CLOUDURABLE ™

Cassandra and Kafka Support on AWS/EC2

Cloudurable

Support around Cassandra and Kafka running in EC2

CLOUDURABLE

CLOUDURABLE ™

Cassandra / Kafka Support in EC2/AWS

Kafka Introduction

Kafka messaging

What is Kafka?

- ❖ Distributed Streaming Platform

  - ❖ Publish and Subscribe to streams of records

  - ❖ Fault tolerant storage

  - ❖ Process records as they occur

Kafka Usage

❖ Build real-time streaming data pipe-lines

    ❖ Enable in-memory microservices (actors, Akka, Vert.x, Qbit)

❖ Build real-time streaming applications that react to streams

    ❖ Real-time data analytics

    ❖ Transform, react, aggregate, join real-time data flows

Kafka Use Cases

❖ Metrics / KPIs gathering

   ❖ Aggregate statistics from many sources

❖ Even Sourcing

   ❖ Used with microservices (in-memory) and actor systems

❖ Commit Log

   ❖ External commit log for distributed systems. Replicated data between nodes, re-sync for nodes to restore state

❖ Real-time data analytics, Stream Processing, Log Aggregation, Messaging, Click-stream tracking, Audit trail, etc.

™

Who uses Kafka?

❖ *LinkedIn*: Activity data and operational metrics

❖ *Twitter*: Uses it as part of Storm – stream processing infrastructure

❖ *Square*: Kafka as bus to move all system events to various Square data centers (logs, custom events, metrics, an so on). Outputs to Splunk, Graphite, Esper-like alerting systems

❖ Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, DataDog, LucidWorks, MailChimp, NetFlix, etc.

Kafka Fundamentals

❖ *Records* have a *key*, *value* and *timestamp*

❖ *Topic* a stream of records

  ❖ *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*: Cluster of Kafka servers running in cluster form broker. Consists on many processes on many servers

❖ *ZooKeeper*: Does coordination of broker and consumers. Consistent file system for configuration information and leadership election
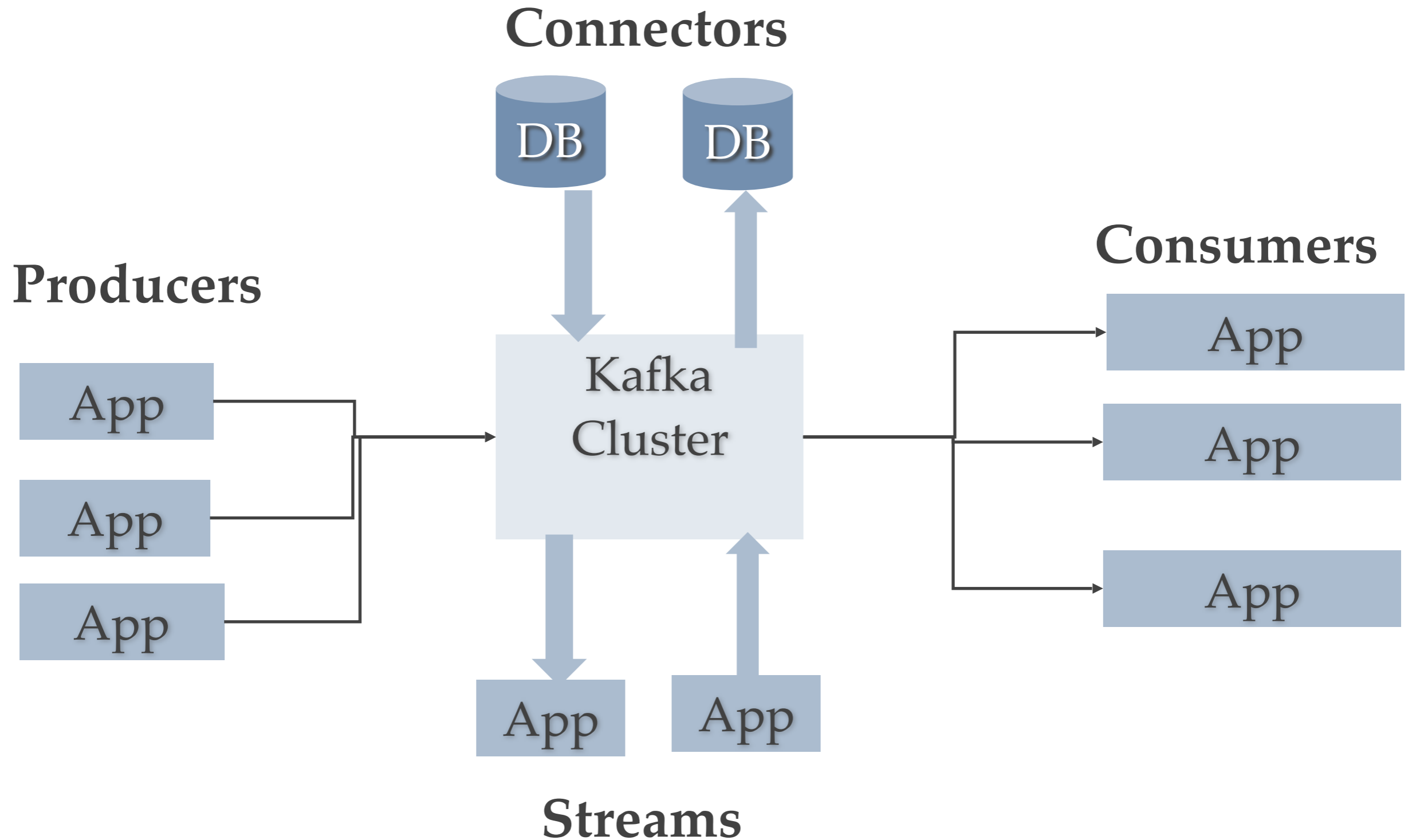
Kafka: Topics, Producers, and Consumers
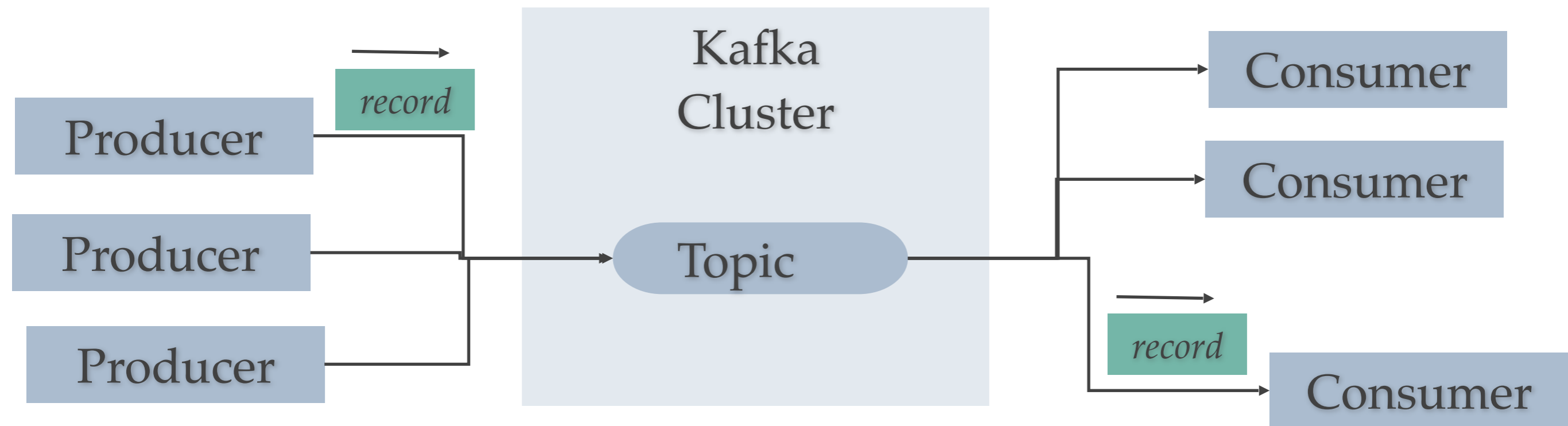
ZooKeeper does coordination for Kafka Consumer and Kafka Cluster
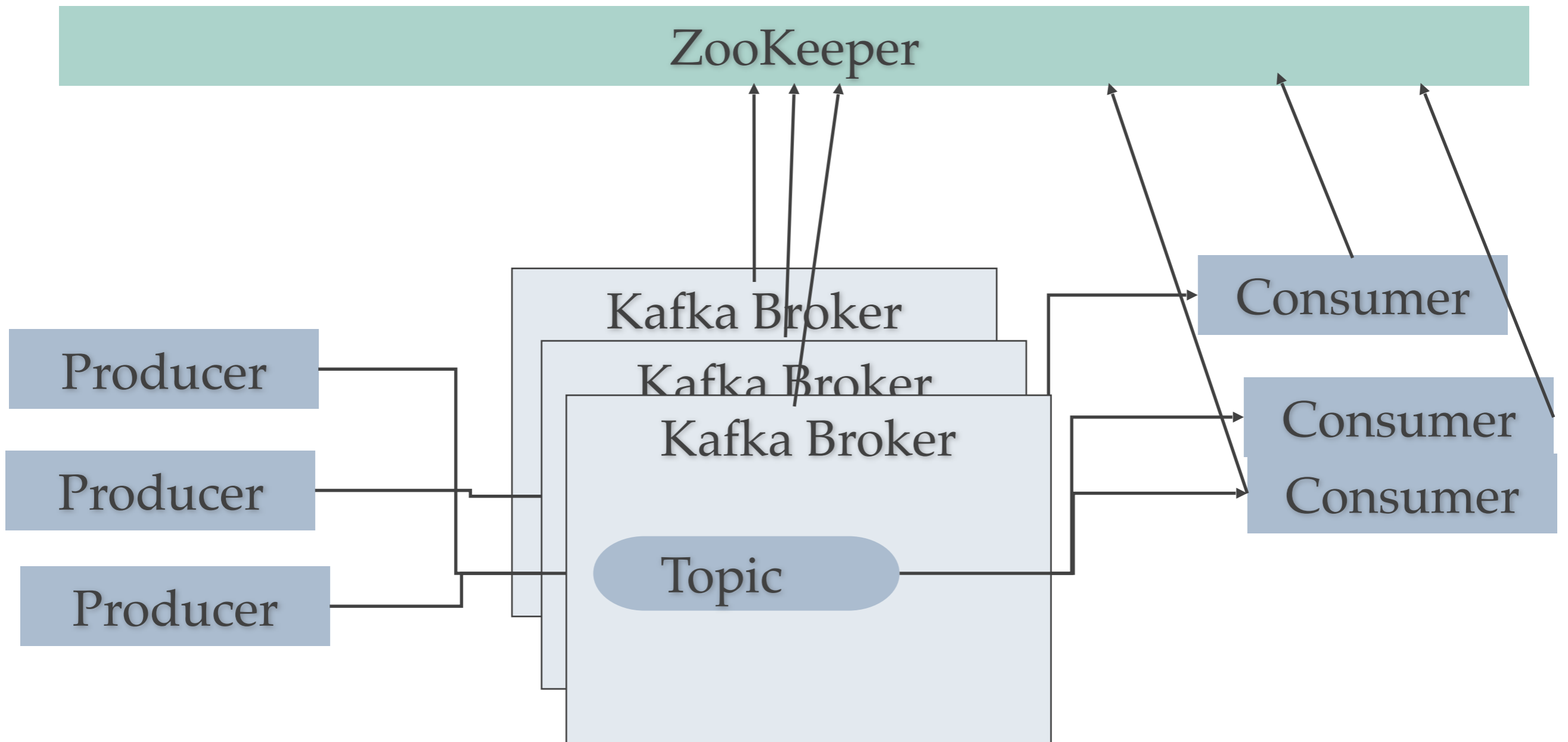
Kafka Extensions

❖ ***Streams*** API to transform, aggregate, process records from a stream and produce derivative streams

❖ ***Connector*** API reusable producers and consumers (e.g., stream of changes from DynamoDB)

Kafka Connectors and Streams

Kafka Polyglot clients / Wire protocol

❖ Kafka communication from clients and servers wire protocol over TCP protocol

❖ Protocol versioned

❖ Maintains backwards compatibility

❖ Many languages supported

Topics and Logs

- ❖ Topic is a stream of records

- ❖ Topics stored in log

- ❖ Log broken up into partitions and segments

- ❖ Topics is a category or stream name

- ❖ Topics are pub/sub

  - ❖ Can have zero or many consumers (subscribers)

- ❖ Topics are broken up into partitions for speed and size

Topic Partitions

❖ *Topics* are broken up into *partitions*

❖ *Partitions* are decided usually by key of record

    ❖ Key of record determines which partition

❖ *Partitions* are used to scale Kafka across many servers

    ❖ Record sent to correct partition by key

❖ *Partitions* are used to facilitate parallel consumers

    ❖ Records are consumed in parallel up to the number of partitions

Partition Log

- *Partition* is ordered, immutable sequence of records that is continually appended to—a structured commit *log*

- Records in partitions are assigned *sequential id* number called the *offset*

- Offset identifies each record within the partition

- *Topic Partitions* allow Kafka log to scale beyond a size that will fit on a single server

  - Topic partition must fit on servers that host it, but topic can span many partitions hosted by many servers

- Topic Partitions are unit of *parallelism* - each consumer in a consumer group can work on one partition at a time

Kafka Topic Partitions Layout
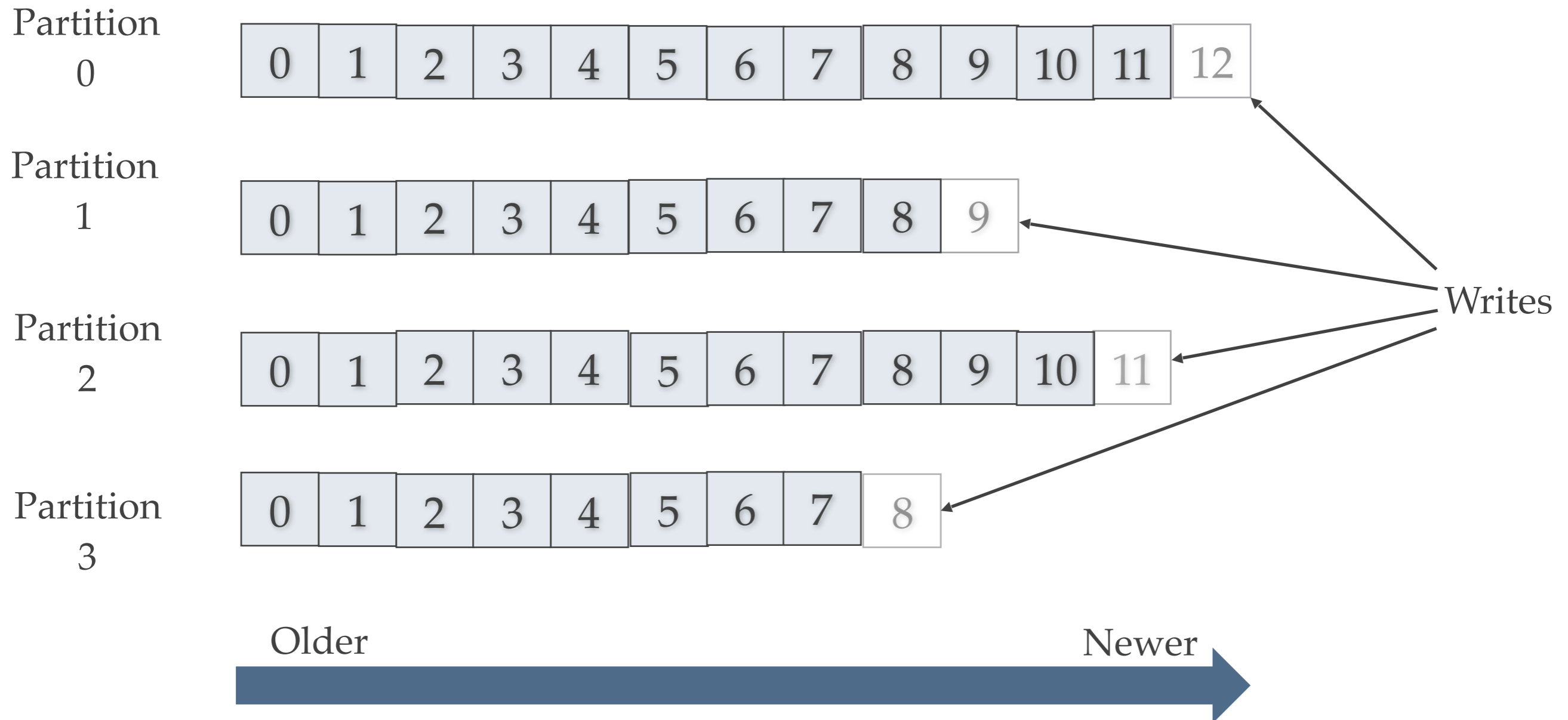
Kafka Record retention

❖ Kafka cluster retains all published records

  ❖ Time based – configurable retention period

  ❖ Size based

  ❖ Compaction

❖ Retention policy of three days or two weeks or a month

❖ It is available for consumption until discarded by time, size or compaction

❖ Consumption speed not impacted by size

Kafka Consumers / Producers

Producers

Partition
0

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Consumer Group B
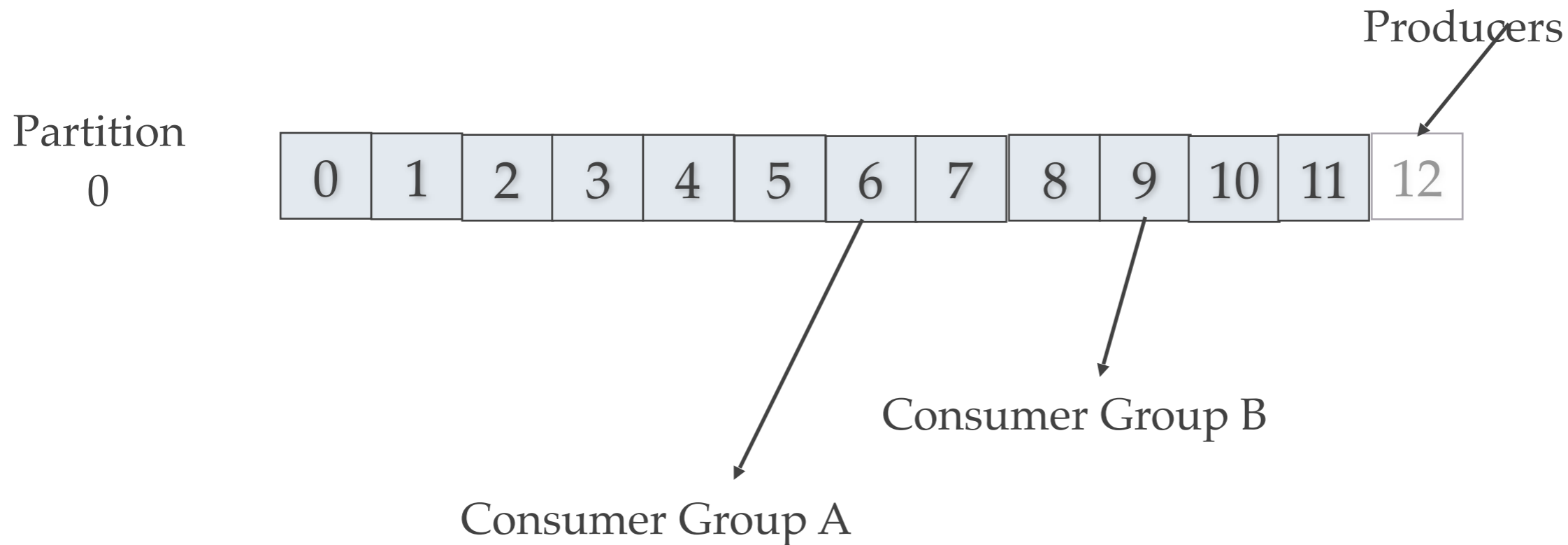
Consumer Group A

Consumers remember offset where they left off.

Consumers groups each have their own offset.

Kafka Partition Distribution

❖ Each partition has **leader server** and zero or more **follower servers**

    ❖ **Leader** handles all read and write requests for partition

    ❖ **Followers** replicate leader, and take over if leader dies

    ❖ Used for parallel consumer handling within a group

❖ Partitions of log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of partitions

❖ Each partition can be replicated across a configurable number of Kafka servers

    ❖ Used for fault tolerance

Kafka Producers

❖ ***Producers*** send records to topics

❖ ***Producer*** picks which partition to send record to per topic

    ❖ Can be done in a ***round-robin***

    ❖ Can be based on priority

    ❖ Typically based on ***key*** of ***record***

❖ Important: *Producer picks partition*

Kafka Consumers

❖ Consumers are grouped into a ***Consumer Group***

  ❖ ***Consumer group*** has a unique name

  ❖ Each ***consumer group*** is a subscriber

  ❖ Each ***consumer group*** maintains its own offset

  ❖ Multiple subscribers = multiple consumer groups

❖ ***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***

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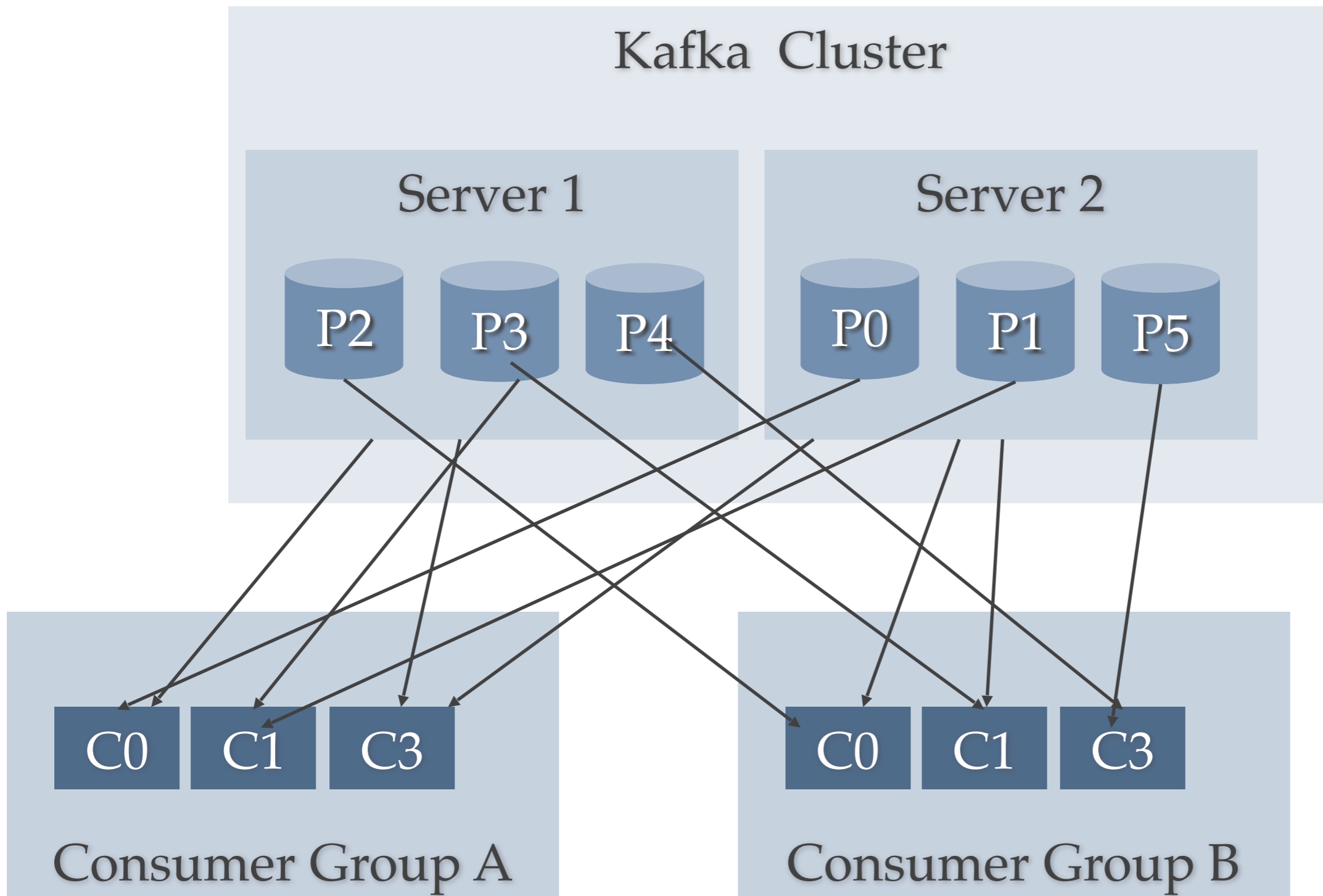
Kafka Consumer Consumption

❖ Kafka **Consumer** consumption divides partitions over consumer instances

  ❖ Each Consumer is exclusive consumer of a "fair share" of partitions

  ❖ Consumer membership in group is handled by the Kafka protocol dynamically

  ❖ If new Consumers join Consumer group they get share of partitions

  ❖ If Consumer dies, its partitions are split among remaining live Consumers in group

❖ Order is only guaranteed within a single partition

❖ Since **records** are typically stored **by key into a partition** then order per partition is sufficient for most use cases

Kafka vs JMS Messaging

❖ It is a bit like both Queues and Topics in JMS

❖ Kafka is a queue system per consumer in consumer group so load balancing like JMS queue

❖ Kafka is a topic/pub/sub by offering Consumer Groups which act like subscriptions

    ❖ Broadcast to multiple consumer groups

❖ By design Kafka is better suited for scale due to partition topic log

❖ Also by moving location in log to client/consumer side of equation instead of the broker, less tracking required by Broker

❖ Handles parallel consumers better

Kafka scalable message storage

❖ Kafka acts as a good storage system for records/messages

❖ Records written to Kafka topics are persisted to disk and replicated to other servers for fault-tolerance

❖ Kafka Producers can wait on acknowledgement

  ❖ Write not complete until fully replicated

❖ Kafka disk structures scales well

  ❖ Writing in large streaming batches is fast

❖ Clients/Consumers control read position (offset)

  ❖ Kafka acts like high-speed file system for commit log storage, replication

Kafka Stream Processing

- ❖ Kafka for Stream Processing

    - ❖ Kafka enable **real-time** processing of streams.

- ❖ Kafka supports stream processor

    - ❖ Stream processor takes continual streams of records from input topics, performs some processing, transformation, aggregation on input, and produces one or more output streams

- ❖ A video player app might take in input streams of videos watched and videos paused, and output a stream of user preferences and gear new video recommendations based on recent user activity or aggregate activity of many users to see what new videos are hot

- ❖ Kafka Stream API solves hard problems with out of order records, aggregating across multiple streams, joining data from multiple streams, allowing for stateful computations, and more

- ❖ Stream API builds on core Kafka primitives and has a life of its own

Using Kafka Single Node

Run Kafka

❖ Run ZooKeeper

❖ Run Kafka Server/Broker

❖ Create Kafka Topic

❖ Run producer

❖ Run consumer

Run ZooKeeper

```
run-zookeeper.sh

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

```
rick@Richards-MacBook-Pro-2.local:~/kafka-training
$ ./run-zookeeper.sh
rick@Richards-MacBook-Pro-2.local:~/kafka-training
$ [2017-04-14 17:45:53,408] INFO Accepted socket connection from /0:0:0:0:0:0:0:1:56952 (org.apache.zookeeper.server.NIOServer
CnxnFactory)
[2017-04-14 17:45:53,415] INFO Client attempting to establish new session at /0:0:0:0:0:0:0:1:56952 (org.apache.zookeeper.serv
er.ZooKeeperServer)
[2017-04-14 17:45:53,417] INFO Established session 0x15b6ec06f690014 with negotiated timeout 6000 for client /0:0:0:0:0:0:0:1:
56952 (org.apache.zookeeper.server.ZooKeeperServer)
[2017-04-14 17:45:57,612] INFO Reading configuration from: kafka/config/zookeeper.properties (org.apache.zookeeper.server.quor
um.QuorumPeerConfig)
```

Run Kafka Server

```
run-kafka.sh  ×
1    #!/usr/bin/env bash
2    cd ~/kafka-training
3
4    kafka/bin/kafka-server-start.sh kafka/config/server.properties
```

```
rick@Richards-MacBook-Pro-2.local:~/kafka-training
[$ kafka/bin/kafka-server-start.sh kafka/config/server.properties
 [2017-04-14 17:49:09,709] 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
```

Create Kafka Topic

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

cd ~/kafka-training

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

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

Kafka Producer

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

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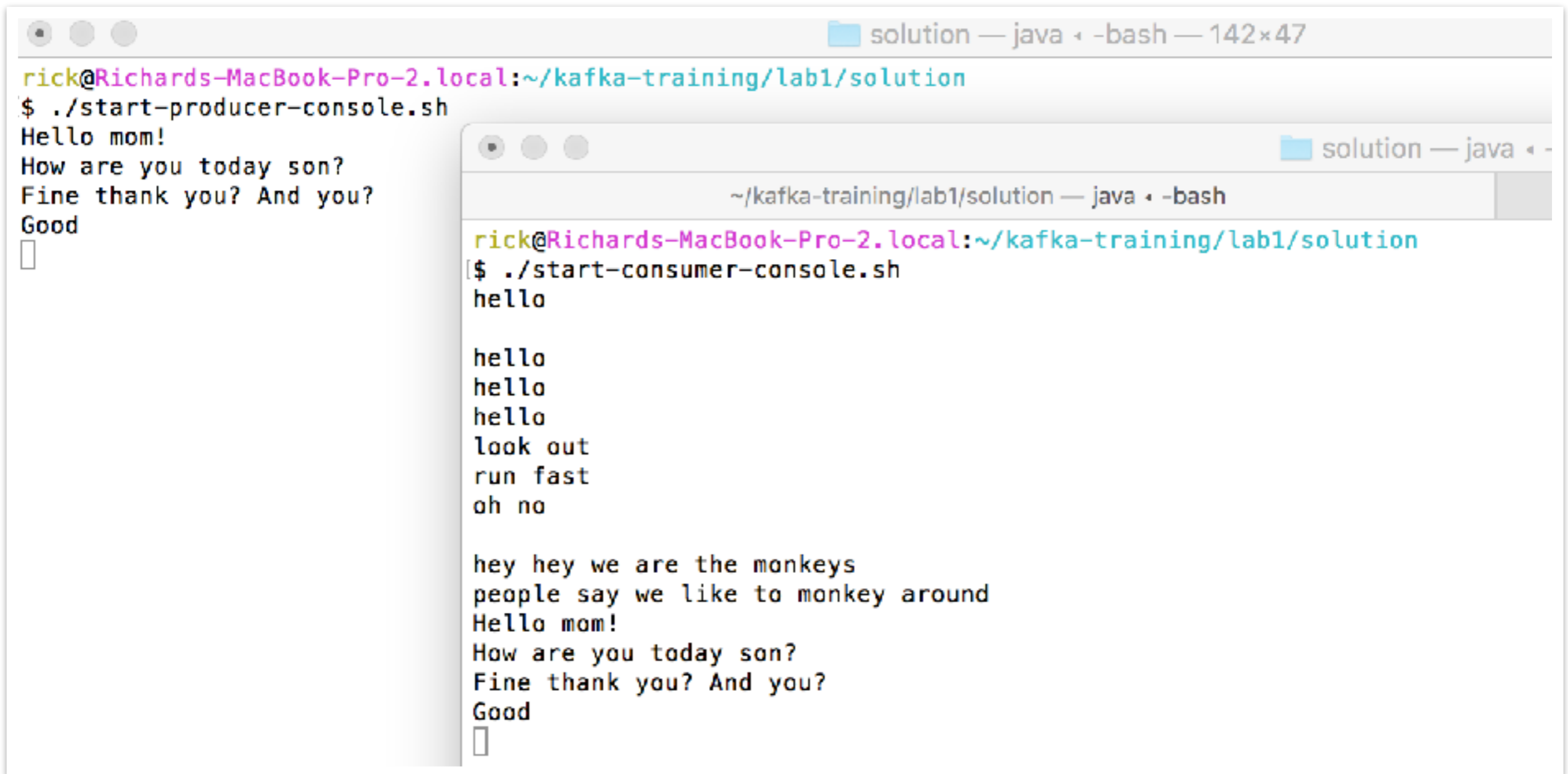
Kafka Consumer

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

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

Running Kafka Producer and Consumer

Use Kafka to send and receive messages

Lab 1-A Use Kafka

Use single server version of Kafka

CLOUDURABLE ™

Using Kafka Cluster

Running many nodes

- ❖ Modify properties files

  - ❖ Change port

  - ❖ Change Kafka log location

- ❖ Start up many Kafka server instances

- ❖ Create Replicated Topic

Leave everything from before running

```
run-zookeeper.sh  ×

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

```
run-kafka.sh  ×

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

Create two new server.properties files

❖ Copy existing *server.properties* to *server-1.properties*, *server-2.properties*

❖ Change *server-1.properties* to use *port 9093, broker id 1,* and *log.dirs " /tmp/kafka-logs-1"*

❖ Change *server-2.properties* to use *port 9094, broker id 2,* and *log.dirs " /tmp/kafka-logs-2"*

server-x.properties

```
server-1.properties  ×
1   broker.id=1
2   port=9093
3   log.dirs=/tmp/kafka-logs-1
4
5
```

```
server-2.properties  ×
1   broker.id=2
2   port=9094
3   log.dirs=/tmp/kafka-logs-2
4
```

Start second and third servers

```
start-2nd-server.sh ×    start-3rd-server.sh ×

1    #!/usr/bin/env bash
2    CONFIG=`pwd`/config
3    cd ~/kafka-training
4    kafka/bin/kafka-server-start.sh $CONFIG/server-1.properties
```

```
start-2nd-server.sh ×    start-3rd-server.sh ×

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

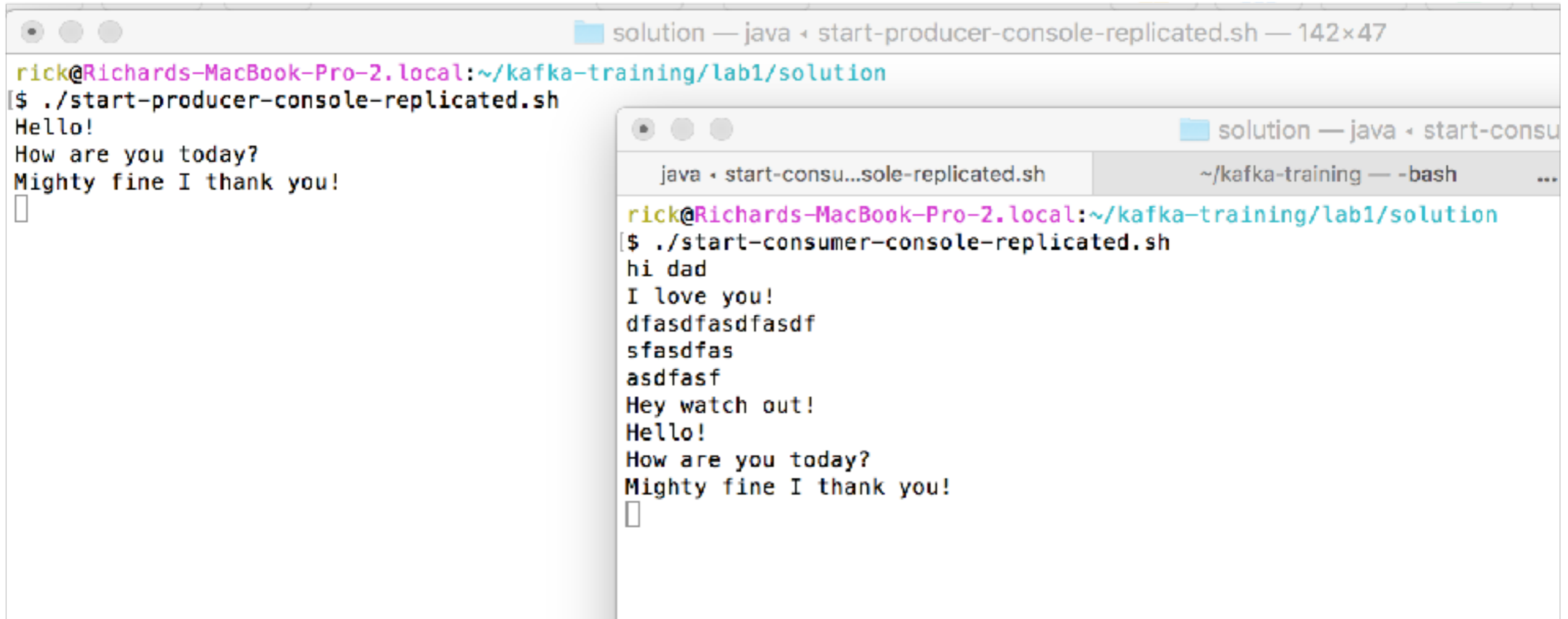# Create Kafka replicated topic **my-failsafe-topic**

```
create-replicated-topic.sh  ×

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

Start Kafka consumer and producer

```
start-producer-console-replicated.sh ×    start-consumer-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-console-replicated.sh ×    start-consumer-console-replicated.sh ×
1    #!/usr/bin/env bash
2    cd ~/kafka-training
3
4    kafka/bin/kafka-console-consumer.sh --bootstrap-server \
5    localhost:9092 --topic my-failsafe-topic --from-beginning
```

Kafka consumer and producer running

Use Kafka Describe Topic

```
$ kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-failsafe-topic
Topic:my-failsafe-topic PartitionCount:1      ReplicationFactor:3      Configs:
        Topic: my-failsafe-topic      Partition: 0    Leader: 0      Replicas: 0,2,1 Isr: 0,1,2
rick@Richards-MacBook-Pro-2.local:~/kafka-training
```

The leader is broker 0

There is only one partition

There are three in-sync replicas (ISR)

Test Failover by killing 1st server

```
[$ ps aux | grep "server.properties" | tr -s " " | cut -d " " -f2 | head -n 1
24822
rick@Richards-MacBook-Pro-2.local:~/kafka-training
[$ kill 24822
```

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

```
[$ kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-failsafe-topic
Topic:my-failsafe-topic	PartitionCount:1	ReplicationFactor:3	Configs:
	Topic: my-failsafe-topic	Partition: 0	Leader: 2	Replicas: 0,2,1 Isr: 1,2
rick@Richards-MacBook-Pro-2.local:~/kafka-training
```

NEW LEADER IS 2!

CLOUDURABLE™

Use Kafka to send and receive messages

Lab 2-A Use Kafka

Use a Kafka Cluster to replicate a Kafka topic log

Kafka Consumer and Producers

Working with producers and consumers

Step by step first example

CLOUDURABLE ™

Objectives Create Producer and Consumer example

❖ Create simple example that creates a *Kafka Consumer* and a *Kafka Producer*

❖ Create a new replicated *Kafka topic*

❖ *Create Producer* that uses topic to send records

❖ *Send records* with *Kafka Producer*

❖ *Create Consumer* that uses topic to receive messages

❖ *Process messages* from Kafka with *Consumer*

Create Replicated Kafka Topic

```
create-topic.sh  ×
1   #!/usr/bin/env bash
2   cd ~/kafka-training
3   kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
4   --replication-factor 3 --partitions 1 --topic my-example-topic
5   kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
```

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

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       testCompile group: 'junit', name: 'junit', version: '4.11'
14       compile group: 'org.apache.kafka', name: 'kafka-clients', version: '0.10.2.0'
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

```java
package com.cloudurable.kafka;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

public class KafkaExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";
```

Create Kafka Producer to send records

```java
private static Producer<Long, String> createProducer() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
    return new KafkaProducer<>(props);
}
```

Send async records with Kafka Producer

```java
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();
```

Send **sync** records with Kafka Producer

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

    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            final ProducerRecord<Long, String> record =
                    new ProducerRecord<>(TOPIC, index,  value: "Hello Mom " + index);

            RecordMetadata metadata = producer.send(record).get();

            long elapsedTime = System.currentTimeMillis() - time;
            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);

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

Create Consumer using Topic to Receive Records

❖ Specify bootstrap servers

❖ Specify client.id

❖ Specify Record Key deserializer

❖ Specify Record Value deserializer

❖ Specify Consumer Group

❖ Subscribe to Topic

Create Consumer using Topic to Receive Records

```java
private static Consumer<Long, String> createConsumer() {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleConsumer");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            LongDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());
    Consumer<Long, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Collections.singletonList(TOPIC));
    return consumer;
}
```
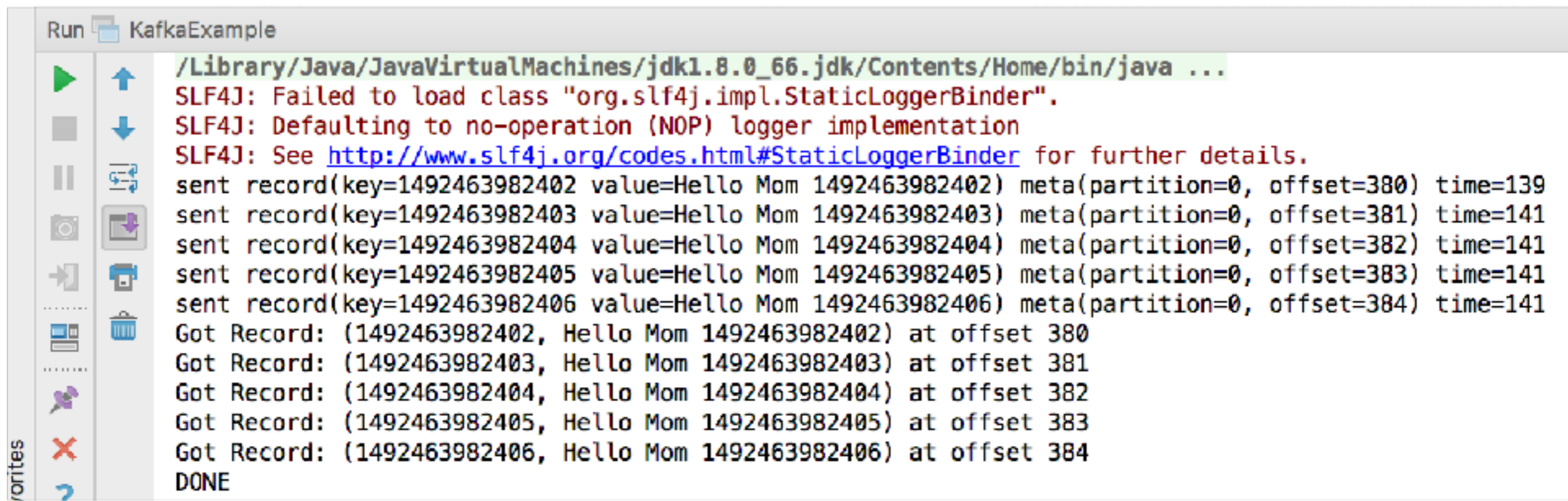
Process messages from Kafka with Consumer

```java
static void runConsumer() throws InterruptedException {
    Consumer<Long, String> consumer = createConsumer();

    while (true) {
        final ConsumerRecords<Long, String> consumerRecords = consumer.poll( timeout: 100);

        if (consumerRecords.count()==0) {
            break;
        }

        consumerRecords.forEach(record -> {
            System.out.println("Got Record: (" + record.key() + ", " + record.value()
                    + ") at offset " + record.offset());
        });
        consumer.commitAsync();
    }
    consumer.close();
    System.out.println("DONE");
}
```

Running both Consumer and Producer

```java
public static void main(String... args) throws InterruptedException {
    runProducer( sendMessageCount: 5);
    runConsumer();
}
```

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

Java Kafka simple example recap

- Created simple example that creates a *Kafka Consumer* and a *Kafka Producer*

- Created a new replicated *Kafka topic*

- *Created Producer* that uses topic to send records

- *Send records* with *Kafka Producer*

- *Created Consumer* that uses topic to receive messages

- *Processed records* from Kafka with *Consumer*