value=Hello Mom 1495042369503) meta(partition=4, offset=10)
ey=1495042369505 value=Hello Mom 1495042369505) meta(partition=4, offset=11)
ey=1495042369496 value=Hello Mom 1495042369496) meta(partition=5, offset=7) t
ey=1495042369510 value=Hello Mom 1495042369510) meta(partition=5, offset=8) t
ey=1495042369489 value=Hello Mom 1495042369489) meta(partition=8, offset=9) t
ey=1495042369502 value=Hello Mom 1495042369502) meta(partition=8, offset=10)
ey=1495042369501 value=Hello Mom 1495042369501) meta(partition=11, offset=8)
ey=1495042369506 value=Hello Mom 1495042369506) meta(partition=11, offset=9)
ey=1495042369494 value=Hello Mom 1495042369494) meta(partition=2, offset=9) t
ey=1495042369499 value=Hello Mom 1495042369499) meta(partition=2, offset=10)
```

Which consumer owns partition 10?
How many ConsumerRecords objects did Consumer 0 get?
What is the next offset from Partition 5 that Consumer 2 should get?
Why does each consumer get unique

Try this: Consumers in Different Groups

- Three consumers with unique group and one producer sending 5 records
- Modify Consumer to have unique group id
- Run three consumers processes
- Run one producer
- What happens?

Pass Unique Group Id

KafkaConsumerExample.java

KafkaConsumerExample | createConsumer()

```
16
17      private static Consumer<Long, String> createConsumer() {
18          final Properties props = new Properties();
19
20          props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
21                                    BOOTSTRAP_SERVERS);
22
23          props.put(ConsumerConfig.GROUP_ID_CONFIG,
24                                    "KafkaExampleConsumer" +
25                                            System.currentTimeMillis());
26
27          props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
28                  LongDeserializer.class.getName());
29          props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
30                  StringDeserializer.class.getName());
31
32
33          //Take up to 100 records at a time
34          props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);
```

Outcome 3 Subscribers

Consumer 0 (key, value, partition, offset)

```
Consumer Record:(1495043607696, Hello Mom 1495043607696, 0, 10)
Consumer Record:(1495043607699, Hello Mom 1495043607699, 7, 13)
Consumer Record:(1495043607700, Hello Mom 1495043607700, 2, 11)
Consumer Record:(1495043607697, Hello Mom 1495043607697, 10, 9)
Consumer Record:(1495043607698, Hello Mom 1495043607698, 10, 10)
```

Consumer 1 (key, value, partition, offset)

```
Consumer Record:(1495043607696, Hello Mom 1495043607696, 0, 10)
Consumer Record:(1495043607699, Hello Mom 1495043607699, 7, 13)
Consumer Record:(1495043607700, Hello Mom 1495043607700, 2, 11)
Consumer Record:(1495043607697, Hello Mom 1495043607697, 10, 9)
Consumer Record:(1495043607698, Hello Mom 1495043607698, 10, 10)
```

Consumer 2 (key, value, partition, offset)

```
Consumer Record:(1495043607696, Hello Mom 1495043607696, 0, 10)
Consumer Record:(1495043607699, Hello Mom 1495043607699, 7, 13)
Consumer Record:(1495043607700, Hello Mom 1495043607700, 2, 11)
Consumer Record:(1495043607697, Hello Mom 1495043607697, 10, 9)
Consumer Record:(1495043607698, Hello Mom 1495043607698, 10, 10)
```

Producer

```
sent (key=1495043832401 ) meta(partition=7, offset=14)
sent (key=1495043832400 ) meta(partition=1, offset=11)
sent (key=1495043832404 ) meta(partition=6, offset=7)
sent (key=1495043832402 ) meta(partition=0, offset=11)
sent (key=1495043832403 ) meta(partition=3, offset=13)
```

Which consumer(s) owns partition 10?

How many ConsumerRecords objects did Consumer 0 get?

What is the next offset from Partition 2 that Consumer 2 should get?

Why does each consumer get the same messages?

Try this: Consumers in Different Groups

- Modify consumer: change group id back to non-unique value
- Make the batch size 5
- Add a 100 ms delay in the consumer after each message poll and print out record count and partition count
- Modify the Producer to run 10 times with a 30 second delay after each run and to send 50 messages each run
- Run producer

Modify Consumer

KafkaConsumerExample.java

```
KafkaConsumerExample
 8  import java.util.Properties;
 9
10  public class KafkaConsumerExample {
11
12      private final static String TOPIC = "my-example-topic";
13      private final static String BOOTSTRAP_SERVERS =
14              "localhost:9092,localhost:9093,localhost:9094";
15
16
17      private static Consumer<Long, String> createConsumer() {
18          final Properties props = new Properties();
19
20          props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
21                                      BOOTSTRAP_SERVERS);
22
23          props.put(ConsumerConfig.GROUP_ID_CONFIG,
24                                      "KafkaExampleConsumer");
25
26          props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
27                  LongDeserializer.class.getName());
28          props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
29                  StringDeserializer.class.getName());
30
31          props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 5);
```

- Change group name to common name
- Change batch size to 5

Add a 100 ms delay to Consumer after poll

```
47          try {
48              final int giveUp = 1000; int noRecordsCount = 0;
49
50              while (true) {
51                  final ConsumerRecords<Long, String> consumerRecords =
52                          consumer.poll( timeout: 1000);
53
54                  if (consumerRecords.count() == 0) {
55                      noRecordsCount++;
56                      if (noRecordsCount > giveUp) break;
57                      else continue;
58                  }
59
60                  System.out.printf("New ConsumerRecords par count %d count %d\n",
61                          consumerRecords.partitions().size(),
62                          consumerRecords.count());
63
64                  consumerRecords.forEach(record -> {
65                      System.out.printf("Consumer Record:(%d, %s, %d, %d)\n",
66                              record.key(), record.value(),
67                              record.partition(), record.offset());
68                  });
69                  Thread.sleep( millis: 100);
70                  consumer.commitAsync();
71              }
72          }
73          finally {
74              consumer.close();
75          }
```

Modify Producer: Run 10 times, add 30 second delay

KafkaProducerExample.java

```
KafkaProducerExample  main()
104
105     public static void main(String... args)
106                                         throws Exception {
107         for (int index = 0; index < 10; index++) {
108             runProducer( sendMessageCount: 50);
109             Thread.sleep( millis: 30_000);
110         }
111     }
112 }
113
```

- Run 10 times
- Add 30 second delay
- Send 50 records

Notice one or more partitions per ConsumerRecords

KafkaConsumerExample | KafkaConsumerExample | KafkaConsumerExample

```
New ConsumerRecords par count 1 count 4
Consumer Record:(1495055263352, Hello Mom 1495055263352, 2, 189)
Consumer Record:(1495055263355, Hello Mom 1495055263355, 2, 190)
Consumer Record:(1495055263365, Hello Mom 1495055263365, 2, 191)
Consumer Record:(1495055263368, Hello Mom 1495055263368, 2, 192)
New ConsumerRecords par count 2 count 4
Consumer Record:(1495055263340, Hello Mom 1495055263340, 0, 175)
Consumer Record:(1495055263362, Hello Mom 1495055263362, 0, 176)
Consumer Record:(1495055263335, Hello Mom 1495055263335, 4, 176)
Consumer Record:(1495055263358, Hello Mom 1495055263358, 4, 177)
New ConsumerRecords par count 2 count 5
Consumer Record:(1495055263338, Hello Mom 1495055263338, 1, 219)
Consumer Record:(1495055263341, Hello Mom 1495055263341, 1, 220)
Consumer Record:(1495055263339, Hello Mom 1495055263339, 3, 185)
Consumer Record:(1495055263354, Hello Mom 1495055263354, 3, 186)
Consumer Record:(1495055263371, Hello Mom 1495055263371, 3, 187)
New ConsumerRecords par count 1 count 5
Consumer Record:(1495055263351, Hello Mom 1495055263351, 1, 221)
Consumer Record:(1495055263353, Hello Mom 1495055263353, 1, 222)
Consumer Record:(1495055263356, Hello Mom 1495055263356, 1, 223)
Consumer Record:(1495055263366, Hello Mom 1495055263366, 1, 224)
Consumer Record:(1495055263367, Hello Mom 1495055263367, 1, 225)
```

Now run it again but..

- Run the consumers and producer again
- Wait 30 seconds
- While the producer is running kill one of the consumers and see the records go to the other consumers
- Now leave just one consumer running, all of the messages should go to the remaining consumer
  - Now change consumer batch size to 500
    props.put(ConsumerConfig.***MAX_POLL_RECORDS_CONFIG***, 500)
  - and run it again

Output form batch size 500

```
New ConsumerRecords par count 7 count 28
Consumer Record:(1495056566578, Hello Mom 1495056566578, 5, 266)
Consumer Record:(1495056566591, Hello Mom 1495056566591, 5, 267)
Consumer Record:(1495056566603, Hello Mom 1495056566603, 5, 268)
Consumer Record:(1495056566605, Hello Mom 1495056566605, 5, 269)
Consumer Record:(1495056566581, Hello Mom 1495056566581, 8, 238)
Consumer Record:(1495056566592, Hello Mom 1495056566592, 8, 239)
Consumer Record:(1495056566597, Hello Mom 1495056566597, 8, 240)
Consumer Record:(1495056566598, Hello Mom 1495056566598, 8, 241)
Consumer Record:(1495056566607, Hello Mom 1495056566607, 8, 242)
Consumer Record:(1495056566609, Hello Mom 1495056566609, 8, 243)
Consumer Record:(1495056566625, Hello Mom 1495056566625, 8, 244)
Consumer Record:(1495056566626, Hello Mom 1495056566626, 8, 245)
Consumer Record:(1495056566584, Hello Mom 1495056566584, 10, 253)
Consumer Record:(1495056566585, Hello Mom 1495056566585, 10, 254)
Consumer Record:(1495056566594, Hello Mom 1495056566594, 10, 255)
Consumer Record:(1495056566601, Hello Mom 1495056566601, 10, 256)
Consumer Record:(1495056566618, Hello Mom 1495056566618, 10, 257)
Consumer Record:(1495056566619, Hello Mom 1495056566619, 10, 258)
Consumer Record:(1495056566593, Hello Mom 1495056566593, 11, 230)
Consumer Record:(1495056566600, Hello Mom 1495056566600, 11, 231)
Consumer Record:(1495056566586, Hello Mom 1495056566586, 2, 265)
Consumer Record:(1495056566596, Hello Mom 1495056566596, 1, 296)
Consumer Record:(1495056566624, Hello Mom 1495056566624, 1, 297)
Consumer Record:(1495056566595, Hello Mom 1495056566595, 4, 242)
Consumer Record:(1495056566604, Hello Mom 1495056566604, 4, 243)
Consumer Record:(1495056566610, Hello Mom 1495056566610, 4, 244)
Consumer Record:(1495056566622, Hello Mom 1495056566622, 4, 245)
Consumer Record:(1495056566623, Hello Mom 1495056566623, 4, 246)
New ConsumerRecords par count 5 count 22
Consumer Record:(1495056566599, Hello Mom 1495056566599, 6, 236)
Consumer Record:(1495056566613, Hello Mom 1495056566613, 6, 237)
Consumer Record:(1495056566620, Hello Mom 1495056566620, 6, 238)
Consumer Record:(1495056566579, Hello Mom 1495056566579, 9, 267)
Consumer Record:(1495056566583, Hello Mom 1495056566583, 9, 268)
```

Java Kafka Simple Consumer Example Recap

- Created simple example that creates a ***Kafka Consumer*** to consume messages from our ***Kafka Producer***
- Used the replicated ***Kafka topic*** from first example
- ***Created Consumer*** that uses topic to receive messages
- ***Processed records*** from Kafka with ***Consumer***
- ***Consumers*** in same group divide up and share partitions
- ***Each Consumer groups gets a copy of the same data (really has a unique set of offset partition pairs per Consumer Group)***

? Kafka Consumer Review

- How did we demonstrate Consumers in a Consumer Group dividing up topic partitions and sharing them?
- How did we demonstrate Consumers in different Consumer Groups each getting their own offsets?
- How many records does poll get?
- Does a call to poll ever get records from two different partitions?

Related Content

[Creating a Kafka Consumer in Java]

[Creating a Kafka Producer in Java]

[Kafka from the command line]

[Kafka clustering and failover basics]

[Kafka Architecture]

[What is Kafka?]

[Kafka Topic Architecture]

[Kafka Consumer Architecture]

[Kafka Producer Architecture]

[Kafka and Schema Registry]

[Kafka and Avro]

Kafka Low Level Architecture

Kafka Low-Level Design

Design discussion of Kafka low level design

Kafka Architecture: Low-Level Design

Kafka Design Motivation Goals

- Kafka built to support real-time analytics
  - Designed to feed analytics system that did real-time processing of streams
  - Unified platform for real-time handling of streaming data feeds
- Goals:
  - high-throughput streaming data platform
  - supports high-volume event streams like log aggregation, user activity, etc.

Kafka Design Motivation Scale

- To scale Kafka is
  - distributed,
  - supports sharding
    - load balancing
- Scaling needs inspired Kafka partitioning and consumer model
- Kafka scales writes and reads with partitioned, distributed, commit logs

Kafka Design Motivation Use Cases

- Also designed to support these Use Cases
  - Handle periodic large data loads from offline systems
  - Handle traditional messaging use-cases, low-latency.
- Like MOMs, Kafka is fault-tolerance for node failures through replication and leadership election
- Design more like a distributed database transaction log
- Unlike MOMs, replication, scale not afterthought

Persistence: Embrace filesystem

- Kafka relies heavily on filesystem for storing and caching messages/records
- Disk performance of hard drives performance of sequential writes is fast
  - JBOD with six 7200rpm SATA RAID-5 array clocks at 600MB/sec
  - Heavily optimized by operating systems
- Ton of cache: Operating systems use available of main memory for disk caching
- JVM GC overhead is high for caching objects OS file caches are almost free
- Kafka greatly simplifies code for cache coherence by using OS page cache
- Kafka disk does sequential reads easily optimized by OS page cache

Big fast HDDs and long sequential access

- Like Cassandra, LevelDB, RocksDB, and others, Kafka uses long sequential disk access for read and writes
- Kafka uses tombstones instead of deleting records right away
- Modern Disks have somewhat unlimited space and are fast
- Kafka can provide features not usually found in a messaging system like holding on to old messages for a really long time
  - This flexibility allows for interesting application of Kafka

Kafka Record Retention Redux

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based - configurable based on size
  - Compaction - keeps latest record
- Kafka uses Topic ***Partitions***
- Partitions are broken down into ***Segment*** files

Broker Log Config

Kafka Broker Config for Logs

NAME	DESCRIPTION	DEFAULT
log.dir	Log Directory will topic logs will be stored use this or log.dirs.	/tmp/kafka-logs
log.dirs	The directories where the Topics logs are kept used for JBOD.	
log.flush.interval.messages	Accumulated messages count on a log partition before messages are flushed to disk.	9,223,372,036,854,780,000
log.flush.interval.ms	Maximum time that a topic message is kept in memory before flushed to disk. If not set, uses log.flush.scheduler.interval.ms.	
log.flush.offset.checkpoint.interval.ms	Interval to flush log recovery point.	60,000
log.flush.scheduler.interval.ms	Interval that topic messages are periodically flushed from memory to log.	9,223,372,036,854,780,000

Broker Log Retention Config

Kafka Broker Config for Logs

log.retention.bytes	Delete log records by size. The maximum size of the log before deleting its older records.	long	-1
log.retention.hours	Delete log records by time hours. Hours to keep a log file before deleting older records (in hours), tertiary to log.retention.ms property.	int	168
log.retention.minutes	Delete log records by time minutes. Minutes to keep a log file before deleting it, secondary to log.retention.ms property. If not set, use log.retention.hours is used.	int	null
log.retention.ms	Delete log records by time milliseconds. Milliseconds to keep a log file before deleting it, If not set, use log.retention.minutes.	long	null

Broker Log Segment File Config

Kafka Broker Config - Log Segments

NAME	DESCRIPTION	TYPE	DEFAULT
log.roll.hours	Time period before rolling a new topic log segment. (secondary to log.roll.ms property)	int	168
log.roll.ms	Time period in milliseconds before rolling a new log segment. If not set, uses log.roll.hours.	long	
log.segment.bytes	The maximum size of a single log segment file.	int	1,073,741,824
log.segment.delete.delay.ms	Time period to wait before deleting a segment file from the filesystem.	long	60,000

Kafka Producer Load Balancing

- Producer sends records directly to Kafka broker partition leader
- Producer asks Kafka broker for metadata about which Kafka broker has which topic partitions leaders - thus no routing layer needed
- Producer client controls which partition it publishes messages to
- Partitioning can be done by key, round-robin or using a custom semantic partitioner

Kafka Producer Record Batching

- Kafka producers support record batching. by the size of records and auto-flushed based on time
- Batching is good for network IO throughput.
- Batching speeds up throughput drastically.
- Buffering is configurable
  - lets you make a tradeoff between additional latency for better throughput.
- Producer sends multiple records at a time which equates to fewer IO requests instead of lots of one by one sends

QBit a microservice library uses message batching in an identical fashion as K
to send messages over WebSocket between nodes and from client to QBit se

More producer settings for performance

KafkaExample.java

```
KafkaExample
21      private static Producer<Long, String> createProducer() {
22          Properties props = new Properties();
23          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
24          props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
25          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
26          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(
27
28          //The batch.size in bytes of record size, 0 disables batching
29          props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);
30
31          //Linger how much to wait for other records before sending the batch over the network.
32          props.put(ProducerConfig.LINGER_MS_CONFIG, 20);
33
34          // The total bytes of memory the producer can use to buffer records waiting to be sent
35          // to the Kafka broker. If records are sent faster than broker can handle than
36          // the producer blocks. Used for compression and in-flight records.
37          props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67_108_864);
38
39          //Control how much time Producer blocks before throwing BufferExhaustedException.
40          props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 1000);
```

For higher throughput, Kafka Producer allows buffering based on time and size.

Multiple records can be sent as a batches with fewer network requests.

Speeds up throughput drastically.

Kafka Compression

- Kafka provides ***End-to-end Batch Compression***
- Bottleneck is not always CPU or disk but often network bandwidth
  - especially in cloud, containerized and virtualized environments
  - especially when talking datacenter to datacenter or WAN
- Instead of compressing records one at a time, compresses whole batch
- Message batches can be compressed and sent to Kafka broker/server in one go
- Message batch get written in compressed form in log partition
  - don’t get decompressed until they consumer
- GZIP, Snappy and LZ4 compression protocols supported

Read more at Kafka documents on end to end compression

Kafka Compression Config

Kafka Broker Compression Config

compression.type	Configures compression type for topics. Can be set to codecs ‘gzip', 'snappy', ‘lz4' or ‘uncompressed’. If set to 'producer' then it retains compression codec set by the producer (so it does not have to be uncompressed and then recompressed).	Default: producer

Pull vs. Push/Streams: Pull

- With Kafka consumers ***pull*** data from brokers
- Other systems are push based or stream data to consumers
- Messaging is usually a pull-based system (SQS, most MOM is pull)
  - if consumer fall behind, it catches up later when it can
- Pull-based can implement aggressive batching of data
- Pull based systems usually implement some sort of ***long poll***
  - long poll keeps a connection open for response after a request for a period
- Pull based systems have to pull data and then process it
  - There is always a pause between the pull

Pull vs. Push/Streams: Push

- Push based push data to consumers (scribe, flume, reactive streams, RxJava, Akka)
  - push-based have problems dealing with slow or dead consumers
  - push system consumer can get overwhelmed
  - push based systems use back-off protocol (back pressure)
    - consumer can indicate it is overwhelmed, (http://www.reactive-streams.org/)
- Push-based streaming system can
  - send a request immediately or accumulate request and send in batches
- Push-based systems are always pushing data or streaming data
  - Advantage: Consumer can accumulate data while it is processing data already sent
  - Disadvantage: If consumer dies, how does broker know and when does data get resent to another consumer (harder to manage message acks; more complex)

MOM Consumer Message State

- With most MOM it is brokers responsibility to keep track of which messages have been consumed
- As message is consumed by a consumer, broker keeps track
  - broker may delete data quickly after consumption
- Trickier than it sounds (acknowledgement feature), lots of state to track per message, sent, acknowledge

Kafka Consumer Message State

- Kafka topic is divided into ordered partitions - A topic partition gets read by only one ***consumer*** per ***consumer group***
- Offset data is not tracked per message - ***a lot less data to track***
  - just stores offset of each ***consumer group, partition pairs***
  - Consumer sends offset Data periodically to Kafka Broker
  - Message acknowledgement is cheap compared to MOM
- Consumer can rewind to older offset (replay)
  - If bug then fix, rewind consumer and replay

Message Delivery Semantics

- At most once
  - Messages may be lost but are never redelivered
- At least once
  - Messages are never lost but may be redelivered
- Exactly once
  - this is what people actually want, each message is delivered once and only once

Consumer: Message Delivery Semantics

- "at-most-once" - Consumer reads message, save offset, process message
  - Problem: consumer process dies after saving position but before processing message - consumer takes over starts at last position and message never processed
- “at-least-once" - Consumer reads message, process messages, saves offset
  - Problem: consumer could crash after processing message but before saving position - consumer takes over receives already processed message
- “exactly once” - need a two-phase commit for consumer position, and message process output - or, store consumer message process output in same location as last position
- Kafka offers the first two and you can implement the third

Kafka Producer Acknowledgement

- Kafka's offers operational predictable semantics
- When publishing a message, message get ***committed*** to the log
  - Durable as long as at least one replica lives
- If Producer connection goes down during of send
  - Producer not sure if message sent; resends until message sent ack received (log could have duplicates)
  - Important: use message keys, idempotent messages
  - Not guaranteed to not duplicate from producer retry

Producer Durability Levels

- Producer can specify durability level
- Producer can wait on a message being committed. Waiting for commit ensures all replicas have a copy of the message
- Producer can send with no acknowledgments (0)
- Producer can send with acknowledgment from partition leader (1)
- Producer can send and wait on acknowledgments from all replicas (-1) (default)
- As of June 2017: producer can ensure a message or group of messages was sent "exactly once"

Improved Producer (coming soon)

- New feature:
  - exactly once delivery from producer, atomic write across partitions, (coming soon),
  - producer sends sequence id, broker keeps track if producer already sent this sequence
  - if producer tries to send it again, it gets ack for duplicate, but nothing is save to log
  - NO API changed

Coming Soon: Kafka Atomic Log Writes

New Producer API for transactions

```
producer.initTransaction();

try {
  producer.beginTransaction();
  producer.send(debitAccountMessage);
  producer.send(creditOtherAccountMessage);
  producer.sentOffsetsToTxn(...);
  producer.commitTransaction();
} catch (ProducerFencedTransactionException pfte) {
  ...
  producer.close();
} catch (KafkaException ke) {
  ...
  producer.abortTransaction();
}
```

Consumer only see committed logs

Marker written to log to sign what has been successful transacted

Transaction coordinator and transaction log maintain stat

New producer API for transactions

Kafka Replication

- Kafka replicates each topic's partitions across a configurable number of Kafka brokers
- Kafka is replicated by default not a bolt-on feature
- Each topic partition has one leader and zero or more followers
  - leaders and followers are called replicas
  - replication factor = 1 leader + N followers
- Reads and writes always go to leader
- Partition leadership is evenly shared among Kafka brokers
  - logs on followers are in-sync to leader's log - identical copy - sans un-replicated offsets
- Followers pull records in batches records from leader like a regular Kafka consumer

Kafka Broker Failover

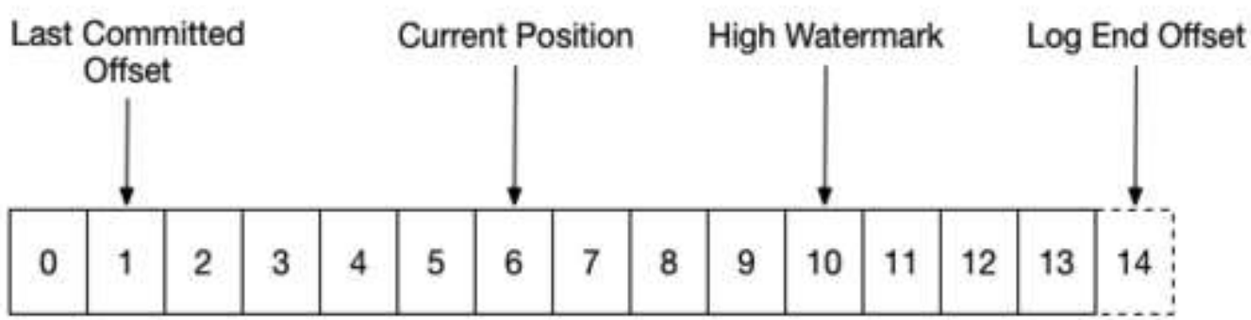
- Kafka keeps track of which Kafka Brokers are alive (in-sync)
  - To be alive Kafka Broker must maintain a ZooKeeper session (heart beat)
  - Followers must replicate writes from leader and not fall "too far" behind
- Each leader keeps track of set of "in sync replicas" aka ISRs
- If ISR/follower dies, falls behind, leader will removes follower from ISR set - falling behind ***replica.lag.time.max.ms > lag***
- Kafka guarantee: committed message not lost, as long as one live ISR - "committed" when written to all ISRs logs
- Consumer only reads committed messages

Replicated Log Partitions

- A Kafka partition is a replicated log - replicated log is a distributed data system primitive
- Replicated log useful for building distributed systems using state-machines
- A replicated log models “coming into consensus” on ordered series of values
  - While leader stays alive, all followers just need to copy values and ordering from leader
- When leader does die, a new leader is chosen from its in-sync followers
- If producer told a message is committed, and then leader fails, new elected leader must have that committed message
- More ISRs; more to elect during a leadership failure

Kafka Consumer Replication Redux

- What can be consumed?
- ***"Log end offset"*** is offset of last record written to log partition and where ***Producers*** write to next
- ***"High watermark"*** is offset of last record successfully replicated to all partitions followers
- ***Consumer*** only reads up to “high watermark”. ***Consumer can’t read un-replicated data***



Kafka Broker Replication Config

Kafka Broker Config

NAME	DESCRIPTION	TYPE	DEFAULT
auto.leader.rebalance.enable	Enables auto leader balancing.	boolean	TRUE
leader.imbalance.check.interval.seconds	The interval for checking for partition leadership balancing.	long	300
leader.imbalance.per.broker.percentage	Leadership imbalance for each broker. If imbalance is too high then a rebalance is triggered.	int	10
min.insync.replicas	When a producer sets acks to all (or -1), This setting is the minimum replicas count that must acknowledge a write for the write to be considered successful. If not met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).	int	1
num.replica.fetchers	Replica fetcher count. Used to replicate messages from a broker that has a leadership partition. Increase this if followers are falling behind.	int	1

Kafka Replication Broker Config 2

Kafka Broker Config

NAME	DESCRIPTION
replica.high.watermark.checkpoint.interval.ms	The frequency with which the high watermark is saved out to disk used for knowing what consumers can consume. ***Consumer*** only reads up to "high watermark". ***Consumer can't read un-replicated data.***
replica.lag.time.max.ms	Determines which Replicas are in the ISR set and which are not. ISR is is important for acks and quorum.
replica.socket.receive.buffer.bytes	The socket receive buffer for network requests
replica.socket.timeout.ms	The socket timeout for network requests. Its value should be at least replica.fetch.wait.max.ms
unclean.leader.election.enable	What happens if all of the nodes go down? Indicates whether to enable replicas not in the ISR. Replicas that are not in-sync. Set to be elected as leader as a last resort, even though doing so may result in data loss. Availability over Consistency. True is the default.

Kafka and Quorum

- Quorum is number of acknowledgements required and number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap
- Most systems use a majority vote - Kafka does not use a majority vote
- Leaders are selected based on having the most complete log
- Problem with majority vote Quorum is it does not take many failure to have inoperable cluster

Kafka and Quorum 2

- If we have a replication factor of 3
  - Then at least two ISRs must be in-sync before the leader declares a sent message committed
  - If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages
  - Among the followers there must be at least one replica that contains all committed messages

Kafka Quorum Majority of ISRs

- Kafka maintains a set of ISRs
- Only this set of ISRs are eligible for leadership election
- Write to partition is not committed until all ISRs ack write
- ISRs persisted to ZooKeeper whenever ISR set changes

Kafka Quorum Majority of ISRs 2

- Any replica that is member of ISRs are eligible to be elected leader
- Allows producers to keep working with out majority nodes
- Allows a replica to rejoin ISR set
  - must fully re-sync again
  - even if replica lost un-flushed data during crash

All nodes die at same time. Now what?

- Kafka's guarantee about data loss is only valid if at least one replica being in-sync
- If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid.
- If all replicas are down for a partition, Kafka chooses first replica (not necessarily in ISR set) that comes alive as the leader
  - Config ***unclean.leader.election.enable=true*** is default
- If ***unclean.leader.election.enable=false, i***f all replicas are down for a partition, Kafka waits for the ISR member that comes alive as new leader.

Producer Durability Acks

- Producers can choose durability by setting ***acks*** to - 0, 1 or all replicas
- acks=all is ***default***, acks happens when all current in-sync replicas (ISR) have received the message
- If durability over availability is prefer
  - Disable unclean leader election
  - Specify a minimum ISR size
    - trade-off between consistency and availability
    - higher minimum ISR size guarantees better consistency
    - but higher minimum ISR reduces availability since partition won't be unavailable for writes if size of ISR set is less than threshold

Quotas

- Kafka has quotas for Consumers and Producers
- Limits bandwidth they are allowed to consume
- Prevents Consumer or Producer from hogging up all Broker resources
- Quota is by client id or user
- Data is stored in ZooKeeper; changes do not necessitate restarting Kafka

? Kafka Low-Level Review

- How would you prevent a denial of service attack from a poorly written consumer?
- What is the default producer durability (acks) level?
- What happens by default if all of the Kafka nodes go down at once?
- Why is Kafka record batching important?
- What are some of the design goals for Kafka?
- What are some of the new features in Kafka as of June 2017?
- What are the different message delivery semantics?

Kafka Log Compaction Architecture

Kafka Log Compaction

Design discussion of Kafka Log Compaction

Kafka Architecture: Log Compaction

Kafka Log Compaction Overview

- Recall Kafka can delete older records based on
  - time period
  - size of a log
- Kafka also supports log compaction for record key compaction
- Log compaction: keep latest version of record and delete older versions

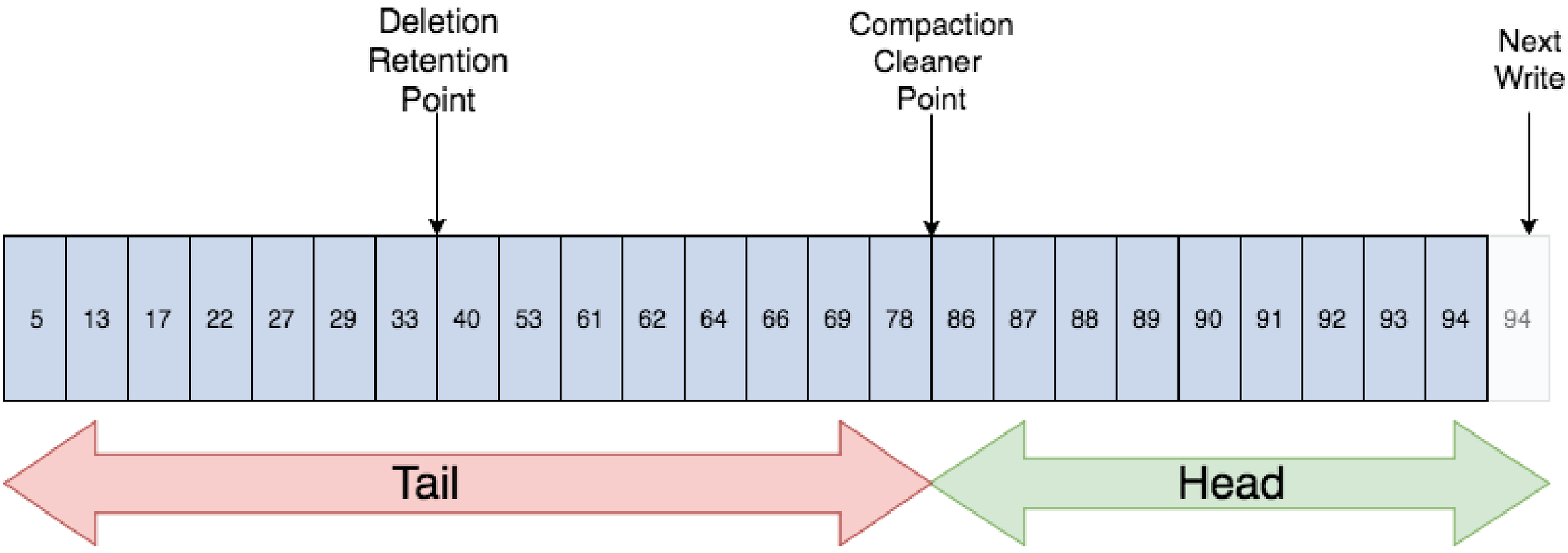
Log Compaction

- Log compaction retains last known value for each record key
- Useful for restoring state after a crash or system failure, e.g., in-memory service, persistent data store, reloading a cache
- Data streams is to log changes to keyed, mutable data,
  - e.g., changes to a database table, changes to object in in-memory microservice
- Topic log has full snapshot of final values for every key - not just recently changed keys
- Downstream consumers can restore state from a log compacted topic

Log Compaction Structure

- Log has head and tail
- Head of compacted log identical to a traditional Kafka log
- New records get appended to the head
- Log compaction works at tail of the log
- Tail gets compacted
- Records in tail of log retain their original offset when written after compaction

Compaction Tail/Head



Log Compaction Basics

- All offsets remain valid, even if record at offset has been compacted away (next highest offset)
- Compaction also allows for deletes. A message with a key and a null payload acts like a tombstone (a delete marker for that key)
  - Tombstones get cleared after a period.
- Log compaction periodically runs in background by recopying log segments.
- Compaction does not block reads and can be throttled to avoid impacting I/O of producers and consumers

Log Compaction Cleaning

Before Compaction

Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

Cleaning

Only keeps latest version of key. Older duplicates not needed.

Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

After Compaction

Log Compaction Guarantees

- If consumer stays caught up to head of the log, it sees every record that is written.
  - Topic config ***min.compaction.lag.ms*** used to guarantee minimum period that must pass before message can be compacted.
- Consumer sees all tombstones as long as the consumer reaches head of log in a period less than the topic config ***delete.retention.ms*** (the default is 24 hours).
- Compaction will never re-order messages, just remove some.
- Offset for a message never changes.
- Any consumer reading from start of the log, sees at least final state of all records in order they were written

Log Cleaner

- Log cleaner does log compaction.
  - Has a pool of background compaction threads that recopy log segments, removing records whose key appears in head of log
- Each compaction thread works as follows:
  - Chooses topic log that has highest ratio: log head to log tail
  - Recopies log from start to end removes records whose keys occur later
- As log partition segments cleaned, they get swapped into log partition
  - Additional disk space required: only one log partition segment
  - not whole partition

Topic Config for Log Compaction

- To turn on compaction for a topic
  - topic config ***log.cleanup.policy=compact***
- To start compacting records after they are written
  - topic config ***log.cleaner.min.compaction.lag.ms***
  - Records wont be compacted until after this period

Broker Config for Log Compaction

Kafka Broker Log Compaction Config

NAME	DESCRIPTION	TYPE	DEFAULT
log.cleaner.backoff.ms	Sleep period when no logs need cleaning	long	15,000
log.cleaner.dedupe.buffer.size	The total memory for log dedupe process for all cleaner threads	long	134,217,728
log.cleaner.delete.retention.ms	How long record delete markers (tombstones) are retained.	long	86,400,000
log.cleaner.enable	Turn on the Log Cleaner. You should turn this on if any topics are using clean.policy=compact.	boolean	TRUE
log.cleaner.io.buffer.size	Total memory used for log cleaner I/O buffers for all cleaner threads	int	524,288
log.cleaner.io.max.bytes.per.second	This is a way to throttle the log cleaner if it is taking up too much time.	double	1.7976931348623157E308
log.cleaner.min.cleanable.ratio	The minimum ratio of dirty head log to total log (head and tail) for a log to get selected for cleaning.	double	0.5
log.cleaner.min.compaction.lag.ms	Minimum time period a new message will remain uncompacted in the log.	long	0
log.cleaner.threads	Threads count used for log cleaning. Increase this if you have a lot of log compaction going on across many topic log partitions.	int	1
log.cleanup.policy	The default cleanup policy for segment files that are beyond their retention window. Valid policies are: "delete" and “compact”. You could use log compaction just for older segment files. instead of deleting them, you could just compact them.	list	[delete]

? Log Compaction Review

- What are three ways Kafka can delete records?
- What is log compaction good for?
- What is the structure of a compacted log? Describe the structure.
- After compaction, do log record offsets change?
- What is a partition segment?

References

- **Learning Apache Kafka**, Second Edition 2nd Edition by Nishant Garg (Author), 2015, ISBN 978-1784393090, Packet Press
- ***Apache Kafka Cookbook***, 1st Edition, Kindle Edition by Saurabh Minni (Author), 2015, ISBN 978-1785882449, Packet Press
- Kafka Streams for Stream processing: A few words about how Kafka works, Serban Balamaci, 2017, ***Blog: Plain Ol' Java***
- Kafka official documentation, 2017
- Why we need Kafka? Quora
- Why is Kafka Popular? Quora
- Why is Kafka so Fast? Stackoverflow
- Kafka growth exploding (Tech Republic)

Working with Kafka Producers

Kafka Producer Advanced

Working with producers in Java

Details and advanced topics

Objectives Create Producer

- Cover advanced topics regarding Java Kafka Consumers
- Custom Serializers
- Custom Partitioners
- Batching
- Compression
- Retries and Timeouts

Kafka Producer

- Kafka client that publishes records to Kafka cluster
- Thread safe
- Producer has pool of buffer that holds to-be-sent records
  - background I/O threads turning records into request bytes and transmit requests to Kafka
- Close producer so producer will not leak resources

Kafka Producer Send, Acks and Buffers

- ***send***() method is asynchronous
  - adds the record to output buffer and return right away
  - buffer used to batch records for efficiency IO and compression
- acks config controls Producer record durability. ”all" setting ensures full commit of record, and is most durable and least fast setting
- Producer can retry failed requests
- Producer has buffers of unsent records per topic partition (sized at ***batch.size***)

Kafka Producer: Buffering and batching

- Kafka Producer buffers are available to send immediately as fast as broker can keep up (limited by inflight ***max.in.flight.requests.per.connection***)
- To reduce requests count, set ***linger.ms*** > 0
  - wait up to linger.ms before sending or until batch fills up whichever comes first
  - Under heavy load linger.ms not met, under light producer load used to increase broker IO throughput and increase compression
- ***buffer.memory*** controls total memory available to producer for buffering
  - If records sent faster than they can be transmitted to Kafka then this buffer gets exceeded then additional send calls block. If period blocks (max.block.ms) after then Producer throws a TimeoutException

Producer Acks

- Producer Config property ***acks***
  - ***(default all)***
- Write Acknowledgment received count required from partition leader before write request deemed complete
- Controls ***Producer*** sent records durability
- Can be all (-1), none (0), or leader (1)

Acks 0 (NONE)

- acks=0
- Producer does not wait for any ack from broker at all
- Records added to the socket buffer are considered sent
- No guarantees of durability - maybe
- Record Offset returned is set to -1 (unknown)
- Record loss if leader is down
- Use Case: maybe log aggregation

Acks 1 (LEADER)

- acks=1
- Partition leader wrote record to its local log but responds without followers confirmed writes
- If leader fails right after sending ack, record could be lost
  - Followers might have not replicated the record
- Record loss is rare but possible
- Use Case: log aggregation

Acks -1 (ALL)

- acks=all or acks=-1
- Leader gets write confirmation from full set of ISRs before sending ack to producer
- Guarantees record not be lost as long as one ISR remains alive
- Strongest available guarantee
- Even stronger with broker setting ***min.insync.replicas*** (specifies the minimum number of ISRs that must acknowledge a write)
- Most Use Cases will use this and set a ***min.insync.replicas > 1***

KafkaProducer config Acks

StockPriceKafkaProducer.java

```
StockPriceKafkaProducer
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Stc
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27
28          //Set number of acknowledgments - acks - default is all
29          props.put(ProducerConfig.ACKS_CONFIG, "all");
30
31          return new KafkaProducer<>(props);
32      }
```

Producer Buffer Memory Size

- Producer config property: ***buffer.memory***
  - ***default 32MB***
  - Total memory (bytes) producer can use to buffer records to be sent to broker
  - Producer blocks up to ***max.block.ms*** if ***buffer.memory*** is exceeded
    - if it is sending faster than the broker can receive, exception is thrown

Batching by Size

- Producer config property: ***batch.size***
  - Default 16K
- Producer batch records
  - fewer requests for multiple records sent to same partition
  - Improved IO throughput and performance on both producer and server
- If record is larger than the batch size, it will not be batched
- Producer sends requests containing multiple batches
  - batch per partition
- Small batch size reduce throughput and performance. If batch size is too big, memory allocated for batch is wasted

Batching by Time and Size - 1

- ***Producer*** config property: ***linger.ms***
  - Default 0
    - Producer groups together any records that arrive before they can be sent into a batch
      - good if records arrive faster than they can be sent out
- ***Producer*** can reduce requests count even under moderate load using ***linger.ms***

Batching by Time and Size - 2

- ***linger.ms*** adds delay to wait for more records to build up so larger batches are sent
  - ***good brokers throughput*** at ***cost of producer latency***
- If ***producer*** gets records who size is ***batch.size*** or more for a broker's leader partitions, then it is sent right away
- If ***Producers*** gets less than ***batch.size*** but ***linger.ms*** interval has passed, then records for that partition are sent
- Increase to improve throughput of Brokers and reduce broker load (common improvement)

Compressing Batches

- ***Producer*** config property: ***compression.type***
  - Default 0
- Producer compresses request data
- By default producer does not compress
- Can be set to none, gzip, snappy, or lz4
- Compression is by batch
  - improves with larger batch sizes
- End to end compression possible if Broker config “compression.type” set to producer. Compressed data from producer sent to log and consumer by broker

Batching and Compression Example

```
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
```

```
57
58      private static void setupBatchingAndCompression(Properties props) {
59          //Wait up to 50 ms to batch to Kafka - linger.ms
60          props.put(ProducerConfig.LINGER_MS_CONFIG, 200);
61
62          //Holds up to 64K per partition default is 16K - batch.size
63          props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);
64
65          //Holds up to 64 MB default is 32MB for all partition buffers
66          // - "buffer.memory"
67          props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33_554_432 * 2);
68
69          //Set compression type to snappy - compression.type
70          props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
71      }
```

Custom Serializers

- You don’t have to use built in serializers
- You can write your own
- Just need to be able to convert to/fro a byte[]
- Serializers work for keys and values
- ***value.serializer*** and ***key.serializer***

Custom Serializers Config

```
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19
```

```
30
31      private static void setupBootstrapAndSerializers(Properties props) {
32          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
33                  StockAppConstants.BOOTSTRAP_SERVERS);
34          props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
35          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
36                  StringSerializer.class.getName());
37
38          //Custom Serializer - config "value.serializer"
39          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
40                  StockPriceSerializer.class.getName());
41      }
```

Custom Serializer

StockPriceSerializer.java

```
1   package com.cloudurable.kafka.producer;
2
3   import com.cloudurable.kafka.producer.model.StockPrice;
4   import org.apache.kafka.common.serialization.Serializer;
5
6   import java.nio.charset.StandardCharsets;
7   import java.util.Map;
8
9   public class StockPriceSerializer implements Serializer<StockPrice> {
10
11      @Override
12      public byte[] serialize(String topic, StockPrice data) {
13          return data.toJson().getBytes(StandardCharsets.UTF_8);
14      }
15
16      @Override
17      public void configure(Map<String, ?> configs, boolean isKey) {
18      }
19
20      @Override
21      public void close() {
22      }
23  }
24
```

StockPrice


StockPrice.java
StockPrice
1 package com.cloudurable.kafka.producer.model;
2
3 import io.advantageous.boon.json.JsonFactory;
4
5 public class StockPrice {
6
7 private final int dollars;
8 private final int cents;
9 private final String name;
10
11 public String toJson() {
12 return "{" +
13 "\"dollars\": " + dollars +
14 ", \"cents\": " + cents +
15 ", \"name\": \"" + name + '\"' +
16 '}';
17 }
18

Broker Follower Write Timeout

- ***Producer*** config property: ***request.timeout.ms***
  - Default 30 seconds (30,000 ms)
- Maximum time broker waits for confirmation from followers to meet Producer acknowledgment requirements for ***ack=all***
- Measure of broker to broker latency of request
- 30 seconds is high, long process time is indicative of problems

Producer Request Timeout

- ***Producer*** config property: ***request.timeout.ms***
  - Default 30 seconds (30,000 ms)
  - Maximum time producer waits for request to complete to broker
  - Measure of producer to broker latency of request
  - 30 seconds is very high, long request time is an indicator that brokers can’t handle load

Producer Retries

- Producer config property: ***retries***
  - ***Default 0***
- Retry count if ***Producer*** does not get ack from Broker
  - only if record send fail deemed a transient error (API)
  - as if your producer code resent record on failed attempt
- timeouts are retried, ***retry.backoff.ms*** (default to 100 ms) to wait after failure before ***retry***

Retry, Timeout, Back-off Example

```
13
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
20          setupRetriesInFlightTimeout(props);
```

```
40
41      private static void setupRetriesInFlightTimeout(Properties props) {
42
43          //Only two in-flight messages per Kafka broker connection
44          // - max.in.flight.requests.per.connection (default 5)
45          props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
46                  1);
47
48          //Set the number of retries - retries
49          props.put(ProducerConfig.RETRIES_CONFIG, 2);
50
51          //Request timeout - request.timeout.ms
52          props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);
53
54          //Only retry after one second.
55          props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
56      }
57
```

Producer Partitioning

- ***Producer*** config property: ***partitioner.class***
  - org.apache.kafka.clients.producer.internals.DefaultPartitioner
  - Partitioner class implements Partitioner interface
  - Default Partitioner partitions using hash of key if record has key
  - Default Partitioner partitions uses round-robin if record has no key

Configuring Partitioner

StockPriceKafkaProducer.java

StockPriceKafkaProducer | createProducer()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install partitioner -- "partitioner.class"
28          props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
29                  StockPricePartitioner.class.getName());
30
31          props.put("importantStocks", "IBM,UBER");
```

StockPricePartitioner

StockPriceKafkaProducer.java × StockPricePartitioner.java ×

StockPricePartitioner | StockPricePartitioner()

```
1   package com.cloudurable.kafka.producer;
2
3   import org.apache.kafka.clients.producer.Partitioner;
4   import org.apache.kafka.common.Cluster;
5   import org.apache.kafka.common.PartitionInfo;
6
7   import java.util.*;
8
9   public class StockPricePartitioner implements Partitioner{
10
11      private final Set<String> importantStocks;
12      public StockPricePartitioner() { importantStocks = new HashSet<>(); }
15
16      @Override
17      public int partition(final String topic,
18                           final Object objectKey,
19                           byte[] keyBytes, final Object value,
20                           final byte[] valueBytes,
21                           final Cluster cluster) {...}
38
39      @Override
40      public void close() {
41      }
42
43      @Override
44      public void configure(Map<String, ?> configs) {...}
48  }
```

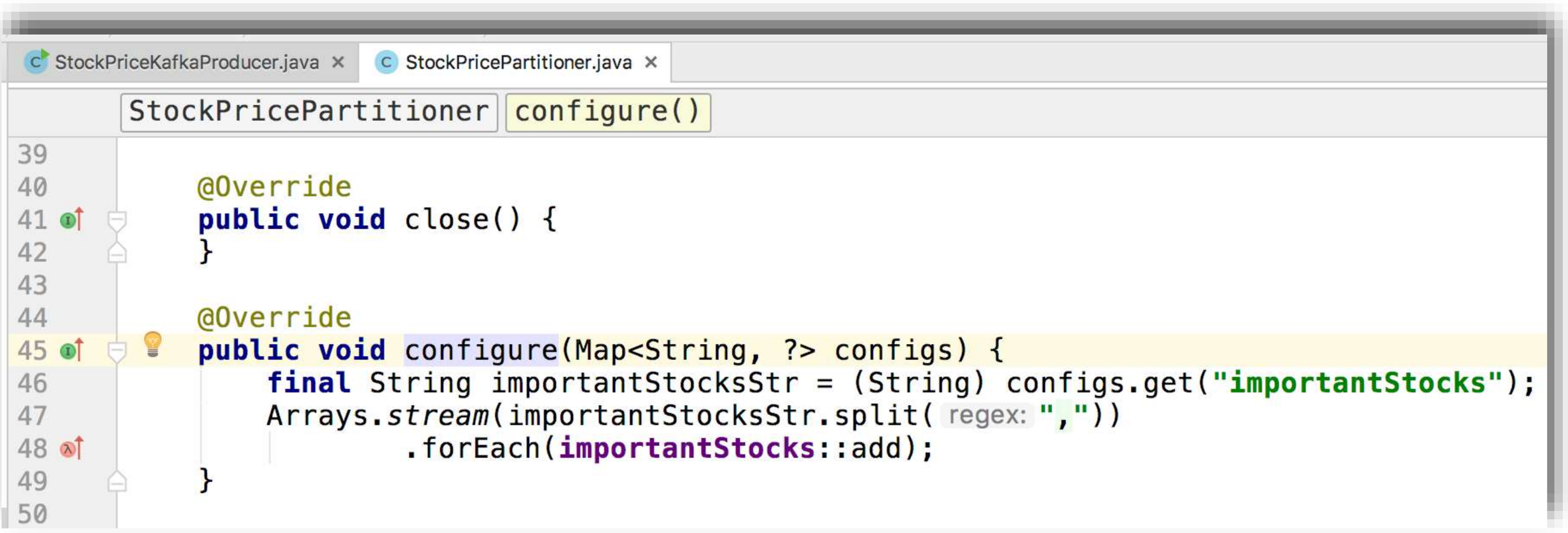
StockPricePartitioner partition()

StockPriceKafkaProducer.java × | StockPricePartitioner.java ×

StockPricePartitioner | partition()

```
17      public int partition(final String topic,
18                           final Object objectKey,
19                           final byte[] keyBytes,
20                           final Object value,
21                           final byte[] valueBytes,
22                           final Cluster cluster) {
23
24          final List<PartitionInfo> partitionInfoList =
25                  cluster.availablePartitionsForTopic(topic);
26          final int partitionCount = partitionInfoList.size();
27          final int importantPartition = partitionCount -1;
28          final int normalPartitionCount = partitionCount -1;
29
30          final String key = ((String) objectKey);
31
32          if (importantStocks.contains(key)) {
33              return importantPartition;
34          } else {
35              return Math.abs(key.hashCode()) % normalPartitionCount;
36          }
37
38      }
```

StockPricePartitioner



Producer Interception

- ***Producer*** config property: ***interceptor.classes***
  - empty (you can pass an comma delimited list)
  - interceptors implementing ProducerInterceptor interface
  - intercept records producer sent to broker and after acks
  - you could mutate records

KafkaProducer - Interceptor Config

StockPriceKafkaProducer.java

StockPriceKafkaProducer | getStockSenderList()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Stc
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install interceptor list - config "interceptor.classes"
28          props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
29                  StockProducerInterceptor.class.getName());
30
31          //Set number of acknowledgments - acks - default is all
32          props.put(ProducerConfig.ACKS_CONFIG, "all");
33
34          return new KafkaProducer<>(props);
35      }
```

KafkaProducer ProducerInterceptor

StockProducerInterceptor.java

```
StockProducerInterceptor
10
11  public class StockProducerInterceptor implements ProducerInterceptor {
12
13      private final Logger logger = LoggerFactory
14              .getLogger(StockProducerInterceptor.class);
15      private int onSendCount;
16      private int onAckCount;
17
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {...}
36
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {...}
54
55      @Override
56      public void close() {...}
59
60      @Override
61      public void configure(Map<String, ?> configs) {...}
64  }
```

ProducerInterceptor onSend

StockProducerInterceptor.java

StockProducerInterceptor

```
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {
20          onSendCount++;
21          if (logger.isDebugEnabled()) {
22              logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
23                      record.topic(), record.key(), record.value().toString(),
24                      record.partition()
25              ));
26          } else {
27              if (onSendCount % 100 == 0) {
28                  logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
29                          record.topic(), record.key(), record.value().toString(),
30                          record.partition()
31                  ));
32              }
33          }
34          return record;
35      }
```

Output

```
ic=stock-prices2 key=UBER value=StockPrice{dollars=737, cents=78, name='
```

ProducerInterceptor onAck

StockProducerInterceptor.java

```
StockProducerInterceptor  onAcknowledgement()
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {
40          onAckCount++;
41
42          if (logger.isDebugEnabled()) {
43              logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
44                      metadata.topic(), metadata.partition(), metadata.offset()
45              ));
46          } else {
47              if (onAckCount % 100 == 0) {
48                  logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
49                          metadata.topic(), metadata.partition(), metadata.offset()
50                  ));
51              }
52          }
53      }
```

Output

```
onAck topic=stock-prices2, part=0, offset=18360
```

ProducerInterceptor the rest

StockProducerInterceptor.java

StockProducerInterceptor | configure()

```
48                      logger.info(String.format("onAck topic=%s,
49                              metadata.topic(), metadata.partitio
50                      ));
51                  }
52              }
53          }
54
55          @Override
56          public void close() {
57          }
58
59          @Override
60          public void configure(Map<String, ?> configs) {
61          }
62      }
```

KafkaProducer send() Method

- Two forms of send with callback and with no callback both return Future
  - Asynchronously sends a record to a topic
  - Callback gets invoked when send has been acknowledged.
- send is asynchronous and return right away as soon as record has added to send buffer
- Sending many records at once without blocking for response from Kafka broker
- Result of send is a RecordMetadata
  - record partition, record offset, record timestamp
- Callbacks for records sent to same partition are executed in order

KafkaProducer send() Exceptions

- ***InterruptException*** - If the thread is interrupted while blocked (API)
- ***SerializationException*** - If key or value are not valid objects given configured serializers (API)
- ***TimeoutException*** - If time taken for fetching metadata or allocating memory exceeds max.block.ms, or getting acks from Broker exceed timeout.ms, etc. (API)
- ***KafkaException*** - If Kafka error occurs not in public API. (API)

Using send method

StockSender.java

StockSender > run()

```
51              try {
52
53                  final Future<RecordMetadata> future = producer.send(record);
54
55                  if (sentCount % 100 == 0) {
56                      displayRecordMetaData(record, future);
```

StockSender.java

StockSender > displayRecordMetaData()

```
74      private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
75                                         final Future<RecordMetadata> future)
76                                 throws InterruptedException, ExecutionException {
77          final RecordMetadata recordMetadata = future.get();
78          logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
79                          "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
80                  record.key(),
81                  record.value().toJson(),
82                  recordMetadata.topic(),
83                  recordMetadata.partition(),
84                  recordMetadata.offset(),
85                  new Date(recordMetadata.timestamp())
86                  ));
87      }
```

KafkaProducer flush() method

- flush() method sends all buffered records now (even if ***linger.ms*** > 0)
  - blocks until requests complete
- Useful when consuming from some input system and pushing data into Kafka
- ***flush***() ensures all previously sent messages have been sent
  - you could mark progress as such at completion of flush

KafkaProducer close()

- close() closes producer
  - frees resources (threads and buffers) associated with producer
- Two forms of method
  - both block until all previously sent requests complete or duration passed in as args is exceeded
  - close with no params equivalent to close(Long.MAX_VALUE, TimeUnit.MILLISECONDS).
  - If producer is unable to complete all requests before the timeout expires, all unsent requests fail, and this method fails

Orderly shutdown using close

StockPriceKafkaProducer.java

StockPriceKafkaProducer › main()

```
95          Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97              executorService.shutdown();
98              try {
99                  executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
100                 logger.info("Shutting down executorService for workers nicely");
101             } catch (InterruptedException e) {
102                 logger.warn("shutting down", e);
103             }
104
105             logger.info("Flushing producer");
106             producer.flush();
107             logger.info("Closing producer");
108             producer.close();
109
110             if (!executorService.isShutdown()) {
111                 logger.info("Forcing shutdown of workers");
112                 executorService.shutdownNow();
113             }
114         }));
```

Wait for clean close

StockPriceKafkaProducer.java

StockPriceKafkaProducer › main()

```
95        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97            executorService.shutdown();
98            try {
99                executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
100               logger.info("Flushing and closing producer");
101               producer.flush();
102               producer.close( timeout: 10_000, TimeUnit.MILLISECONDS);
103           } catch (InterruptedException e) {
104               logger.warn("shutting down", e);
105           }
106
107       }));
```

KafkaProducer partitionsFor() method

- partitionsFor(topic) returns meta data for partitions
- `public List<PartitionInfo> partitionsFor(String topic)`
- Get partition metadata for give topic
- Produce that do their own partitioning would use this
  - for custom partitioning
- `PartitionInfo(String topic, int partition, Node leader, Node[] replicas, Node[] inSyncReplicas)`
  - `Node(int id, String host, int port, optional String rack)`

KafkaProducer metrics() method

- metrics() method get map of metrics
- **public** Map<MetricName,? extends Metric> metrics()
- Get the full set of producer metrics

MetricName(

```
String name,
String group,
String description,
Map<String,String> tags
```

)

public interface Metric

A numerical metric tracked for monitoring purposes

Method Summary

Methods

Modifier and Type	Method and Description
MetricName	**metricName()** A name for this metric
double	**value()** The value of the metric

Metrics producer.metrics()

MetricsProducerReporter.java

MetricsProducerReporter

```
25
26          final Map<MetricName, ? extends Metric> metrics = producer.metrics();
27
28          metrics.forEach((metricName, metric) ->
29                  logger.info(
30                          String.format("\nMetric\t %s,\t %s,\t %s, " +
31                                  "\n\t\t%s\n",
32                          metricName.group(),
33                          metricName.name(),
34                          metric.value(),
35                          metricName.description())
36          ));
```

- Call producer.metrics()
- Prints out metrics to log

Metrics producer.metrics() output

```
Metric    producer-metrics,     record-queue-time-max,    508.0,
          The maximum time in ms record batches spent in the record accumulator.

17:09:22.721 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric    producer-node-metrics,    request-rate,    0.02503128911389236,
          The average number of requests sent per second.

17:09:22.721 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric    producer-metrics,     records-per-request-avg,  205.55263157894737,
          The average number of records per request.

17:09:22.722 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric    producer-metrics,     record-size-avg, 71.02631578947368,
          The average record size

17:09:22.722 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric    producer-node-metrics,    request-size-max,    56.0,
          The maximum size of any request sent in the window.

17:09:22.723 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric    producer-metrics,     request-size-max,    12058.0,
          The maximum size of any request sent in the window.

17:09:22.723 [pool-1-thread-9] INFO  c.c.k.p.MetricsProducerReporter -
Metric    producer-metrics,     compression-rate-avg,     0.4144136027285927 3,
          The average compression rate of record batches.
```

Metrics via JMX

```
~/kafka-training
$ jconsole
```