



Exploring Kafka

Table of Content



1. Kafka Overview: 5
2. What is Kafka: 15
3. Kafka Architecture: 26
4. Kafka Versus: 40
5. Kafka Topics: 45
6. Kafka Producers: 56
7. Kafka Consumers: 61
8. Using Kafka Single Node: 73
9. Kafka Cluster and Failover: 85
10. Kafka Ecosystem: 106

Table of Content



11. Intro to Producers: 115
12. Advanced Producers: 131
13. About the App: 179
14. Producer Shutdown: 200
15. Kafka Low Level Design: 205
16. Log Compaction: 244
17. Introduction to Consumers: 256
18. Advanced Consumers: 280
19. Avro and the Schema Registry: 326
20. Security: 356

Table of Content



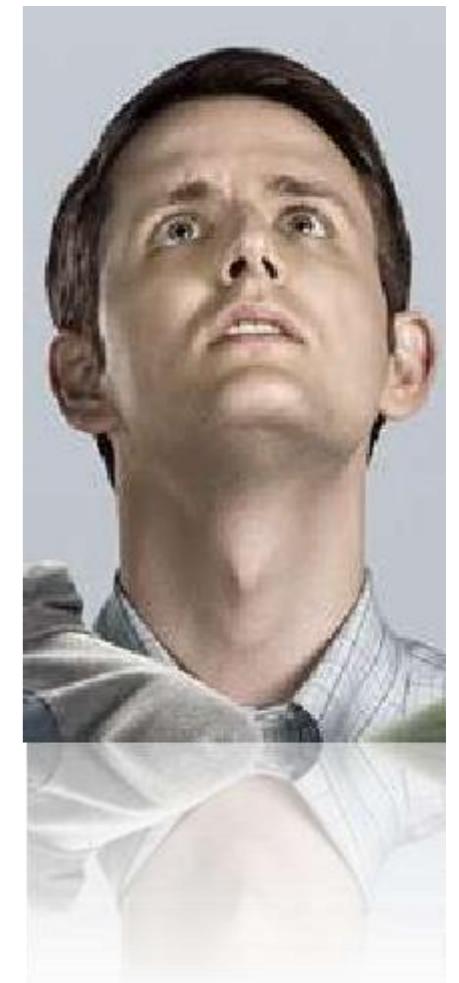
21. SSL : 360
22. SASL: 379
23. SASL Plain: 391
24. SASL Scram: 406
25. Mirror Maker: 415

1. Kafka Overview

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)

4
commas!



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 work 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 Use Cases

- ❖ Stream Processing
- ❖ Microservices + Cassandra
- ❖ 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***
- ❖ ***Twitter***
- ❖ ***Square***
- ❖ 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

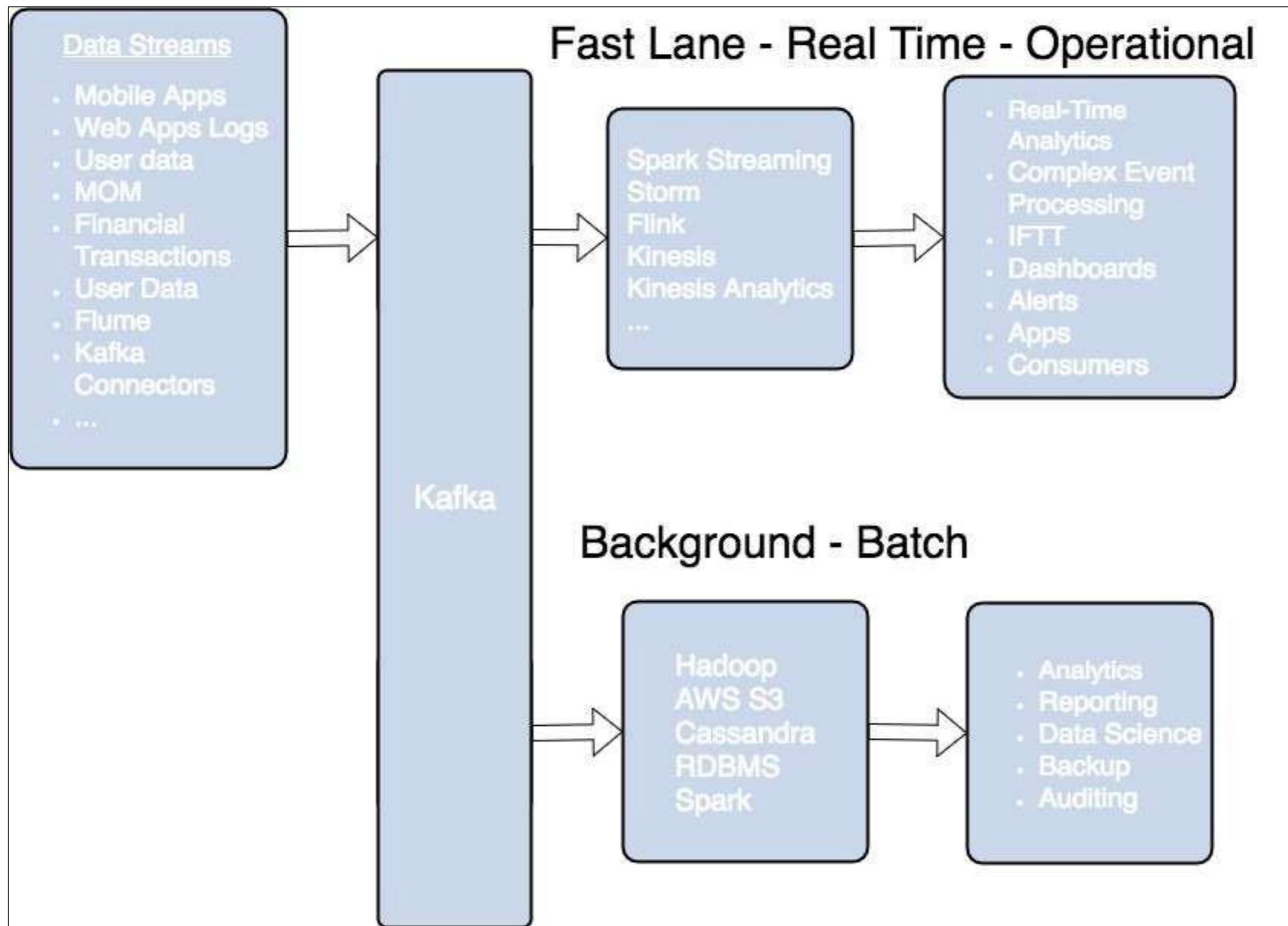
Most important reason: ***Kafka's great performance***: throughput, latency, obtained through great engineering

Why is Kafka so fast?

- ❖ **Zero Copy**
- ❖ **Batch Data in Chunks**
- ❖ **Sequential Disk Writes**
- ❖ **Horizontal Scale**



Kafka Streaming Architecture



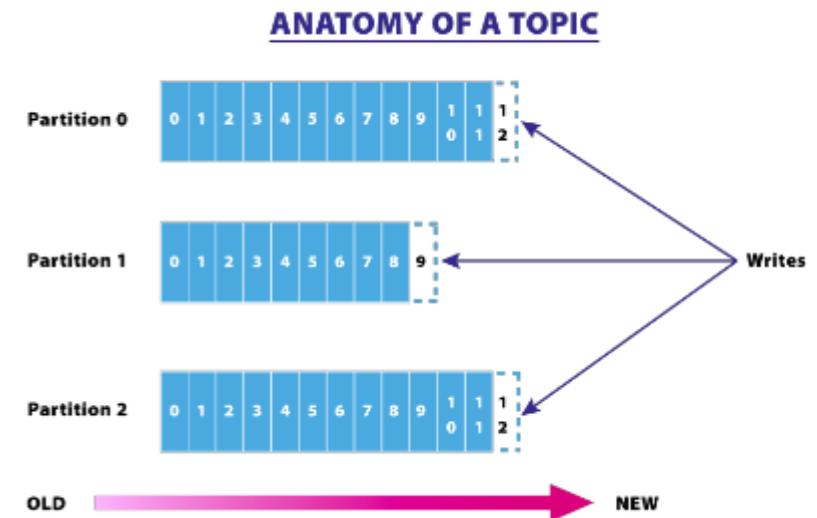


Why Kafka Review

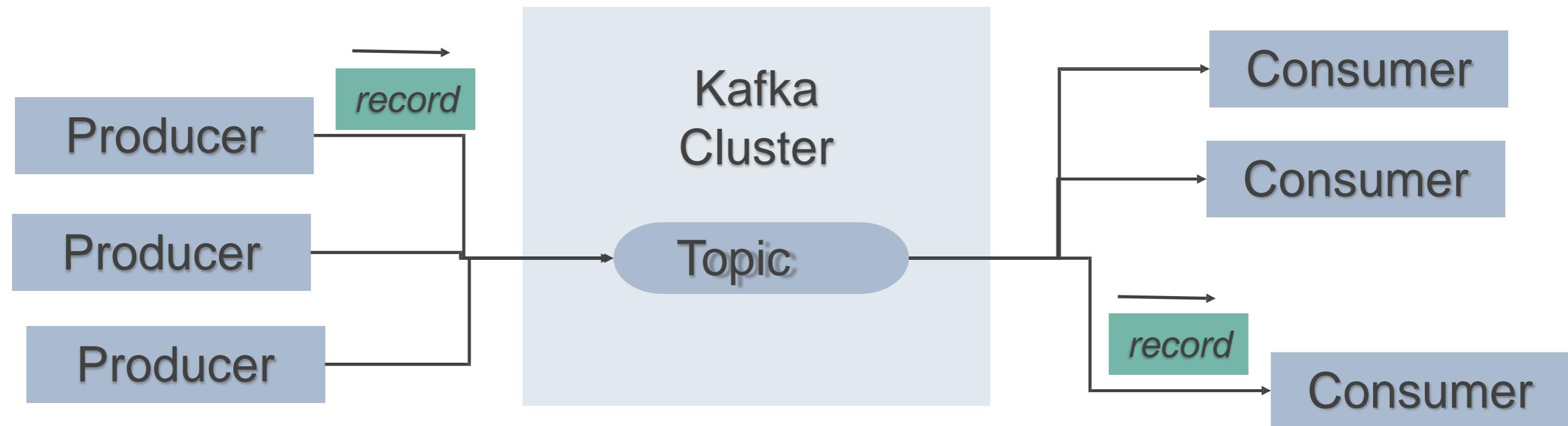
- ❖ Why is Kafka so fast?
- ❖ How fast is Kafka usage growing?
- ❖ How is Kafka getting used?
- ❖ Where does Kafka fit in the Big Data Architecture?
- ❖ How does Kafka relate to real-time analytics?
- ❖ Who uses Kafka?

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



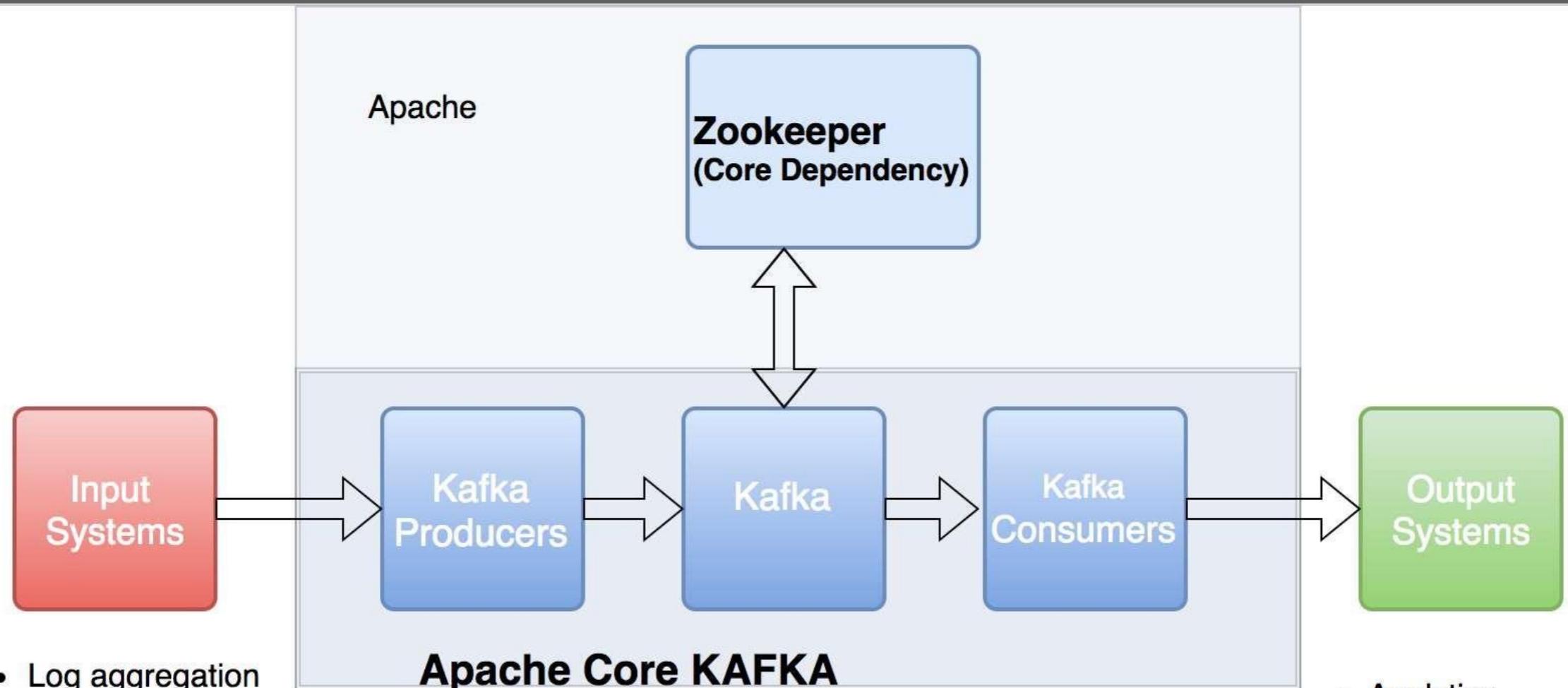
Kafka: Topics, Producers, and Consumers



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 Kafka



- Log aggregation
- Metrics
- KPIs
- Batch imports
- Audit trail
- User activity logs
- Web logs

Not part of core

- Schema Registry
- Avro
- Kafka REST Proxy
- Kafka Connect
- Kafka Streams

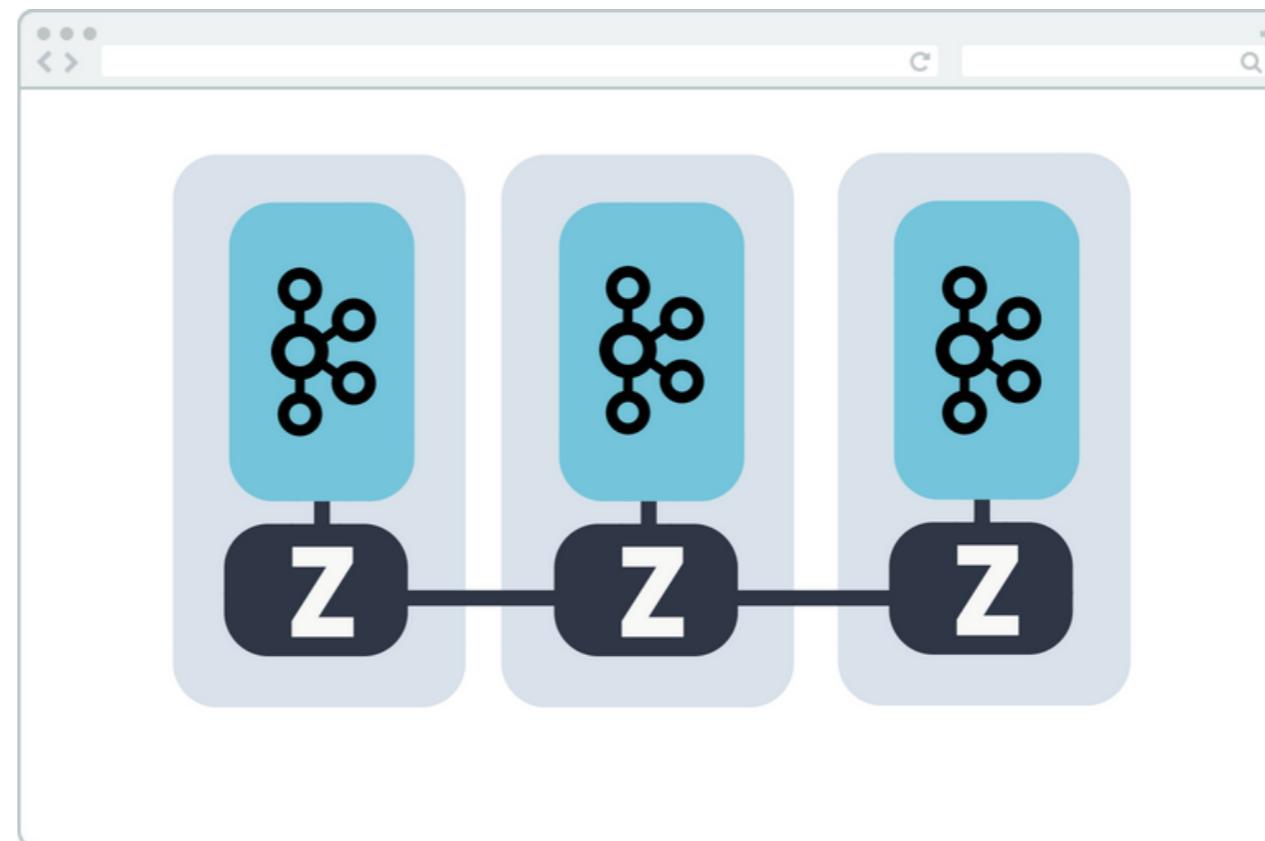
Apache Kafka Core

- Server/Broker
- Scripts to start libs
- Script to start up Zookeeper
- Utils to create topics
- Utils to monitor stats

- Analytics
- Databases
- Machine Learning
- Dashboards
- Indexed for Search
- Business Intelligence

Kafka needs Zookeeper

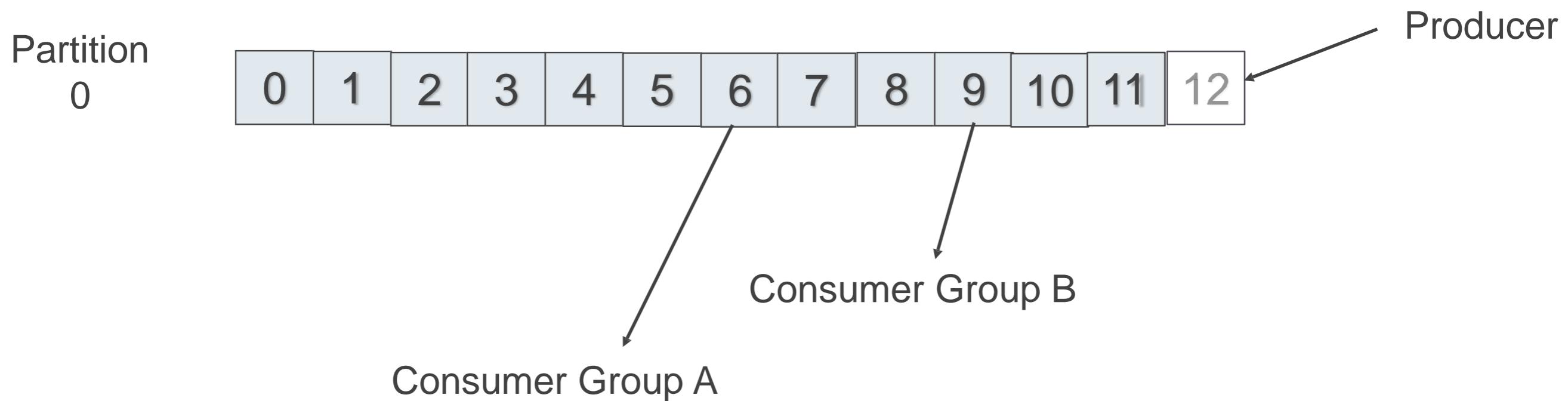
- ❖ Zookeeper helps with leadership election of Kafka Broker and Topic Partition pairs
- ❖ Zookeeper manages service discovery for Kafka Brokers that form the cluster
- ❖ Zookeeper sends changes to Kafka
 - ❖ New Broker join, Broker died, etc.
 - ❖ Topic removed, Topic added, etc.
- ❖ Zookeeper provides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



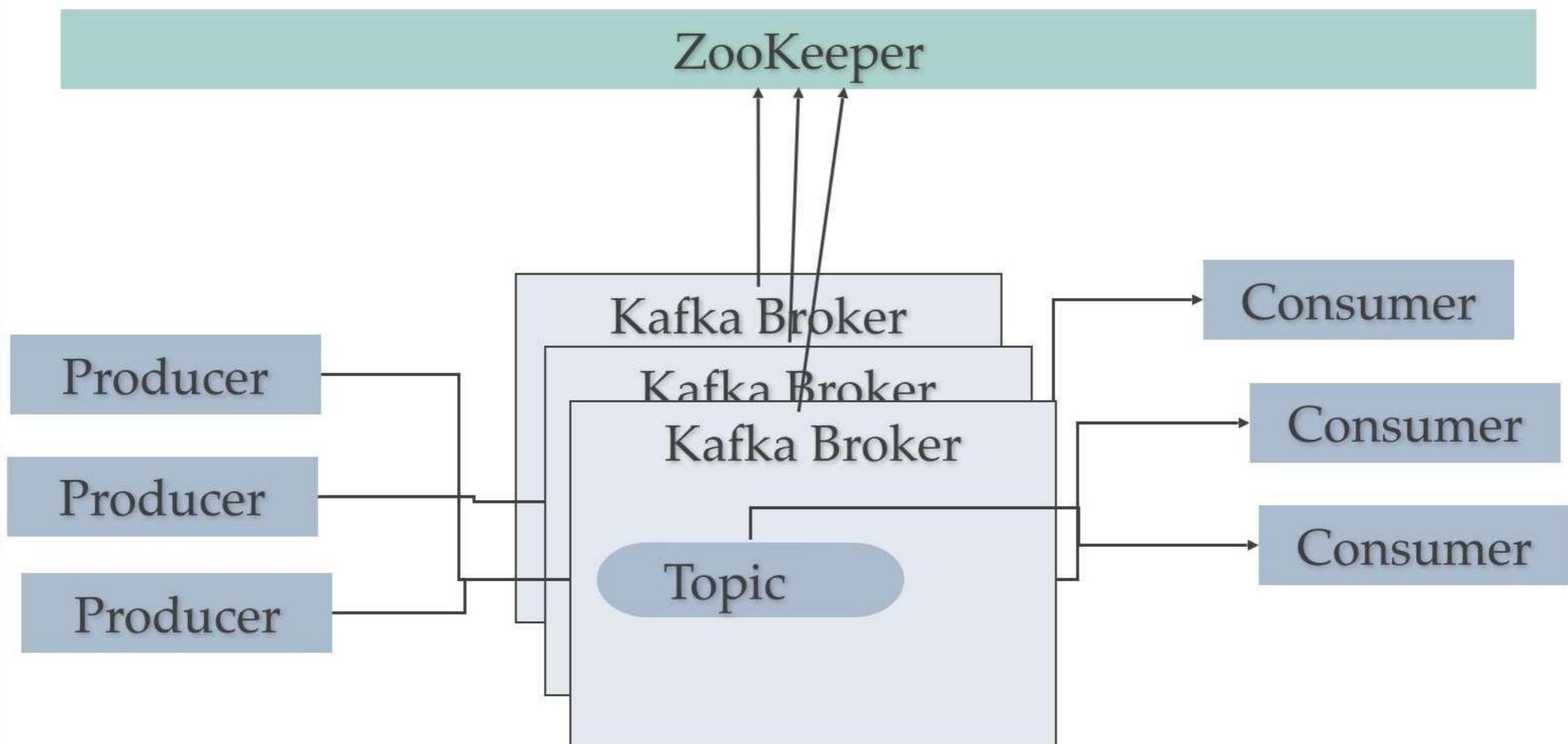
- ❖ Kafka Cluster is made up of multiple Kafka Brokers
- ❖ Each Broker has an ID (number)
- ❖ Brokers contain topic log partitions
- ❖ Connecting to one broker bootstraps client to entire cluster
- ❖ Start with at least three brokers, cluster can have, 10, 100, 1000 brokers if needed

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?

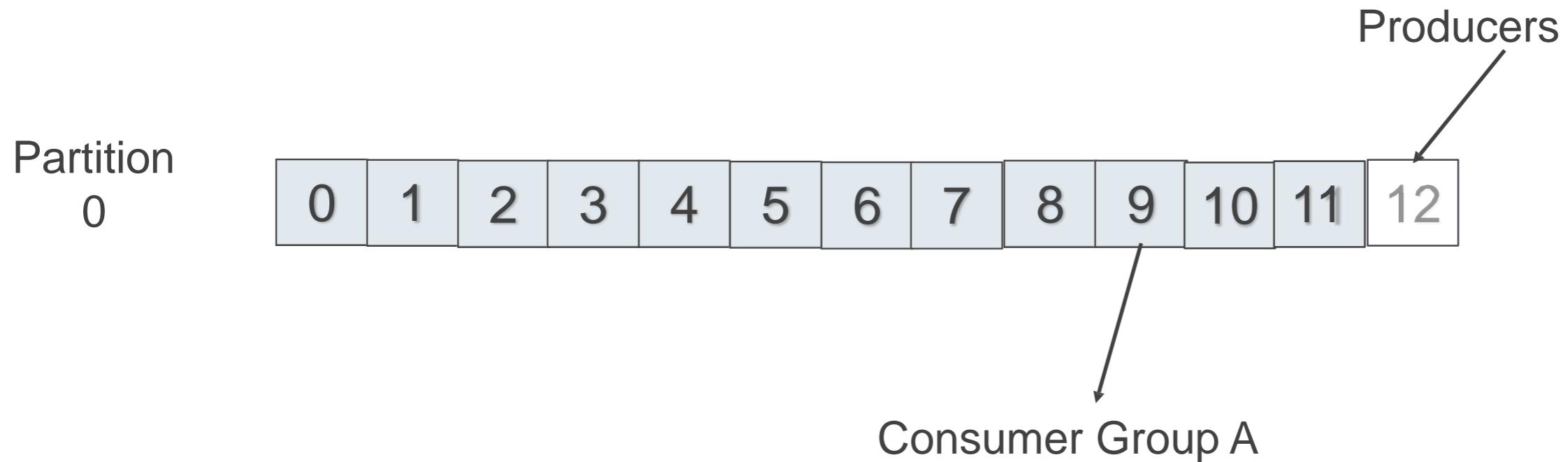
6. Kafka Producers

Kafka Producers



- ❖ **Producers** send records to topics
- ❖ **Producer** picks which partition to send record to per topic
- ❖ 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



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?

7. Kafka Consumers

Kafka Consumer Groups



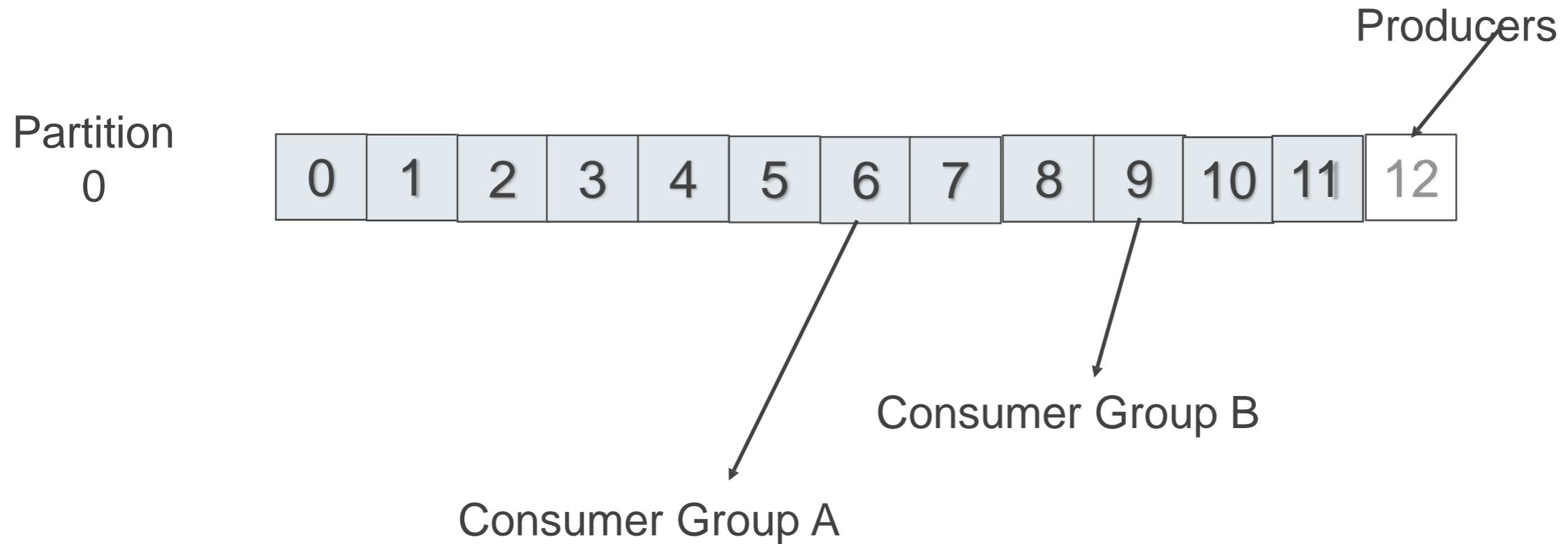
- ❖ Consumers are grouped into a ***Consumer Group***
 - ❖ ***Consumer group*** has a unique id
 - ❖ Each ***consumer group*** is a subscriber
 - ❖ Each ***consumer group*** maintains its own offset
 - ❖ Multiple subscribers = multiple consumer groups
 - ❖ Each has different function: one might delivering 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
- ❖ 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

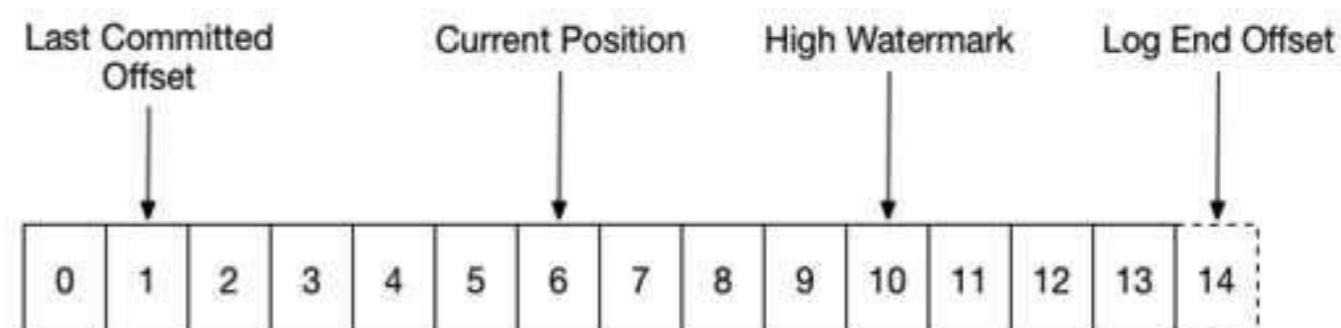


- ❖ Kafka stores offsets in topic called “`__consumer_offset`”
 - ❖ Uses Topic Log Compaction
- ❖ When a consumer has processed data, it should commit offsets
- ❖ If consumer process dies, it will be able to start up and start reading where it left off based on offset stored in “`__consumer_offset`”

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

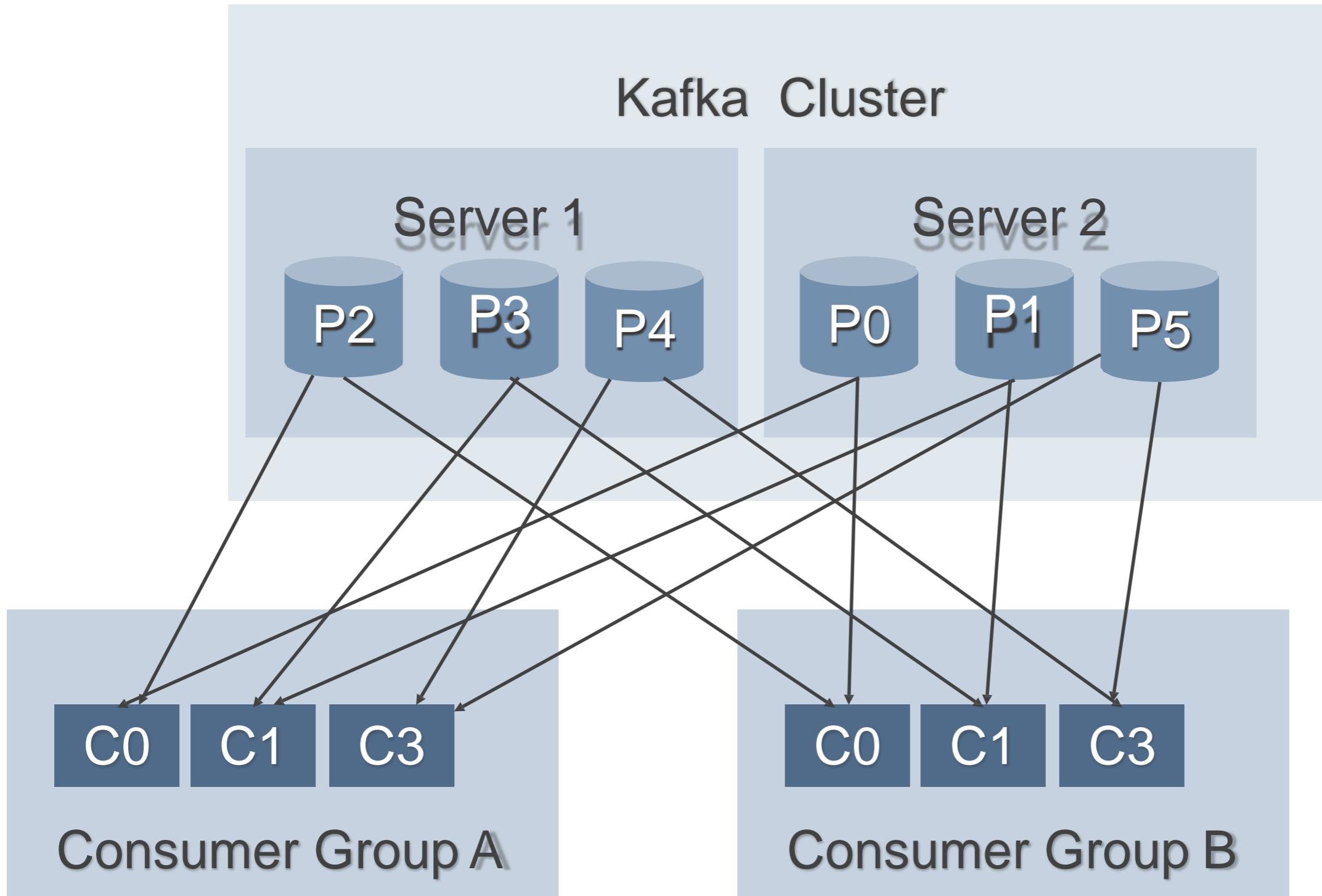


Consumer to Partition Cardinality



- ❖ 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 manage 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?

8. 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 x

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/zookeeper-server-start.sh \
5   kafka/config/zookeeper.properties
6
```

```
$ ./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.DatadirCleanupManager)
[2017-05-13 13:34:52,491] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.DatadirCleanupManager)
[2017-05-13 13:34:52,491] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCleanupManager)
[2017-05-13 13:34:52,491] WARN Either no config or no quorum defined in config, running in stand-alone 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.server.ZooKeeperServer)
```



Run Kafka Server

run-kafka.sh x

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-server-start.sh \
5     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 >
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # Create a topic
6 kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
7 --replication-factor 1 --partitions 13 --topic my-topic
```

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



List Topics

```
> list-topics.sh x
```

```
1 #!/usr/bin/env bash  
2  
3 cd ~/kafka-training  
4  
5 # List existing topics  
6 kafka/bin/kafka-topics.sh --list \  
7 --zookeeper localhost:2181  
8
```

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



Run Kafka Consumer

```
start-consumer-console.sh x
```

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
5 --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.

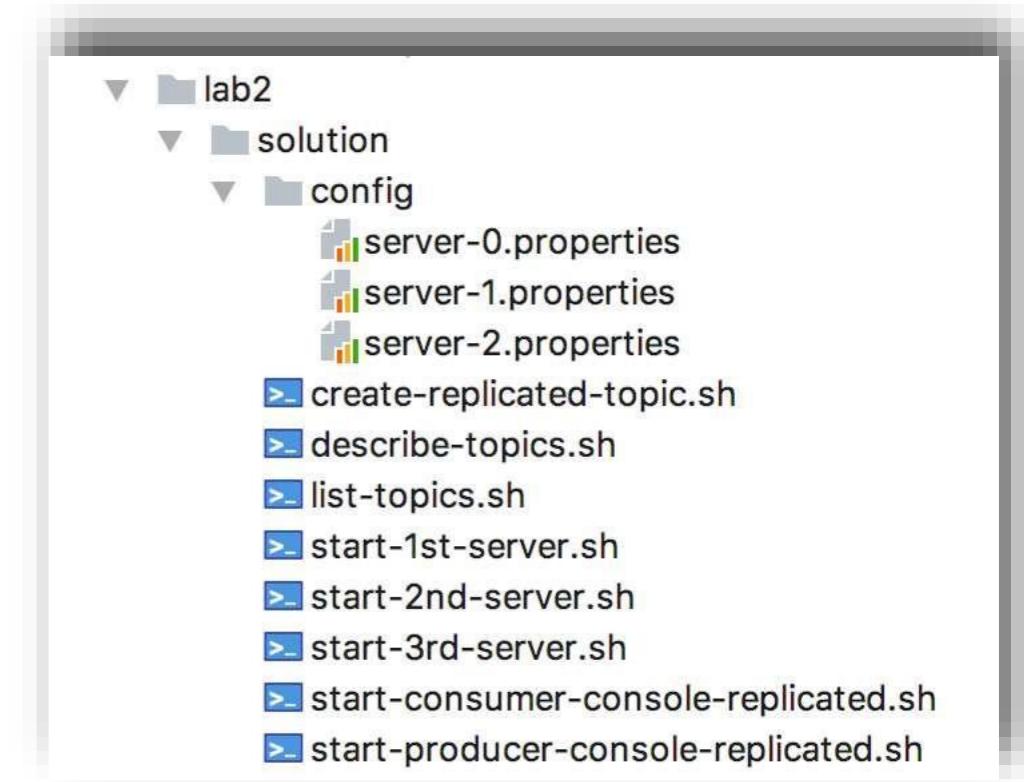
Complete Lab 1

9. Kafka 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
- ❖ 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

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

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 x
1 broker.id=0
2 port=9092
3 log.dirs=./logs/kafka-0
.
.
.
server-1.properties x
1 broker.id=1
2 port=9093
3 log.dirs=./logs/kafka-1
4
.
.
.
server-2.properties x
1 broker.id=2
2 port=9094
3 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 x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3
4 cd ~/kafka-training
5
6 ## Run Kafka
7 kafka/bin/kafka-server-start.sh \
8     "$CONFIG/server-0.properties"
9

start-2nd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka
6 kafka/bin/kafka-server-start.sh \
7     "$CONFIG/server-1.properties"
8

start-3rd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka
6 kafka/bin/kafka-server-start.sh \
7     "$CONFIG/server-2.properties"
8
```



```
start-2nd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka
6 kafka/bin/kafka-server-start.sh \
7     "$CONFIG/server-1.properties"
8
9
```

- ❖ 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 values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
    authorizer.class.name =
```



Create Kafka replicated topic my-failsafe-topic

```
create-replicated-topic.sh >
```

```
1 #!/usr/bin/env bash  
2  
3 cd ~/kafka-training  
4  
5 kafka/bin/kafka-topics.sh --create \  
6   --zookeeper localhost:2181 \  
7   --replication-factor 3 \  
8   --partitions 13 \  
9   --topic my-failsafe-topic  
10
```

- ❖ **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 x
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   --from-beginning
8
```

- ❖ 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!
```

```
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-4
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-consumer-console-replicated.sh
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

A screenshot of a desktop environment showing four terminal windows side-by-side. Each window has a title bar labeled 'solution — java < sta'.

- The top-left window shows the command '\$./start-producer-console-replicated.sh' followed by seven messages: m1, m2, m3, m4, m5, m6, m7.
- The top-right window shows the command '\$./start-consumer-console-replicated.sh' followed by three messages: m3, m5, and an empty line.
- The bottom-left window shows the command '\$./start-consumer-console-replicated.sh' followed by two messages: m2 and m6.
- The bottom-right window shows the command '\$./start-consumer-console-replicated.sh' followed by three messages: m1, m4, and m7.

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



Consumer Failover

```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
m17
m18
m19
m20
m21
m22
m23
m24
m25
m26
m27
m28
m29
m30
m31
m32
m33
m34
m35
m36
m37
m38
m39
m40
m41
m42
m43
m44
m45
m46
m47
m48
m49
m50
m51
m52
m53
m54
m55
m56
m57
m58
m59
m60
m61
m62
m63
m64
m65
m66
m67
m68
m69
m70
m71
m72
m73
m74
m75
m76
m77
m78
m79
m80
m81
m82
m83
m84
m85
m86
m87
m88
m89
m90
m91
m92
m93
m94
m95
m96
m97
m98
m99
m100
m101
m102
m103
m104
m105
m106
m107
m108
m109
m110
m111
m112
m113
m114
m115
m116
m117
m118
m119
m120
m121
m122
m123
m124
m125
m126
m127
m128
m129
m130
m131
m132
m133
m134
m135
m136
m137
m138
m139
m140
m141
m142
m143
m144
m145
m146
m147
m148
m149
m150
m151
m152
m153
m154
m155
m156
m157
m158
m159
m160
m161
m162
m163
m164
m165
m166
m167
m168
m169
m170
m171
m172
m173
m174
m175
m176
m177
m178
m179
m180
m181
m182
m183
m184
m185
m186
m187
m188
m189
m190
m191
m192
m193
m194
m195
m196
m197
m198
m199
m200
m201
m202
m203
m204
m205
m206
m207
m208
m209
m210
m211
m212
m213
m214
m215
m216
m217
m218
m219
m220
m221
m222
m223
m224
m225
m226
m227
m228
m229
m230
m231
m232
m233
m234
m235
m236
m237
m238
m239
m240
m241
m242
m243
m244
m245
m246
m247
m248
m249
m250
m251
m252
m253
m254
m255
m256
m257
m258
m259
m260
m261
m262
m263
m264
m265
m266
m267
m268
m269
m270
m271
m272
m273
m274
m275
m276
m277
m278
m279
m280
m281
m282
m283
m284
m285
m286
m287
m288
m289
m290
m291
m292
m293
m294
m295
m296
m297
m298
m299
m300
m301
m302
m303
m304
m305
m306
m307
m308
m309
m310
m311
m312
m313
m314
m315
m316
m317
m318
m319
m320
m321
m322
m323
m324
m325
m326
m327
m328
m329
m330
m331
m332
m333
m334
m335
m336
m337
m338
m339
m340
m341
m342
m343
m344
m345
m346
m347
m348
m349
m350
m351
m352
m353
m354
m355
m356
m357
m358
m359
m360
m361
m362
m363
m364
m365
m366
m367
m368
m369
m370
m371
m372
m373
m374
m375
m376
m377
m378
m379
m380
m381
m382
m383
m384
m385
m386
m387
m388
m389
m390
m391
m392
m393
m394
m395
m396
m397
m398
m399
m400
m401
m402
m403
m404
m405
m406
m407
m408
m409
m410
m411
m412
m413
m414
m415
m416
m417
m418
m419
m420
m421
m422
m423
m424
m425
m426
m427
m428
m429
m430
m431
m432
m433
m434
m435
m436
m437
m438
m439
m440
m441
m442
m443
m444
m445
m446
m447
m448
m449
m450
m451
m452
m453
m454
m455
m456
m457
m458
m459
m460
m461
m462
m463
m464
m465
m466
m467
m468
m469
m470
m471
m472
m473
m474
m475
m476
m477
m478
m479
m480
m481
m482
m483
m484
m485
m486
m487
m488
m489
m490
m491
m492
m493
m494
m495
m496
m497
m498
m499
m500
m501
m502
m503
m504
m505
m506
m507
m508
m509
m510
m511
m512
m513
m514
m515
m516
m517
m518
m519
m520
m521
m522
m523
m524
m525
m526
m527
m528
m529
m530
m531
m532
m533
m534
m535
m536
m537
m538
m539
m540
m541
m542
m543
m544
m545
m546
m547
m548
m549
m550
m551
m552
m553
m554
m555
m556
m557
m558
m559
m5510
m5511
m5512
m5513
m5514
m5515
m5516
m5517
m5518
m5519
m5520
m5521
m5522
m5523
m5524
m5525
m5526
m5527
m5528
m5529
m55210
m55211
m55212
m55213
m55214
m55215
m55216
m55217
m55218
m55219
m55220
m55221
m55222
m55223
m55224
m55225
m55226
m55227
m55228
m55229
m552210
m552211
m552212
m552213
m552214
m552215
m552216
m552217
m552218
m552219
m552220
m552221
m552222
m552223
m552224
m552225
m552226
m552227
m552228
m552229
m5522210
m5522211
m5522212
m5522213
m5522214
m5522215
m5522216
m5522217
m5522218
m5522219
m5522220
m5522221
m5522222
m5522223
m5522224
m5522225
m5522226
m5522227
m5522228
m5522229
m55222210
m55222211
m55222212
m55222213
m55222214
m55222215
m55222216
m55222217
m55222218
m55222219
m55222220
m55222221
m55222222
m55222223
m55222224
m55222225
m55222226
m55222227
m55222228
m55222229
m552222210
m552222211
m552222212
m552222213
m552222214
m552222215
m552222216
m552222217
m552222218
m552222219
m552222220
m552222221
m552222222
m552222223
m552222224
m552222225
m552222226
m552222227
m552222228
m552222229
m5522222210
m5522222211
m5522222212
m5522222213
m5522222214
m5522222215
m5522222216
m5522222217
m5522222218
m5522222219
m5522222220
m5522222221
m5522222222
m5522222223
m5522222224
m5522222225
m5522222226
m5522222227
m5522222228
m5522222229
m55222222210
m55222222211
m55222222212
m55222222213
m55222222214
m55222222215
m55222222216
m55222222217
m55222222218
m55222222219
m55222222220
m55222222221
m55222222222
m55222222223
m55222222224
m55222222225
m55222222226
m55222222227
m55222222228
m55222222229
m552222222210
m552222222211
m552222222212
m552222222213
m552222222214
m552222222215
m552222222216
m552222222217
m552222222218
m552222222219
m552222222220
m552222222221
m552222222222
m552222222223
m552222222224
m552222222225
m552222222226
m552222222227
m552222222228
m552222222229
m5522222222210
m5522222222211
m5522222222212
m5522222222213
m5522222222214
m5522222222215
m5522222222216
m5522222222217
m5522222222218
m5522222222219
m5522222222220
m5522222222221
m5522222222222
m5522222222223
m5522222222224
m5522222222225
m5522222222226
m5522222222227
m5522222222228
m5522222222229
m55222222222210
m55222222222211
m55222222222212
m55222222222213
m55222222222214
m55222222222215
m55222222222216
m55222222222217
m55222222222218
m55222222222219
m55222222222220
m55222222222221
m55222222222222
m55222222222223
m55222222222224
m55222222222225
m55222222222226
m55222222222227
m55222222222228
m55222222222229
m552222222222210
m552222222222211
m552222222222212
m552222222222213
m552222222222214
m552222222222215
m552222222222216
m552222222222217
m552222222222218
m552222222222219
m552222222222220
m552222222222221
m552222222222222
m552222222222223
m552222222222224
m552222222222225
m552222222222226
m552222222222227
m552222222222228
m552222222222229
m5522222222222210
m5522222222222211
m5522222222222212
m5522222222222213
m5522222222222214
m5522222222222215
m5522222222222216
m5522222222222217
m5522222222222218
m5522222222222219
m5522222222222220
m5522222222222221
m5522222222222222
m5522222222222223
m5522222222222224
m5522222222222225
m5522222222222226
m5522222222222227
m5522222222222228
m5522222222222229
m55222222222222210
m55222222222222211
m55222222222222212
m55222222222222213
m55222222222222214
m55222222222222215
m55222222222222216
m55222222222222217
m55222222222222218
m55222222222222219
m55222222222222220
m55222222222222221
m55222222222222222
m55222222222222223
m55222222222222224
m55222222222222225
m55222222222222226
m55222222222222227
m55222222222222228
m55222222222222229
m552222222222222210
m552222222222222211
m552222222222222212
m552222222222222213
m552222222222222214
m552222222222222215
m552222222222222216
m552222222222222217
m552222222222222218
m552222222222222219
m552222222222222220
m552222222222222221
m552222222222222222
m552222222222222223
m552222222222222224
m552222222222222225
m552222222222222226
m552222222222222227
m552222222222222228
m552222222222222229
m5522222222222222210
m5522222222222222211
m5522222222222222212
m5522222222222222213
m5522222222222222214
m5522222222222222215
m5522222222222222216
m5522222222222222217
m5522222222222222218
m5522222222222222219
m5522222222222222220
m5522222222222222221
m5522222222222222222
m5522222222222222223
m5522222222222222224
m5522222222222222225
m5522222222222222226
m5522222222222222227
m5522222222222222228
m5522222222222222229
m55222222222222222210
m55222222222222222211
m55222222222222222212
m55222222222222222213
m55222222222222222214
m55222222222222222215
m55222222222222222216
m55222222222222222217
m55222222222222222218
m55222222222222222219
m55222222222222222220
m55222222222222222221
m55222222222222222222
m55222222222222222223
m55222222222222222224
m55222222222222222225
m55222222222222222226
m55222222222222222227
m55222222222222222228
m55222222222222222229
m552222222222222222210
m552222222222222222211
m552222222222222222212
m552222222222222222213
m552222222222222222214
m552222222222222222215
m552222222222222222216
m552222222222222222217
m552222222222222222218
m552222222222222222219
m552222222222222222220
m552222222222222222221
m552222222222222222222
m552222222222222222223
m552222222222222222224
m552222222222222222225
m552222222222222222226
m552222222222222222227
m552222222222222222228
m552222222222222222229
m5522222222222222222210
m5522222222222222222211
m5522222222222222222212
m5522222222222222222213
m5522222222222222222214
m5522222222222222222215
m5522222222222222222216
m5522222222222222222217
m5522222222222222222218
m5522222222222222222219
m5522222222222222222220
m5522222222222222222221
m5522222222222222222222
m5522222222222222222223
m5522222222222222222224
m5522222222222222222225
m5522222222222222222226
m5522222222222222222227
m5522222222222222222228
m5522222222222222222229
m55222222222222222222210
m55222222222222222222211
m552222222
```

Create Kafka Describe Topic



```
>_ describe-topics.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic my-failsafe-topic \
8   --zookeeper localhost:2181
9
```

- ❖ —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`
```

use 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



```
~/kafka-training/lab2/solution
$ ./start-producer-console-replicated.sh
m1
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m2
m3
m4
m5
m6
m7
m8
m9
m10
m11
m12
m13
m14
m15
m16
~/kafka-training/lab2/solution
$ ./start-consumer-console-replicated.sh
m3
m5
m8
m9
m11
m14
[2017-05-15 12:00:58,462] WARN Auto-commit
ta='', my-failsafe-topic-3=OffsetAndMetad
ffset=1, metadata=''}, my-failsafe-topic-1
etAndMetadata{offset=1, metadata=''}, my-f
fe-topic-5=OffsetAndMetadata{offset=2, met
triable exception. You should retry commit
Coordinator)
m16
```

- ❖ 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?

Complete Lab 1.2

10. Kafka Ecosystem

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

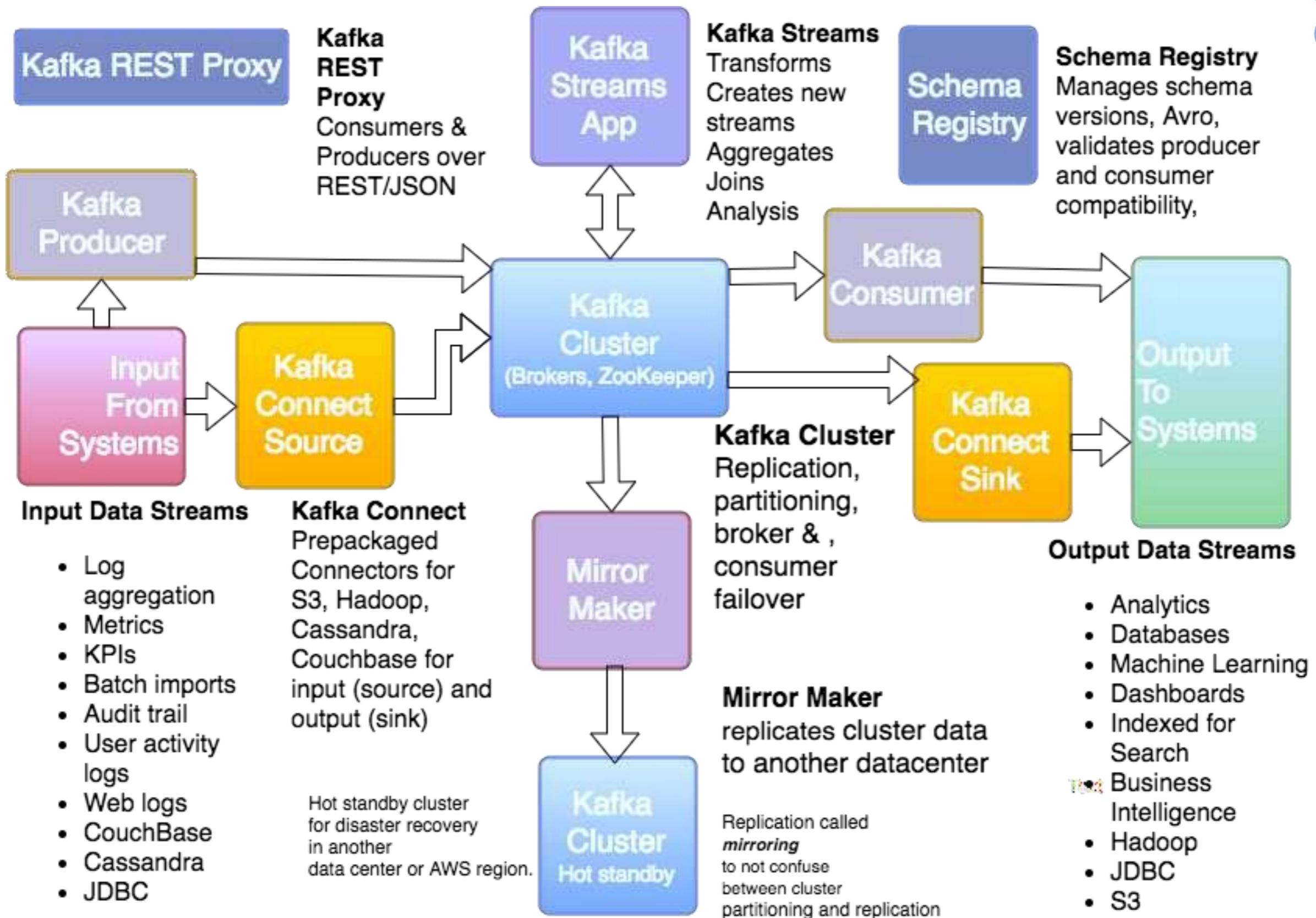
What comes in Kafka Extensions?



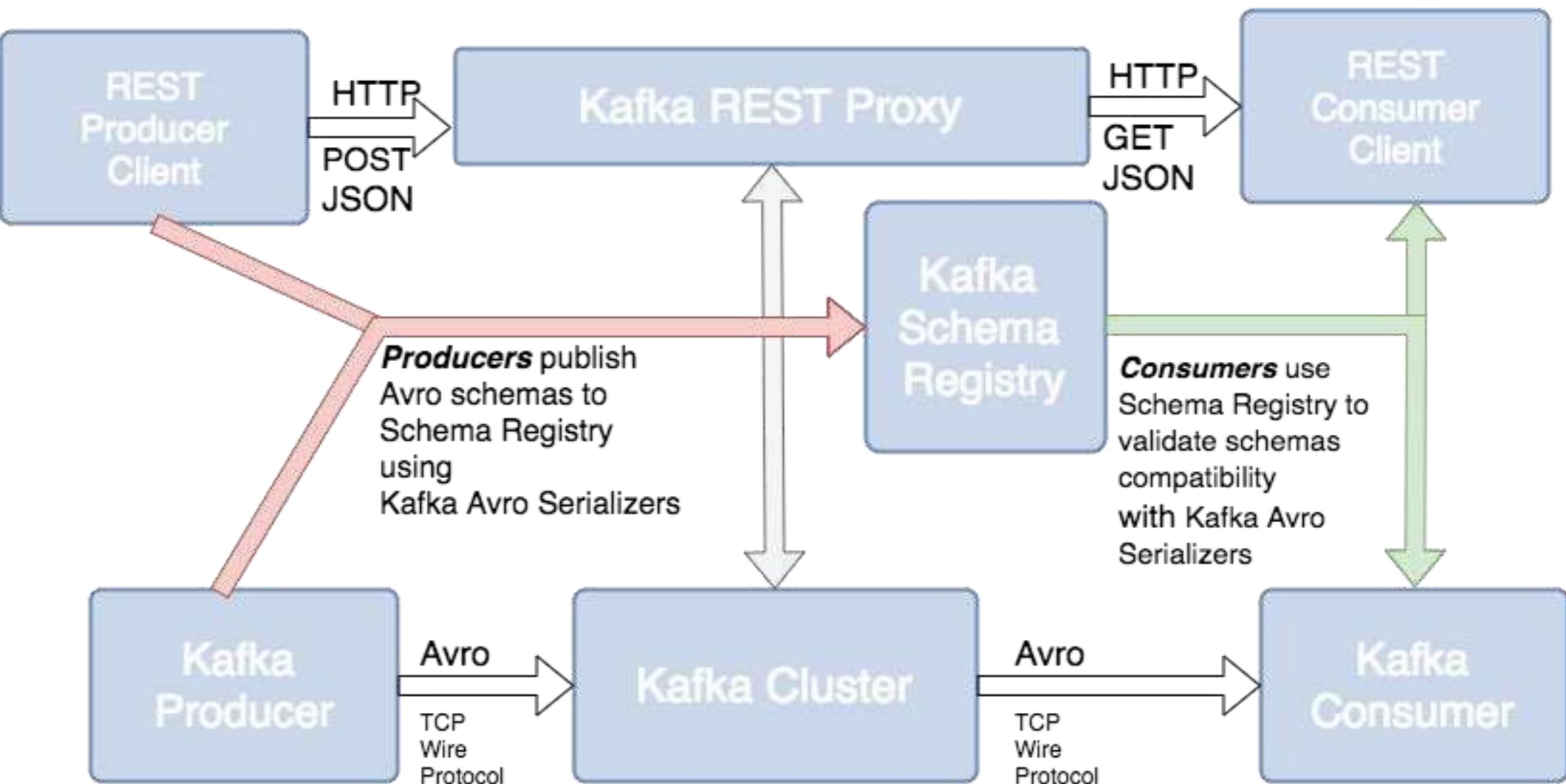
Confluent.io:

- ❖ 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 to download ***but you must register with [Confluent](#)***
- Community of Kafka Connectors from 3rd parties and [Confluent](#)

Kafka Universe



Kafka REST Proxy and Schema Registry

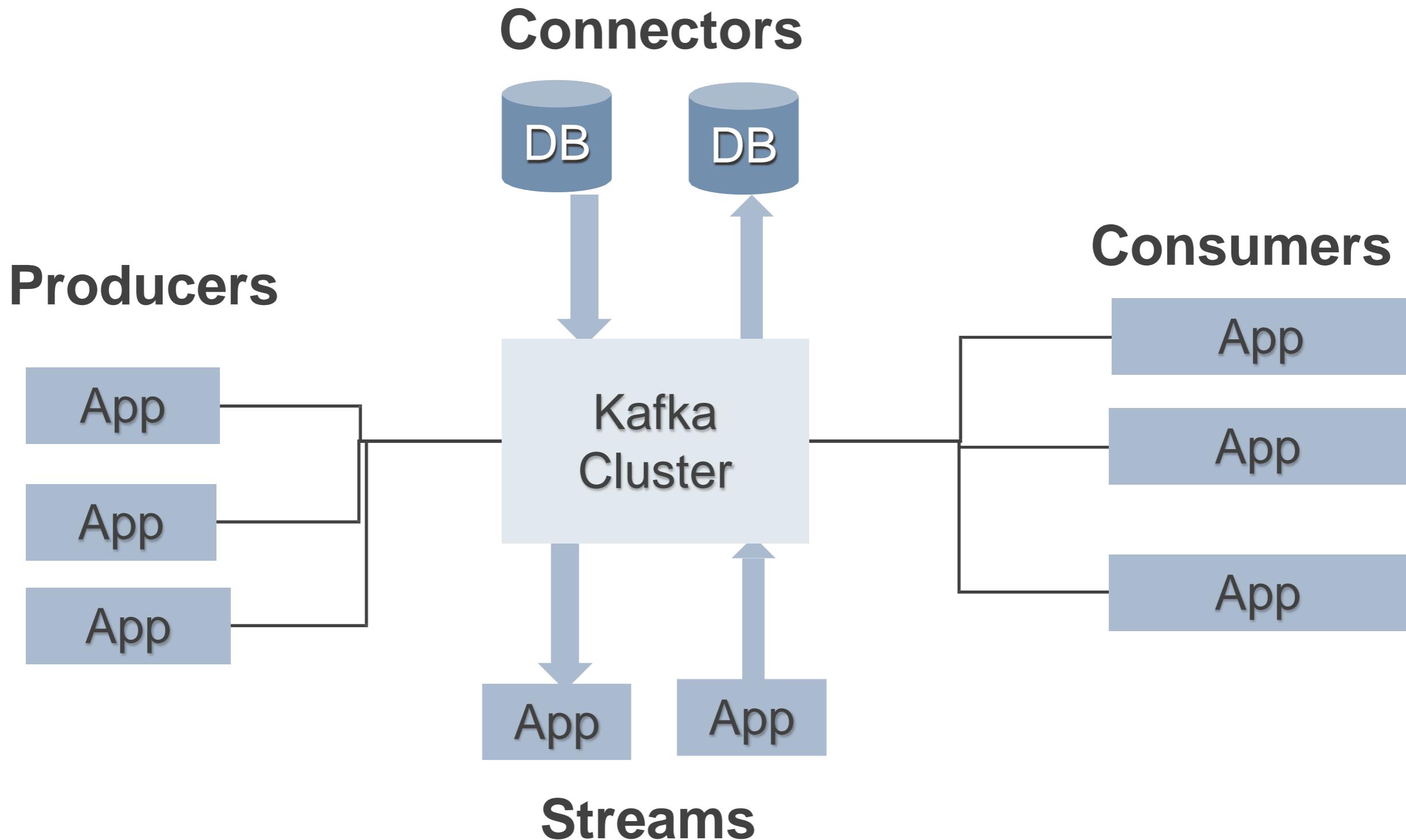


Kafka Stream : Stream Processing



- ❖ **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?

11. Intro to Producers

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 ×

```
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 ## Create topics
5 kafka/bin/kafka-topics.sh --create \
6   --replication-factor 3 \
7   --partitions 13 \
8   --topic my-example-topic \
9   --zookeeper localhost:2181
10
11 ## List created topics
12 kafka/bin/kafka-topics.sh --list \
13   --zookeeper localhost:2181
```

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

Gradle Build script



kafka-training ×

```
1 group 'cloudurable-kafka'  
2 version '1.0-SNAPSHOT'  
3  
4 apply plugin: 'java'  
5  
6 sourceCompatibility = 1.8  
7  
8 repositories {  
9     mavenCentral()  
10 }  
11  
12 dependencies {  
13     compile 'org.apache.kafka:kafka-clients:0.10.2.0'  
14     compile 'ch.qos.logback:logback-classic:1.2.2'  
15 }
```

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



```
c KafkaProducerExample.java x
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";
14 }
```

Create Kafka Producer to send records



```
KafkaProducerExample.java x
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 x
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]));
    }
}
```

KafkaExample

Run

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-validation.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-validation**
 - ❖ 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?

Complete Lab 2

12. Advanced Producers

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, Acknowledgments 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



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer
17 > public class StockPriceKafkaProducer {
18     private static final Logger logger = LoggerFactory.getLogger(StockPriceKafkaProducer.class);
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 whose 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

c StockPriceSerializer.java x

```
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

```
c StockPrice.java x
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);
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40      private static void setupRetriesInFlightTimeout(Properties props) {
41
42          //Only two in-flight messages per Kafka broker connection
43          // - max.in.flight.requests.per.connection (default 5)
44          props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
45                  1);
46
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 x
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 x StockPricePartitioner.java x
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<>(); }
13
14     @Override
15     public int partition(final String topic,
16                         final Object objectKey,
17                         byte[] keyBytes, final Object value,
18                         final byte[] valueBytes,
19                         final Cluster cluster) {...}
20
21     @Override
22     public void close() {
23     }
24
25     @Override
26     public void configure(Map<String, ?> configs) {...}
27 }
```

StockPricePartitioner

partition()



```
StockPriceKafkaProducer.java x StockPricePartitioner.java x
StockPricePartitioner partition()
17 ① 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



```
StockPriceKafkaProducer.java x StockPricePartitioner.java x
StockPricePartitioner configure()
39
40
41 ①↑   public void close() {
42    }
43
44 ①↑   @Override
45 ①↑   public void configure(Map<String, ?> configs) {
46    final String importantStocksStr = (String) configs.get("importantStocks");
47    Arrays.stream(importantStocksStr.split( regex: "," ))
48    .forEach(importantStocks::add);
49
50 }
```

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



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer getStockSenderList()
17 > public class StockPriceKafkaProducer {
18     private static final Logger logger = LoggerFactory.getLogger(Stock
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



```
c StockProducerInterceptor.java x
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



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

Output

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

ProducerInterceptor onAck



```
c StockProducerInterceptor.java x
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



```
c StockProducerInterceptor.java x

StockProducerInterceptor configure()
48     Logger.info(String.format("onAck topic=%s",
49                         metadata.topic(), metadata.partition
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



- ❖ ***InterruptedException*** - 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

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



```
c StockSender.java x
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\tkey=%s, value=%s " +
79                           "\n\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



```
c StockPriceKafkaProducer.java x
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



```
c StockPriceKafkaProducer.java x
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
- ❖ Producers 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

MetricName

double

Method and Description

metricName()

A name for this metric

value()

The value of the metric

Metrics producer.metrics()



```
c MetricsProducerReporter.java x
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, 0.025031289111389236,
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, 0.41441360272859273,
The average compression rate of record batches.

Metrics via JMX



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

Java Monitoring & Management Console

Connection Window Help

pid: 31636 com.intellij.rt.execution.application.AppMain com.cloudurable.kafka.producer.StockPriceKafkaProducer

Overview Memory Threads Classes VM Summary MBeans

Attribute value

Name	Value
compression-rate	0.4308939902619882

Refresh

MBeanAttributeInfo

Name	Value
Attribute:	compression-rate
Name	compression-rate
Description	
Readable	true
Writable	false

Descriptor

Name	Value

request-latency-max
request-latency-avg
incoming-byte-rate
request-size-avg
outgoing-byte-rate
request-size-max

- ▶ node--2
- ▶ node--3
- ▶ node-0
- ▶ node-2

▼ producer-topic-metrics

 ▼ StockPriceKafkaProducer

 ▼ stock-prices2

 ▼ Attributes

 record-retry-rate
 record-send-rate
 compression-rate
 byte-rate
 record-error-rate

Complete Lab 5.1

13. About the App (DEMO)

StockPrice App to demo Advanced Producer



- ❖ **StockPrice** - holds a stock price has a name, dollar, and cents
- ❖ **StockPriceKafkaProducer** - Configures and creates **KafkaProducer<String, StockPrice>**, **StockSender** list, ThreadPool (ExecutorService), starts **StockSender** runnable into thread pool
- ❖ **StockAppConstants** - holds topic and broker list
- ❖ **StockPriceSerializer** - can serialize a **StockPrice** into **byte[]**
- ❖ **StockSender** - generates somewhat random stock prices for a given **StockPrice** name, Runnable, 1 thread per StockSender
 - ❖ Shows using **KafkaProducer** from many threads

StockPrice domain object



c StockPrice.java x

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}
```

- ❖ has name
- ❖ dollars
- ❖ cents
- ❖ converts itself to JSON

StockPriceKafkaProducer



- ❖ Import classes and setup logger
- ❖ Create ***createProducer*** method to create ***KafkaProducer*** instance
- ❖ Create ***setupBootstrapAndSerializers*** to initialize bootstrap servers, client id, key serializer and custom serializer (***StockPriceSerializer***)
- ❖ Write ***main()*** method - creates ***producer***, create ***StockSender*** list passing each instance a ***producer***, creates a thread pool so every stock sender gets its own thread, runs each ***stockSender*** in its own thread

StockPriceKafkaProducer imports, createProducer



StockPriceKafkaProducer main()

```
1 package com.cloudurable.kafka.producer;
2
3 import com.cloudurable.kafka.StockAppConstants;
4 import com.cloudurable.kafka.producer.model.StockPrice;
5 import io.advantageous.boon.core.Lists;
6 import org.apache.kafka.clients.producer.*;
7 import org.apache.kafka.common.serialization.StringSerializer;
8 import org.slf4j.Logger;
9 import org.slf4j.LoggerFactory;
10
11 import java.util.List;
12 import java.util.Properties;
13 import java.util.concurrent.ExecutorService;
14 import java.util.concurrent.Executors;
15 import java.util.concurrent.TimeUnit;
16
17 public class StockPriceKafkaProducer {
18     private static final Logger logger =
19         LoggerFactory.getLogger(StockPriceKafkaProducer.class);
20
21     private static Producer<String, StockPrice> createProducer() {
22         final Properties props = new Properties();
23         setupBootstrapAndSerializers(props);
24         return new KafkaProducer<>(props);
25     }
}
```

Import classes and
setup logger

createProducer
used to create a
KafkaProducer

createProducer()
calls
setupBootstrapAnd
Serializers()

Configure Producer Bootstrap and Serializer



StockPriceKafkaProducer

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

- ❖ Create **setupBootstrapAndSerializers** to initialize bootstrap servers, client id, key serializer and custom serializer (**StockPriceSerializer**)
- ❖ **StockPriceSerializer** will serialize **StockPrice** into bytes

StockPriceKafkaProducer.main()



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer main()
39 >     public static void main(String... args)
40             throws Exception {
41
42         //Create Kafka Producer
43         final Producer<String, StockPrice> producer = createProducer();
44
45         //Create StockSender list
46         final List<StockSender> stockSenders = getStockSenderList(producer);
47
48         //Create a thread pool so every stock sender gets its own.
49         final ExecutorService executorService =
50             Executors.newFixedThreadPool(stockSenders.size());
51
52         //Run each stock sender in its own thread.
53         stockSenders.forEach(executorService::submit);
54
55     }
```

- ❖ **main** method - creates **producer**,
- ❖ create **StockSender** list passing each instance a **producer**
- ❖ creates a thread pool (executorService)
- ❖ every StockSender runs in its own thread

StockAppConstants



```
1 package com.cloudurable.kafka;  
2  
3 public class StockAppConstants {  
4     public final static String TOPIC = "stock-prices";  
5     public final static String BOOTSTRAP_SERVERS =  
6         "localhost:9092,localhost:9093,localhost:9094";  
7  
8 }
```

- ❖ topic name for Producer example
- ❖ List of bootstrap servers

StockPriceKafkaProducer.getStockSenderList



```
StockPriceKafkaProducer getStockSenderList()
57 @ private static List<StockSender> getStockSenderList(
58     final Producer<String, StockPrice> producer) {
59     return Lists.list(
60         new StockSender(StockAppConstants.TOPIC,
61             new StockPrice( name: "IBM", dollars: 100, cents: 99),
62             new StockPrice( name: "IBM", dollars: 50, cents: 10),
63             producer,
64             delayMinMs: 1, delayMaxMs: 10
65         ),
66         new StockSender(
67             StockAppConstants.TOPIC,
68             new StockPrice( name: "SUN", dollars: 100, cents: 99),
69             new StockPrice( name: "SUN", dollars: 50, cents: 10),
70             producer,
71             delayMinMs: 1, delayMaxMs: 10
72         ),
73         new StockSender(
74             StockAppConstants.TOPIC,
75             new StockPrice( name: "GOOG", dollars: 500, cents: 99),
76             new StockPrice( name: "GOOG", dollars: 400, cents: 10),
77             producer,
78             delayMinMs: 1, delayMaxMs: 10
79         ),
80         new StockSender(
81             StockAppConstants.TOPIC,
82             new StockPrice( name: "INEL", dollars: 100, cents: 99),
83             new StockPrice( name: "INEL", dollars: 50, cents: 10),
84             producer,
85             delayMinMs: 1, delayMaxMs: 10
)
```



StockPriceSerializer

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

- ❖ Converts **StockPrice** into byte array



StockSender

- ❖ Generates random stock prices for a given ***StockPrice*** name,
- ❖ StockSender is Runnable
- ❖ 1 thread per StockSender
- ❖ Shows using ***KafkaProducer*** from many threads
- ❖ Delays random time between delayMin and delayMax,
 - ❖ then sends random StockPrice between ***stockPriceHigh*** and ***stockPriceLow***

StockSender imports, Runnable



```
StockSender
1 package com.cloudurable.kafka.producer;
2
3 import com.cloudurable.kafka.producer.model.StockPrice;
4 import org.apache.kafka.clients.producer.Producer;
5 import org.apache.kafka.clients.producer.ProducerRecord;
6 import org.apache.kafka.clients.producer.RecordMetadata;
7 import org.slf4j.Logger;
8 import org.slf4j.LoggerFactory;
9
10 import java.util.Date;
11 import java.util.Random;
12 import java.util.concurrent.ExecutionException;
13 import java.util.concurrent.Future;
14
15 public class StockSender implements Runnable{
```

- ❖ Imports Kafka **Producer**, **ProducerRecord**, **RecordMetadata**, **StockPrice**
- ❖ Implements Runnable, can be submitted to **ExecutionService**



StockSender constructor

StockSender

```
17     private final StockPrice stockPriceHigh;
18     private final StockPrice stockPriceLow;
19     private final Producer<String, StockPrice> producer;
20     private final int delayMinMs;
21     private final int delayMaxMs;
22     private final Logger logger = LoggerFactory.getLogger(StockSender.class);
23     private final String topic;
24
25     public StockSender(final String topic, final StockPrice stockPriceHigh,
26                         final StockPrice stockPriceLow,
27                         final Producer<String, StockPrice> producer,
28                         final int delayMinMs,
29                         final int delayMaxMs) {
30         this.stockPriceHigh = stockPriceHigh;
31         this.stockPriceLow = stockPriceLow;
32         this.producer = producer;
33         this.delayMinMs = delayMinMs;
34         this.delayMaxMs = delayMaxMs;
35         this.topic = topic;
36     }
```

- ❖ takes a topic, high & low stockPrice, producer, delay min & max



StockSender run()

```
StockSender run()

38
39 ①↑ public void run() {
40     final Random random = new Random(System.currentTimeMillis());
41     int sentCount = 0;
42
43     while (true) {
44         sentCount++;
45         final ProducerRecord <String, StockPrice> record =
46             createRandomRecord(random);
47         final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);
48
49         try {
50             final Future<RecordMetadata> future = producer.send(record);
51             if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
52             Thread.sleep(delay);
53         } catch (InterruptedException e) {
54             if (Thread.interrupted()) {
55                 break;
56             }
57         } catch (ExecutionException e) {
58             logger.error("problem sending record to producer", e);
59         }
60     }
61 }
```

- ❖ In loop, creates random record, **send** record, waits random time

StockSender

createRandomRecord



```
StockSender createRandomRecord()

77
78     private final int randomIntBetween(final Random random,
79                                         final int max,
80                                         final int min) {
81         return random.nextInt( bound: max - min + 1 ) + min;
82     }
83
84     private ProducerRecord<String, StockPrice> createRandomRecord(
85         final Random random) {
86
87         final int dollarAmount = randomIntBetween(random,
88             stockPriceHigh.getDollars(), stockPriceLow.getDollars());
89
90         final int centAmount = randomIntBetween(random,
91             stockPriceHigh.getCents(), stockPriceLow.getCents());
92
93         final StockPrice stockPrice = new StockPrice(
94             stockPriceHigh.getName(), dollarAmount, centAmount);
95
96         return new ProducerRecord<>(topic, stockPrice.getName(),
97             stockPrice);
98     }
```

- ❖ *createRandomRecord* uses *randomIntBetween*
- ❖ creates **StockPrice** and then wraps **StockPrice** in **ProducerRecord**

StockSender

displayRecordMetaData



StockSender displayRecordMetaData()

```
62
63     private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
64                                     final Future<RecordMetadata> future)
65                                     throws InterruptedException, ExecutionException {
66     final RecordMetadata recordMetadata = future.get();
67     logger.info(String.format("\n\t\tkey=%s, value=%s " +
68                 "\n\t\tsent to topic=%s part=%d off=%d at time=%s",
69             record.key(),
70             record.value().toJson(),
71             recordMetadata.topic(),
72             recordMetadata.partition(),
73             recordMetadata.offset(),
74             new Date(recordMetadata.timestamp())
75         ));
76 }
```

- ❖ Every 100 records displayRecordMetaData gets called
- ❖ Prints out record info, and recordMetadata info:
 - ❖ key, JSON value, topic, partition, offset, time
 - ❖ uses Future from call to *producer.send()*



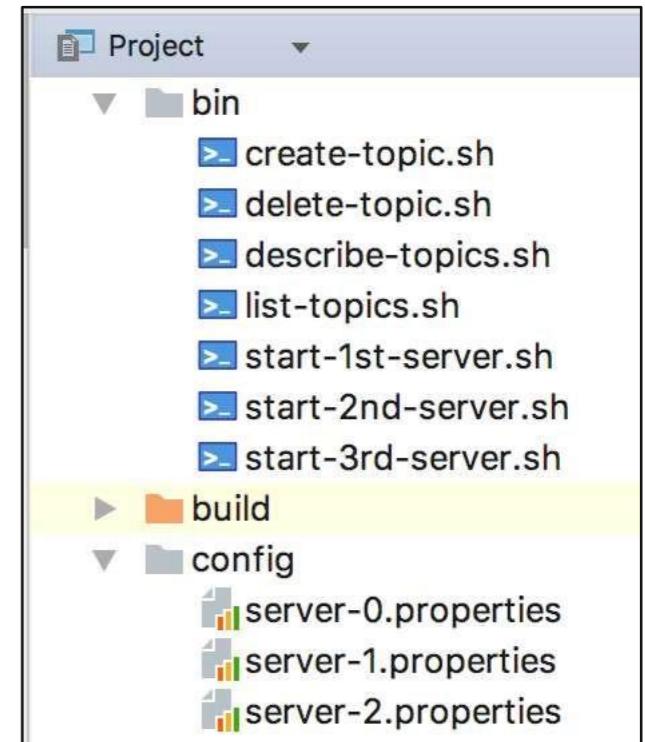
Run it

- ❖ Run ZooKeeper
- ❖ Run three Brokers
- ❖ run `create-topic.sh`
- ❖ Run ***StockPriceKafkaProducer***



Run scripts

- ❖ run ZooKeeper from ~/kafka-training
- ❖ use ***bin/create-topic.sh*** to create topic
- ❖ use ***bin/delete-topic.sh*** to delete topic
- ❖ use ***bin/start-1st-server.sh*** to run Kafka Broker 0
- ❖ use ***bin/start-2nd-server.sh*** to run Kafka Broker 1
- ❖ use ***bin/start-3rd-server.sh*** to run Kafka Broker 2



Config is under directory called config

server-0.properties is for Kafka Broker 0
server-1.properties is for Kafka Broker 1
server-2.properties is for Kafka Broker 2



Run All 3 Brokers

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka
5 kafka/bin/kafka-server-start.sh \
6   "$CONFIG/server-0.properties"
7
```

```
start-2nd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka
5 kafka/bin/kafka-server-start.sh \
6   "$CONFIG/server-1.properties"
7
```

```
start-3rd-server.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka
5 kafka/bin/kafka-server-start.sh \
6   "$CONFIG/server-2.properties"
7
```

```
server-0.properties x
1 broker.id=0
2 port=9092
3 log.dirs=./logs/kafka-0
4 min.insync.replicas=3
5 compression.type=producer
6 auto.create.topics.enable=false
7 message.max.bytes=65536
8 replica.lag.time.max.ms=5000
9 delete.topic.enable=true
```

```
server-1.properties x
1 broker.id=1
2 port=9093
3 log.dirs=./logs/kafka-1
4 min.insync.replicas=3
5 compression.type=producer
```

```
server-2.properties x
1 broker.id=2
2 port=9094
3 log.dirs=./logs/kafka-2
4 min.insync.replicas=3
5 compression.type=producer
6 auto.create.topics.enable=false
7 message.max.bytes=65536
8 replica.lag.time.max.ms=5000
9 delete.topic.enable=true
```

Run create-topic.sh script



```
create-topic.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 kafka/bin/kafka-topics.sh \
6   --create \
7   --zookeeper localhost:2181 \
8   --replication-factor 3 \
9   --partitions 3 \
10  --topic stock-prices
11
```

Name of the topic
is stock-prices

Three partitions

Replication factor
of three

```
Terminal
+
X $ ./bin/create-topic.sh
Created topic "stock-prices".
```

Run StockPriceKafkaProducer



The screenshot shows a Java IDE's run window titled "StockPriceKafkaProducer". The log output displays several INFO messages from the `c.c.kafka.producer.StockSender` class. Each message indicates a stock price record is being sent to the "stock-prices" topic. The records include a key (stock name), a JSON value (dollars and cents), and metadata like partition, offset, and timestamp (Sun May 21 23:53:13 PDT 2017). The log entries are as follows:

```
key=SON, value={"dollars": 68, "cents": 37, "name": "SON"}  
sent to topic=stock-prices part=2 off=41360 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.024 [pool-1-thread-6] INFO c.c.kafka.producer.StockSender -  
key=ABC, value={"dollars": 68, "cents": 37, "name": "ABC"}  
sent to topic=stock-prices part=0 off=48256 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.049 [pool-1-thread-3] INFO c.c.kafka.producer.StockSender -  
key=G00G, value={"dollars": 498, "cents": 37, "name": "G00G"}  
sent to topic=stock-prices part=0 off=48287 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.049 [pool-1-thread-1] INFO c.c.kafka.producer.StockSender -  
key=IBM, value={"dollars": 68, "cents": 37, "name": "IBM"}  
sent to topic=stock-prices part=2 off=41399 at time=Sun May 21 23:53:13 PDT 2017  
23:53:13.171 [pool-1-thread-8] INFO c.c.kafka.producer.StockSender -  
key=DEF, value={"dollars": 63, "cents": 26, "name": "DEF"}  
...
```

- ❖ Run **StockPriceKafkaProducer** from the IDE
- ❖ You should see log messages from **StockSender(s)** with **StockPrice** name, JSON value, partition, offset, and time

14. Producer Shutdown

Shutdown Producer nicely



- ❖ Handle ctrl-C shutdown from Java
- ❖ Shutdown thread pool and wait
- ❖ Flush producer to send any outstanding batches if using batches (***producer.flush()***)
- ❖ Close Producer (***producer.close()***) and wait

Nice Shutdown producer.close()



```
c StockPriceKafkaProducer.java x
StockPriceKafkaProducer main()
41 > 1 public static void main(String... args)
42           throws Exception {
43             //Create Kafka Producer
44             final Producer<String, StockPrice> producer = createProducer();
45             //Create StockSender list
46             final List<StockSender> stockSenders = getStockSenderList(producer);
47             //Create a thread pool so every stock sender gets its own.
48             final ExecutorService executorService =
49               Executors.newFixedThreadPool(stockSenders.size());
50             //Run each stock sender in its own thread.
51             stockSenders.forEach(executorService::submit);
52
53             //Register nice shutdown of thread pool, then flush and close producer.
54             Runtime.getRuntime().addShutdownHook(new Thread(() -> {
55               executorService.shutdown();
56               try {
57                 executorService.awaitTermination(timeout: 200, TimeUnit.MILLISECONDS);
58                 logger.info("Flushing and closing producer");
59                 producer.flush();
60                 producer.close(timeout: 10_000, TimeUnit.MILLISECONDS);
61               } catch (InterruptedException e) {
62                 logger.warn("shutting down", e);
63               }
64             }));
65           });
}
```

Restart Producer then shut it down



- ❖ Add shutdown hook
- ❖ Start StockPriceKafkaProducer
- ❖ Now stop it (CTRL-C or hit stop button in IDE)

```
External Libraries
Run StockPriceKafkaProducer
00:06:52.709 [pool-1-thread-4] INFO c.c.kafka.producer.StockSender -
    key=INEL, value={"dollars": 73, "cents": 10, "name": "INEL"}
    sent to topic=stock-prices part=0 off=93522 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.710 [pool-1-thread-5] INFO c.c.kafka.producer.StockSender -
    key=UBER, value={"dollars": 547, "cents": 90, "name": "UBER"}
    sent to topic=stock-prices part=2 off=80143 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.717 [pool-1-thread-2] INFO c.c.kafka.producer.StockSender -
    key=SUN, value={"dollars": 73, "cents": 10, "name": "SUN"}
    sent to topic=stock-prices part=2 off=80146 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.718 [pool-1-thread-1] INFO c.c.kafka.producer.StockSender -
    key=IBM, value={"dollars": 73, "cents": 10, "name": "IBM"}
    sent to topic=stock-prices part=2 off=80148 at time=Mon May 22 00:06:52 PDT 2017
00:06:52.967 [Thread-1] INFO c.c.k.p.StockPriceKafkaProducer - Flushing and closing producer
00:06:52.968 [Thread-1] INFO o.a.k.clients.producer.KafkaProducer - Closing the Kafka producer
?
Process finished with exit code 130 (interrupted by signal 2: SIGINT)
```

Complete Lab 5.2 (just read
but don't complete) and
Complete Labs 5.3-5.8

15. Kafka 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 where topic logs will be stored using 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

	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 x
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);
41
```

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	<p>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).</p>	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 sta

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

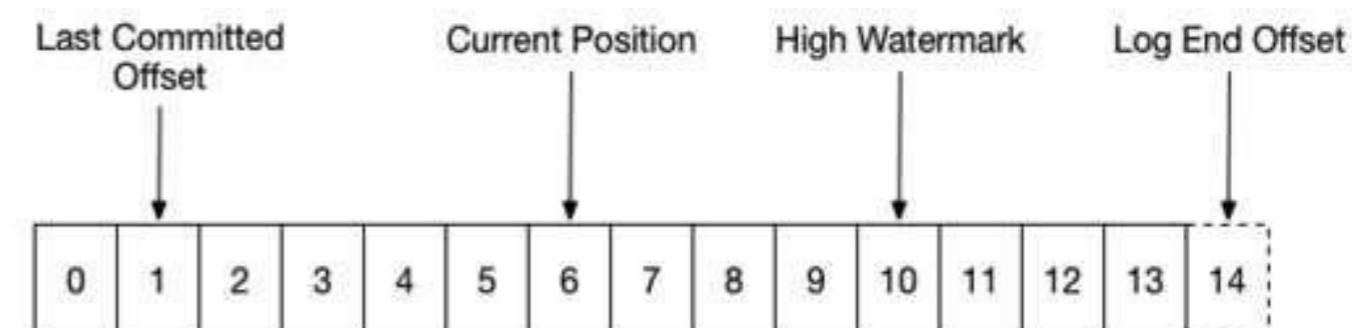


- ❖ A Kafka partition is a replicated log - replicated log is a distributed data system primitive
- ❖ Replicated log useful for building distributed systems using state-machines
- ❖ A replicated log models “coming into consensus” on ordered series of values
 - ❖ While leader stays alive, all followers 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 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***, if 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?

16. 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



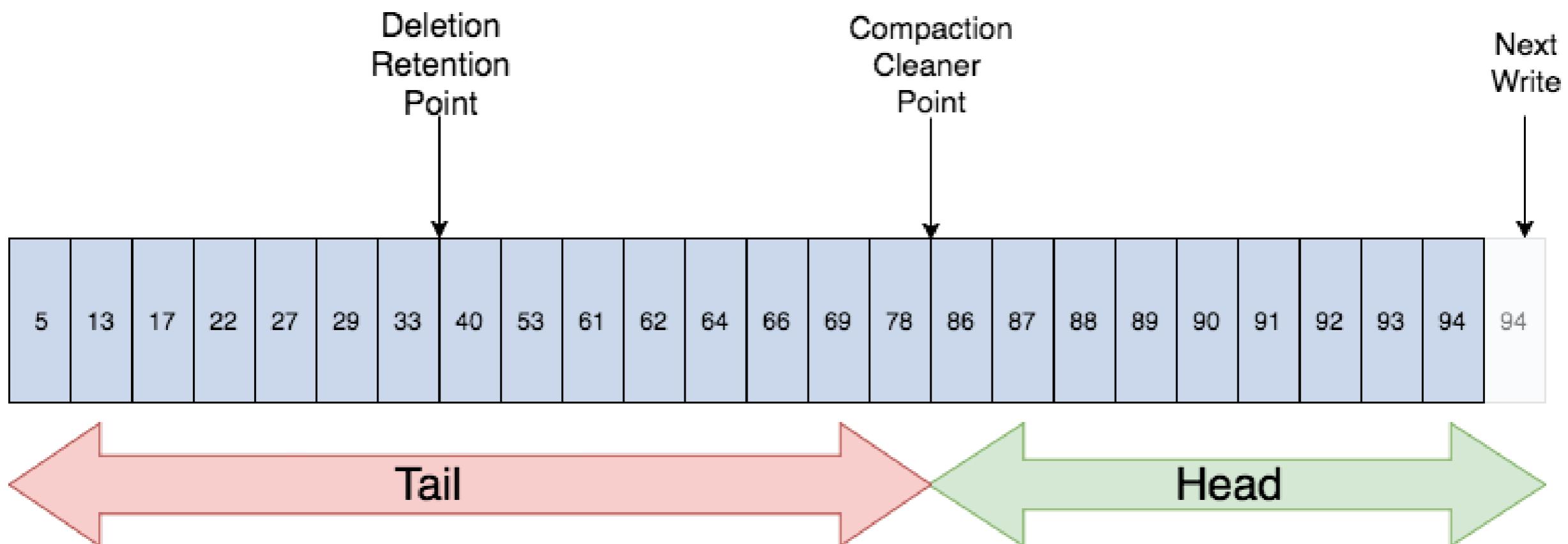
- ❖ Log compaction retains last known value for each record key
- ❖ Useful for restoring state after a crash or system failure, e.g., in-memory service, persistent data store, reloading a cache
- ❖ Data streams is to log changes to keyed, mutable data,
 - ❖ e.g., changes to a database table, changes 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?

17. Introduction to Consumer

Objectives Create a Consumer



- ❖ Create simple example that creates a ***Kafka Consumer***
 - ❖ that consumes messages from 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 x
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 x
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 x

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



```
67  
68 >   public static void main(String... args) throws Exception {  
69     runConsumer();  
70   }  
71 }
```

Run KafkaConsumerExample

ssl.protocol = TLS
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 x

1 <configuration>
2   <appender name="STDOUT"
3     class="ch.qos.logback.core.ConsoleAppender">
4     <encoder>
5       <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
6         %logger{36} - %msg%n</pattern>
7     </encoder>
8   </appender>
9
10  <logger name="org.apache.kafka" level="INFO"/>
11  <logger name="org.apache.kafka.common.metrics" level="INFO"/>
12
13  <root level="debug">
14    <appender-ref ref="STDOUT" />
15  </root>
16</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)

```
10.32.155.193 [main] INFO org.apache.kafka.clients.consumer.ConsumerCoordinator - Discovered
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

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

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)
```

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)
```

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?

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)
```

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 x
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 x
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 from 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?

Complete Labs 3

18. Advanced Consumers

Objectives Advanced Kafka Producers



- ❖ Using auto commit / Turning auto commit off
- ❖ Managing a custom partition and offsets
 - ❖ ***ConsumerRebalanceListener***
- ❖ Manual Partition Assignment (**assign()** vs. **subscribe()**)
- ❖ Consumer Groups, aliveness
 - ❖ **poll()** and **session.timeout.ms**
- ❖ Consumers and message delivery semantics
 - ❖ at most once, at least once, exactly once
- ❖ Consumer threading models
 - ❖ thread per consumer, consumer with many threads (both)

Java Consumer Examples Overview



- ❖ Rewind Partition using ***ConsumerRebalanceListener*** and ***consumer.seekX()*** full Java example
- ❖ At-Least-Once Delivery Semantics full Java Example
- ❖ At-Most-Once Delivery Semantics full Java Example
- ❖ Exactly-Once Delivery Semantics full Java Example
 - ❖ full example storing topic, partition, offset in RDBMS with JDBC
 - ❖ Uses ***ConsumerRebalanceListener*** to restore to correct location in log partitions based off of database records
 - ❖ Uses transactions, rollbacks if commitSync fails
- ❖ Threading model Java Examples
 - ❖ Thread per Consumer
 - ❖ Consumer with multiple threads (***TopicPartition*** management of ***commitSync()***)
- ❖ Implement full “priority queue” behavior using Partitioner and manual partition assignment
 - ❖ Manual Partition Assignment (***assign()*** vs. ***subscribe()***)



KafkaConsumer

- ❖ A Consumer client
 - ❖ consumes records from Kafka cluster
- ❖ Automatically handles Kafka broker failure
 - ❖ adapts as topic partitions leadership moves in Kafka cluster
- ❖ Works with Kafka broker to form consumers groups and load balance consumers
- ❖ Consumer maintains connections to Kafka brokers in cluster
- ❖ Use ***close()*** method to not leak resources
- ❖ NOT thread-safe

StockPrice App to demo Advanced Producer



- ❖ ***StockPrice*** - holds a stock price has a name, dollar, and cents
- ❖ ***StockPriceConsumer*** - Kafka ***Consumer*** that consumes ***StockPrices***
- ❖ ***StockAppConstants*** - holds topic and broker list
- ❖ ***StockPriceDeserializer*** - can deserialize a ***StockPrice*** from ***byte[]***

Kafka Consumer Example



- ❖ **StockPriceConsumer** to consume StockPrices and display batch lengths for **poll()**
- ❖ Shows using poll(), and basic configuration (review)
- ❖ Also Shows how to use a Kafka Deserializer
- ❖ How to configure the **Deserializer** in Consumer config
- ❖ All the other examples build on this and this builds on Advanced Producer StockPrice example

Consumer: createConsumer / Consumer Config



```
c SimpleStockPriceConsumer.java x
SimpleStockPriceConsumer

11
12 > public class SimpleStockPriceConsumer {
13
14     private static Consumer<String, StockPrice> createConsumer() {
15         final Properties props = new Properties();
16         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
17             StockAppConstants.BOOTSTRAP_SERVERS);
18         props.put(ConsumerConfig.GROUP_ID_CONFIG,
19             "KafkaExampleConsumer");
20         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
21             StringDeserializer.class.getName());
22         //Custom Deserializer
23         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
24             StockDeserializer.class.getName());
25         props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
26         // Create the consumer using props.
27         final Consumer<String, StockPrice> consumer =
28             new KafkaConsumer<>(props);
29         // Subscribe to the topic.
30         consumer.subscribe(Collections.singletonList(
31             StockAppConstants.TOPIC));
32         return consumer;
33     }
}
```

- ❖ Similar to other Consumer examples so far
- ❖ Subscribes to **stock-prices** topic
- ❖ Has custom serializer

SimpleStockPriceConsumer.runConsumer



```
c SimpleStockPriceConsumer.java x
SimpleStockPriceConsumer

35
36     static void runConsumer() throws InterruptedException {
37         final Consumer<String, StockPrice> consumer = createConsumer();
38         final Map<String, StockPrice> map = new HashMap<>();
39         try {
40             final int giveUp = 1000; int noRecordsCount = 0;
41             int readCount = 0;
42             while (true) {
43                 final ConsumerRecords<String, StockPrice> consumerRecords =
44                     consumer.poll( timeout: 1000 );
45                 if (consumerRecords.count() == 0) {
46                     noRecordsCount++;
47                     if (noRecordsCount > giveUp) break;
48                     else continue;
49                 }
50                 readCount++;
51                 consumerRecords.forEach(record -> {
52                     map.put(record.key(), record.value());
53                 });
54                 if (readCount % 100 == 0) {
55                     displayRecordsStatsAndStocks(map, consumerRecords);
56                 }
57                 consumer.commitAsync();
58             }
59         }
```

- ❖ Drains topic; Creates map of current stocks; Calls ***displayRecordsStatsAndStocks()***

Using ConsumerRecords : SimpleStockPriceConsumer.display



SimpleStockPriceConsumer.java ×

SimpleStockPriceConsumer displayRecordsStatsAndStocks()

```
66     private static void displayRecordsStatsAndStocks(
67         final Map<String, StockPrice> stockPriceMap,
68         final ConsumerRecords<String, StockPrice> consumerRecords) {
69     System.out.printf("New ConsumerRecords par count %d count %d\n",
70             consumerRecords.partitions().size(),
71             consumerRecords.count());
72     stockPriceMap.forEach((s, stockPrice) ->
73         System.out.printf("ticker %s price %d.%d \n",
74             stockPrice.getName(),
75             stockPrice.getDollars(),
76             stockPrice.getCents()));
77     System.out.println();
78 }
```

- ❖ Prints out size of each partition read and total record count
- ❖ Prints out each stock at its current price

Consumer Deserializer: StockDeserializer



```
c SimpleStockPriceConsumer.java x c StockDeserializer.java x
StockDeserializer
3 import ...
8
9 public class StockDeserializer implements Deserializer<StockPrice> {
10
11     @Override
12     public StockPrice deserialize(final String topic, final byte[] data) {
13         return new StockPrice(new String(data, StandardCharsets.UTF_8));
14     }
15
16     @Override
17     public void configure(Map<String, ?> configs, boolean isKey) {
18
19     }
20
21     @Override
22     public void close()
23 }
```

```
c SimpleStockPriceConsumer.java x c StockDeserializer.java x c StockPrice.java x
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 StockPrice(final String json) {
12         this(JsonFactory.fromJson(json, StockPrice.class));
13     }
14 }
```

Kafka Consumer: Offsets and Consumer Position

- ❖ Consumer position is offset and partition of last record per partition consuming from
 - ❖ offset for each record in a partition as a unique identifier record location in partition
- ❖ Consumer position gives offset of next record that it consume (next highest)
 - ❖ position advances automatically for each call to ***poll(..)***
- ❖ Consumer committed position is last offset that has been stored to broker
 - ❖ If consumer fails, it picks up at last committed position
- ❖ Consumer can auto commit offsets (***enable.auto.commit***) periodically (***auto.commit.interval.ms***) or do commit explicitly using ***commitSync()*** and ***commitAsync()***

KafkaConsumer: Consumer Groups



- ❖ Consumers organized into consumer groups (Consumer instances with same **group.id**)
 - ❖ Pool of consumers divide work of consuming and processing records
 - ❖ Processes or threads running on same box or distributed for scalability/fault tolerance
- ❖ Kafka shares topic partitions among all consumers in a consumer group
 - ❖ each partition is assigned to exactly one consumer in consumer group
 - ❖ Example topic has six partitions, and a consumer group has two consumer processes, each process gets consume three partitions
- ❖ Failover and Group rebalancing
 - ❖ if a consumer fails, Kafka reassigned partitions from failed consumer to other consumers in same consumer group
 - ❖ if new consumer joins, Kafka moves partitions from existing consumers to new one

Consumer Groups and Topic Subscriptions



- ❖ Consumer group form ***single logical subscriber*** made up of multiple consumers
- ❖ Kafka is a multi-subscriber system, Kafka supports N number of ***consumer groups*** for a given topic without duplicating data
- ❖ To get something like a MOM queue all consumers would be in single consumer group
 - ❖ Load balancing like a MOM queue
- ❖ Unlike a MOM, you can have multiple such consumer groups, and price of subscribers does not incur duplicate data
- ❖ To get something like MOM pub-sub each process would have its own consumer group

KafkaConsumer: Partition Reassignment



- ❖ Consumer partition reassignment in a consumer group happens automatically
- ❖ Consumers are notified via ***ConsumerRebalanceListener***
 - ❖ Triggers consumers to finish necessary clean up
- ❖ Consumer can use API to assign specific partitions using ***assign(Collection)***
 - ❖ disables dynamic partition assignment and consumer group coordination
- ❖ Dead client may see CommitFailedException thrown from a call to `commitSync()`
 - ❖ Only active members of consumer group can commit offsets.

Controlling Consumers Position



- ❖ You can control consumer position
 - ❖ moving consumer forward or backwards - consumer can re-consume older records or skip to most recent records
- ❖ Use ***consumer.seek(TopicPartition, long)*** to specify new position
 - ❖ ***consumer.seekToBeginning(Collection) and seekToEnd(Collection) respectively***
- ❖ Use Case Time-sensitive record processing: Skip to most recent records
- ❖ Use Case Bug Fix: Reset position before bug fix and replay log from there
- ❖ Use Case Restore State for Restart or Recovery: Consumer initialize position on start-up to whatever is contained in local store and replay missed parts (cache warm-up or replacement in case of failure assumes Kafka retains sufficient history or you are using log compaction)

Storing Offsets Outside: Managing Offsets



- ❖ For the consumer to manage its own offset you just need to do the following:
 - ❖ Set ***enable.auto.commit=false***
 - ❖ Use offset provided with each ConsumerRecord to save your position (partition/offset)
 - ❖ On restart restore consumer position using ***kafkaConsumer.seek(TopicPartition, long)***.
- ❖ Usage like this simplest when the partition assignment is also done manually using ***assign()*** instead of ***subscribe()***

Storing Offsets Outside: Managing Offsets



- ❖ If using automatic partition assignment, you must handle cases where partition assignments change
 - ❖ Pass ***ConsumerRebalanceListener*** instance in call to ***kafkaConsumer.subscribe(Collection, ConsumerRebalanceListener)*** and ***kafkaConsumer.subscribe(Pattern, ConsumerRebalanceListener)***.
 - ❖ when partitions taken from consumer, commit its offset for partitions by implementing ***ConsumerRebalanceListener.onPartitionsRevoked(Collection)***
 - ❖ When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ***ConsumerRebalanceListener.onPartitionsAssigned(Collection)***

```
// Subscribe to the topic.  
consumer.subscribe(Collections.singletonList(  
    StockAppConstants.TOPIC),  
    new SeekToConsumerRebalanceListener(consumer, seekTo, location));
```

Controlling Consumers Position Example



```
SeekToConsumerRebalanceListener onPartitionsAssigned()  
o  
9  
10 public class SeekToConsumerRebalanceListener implements ConsumerRebalanceListener {  
11     private final Consumer<String, StockPrice> consumer;  
12     private final SeekTo seekTo; private boolean done;  
13     private final long location;  
14     private final long startTime = System.currentTimeMillis();  
15     public SeekToConsumerRebalanceListener(final Consumer<String, StockPrice> consumer,  
16  
21         @Override  
22     ⚡ public void onPartitionsAssigned(final Collection<TopicPartition> partitions) {  
23         if (done) return;  
24         else if (System.currentTimeMillis() - startTime > 30_000) {  
25             done = true;  
26             return;  
27         }  
28         switch (seekTo) {  
29             case END: //Seek to end  
30                 consumer.seekToEnd(partitions);  
31                 break;  
32             case START: //Seek to start  
33                 consumer.seekToBeginning(partitions);  
34                 break;  
35             case LOCATION: //Seek to a given location  
36                 partitions.forEach(topicPartition ->  
37                     consumer.seek(topicPartition, location));  
38                 break;  
39         }  
40     }
```

Kafka Consumer: Consumer Alive Detection

- ❖ Consumers join consumer group after subscribe and then **poll()** is called
- ❖ Automatically, consumer sends periodic heartbeats to Kafka brokers server
- ❖ If consumer crashes or unable to send heartbeats for a duration of **session.timeout.ms**, then consumer is deemed dead and its partitions are reassigned

Kafka Consumer: Manual Partition Assignment



- ❖ Instead of subscribing to the topic using `subscribe`, you can call **`assign(Collection)`** with the full topic partition list

```
String topic = "log-replication";  
  
TopicPartition part0 = new TopicPartition(topic, 0);  
  
TopicPartition part1 = new TopicPartition(topic, 1);  
  
consumer.assign(Arrays.asList(part0, part1));
```

- ❖ Using consumer as before with **`poll()`**
- ❖ Manual partition assignment negates use of group coordination, and auto consumer fail over - Each consumer acts independently even if in a consumer group (use unique group id to avoid confusion)
- ❖ You have to use **`assign()`** or **`subscribe()`** but not both

KafkaConsumer: Consumer Alive if Polling



- ❖ Calling ***poll()*** marks consumer as alive
 - ❖ If consumer continues to call ***poll()***, then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every ***max.poll.interval.ms interval***)
 - ❖ Not calling ***poll()***, even if consumer is sending heartbeats, consumer is still considered dead
- ❖ Processing of records from ***poll*** has to be faster than ***max.poll.interval.ms*** interval or your consumer could be marked dead!
- ❖ ***max.poll.records*** is used to limit total records returned from a poll call - easier to predict max time to process records on each poll interval



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

“At-Least-Once” - Delivery Semantics



```
SimpleStockPriceConsumer pollRecordsAndProcess()
```

```
76
77     final ConsumerRecords<String, StockPrice> consumerRecords =
78         consumer.poll( timeout: 1000 );
79
80     try {
81         startTransaction();           //Start DB Transaction
82
83                     //Process the records
84         processRecords(map, consumerRecords);
85
86                     //Commit the Kafka offset
87         consumer.commitSync();
88
89         commitTransaction();          //Commit DB Transaction
90     } catch(CommitFailedException ex) {
91         logger.error("Failed to commit sync to log", ex);
92         rollbackTransaction();       //Rollback Transaction
93     } catch (DatabaseException dte) {
94         logger.error("Failed to write to DB", dte);
95         rollbackTransaction();       //Rollback Transaction
96     }
```

“At-Most-Once” - Delivery Semantics



```
SimpleStockPriceConsumer pollRecordsAndProcess()  
 76  
 77    final ConsumerRecords<String, StockPrice> consumerRecords =  
 78        consumer.poll( timeout: 1000 );  
 79  
 80    try {  
 81        startTransaction();           //Start DB Transaction  
 82  
 83        consumer.commitSync();      //Commit the Kafka offset  
 84  
 85        processRecords(map, consumerRecords);  
 86  
 87        commitTransaction();        //Process the records  
 88    } catch( CommitFailedException ex ) {  
 89        logger.error("Failed to commit sync to log", ex);  
 90        rollbackTransaction();       //Rollback Transaction  
 91    } catch ( DatabaseException dte ) {  
 92        logger.error("Failed to write to DB", dte);  
 93        rollbackTransaction();       //Rollback Transaction  
 94    }  
 95  
 96 }
```

Fine Grained “At-Most-Once”



```
80  ↗💡 consumerRecords.forEach(record -> {
81
82     try {
83
83         startTransaction();           //Start DB Transaction
84
85         processRecord(record);
86
87         // Commit Kafka at exact location for record, and only this record.
88         final TopicPartition recordTopicPartition =
89             new TopicPartition(record.topic(), record.partition());
90
91         final Map<TopicPartition, OffsetAndMetadata> commitMap =
92             Collections.singletonMap(recordTopicPartition,
93                 new OffsetAndMetadata( offset: record.offset() + 1));
94
95         consumer.commitSync(commitMap);
96
97         commitTransaction();          //Commit DB Transaction
98     } catch (CommitFailedException ex) {
99         logger.error("Failed to commit sync to log", ex);
100        rollbackTransaction();       //Rollback Transaction
101    } catch (DatabaseException dte) {
102        logger.error("Failed to write to DB", dte);
103        rollbackTransaction();       //Rollback Transaction
104    }
105});
```



Fine Grained “At-Least-Once”

```
SimpleStockPriceConsumer pollRecordsAndProcess()  
78  
79  
80  ↗ consumerRecords.forEach(record -> {  
81    try {  
82      startTransaction();           //Start DB Transaction  
83  
84      // Commit Kafka at exact location for record, and only this record.  
85      final TopicPartition recordTopicPartition =  
86          new TopicPartition(record.topic(), record.partition());  
87  
88      final Map<TopicPartition, OffsetAndMetadata> commitMap =  
89          Collections.singletonMap(recordTopicPartition,  
90              new OffsetAndMetadata( offset: record.offset() + 1));  
91  
92      consumer.commitSync(commitMap); //Kafka Commit  
93  
94      processRecord(record);       //Process the record  
95  
96      commitTransaction();        //Commit DB Transaction  
97    } catch (CommitFailedException ex) {  
98      logger.error("Failed to commit sync to log", ex);  
99      rollbackTransaction();       //Rollback Transaction  
100 } catch (DatabaseException dte) {  
101   logger.error("Failed to write to DB", dte);  
102   rollbackTransaction();       //Rollback Transaction  
103 }  
104});
```

Consumer: Exactly Once, Saving Offset



- ❖ Consumer do not have to use Kafka's built-in offset storage
- ❖ Consumers can choose to store offsets with processed record output to make it “exactly once” message consumption
- ❖ If Consumer output of record consumption is stored in RDBMS then storing offset in database allows committing both process record output and location (partition/offset of record) in a single transaction implementing “exactly once” messaging.
- ❖ Typically to achieve “exactly once” you store record location with output of record

Saving Topic, Offset, Partition in DB



```
DatabaseUtilities saveStockPrice()  
22  
23     public static void saveStockPrice(final StockPriceRecord stockRecord,  
24                                         final Connection connection) throws SQLException {  
25  
26         final PreparedStatement preparedStatement = getUpsertPreparedStatement(  
27             stockRecord.getName(), connection);  
28  
29  
30         //Save partition, offset and topic in database.  
31         preparedStatement.setLong( parameterIndex: 1, stockRecord.getOffset());  
32         preparedStatement.setLong( parameterIndex: 2, stockRecord.getPartition());  
33         preparedStatement.setString( parameterIndex: 3, stockRecord.getTopic());  
34  
35         //Save stock price, name, dollars, and cents into database.  
36         preparedStatement.setInt( parameterIndex: 4, stockRecord.getDollars());  
37         preparedStatement.setInt( parameterIndex: 5, stockRecord.getCents());  
38         preparedStatement.setString( parameterIndex: 6, stockRecord.getName());  
39  
40         //Save the record with offset, partition, and topic.  
41         preparedStatement.execute();  
42  
43     }  
44 }
```

to exactly once, you need to save the offset and partition with the output of the consumer process.

“Exactly-Once” - Delivery Semantics



SimpleStockPriceConsumer pollRecordsAndProcess()

```
92
93     //Get rid of duplicates and keep only the latest record.
94     consumerRecords.forEach(record -> currentStocks.put(record.key(),
95                               new StockPriceRecord(record.value(), saved: false, record)));
96
97     final Connection connection = getConnection();
98     try {
99         startJdbcTransaction(connection);                      //Start DB Transaction
100        for (StockPriceRecord stockRecordPair : currentStocks.values()) {
101            if (!stockRecordPair.isSaved()) {                     //Save the record
102                saveStockPrice(stockRecordPair, connection);      // with partition/offset to DB.
103                //Mark the record as saved
104                currentStocks.put(stockRecordPair.getName(), new
105                                StockPriceRecord(stockRecordPair, saved: true));
106            }
107        }
108        consumer.commitSync();                                //Commit the Kafka offset
109        connection.commit();                                //Commit DB Transaction
110    } catch (CommitFailedException ex) {
111        logger.error("Failed to commit sync to log", ex);
112        connection.rollback();                            //Rollback Transaction
113    } catch (SQLException sqle) {
114        logger.error("Failed to write to DB", sqle);
115        connection.rollback();                            //Rollback Transaction
116    } finally {
117        connection.close();
118    }
119}
```

Move Offsets past saved Records



- ❖ If implementing “**exactly once**” message semantics, then you have to manage offset positioning
 - ❖ Pass ***ConsumerRebalanceListener*** instance in call to ***kafkaConsumer.subscribe(Collection, ConsumerRebalanceListener)*** and ***kafkaConsumer.subscribe(Pattern, ConsumerRebalanceListener)***.
 - ❖ when partitions taken from consumer, commit its offset for partitions by implementing ***ConsumerRebalanceListener.onPartitionsRevoked(Collection)***
 - ❖ When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ***ConsumerRebalanceListener.onPartitionsAssigned(Collection)***

Exactly Once - Move Offsets past saved Records



```
SeekToLatestRecordsConsumerRebalanceListener onPartitionsAssigned()  
16  
17 public class SeekToLatestRecordsConsumerRebalanceListener  
18     implements ConsumerRebalanceListener {  
19  
20     private final Consumer<String, StockPrice> consumer;  
21     private static final Logger logger = getLogger(SimpleStockPriceConsumer.class);  
22  
23     public SeekToLatestRecordsConsumerRebalanceListener(  
24         final Consumer<String, StockPrice> consumer) {  
25         this.consumer = consumer;  
26     }  
27  
28     @Override  
29     public void onPartitionsAssigned(final Collection<TopicPartition> partitions) {  
30         final Map<TopicPartition, Long> maxOffsets = getMaxOffsetsFromDatabase();  
31         maxOffsets.entrySet().forEach(  
32             entry -> partitions.forEach(topicPartition -> {  
33                 if (entry.getKey().equals(topicPartition)) {  
34                     long maxOffset = entry.getValue();  
35  
36                     // Call to consumer.seek to move to the partition.  
37                     consumer.seek(topicPartition, offset: maxOffset + 1);  
38  
39                     displaySeekInfo(topicPartition, maxOffset);  
40             }  
41         }));  
42     }
```

Kafka Consumer: Consumption Flow Control

- ❖ You can control consumption of topics using by using ***consumer.pause(Collection)*** and ***consumer.resume(Collection)***
 - ❖ This pauses or resumes consumption on specified assigned partitions for future ***consumer.poll(long)*** calls
- ❖ Use cases where consumers may want to first focus on fetching from some subset of assigned partitions at full speed, and only start fetching other partitions when these partitions have few or no data to consume
 - ❖ Priority queue like behavior from traditional MOM
 - ❖ Other cases is stream processing if preforming a join and one topic stream is getting behind another.

Kafka Consumer: MultiThreaded Process

- ❖ Kafka consumer is NOT thread-safe
- ❖ All network I/O happens in thread of the application making call
- ❖ Only exception thread safe method is ***consumer.wakeup()***
 - ❖ **forces** WakeupException to be thrown from thread blocking on operation
 - ❖ Use Case to shutdown consumer from another thread

Kafka Consumer: One Consumer Per Thread

- ❖ Pro
 - ❖ Easiest to implement
 - ❖ Requires no inter-thread co-ordination
 - ❖ In-order processing on a per-partition basis easy to implement
 - ❖ Process in-order that you receive them
- ❖ Con
 - ❖ More consumers means more TCP connections to the cluster (one per thread) - low cost has Kafka uses async IO and handles connections efficiently

One Consumer Per Thread: Runnable



StockPriceConsumerRunnable

```
13 import java.util.Map;
14 import java.util.concurrent.atomic.AtomicBoolean;
15
16 import static com.cloudurable.kafka.StockAppConstants.TOPIC;
17
18 
19 public class StockPriceConsumerRunnable implements Runnable{
20     private static final Logger logger =
21         LoggerFactory.getLogger(StockPriceConsumerRunnable.class);
22
23     private final Consumer<String, StockPrice> consumer;
24     private final int readCountStatusUpdate;
25     private final int threadIndex;
26     private final AtomicBoolean stopAll;
27     private boolean running = true;
28
29     @Override
30     public void run() {
31         try {
32             runConsumer();
33         } catch (Exception ex) {
34             logger.error("Run Consumer Exited with", ex);
35             throw new RuntimeException(ex);
36         }
37     }
38 }
```

One Consumer Per Thread: runConsumer



c StockPriceConsumerRunnable.java x

StockPriceConsumerRunnable pollRecordsAndProcess()

```
48
49     void runConsumer() throws Exception {
50         // Subscribe to the topic.
51         consumer.subscribe(Collections.singletonList(TOPIC));
52         final Map<String, StockPriceRecord> lastRecordPerStock = new HashMap<>();
53         try {
54             int readCount = 0;
55             while (isRunning()) {
56                 pollRecordsAndProcess(lastRecordPerStock, readCount);
57             }
58         } finally {
59             consumer.close();
60         }
61     }
```

One Consumer Per Thread: runConsumer



```
StockPriceConsumerRunnable pollRecordsAndProcess()  
64     private void pollRecordsAndProcess(  
65         final Map<String, StockPriceRecord> currentStocks,  
66         final int readCount) throws Exception {  
67  
68     final ConsumerRecords<String, StockPrice> consumerRecords =  
69         consumer.poll( timeout: 100 );  
70  
71     if (consumerRecords.count() == 0) {  
72         if (stopAll.get()) this.setRunning(false);  
73         return;  
74     }  
75  
76     consumerRecords.forEach(record -> currentStocks.put(record.key(),  
77                     new StockPriceRecord(record.value(), saved: true, record)));  
78  
79  
80     try {  
81         startTransaction();                                //Start DB Transaction  
82  
83         processRecords(currentStocks, consumerRecords);  
84         consumer.commitSync();                            //Commit the Kafka offset  
85         commitTransaction();                            //Commit DB Transaction  
86     } catch (CommitFailedException ex) {  
87         logger.error("Failed to commit sync to log", ex);  
88         rollbackTransaction();                         //Rollback Transaction  
89     }
```

One Consumer Per Thread: Thread Pool



```
ConsumerMain main()
48
49 ► ⚡ public static void main(String... args) throws Exception {
50     final int threadCount = 5;
51     final ExecutorService executorService = newFixedThreadPool(threadCount);
52     final AtomicBoolean stopAll = new AtomicBoolean();
53
54 ⚡ IntStream.range(0, threadCount).forEach(index -> {
55     final StockPriceConsumerRunnable stockPriceConsumer =
56         new StockPriceConsumerRunnable(createConsumer(),
57                                         readCountStatusUpdate: 10, index, stopAll);
58     executorService.submit(stockPriceConsumer);
59});
```

KafkaConsumer: One Consumer with Worker Threads



- ❖ Decouple Consumption and Processing: One or more consumer threads that consume from Kafka and hands off to ConsumerRecords instances to a blocking queue processed by a processor thread pool that process the records.
- ❖ PROs
 - ❖ This option allows independently scaling consumers count and processors count. Processor threads are independent of topic partition count
- ❖ CONS
 - ❖ Guaranteeing order across processors requires care as threads execute independently a later record could be processed before an earlier record and then you have to do consumer commits somehow
 - ❖ How do you committing the position unless there is some ordering? You have to provide the ordering. (Concurrently HashMap of BlockingQueues where topic, partition is the key (TopicPartition)?)

One Consumer with Worker Threads



StockPriceConsumerRunnable

```
18  public class StockPriceConsumerRunnable implements Runnable{  
19      private static final Logger logger =  
20          LoggerFactory.getLogger(StockPriceConsumerRunnable.class);  
21  
22      private final Consumer<String, StockPrice> consumer;  
23      private final int readCountStatusUpdate;  
24      private final int threadIndex;  
25      private final AtomicBoolean stopAll;  
26      private boolean running = true;  
27  
28      //Store blocking queue by TopicPartition.  
29      private final Map<TopicPartition, BlockingQueue<ConsumerRecord>>  
30          commitQueueMap = new ConcurrentHashMap<>();  
31  
32      //Worker pool.  
33      private final ExecutorService threadPool;  
34  
35
```



Worker

```
StockPriceConsumerRunnable pollRecordsAndProcess()
72     private void pollRecordsAndProcess(
73         final Map<String, StockPriceRecord> currentStocks,
74         final int readCount) throws Exception {
75
76     final ConsumerRecords<String, StockPrice> consumerRecords =
77         consumer.poll( timeout: 100 );
78
79     if (consumerRecords.count() == 0) {
80         if (stopAll.get()) this.setRunning(false);
81         return;
82     }
83
84     consumerRecords.forEach(record ->
85         currentStocks.put(record.key(),
86             new StockPriceRecord(record.value(), saved: true, record)
87         ));
88
89     threadPool.execute(() ->
90         processRecords(currentStocks, consumerRecords));
91
92
```

processCommits

```
StockPriceConsumerRunnable processCommits()
120     private void processCommits() {
121
122     commitQueueMap.entrySet().forEach(queueEntry -> {
123         final BlockingQueue<ConsumerRecord> queue = queueEntry.getValue();
124         final TopicPartition topicPartition = queueEntry.getKey();
125
126         ConsumerRecord consumerRecord = queue.poll();
127         ConsumerRecord highestOffset = consumerRecord;
128
129         while (consumerRecord != null) {
130             if (consumerRecord.offset() > highestOffset.offset()) {
131                 highestOffset = consumerRecord;
132             }
133             consumerRecord = queue.poll();
134         }
135
136         if (highestOffset != null) {
137             logger.info(String.format("Sending commit %s %d",
138                                     topicPartition, highestOffset.offset()));
139             try {
140                 consumer.commitSync(Collections.singletonMap(topicPartition,
141                                               new OffsetAndMetadata(highestOffset.offset())));
142             } catch (CommitFailedException cfe) {
143                 logger.info("Failed to commit record", cfe);
144             }
145         }
146     }
147 }
```

- ❖ Creating Priority processing queue
- ❖ Use **consumer.partitionsFor(TOPIC)** to get a list of partitions
- ❖ Usage like this simplest when the partition assignment is also done manually using **assign()** instead of **subscribe()**
 - ❖ Use assign(), pass TopicPartition to worker
 - ❖ Use Partitioner from earlier example for Producer

Using partitionsFor() for Priority Queue



ConsumerMain main()

```
49
50  public static void main(String... args) throws Exception {
51
52      final AtomicBoolean stopAll = new AtomicBoolean();
53      final Consumer<String, StockPrice> consumer = createConsumer();
54
55      //Get the partitions.
56      final List<PartitionInfo> partitionInfos = consumer.partitionsFor(TOPIC);
57
58      final int threadCount = partitionInfos.size();
59      final int numWorkers = 5;
60      final ExecutorService executorService = newFixedThreadPool(threadCount);
61
62
63      IntStream.range(0, threadCount).forEach(index -> {
64          final PartitionInfo partitionInfo = partitionInfos.get(index);
65
66
67          final StockPriceConsumerRunnable stockPriceConsumer =
68              new StockPriceConsumerRunnable(partitionInfo, createConsumer(),
69                  readCountStatusUpdate: 10, index, stopAll, workerCount);
70          executorService.submit(stockPriceConsumer);
71      });
}
```

Using assign() for Priority Queue



```
StockPriceConsumerRunnable runConsumer()
```

```
61     void runConsumer() throws Exception {
62         // Assign a partition.
63         consumer.assign(Collections.singleton(topicPartition));
64         final Map<String, StockPriceRecord> lastRecordPerStock = new HashMap<
65         try {
66             int readCount = 0;
67             while (isRunning()) {
68                 pollRecordsAndProcess(lastRecordPerStock, readCount);
69             }
70         } finally {
71             consumer.close();
72         }
73     }
```

Complete Labs 6.1-6.7

19. Avro and the Schema Registry

Confluent Schema Registry

- ❖ Confluent Schema Registry stores Avro Schemas for Kafka clients
- ❖ Provides REST interface for putting and getting Avro schemas
- ❖ Stores a history of schemas
 - ❖ versioned
 - ❖ allows you to configure compatibility setting
 - ❖ supports evolution of schemas
- ❖ Provides serializers used by Kafka clients which handles schema storage and serialization of records using Avro

Why Schema Registry?

- ❖ Producer creates a record/message, which is an Avro record
- ❖ Record contains the schema and data
- ❖ Schema Registry Avro Serializer serializes the data and schema id (just id)
 - ❖ Keeps a cache of registered schemas from Schema Registry to ids
- ❖ Consumer receives payload and deserializes it with Schema Registry Avro Deserializers
- ❖ Deserializer looks up the full schema from cache or Schema Registry based on id
- ❖ Consumer has its schema, one it is expecting record/message to conform to
 - ❖ Compatibility check is performed on two schemas
 - ❖ if no match, but are compatible, then payload transformation happens aka Schema Evolution
 - ❖ if not failure
- ❖ Kafka records have Key and Value and schema can be done on both

Schema Registry Config

- ❖ Compatibility can be configured globally or per schema
- ❖ Options are:
 - ❖ NONE - don't check for schema compatibility
 - ❖ FORWARD - check to make sure last schema version is forward compatible with new schemas
 - ❖ BACKWARDS (default) - make sure new schema is backwards compatible with latest
 - ❖ FULL - make sure new schema is forwards and backwards compatible from latest to new and from new to latest

Schema Registry Actions

- ❖ Register schemas for key and values of Kafka records
- ❖ List schemas (subjects)
- ❖ List all versions of a subject (schema)
- ❖ Retrieve a schema by version or id
 - ❖ get latest version of schema
- ❖ Check to see if schema is compatible with a certain version
- ❖ Get the compatibility level setting of the Schema Registry
 - ❖ BACKWARDS, NONE
- ❖ Add compatibility settings to a subject/schema

Schema Evolution

- ❖ Avro schema is changed after data has been written to store using an older version of that schema, then Avro might do a Schema Evolution
- ❖ Schema evolution is automatic transformation of Avro schema
 - ❖ transformation is between version of consumer schema and what the producer put into the Kafka log
 - ❖ When Consumer schema is not identical to the Producer schema used to serialize the Kafka Record then a data transformation is performed on the Kafka record (key or value)
 - ❖ If the schemas match then no need to do a transformation
 - ❖ Schema evolution is happens only during deserialization at the Consumer
 - ❖ If Consumer's schema is different from Producer's schema, then value or key is automatically modified during deserialization to conform to consumers reader schema

Allowed Schema Modifications

- ❖ Add a field with a default
- ❖ Remove a field that had a default value
- ❖ Change a fields order attribute
- ❖ Change a fields default value
- ❖ Remove or add a field alias
- ❖ Remove or add a type alias
- ❖ Change a type to a union that contains original type

Rules of the Road for modifying Schema

- ❖ Provide a default value for fields in your schema
 - ❖ Allows you to delete the field later
- ❖ Don't change a field's data type
- ❖ When adding a new field to your schema, you have to provide a default value for the field
- ❖ Don't rename an existing field
 - ❖ You can add an alias

Remember our example Employee



The screenshot shows an IDE interface with a project structure on the left and a code editor on the right. The project structure under the 'avro' folder includes '.gradle', '.idea', 'build' (containing 'classes' and 'dependency-cache'), 'generated-main-avro-java [main]' (containing 'com.cloudurable.phonebook' which has an 'Employee' file), 'libs', 'tmp', 'gradle', 'src' (containing 'main' which has an 'avro' folder and 'com.cloudurable.phonebook' containing 'Employee.avsc'), and 'Employee.avsc'. The code editor displays the Avro schema for 'Employee':

```
1 {"namespace": "com.cloudurable.phonebook",
2   "type": "record",
3   "name": "Employee",
4   "fields": [
5     {"name": "firstName", "type": "string"},
6     {"name": "nickName", "type": ["string", "null"]},
7     {"name": "lastName", "type": "string"},
8     {"name": "age", "type": "int"},
9     {"name": "phoneNumber", "type": "string"}]
```

Avro covered in [Avro/Kafka Tutorial](#)

Let's say

- ❖ Employee did not have an age in version 1 of the schema
- ❖ Later we decided to add an age field with a default value of -1
- ❖ Now let's say we have a Producer using version 2, and a Consumer using version 1

Scenario adding a new field age with default value

- ❖ Producer uses version 2 of the Employee schema and creates a com.cloudurable.Employee record, and sets age field to 42, then sends it to Kafka topic new-employees
- ❖ Consumer consumes records from new-employees using version 1 of the Employee Schema
 - ❖ Since Consumer is using version 1 of schema, age field is removed during deserialization
 - ❖ Same consumer modifies name field and then writes the record back to a NoSQL store
 - ❖ When it does this, the age field is missing from value that it writes to the store
 - ❖ Another client using version 2 reads the record from the NoSQL store
 - ❖ Age field is missing from the record (because the Consumer wrote it with version 1), age is set to default value of -1

Schema Registry Actions

- ❖ Register schemas for key and values of Kafka records
- ❖ List schemas (subjects)
- ❖ List all versions of a subject (schema)
- ❖ Retrieve a schema by version or id
 - ❖ get latest version of schema
- ❖ Check to see if schema is compatible with a certain version
- ❖ Get the compatibility level setting of the Schema Registry
 - ❖ BACKWARDS, FORWARD, FULL, NONE
- ❖ Add compatibility settings to a subject/schema



Register a Schema

```
private final static MediaType SCHEMA_CONTENT =
    MediaType.parse("application/vnd.schemaregistry.v1+json");

private final static String EMPLOYEE_SCHEMA = "{\n" +
    "  \"schema\": \"\"\" +\n    \"{\" +\n      \"namespace\": \"com.cloudurable.phonebook\", \" +\n      \"type\": \"record\", \" +\n      \"name\": \"Employee\", \" +\n      \"fields\": [\" +\n        {\"name\": \"firstName\", \"type\": \"string\"}, \" +\n        {\"name\": \"lastName\", \"type\": \"string\"}, \" +\n        {\"name\": \"age\", \"type\": \"int\"}, \" +\n        {\"name\": \"phoneNumber\", \"type\": \"string\"}\" +\n      ]\" +\n    }\"\"\" +\n  \"\"\"};
```

Register a Schema



```
final OkHttpClient client = new OkHttpClient();

//POST A NEW SCHEMA
Request request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/subjects/Employee/versions")
    .build();

String output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
curl -X POST -H "Content-Type: application/vnd.schema.registry.v1+json"
--data '{"schema": "{\"type\": ..."}' \
http://localhost:8081/subjects/Employee/versions
```

{"id":2}



List All Schema

```
//LIST ALL SCHEMAS
request = new Request.Builder()
    .url("http://localhost:8081/subjects")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
curl -X GET http://localhost:8081/subjects
```

["Employee","Employee2","FooBar"]

Working with versions

```
//SHOW ALL VERSIONS OF EMPLOYEE
request = new Request.Builder()
    .url("http://localhost:8081/subjects/Employee/versions/")
    .build(); [1,2,3,4,5]

output = client.newCall(request).execute().body().string();
System.out.println(output);

//SHOW VERSION 2 OF EMPLOYEE
request = new Request.Builder()
    .url("http://localhost:8081/subjects/Employee/versions/2")
    .build(); {"subject":"Employee","version":2,"id":4,"schema":
    {"type":"record","name":"Employee",
    "namespace":"com.cloudurable.phonebook",...}

output = client.newCall(request).execute().body().string();
System.out.println(output);

//SHOW THE SCHEMA WITH ID 3
request = new Request.Builder()
    .url("http://localhost:8081/schemas/ids/3")
    .build(); {"subject":"Employee","version":1,"id":3,"schema":
    {"type":"record","name":"Employee",
    "namespace":"com.cloudurable.phonebook",...}

output = client.newCall(request).execute().body().string();
System.out.println(output);
```

Working with Schemas



```
//SHOW THE LATEST VERSION OF EMPLOYEE 2
request = new Request.Builder()
    .url("http://localhost:8081/subjects/Employee/versions/latest")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);

//CHECK IF SCHEMA IS REGISTERED
request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/subjects/Employee")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);

//TEST COMPATIBILITY
request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/compatibility/subjects/Employee/versions/latest")
    .build();
```

Changing Compatibility Checks



```
// SET TOP LEVEL CONFIG
// VALUES are none, backward, forward and full
request = new Request.Builder()
    .put(RequestBody.create(SCHEMA_CONTENT, "{\"compatibility\": \"none\"}"))
    .url("http://localhost:8081/config")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);

// SET CONFIG FOR EMPLOYEE
// VALUES are none, backward, forward and full
request = new Request.Builder()
    .put(RequestBody.create(SCHEMA_CONTENT, "{\"compatibility\": \"backward\"}"))
    .url("http://localhost:8081/config/Employee")
    .build();

output = client.newCall(request).execute().body().string();
System.out.println(output);
```

Incompatible Change

```
private final static String EMPLOYEE_SCHEMA = "{\n    \"schema\": \"\" +\n    \"{\" +\n        \"\\\\\\\"namespace\\\\\\\": \\\\\"com.cloudurable.phonebook\\\\\",\" +\n        \"\\\\\\\"type\\\\\\\": \\\\\"record\\\\\",\" +\n        \"\\\\\\\"name\\\\\\\": \\\\\"Employee\\\\\",\" +\n        \"\\\\\\\"fields\\\\\\\": [\" +\n            {\\\\\\\"name\\\\\\\": \\\\\"fName\\\\\", \\\\\"type\\\\\\\": \\\\\"string\\\\\"},\" +\n            {\\\\\\\"name\\\\\\\": \\\\\"lName\\\\\", \\\\\"type\\\\\\\": \\\\\"string\\\\\"},\" +\n            {\\\\\\\"name\\\\\\\": \\\\\"age\\\\\", \\\\\"type\\\\\\\": \\\\\"int\\\\\"},\" +\n            {\\\\\\\"name\\\\\\\": \\\\\"phoneNumber\\\\\", \\\\\"type\\\\\\\": \\\\\"string\\\\\"}\" +\n        ]\" +\n    }\" +\n}";
```

```
//POST A NEW SCHEMA
Request request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/subjects/Employee/versions")
    .build();

String output = client.newCall(request).execute().body().string();
System.out.println(output);
```

```
{"error_code":409,"  
message":"Schema being registered is incompatible with an e
```

Incompatible Change

```
private final static String EMPLOYEE_SCHEMA = "{\n    \"schema\": \"\" +\n    \"{\" +\n        \"\\\"namespace\\\": \\\"com.cloudurable.phonebook\\\",\" +\n        \"\\\"type\\\": \\\"record\\\",\" +\n        \"\\\"name\\\": \\\"Employee\\\",\" +\n        \"\\\"fields\\\": [\" +\n            {\\\"name\\\": \\\"fName\\\", \\\"type\\\": \\\"string\\\"},\" +\n            {\\\"name\\\": \\\"lName\\\", \\\"type\\\": \\\"string\\\"},\" +\n            {\\\"name\\\": \\\"age\\\", \\\"type\\\": \\\"int\\\"},\" +\n            {\\\"name\\\": \\\"phoneNumber\\\", \\\"type\\\": \\\"string\\\"}\" +\n        ]\" +\n    }\" +\n};"
```

```
//TEST COMPATIBILITY
request = new Request.Builder()
    .post(RequestBody.create(SCHEMA_CONTENT, EMPLOYEE_SCHEMA))
    .url("http://localhost:8081/compatibility/subjects/Employee/versions/latest")
    .build();
```

```
{"is_compatible":false}
```

Use Schema Registry

- ❖ Start up Schema Registry server pointing to Zookeeper cluster
- ❖ Import Kafka Avro Serializer and Avro Jars
- ❖ Configure Producer to use Schema Registry
- ❖ Use KafkaAvroSerializer from Producer
- ❖ Configure Consumer to use Schema Registry
- ❖ Use KafkaAvroDeserializer from Consumer

Start up Schema Registry Server

```
cat ~/tools/confluent-3.2.1/etc/schema-registry/schema-registry.properties
```

```
listeners=http://0.0.0.0:8081
kafkastore.connection.url=localhost:2181
kafkastore.topic=_schemas
debug=false
```

```
$ bin/schema-registry-start ~/tools/confluent-3.2.1/etc/schema-registry/schema-registry.properties
[2017-05-09 15:45:34,055] INFO SchemaRegistryConfig values:
  metric.reporters = []
  kafkastore.sasl.kerberos.kinit.cmd = /usr/bin/kinit
  response.mediatype.default = application/vnd.schemaregistry.v1+json
  kafkastore.ssl.trustmanager.algorithm = PKIX
  authentication.realm =
  ssl.keystore.type = JKS
  kafkastore.topic = _schemas
  metrics.jmx.prefix = kafka.schema.registry
```

Import Kafka Avro Serializer & Avro Jars



The screenshot shows a Java project structure in an IDE. The project is named "schema-registry". The "src" directory contains "main" and "com" packages. The "main" package has "avro" and "java" sub-directories. The "avro" directory contains a JSON file named "emp". The "java" directory contains three classes: "Employee", "Person", and "SchemaUtil". The "build.gradle" file is open in the editor, showing the following code:

```
group 'cludurable'
version '1.0-SNAPSHOT'
apply plugin: 'java'
sourceCompatibility = 1.8

dependencies {
    compile "org.apache.avro:avro:1.8.1"
    compile 'com.squareup.okhttp3:okhttp:3.7.0'
    testCompile 'junit:junit:4.11'
    compile 'org.apache.kafka:kafka-clients:0.10.2.0'
    compile 'io.confluent:kafka-avro-serializer:3.2.1'
}

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "http://packages.confluent.io/maven/"
    }
}

avro {
    createSetters = false
    fieldVisibility = "PRIVATE"
}
```

Configure Producer to use Schema Registry



schema-registry > src > main > java > com > cludurable > kafka > schema > AvroProducer

Project + - ⚙ ↻

com.cludurable.phonebo
 Employee
 PhoneNumber
 Status

libs
resources
 main
tmp
gradle
src
 main
 avro
 employee.avsc
 java
 com.cludurable.kafka
 AvroConsumer
 AvroProducer
 SchemaMain

build.gradle
gradlew
gradlew.bat
settings.gradle
External Libraries
< 1.8 > /Library/Java/JavaVirtualMachine
Gradle: com.101tec:zkclient:0.10
Gradle: com.fasterxml.jackson.core:jackson-core:2.10.2
Gradle: com.fasterxml.jackson.core:jackson-databind:2.10.2
Gradle: com.fasterxml.jackson.core:jackson-annotations:2.10.2
Gradle: com.squareup.okhttp3:okhttp:4.3.1

AvroProducer

```
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.LongSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializer;

import java.util.Properties;
import java.util.stream.IntStream;

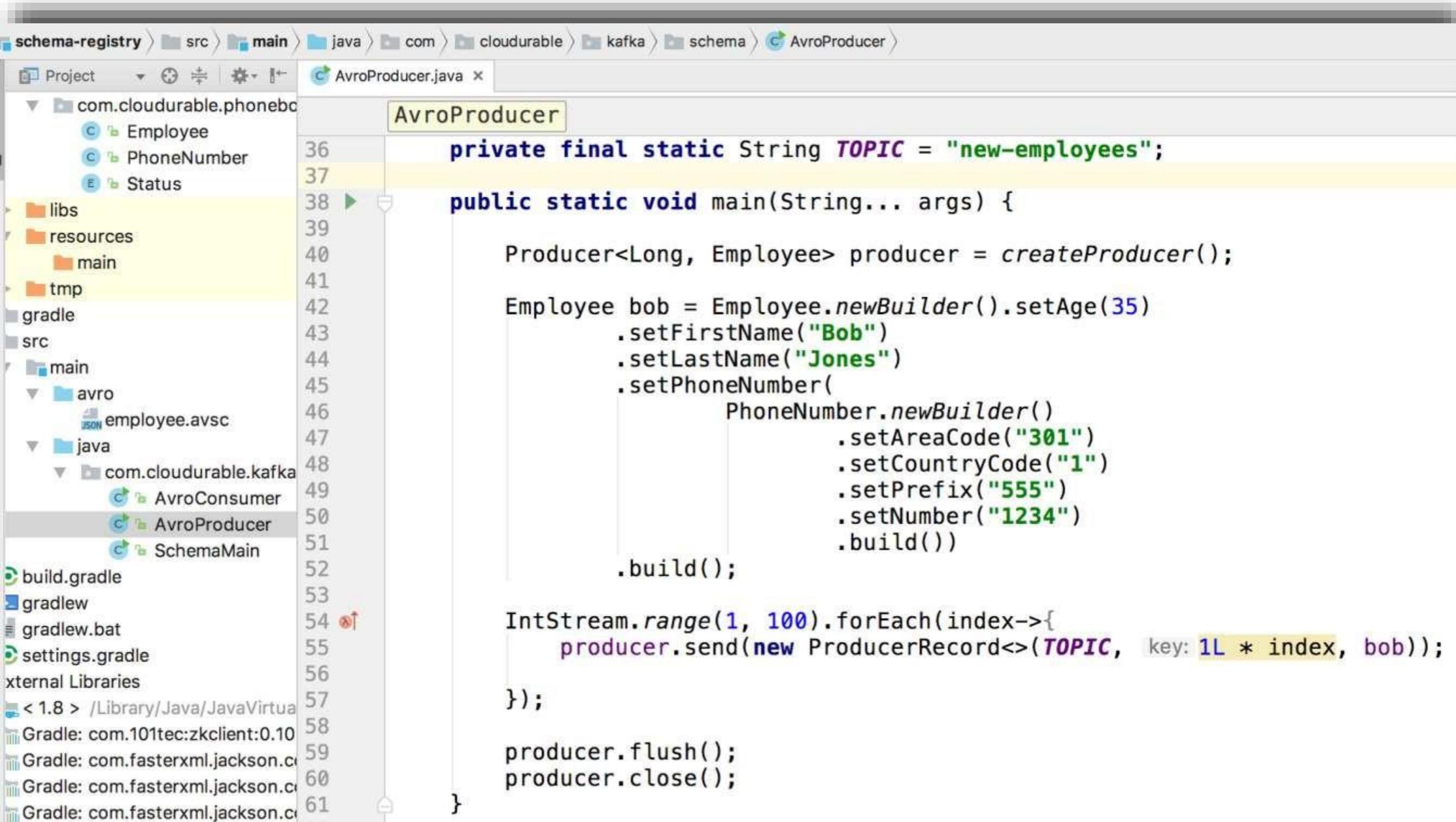
public class AvroProducer {

    private static Producer<Long, Employee> createProducer() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "AvroProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                  LongSerializer.class.getName());

        // Configure the KafkaAvroSerializer.
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                  KafkaAvroSerializer.class.getName());
    }

    return new KafkaProducer<>(props);
}
```

Use KafkaAvroSerializer from Producer



The screenshot shows a Java code editor with the file `AvroProducer.java` open. The code is part of a project named `schema-registry`. The code itself is as follows:

```
private final static String TOPIC = "new-employees";

public static void main(String... args) {
    Producer<Long, Employee> producer = createProducer();

    Employee bob = Employee.newBuilder().setAge(35)
        .setFirstName("Bob")
        .setLastName("Jones")
        .setPhoneNumber(
            PhoneNumber.newBuilder()
                .setAreaCode("301")
                .setCountryCode("1")
                .setPrefix("555")
                .setNumber("1234")
                .build())
        .build();

    IntStream.range(1, 100).forEach(index->{
        producer.send(new ProducerRecord<>(TOPIC, key: 1L * index, bob));
    });

    producer.flush();
    producer.close();
}
```

The code defines a `TOPIC` constant and a `main` method. Inside the `main` method, it creates a `producer` using `createProducer()`. It then creates an `Employee` object named `bob` with various fields set. It uses `IntStream.range(1, 100)` to iterate and send `ProducerRecord`s to the `TOPIC` with a key of `1L * index` and the `bob` object as value. Finally, it calls `producer.flush()` and `producer.close()`.

Configure Consumer to use Schema Registry



```
AvroConsumer.java x
AvroConsumer
16 > public class AvroConsumer {
17
18     private final static String BOOTSTRAP_SERVERS = "localhost:9092";
19     private final static String TOPIC = "new-employees";
20
21     private static Consumer<Long, Employee> createConsumer() {
22         Properties props = new Properties();
23         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
24         props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleAvroConsumer");
25         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
26                   LongDeserializer.class.getName());
27
28
29
30
31
32     //Use Specific Record or else you get Avro GenericRecord.
33     props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
34
35
36     //Schema registry location.
37     props.put(KafkaAvroDeserializerConfig.SCHEMA_REGISTRY_URL_CONFIG,
38               "http://localhost:8081"); //----- Run Schema Registry on 8081
39
40
41     return new KafkaConsumer<>(props);
42 }
43
```

Use KafkaAvroDeserializer from Consumer



```
public static void main(String... args) {  
  
    final Consumer<Long, Employee> consumer = createConsumer();  
    consumer.subscribe(Collections.singletonList(TOPIC));  
  
    IntStream.range(1, 100).forEach(index -> {  
  
        final ConsumerRecords<Long, Employee> records =  
            consumer.poll( timeout: 100);  
  
        if (records.count() == 0) {  
            System.out.println("None found");  
        } else records.forEach(record -> {  
  
            Employee employeeRecord = record.value();  
  
            System.out.printf("%s %d %d %s \n", record.topic(),  
                record.partition(), record.offset(), employeeRecord);  
        });  
    });  
}
```

Schema Registry

- ❖ Confluent provides Schema Registry to manage Avro Schemas for Kafka Consumers and Producers
- ❖ Avro provides Schema Migration
- ❖ Confluent uses Schema compatibility checks to see if Producer schema and Consumer schemas are compatible and to do Schema evolution if needed
- ❖ Use KafkaAvroSerializer from Producer
- ❖ Use KafkaAvroDeserializer from Consumer

Complete Lab 7.2

20. Security



Kafka Security

- ❖ Authentication
 - ❖ SASL, SSL
- ❖ Authorization
 - ❖ Pluggable
- ❖ Encryption
 - ❖ Communication SSL
- ❖ ACLS



Authentication

- ❖ Kafka Broker Authentication
 - ❖ producers and consumers, brokers, tools
 - ❖ SSL or SASL
- ❖ SASL is Simple Authentication and Security Layer
- ❖ Kafka supports the following SASL mechanisms:
 - ❖ SASL/GSSAPI Kerberos
 - ❖ SASL/PLAIN
 - ❖ SASL/SCRAM-SHA-256 and SASL/SCRAM-SHA-512
- ❖ ZooKeeper Authentication
 - ❖ brokers to ZooKeeper



Encryption and Authorization

- ❖ Encryption of data transferred (using SSL)
 - ❖ broker, producer, consumers using
- ❖ Authorization
 - ❖ read/write operations
- ❖ Pluggable Authorization
 - ❖ integration with 3rd party providers

21. SSL

SSL/TLS Overhead



- ❖ SSL/TLS have some overhead
 - ❖ Especially true in JVM world which is not as performant for handling SSL/TLS unless you are using Netty/OpenSSL integration
 - ❖ Understanding SSL/TLS support for Kafka is important for developers, DevOps and DBAs
 - ❖ If possible, use no encryption for cluster communication, and deploy your Kafka Cluster Broker nodes in a private subnet, and limit access to this subnet to client transport
 - ❖ Also if possible avoid using TLS/SSL on client transport and do client operations from your app tier, which is located in a non-public subnet

Sometimes you have to encrypt



- ❖ However, that is not always possible to avoid TLS/SSL
- ❖ Regulations and commons sense
 - ❖ U.S. Health Insurance Portability and Accountability Act (HIPAA), Germany's Federal Data Protection Act, The Payment Card Industry Data Security Standard (PCI DSS), or U.S. Sarbanes-Oxley Act of 2002
 - ❖ Or you might work for a bank or other financial institution
 - ❖ Or it just might be a corporate policy to encrypt such transports.
- ❖ Kafka has essential security features: authentication, role-based authorization, transport encryption, but is ***missing data at rest encryption***, up to you to encrypt records via OS file systems or use AWS KMS to encrypt EBS volume



Encrypting client transports

- ❖ Data that travels over the client transport across a network could be accessed by someone you don't want accessing it
 - ❖ with tools like wire shark.
 - ❖ If data includes private information, SSN number, credentials (password, username), credit card numbers or account numbers, then we want to make that data unintelligible (encrypted) to any and all 3rd parties
 - ❖ ***especially important if we don't control the network***
- ❖ You can also use TLS/SSL to ensure data has not been tampered with whilst traveling network
 - ❖ Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols - designed to provide these features
- ❖ Kafka is written in Java so uses JSSE framework for TLS support
 - ❖ which in turn uses the Java Cryptography Architecture (JCA)
- ❖ Truststores are for Authentication
- ❖ Keystores are for Encryption

Avoid Man in the middle attacks



- ❖ Kafka Broker Config:
 - ❖ ***ssl.endpoint.identification.algorithm=HTTPS***
- ❖ Default `ssl.endpoint.identification.algorithm` is `null`
 - ❖ not a secure default
 - ❖ possible: man in the middle attacks
- ❖ HTTPS better option
 - ❖ producers and consumers verify server's fully qualified domain name (FQDN) against Common Name (CN) or Subject Alternative Name (SAN)

Certificate Authority



- ❖ Each Kafka Broker in cluster has a public-private key pair, and a certificate to identify broker
- ❖ To prevent forged certificates, you have to sign them
- ❖ Certificate authority (CA) signs certificates
- ❖ CA signs certificates
 - ❖ signed certificates are hard to forge
 - ❖ If you trust the CA, clients (producers, consumers, other brokers) can trust the authenticity of Kafka brokers

Steps to use SSL for Consumer and Producers



- ❖ Generate SSL key and certificate for each Kafka broker
- ❖ Generate ***cluster certificate*** into a keystore use ***keytool***
- ❖ Generate or use CA (Certificate Authority) use ***openssl***
- ❖ Import CA into Kafka's ***truststore*** use ***keytool***
- ❖ Sign cluster certificate with CA use ***openssl***
- ❖ Import **CA** and ***signed cluster certificate*** into Kafka's ***keystore*** use ***keytool***
- ❖ Run ***bin/create-ssl-key-keystore.sh*** copy files to /opt/kafka
- ❖ Configure servers (Kafka Broker)
- ❖ Configure clients (Kafka Producers, Kafka Consumers)



Common Variables for SSL Key Gen

```
1 #!/usr/bin/env bash
2 set -e
3
4 CERT_OUTPUT_PATH="$PWD/resources/opt/kafka/conf/certs"
5 KEY_STORE="$CERT_OUTPUT_PATH/kafka.keystore"
6 TRUST_STORE="$CERT_OUTPUT_PATH/kafka.truststore"
7 PASSWORD=kafka123
8 KEY_KEY_PASS="$PASSWORD"
9 KEY_STORE_PASS="$PASSWORD"
10 TRUST_KEY_PASS="$PASSWORD"
11 TRUST_STORE_PASS="$PASSWORD"
12 CLUSTER_NAME=kafka
13 CERT_AUTH_FILE="$CERT_OUTPUT_PATH/ca-cert"
14 CLUSTER_CERT_FILE="$CERT_OUTPUT_PATH/${CLUSTER_NAME}-cert"
15 D_NAME="CN=CloudDurable Image $CLUSTER_NAME cluster, OU=Cloudurable, O=Cloudurable"
16 D_NAME="${D_NAME}, L=San Francisco, ST=CA, C=USA, DC=cloudurable, DC=com"
17 DAYS_VALID=365
18
19 mkdir -p "$CERT_OUTPUT_PATH"
```

❖ bin/create-ssl-key-keystore.sh

Generate cluster certificate into a keystore



```
22 echo "Create the cluster key for cluster communication."
23 keytool -genkey -keyalg RSA -alias "${CLUSTER_NAME}_cluster" \
24 -keystore "$KEY_STORE" -storepass "$KEY_STORE_PASS" \
25 -keypass "$KEY_KEY_PASS" -dname "$D_NAME" -validity "$DAYS_VALID"
```

- ❖ **keytool** ships with Java used for SSL/TLS
- ❖ **—genkey** generate a key
- ❖ **—keystore** location of keystore to add the key
- ❖ **—keyalg** RSA use the RSA algorithm for the key
- ❖ **—alias** Alias of the key we use this later to extract and sign key
- ❖ **—storepass** password for the keystore, **—keypass** password for key
- ❖ **—validity** how many days is this key valid



Generate Certificate Authority - CA

```
27 echo "Create the Certificate Authority (CA) file to sign keys."  
28 openssl req -new -x509 -keyout ca-key -out "$CERT_AUTH_FILE"  
29 -days "$DAYS_VALID" \  
30 -passin pass:"$PASSWORD" -passout pass:"$PASSWORD" \  
31 -subj "/C=US/ST=CA/L=San Francisco/O=Engineering/CN=cloudurable.com"
```

- ❖ —req —new -x509 - create a new CA file using x509 format
- ❖ X.509 certificate contains a public key and an identity
 - ❖ identify is hostname, or an organization, or an individual
 - ❖ and is either signed by a certificate authority or self-signed
- ❖ —days how many days is this certificate valid
- ❖ —passin pass: / —passout pass: Passwords to access the certificate
- ❖ —subj Pass identity information about the certificate

Import CA into Kafka's truststore



```
33 echo "Import the Certificate Authority file into the trust store."  
34 keytool -keystore "$TRUST_STORE" -alias CARoot \  
35   -import -file "$CERT_AUTH_FILE" \  
36   -storepass "$TRUST_STORE_PASS" -keypass "$TRUST_KEY_PASS" \  
37   -noprompt
```

- ❖ `-import -file $CERT_AUTH_FILE` is CA file we generated in the last step
- ❖ `-keystore` is location of *trust* keystore file
- ❖ **—storepass** password for the keystore, **—keypass** password for key



Sign cluster certificate with CA

```
40 echo "Export the cluster certificate from the key store."  
41 keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}_cluster" \  
42     -certreq -file "$CLUSTER_CERT_FILE" \  
43     -storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt  
44  
45 echo "Sign the cluster certificate with the CA."  
46 openssl x509 -req -CA "$CERT_AUTH_FILE" -CAkey ca-key \  
47     -in "$CLUSTER_CERT_FILE" -out "${CLUSTER_CERT_FILE}-signed" \  
48     -days "$DAYS_VALID" -CAcreateserial -passin pass:"$PASSWORD"
```

- ❖ Export the CLUSTER_CERT_FILE from the first step from the keystore
- ❖ Then sign the CLUSTER_CERT_FILE with the CA

Import CA and Signed Cluster Certificate into Kafka's keystore



```
51 echo "Import the Certificate Authority (CA) file into the key store."  
52 keytool -keystore "$KEY_STORE" -alias CARoot -import -file "$CERT_AUTH_FILE" \  
53     -storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt  
54  
55 echo "Import the Signed Cluster Certificate into the key store."  
56 keytool -keystore "$KEY_STORE" -alias "${CLUSTER_NAME}_cluster" \  
57     -import -file "${CLUSTER_CERT_FILE}-signed" \  
58     -storepass "$KEY_STORE_PASS" -keypass "$KEY_KEY_PASS" -noprompt
```

- ❖ Import the CA file into keystore
 - ❖ it was already imported into the truststore
- ❖ Import the signed version of the cluster certificate into the keystore
 - ❖ This was the file we create in the last step

Generate / then Copy or move files so Kafka can see them



- ❖ Copy and/or move files so that each Broker, Producer or Consumer has access to /opt/kafka/conf/certs/

~/kafka-training/lab8/solution

```
$ bin/create-ssl-key-keystore.sh
```

Create the cluster key for cluster communication.

Create the Certificate Authority (CA) file to sign keys.

Generating a 1024 bit RSA private key
writing new private key to 'ca-key'

...

Certificate was added to keystore

Import the Signed Cluster Certificate into the key store.

Certificate reply was installed in keystore

```
$ sudo cp -R resources/opt/kafka/ /opt/
```



Files generated

```
$ ls /opt/kafka/conf/certs/
ca-cert                           kafka-cert
kafka-cert-signed                 kafka.keystore
kafka.truststore
```

- ❖ **ca-cert** - Certificate Authority file - ***don't ship this around***
- ❖ **kafka-cert** - Kafka Certification File - public key and private key, don't ship this around
- ❖ **kafka-cert-signed** - Kafka Certification File signed with CA - don't ship around
- ❖ **kafka.keystore** - needed on all clients and servers
- ❖ **kafka.truststore** - needed on all clients and servers



Configure servers (Kafka Broker)

```
server-0.properties x
1 broker.id=0
2 listeners=PLAINTEXT://localhost:9092,SSL://localhost:10092
3 ssl.keystore.location=/opt/kafka/conf/certs/kafka.keystore
4 ssl.keystore.password=kafka123
5 ssl.key.password=kafka123
6 ssl.truststore.location=/opt/kafka/conf/certs/kafka.truststore
7 ssl.truststore.password=kafka123
8 ssl.client.auth=required
9 security.inter.broker.protocol=SSL

server-0.properties x server-1.properties x server-2.properties x
1 broker.id=2
2 listeners=PLAINTEXT://localhost:9094,SSL://localhost:10094
```

- ❖ Listeners configure protocols - Making Kafka available on SSL and plaintext
- ❖ Plaintext important for tools, block Plaintext at firewall or routes
- ❖ Passing in truststore and keystore locations and passwords
- ❖ security.inter.broker.protocol=SSL perhaps not needed if Kafka cluster on single private subnet
 - ❖ SSL makes it go slower, and adds extra CPU load on Kafka Brokers

Setup Kafka Consumer for SSL



```
ConsumerUtil createConsumer()  
3 import ...  
20  
21 public class ConsumerUtil {  
22  
23     public static final String BROKERS = "localhost:10092,localhost:10093,localhost:10094";  
24  
25     private static Consumer<String, StockPrice> createConsumer(  
26         final String bootstrapServers, final String clientId ) {  
27  
28         final Properties props = new Properties();  
29  
30         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,  
31             BROKERS);  
32  
33         props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");  
34         props.put("ssl.truststore.location", "/opt/kafka/conf/certs/kafka.truststore");  
35         props.put("ssl.truststore.password", "kafka123");  
36         props.put("ssl.keystore.location", "/opt/kafka/conf/certs/kafka.keystore");  
37         props.put("ssl.keystore.password", "kafka123");
```

- ❖ Passing in truststore and keystore locations and passwords



Setup Kafka Producer for SSL

```
19 public class StockPriceProducerUtils {  
20  
21     private static Producer<String, StockPrice> createProducer() {  
22         final Properties props = new Properties();  
23         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,  
24                 "localhost:10092,localhost:10093");  
25         props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");  
26         props.put("ssl.truststore.location",  
27                   "/opt/kafka/conf/certs/kafka.truststore");  
28         props.put("ssl.truststore.password", "kafka123");  
29         props.put("ssl.keystore.location",  
30                   "/opt/kafka/conf/certs/kafka.keystore");  
31         props.put("ssl.keystore.password", "kafka123");  
32     }  
33 }
```

- ❖ Passing in truststore and keystore locations and passwords

Complete Lab 8.1

22. SASL

Kafka SASL Authentication - Brokers



- ❖ Kafka uses JAAS for SASL configuration
 - ❖ Java Authentication and Authorization Service (JAAS)
- ❖ Kafka Broker JAAS config
 - ❖ section name KafkaServer for JAAS file
 - ❖ Provides SASL configuration options
 - ❖ SASL client connections are configured
- ❖ Client section (-Dzookeeper.sasl.client=Client is default)
 - ❖ Used to authenticate a SASL connection with **zookeeper (service name, -Dzookeeper.sasl.client.username=zookeeper by default)**
 - ❖ Allows Kafka brokers to set SASL ACL on zookeeper nodes
 - ❖ locks nodes down so only brokers can modify ZooKeeper nodes
 - ❖ Same principal must be used by all brokers in cluster



Kafka SASL Authentication - Clients

```
kafka_consumer_stocks_jaas.conf x

1 KafkaClient {
2     com.sun.security.auth.module.Krb5LoginModule required;
3     useKeyTab=true
4     storeKey=true
5     keyTab="/opt/kafka/conf/security/kafka_client.keytab"
6     principal="kafka-consumer-stocks@cloudurable.com";
7 }
```

- ❖ Clients (Producers and Consumers) configure JAAS using client configuration property **sasl.jaas.config** or using the static JAAS config file
 - ❖ Configure a login module in KafkaClient for the selected mechanism GSSAPI (Kerberos), PLAIN or SCRAM
 - ❖ -
 - Djava.security.auth.login.config=/opt/kafka/conf/kafka_consumer_stocks_jaas.conf

SASL Broker config



- ❖ Kafka Broker Config : SASL configured with transport PLAINTEXT or SSL
 - ❖ listeners=SASL_PLAINTEXT://hostname:port
 - ❖ listener= SASL_SSL://hostname:port
 - ❖ security.inter.broker.protocol=SASL_PLAINTEXT or SASL_SSL
- ❖ If SASL_SSL is used, then SSL has to be configured
- ❖ Kafka SASL Mechanisms
 - ❖ GSSAPI (Kerberos)
 - ❖ PLAIN
 - ❖ SCRAM-SHA-256
 - ❖ SCRAM-SHA-512

Kafka Authentication using SASL/Kerberos



- ❖ If you use Active Directory then no need to setup Kerberos server
- ❖ If not using Active Directory you will need to install it
- ❖ If Oracle Java, download JCE policy files for your Java version to \$JAVA_HOME/jre/lib/security

SASL Kerberos: Create Kerberos Principals for Kafka Broker



- ❖ Ask your Kerberos or Active Directory admin for a principal for each Kafka broker in cluster
- ❖ Ensure all hosts are reachable using hostnames
 - ❖ Kerberos requirement that all hosts are resolvable with FQDNs
 - ❖ If running your own Kerberos server, create these principals

```
$ sudo /usr/sbin/kadmin.local -q 'addprinc -randkey \
    kafka/{hostname}@{REALM}'

$ sudo /usr/sbin/kadmin.local -q "ktadd -k /etc/security/keytabs/{keytabname}.keytab \
    kafka/{hostname}@{REALM}"
```

SASL Kerberos: Configuring Kafka Brokers for Kerberos



```
kafka_broker_jaas.conf x

1 KafkaServer {
2     com.sun.security.auth.module.Krb5LoginModule required
3         useKeyTab=true
4         storeKey=true
5         keyTab="/opt/kafka/conf/security/kafka_broker.keytab"
6         principal="kafka/kafka-broker.hostname.com1@cloudurable.com";
7 }
8
9 // Zookeeper client authentication
10 Client {
11     com.sun.security.auth.module.Krb5LoginModule required
12         useKeyTab=true
13         storeKey=true
14         keyTab="/opt/kafka/conf/security/kafka_broker.keytab"
15         principal="kafka/kafka-broker1.hostname.com@cloudurable.com";
16 }
```

- ❖ Pass to JVM starting up broker
 - ❖ -Djava.security.krb5.conf=/etc/kafka krb5.conf
 - ❖ -Djava.security.auth.login.config=/var/kafka/conf/secutiry/kafka_server_jaas.conf

SASL Kerberos: Configuring Kafka Broker Config for Kerberos



```
server-0.properties x
1 broker.id=0
2
3 listeners=SASL_PLAINTEXT://localhost:9092,SASL_SSL://localhost:10092
4 sasl.mechanism.inter.broker.protocol=GSSAPI
5 sasl.enabled.mechanisms=GSSAPI
6 sasl.kerberos.service.name=kafka
7 security.inter.broker.protocol=SASL_PLAINTEXT
```

- ❖ Configure SASL port and SASL mechanisms in server.properties as described
- ❖ Configure service name (**sasl.kerberos.service.name**),
 - ❖ match principal name of the kafka brokers from JAAS config on last slide
 - ❖ recall principal was "[kafka/kafka-broker1.hostname.com@cloudurable.com](#)"
- ❖ Set sasl.enabled.mechansim to GSSAPI (Kerberos)
- ❖ Set inter broker communication to **SASL_PLAINTEXT** or **SASL_SSL**

SASL Kerberos: Configuring Clients for SASL Kerberos



```
ConsumerUtil createConsumer()
1 package com.cloudurable.kafka.consumer;
2
3 import ...
20
21 public class ConsumerUtil {
22
23     private static Consumer<String, StockPrice> createConsumer(
24         final String bootstrapServers, final String clientId ) {
25
26         final Properties props = new Properties();
27
28         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
29             BROKERS);
30
31         props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
32         props.put("sasl.kerberos.service.name", "kafka");
33         props.put("sasl.mechanism", "GSSAPI");
```

- ❖ Sets the connection protocol to SASL_SSL
 - ❖ encrypt with SSL, authenticate with SASL
- ❖ Sets the service name to Kafka
- ❖ Sets the sasl.mechanism to Kerberos (GSSAPI)

Kafka support multiple SASL Providers



kafka_broker_jaas.conf x

```
1 KafkaServer {
2     org.apache.kafka.common.security.scram.ScramLoginModule required ~
3         username="admin"
4         password="kafka123";
5
6
7     org.apache.kafka.common.security.plain.PlainLoginModule required ~
8         username="admin"
9         password="admin-secret"
10        user_admin="foobar"
11        user_alice="barbaz";
12    };
13    // broker config
14    // sasl.enabled.mechanisms=GSSAPI,PLAIN,SCRAM-SHA-256,SCRAM-SHA-512
15    // security.inter.broker.protocol=SASL_PLAINTEXT (or SASL_SSL)
16    // sasl.mechanism.inter.broker.protocol=GSSAPI (or one of the other enabled mechanisms)
```

- ❖ Kafka supports more than one SASL provider

Modifying SASL mechanism in a Running Cluster



- ❖ SASL mechanism can be modified in a running cluster using the following sequence:
- ❖ Enable new SASL mechanism
 - ❖ add mechanism ***sasl.enabled.mechanisms*** in Broker Config server.properties
- ❖ Update JAAS config file to include both mechanisms as describe
- ❖ Bounce one Kafka Broker at a time
- ❖ Restart clients using new mechanism.
- ❖ Change mechanism of inter-broker communication (if this is required), set ***sasl.mechanism.inter.broker.protocol*** in Broker Config server.properties to new mechanism and bounce Kafka Brokers one at a time
- ❖ Remove old mechanism (if this is required), remove old mechanism from ***sasl.enabled.mechanisms*** in Broker Config server.properties and remove entries for old mechanism from JAAS config file, and once again bounce Kafka Broker one at a time



Adding ACLs to users

```
server-0.properties x
1 broker.id=0
2
3 authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
4 allow.everyone.if.no.acl.found=true
```

```
create-acl.sh x
1 #!/usr/bin/env bash
2 cd ~/kafka-training
3
4 kafka/bin/kafka-acls.sh \
5   --authorizer-properties zookeeper.connect=localhost:2181 \
6   --add --allow-principal User:stocks_consumer \
7   --allow-host 10.0.1.11 --allow-host 198.51.100.1 \
8   --operation Read --topic stock-prices
9
10 kafka/bin/kafka-acls.sh \
11   --authorizer-properties zookeeper.connect=localhost:2181 \
12   --add --allow-principal User:stocks_producer \
13   --allow-host 10.0.1.11 --allow-host 198.51.100.1 \
14   --operation Write --topic stock-prices
```

23. SASL Plain



Kafka SASL Plain

- ❖ SASL/PLAIN
 - ❖ simple username/password authentication mechanism used with TLS for encryption to implement secure authentication
 - ❖ Kafka supports a default implementation for SASL/PLAIN
- ❖ Use SASL/PLAIN with SSL only as transport layer
 - ❖ ensures no clear text passwords are not transmitted
- ❖ Default implementation of SASL/PLAIN in Kafka specifies usernames and passwords in JAAS conf
 - ❖ To avoid storing passwords on disk, use your own implementation of `javax.security.auth.spi.LoginModule`
 - ❖ Or use disk encryption
 - ❖ Use `org.apache.kafka.common.security.plain.PlainLoginModule` as an example.



Kafka SASL Plain

- ❖ In production systems, external authentication servers may implement password authentication
- ❖ Kafka brokers can be integrated to work with these servers by adding your own implementation of ***javax.security.sasl.SaslServer***
 - ❖ Default implementation included in Kafka in the package ***org.apache.kafka.common.security.plain*** can be used as an example
- ❖ New JaaS providers must be installed and registered at the JVM level
 - ❖ CLASSPATH or JAVA_HOME/lib/ext
- ❖ Providers can be registered statically by adding a provider to the security properties file ***JAVA_HOME/lib/security/java.security*** or dynamically
 - ❖ or `Dynamic Security.addProvider(new PlainSaslServerProvider());`



Steps to add Plain SASL via JAAS

- ❖ Create JAAS config for **ZooKeeper** add admin user
- ❖ Modify **ZooKeeper** properties file add SASL config
- ❖ Modify **ZooKeeper** startup script add JAAS config location
- ❖ Create JAAS config for **Kafka Brokers** add users (admin, consumer, producer)
- ❖ Modify **Kafka Brokers** properties file add SASL config
- ❖ Modify **Kafka Broker** startup script add JAAS config location
- ❖ Create JAAS config for **Consumer** add user
- ❖ Modify Consumer **createConsumer()** add SASL config and JAAS config location
- ❖ Create JAAS config for **Producer** add user
- ❖ Modify Producer **createProducer()** add SASL config and JAAS config location



Create JAAS config for ZooKeeper

```
zookeeper_jaas.conf x

1 // Zookeeper server authentication
2 Server {
3     org.apache.kafka.common.security.plain.PlainLoginModule required
4         username="admin"
5         password="kafka-123"
6         user_admin="kafka-123";
7 }
```

- ❖ To log into ZooKeeper, you would need user admin and kafka-123
- ❖ Located **/opt/kafka/config/security/zookeeper_jass.conf**



Modify ZooKeeper properties file

```
zookeeper.properties x
1 dataDir=/tmp/zookeeper-secure2
2 clientPort=2181
3 maxClientCnxns=0
4
5 authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
6 requireClientAuthScheme=sasl
7 jaasLoginRenew=3600000
```

- ❖ Use Auth provider org.apache.zookeeper.server.auth.SASLAuthenticationProvider
- ❖ Require JaaS login via SASL
- ❖ config/zookeeper.properties



Modify ZooKeeper startup script

```
> run-zookeeper.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 export KAFKA_JAAS_FILE="/opt/kafka/conf/security/zookeeper_jaas.conf"
6 export KAFKA_OPTS="-Djava.security.auth.login.config=$KAFKA_JAAS_FILE"
7
8 ## Run ZooKeeper for 1st Cluster
9 kafka/bin/zookeeper-server-start.sh \
10 "$CONFIG/zookeeper.properties"
```

- ❖ bin/run-zookeeper.sh
- ❖ Copy JAAS config files to **/opt/kafka/config/security**

```
cp -R resources/opt/kafka/conf/security /opt/kafka/conf/
```

- ❖ **KAFKA_OPTS** used by kafka startup scripts to pass extra args to JVM

Create JAAS config for Kafka Brokers



```
kafka_broker_jaas.conf x

1 KafkaServer {
2     org.apache.kafka.common.security.plain.PlainLoginModule required
3     username="admin"
4     password="kafka-123"
5     user_admin="kafka-123"
6     user_stocks_consumer="consumer123"
7     user_stocks_producer="producer123";
8 };
9
10 // Zookeeper client authentication
11 Client {
12     org.apache.kafka.common.security.plain.PlainLoginModule required
13     username="admin"
14     password="kafka-123";
15 };
```

- ❖ /opt/kafka/conf/security/kafka_broker_jaas.conf
- ❖ Sets up users for admin for zookeeper, and for inter-broker communication, and sets up users for consumers and producers



Modify Kafka Brokers Properties File

```
server-0.properties x server-1.properties x server-2.properties x
1 broker.id=2
2
3 listeners=PLAINTEXT://localhost:9094,SASL_SSL://localhost:10094
4 sasl.mechanism.inter.broker.protocol=PLAIN
5 sasl.enabled.mechanisms=PLAIN
6 security.inter.broker.protocol=SASL_SSL
7
8 ssl.keystore.location=/opt/kafka/conf/certs/kafka.keystore
9 ssl.keystore.password=kafka123
10 ssl.key.password=kafka123
11 ssl.truststore.location=/opt/kafka/conf/certs/kafka.truststore
12 ssl.truststore.password=kafka123
13 ssl.client.auth=required
```

- ❖ config/server-2.properties, config/server-1.properties, config/server-0.properties
- ❖ Enabled SASL support to use PLAIN SASL
- ❖ Inter-broker communication is using SASL_SSL
- ❖ Producers and Consumers to use 10092, 10093, 10094 SASL_SSL protocol



Modify Kafka Broker Startup Script

```
start-1st-server.sh x start-2nd-server.sh x start-3rd-server.sh x

1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 export KAFKA_JAAS_FILE="/opt/kafka/conf/security/kafka_broker_jaas.conf"
6 export KAFKA_OPTS="-Djava.security.auth.login.config=$KAFKA_JAAS_FILE"
7
8 ## Run Kafka for 3rd Server
9 kafka/bin/kafka-server-start.sh \
10   "$CONFIG/server-2.properties"
11
```

- ❖ bin/start-1st-server.sh, bin/start-2nd-server.sh, bin/start-3rd-server.sh
- ❖ Set location of JaaS using system property ***java.security.auth.login.config***
- ❖ JaaS file located at ***/opt/kafka/conf/security/kafka_broker_jaas.conf***
 - ❖ Must be copied there
- ❖ **KAFKA_OPTS** used by kafka startup scripts to pass extra args to JVM



Create JAAS config for Consumer

```
kafka_consumer_stocks_jaas.conf x

1 KafkaClient {
2     org.apache.kafka.common.security.plain.PlainLoginModule required;
3     username="stocks_consumer"
4     password="consumer123";
5 }
```

- ❖ **/opt/kafka/conf/security/kafka_consumer_stocks_jaas.conf**
- ❖ username is stocks_consumer, and password is consumer123
- ❖ this username/password combo used to log into Brokers

Modify Consumer createConsumer()



```
c ConsumerUtil.java x
ConsumerUtil createConsumer()

21 public class ConsumerUtil {
22     public static final String BROKERS = "localhost:10092,localhost:10093";
23
24     private static Consumer<String, StockPrice> createConsumer(
25         final String bootstrapServers, final String clientId ) {
26
27         System.setProperty("java.security.auth.login.config",
28             "/opt/kafka/conf/security/kafka_consumer_stocks_jaas.conf");
29
30         final Properties props = new Properties();
31         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
32             bootstrapServers);
33         props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
34         props.put("sasl.mechanism", "PLAIN");
35
36         props.put("ssl.keystore.location",
37             "/opt/kafka/conf/certs/kafka.keystore");
38         props.put("ssl.keystore.password", "kafka123");
39         props.put("ssl.truststore.location",
40             "/opt/kafka/conf/certs/kafka.truststore");
41         props.put("ssl.truststore.password", "kafka123");
```

- ❖ Modify Consumer createConsumer() add SASL config and JAAS config location

Create JAAS Config for Producer



kafka_producer_stocks_jaas.conf x

```
1 KafkaClient {  
2     org.apache.kafka.common.security.plain.PlainLoginModule required  
3     username="stocks_producer"  
4     password="producer123";  
5 };
```

- ❖ **/opt/kafka/conf/security/kafka_producer_stocks_jaas.conf**
- ❖ username is **stocks_producer**, and password is **producer123**
- ❖ this username/password combo used to log into Kafka Brokers



Modify Producer createProducer()

```
StockPriceProducerUtils createProducer()
```

```
20
21     private static Producer<String, StockPrice> createProducer() {
22
23         System.setProperty("java.security.auth.login.config",
24             "/opt/kafka/conf/security/kafka_producer_stocks_jaas.conf");
25
26     final Properties props = new Properties();
27
28     props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
29             "localhost:10092,localhost:10093");
30     props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
31     props.put("sasl.mechanism", "PLAIN");
32
33     props.put("ssl.keystore.location",
34             "/opt/kafka/conf/certs/kafka.keystore");
35     props.put("ssl.keystore.password", "kafka123");
36     props.put("ssl.truststore.location",
37             "/opt/kafka/conf/certs/kafka.truststore");
38     props.put("ssl.truststore.password", "kafka123");
```

- ❖ Modify Producer createProducer() add SASL config and JAAS config location

Complete Lab 8.3

24. SASL SCRAM



SASL SCRAM

- ❖ SCRAM is Salted Challenge Response Authentication Mechanism
 - ❖ RFC 5802
- ❖ SASL mechanisms addresses security concerns traditional mechanisms
 - ❖ better than PLAIN and DIGEST-MD5
- ❖ Kafka supports SCRAM-SHA-256 and SCRAM-SHA-512 can be used with SSL/TLS to perform secure authentication
- ❖ Username is used as authenticated Principal for configuration of ACLs etc.
- ❖ Default SCRAM implementation stores SCRAM credentials in Zookeeper



Create SCRAM Users

```
create-scram-users.sh >

1 #!/usr/bin/env bash
2 cd ~/kafka-training
3 SCRAM_CONFIG='SCRAM-SHA-256=[iterations=8192,password=kafka123]'
4 SCRAM_CONFIG="$SCRAM_CONFIG,SCRAM-SHA-512=[password=kafka123]"
5
6 kafka/bin/kafka-configs.sh \
7   --alter --add-config "$SCRAM_CONFIG" \
8   --entity-type users --entity-name stocks_consumer
9   --zookeeper localhost:2181 \
10
11 kafka/bin/kafka-configs.sh \
12   --alter --add-config "$SCRAM_CONFIG" \
13   --entity-type users --entity-name stocks_producer
14   --zookeeper localhost:2181 \
15
16 kafka/bin/kafka-configs.sh \
17   --alter --add-config "$SCRAM_CONFIG" \
18   --entity-type users --entity-name admin
19   --zookeeper localhost:2181 \
```

- ❖ Create users admin, stocks_consumer, stocks_producer store in ZooKeeper

Kafka Broker JAAS Scram Config



```
kafka_broker_jaas.conf x

1 KafkaServer {
2     org.apache.kafka.common.security.scram.ScramLoginModule required
3         username="admin"
4         password="kafka123";
5 };
6
7 // Zookeeper client authentication
8 Client {
9     org.apache.kafka.common.security.plain.PlainLoginModule required
10        username="admin"
11        password="kafka-123";
12 };
```

- ❖ Uses Scram for KafkaServer and Plain for ZooKeeper

Kafka Consumer/Producer JAAS Scram Config



kafka_consumer_stocks_jaas.conf x

```
1 KafkaClient {  
2     org.apache.kafka.common.security.scram.ScramLoginModule required  
3     username="stocks_consumer"  
4     password="kafka123";  
5 };
```

kafka_producer_stocks_jaas.conf x

```
1 KafkaClient {  
2     org.apache.kafka.common.security.scram.ScramLoginModule required  
3     username="stocks_producer"  
4     password="kafka123";  
5 };
```

- ❖ Use Scram as login credentials



Configure SCRAM in Producer

```
c StockPriceProducerUtils.java x
StockPriceProducerUtils startProducer()
19 public class StockPriceProducerUtils {
20
21     private static Producer<String, StockPrice> createProducer() {
22
23         System.setProperty("java.security.auth.login.config",
24             "/opt/kafka/conf/security/kafka_producer_stocks_jaas.conf");
25
26         final Properties props = new Properties();
27
28         props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
29             "localhost:10092,localhost:10093");
30         props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
31         props.put("sasl.mechanism", "SCRAM-SHA-256");
```

- ❖ Configure SCRAM_SHA_256



Configure SCRAM in Consumer

```
c ConsumerUtil.java x
ConsumerUtil createConsumer()
24     private static Consumer<String, StockPrice> createConsumer(
25         final String bootstrapServers, final String clientId ) {
26
27     System.setProperty("java.security.auth.login.config",
28                         "/opt/kafka/conf/security/kafka_consumer_stocks_jaas.conf");
29
30     final Properties props = new Properties();
31     props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
32               bootstrapServers);
33     props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SASL_SSL");
34     props.put("sasl.mechanism", "SCRAM-SHA-256");
```

- ❖ Configure SCRAM_SHA_256



Kafka and SASL SCRAM

- ❖ Kafka stores SCRAM credentials in Zookeeper
 - ❖ Zookeeper should be on private network
- ❖ Kafka supports only SHA-256 and SHA-512 with a minimum iteration count of 4096
 - ❖ Strong hash functions, strong passwords, high iteration counts protect against brute force attacks
- ❖ Use SCRAM only with SSL/TLS-encryption to wire snooping
- ❖ SASL/SCRAM implementation can be overridden using custom login modules to store outside of ZooKeeper

Complete Lab 8.4

25. Mirror Maker



MirrorMaker and Mirroring

- ❖ Mirroring is replication between clusters - called ***mirroring*** to not confuse with ***replication***
 - ❖ replication uses cluster involving brokers, partition leaders, partition followers, ISRs and ZooKeeper
 - ❖ mirroring is just a consumer/producer pair in two clusters
- ❖ ***MirrorMaker*** is used for replicating one clusters data to another cluster



Mirror Maker

- ❖ ***MirrorMaker*** acts like a ***consumer*** to a ***source cluster***
- ❖ ***MirrorMaker*** acts like a ***producer*** to a ***destination cluster***
- ❖ ***Data read from source topics in source cluster and written to same named topics in destination cluster***
- ❖ Source and destination clusters are independent and not coupled
 - ❖ Topics can be configured differently, have different offsets
 - ❖ e.g., different partition count and different replication factors

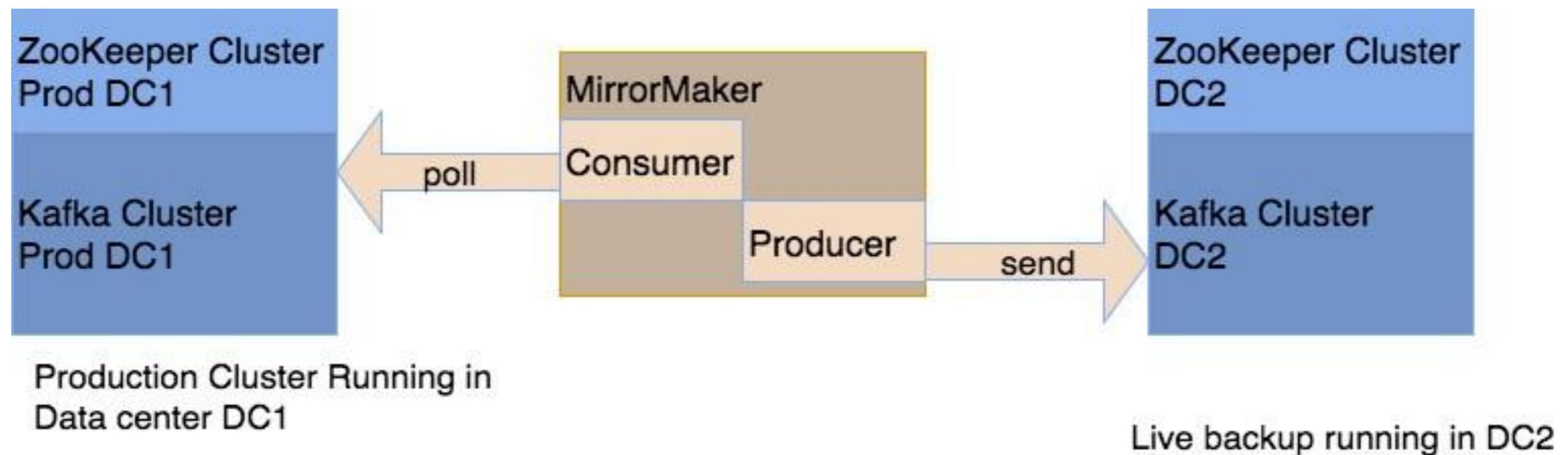


MirrorMaker Use Cases

- ❖ Provide a replica to another datacenter or AWS region
- ❖ ***Mirroring*** used for ***disaster recovery***
 - ❖ datacenter or region goes down
 - ❖ cluster is used for normal fault-tolerance
- ❖ ***Mirroring*** can also be used for ***increased throughput***
 - ❖ scale consumers
 - ❖ scales reads



Disaster Recovery

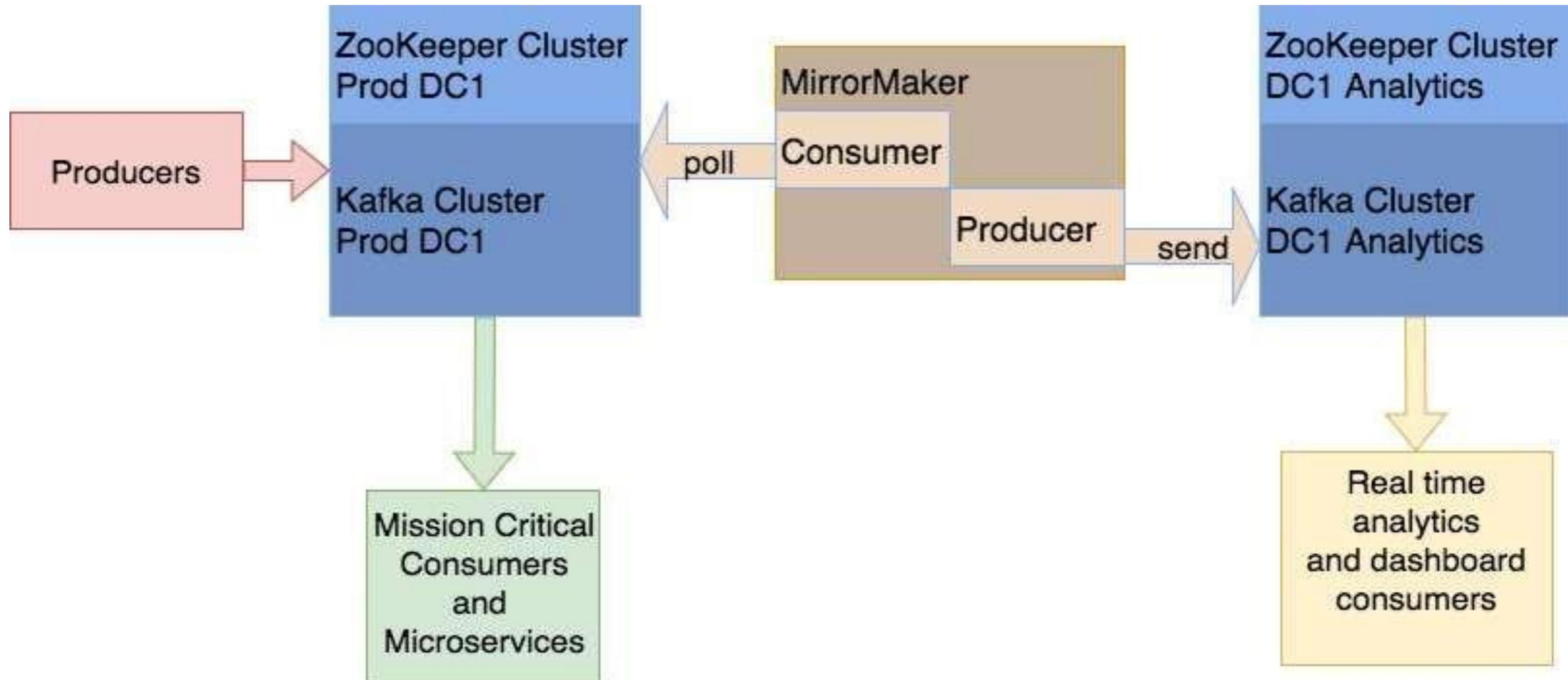


Scale Reads / Scale Consumers

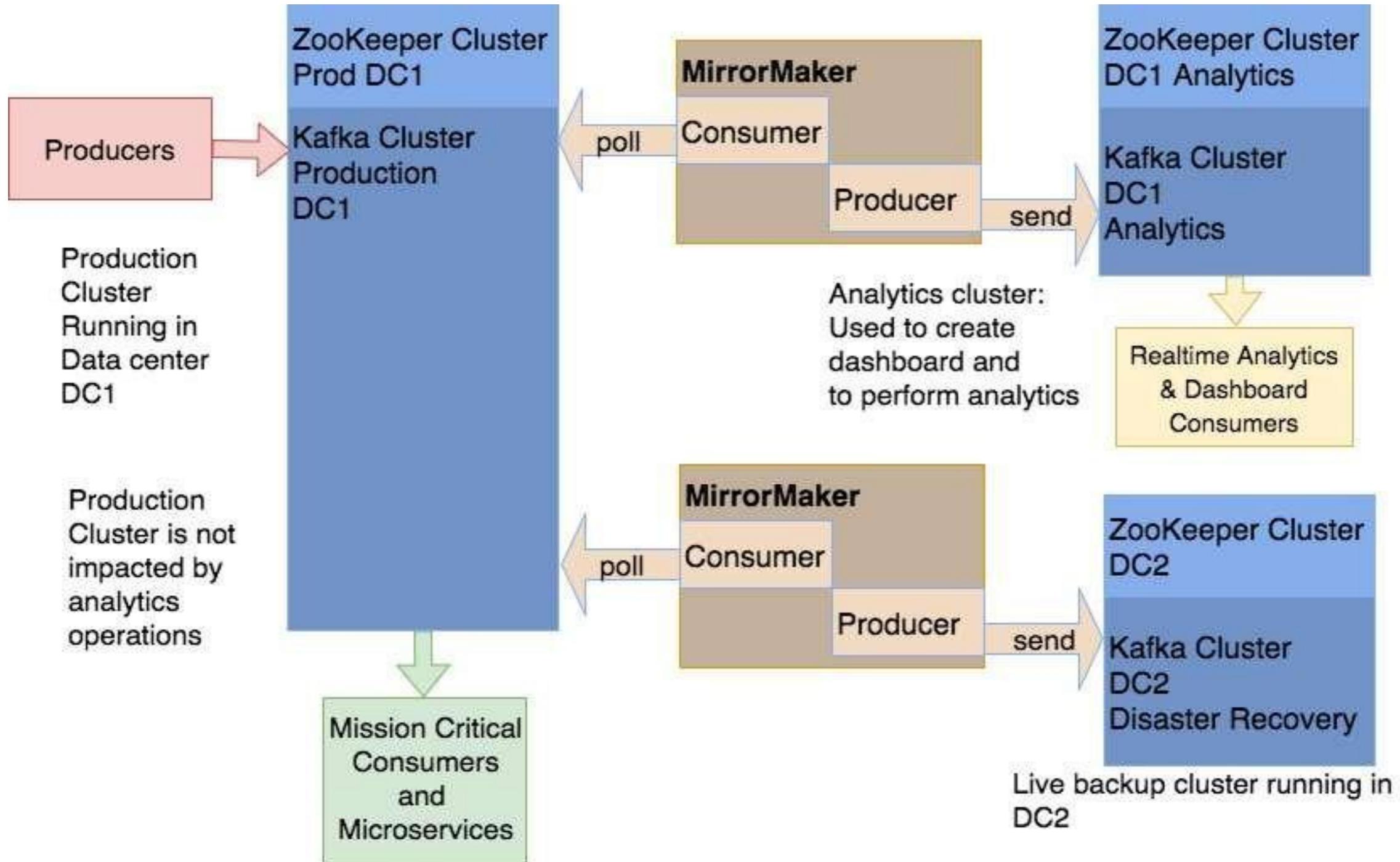


- ❖ You can use MirrorMaker to scale reads
- ❖ You could move non-mission critical consumers to another cluster and replicate to this other cluster
- ❖ Other cluster can replay log or do read intensive log operations and analytics **w/o impacting Production**
- ❖ Production cluster **to serve mission critical services**
- ❖ Analytics cluster could be doing real time dash boards and analytics

Scale Write, Avoid Impacting Mission Critical Services



Many MirrorMakers for different Purposes





Mirror Maker Command Line

- ❖ ***kafka-mirror-maker.sh***
- ❖ **--whitelist** specifies regex for topics to mirror
 - ❖ ‘stock-prices|stocks’ selects two topics
 - ❖ ‘*’ selects all topics
- ❖ **--blacklist** —whitelist regex for topics to exclude
- ❖ Using mirroring with broker config
auto.create.topics.enable=true on destination cluster makes auto replication with no config possible (—whitelist “*”)



Mirror Maker Command Line

```
start-mirror-maker-1st-to-2nd.sh x
```

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run Kafka Mirror Maker
6 kafka/bin/kafka-mirror-maker.sh \
7   --consumer.config "$CONFIG/mm-consumer-1st.properties" \
8   --producer.config "$CONFIG/mm-producer-2nd.properties" \
9   --whitelist "./*"
10
```

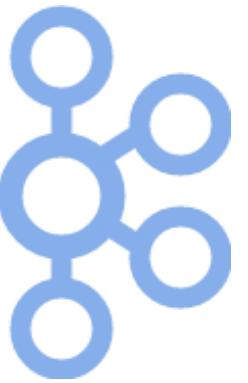
- ❖ Pass consumer properties to read from 1st cluster
- ❖ Pass producer properties to write to 2nd cluster
- ❖ Specify that you want to replicate all topics via whitelist regex



MirrorMaker Review



- ❖ What is the difference between failover and disaster recovery?
- ❖ What are two use cases where you would use MirrorMaker?
- ❖ Why might you want to separate a production microservice messages from a more ad hoc analytics system?
- ❖ If you had to run a nightly job that tallied analytics to all of the calls to a 24/7 production microservice for the last month would you run that in the production cluster?



Lab Running Mirror Maker



Lab Objectives

- ❖ Show running MirrorMaker to mirror a Kafka Cluster to another Kafka Cluster
- ❖ Show how topics can be configured differently per Cluster
- ❖ Demonstrate how to run MirrorMaker
- ❖ Demonstrate how to configure MirrorMakers Consumer
- ❖ Demonstrate how to configure MirrorMakers Producer



Example Details Java

- ❖ Uses a version of StockPriceProducer example
- ❖ Modifies Consumer to consume from one of three Clusters
- ❖ Three clusters in example
- ❖ Producer sends to 1st Cluster
- ❖ Consumer to consume from 1st Cluster
- ❖ Consumer to consume from 2nd Cluster
- ❖ Consumer to consume from 3rd Cluster



Example Details Scripts

- ❖ Script to startup first, second and 3rd cluster
 - ❖ start up Broker and ZooKeeper
- ❖ Script to mirror records between 1st and 2nd cluster
- ❖ Script to mirror records between 2nd and 3rd cluster
- ❖ Scripts to describe topic for each cluster



Scripts

Scripts

Script Name	Description
start-1st-cluster.sh	Starts up ZooKeeper and 1 Kafka Broker for the first cluster.
start-2nd-cluster.sh	Starts up ZooKeeper and 1 Kafka Broker for the second cluster.
start-3rd-cluster.sh	Starts up ZooKeeper and 1 Kafka Broker for the third cluster.
start-mirror-maker-1st-to-2nd.sh	Starts up MirrorMaker to mirror records from the 1st cluster to the second cluster.
start-mirror-maker-2nd-to-3rd.sh	Starts up MirrorMaker to mirror records from the 2nd cluster to the 3rd cluster.
create-topic.sh	Creates stock-prices topic.
describe-topic-1st.sh	Describes stock-prices on first cluster.
describe-topic-2nd.sh	Describes stock-prices on second cluster.
describe-topic-3rd.sh	Describes stock-prices on third cluster.

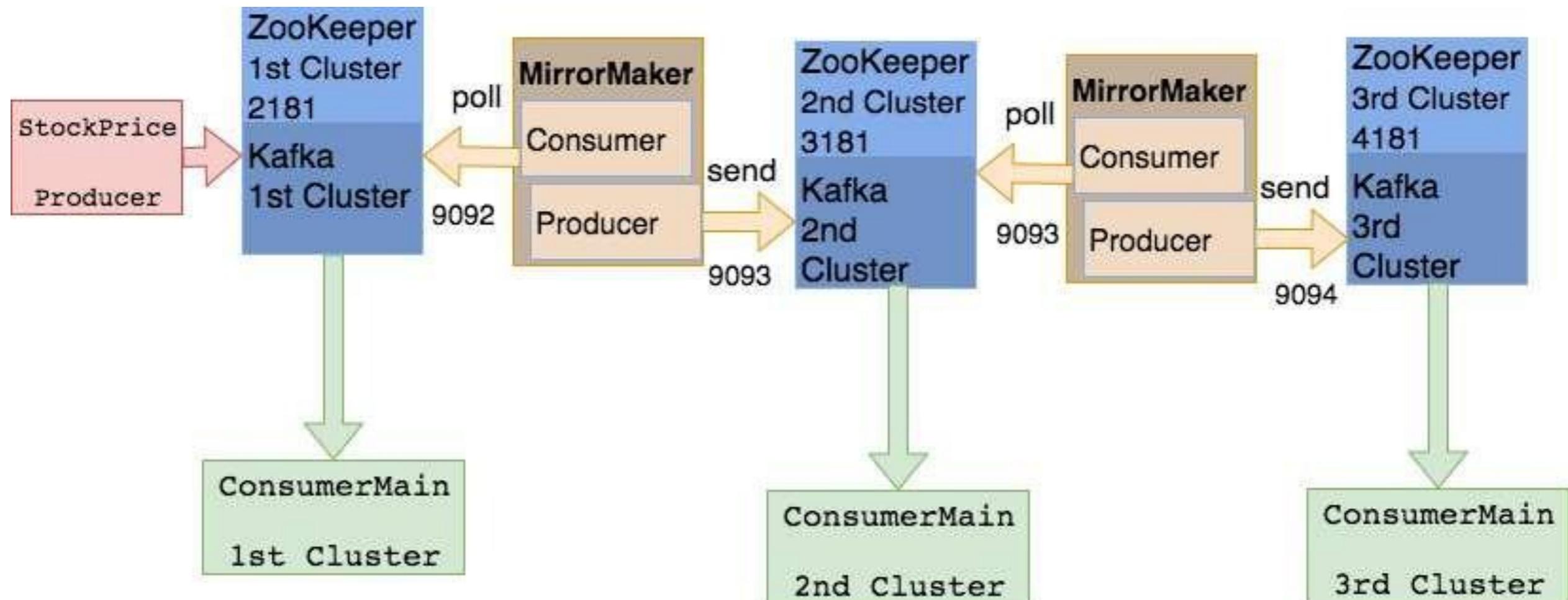


Cluster Ports

Cluster Config

Cluster	Kafka Client Port	ZooKeeper Port
1st Cluster	9092	2181
2nd Cluster	9093	3181
3rd Cluster	9094	4181

Example MirrorMaker Cluster Producer Consumer





Start Cluster Scripts

start-1st-cluster.sh ×

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run ZooKeeper for 1st Cluster
6 kafka/bin/zookeeper-server-start.sh \
7 "$CONFIG/zookeeper-0.properties" &
8
9
10 ## Run Kafka for 1st Cluster
11 kafka/bin/kafka-server-start.sh \
12 "$CONFIG/server-0.properties"
```

start-2nd-cluster.sh ×

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run ZooKeeper for 2nd Cluster
6 kafka/bin/zookeeper-server-start.sh \
7 "$CONFIG/zookeeper-1.properties" &
8
9
10 ## Run Kafka for 2nd Cluster
11 kafka/bin/kafka-server-start.sh \
12 "$CONFIG/server-1.properties"
```

start-3rd-cluster.sh ×

```
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4
5 ## Run ZooKeeper for 3rd Cluster
6 kafka/bin/zookeeper-server-start.sh \
7 "$CONFIG/zookeeper-2.properties" &
8
9
10 ## Run Kafka for 3rd Cluster
11 kafka/bin/kafka-server-start.sh \
12 "$CONFIG/server-2.properties"
```

Server and MirrorMaker Config

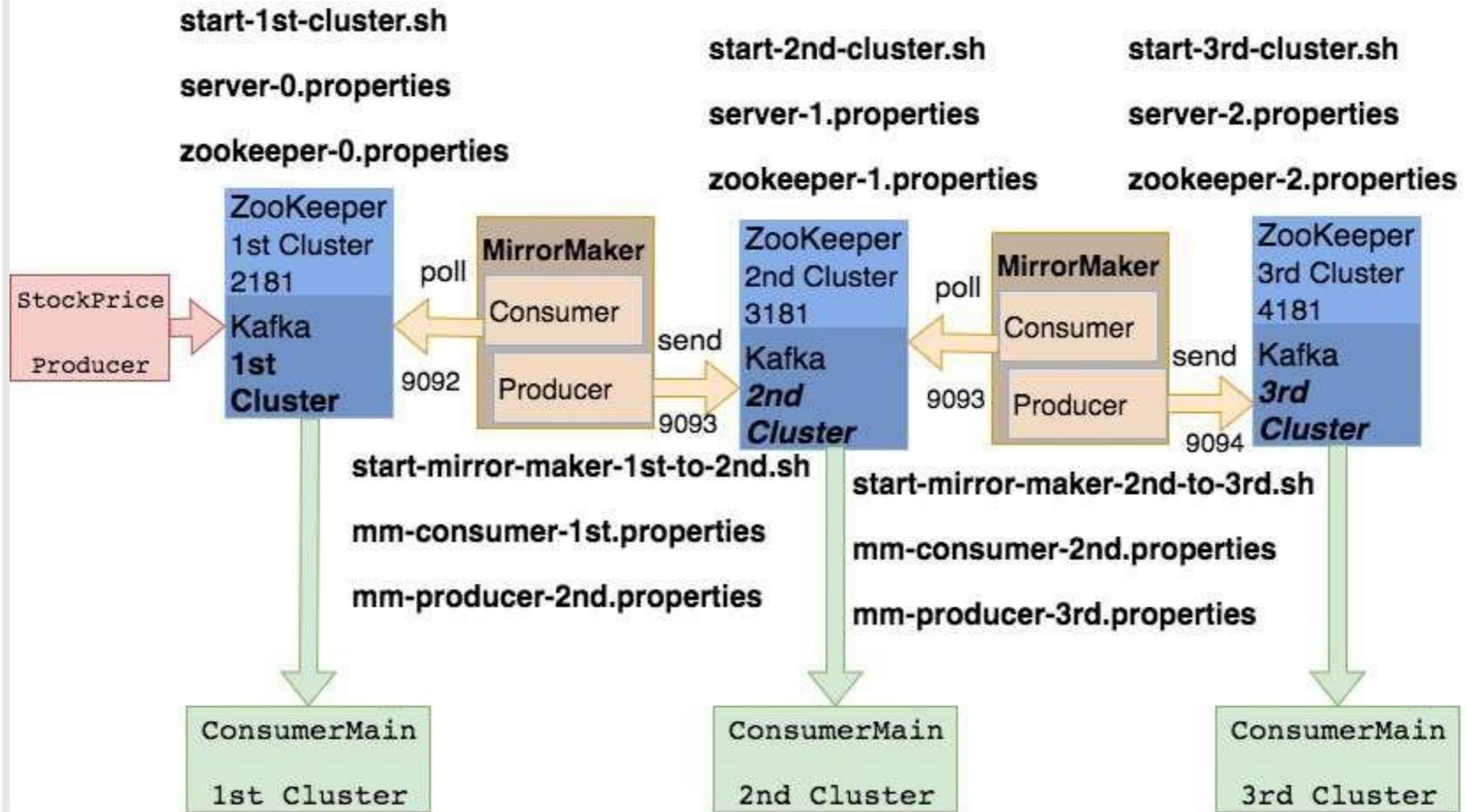


Config

Config File	Description	Config Type
server-0.properties	Kafka Server 0 for 1st Cluster	Kafka Broker
server-0.properties	Kafka Server 1 for 2nd Cluster	Kafka Broker
server-0.properties	Kafka Server 2 for 3rd Cluster	Kafka Broker
zookeeper-0.properties	ZooKeeper Server Config for 1st Cluster	ZooKeeper
zookeeper-1.properties	ZooKeeper Server Config for 2nd Cluster	ZooKeeper
zookeeper-2.properties	ZooKeeper Server Config for 3rd Cluster	ZooKeeper
mm-consumer-1st.properties	MirrorMaker Consumer Config for 1st Cluster	Kafka Consumer
mm-consumer-2nd.properties	MirrorMaker Consumer Config for 2nd Cluster	Kafka Consumer
mm-producer-2nd.properties	MirrorMaker Producer Config for 2nd Cluster	Kafka Producer
mm-producer-3rd.properties	MirrorMaker Producer Config for 3rd Cluster	Kafka Producer



Diagram to Script/Config





Create Topic

```
>_ create-topic.sh <

1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 kafka/bin/kafka-topics.sh \
6   --create \
7   --zookeeper localhost:2181 \
8   --replication-factor 1 \
9   --partitions 3 \
10  --topic stock-prices \
11  --config min.insync.replicas=1
```

- ❖ Create a topic but only in the ***first*** cluster (**2181**)
- ❖ Only one broker so only one replication factor too



1st Cluster Config

```
server-0.properties x  
1 broker.id=0  
2 port=9092  
3 log.dirs=./logs/kafka-0  
4 ## Require three replicas to respond  
5 ## before acknowledging send from producer.  
6 min.insync.replicas=1  
7 auto.create.topics.enable=false  
8 zookeeper.connect=localhost:2181  
9  
10
```

- ❖ Broker ID 0
- ❖ Kafka Broker Port **9092**
- ❖ Connects to ZooKeeper at port **2181**
- ❖ **No Topic Auto Create**



2nd Cluster Config

```
server-1.properties x  
1 broker.id=1  
2 port=9093  
3 log.dirs=./logs/kafka-1  
4 min.insync.replicas=1  
5 auto.create.topics.enable=true  
6 zookeeper.connect=localhost:3181  
7 num.partitions=13
```

- ❖ Broker ID 1
- ❖ Kafka Broker Port **9093**
- ❖ Connects to ZooKeeper at port **3181**
- ❖ **Topic Auto Create Enabled!**
- ❖ Default Num Partitions **13**



3rd Cluster Config

```
server-2.properties x  
1 broker.id=2  
2 port=9094  
3 log.dirs=./logs/kafka-2  
4 min.insync.replicas=1  
5 auto.create.topics.enable=true  
6 zookeeper.connect=localhost:4181  
7 num.partitions=33
```

- ❖ Broker ID 2
- ❖ Kafka Broker Port **9094**
- ❖ Connects to ZooKeeper at port **4181**
- ❖ **Topic Auto Create Enabled!**
- ❖ Default Num Partitions **33**

ZooKeeper Config



```
zookeeper-0.properties x
1 dataDir=/tmp/zookeeper1
2 clientPort=2181
3 maxClientCnxns=0
4

zookeeper-1.properties x
1 dataDir=/tmp/zookeeper2
2 clientPort=3181
3 maxClientCnxns=0
4

zookeeper-2.properties x
1 dataDir=/tmp/zookeeper3
2 clientPort=4181
3 maxClientCnxns=0
4
```

- ❖ Each ZooKeeper instance runs on its own port and is independent of the others

Mirror Maker Consumer Config



```
mm-consumer-2nd.properties x
1 bootstrap.servers=localhost:9093
2 client.id=mm2.Consumer
3 key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
4 value.deserializer=org.apache.kafka.common.serialization.ByteArrayDeserializer
5 group.id=mm2.Consumer
6 partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor

mm-consumer-1st.properties x
1 bootstrap.servers=localhost:9092
2 client.id=mm1.Consumer
3 key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
4 value.deserializer=org.apache.kafka.common.serialization.ByteArrayDeserializer
5 group.id=mm1.Consumer
6 partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor
```

- ❖ Two Consumer for 2 different MirrorMakers
- ❖ One consumes 2nd Cluster (9093)
- ❖ One consumes 1st Cluster (9092)
- ❖ Notice we use ByteArrayDeserializer because we want MirrorMaker treating payload as opaque



Mirror Maker Producer Config

```
mm-consumer-2nd.properties x mm-producer-3rd.properties x
1 bootstrap.servers=localhost:9094
2 client.id=mml1.Producer
3 key.serializer=org.apache.kafka.common.serialization.StringSerializer
4 value.serializer=org.apache.kafka.common.serialization.BytesSerializer
5 compression.type=lz4
6 linger.ms=100
7 batch.size=65536
8

mm-consumer-1st.properties x mm-producer-2nd.properties x
1 bootstrap.servers=localhost:9093
2 client.id=mml1.Producer
3 key.serializer=org.apache.kafka.common.serialization.StringSerializer
4 value.serializer=org.apache.kafka.common.serialization.BytesSerializer
5 compression.type=lz4
6 linger.ms=100
7 batch.size=65536
```

- ❖ Two Consumer for 2 different MirrorMakers
- ❖ One produces to 3rd Cluster
- ❖ One produces to 2nd Cluster
- ❖ Notice we use BytesSerializer because we want MirrorMaker treating payload as opaque



Mirror Maker Start Scripts

```
mm-consumer-2nd.properties x mm-producer-3rd.properties x start-mirror-maker-2nd-to-3rd.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka Mirror Maker: Mirror 2nd Cluster to 3rd Cluster
5 kafka/bin/kafka-mirror-maker.sh \
6   --consumer.config "$CONFIG/mm-consumer-2nd.properties" \
7   --producer.config "$CONFIG/mm-producer-3rd.properties" \
8   --whitelist ".*"

mm-consumer-1st.properties x mm-producer-2nd.properties x start-mirror-maker-1st-to-2nd.sh x
1 #!/usr/bin/env bash
2 CONFIG=`pwd`/config
3 cd ~/kafka-training
4 ## Run Kafka Mirror Maker: Mirror 1st Cluster to 2nd Cluster
5 kafka/bin/kafka-mirror-maker.sh \
6   --consumer.config "$CONFIG/mm-consumer-1st.properties" \
7   --producer.config "$CONFIG/mm-producer-2nd.properties" \
8   --whitelist ".*"
```

- ❖ Mirror 2nd Cluster to 3rd using Producer and Consumer config
- ❖ Mirror 1st Cluster to 2nd using Producer and Consumer config



Run Scripts

- ❖ Start up ***first*** cluster: ***bin/start-1st-cluster.sh***
- ❖ In a new terminal, Start up ***2nd*** cluster: ***bin/start-2nd-cluster.sh***
- ❖ In a new terminal, Start up ***3rd*** cluster: ***bin/start-3rd-cluster.sh***
- ❖ Create Topic: ***bin/create-topic.sh***
- ❖ Wait 30 seconds
- ❖ Startup Mirror Maker for 1st to 2nd mirroring:
 - ❖ ***start-mirror-maker-1st-to-2nd.sh***
- ❖ Startup Mirror Maker for 2nd to 3rd mirroring
 - ❖ ***start-mirror-maker-2nd-to-3rd.sh***

```
Terminal
+ Cluster1 Cluster2 Cluster3 mirrorMaker1to2 mirrorMaker2to3
X ~/kafka-training/lab9/solution
$ bin/start-mirror-maker-2nd-to-3rd.sh
```

Run Java Consumer(s) and Producer



- ❖ Run ***ConsumerMain1stCluster*** in IDE
- ❖ Run ***ConsumerMain2ndCluster*** in IDE
- ❖ Run ***ConsumerMain3rdCluster*** in IDE
- ❖ Run ***StockPriceProducer*** in IDE
- ❖ After 30 Seconds stop StockPriceProducer
- ❖ Wait 30 seconds and ensure consumers have same stock prices

Output, Stocks should have same value



```
Run: ConsumerMain1stCluster
New ConsumerRecords par
ticker BBB price 80.14
ticker FFF price 80.14

New ConsumerRecords par
ticker AAA price 80.14
ticker EEE price 80.14
ticker IBM price 88.34

Run: ConsumerMain1stCluster ConsumerMain2ndCluster
New ConsumerRecords par count 0 count 0, max offset 0
ticker IBM price 88.34 saved true Thread 9

New ConsumerRecords par count 0 count 0, max offset 0
ticker AAA price 80.14 saved true Thread 7
ticker GOOG price 482.11 saved true Thread 5

Run: ConsumerMain1stCluster ConsumerMain2ndCluster ConsumerMain3rdCluster
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
New ConsumerRecords par count 0 count 0, max offset 0
ticker AAA price 80.14 saved true Thread 13
```



ConsumerMain1stCluster

```
c ConsumerMain1stCluster.java x
1 package com.cloudurable.kafka.consumer;
2
3 import static com.cloudurable.kafka.consumer.ConsumerUtil.FIRST_CLUSTER;
4 import static com.cloudurable.kafka.consumer.ConsumerUtil.startConsumers;
5
6 public class ConsumerMain1stCluster {
7     public static void main(String... args) throws Exception {
8         startConsumers(FIRST_CLUSTER);
9     }
10 }
```

```
c ConsumerUtil.java x
ConsumerUtil
17 import java.util.stream.IntStream;
18
19 import static com.cloudurable.kafka.StockAppConstants.TOPIC;
20 import static java.util.concurrent.Executors.newFixedThreadPool;
21
22 public class ConsumerUtil {
23
24     private static final Logger logger =
25             LoggerFactory.getLogger(ConsumerUtil.class);
26
27     public static final String FIRST_CLUSTER = "localhost:9092";
28     public static final String SECOND_CLUSTER = "localhost:9093";
29     public static final String THIRD_CLUSTER = "localhost:9094";
30 }
```

ConsumerUtils startConsumers



```
C ConsumerUtil.java x
ConsumerUtil startConsumers() -> Runnable

62
63     public static void startConsumers(final String cluster) {
64         final Consumer<String, StockPrice> consumer = createConsumer(cluster);
65         final int threadCount = getPartitionCount(consumer);
66         final ExecutorService executorService = newFixedThreadPool(threadCount);
67
68         final List<StockPriceConsumerRunnable> workers = new ArrayList<>(threadCount);
69
70         IntStream.range(0, threadCount).forEach(index -> {
71             final StockPriceConsumerRunnable stockPriceConsumer =
72                 new StockPriceConsumerRunnable(createConsumer(cluster),
73                     readCountStatusUpdate: 5, index);
74             workers.add(stockPriceConsumer);
75             executorService.submit(stockPriceConsumer);
76         });
    }
```

- ❖ Start Consumers takes cluster which is broker list
- ❖ Same code as earlier examples runs Consumer per Thread
- ❖ thread count determined by partition count for topic



Describe Topic per Cluster

```
describe-topic-1st.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic stock-prices \
8   --zookeeper localhost:2181
```

```
describe-topic-2nd.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic stock-prices \
8   --zookeeper localhost:3181
```

```
describe-topic-3rd.sh x
1 #!/usr/bin/env bash
2
3 cd ~/kafka-training
4
5 # List existing topics
6 kafka/bin/kafka-topics.sh --describe \
7   --topic stock-prices \
8   --zookeeper localhost:4181
```

- ❖ Script per cluster to describe **stock-prices**

stock-prices Topic on 1st Cluster



Terminal

	Cluster1	Cluster2	Cluster3	mirrorMaker1to2	mirrorMaker2to3	Util	describe1	describe2	describe3
+									
✖									

```
~/kafka-training/lab9/solution
$ bin/describe-topic-1st.sh
Topic:stock-prices      PartitionCount:3      ReplicationFactor:1      Configs:min.insync.replicas=1
  Topic: stock-prices    Partition: 0        Leader: 0          Replicas: 0        Isr: 0
  Topic: stock-prices    Partition: 1        Leader: 0          Replicas: 0        Isr: 0
  Topic: stock-prices    Partition: 2        Leader: 0          Replicas: 0        Isr: 0
~/kafka-training/lab9/solution
```

- ❖ ***bin/describe-topic-1st.sh***
- ❖ Why does this only have three partitions?
- ❖ Where did we configure three partitions for **stock-prices**?



stock-prices Topic on 2nd Cluster

Terminal

+	Cluster1	Cluster2	Cluster3	mirrorMaker1to2	mirrorMaker2to3	Util	describe1	describe2	describe3
✖	~/kafka-training/lab9/solution								
	\$ bin/describe-topic-2nd.sh								
	Topic:stock-prices PartitionCount:13 ReplicationFactor:1 Configs:								
	Topic: stock-prices	Partition: 0	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 1	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 2	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 3	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 4	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 5	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 6	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 7	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 8	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 9	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 10	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 11	Leader: 1	Replicas: 1	Isr: 1				
	Topic: stock-prices	Partition: 12	Leader: 1	Replicas: 1	Isr: 1				

- ❖ ***bin/describe-topic-2nd.sh***
- ❖ Why does this only have 13 partitions?
- ❖ Where did we configure 13 partitions for ***stock-prices***?

stock-prices Topic on 3rd Cluster



Terminal

+

Cluster1

Cluster2

Cluster3

mirrorMaker1to2

mirrorMaker2to3

Util

describe1

describe2

describe3

×

~/kafka-training/lab9/solution

\$ bin/describe-topic-3rd.sh

```
Topic:stock-prices PartitionCount:33 ReplicationFactor:1 Configs:  
Topic: stock-prices Partition: 0 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 1 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 2 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 3 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 4 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 5 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 6 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 7 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 8 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 9 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 10 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 11 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 12 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 13 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 14 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 15 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 16 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 17 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 18 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 19 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 20 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 21 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 22 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 23 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 24 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 25 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 26 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 27 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 28 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 29 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 30 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 31 Leader: 2 Replicas: 2 Isr: 2  
Topic: stock-prices Partition: 32 Leader: 2 Replicas: 2 Isr: 2
```

- ❖ ***bin/describe-topic-3rd.sh***
- ❖ Why does this only have 33 partitions?
- ❖ Where did we configure 33 partitions for ***stock-prices***?

Lab Conclusion



- ❖ Used MirrorMaker to mirror a Kafka Cluster to another Kafka Cluster
 - ❖ Mirrored 1st Cluster records to 2nd Cluster
 - ❖ Mirrored 2nd Cluster records to 3rd Cluster
- ❖ Ran 1 producer that produced to 1st Cluster
- ❖ Ran 1 consumer group that read from 1st Cluster
- ❖ Ran 1 consumer group that read from 2nd Cluster
- ❖ Ran 1 consumer group that read from 3rd Cluster
- ❖ Configured topics differently per cluster (3, 13 and 33 partitions)
- ❖ Used describe topics to show topics were configured differently on different clusters

Complete Lab 9