Integrating Apache Kafka with Big Data and NoSQL Using Python and Spark

ROITRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

The following course materials are copyright protected materials. They may not be reproduced or distributed and may only be used by students attending the *Integrating Apache Kafka with Big Data and NoSQL Using Python and Spark* course.

INTRODUCTION

Welcome!

- ROI leads the industry in designing and delivering customized technology and management training solutions
- Meet your instructor
  - Name
  - Background
  - Contact info
- Let's get started!

Course Objectives

In this course, we will:

- Describe the architecture of Kafka
- Explore Kafka producers and consumers for writing and reading messages
- Understand publish-subscribe messaging and how it fits in the big data ecosystem
- Explore how Kafka's stream delivery capabilities make it a perfect source for stream processing systems
- Learn various strategies of monitoring Kafka
- Get best practices for building data pipelines and applications with Kafka

Course Contents

Class Schedule

- Start of class _____
- Morning breaks approximately on the hour
- Lunch _____
- Afternoon breaks approximately on the hour
- Class end _____

Student Introductions

Please introduce yourself stating:

- Name
- Position or role
- How many years of experience you have with Apache Kafka
- Expectations or a question you'd like answered during this class

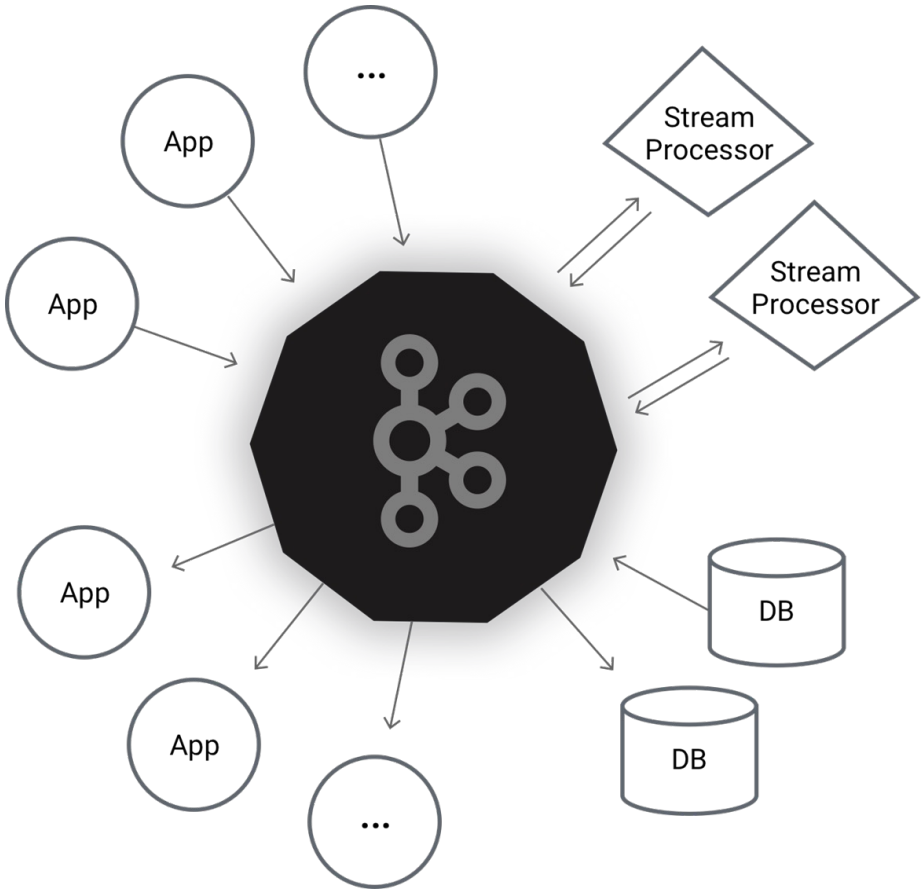
ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

CHAPTER 1: KAFKA ARCHITECTURE

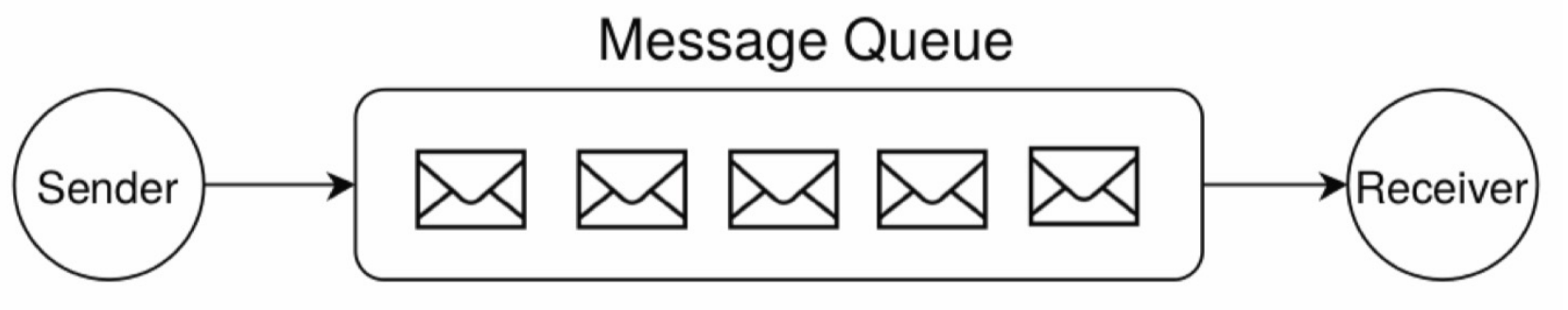
What Is Kafka?

- Apache Kafka is a distributed streaming platform
  - It can be used for anything ranging from a distributed message broker to a platform for processing data streams
- Kafka was originally developed by LinkedIn and was subsequently open sourced in early 2011
- Kafka was created to address the data pipeline problem at LinkedIn
- It was designed to provide a high-performance messaging system that can handle many types of data in real time



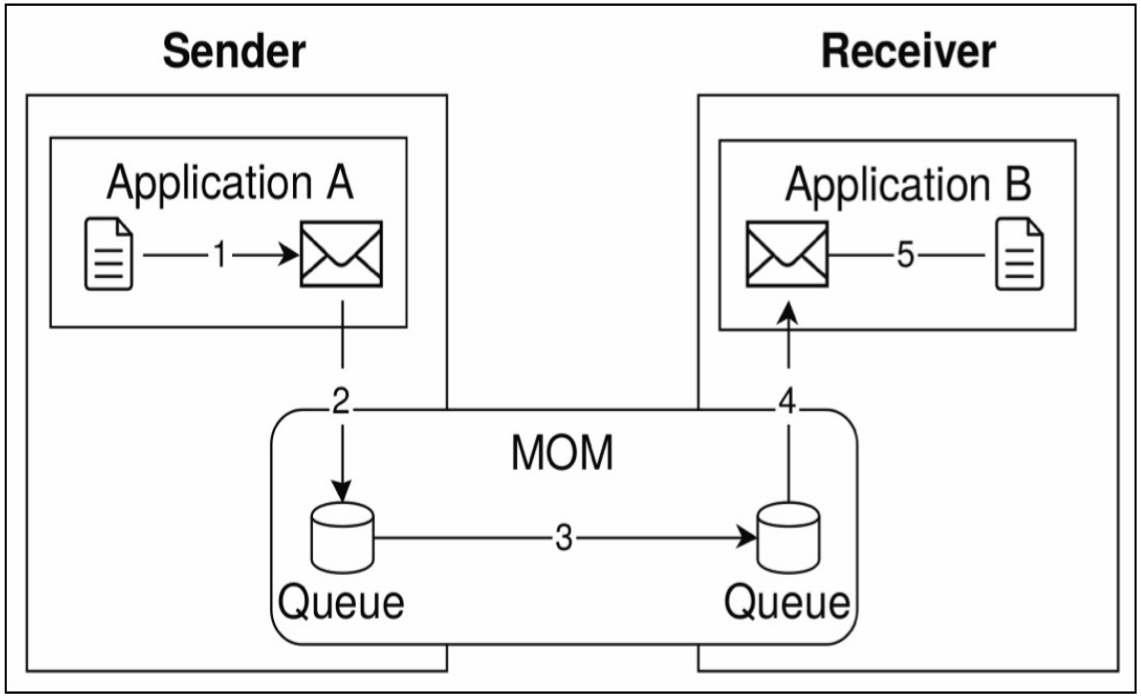
Background Concepts: Messaging

- Messaging enables the flow of information in a distributed system
- Messaging is usually based on one or more shared buffers and Message Queues
- One application can publish its messages to the queue and another application can asynchronously read from it at any time
- A message is just a simple and independent unit with a data structure which contains a meta header and the transmitted information
  - The messaging technology enables high- speed, asynchronous, program-to-program communication with reliable delivery



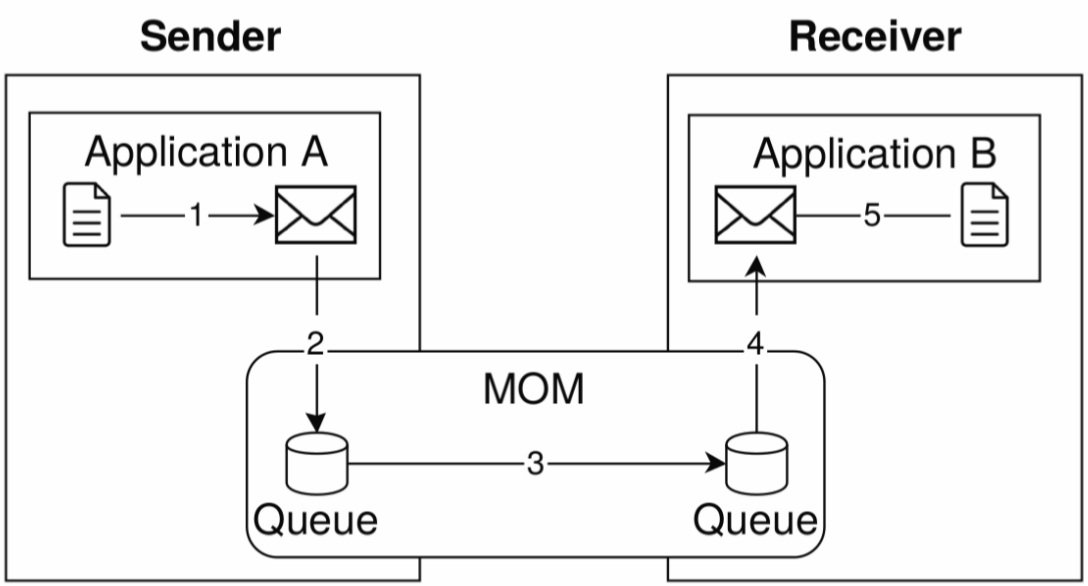
Message-Oriented Middleware (MOM)

- Message-oriented middleware (MOM) is responsible for passing messages in any format from one application to another where the parties do not need to know each other directly
- The MOM provides a specific API for both the sender and the receiver



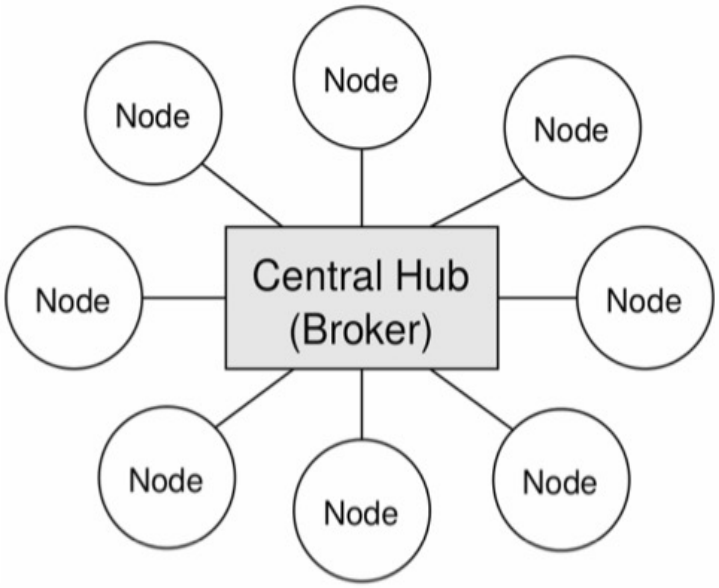
Flow of Communication in MOM

- The sender packs its data into the appropriate message format (1) and puts the message in a local queue (2)
- The MOM then delivers the message over the network to the local queue of the receiver (3)
- Finally, the receiver system fetches the message and stores it in its own local queue (4) to further unpack and finally get the transmitted data (5)



Topologies of Messaging Systems: Central Hub (Message Broker)

- A message broker decouples source and target systems
- The main tasks are:
  - Dynamic registration of endpoints
  - Routing of the messages
  - Serialization/deserialization of the messages before sending and receiving
  - Temporal persistence of messages

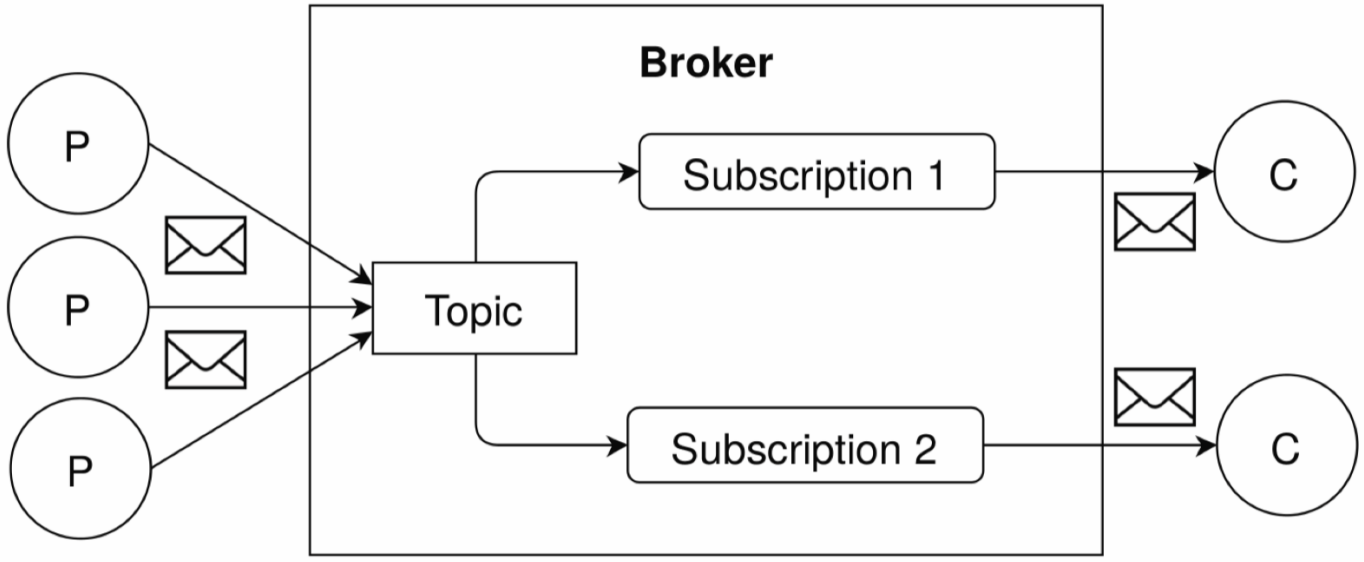


Event-Based vs. Pull-Based Consumption

- Message brokers can be push-based or pull-based
  - In push-based brokers, the message is pushed to the consumer
  - In pull-based brokers, the message is pulled by the consumer from the broker

Topic and Subscription Model

- Message broker systems are persisting incoming messages in an enhanced type of a message queue, named topic
- A topic allows the broker to deliver a message to multiple independent consumers
- Sending messages to the broker in the form of publishing to a specific topic and on the other hand receiving messages only for the specified topic, is called publish/subscribe



Characteristics of a Messaging System

- Independence from any specific technology
- Scalability
- Fault tolerance
- Ability to offer durable persistence

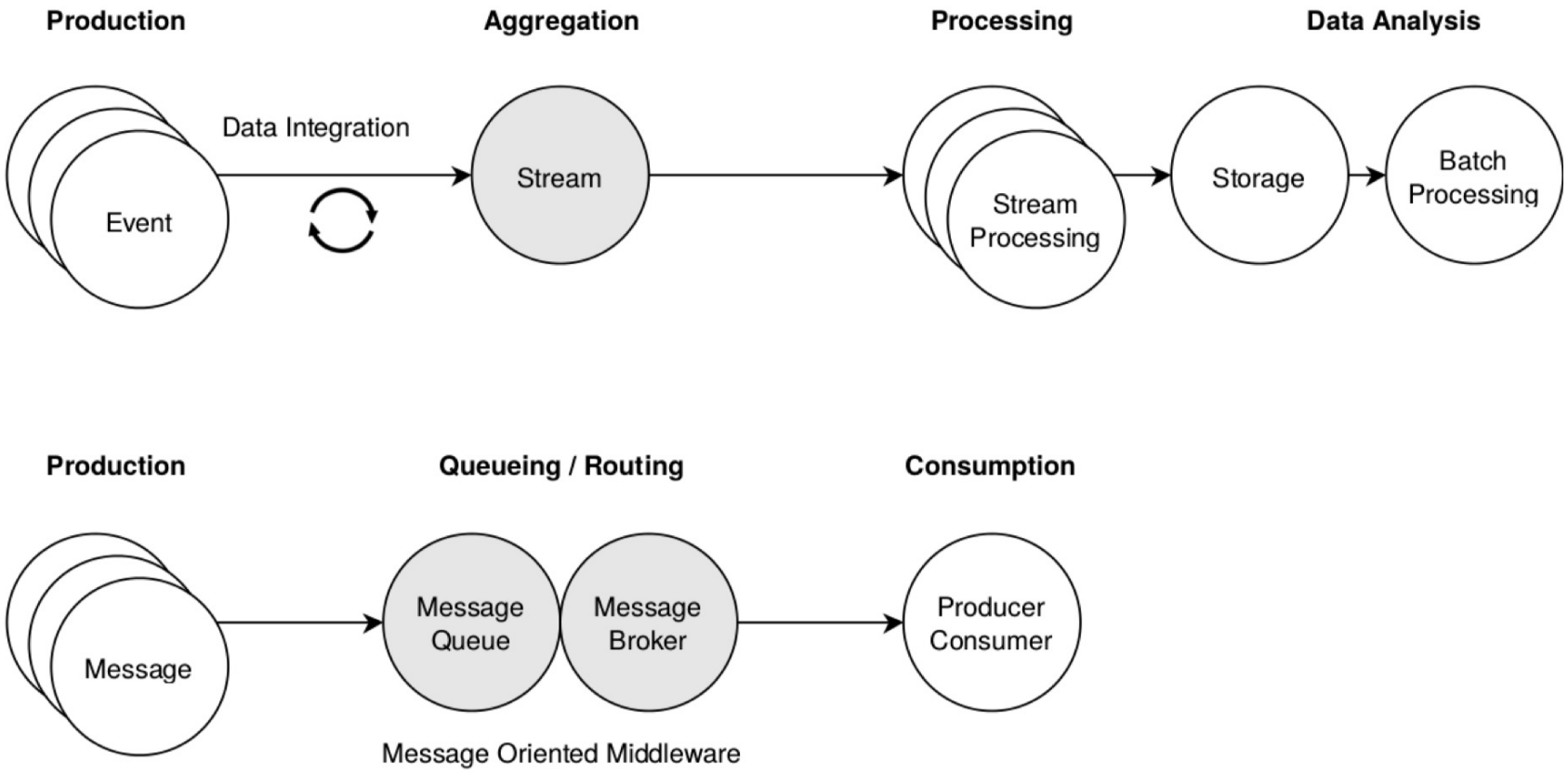
Characteristics of Kafka

- Persistent messaging
  - To derive the real value from big data, any kind of information loss cannot be afforded
  - Apache Kafka is designed with O(1) disk structures that provide constant-time performance even with very large volumes of stored messages that are in the order of TBs
  - With Kafka, messages are persisted on disk, as well as replicated within the cluster to prevent data loss
- High throughput
  - Keeping big data in mind, Kafka is designed to work on commodity hardware and to handle hundreds of MBs of reads and writes per second from large number of clients
- Distributed
  - Apache Kafka explicitly supports message partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics
  - Kafka cluster can grow elastically and transparently without any downtime

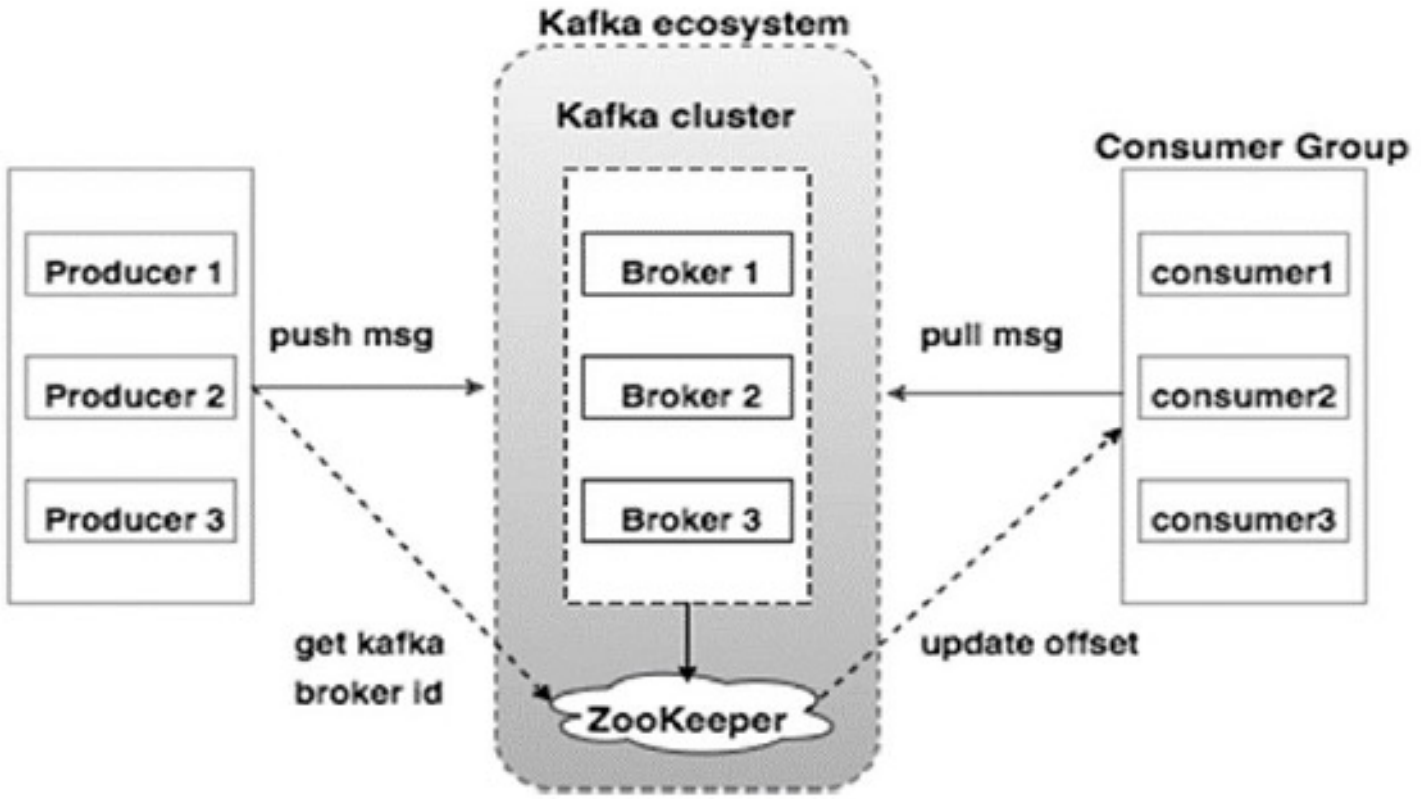
Characteristics of Kafka (continued)

- Multiple client support
  - It supports easy integration of clients from different platforms, such as Java, .NET, PHP, Ruby, and Python
- Real time
  - Messages produced by the producer threads should be immediately visible to consumer threads
    - This feature is critical to event-based systems such as Complex Event Processing (CEP) systems

Event Stream vs. Messaging

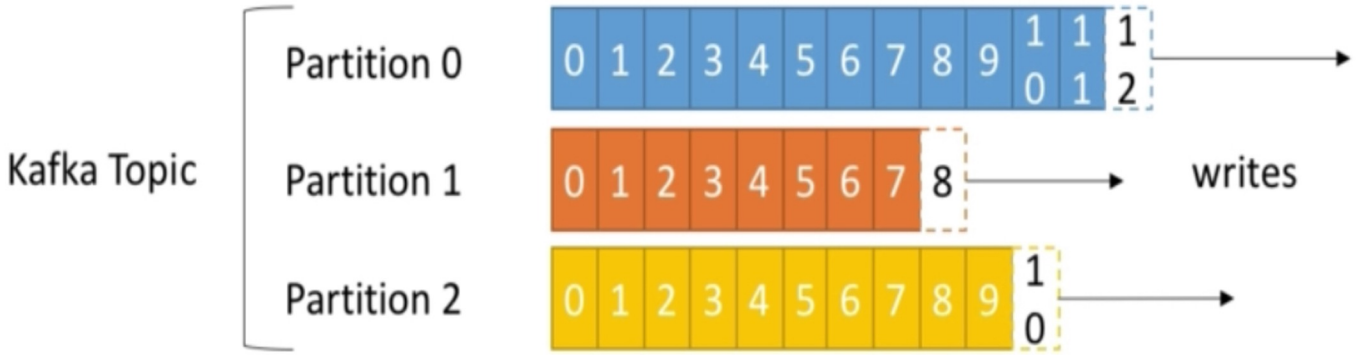


Apache Kafka — Cluster Architecture



Topics, Partitions, Offsets, and Replicas

- Topics are a particular stream of data
  - Similar to a table in a database
  - You can have as many topics as needed
  - A topic is identified by its name
- Topics are split in partitions
  - Each partition is ordered
  - Each message within a partition gets an incremental id, called offset
- Replicas are "backups" of a partition



Brokers

- Kafka, as a distributed system, runs in a cluster
  - Each node in the cluster is called a Kafka broker
- Brokers are responsible for maintaining the published data
  - Each broker may have zero or more partitions per topic
- Brokers are the "pipes" in the data pipeline, which store and emit data
  - A consumer pulls messages off a Kafka topic while producers push messages into a Kafka topic

Producers

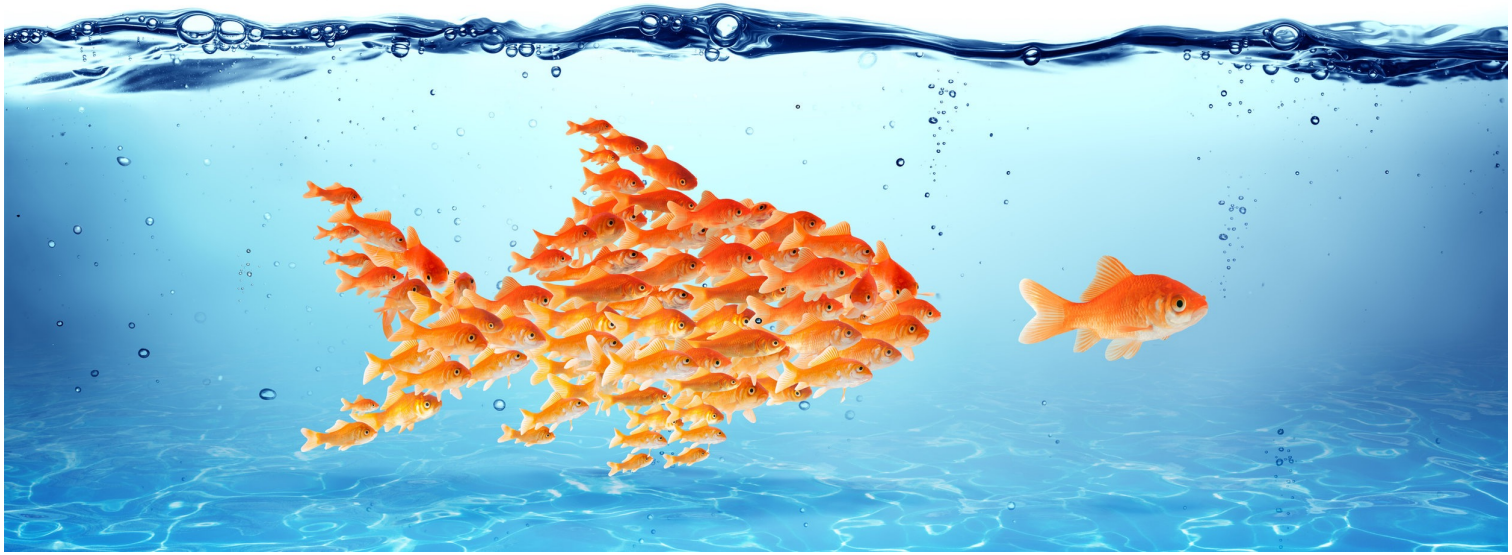
- Producers generate data and send it to the Kafka brokers
- Every time a producer publishes a message to a broker, the broker simply appends the message to a partition
- Producer can also send messages to a partition of their choice

Consumers

- Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers
- The consumer has to maintain how many messages have been consumed by using partition offset as Kafka brokers are stateless
- If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages

Leader and Follower

- *Leader* is the node responsible for all reads and writes for the given partition
  - Every partition has one server acting as a leader
- Nodes which follow leader instructions are called *follower*
  - If the leader fails, one of the followers will automatically become the new leader
  - A follower keeps itself in-sync with the leader



ZooKeeper and Kafka

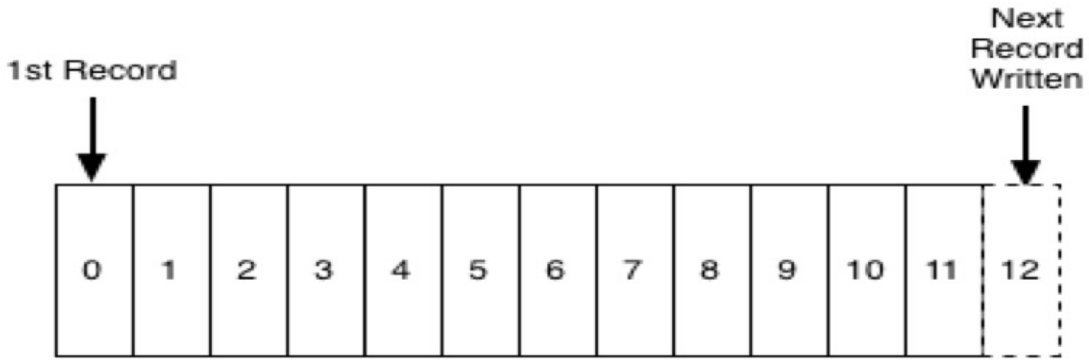
- ZooKeeper handles the leader election in case of failure
  - In order to do this, a leader node constantly informs ZooKeeper about changes in its ISR (in-sync replica) set
  - If the leader node crashes, all surviving followers register themselves in ZooKeeper
  - The replica that registers first becomes the new leader

Publish/Subscribe Messaging

- The sender (publisher) of a piece of data (message) sends data, not specifically directing it to a receiver
- The publisher classifies the message somehow, and that receiver (subscriber) subscribes to receive certain classes of messages
- Pub/sub systems often have a broker, where messages are published, to facilitate this

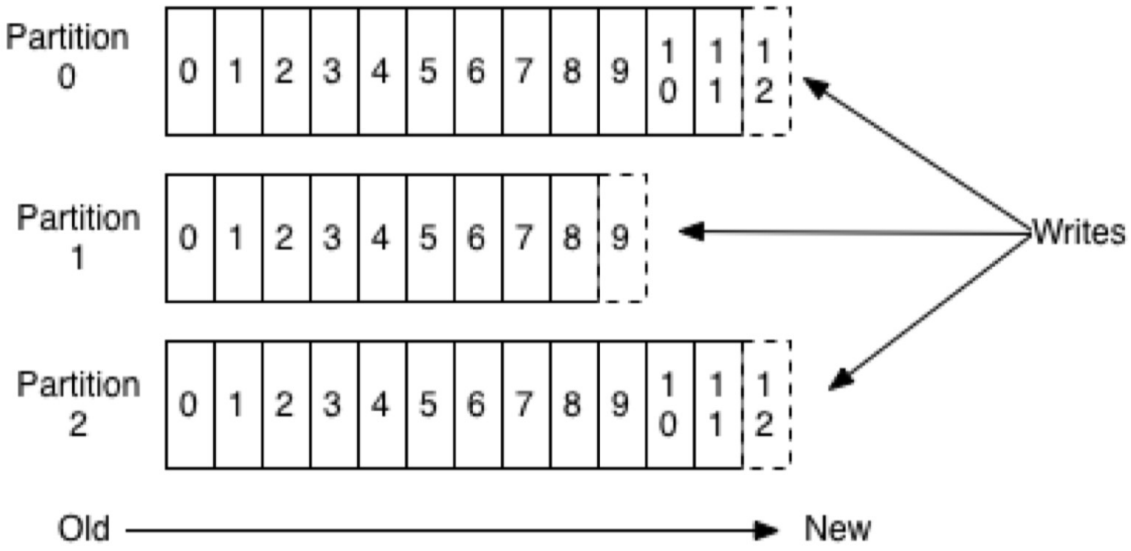
The Log

- Apache Kafka uses a so-called *commit log* for managing messages within the broker
- It is a simple, append-only data structure which contains a sequence of records ordered by time where each entry is assigned to a unique number called offset
- Apache Kafka handles its own log for every topic which contains all published messages as single records
- Kafka does not delete a record after consumption
- Every consumer controls its position in the log on its own



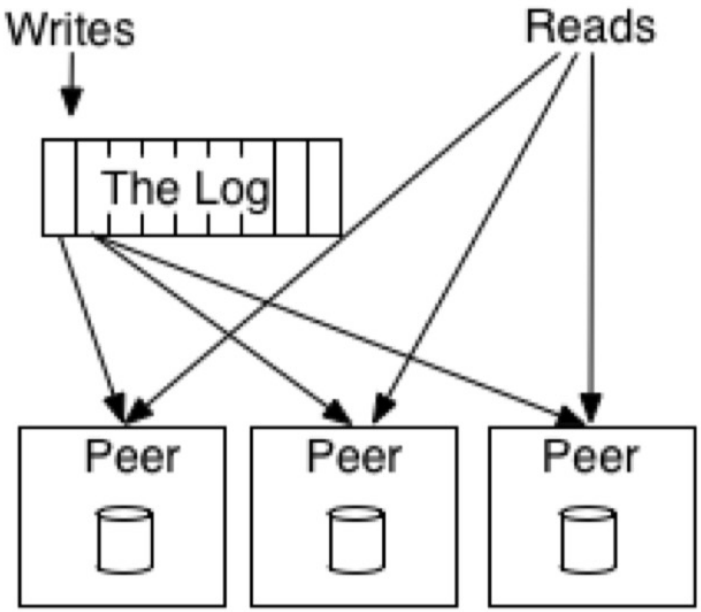
Partitioning

- Partitioning is a way to parallelize the consumption of the messages
- The total number of partitions in a broker cluster needs to be at least the same as the number of consumers (or producers) in a consumer (or producer) group
- Each partition is consumed by exactly one consumer (in a consumer group) and by doing so, Kafka ensures that the consumer is the only reader of that partition and consumes the data in order



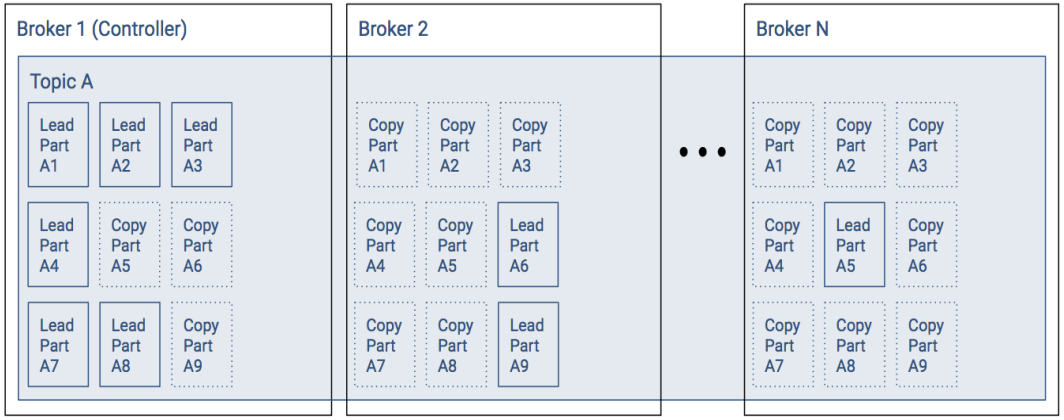
Replication

- Apache Kafka supports replication on the level of log partitions
  - Thereby, it can replicate specific topics across a configurable number of other Kafka servers
- Every topic has defined a leader node which is active in normal operation
  - Producers send data directly to the leader broker
- A leader node can have zero or more follower nodes which are responsible for replicating the entries of the active log
- The followers do this by simply acting as a normal Kafka consumer of the leader node and constantly update their own log so that it is identical



In-Sync Replica (IRS)

- The leader node constantly keeps track of its followers by managing a set of *in-sync* nodes, called ISR (in-sync replica)
- If one of the followers crash or falls behind in replicating, the leader will dynamically remove it from its ISR
- If a follower comes back and catches up to the actual state, the leader will reinsert it
- Only members of ISR can be elected as the new leader
- An incoming message needs to be replicated by every *in-sync* follower before any other consumer can get it
- A fully replicated message is considered as *committed*
  - This guarantees that the consumer does not need to worry about potentially seeing a message that could be lost if the leader fails



ZooKeeper

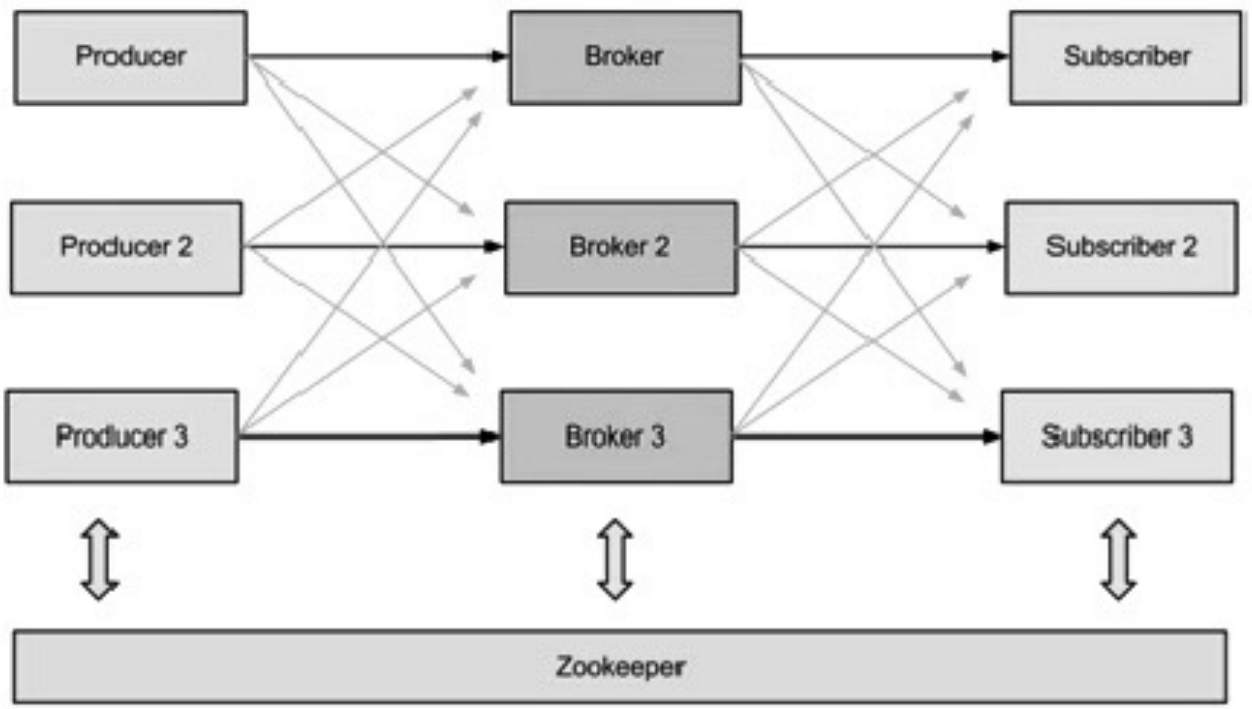
- ZooKeeper is used for managing and coordinating Kafka broker
- ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system
- As per the notification received by the ZooKeeper regarding presence or failure of the broker, the producer and consumer takes decision and starts coordinating their task with some other broker

ZooKeeper (continued)

- ZooKeeper manages broker (keeps a list of them)
- ZooKeeper helps in performing leader election for partitions
- ZooKeeper sends notifications to Kafka in case of changes
  - New topics are created
  - Broker dies
  - Broker comes up
  - Topic is deleted
- ZooKeeper by design operates with an odd number of servers
- ZooKeeper has a leader (handle writes), the rest of the server are followers (handle reads)
- ZooKeeper does not store consumer offsets with Kafka >v0.10

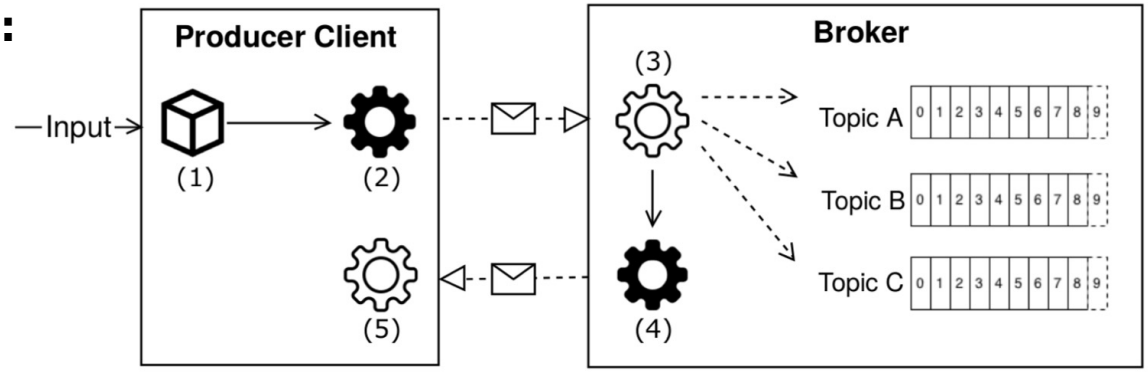
ZooKeeper as Kafka's Clustering Manager

- Apache Kafka always runs with Apache ZooKeeper as the underlying manager for all Kafka nodes
- ZooKeeper handles the coordination between the leader and follower nodes and also notifies producer and consumer about the presence or failure of brokers in a Kafka system



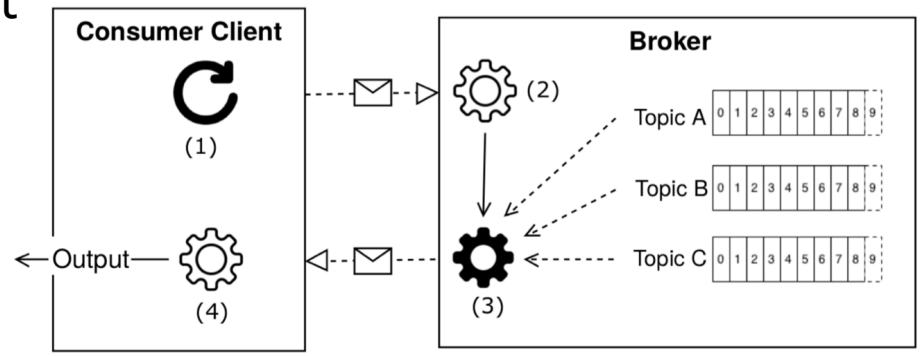
Produce Messages, Details

1. **Packing Produce Request:** Transforming input data into protocol-conforming data structure.
2. **Serializing and Sending Produce Request:** Encoding data structure to a binary data and transmitting over network to the broker.
3. **Parsing and Handling Produce Request:** Broker receives the binary string and parses it back to the appropriate data structure. The request handler of the broker checks the API Key of the request. If it is a produce request, the containing message will be written to the appropriate topic log.
4. **Send Produce Response:** A response is packed, serialized, and transmitted back to the client. The response contains an error code which determines if everything worked well or a specific problem occurred.
5. **Parse Produce Response:** Producer client receives a binary string and parses it to a valid response data structure.



Consume Messages, Details

1. **Continuously Send Fetch Request:** Consumer client sends fetch requests in configurable interval as a binary string to the broker.
2. **Parsing and Handling Fetch Request:** Broker receives the binary string and parses it back to the appropriate data structure. The request handler of the broker checks the API Key of the request. If it is a fetch request, the broker reads messages of the requested topic and packs them to a fetch response.
3. **Send Fetch Response:** The fetch request which contains the requested messages is sent back to the consumer client.
4. **Parse Response:** Consumer client receives a binary string and parses it to a valid response data structure.



Installing ZooKeeper

- Step 1:
  - Download ZooKeeper: http://zookeeper.apache.org/releases.html
- Step 2:
  - Extract tar file

```
$ tar -zxf zookeeper-3.4.6.tar.gz
$ cd zookeeper-3.4.6
$ mkdir data
```

- Step 3:
  - Create configuration file

```
$ vi conf/zoo.cfg
tickTime=2000
dataDir=/path/to/zookeeper/data
clientPort=2181
initLimit=5
syncLimit=2
```

Running ZooKeeper

- To start ZooKeeper Server:

```
$ZOO_HOME/bin/zkServer.sh start
```

- To start CLI:

```
$ZOO_HOME/bin/zkCli.sh
```

- To stop ZooKeeper Server:

```
$ZOO_HOME/bin/zKServer.sh stop
```

Installing Kafka

- Step 1:
  - Download Kafka: https://www.apache.org/dyn/closer.cgi?path=/kafka/0.9.0.0/kafka_2.11-0.9.0.0.tgz
- Step 2:
  - Extract the tar file

```
$ tar -zxf kafka_2.11.0.9.0.0 tar.gz
$ cd kafka_2.11.0.9.0.0
```

Running Kafka

- Start server:

```
$KAFKA_HOME/bin/kafka-server-start.sh
config/server.properties
```

- Stop server:

```
$KAFKA_HOME/bin/kafka-server-stop.sh
config/server.properties
```

ZooKeeper Default Configuration File

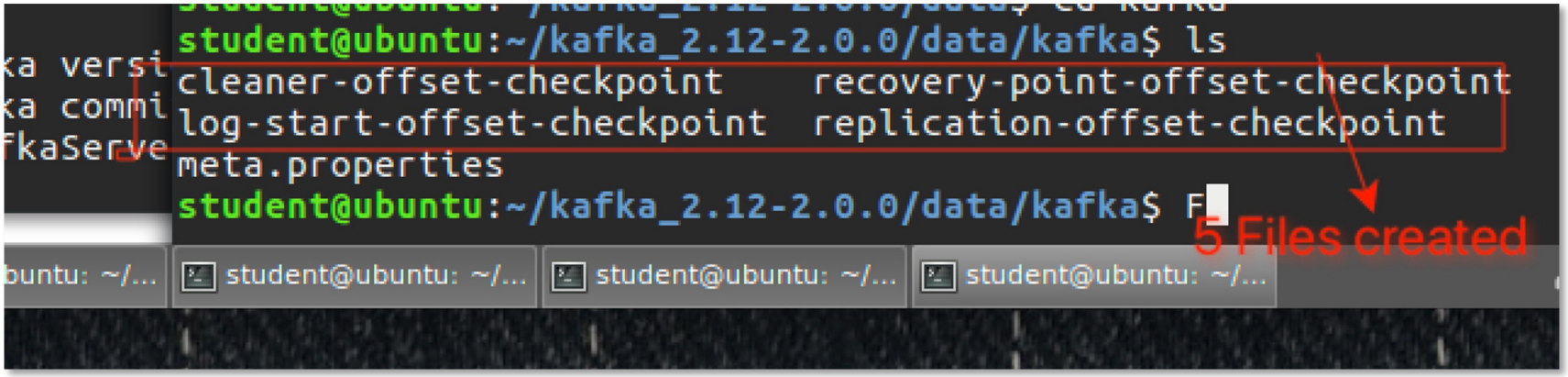
- To start ZooKeeper, you need a configuration file: `conf/zoo.cfg`
- A simple example:

```
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
```

- `tickTime` – The basic time unit in milliseconds used by ZooKeeper
  - It is used to do heartbeats and the minimum session timeout will be twice the `tickTime`
- `dataDir` – The location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database
- `clientPort` – The port to listen for client connections

Kafka Configuration

```
$KAFKA_HOME  Home Directory of Kafka Installation
$KAFKA_HOME/bin  Bin files
$KAFKA_HOME/conf  Configuration files
$KAFKA_HOME/data Data Files
```



Hardware Selection

- Kafka requires relatively small amount of resources
  - By default, Kafka can run on little as 1 core and 1GB memory with storage scaled based on requirements for data retention
- CPU is rarely a bottleneck because Kafka is I/O heavy, but a moderately-sized CPU with enough threads is still important to handle concurrent connections and background tasks

To affect these features	Adjust these parameters
Message retention	Disk size
Client throughput (Producer & Consumer)	Network capacity
Producer throughput	Disk I/O
Consumer throughput	Memory

CLI Operations — Creating a Topic

- Use `kafka-topics.sh` to create topics on the server
- Syntax:

```
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1 --partitions 1
--topic topic-name
```

- Example:

```
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1 --partitions 1
--topic Scotland
```

CLI Operations — Listing Topics

- Use the following syntax to get the list of topics:

```
$KAFKA_HOME/bin/kafka-topics.sh --list --zookeeper
localhost:2181
```

CLI Operations — Start Producer to Send Messages

- Two main parameters are required for the producer command-line client
- Broker-list – The list of brokers that we want to send the messages to
  - In this case, we only have one broker
  - The `config/server.properties` file contains broker port id
  - Since we know our broker is listening on port 9092, you can specify it directly
- Topic name – Here is an example for the topic name
  - Syntax:

```
$KAFKA_HOME/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic topic-name
```

  - Example:

```
$KAFKA_HOME/bin/kafka-console-producer.sh --broker-list
localhost:9092 --Scotland
```

CLI Operations — Start Consumer to Receive Messages

- Similar to producer, the default consumer properties are specified in `config/consumer.properties` file
- Open a new terminal and type the below syntax for consuming messages:

```
$KAFKA_HOME/bin/kafka-console-consumer.sh --zookeeper
localhost:2181 —topic topic-name --from-beginning
```

- Example:

```
$KAFKA_HOME/bin/kafka-console-consumer.sh --zookeeper
localhost:2181 —topic Scotland --from-beginning
```

CLI Operations — Modifying a Topic

- A topic that was created before can be modified by the `alter` command
- Syntax:

```
$KAFKA_HOME/bin/kafka-topics.sh —zookeeper localhost:2181
--alter --topic topic_name --partitions count
```

- Example: `alter` can be used to modify a topic "Scotland" with single partition count and one replica factor to change the partition count

```
$KAFKA_HOME/bin/kafka-topics.sh --zookeeper localhost:2181
--alter --topic Hello-kafka --partitions 2
```

CLI Operations — Deleting a Topic

- `delete` can be used to delete a topic
- Syntax:

```
bin/kafka-topics.sh --zookeeper localhost:2181 --delete
--topic topic_name
```

- Example:

```
bin/kafka-topics.sh --zookeeper localhost:2181 --delete
--topic Hello-kafka
```

CLI Operations — Describe Command

- The `describe` command is used to check which broker is listening on the current created topic
- Example:

```
$KAFKA_HOME/bin/kafka-topics.sh --describe --zookeeper
localhost:2181 --topic ScotlandReplicated
```

CLI Operations — Start Producer to Send Messages

- This procedure remains the same as shown in the single broker setup
- Example:

```
$KAFKA_HOME/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic ScotlandReplicated
```

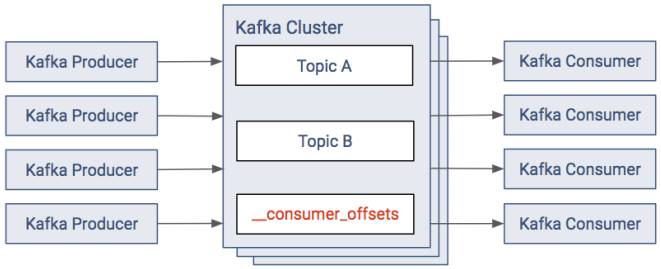
CLI Operations – Start Consumer to Receive Messages

- This procedure remains the same as shown in the single broker setup
- Example:

```
bin/kafka-console-consumer.sh --zookeeper
localhost:2181 —topic ScotlandReplicated --from-
beginning
```

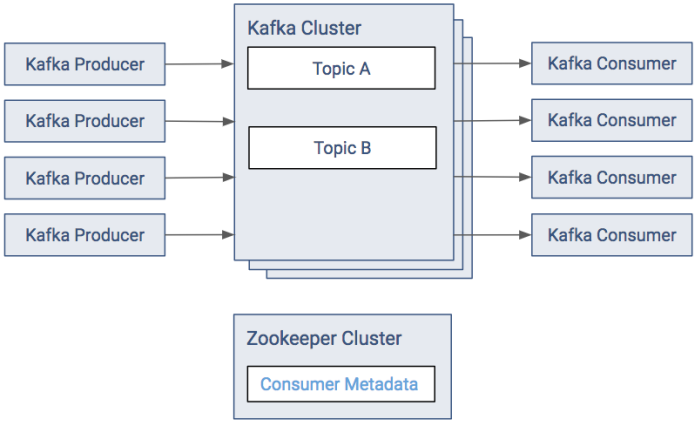
New Kafka Architecture (Without ZooKeeper)

- The default consumer model provides the metadata for offsets in the Kafka cluster
  - There is a topic named `__consumer_offsets` that the Kafka consumers write their offsets to



- In releases before version 2.0 of CDK Powered by Apache Kafka, the same metadata was located in ZooKeeper
  - The new model removes the dependency and load from ZooKeeper

- In the **old approach**:
  - The consumers save their offsets in a "consumer metadata" section of ZooKeeper
  - With most Kafka setups, there are often a large number of Kafka consumers
    - The resulting client load on ZooKeeper can be significant, therefore, this solution is discouraged



Exercise 1

- Working with Multiple Producers and Consumers
- Your instructor will provide instructions

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

CHAPTER 2: PRODUCERS AND CONSUMERS

Introduction

- Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by writing a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles
- Apache Kafka ships with built-in client APIs that developers can use when developing applications that interact with Kafka

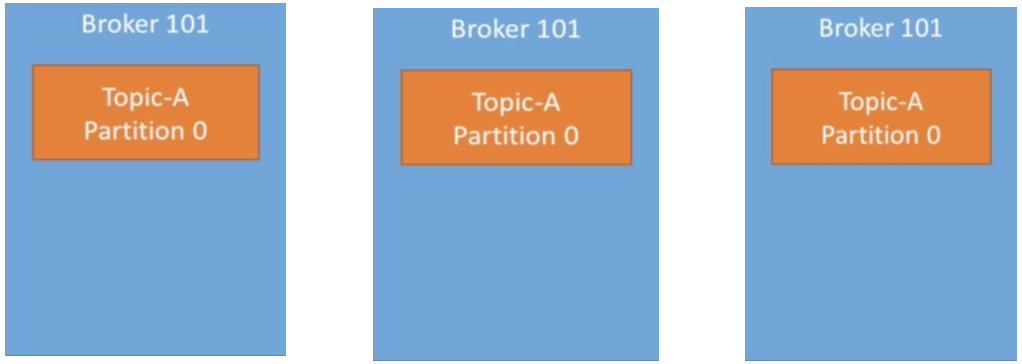
Brokers

- A Kafka cluster is composed of multiple brokers (servers)
- Each broker has its ID which is an integer
- Each broker contains topic partitions
- After connecting to any broker (called bootstrap broker), the user is connected to the entire cluster

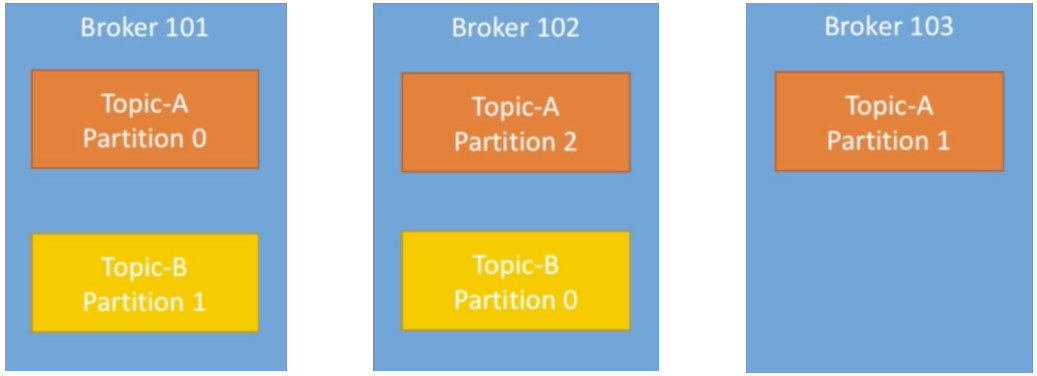
Broker 101 Broker 102 Broker 103

Brokers and Topics

- When a topic is created, Kafka automatically distributes it across the brokers
- Topic-A is created with three partitions

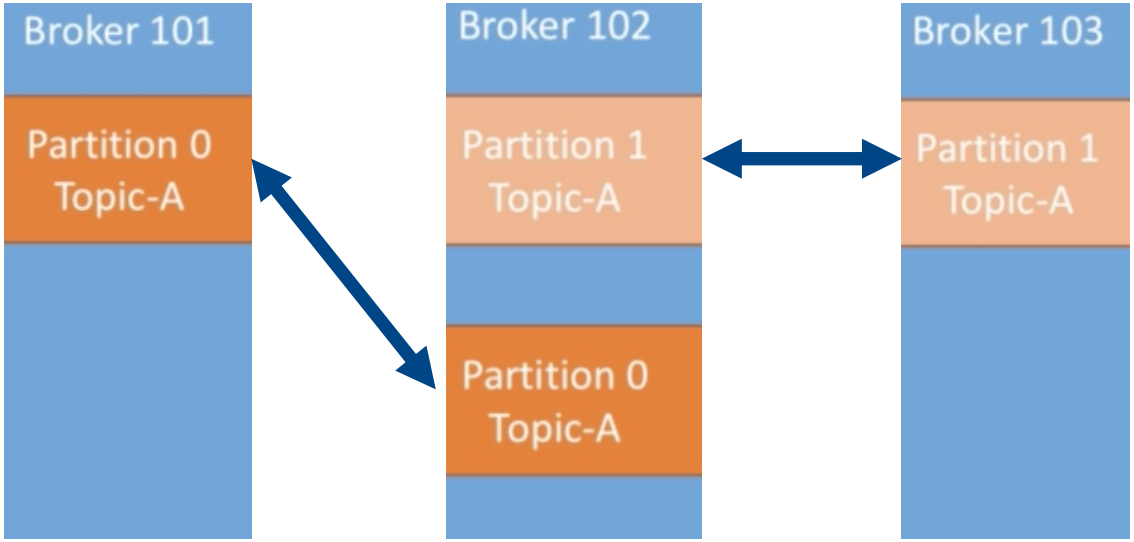


- Topic-B is added with two partitions



Topic Replication Factor

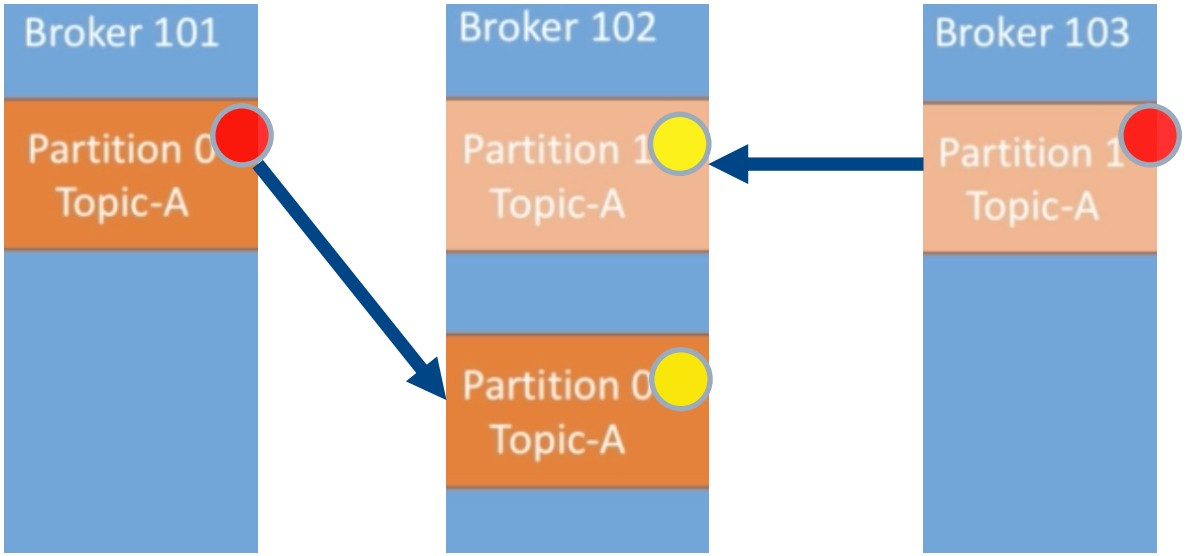
- Topics usually have a replication factor of two or three
- If a broker is down, another broker can provide the data
- Consider Topic-A with two partitions and a replication factor of two



- What happens if we lose Broker 102?

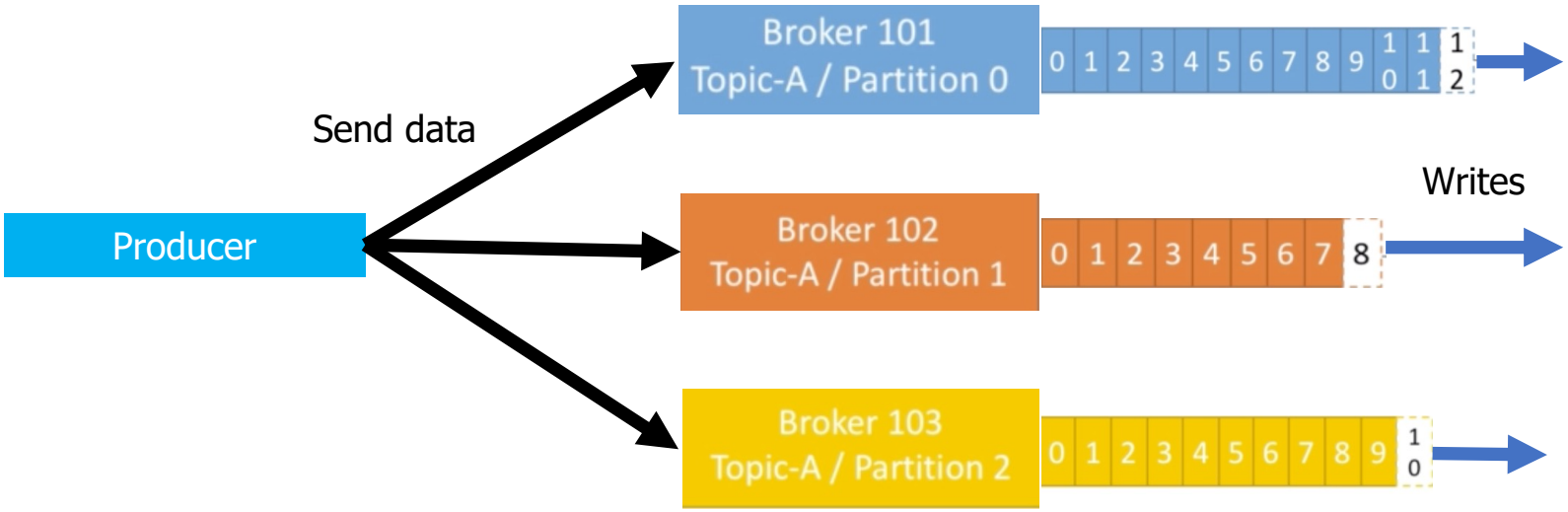
Concept of a Leader for a Partition

- At any time, only one broker can be a leader for a given partition
  - Only the leader receives data
- Each partition has one leader and multiple ISR (in-sync replica)
- ZooKeeper elects leader partitions
- For Topic-A and Topic-B, the leaders are represented by red dots and in-sync replicas are represented by yellow dots



Producers

- Producers write data to topics, which are made of partitions
- Producers know to which broker and partition to write to
- In case of broker failure, producers will automatically recover
- Load is balanced as data is written to more than one partition residing in different brokers

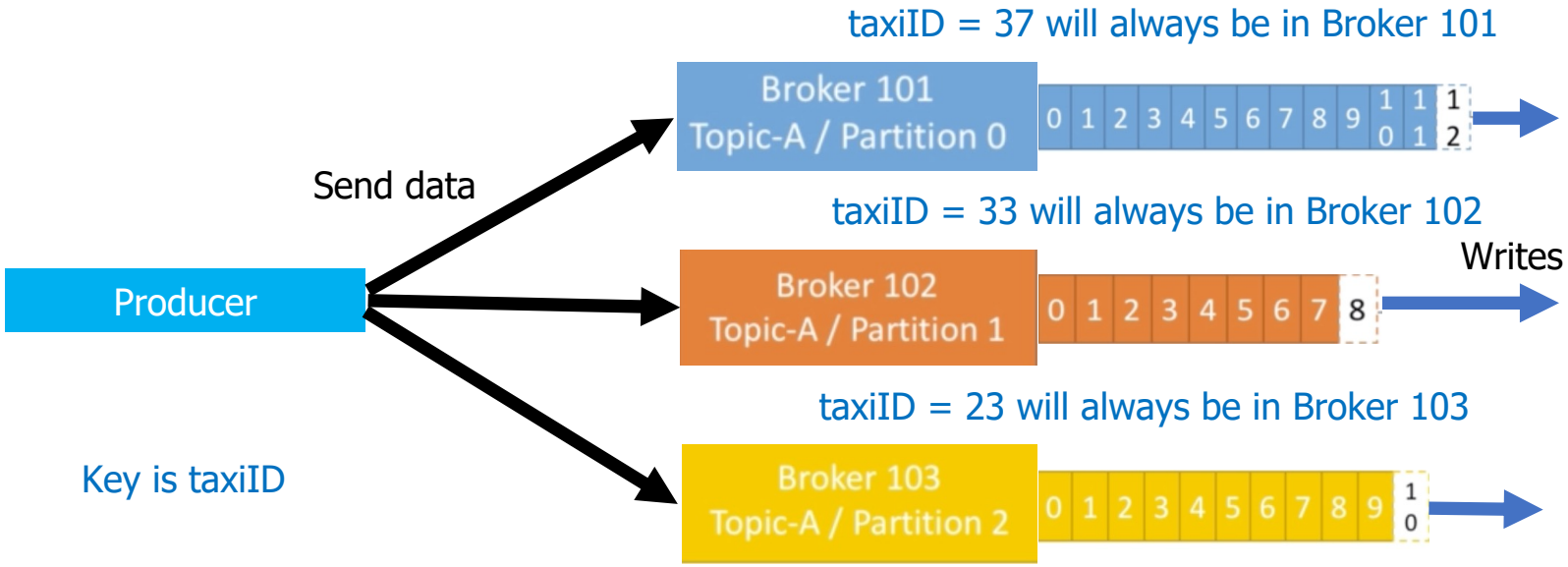


Producers – Acknowledgment Request

- Producers can choose to receive acknowledgment of data writes
- **acks = 0:**
  - Producer will not wait for acknowledgment
    - Producer sends the data, and if the broker is down, producer will not know about it
  - Implications: Data loss is possible
- **acks = 1: [DEFAULT]**
  - Producer will wait for the leader to acknowledge
  - Implications: Limited data loss is possible
- **acks = all:**
  - Producer will wait for the leader and all replicas to acknowledge
  - Implications: No data Loss

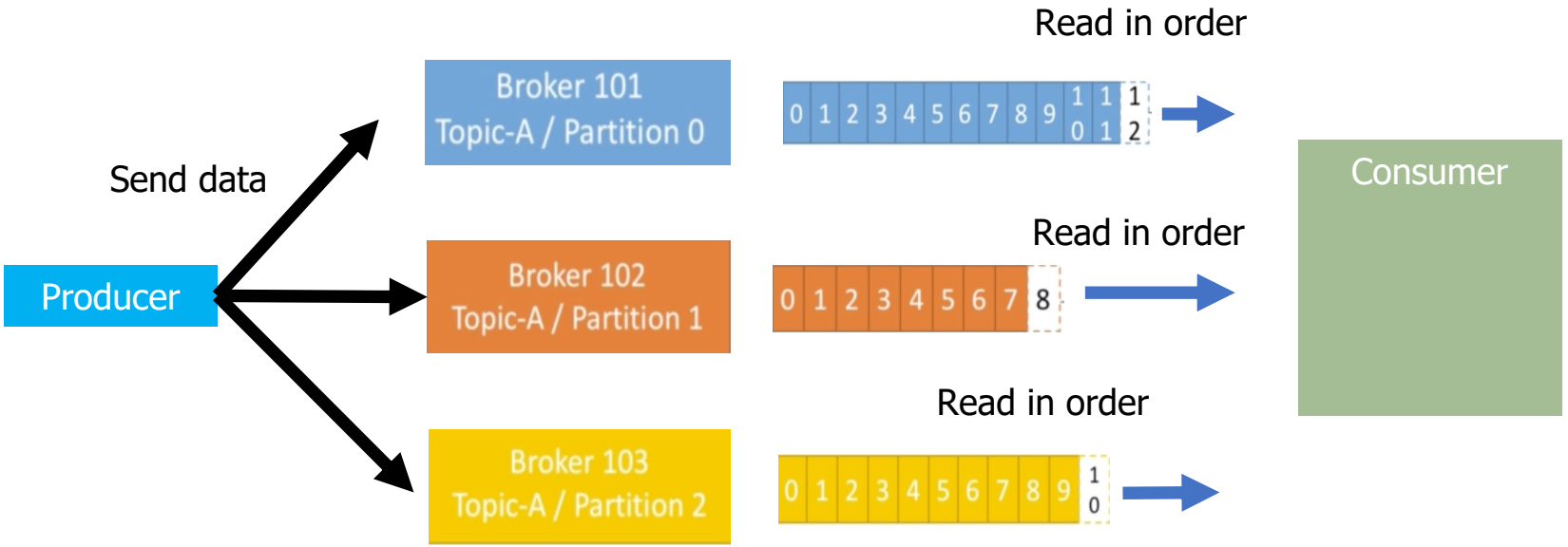
Producer: Message Keys

- Producers can choose to send a key with the message (string, number, etc.)
  - If key = null, i.e., not specified, then the data is sent round robin
- If a key is sent, then all the messages for that key will always go to the same partition
- A key is sent if message ordering for a specific topic is needed
- Exact assignment of key to broker mapping is determined through key hashing



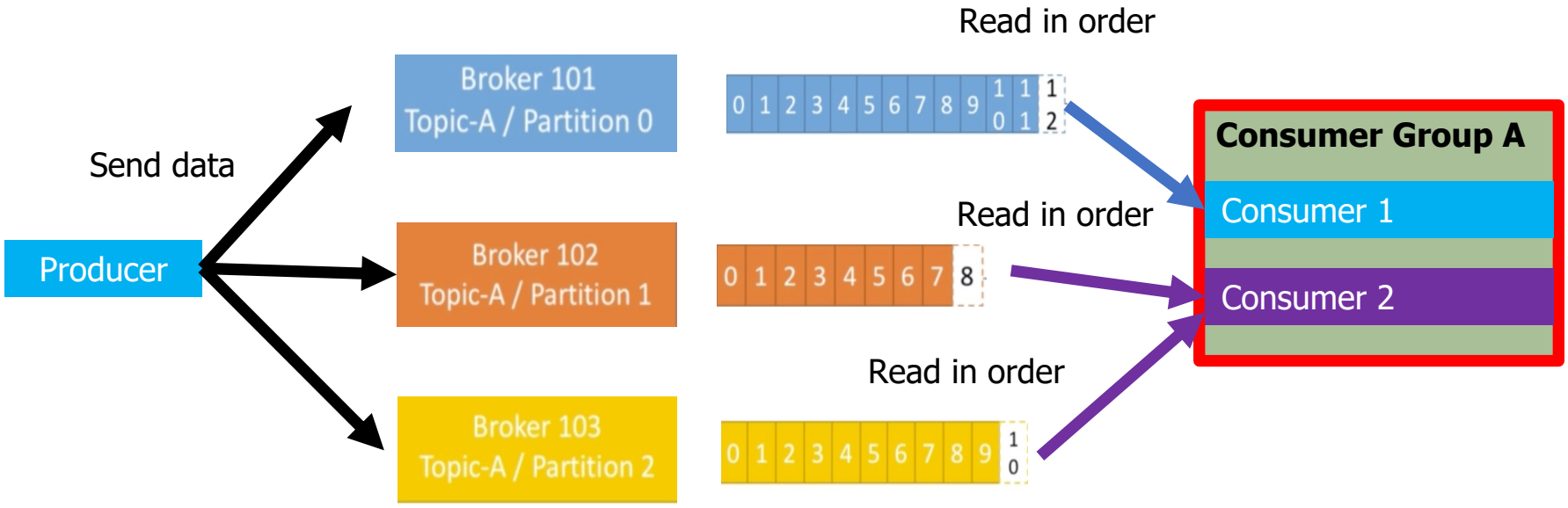
Consumers

- Consumers read data from a topic (identified by name)
- Consumers know which broker to read from
- In case of broker failure, consumers know how to recover
- Data is read in order within each partition



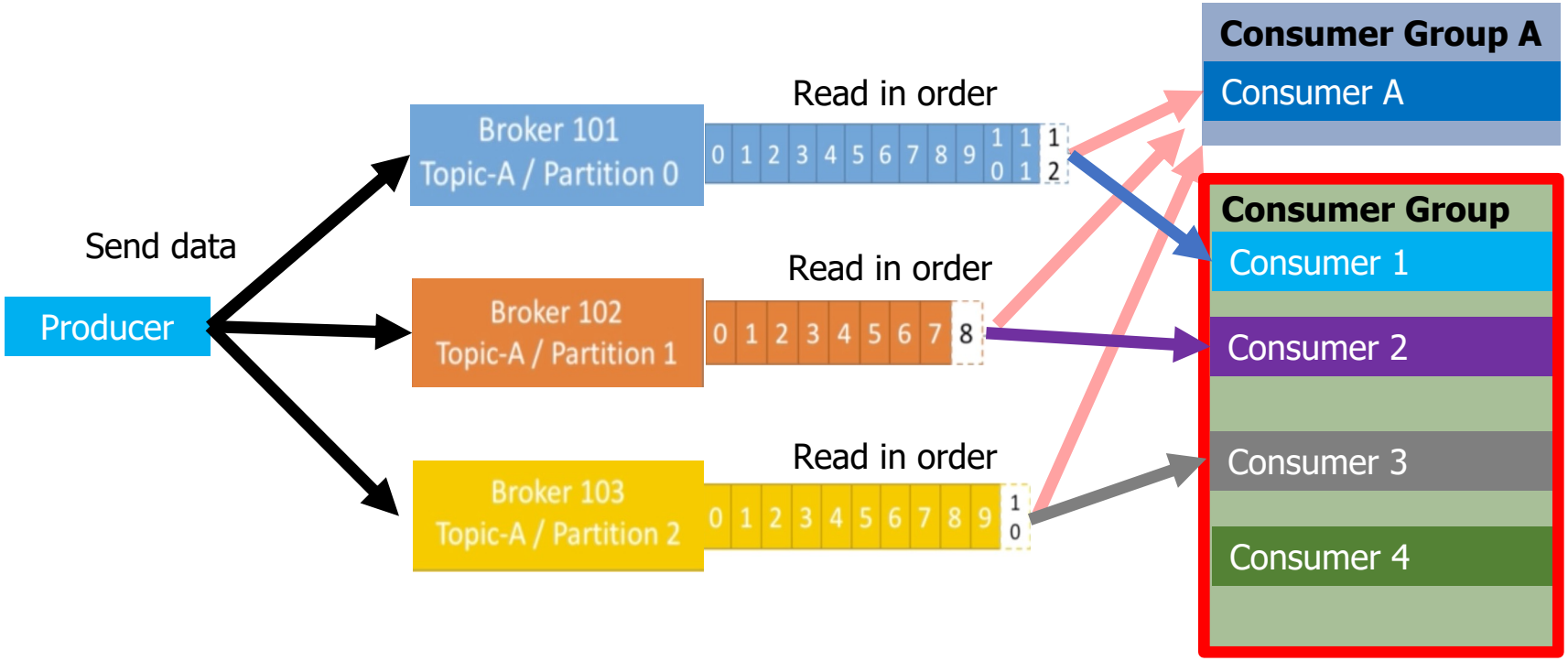
Consumer Groups

- Consumers read data in consumer groups
- Each consumer within a group reads data from exclusive partitions
- Consumers will use GroupCoordinator and ConsumerCoordinator to assign a consumer to a partition



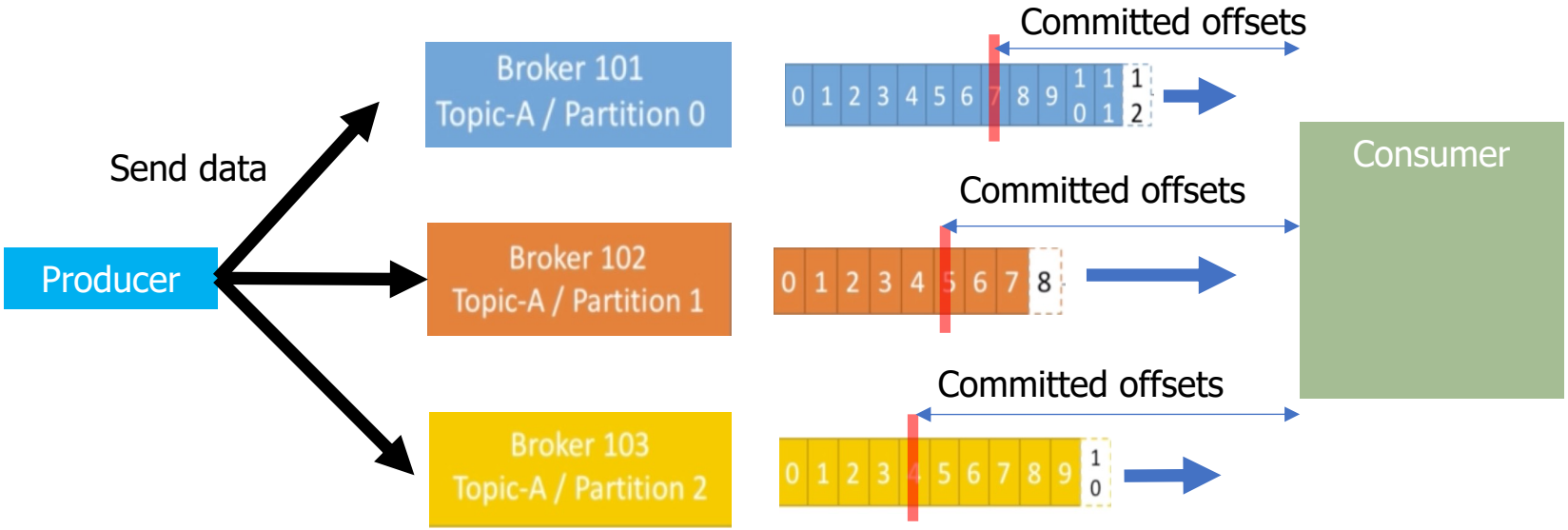
Consumer Groups (continued)

- If a consumer group has only one customer, it will have to get the data from all the brokers (see Consumer A)
- If you have more consumers than partitions, some consumers will be inactive (see Consumer 4)



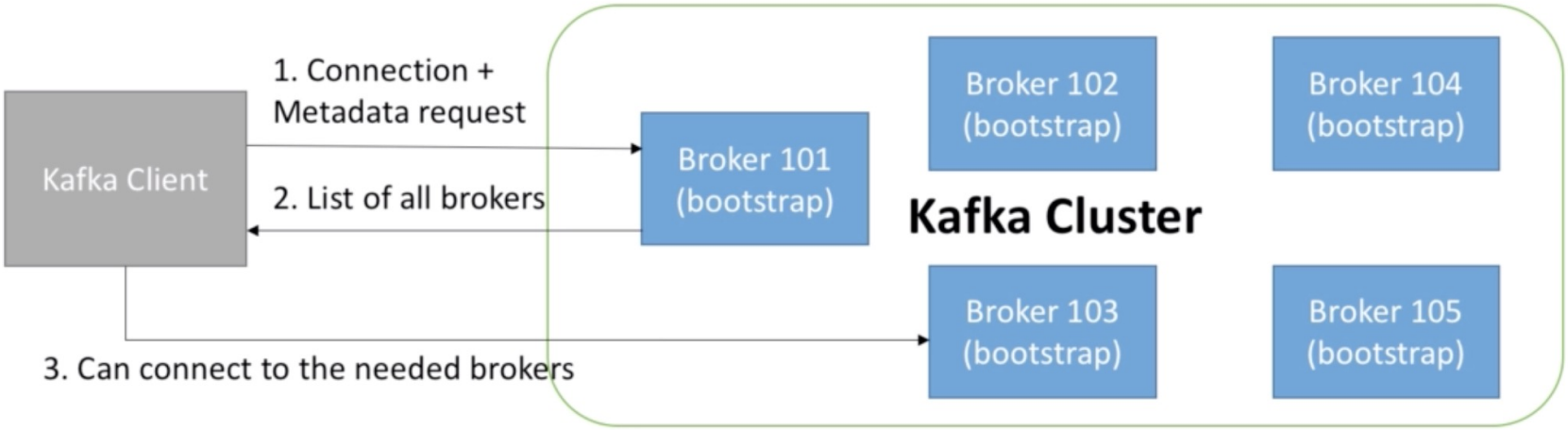
Consumer Offsets

- Kafka stores the offsets at which a consumer group has been reading
  - The offset acts as a bookmark
- The offsets are committed live in a Kafka topic named `___consumer_offsets`
- When a consumer in a group has processed data received from Kafka, it should be committing the offsets
- If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets



Kaka Broker Discovery

- Every Kafka broker is also called a "bootstrap server"
- That means that you need to connect to one broker and you will be connected to the entire cluster
- Each broker knows about all the broker topics and partitions



Kafka Guarantees

- Messages are appended to a topic-partition in the order they are sent
- Consumers read messages in the order stored in a topic-partition
- With a replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- If the number of partitions remains constant for a topic (no new partitions), the same key will always go to the same partition

Delivery Semantics for Consumers

- Consumers choose when to commit offsets
- There are three delivery semantics

1. At most once: Offsets are committed as soon as the message is received
   - Implications:
     - If the processing goes wrong, the message will be lost and will not be read again
2. At least once (preferred choice): Offsets are committed only after the message is processed
   - Implications:
     - If the processing goes wrong, the message will be read again
     - This can result in duplicate processing of messages
       - Make sure that your processing is idempotent (i.e., processing the messages again will not impact the systems)
3. Exactly Once
   - Not easily achievable

CHAPTER 3:
KAFKA SERIALIZATION WITH AVRO

Apache Avro

- Apache Avro is a data serialization system
- Avro provides:
  - Rich data structures
  - A compact, fast, binary data format
  - A container file to store persistent data
  - Remote procedure call (RPC)
  - Avro relies on *schemas*, which are defined with JSON

Serializing Using Apache Avro

- Apache Avro is a language-neutral data serialization format
  - The project was created by Doug Cutting to provide a way to share data files with a large audience

What Is Avro?

- Avro is defined by schema
- Schema is written in JSON
- Avro can be considered as JSON with a schema attached to it

Avro — Advantages and Challenges

Advantages:

- Data is fully typed
- Data is compressed automatically
- Schema comes with the data
- Documentation is embedded in the schema
- Data can be read across any language
- Schema can evolve over time, in a safe manner (schema evolution)

Disadvantages:

- Support for many languages is lacking
- Because it is compressed and serialized, it cannot be printed or directly viewed without using some tools

Avro Data Types

- **Primitive Types**

  null, boolean, int, long, float, double, bytes, and string

- **Complex Types**

  Arrays

  Record

  Enums

Avro Schema

- Avro data is described in a language-independent schema
  - The schema is usually described in JSON and the serialization is usually to binary files, although serializing to JSON is also supported
- Avro assumes that the schema is present when reading and writing files, usually by embedding the schema in the files themselves

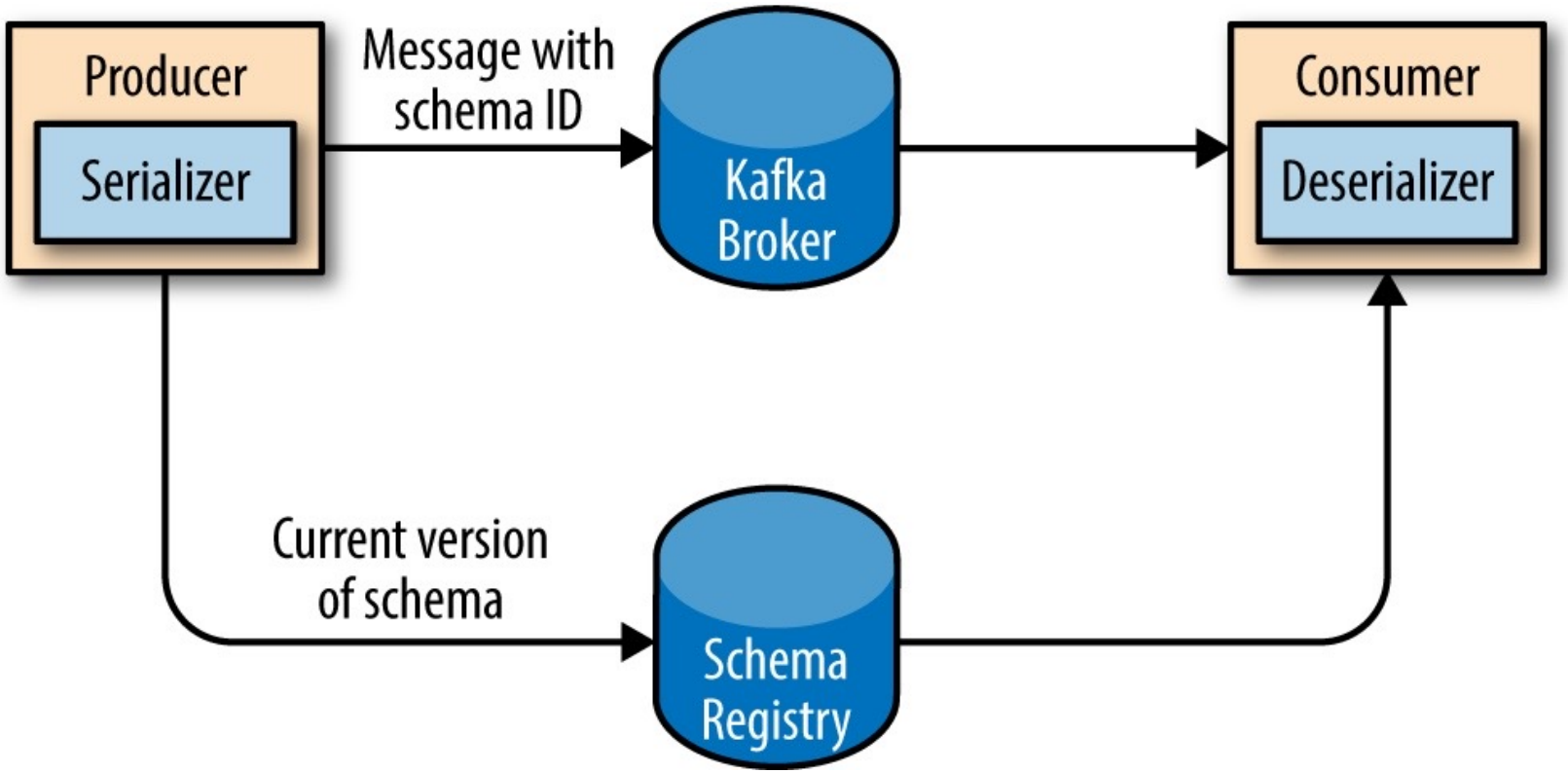
```
{"namespace": "customerManagement.avro",
  "type": "record",
 "name": "Customer",
 "fields": [
       {"name": "id", "type": "int"},
       {"name": "name", "type": "string""},
       {"name": "faxNumber", "type": ["null", "string"], "default": "null"}
  ]
}
```

How to Produce Avro Objects to Kafka

```
Producer<String, Customer> producer = new KafkaProducer<String,
 Customer>(props);
// We keep producing new events until someone ctrl-c
while (true) {
    Customer customer = CustomerGenerator.getNext();
    System.out.println("Generated customer "+ customer.toString());
 ProducerRecord<String, Customer> record =
 new ProducerRecord<>(topic, customer.getId(), customer);
 producer.send(record); }
```

- Customer is our generated object
  - We tell the producer that our records will contain `Customer` as the value
- We also instantiate `ProducerRecord` with `Customer` as the value type, and pass a `Customer` object when creating the new record
- That's it—we send the record with our `Customer` object and `KafkaAvroSerializer` will handle the rest

Flow Diagram of Serialization and Deserialization of Avro Records



ROI TRAINING

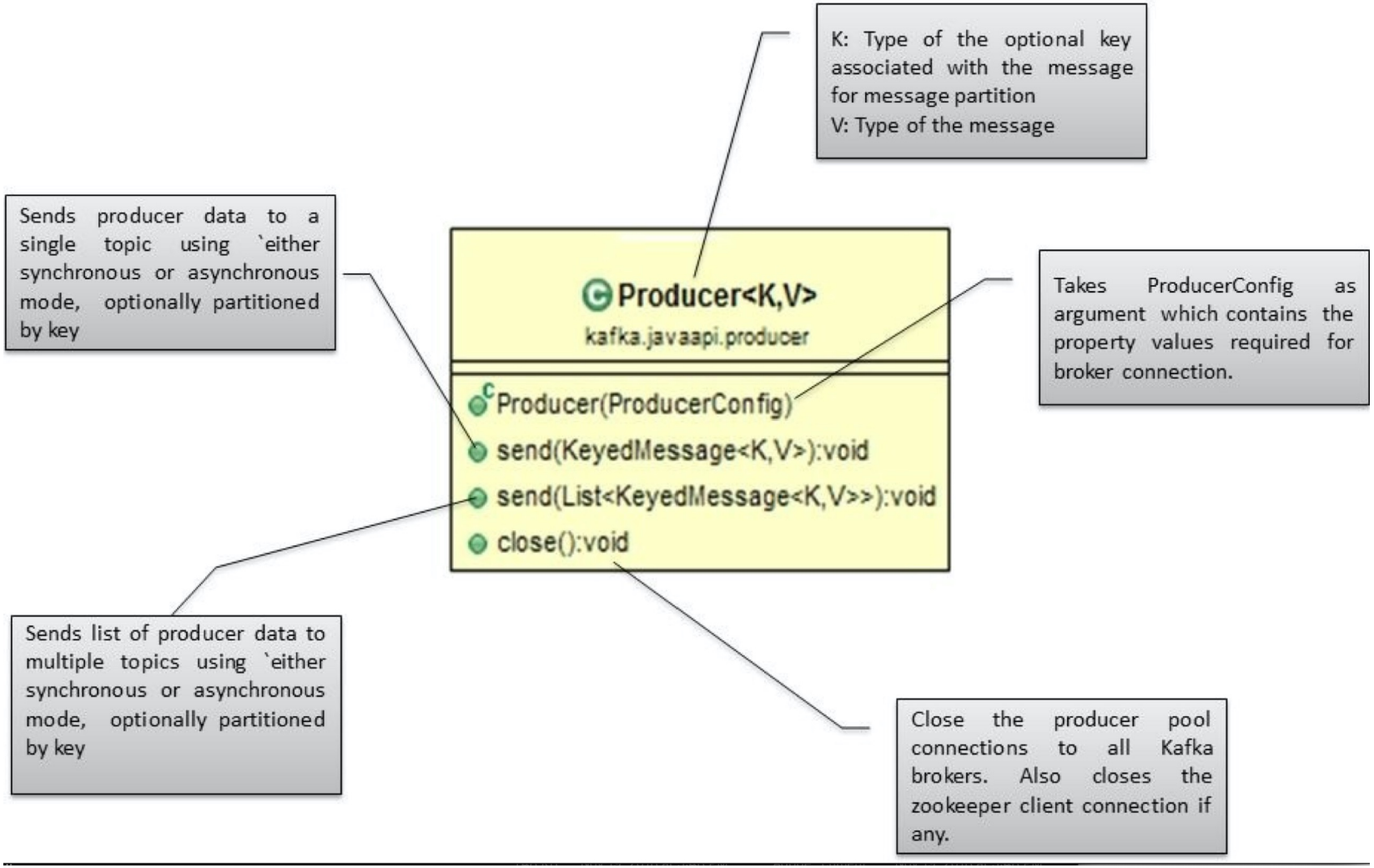
MAXIMIZE YOUR TRAINING INVESTMENT

CHAPTER 4: ADVANCED KAFKA

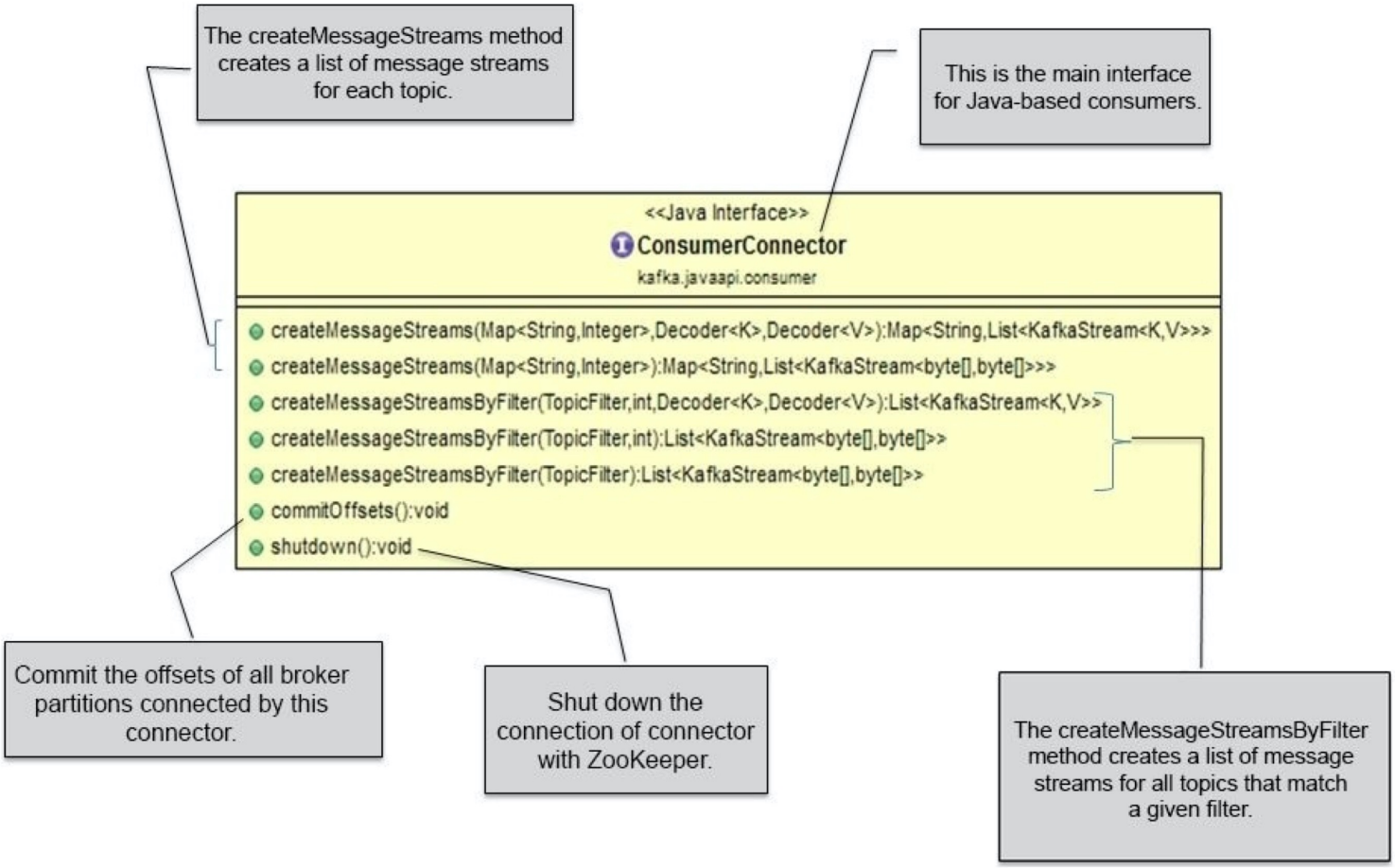
Kafka Core APIs

- Apache Kafka Architecture has four core APIs:
  - Producer API
  - Consumer API
  - Streams API
  - Connector API

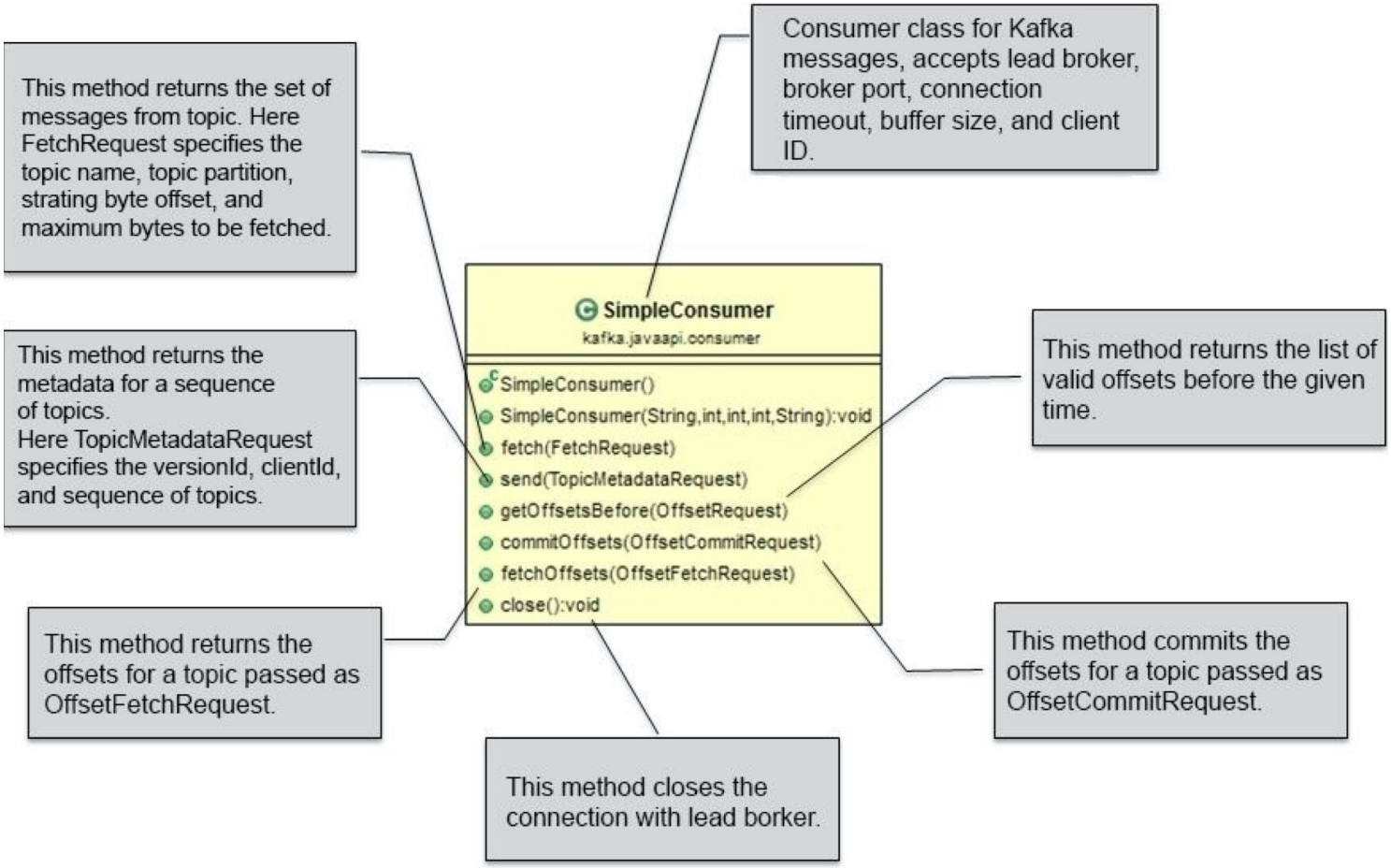
Producer API



ConsumerConnector Interface



Simple Consumer



KafkaProducer API

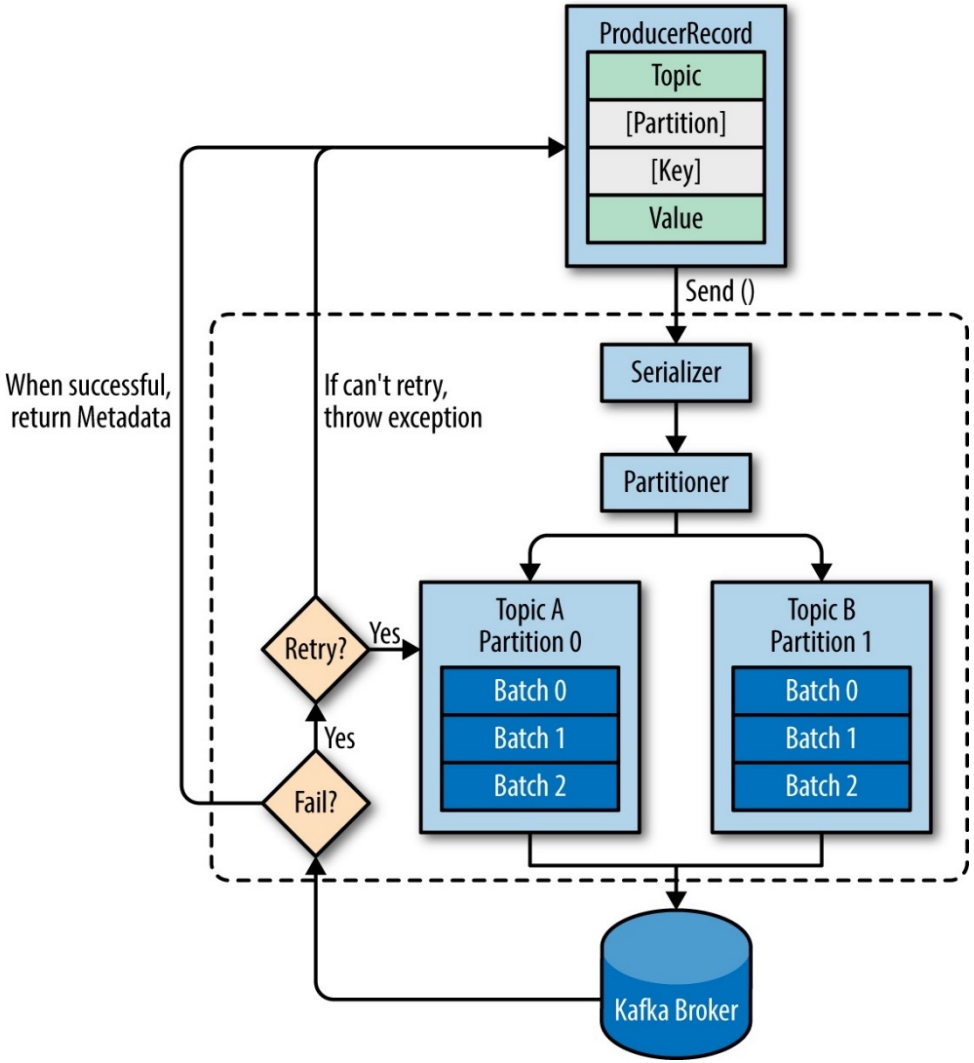
- KafkaProducer class provides send method to send messages asynchronously to a topic
- The signature of `send()` is as follows:

```
producer.send(new ProducerRecord<byte[],byte[]>(topic,
partition, key1, value1) , callback);
```

- **ProducerRecord** – The producer manages a buffer of records waiting to be sent
- **Callback** – A user-supplied callback to execute when the record has been acknowledged by the server (null indicates no callback)
- KafkaProducer class provides a flush method to ensure all previously sent messages have been completed
- Syntax of the flush method is as follows:

```
public void flush()
```

High-Level Overview of KafkaProducer Components



The Producer Class

- The producer class provides send method to send messages to either single or multiple topics using the following signatures

```
public void send(KeyedMessage<k,v> message)
```

- Sends the data to a single topic, partitioned by key using either sync or async producer:

```
public void send(List<KeyedMessage<k,v>>messages)
```

- Sends data to multiple topics:

```
Properties prop = new Properties();
prop.put(producer.type,"async")
ProducerConfig config = new ProducerConfig(prop);
```

The Producer Class (continued)

- There are two types of producers: sync and async
- The difference between them is a sync producer sends messages directly, but sends messages in background
  - Async producer is preferred when you want a higher throughput
- In the previous releases like 0.8, an async producer does not have a callback for `send()` to register error handlers
  - This is available only in the later versions of Kafka

Creating a Simple Producer (Step 1)

- First, you need to create the producer config object with the properties

```
properties properties = new Properties();
properties.put("metadata.broker.list",
"127.0.0.1:9092");
properties.put("serializer.class","kafka.serializer.Str
ingEncoder");
properties.put("request.required.acks", "1");
```

Creating a Simple Producer (Step 2)

- Next, we create the producer object with the settings we just provided

```
KafkaProducer<Integer, String> producer = new
KafkaProducer<Integer, String>(properties);
```

Creating a Simple Producer (Step 3)

- Once we have the producer object ready, we can start preparing a message to be pushed to the Kafka topic

```
ProducerRecord<Integer, String> record = new
ProducerRecord<Integer, String>("mytesttopic", message);
```

Creating a Simple Producer (Step 4)

- After you have the message ready to be sent, you have to call the send method on producer and pass the message as a parameter

```
producer.send(record);
```

Creating a Simple Producer (Step 5)

- After you have sent all the messages, remember to close producer by calling the close method

```
producer.close();
```

Creating a Simple Consumer (Step 1)

- First, find the partition information for the topic from the seed broker address that you have:

```
SimpleConsumer consumer = new SimpleConsumer(seed,port,100000,
64 * 1024, "leaderLookup");
List<String> topics = Collections.singletonList(topic);
TopicMetadataRequest req = new TopicMetadataRequest(topics);
TopicMetadataResponse resp = consumer.send(req);
List<TopicMetadata> metaData = resp.topicsMetadata();
for (TopicMetadata item : metaData) {
   for (PartitionMetadata part : item.partitionsMetadata()) {
     if (part.partitionId() == partition) {
      returnMetaData = part;
    }
   }
 }
```

Creating a Simple Consumer (Step 2)

- Once you have the metadata for the partition, you must create the `SimpleConsumer` class object
  - You need to pass the lead broker host, its port number, timeout, buffer size, and client name as parameters from which you create the `SimpleConsumer` object:

```
SimpleConsumer consumer = new SimpleConsumer(leadBroker,
port, 100000, 64 * 1024, clientName);
```

Creating a Simple Consumer (Step 3)

- Next with this simple consumer, you must get the offset for the topic
  - Depending on your need, you can start with the earliest or the latest offset
    - If you already have a saved offset, you can skip this step:

```
TopicAndPartition topicAndPartition = new TopicAndPartition(topic,
partition);
Map<TopicAndPartition, PartitionOffsetRequestInfo> requestInfo = new
HashMap<TopicAndPartition, PartitionOffsetRequestInfo>();
requestInfo.put(topicAndPartition, new
PartitionOffsetRequestInfo(whichTime, 1));
kafka.javaapi.OffsetRequest request = new kafka.javaapi.OffsetRequest(
 requestInfo, kafka.api.OffsetRequest.CurrentVersion(),
clientName);
OffsetResponse response = consumer.getOffsetsBefore(request);
if (response.hasError()) {
 System.out.println("Error fetching data Offset Data the Broker.
Reason: " + response.errorCode(topic, partition));
 return 0;
}
long[] offsets = response.offsets(topic, partition);
```

Creating a Simple Consumer (Step 4)

- Next, we create a fetch request object with the topic, partition, offset, and the number of bytes to be fetched in one request:

```
FetchRequest req = new FetchRequestBuilder()
.clientId(clientName)
.addFetch(topic, partition, readOffset, 100000)
.build();
```

Creating a Simple Consumer (Step 5)

- Now, we fetch the request from the broker using the following code:

```
FetchResponse fetchResponse = consumer.fetch(req);
```

Creating a Simple Consumer (Step 6)

- With the preceding code, the `fetchResponse` object is populated with the messages from Kafka
  - We should now iterate through it to get the messages based on topic and partition ID
  - Also update the read offset

```
for (MessageAndOffset messageAndOffset :
fetchResponse.messageSet(topic, partition)) {
long currentOffset = messageAndOffset.offset();
 if (currentOffset < readOffset) {
 System.out.println("Found an old offset: " + currentOffset + "
Expecting: " + readOffset);
 continue;
}
readOffset = messageAndOffset.nextOffset();
ByteBuffer payload = messageAndOffset.message().payload();
 byte[] bytes = new byte[payload.limit()];
 payload.get(bytes);
 System.out.println(String.valueOf(messageAndOffset.offset()) + ": " +
new String(bytes, "UTF-8"));
 numRead++;
}
```

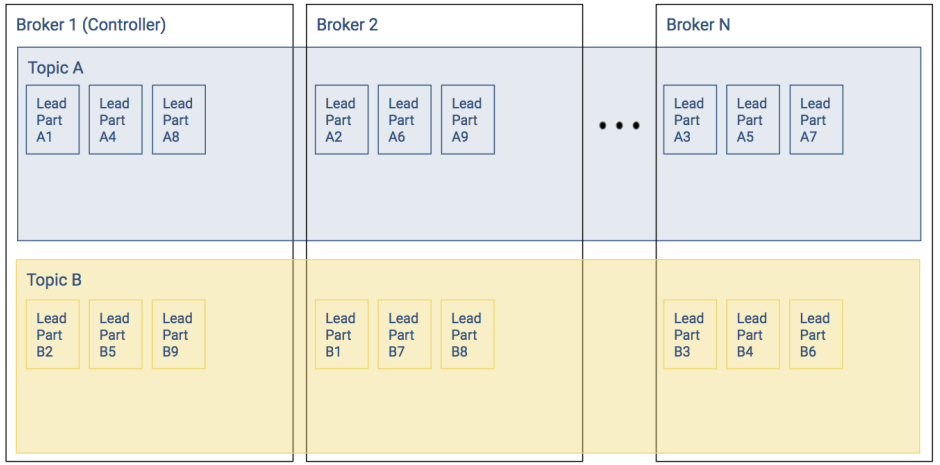
Creating a Simple Consumer (Step 7)

- The final step is to close the connection by calling close on the consumer object

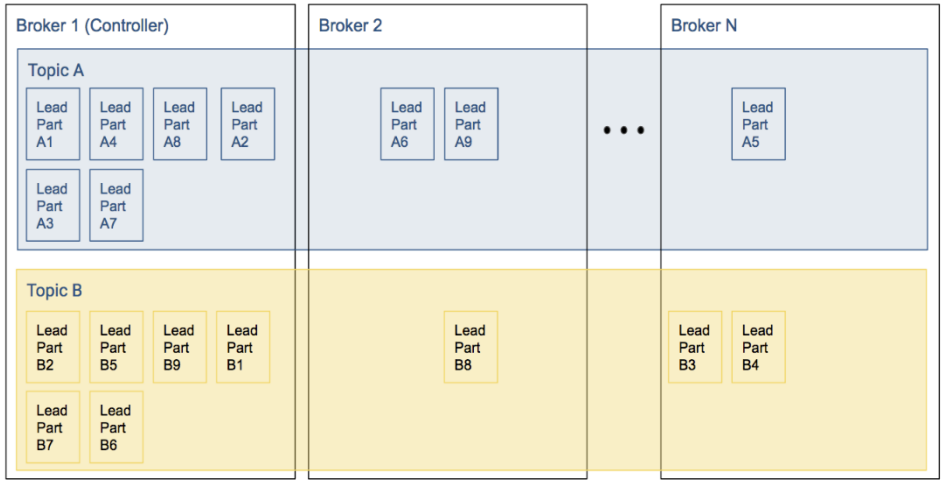
```
consumer.close();
```

Kafka Cluster in Steady State vs. Imbalance

- In the first figure, the Kafka cluster has well balanced leader partitions
  - Recall the following:
    - Producer writes and consumer reads occur at the partition level
    - Leader partitions are responsible for ensuring that the follower partitions keep their records in sync



- In the second figure, since a large chunk of the leaders for Topic A and Topic B are on Broker 1, a lot more of the overall Kafka workload will occur at Broker 1
  - This will cause a backlog of work, which slows down the cluster throughput, which will worsen the backlog



Stream Processing and Kafka

- Kafka was traditionally seen as a powerful message bus, capable of delivering streams of events but without processing or transformation capabilities
- External stream processing systems, such as Apache Spark and Apache Storm, were used for stream processing and Kafka as a service bus
- Kafka 0.10.0 introduced a powerful stream-processing library as part of its collection of client libraries
  - This allows developers to consume, process, and produce events in their own apps, without relying on an external processing framework

Kafka Streams API (KTables and KStreams)

- Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters
  - It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology
- A major new feature in Apache Kafka® v0.10: Kafka's Streams API
  - The Streams API, available as a Java library that is part of the official Kafka project, is the easiest way to write mission-critical real-time applications and microservices with all the benefits of Kafka's server-side cluster technology

Kafka Streams API (KTables and KStreams) (continued)

- What is the Kafka Streams API?
  - The Kafka Streams API allows you to create real-time applications that power your core business
    - It is the easiest to use, yet the most powerful technology to process data stored in Kafka
    - It gives us the implementation of standard classes of Kafka
  - A unique feature of the Kafka Streams API is that the applications you build with it are normal applications
    - These applications can be packaged, deployed, and monitored like any other application, with no need to install separate processing clusters or similar special-purpose and expensive infrastructure!
- In most cases, however, the level of detail provided by the Processor API is not required and the KStream API will get the job done
  - Compared to the declarative approach of the Processor API, KStreams uses a more functional style

Kafka Streams API (KTables and KStreams) (continued)

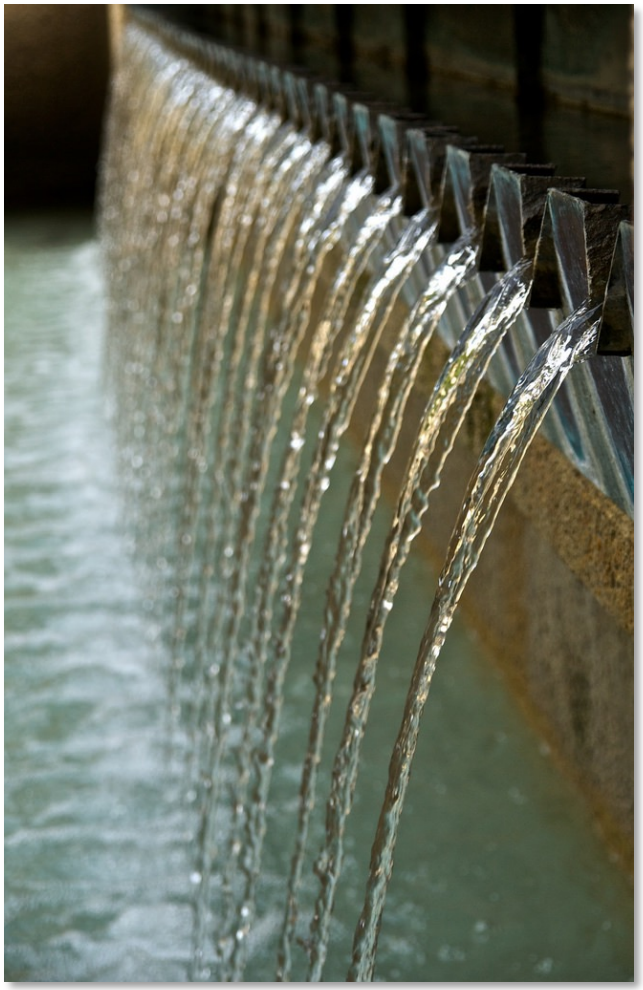
- The KStreams DSL is composed of two main abstractions; the KStream and KTable interfaces
  - A KStream is a record stream where each key-value pair is an independent record
- A KTable, on the other hand, is a "changelog" stream, meaning later records are considered updates to earlier records with the same key

Kafka and SerDes

- Kafka depends on Serializer/Deserializer (SerDes) so that producer and consumer both know how to communicate and understand the messages
- Every Kafka Streams application must provide SerDes for the data types of record *keys* and record *values* to materialize the data when necessary
- SerDes can be provided by using either of these methods:
  - Method 1: By **setting default SerDes** through a `StreamsConfig` instance
  - Method 2: By **specifying explicit SerDes** when calling the appropriate API methods, thus overriding the defaults

Kafka Streaming APIs

- Several stream-processing frameworks, including Google's Dataflow and Kafka Streams, have built-in support for the notion of event time independent of the processing time and the ability to handle events with event times that are older or newer than the current processing time
- The Kafka's Streams API always writes aggregation results to result topics
  - Those are usually compacted topics, which means that only the latest value for each key is preserved



Kafka Streaming APIs (continued)

- Apache Kafka has two streams APIs

1. Low-level Processor API:
   - Allows you to create your own transformations
   - This is rarely required
2. High-level DSL API
   - Allows you to define the stream-processing application by defining a chain of transformations to events in the streams
   - Transformations can be as simple as a filter or as complex as a stream-to-stream join

Stream Processing

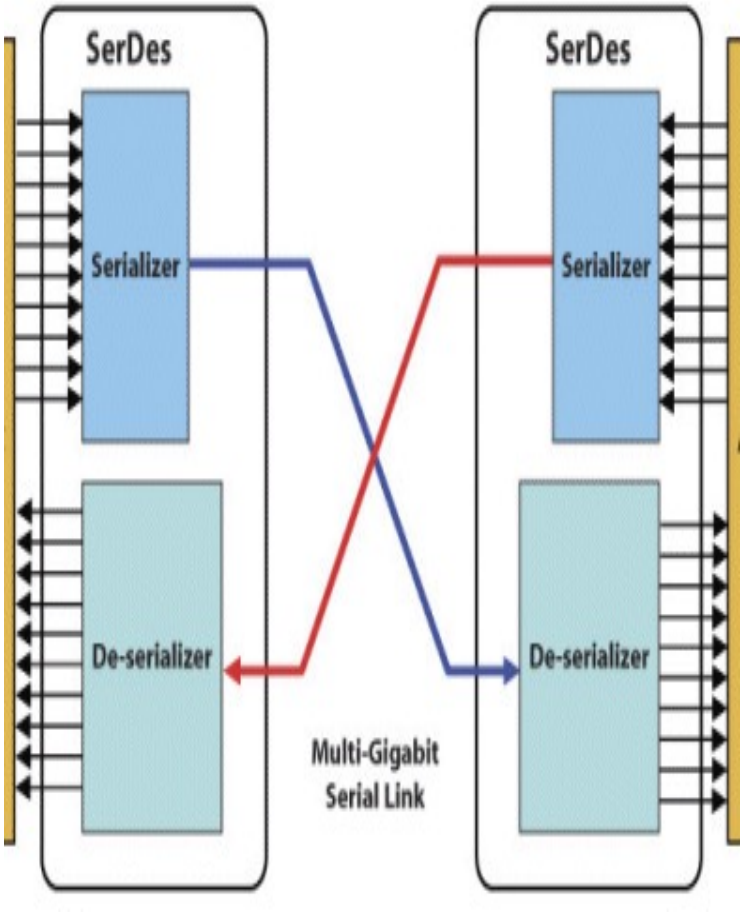
- A data stream is an abstraction representing an unbounded dataset
  - Unbounded means infinite and ever-growing data as new records keep arriving
- Streaming data attributes
  - Event in the data streams are ordered
  - Immutable data records
  - Unboundedness
- Stream-processing systems refer to the following notions of time:
  - **Event Time:** The time when the event actually happened
  - **Log Append Time:** This is the time the event arrived at the Kafka broker and was stored there
  - **Processing Time:** This is the time at which a stream-processing application received the event in order to perform some calculation
- When working with time, it is important to be mindful of *time zones*
  - The entire data pipeline should standardize on a single time zones; otherwise, results of stream operations will be confusing and often meaningless

Time Windows

- Most operations on streams are windowed operations—operating on slices of time
  - For example: moving averages, top products sold this week, 99th percentile load on the system
- Join operations on two streams are also windowed—we join events that occurred at the same slice of time

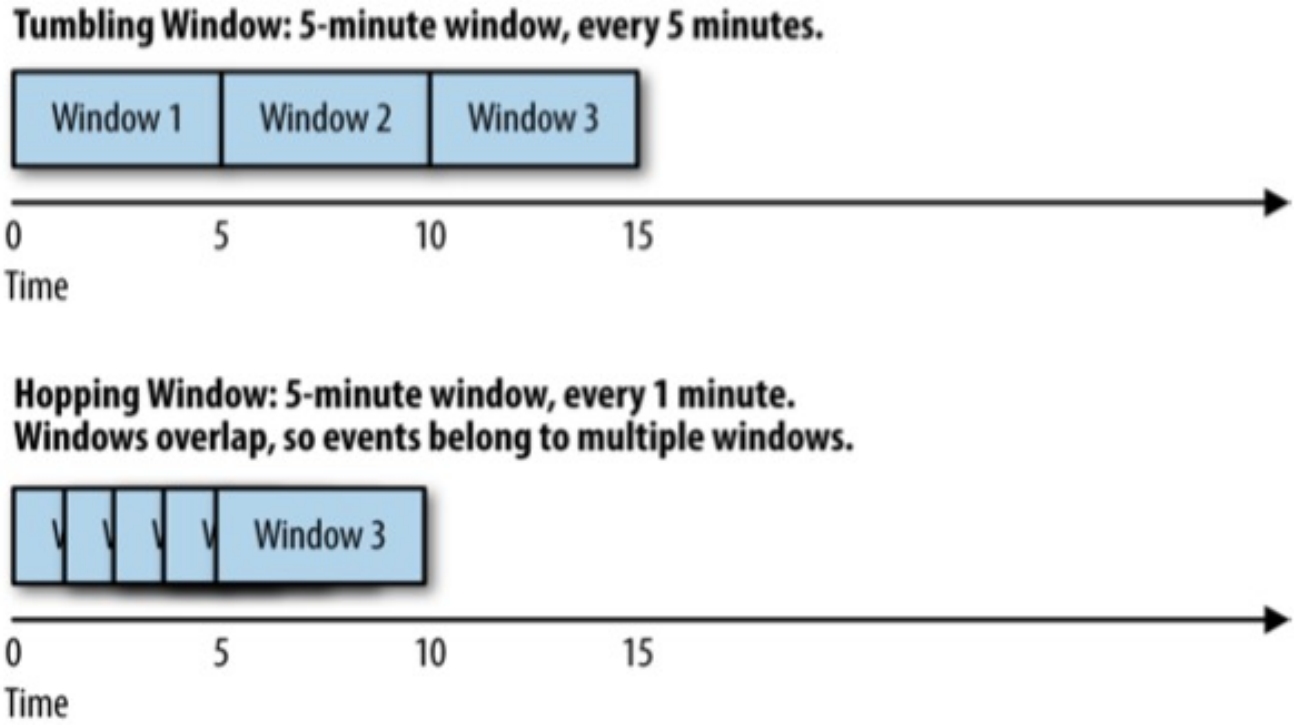
Serializers and Deserializers (SerDes)

- Serialization is the process of turning an object state or data structure in memory into a stream of bytes so that it can be sent over the network
- Deserialization is the reverse process: turning a stream of bytes into an object or a data structure in memory
- The process of serializing an object is also called marshalling an object or a data structure
  - The opposite is called unmarshalling



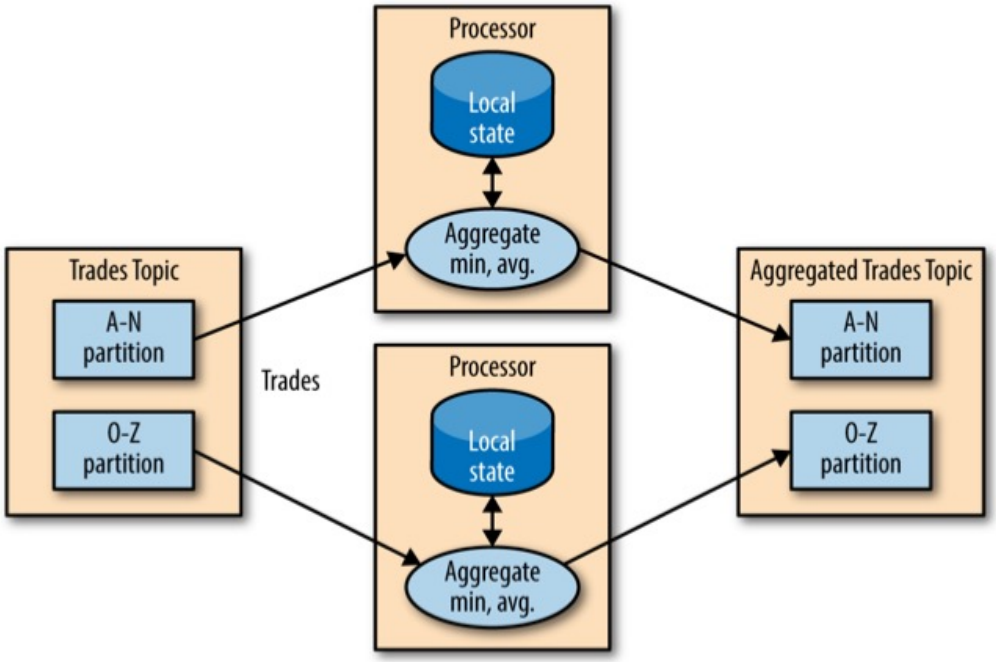
Time Windows: Tumbling vs. Hopping

- What is the size of the window?
- How often does the window move?
- How long does the window remain updatable?



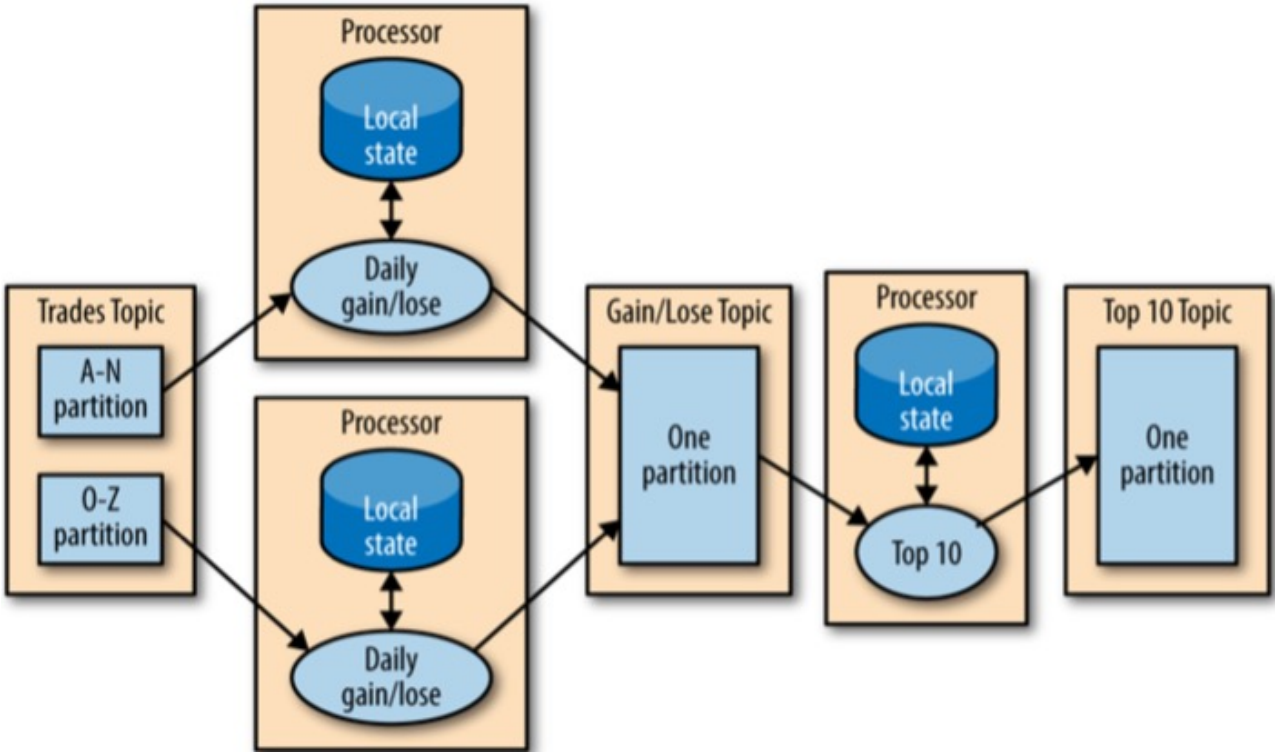
Stream-Processing Design Patterns (Mappers)

- Suppose we want to calculate the min and average of each individual stock in the last on hour
- The most basic pattern of stream processing is the processing of each event in isolation
- The stream-processing app consumes events from the stream, modifies each event, and then produces the event to another stream
- Similar to Mappers in MapReduce or narrow transformations in Spark



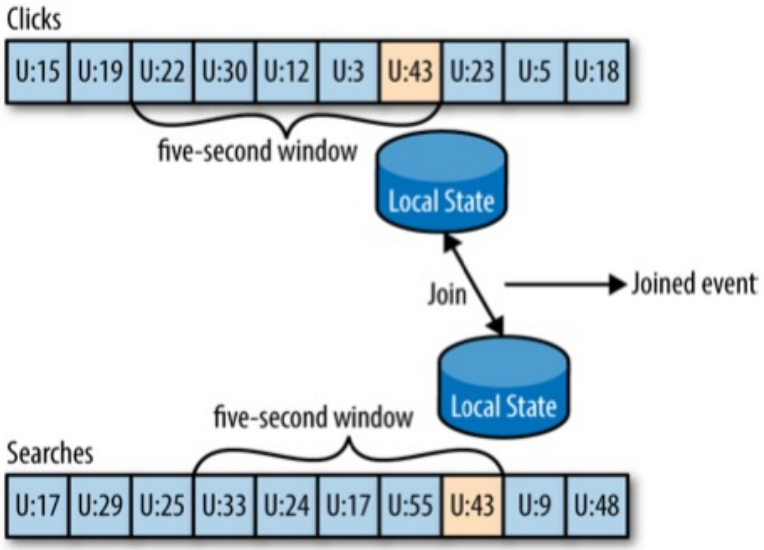
Stream-Processing Design Patterns (Reducers)

- Suppose we want to publish the top 10 stocks each day
- We aggregate overall trends that need data from all the partitions
- It is similar to reducing phase in MapReduce or Wide-Transformation in Spark



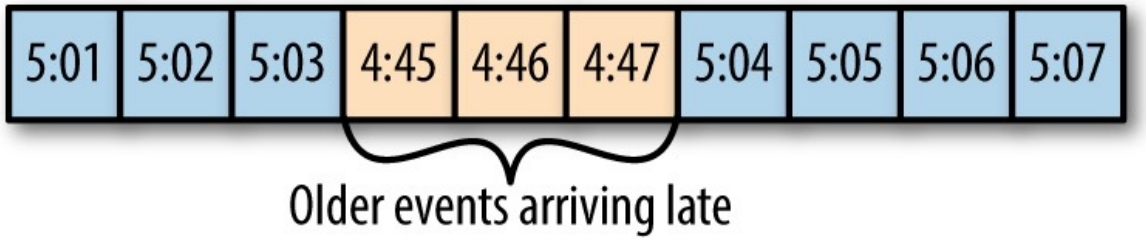
Streaming Joins

- Real-time streams can be joined using windowed-join
- Joining two streams of events; these joins always involve a moving time window
- Example:
  - Stream 1: Search queries that people entered into our website
  - Stream 2: Clicks on search results
  - Objective: We want to match search queries with the results they clicked so that we will know which result is most popular for which query



Out-of-Sequence Events

- Handling out-of-sequence events is a challenge
- Out-of-sequence events happen quite frequently and expectedly in IoT (Internet of Things) scenarios
- Desired characteristics of streaming processing engine
  - Ability to recognize that an event is out of sequence
  - Ability to define a time period during which it will attempt to reconcile out-of-sequence events



Rebalancing of Consumers

- The goal of rebalancing is to ensure that all partitions are equally consumed
  - Equally means here that there is only one consumer linked to one partition
- It's important to stress that the rebalancing applies only to consumers belonging to the same group
  - Thanks to that, Kafka clients can easily handle two messaging approaches: queue (several consumers per group) and publish-subscribe (one consumer per group)
- Rebalancing is done by a special broker called coordinator
  - Each consumers group has its own coordinator
  - One broker can coordinate several different groups
  - It occurs when some changes are detected concerning topics organization (new partitions added) or consumers (consumers joining or leaving group)

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

CHAPTER 5: MONITORING KAFKA

Introduction

- A well-defined monitoring strategy is critical for the proper function of Kafka in production environment
- The objective of monitoring is to identify bottleneck in the system and keep an eye on the reliability of the infrastructure
- Monitoring Kafka involves checking for throughput and latencies
- Monitoring involves simple metrics about the overall rate of traffic, to detailed timing metrics for every request type, to per-topic and per-partition metrics

Monitoring Server Stats

- Kafka exposes various stats for monitoring using Yammer Metrics
- For optimal running of the cluster, it is important to monitor the various metrics exposed by Kafka from the server side
- You need to have the Kafka server up and running with the JMX port
  - To set the JMX port, you need to run Kafka using the following command:

```
JMX_PORT=10101 ./bin/kafka-server-start.sh
config/server.properties
```

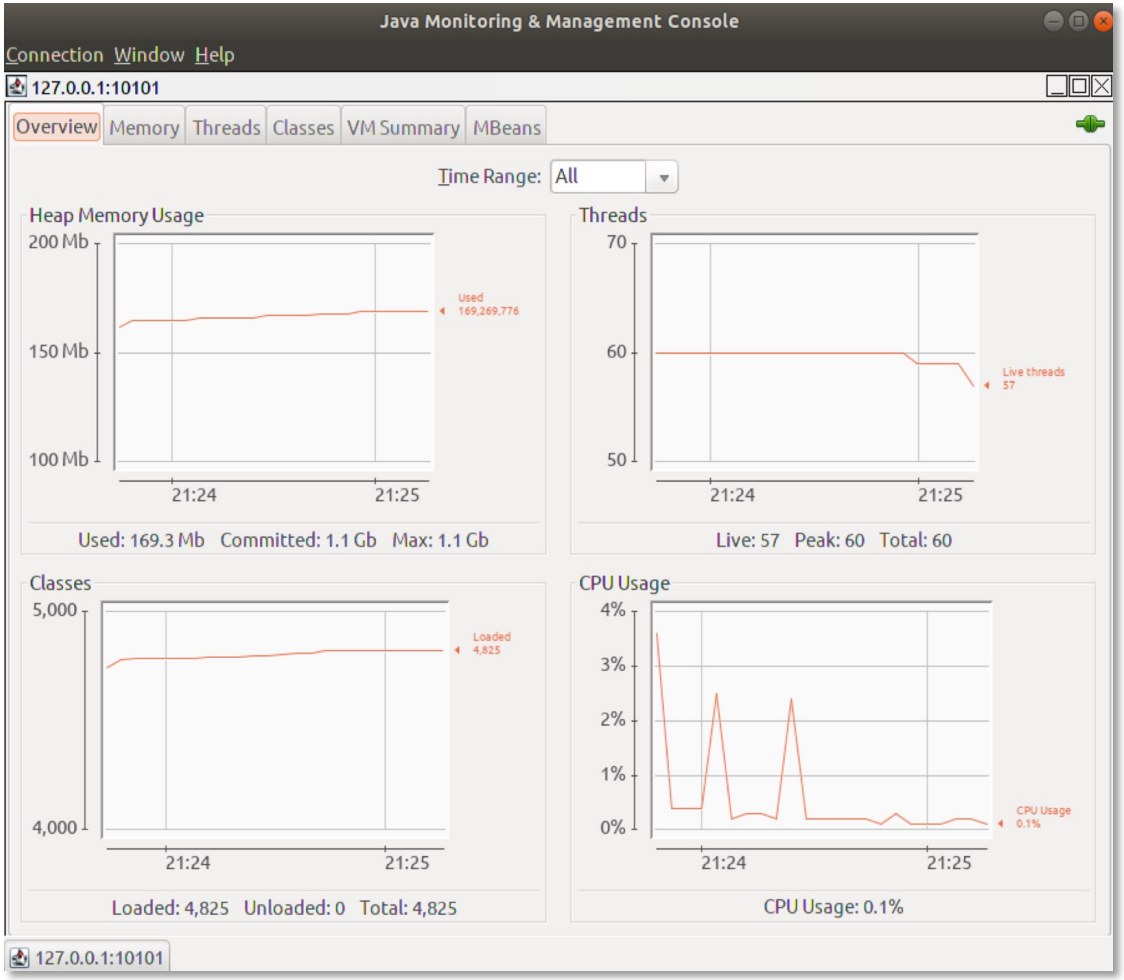
- From your command prompt, you need to run `jconsole` using the following command

```
jconsole 127.0.0.1:10101
```

JConsole

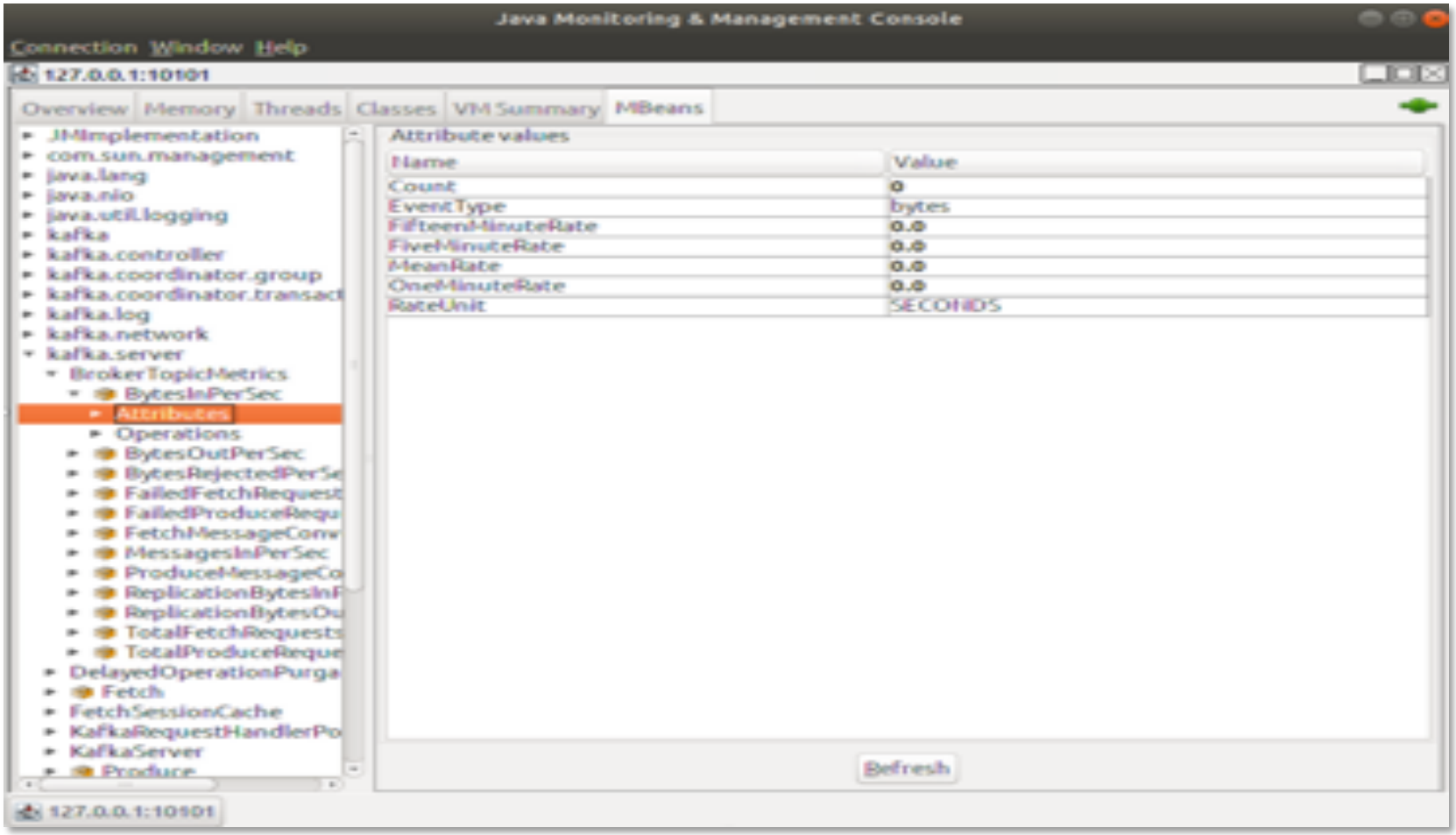
- JConsole is the application that connects to the JMX port exposed by Kafka
  - You can read all the metrics from Kafka using JConsole
- From your command prompt, you need to run `jconsole` using the following command:

```
jconsole 127.0.0.1:10101
```



MBeans Tab

- **MBeans** tab gives the details of the various Kafka metrics



Details of the Metrics with the MBean

`kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec:`

- This gives the number of messages being inserted in Kafka per second
  - This has the attribute values given out as counts; one-minute rate, five-minute rate, fifteen-minute rate, and mean rate

`kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`

- This specifies the number of partitions for which the number of replicas criterion is not met
  - If this value is anything more than zero, it means your cluster has issues replicating the partitions as desired by you

`kafka.controller:type=KafkaController,name=ActiveControllerCount`

- This MBean gives the number of active controllers for Kafka for re-election

`kafka.server:type=ReplicaManager,name=PartitionCount`

- This MBean gives the total number of partitions present in that particular Kafka node

Details of the Metrics with the MBean (continued)

`kafka.server:type=ReplicaManager,name=LeaderCount`

- This MBean gives the total number of leader partitions present in this Kafka node

`kafka.server:type=ReplicaManager,name=IsrShrinksPerSec`

- This MBean specifies the rate at which in-sync replicas shrink
  - It can give these values as a mean, one-minute rate, five-minute rate, and fifteen-minute rate
  - It gives the count of events as well

`kafka.server:type=ReplicaManager,name=IsrExpandsPerSec`

- This MBean specifies the rate at which in-sync replicas expand
  - It can give these values as a mean, one-minute rate, five-minute rate, and fifteen-minute rate
  - It gives the count of events as well

`kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica`

- This MBean specifies the maximum lag between the master and the replicas

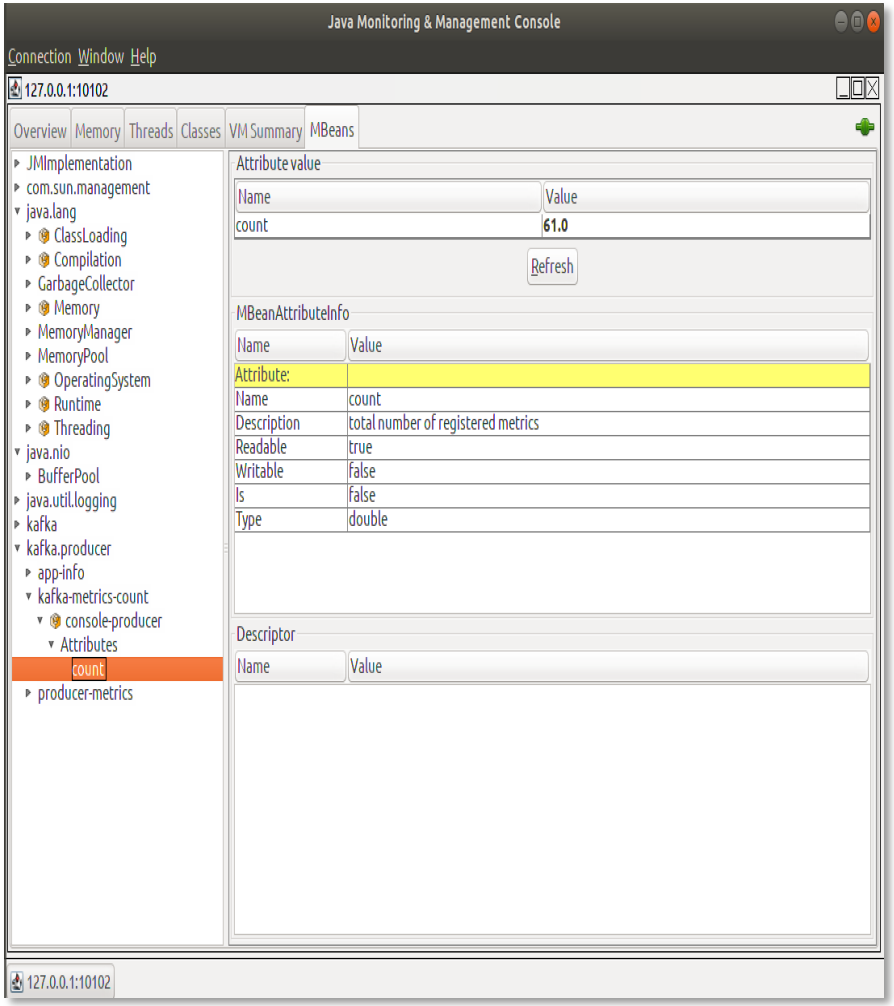
Monitoring Producer Stats

- There are metrics for the producer so that we can track what's happening with it
- Start the console producer with the JMX parameters enabled
  - You can do this as follows:

```
JMX_PORT=10102 bin/kafka-console-producer.sh --broker-list localhost:9092 --topic mytesttopic
```

- You can connect to JMX at port number 10102 on your machine using the `jconsole` application

```
jconsole 127.0.0.1:10102
```



Details of the Tab with MBeans

`kafka.producer:type=ProducerRequestMetrics,name=ProducerRequestRateAndTimeMs, producer:`

- This MBean give values for the rate of producer requests taking place as well as latencies involved in that process
  - It gives latencies as a mean, the 50th, 75th, 95th, 98th, 99th, and 99.9th latency percentiles
  - It also gives the time taken to produce the data as a mean, one-minute average, five-minute average, and fifteen-minute average
  - It gives the count as well

`kafka.producer:type=ProducerRequestMetrics,name=ProducerRequestSize,clientId= producer:`

- This MBean gives the request size for the producer
  - It gives the count, mean, max, min, standard deviation, and the 50th, 75th, 95th, 98th, 99th, and 99.9th percentile of request sizes

Details of the Tab with MBeans (continued)

```
kafka.producer:type=ProducerStats,name=FailedSendsPerSec,
clientId=console- producer:
```

- This gives the number of failed sends per second
  - It gives this value of counts, the mean rate, one-minute average, five-minute average, and fifteen-minute average value for the failed requests per second

```
kafka.producer:type=ProducerStats,name=
SerializationErrorsPerSec,clientId=con producer:
```

- This gives the number of serialization errors per second
  - It gives this value of counts, mean rate, one-minute average, five-minute average, and fifteen-minute average value for the serialization errors per second

```
kafka.producer:type=ProducerTopicMetrics,name=
MessagesPerSec,clientId=console producer:
```

- This gives the number of messages produced per second
  - It gives this value of counts, mean rate, one-minute average, five-minute average, and fifteen-minute average for the messages produced per second

Monitoring Consumer Stats

- There are metrics so that we can track what's happening with the consumer
- Start the console producer with the JMX parameters enabled

```
JMX_PORT=10103 bin/kafka-console-producer.sh --broker-
list localhost:9092 --topic mytesttopic
```

- You can connect to JMX at port number 10102 on your machine using the `jconsole` application

```
jconsole 127.0.0.1:10103
```

Details of the Metrics with MBeans

```
kafka.consumer:type=ConsumerFetcherManager,name=
MaxLag,clientId=test-consumer-group
```

- This gives the number of messages that the consumer is behind by in consuming the messages pushed in by the producer

```
kafka.consumer:type=ConsumerFetcherManager,name=
MinFetchRate,clientId=test-consumer-group:
```

- This gives the minimum rate at which the consumer sends fetch requests to the broker
  - If a consumer is dead, this value becomes close to 0

```
kafka.consumer:type=ConsumerTopicMetrics,name=
BytesPerSec,clientId=test-consumer-group
```

- This gives the number of bytes consumed per second
  - It gives this value of count, mean rate, one-minute average, five-minute average, and fifteen-minute average for the bytes consumed per second

Details of the Metrics with MBeans (continued)

`kafka.consumer:type=FetchRequestAndResponseMetrics, name=FetchRequestRateAndTi consumer-group:`

- This MBean gives values for the rate at which the consumer fetches the requests as well as the latencies involved in that process
  - It gives latencies as a mean, the 50th, 75th, 95th, 98th, 99th, and 99.9th latency percentiles
  - It also gives the time taken to consume the data as a mean, one-minute rate, five-minute rate, and fifteen-minute rate
  - It gives the count as well

`kafka.consumer:type=FetchRequestAndResponseMetrics, name=FetchResponseSize,cli consumer-group:`

- This MBean gives the fetch size for the consumer
  - It gives the count, mean, max, min, standard deviation, 50th, 75th, 95th, 98th, 99th, and 99.9th percentile of request sizes

Details of the Metrics with MBeans (continued)

```
kafka.consumer:type=ZookeeperConsumerConnector,name=Fetch
QueueSize,
clientId=t consumer-group,topic=mytesttopic,threadId=0:
```

- This MBean gives the queue size for the fetch request for the clientID, threaded, and topic mentioned

```
kafka.consumer:type=ZookeeperConsumerConnector,name=Kafka
CommitsPerSec,client consumer-group:
```

- This MBean gives the fetch size for the Kafka commits per second
  - It gives the count, mean, one-minute average, five-minute average, and fifteen-minute average rate of Kafka commits per second

Details of the Metrics with MBeans (continued)

```
kafka.consumer:type=ZookeeperConsumerConnector,name=
RebalanceRateAndTime,client consumer-group:
```

- This MBean gives the latency and rate of rebalance for the consumer
  - It gives latencies as a mean, the 50th, 75th, 95th, 98th, 99th, and 99.9th latency percentiles
  - It also gives the time taken to rebalance as a mean, one-minute rate, five-minute rate, and fifteen-minute average
  - It also gives the count

Metric Basics

- All the metrics exposed by Kafka can be accessed via the Java Management Extensions (JMX) interface
- The easiest way to use them in an external monitoring system is to use a collection agent provided by your monitoring system and attach it to the Kafka process
- To aid with configuring applications that connect to JMX on the Kafka broker directly, such as monitoring systems, the broker sets the configured JMX port in the broker information that is stored in ZooKeeper

Internal or External Measurements

- Metrics provided via an interface such as JMX are internal metrics: they are created and provided by the application that is being monitored
- There are other metrics, such as the overall time for a request or the availability of a particular request type, that can be measured externally
  - This would mean that a Kafka client, or some other third-party application, provides the metrics for the server (the broker, in our case)
  - These are often metrics like availability (is the broker reachable?) or latency (how long does a request take?)
  - These provide an external view of the application that is often more informative

Application Health Checks

- This can be done in two ways:
  - An external process that reports whether the broker is up or down (health check)
  - Alerting on the lack of metrics being reported by the Kafka broker (sometimes called *stale metrics*)
- Though the second method works, it can make it difficult to differentiate between a failure of the Kafka broker and a failure of the monitoring system itself

Kafka Broker Metrics

- Under-replicated partitions
  - This measurement, provided on each broker in a cluster, gives a count of the number of partitions for which the broker is the leader replica, where the follower replicas are not caught up
  - This single measurement provides insight into several problems with the Kafka cluster, from a broker being down to resource exhaustion

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

CHAPTER 6: KAFKA STREAMS

Purpose of Streams API

- Writing a consumer for a Kafka topic could be done with the normal Consumer API we've already explored
  - Scaling it becomes difficult as you have more partitions and need more workers
- Spark could also be used to create a consumer and use the familiar Spark API's which are widely used for processing data other than Kafka
  - Requires setting up a Hadoop/Spark cluster
- Kafka Streams API is another alternative that allows you to write Java or Scala consumers more easily and deploy them in a way that is easily horizontally scalable without the need for setting up a separate cluster
  - Uses a declarative vs. imperative programming approach which means less code and more straight forward business logic

Kstreams & StreamBuilder

- Instead of using a normal Consumer object you create a stream builder object
  - It takes similar parameters but how you process the data coming in is different
  - The KStream object acts as a source

```
package io.confluent.examples.streams;
import org.apache.kafka.streams.KafkaStreams;

final Properties streamsConfiguration = new Properties();
streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONF
IG, ”stream1");
final KafkaStreams streams = new
KafkaStreams(builder.build(), streamsConfiguration);

final KStream<byte[], String> textLines =
   builder.stream("TextLinesTopic",
   Consumed.with(byteArraySerde, stringSerde));
```

Transformations

- mapElements is used to like the SELECT in SQL to transform a message to a new shape

```
final KStream<byte[], String> uppercasedWithMapValues =
   textLines.mapValues(v -> v.toUpperCase());
```

- Filter can be used like a WHERE clause in SQL to only keep messages you want

```
filter ((key, value) › Long. parseLong (value) › 1000)
```

KTables

- A stream represents a series of messages that could be on the same key
  - Sometimes it's useful to just get the last value of a message for a key
  - Ktables represent the current value of a key based on the last message for that key
  - You can think of them as similar to a SQL table

```
StreamBuilder builder = new StreamBuilder();
KTable<String, String> firstKTable = builder.
    table(inputTopic,Materialized.with(Serdes.String(),
    Serdes.String()));
```

JOINS

- To enhance the content of a stream it's often useful to match it to some lookup value
  - Could be from a KStream or another KTable

```
KStream<String, String> leftStream = builder.stream
    ("topic-A");
KStream<String, String> rightStream =
    builder.stream("topic-B");
ValueJoiner<String, String, String> valueJoiner =
    (leftValue, rightValue)-> {return leftValue +
    rightValue;
}

leftStream.join(rightStream, valueJoiner, JoinWindows.of
   (Duration. ofSeconds(10)));
```

Stateful Operations

- Sometimes you want to aggregate data in a stream to combine multiple messages and send one message downstream
  - This requires storing state somewhere for a while
  - And deciding when to emit the state value

```
Aggregator<String, String, Long> characterCountAgg =
    (key, value, charCount) -> value.length() + charCount;

myStream.groupByKey().aggregate(() ->
  0L, characterCountAgg,
  Materialized.with(Serdes.String(), Serdes.Long()))
  .toStream().to ("output-topic");
```

Windows

- To tell the aggregation how long to accumulate the state before releasing it use windows:
  - Fixed or tumbling
  - Sliding or hopping
  - Session

```
KStream<String, String> myStream = builder.stream ("topic-A");

Duration windowSize = Duration.ofSeconds(30);

TimeWindows tumblingWindow = TimeWindows.of(windowSize);

myStream.groupByKey().windowedBy(tumblingWindow).count();
```

Sliding Windows

- Sliding windows are good for seeing the near recent trends

```
KStream<String, String> myStream = builder.stream("topic-
A");

Duration windowSize = Duration.ofMinutes(5);
Duration advanceSize = Duration.ofMinutes(1);

TimeWindows hoppingWindow = TimeWindows.of(windowSize)
   .advanceBy(advanceSize);

myStream.groupByKey().windowedBy(hoppingWindow).count ();
```

Sliding Windows

- Sliding windows are good for seeing the near recent trends

```
KStream<String, String> myStream = builder.stream("topic-
A");

Duration windowSize = Duration.ofMinutes(5);
Duration advanceSize = Duration.ofMinutes(1);

TimeWindows hoppingWindow = TimeWindows.of(windowSize)
   .advanceBy(advanceSize);

myStream.groupByKey().windowedBy(hoppingWindow).count ();
```

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

APPENDIX A: UNDERSTANDING INTERNALS

Introduction

In this chapter, we will focus on three topics:

- How Kafka replication works
- How Kafka handles requests from producers and consumers
- How Kafka handles storage, such as file format and indexes

Cluster Membership

- Kafka uses Apache ZooKeeper to maintain the list of brokers that are currently members of a cluster
- Every broker has a unique identifier that is either set in the broker configuration file or automatically generated
  - Every time a broker process starts, it registers itself with its ID in ZooKeeper by creating an *ephemeral node*
- Different Kafka components subscribe to the `/brokers/ids` path in ZooKeeper where brokers are registered so they get notified when brokers are added or removed

The Controller

- The controller is one of the special Kafka brokers that is responsible for electing partition leaders
- Kafka uses ZooKeeper's ephemeral node feature to elect a controller and to notify the controller when nodes join and leave the cluster
- The controller is responsible for electing leaders among the partitions and replicas whenever it notices nodes join and leave the cluster
- The controller uses the epoch number to prevent a "split brain" scenario where two nodes believe each is the current controller
- The first broker that starts in the cluster becomes the controller by creating an ephemeral node in ZooKeeper
  - When other brokers start, they also try to create this node, but receive a "node already exists" exception
- When the controller broker is stopped, the ephemeral node will disappear Other brokers in the cluster will be notified through the ZooKeeper
  - The first node to create the new controller in ZooKeeper is the new controller, while the other nodes will receive a "node already exists" exception

Replication

- Replication is critical because it is the way Kafka guarantees availability and durability when individual nodes inevitably fail
- Kafka is organized by topics
  - Each topic is partitioned, and each partition can have multiple replicas
  - Those replicas are stored on brokers, and each broker typically stores hundreds or even thousands of replicas belonging to different topics and partitions
- There are two types of replicas:
  - *Leader replica:* Each partition has a single replica designated as the leader
    - All producer and consumer requests go through the leader, in order to guarantee consistency
  - *Follower replica:* Followers keep themselves in-sync with the leader
    - In the event that a leader replica for a partition crashes, one of the follower replicas will be promoted to become the new leader for the partition

Replication (continued)

- Another task the leader is responsible for is knowing which of the follower replicas is up-to-date with the leader
- In order to stay in-sync with the leader, the replicas send the leader Fetch requests, the exact same type of requests that consumers send in order to consume messages

Replication — Syncing Algorithm

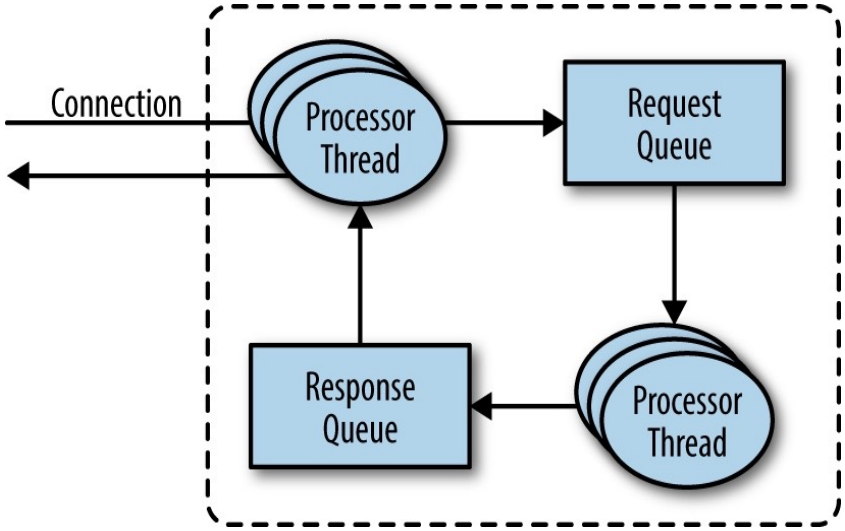
- A replica will not request the next message before it gets all the previous messages
- By looking at the last offset requested by each replica, the leader can tell how far behind each replica is
  - If a replica hasn't requested a message in more than 10 seconds, the replica is considered *out of sync*
- Replicas that are consistently asking for the latest messages, are called *in-sync replicas*
  - Only in-sync replicas are eligible to be elected as partition leaders in case the existing leader fails
- The amount of time a follower can be inactive or behind before it is considered out of sync is controlled by the `replica.lag.time.max.ms` configuration parameter

Request Processing

- Kafka broker processes requests sent to the partition leaders from clients, partition replicas, and the controller
- Kafka has a binary protocol (over TCP) that specifies the format of the requests and how brokers respond to them when:
  - Request is processed successfully
  - The broker encounters errors while processing the request
- Clients always initiate connections and send requests, and the broker processes the requests and responds to them
- All requests sent to the broker from a specific client will be processed in the order in which they were received

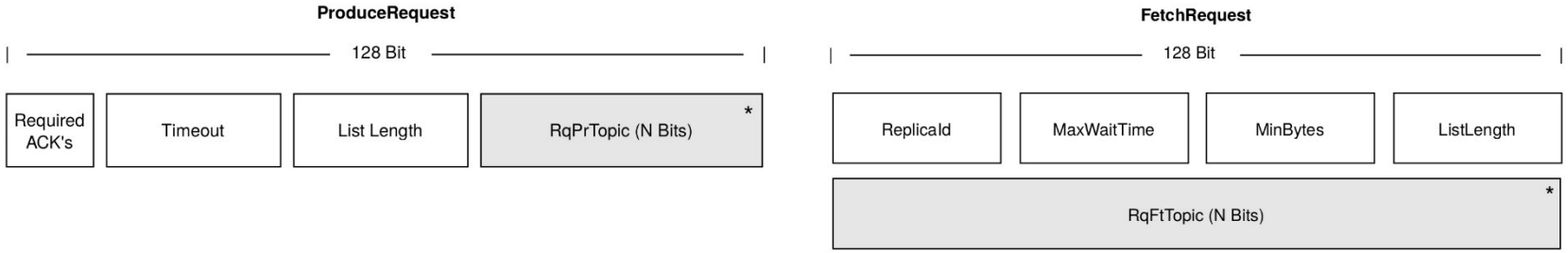
Request Processing Inside Kafka

- For each port the broker listens on, the broker runs an acceptor thread that creates a connection and hands it over to a processor thread for handling
- The network threads are responsible for taking requests from client connections, placing them in a request queue, and picking up responses from a response queue and sending them back to clients
- Once requests are placed on the request queue, IO threads are responsible for picking them up and processing them



Common Type of Requests (Produce and Fetch)

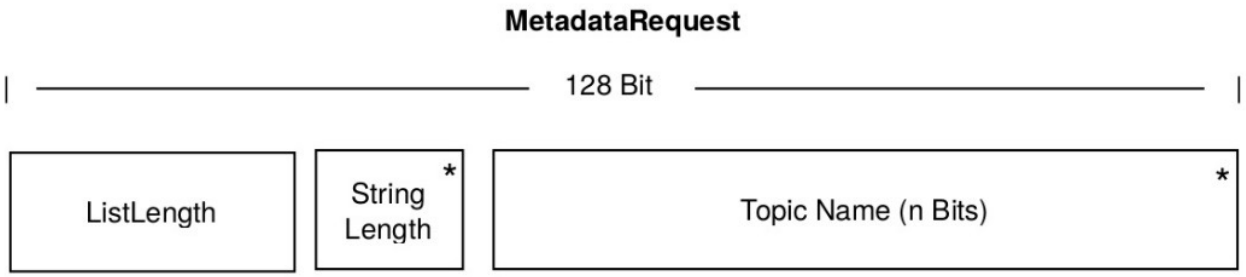
- Produce requests:
  - Sent by producers and contain messages the clients write to Kafka brokers
- Fetch requests:
  - Sent by consumers and follower replicas when they read messages
    - Fetches chunk of one or more logs for some topic-partitions



RequiredAcks	Data.Word16	Indicates acks the broker receive before responding to the request.
ListLength	Data.Word32	Number of elements in the following list.
ReplicaId	Data.Word32	Indicates the node id of the replica initiating this request.
MaxWaitTime	Data.Word32	Max amount of time in milliseconds to block in order to wait.
MinBytes	Data.Word32	Min number of bytes of messages that must be available to provide a response.
ListLength	Data.Word32	Number of elements in the list of topics.
Timeout	Data.Word32	This provides a maximum time in milliseconds the broker can await the receipt of the number of acknowledgements in RequiredAcks.
[RqPrTopic]	Data.List	Sequence of RqPrTopics(see Next Slide)

Common Type of Requests (Metadata Request)

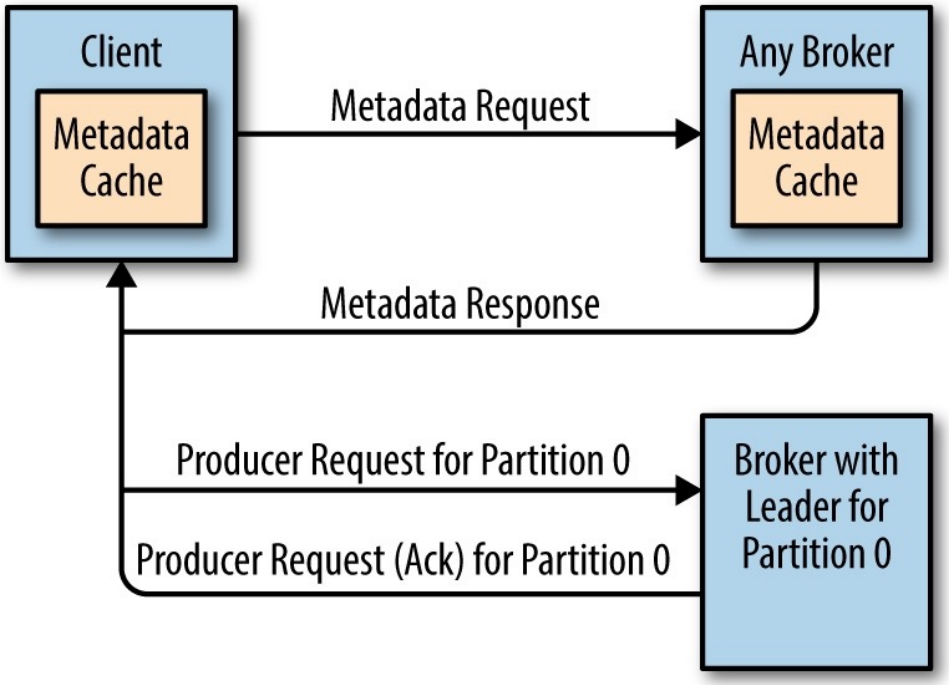
- Kafka clients use another request type called a *metadata request*, which includes a list of topics the client is interested in
- The server response specifies which partitions exist in the topics, the replicas for each partition, and which replica is the leader
- Metadata requests can be sent to any broker because all brokers have a metadata cache that contains this information




```
ListLength      Data.Word32        Number of elements in the list of StringLength and TopicName.
StringLength    Data.Word16        Length in bytes of the string TopicName.
TopicName       Data.ByteString    The topic to produce metadata for.
```

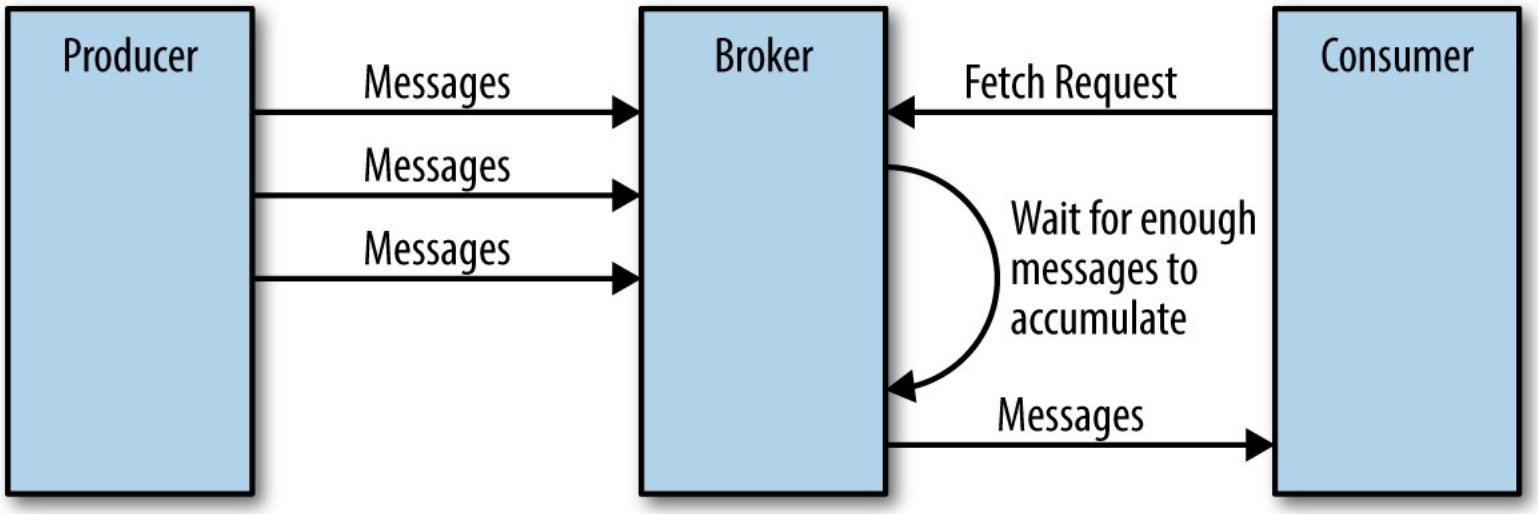
Client Routing Requests

- Clients typically cache this information and use it to direct produce and fetch requests to the correct broker for each partition



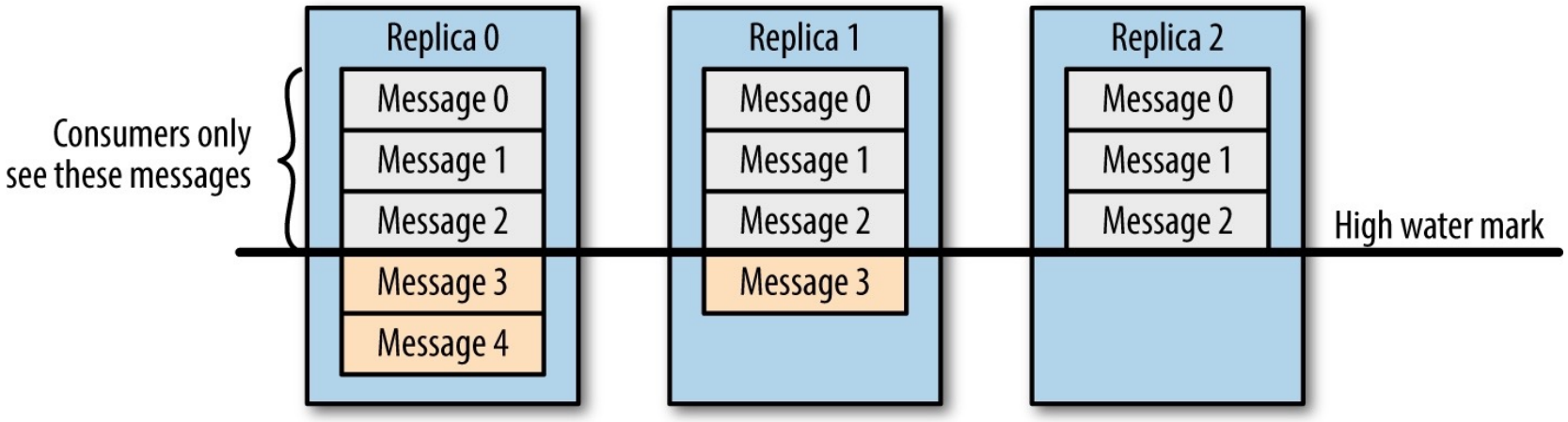
Producer and Consumer Interaction

- Broker delaying response until enough data accumulated



Consumers' Visibility

- Consumers only see messages that were replicated to in-sync replicas



Message Delivery Semantic

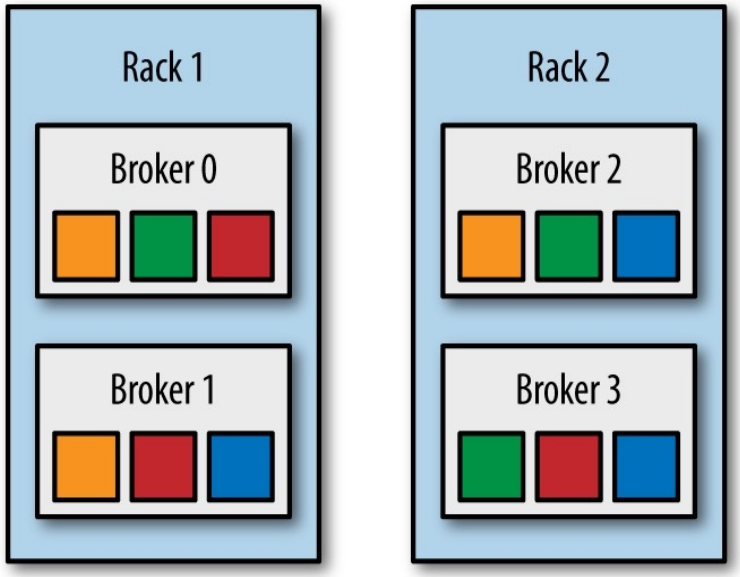
- There are multiple possible ways to deliver messages, such as:
  - At most once – messages may be lost but are never redelivered
  - At least once – messages are never lost but may be redelivered
  - Exactly once – this is what people want—each message is delivered once and only once

Physical Storage

- The basic storage unit of Kafka is a partition replica
  - Partitions cannot be split between multiple brokers and not even between multiple disks on the same broker
- The size of a partition is limited by the space available on a single mount point
- When configuring Kafka, the administrator defines a list of directories in which the partitions will be stored—this is the `log.dirs` parameter

Partition Allocation

- When doing the partitioning allocations, the goals are:
  - To make sure that for each partition, each replica is on a different broker
  - If the brokers have rack information, then assign the replicas for each partition to different racks if possible
    - This ensures that an event that causes downtime for an entire rack does not cause complete unavailability for partitions



Directory Selection

- Once the correct brokers for each partition and replica are chosen, the directory needs to be selected for the new partitions
  - This needs to be independently done for each partition
- The number of partitions on each directory are counted and the new partition is added to the directory with the fewest partitions
- Example:
  - If you add a new disk, all the new partitions will be created on that disk
    - This is because, until things balance out, the new disk will always have the fewest partitions

File Management — Segments

- Because finding the messages that need purging in a large file and then deleting a portion of the file is both time-consuming and error-prone, we instead split each partition into *segments*
  - By default, each segment contains either 1 GB of data or a week of data, whichever is smaller
- As a Kafka broker is writing to a partition, if the segment limit is reached, we close the file and start a new one
- The segment we are currently writing to is called an *active segment*
  - The active segment is never deleted
- A Kafka broker will keep an open file handle to every segment in every partition—even inactive segments
  - This leads to an usually high number of open file handles, and the OS must be tuned accordingly

Segments — File Format

- In Kafka, each partition is divided into segments and each segment is stored in a single data file
  - Inside the file, we store Kafka messages and their offsets
- The format of the data on the disk is identical to the format of the messages that we send from the producer to the broker, and later from the broker to the consumers
- Using the same message format on disk and over the wire is what allows Kafka to use zero-copy optimization when sending messages to consumers, and also avoid decompressing and recompressing messages that the producer already compressed

Contents of a Message

- Each message contains—in addition to its key, value, and offset—things like the message size, checksum code that allows us to detect corruption, magic byte that indicates the version of the message format, compression codec (Snappy, GZip, or LZ4), and a timestamp (added in release 0.10.0)
  - The timestamp is given either by the producer when the message was sent or by the broker when the message arrived, depending on configuration

Messages — Compression/Decompression

- If the producer is sending compressed messages, all the messages in a single producer batch are compressed together and sent as the "value" of a "wrapper message"
  - The broker receives a single message, which it sends to the consumer
  - But when the consumer decompresses the message value, it will see all the messages that were contained in the batch, with their own timestamps and offsets
- This means that if you are using compression on the producer (recommended!), sending larger batches means better compression both over the network and on the broker disks
  - This also means that if we decide to change the message format that consumers use (e.g., add a timestamp to the message), both the wire protocol and the on-disk format need to change, and Kafka brokers need to know how to handle cases in which files contain messages of two formats due to upgrades

Protocol Between Consumer and Broker

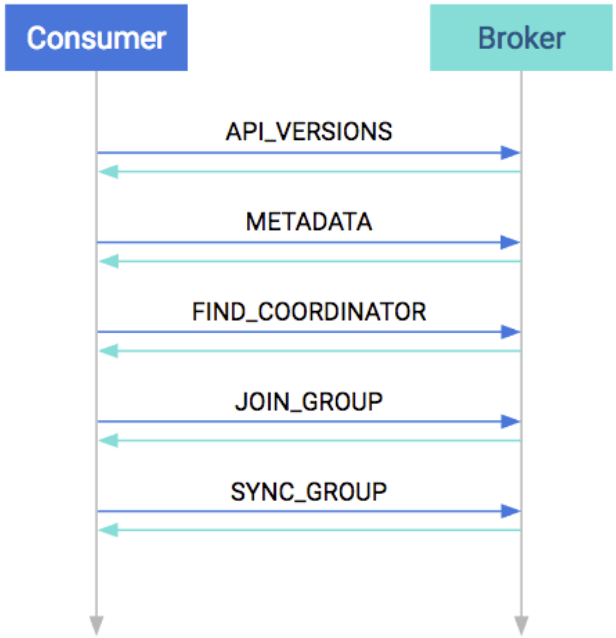
- Heartbeat
  - When the consumer is alive and is part of the consumer group, it sends heartbeats
    - These are short periodic messages that tell the brokers that the consumer is alive and everything is fine
- Session
  - Often one missing heartbeat is not a big deal, but how do you know if a consumer is not sending heartbeats for long enough to indicate a problem?
  - A session is such a time interval
  - If the consumer didn't send any heartbeats for longer than the session, the broker can consider the consumer dead and remove it from the group

Protocol Between Consumer and Broker (continued)

- Coordinator
  - The special broker which manages the group on the broker side is called the coordinator
  - The coordinator handles heartbeats and assigns the leader
  - Every group has a coordinator that organizes the startup of a consumer group and assist whenever a consumer leaves the group
- Leader
  - The leader consumer is elected by the coordinator
  - Its job is to assign partitions to every consumer in the group at startup or whenever a consumer leaves or joins the group
  - The leader holds the assignment strategy, it is decoupled from the broker
  - That means consumers can reconfigure the partition assignment strategy without restarting the brokers

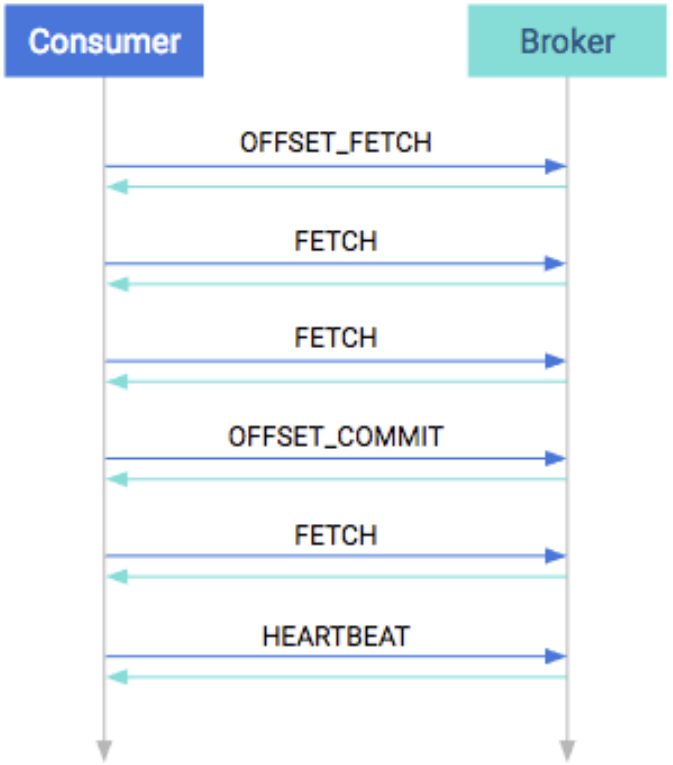
Startup Protocol

- At startup, the first step is to match protocol versions
- The next step is to collect cluster information, such as the addresses of all the brokers (prior to this point we used the bootstrap server as a reference), partition counts, and partition leaders
- After acquiring the metadata, the consumer has the information needed to join the group
  - By this time on the broker side, a coordinator has been selected per consumer group
  - The consumers must find their coordinator with the FIND_COORDINATOR request
- After finding the coordinator, the consumer(s) are ready to join the group
  - Every consumer in the group sends their own member-specific metadata to the coordinator in the JOIN_GROUP request
- The remaining step is to assign partitions to consumers and propagate this state



Consumption Protocol

- When consuming, the first step is to query where should the consumer start
  - This is done in the OFFSET_FETCH request
  - The consumer can also provide the offset manually



ROI TRAINING

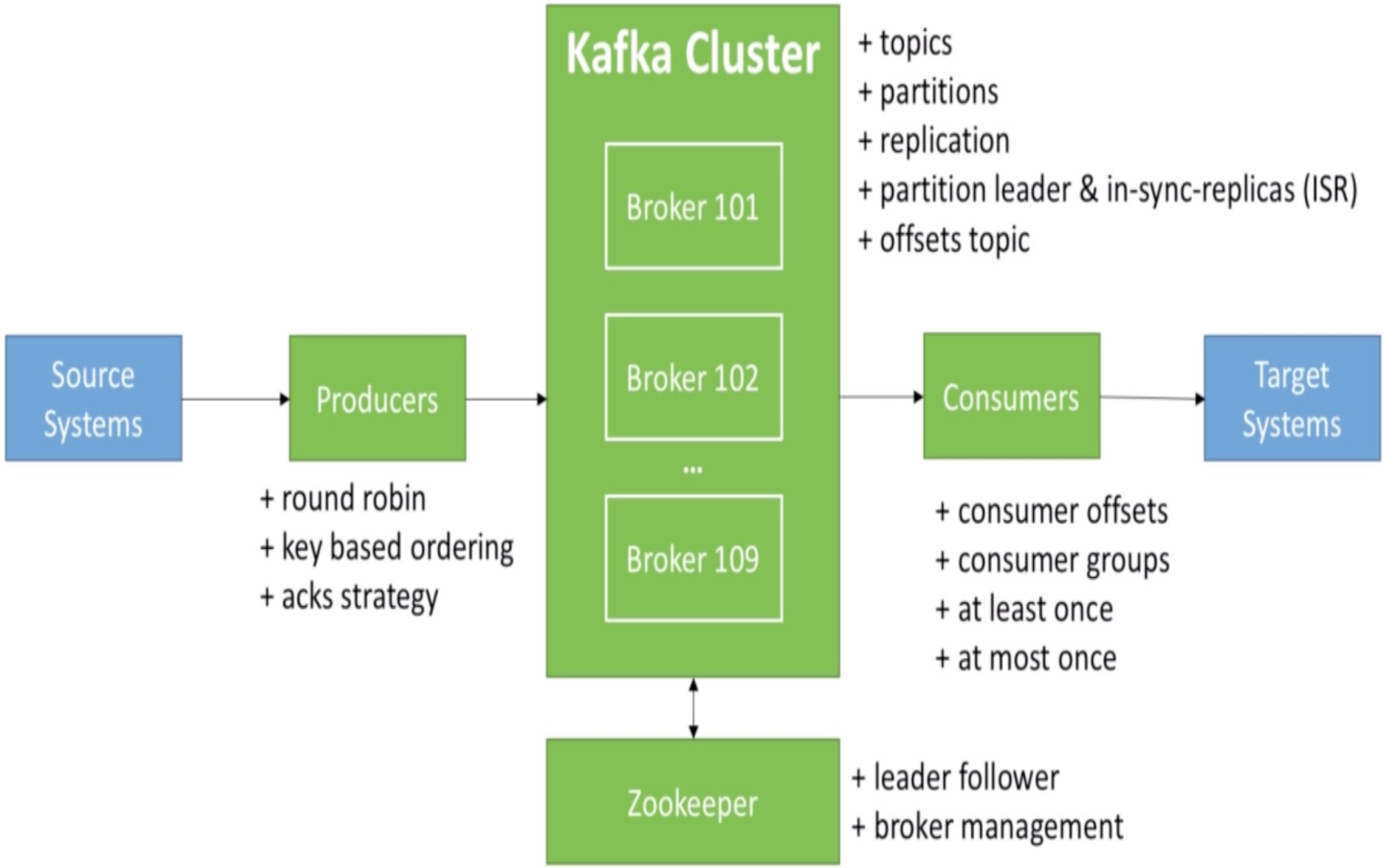
MAXIMIZE YOUR TRAINING INVESTMENT

APPENDIX B: DESIGN CONSIDERATIONS AND BEST PRACTICES

Introduction

- Kafka provides high availability and automatic recovery to distributed computing infrastructure
- Kafka is an ideal fit for communication and integration between components of large-scale data systems

Kafka Design



Kafka Design — ZooKeeper

- Kafka utilizes ZooKeeper for storing metadata information about the brokers, topics, and partitions
- Writes to ZooKeeper are only performed on changes to the membership of consumer groups or on changes to the Kafka cluster itself
  - This amount of traffic is minimal, and it does not justify the use of a dedicated ZooKeeper ensemble for a single Kafka cluster
  - Many deployments will use a single ZooKeeper ensemble for multiple Kafka clusters

Network Design and Deployment Considerations

- Kafka supports rack awareness
  - A single point of failure should not impact more than one ZooKeeper or one broker, or one partition of a topic
- ZooKeepers and brokers should have high availability (HA) communication; if one path is down, another path is used
  - They should be distributed ideally in different racks
- Network redundancy needs to provide the alternate access path to brokers for producers and consumers when failure arises
  - Also, as a good practice, brokers should be distributed across multiple racks

Network Configuration

- Network configuration with Kafka is similar to other distributed systems
- Infrastructure, whether on-premises or cloud, offers a variety of different IP and DNS options
  - Choose an option that keeps inter-broker network traffic on the private subnet and allows clients to connect to the brokers
  - Inter-broker and client communication use the same network interface and port
- When a broker is started, it registers its hostname with ZooKeeper
  - The producer since Kafka 0.8.1 and the consumer since 0.9.0 are configured with a bootstrapped (or "starting") list of Kafka brokers
  - Prior versions were configured with ZooKeeper
- In both cases, the client makes a request (either to a broker in the bootstrap list or to ZooKeeper) to fetch all broker hostnames and begin interacting with the cluster

ZooKeeper Configuration

- Distribute ZooKeeper nodes across multiple racks
  - ZooKeeper should be distributed across multiple racks as well, to increase fault tolerance
  - To tolerate a rack failure, ZooKeeper must be running in at least three different racks
  - Obviously, there is also the private network concept of failure
  - An enterprise can have ZooKeepers separated in different networks
  - That comes at the price of latency
  - It is a conscious decision between performance and reliability
  - Separating in different racks is a good compromise
  - In a configuration where three ZooKeepers are running in two racks, if the rack with two ZooKeepers fails, ZooKeeper will not have quorum and will not be available
- Monitor broker performance and terminate poorly performing brokers
  - Kafka broker performance can decrease unexpectedly over time for unknown reasons
  - It is a good practice to terminate and replace a broker if, for example, the 99th percentile of produce/fetch request latency is higher than is tolerable for your application

Handling Rebalancing

- Two things are relevant to rebalancing
- One is leader re-election, or preferred replica election, and the other one is partition rebalancing
- The first one is more of a case when a node is down and is brought back again within a certain period of time
- The other one is when we want to either decrease or increase the number of nodes in the cluster

Handling Rebalancing (continued)

Scenario 1: When the broker is down because of maintenance or due to server failure and is brought back within a certain period of time

- There are two ways to handle this scenario
  - One is adding the following line to the broker configuration, "`auto.leader.rebalance.enable`", to automatically rebalance, but this is reported to have issues
  - The other way is to manually use the "`kafka-preferred-replica-election.sh`" tool
- Edit `server.properties` of Kafka to add the following line: `auto.leader.rebalance.enable = false`
- Load is not evenly distributed—let's bring Broker 4 back online
- Now we can see that even though the Broker 4 is back online, it is not serving as a leader for any of the partitions
  - Let's run the `kafka-preferred-replica-election.sh` tool to balance the load
- Now we can see that topics are evenly balanced

Handling Rebalancing (continued)

Scenario 2: When a node is down and not recoverable

- We create a new broker and update the `broker.id` with the previous one's id which was not recoverable, and manually run "`kafka-preferred-replica-election.sh`" for topic balancing

Scenario 3: To increase or decrease the number of nodes in a Kafka cluster

- Following are the steps to balance topics when increasing or decreasing number of nodes
- Using partition reassignment tool (`kafka-reassign-partition.sh`), generate (with the `-generate` option) the candidate assignment configuration
  - This shows the current and the proposed replica assignments
- Once the partition reassignment is completed, run "`kafka-preferred-replica-election.sh`" tool to complete the balancing

Kafka — Best Practices with Producers

- Configure your producer to wait for acknowledgments
- Configure retries on your producers
- For high-throughput producers, tune buffer sizes
- Instrument your application to track metrics

Kafka — Best Practices with Brokers

- Compacted topics require memory and CPU resources on your brokers
- Monitor your brokers for network throughput
- Distribute partition leadership among brokers in the cluster
- Don't neglect to monitor your brokers for in-sync replica (ISR) shrinks, under-replicated partitions, and un-preferred leaders
- Modify the `ApacheLog4j_properties` as needed
- Either disable automatic topic creation or establish a clear policy regarding the cleanup of unused topics

Kafka — Best Practices with Brokers (continued)

- For sustained, high-throughput brokers, provision sufficient memory to avoid reading from the disk subsystem
- For a large cluster with high-throughput service level objectives (SLOs), consider isolating topics to a subset of brokers
- Using older clients with newer topic message formats, and vice versa, places extra load on the brokers
- Don’t assume that testing a broker on a local desktop machine is representative of the performance you’ll see in production

Kafka's Main Components

- Kafka Ecosystem
  - The Kafka ecosystem consists of Kafka Core, Kafka Streams, Kafka Connect, Kafka REST Proxy, and the Schema Registry
  - Most of the additional pieces of the Kafka ecosystem come from Confluent and are not part of Apache
- What is Kafka Streams?
  - Kafka Streams enables real-time processing of streams
  - It can aggregate across multiple streams, joining data from multiple streams, allowing for stateful computations, and more
- What is Kafka Connect?
  - Kafka Connect is the connector API to create reusable producers and consumers (e.g., stream of changes from DynamoDB)
  - Kafka Connect Sources are sources of records
  - Kafka Connect Sinks are a destination for records

Kafka's Main Components (continued)

- What is the Schema Registry?
  - The Schema Registry manages schemas using Avro for Kafka records
- What is Kafka Mirror Maker?
  - The Kafka Mirror Maker is used to replicate cluster data to another cluster
- When might you use Kafka REST Proxy?
  - The Kafka REST Proxy is used to producers and consumer over REST (HTTP)
  - You could use it for easy integration of existing code bases

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

COURSE SUMMARY

Course Summary

In this course, we have:

- Described the architecture of Kafka
- Explored Kafka producers and consumers for writing and reading messages
- Understood publish-subscribe messaging and how it fits in the big data ecosystem
- Explored how Kafka's stream delivery capabilities make it a perfect source for stream processing systems
- Learned various strategies of monitoring Kafka
- Gotten best practices for building data pipelines and applications with Kafka

ROI's Training Curricula

Agile Development	.NET and Visual Studio
Amazon Web Services (AWS)	Networking and IPv6
Azure	Oracle and SQL Server Databases
Big Data and Data Analytics	OpenStack and Docker
Business Analysis	Project Management
Cloud Computing and Virtualization	Python and Perl Programming
Excel and VBA	Security
Google Cloud	SharePoint
ITIL® and IT Service Management	Software Analysis and Design
Java	Software Engineering
Leadership and Management Skills	UNIX and Linux
Machine Learning and Neural Networks	Web and Mobile Apps
Microsoft Exchange	Windows and Windows Server

Our full course list is available at www.ROItraining.com