Confluent Developer Training: Building Kafka Solutions

B7/801/A

Kafka Fundamentals

Chapter 3

Course Contents

Kafka Fundamentals

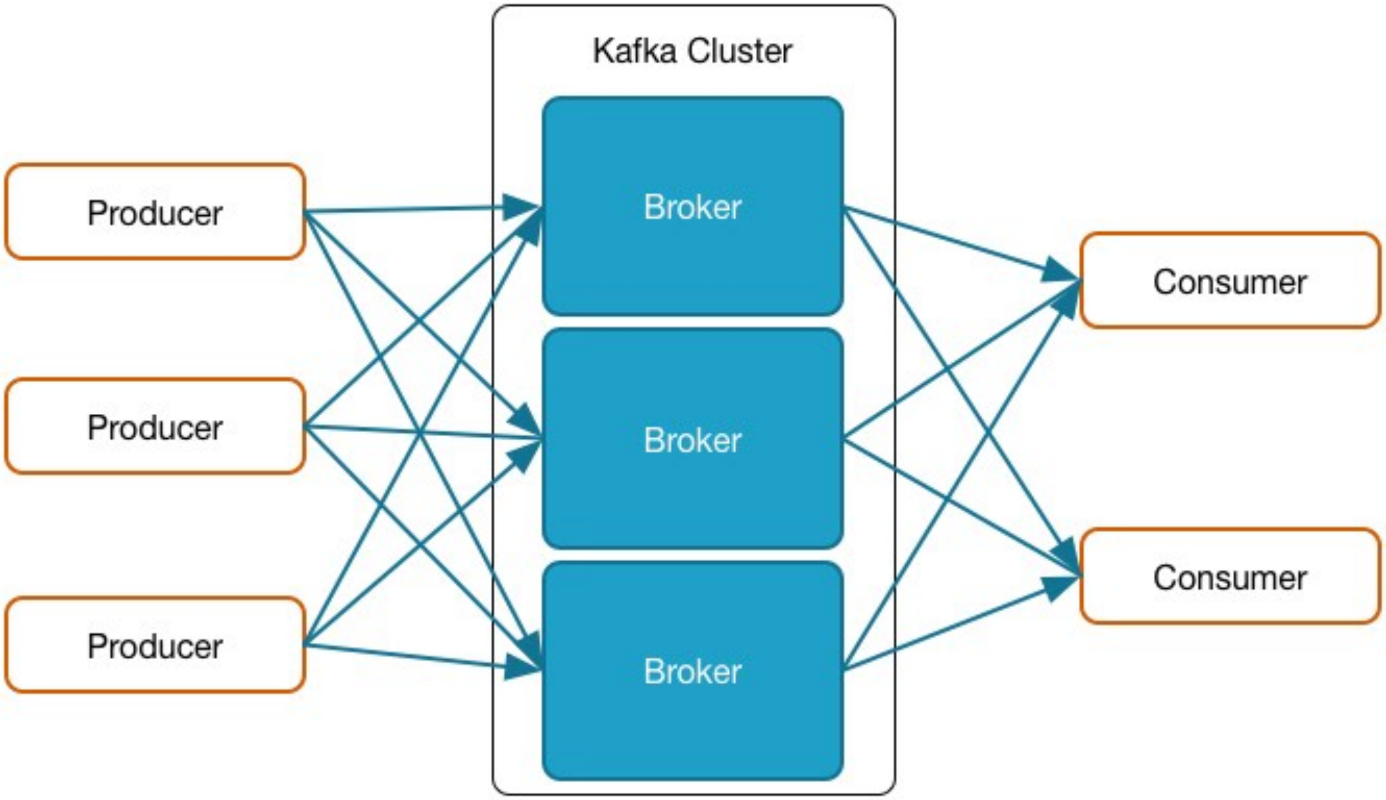
- **In this chapter you will learn:**
  - How Producers write data to a Kafka cluster
  - How data is divided into Partitions, and then stored on Brokers
  - How Consumers read data from the cluster

Kafka Fundamentals

- **An Overview of Kafka**
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

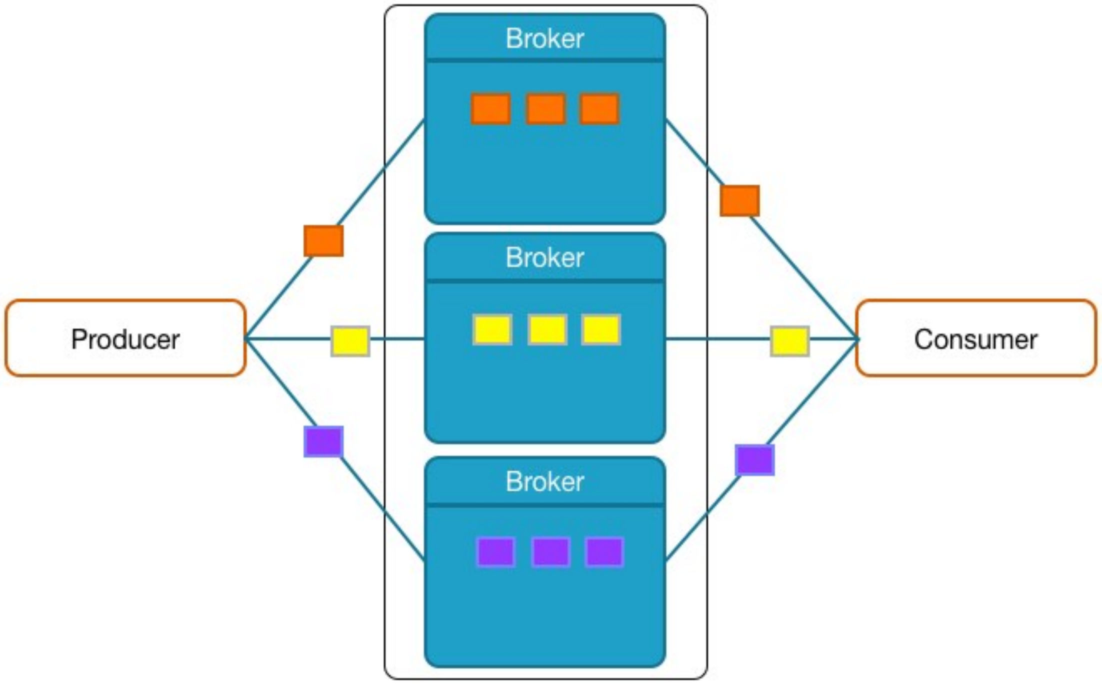
Reprise: A Very High-Level View of Kafka

- ***Producers* send data to the Kafka cluster**
- ***Consumers* read data from the Kafka cluster**
- ***Brokers* are the main storage and messaging components of the Kafka cluster**



Kafka Messages

- **The basic unit of data in Kafka is a *message***
  - *Message* is sometimes used interchangeably with *record*
  - Producers write messages to Brokers
  - Consumers read messages from Brokers

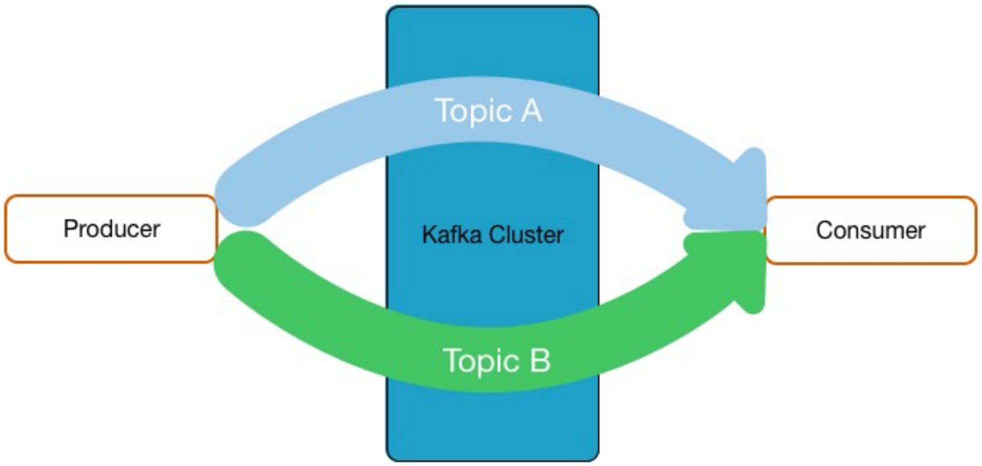


Key-Value Pairs

- **A message is a key-value pair**
  - All data is stored in Kafka as byte arrays
  - Producer provides serializers to convert the key and value to byte arrays
  - Key and value can be any data type

Topics

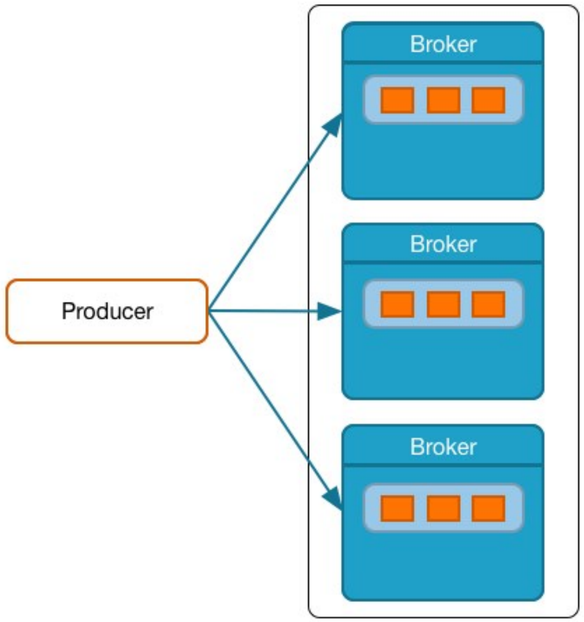
- **Kafka maintains streams of messages called *Topics***
  - Logical representation
  - They categorize messages into groups



- **Developers decide which Topics exist**
  - By default, a Topic is auto-created when it is first used
- **One or more Producers can write to one or more Topics**
- **There is no limit to the number of Topics that can be used**

Partitioned Data Ingestion

- **Producers shard data over a set of *Partitions***
  - Each Partition contains a subset of the Topic's messages
  - Each Partition is an ordered, immutable log of messages
- **Partitions are distributed across the Brokers**
- **Typically, the message key is used to determine which Partition a message is assigned to**



Load Balancing and Semantic Partitioning

- **Producers use a partitioning strategy to assign each message to a Partition**
- **Having a partitioning strategy serves two purposes**
  - Load balancing: shares the load across the Brokers
  - Semantic partitioning: user-specified key allows locality-sensitive message processing
- **The partitioning strategy is specified by the Producer**
  - Default strategy is a hash of the message key
    - `hash(key) % number_of_partitions`
  - If a key is not specified, messages are sent to Partitions on a round-robin basis
- **Developers can provide a custom partitioner class**

Kafka Components

- **There are four key components in a Kafka system**
  - Producers
  - Brokers
  - Consumers
  - ZooKeeper
- **We will now investigate each of these in turn**

Kafka Fundamentals

- *An Overview of Kafka*
- **Kafka Producers**
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

Producer Basics

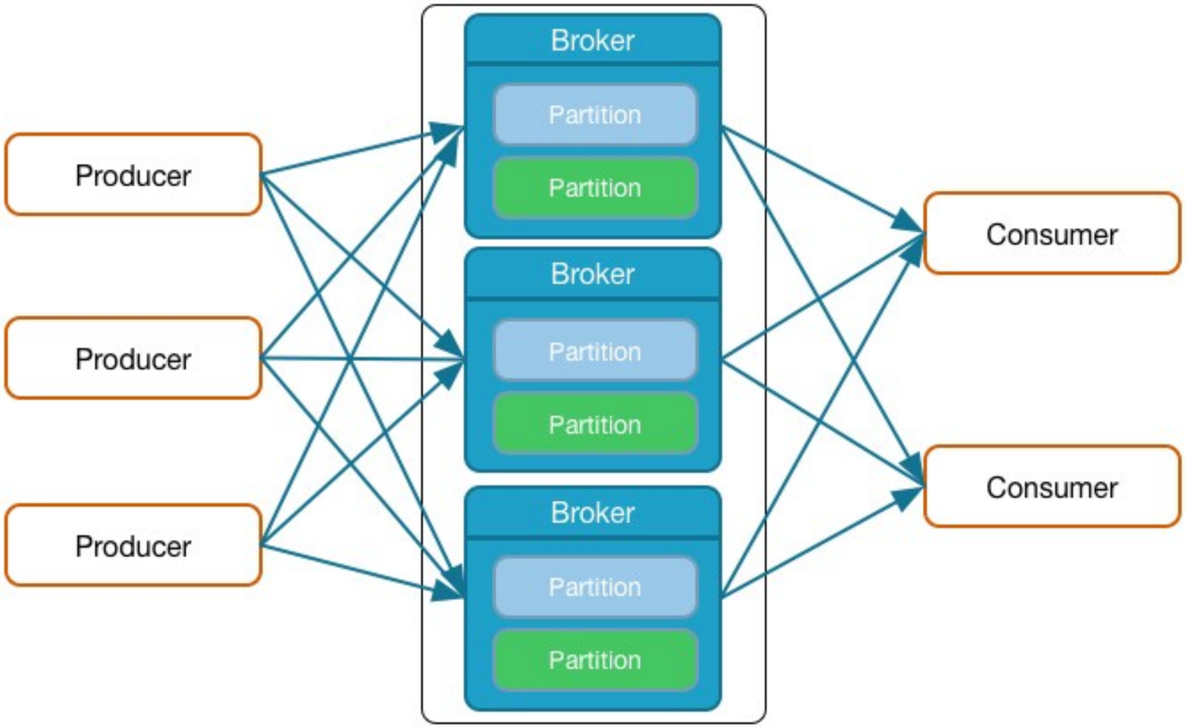
- **Producers write data in the form of *messages* to the Kafka cluster**
- **Producers can be written in any language**
  - Native Java, C/C++, Python, Go, .NET, JMS clients are supported by Confluent
  - Clients for many other languages exist
  - Confluent develops and supports a REST (REpresentational State Transfer) server which can be used by clients written in any language
- **A command-line Producer tool exists to send messages to the cluster**
  - Useful for testing, debugging, etc.

Kafka Fundamentals

- *An Overview of Kafka*
- *Kafka Producers*
- **Kafka Brokers**
- *Kafka Consumers*
- *Kafka’s Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka’s Command-Line Tools*
- *Chapter Review*

Broker Basics

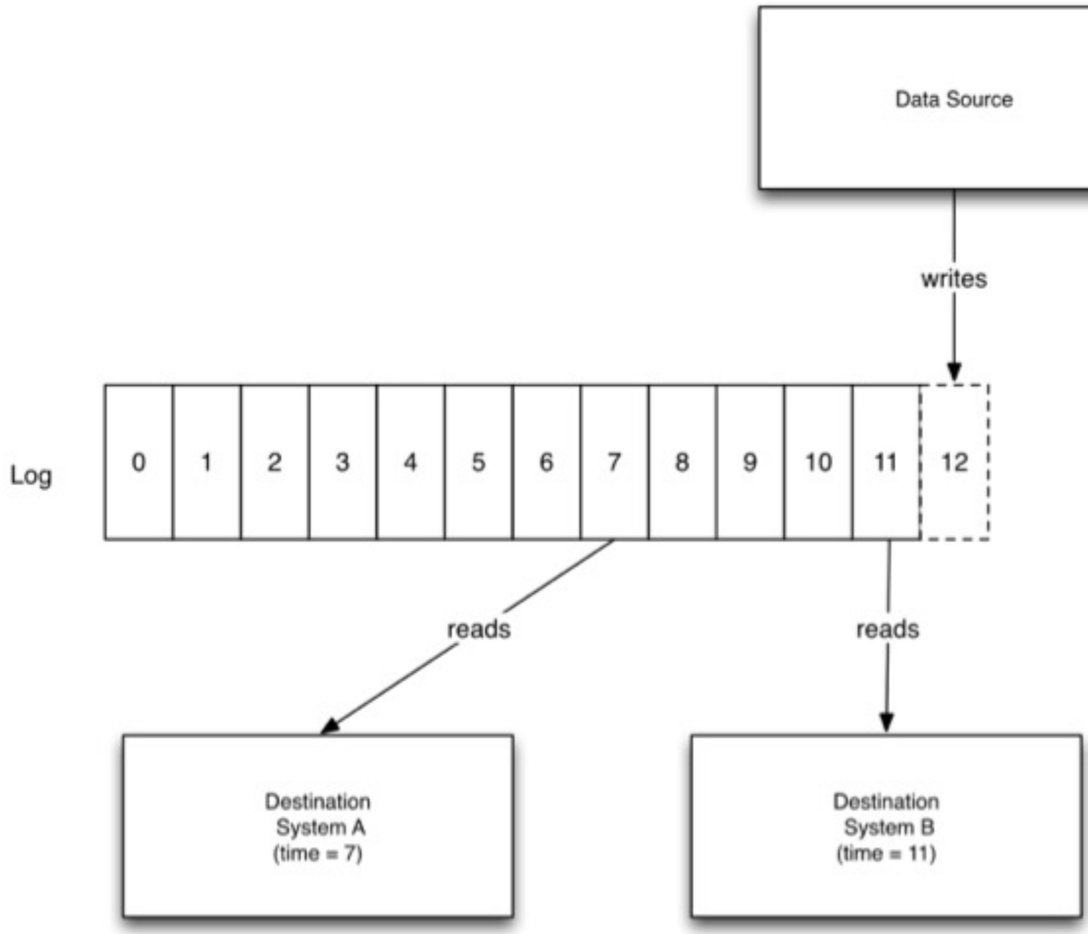
- **Brokers receive and store messages when they are sent by the Producers**
- **A production Kafka cluster will have three or more Brokers**
  - Each can handle hundreds of thousands, or millions, of messages per second



Brokers Manage Partitions

- **Messages in a Topic are spread across Partitions in different Brokers**
- **Typically, a Broker manages multiple Partitions**
- **Each Partition is stored on the Broker's disk as one or more `log` files**
  - Not to be confused with `log4j` files used for monitoring
- **Each message in the log is identified by its *offset* number**
  - A monotonically increasing value
- **Kafka provides a configurable retention policy for messages to manage log file growth**
  - Retention policies can be configured per Topic

Messages are Stored in a Persistent Log

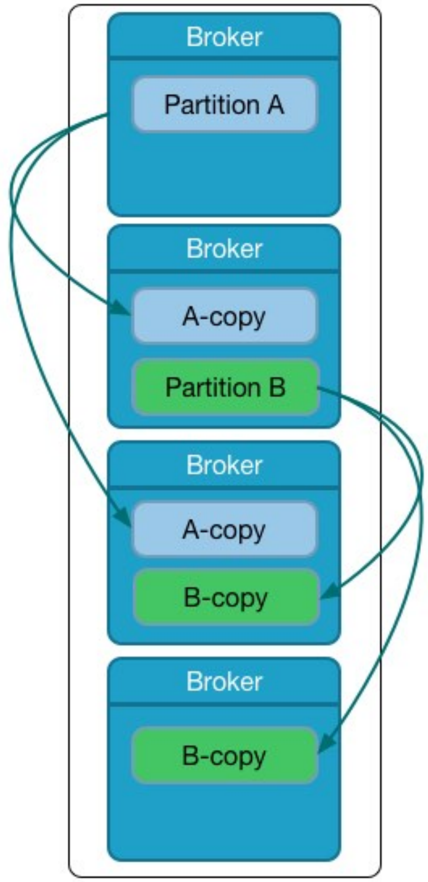


Metadata in a Kafka Message

- **A Kafka Message contains data and metadata**
  - Key/value pair
  - Offset
  - Timestamp
  - Compression type
  - Magic byte
  - Optional message headers API
    - Application teams can add custom key-value paired metadata to messages
  - Additional fields to support batching, exactly once semantics, replication protocol
- **Latest message format:** `http://kafka.apache.org/documentation.html#messageformat`

Fault Tolerance via a Replicated Log

- **Partitions can be replicated across multiple Brokers**
- **Replication provides fault tolerance in case a Broker goes down**
  - Kafka automatically handles the replication



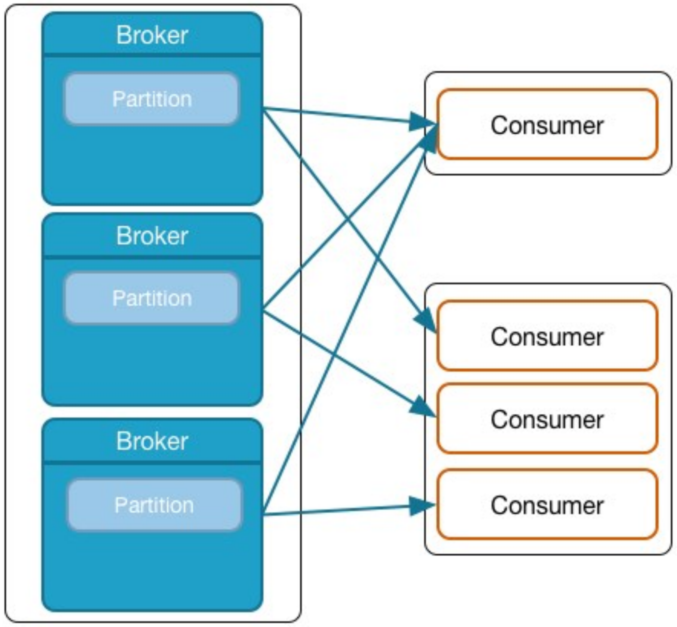
Kafka Fundamentals

Consumer Basics

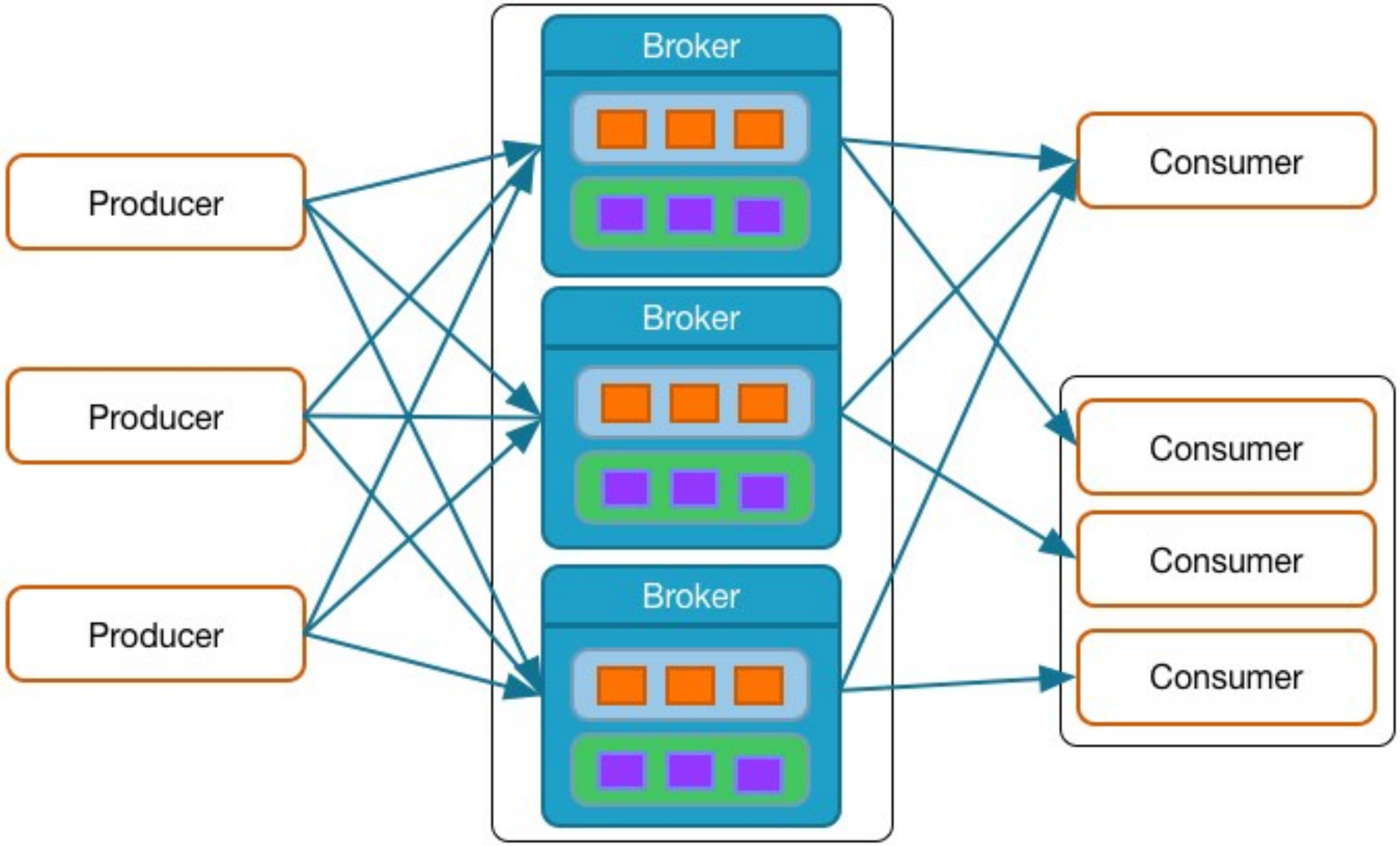
- **Consumers pull messages from one or more Topics in the cluster**
  - As messages are written to a Topic, the Consumer will automatically retrieve them
- **The *Consumer Offset* keeps track of the latest message read**
  - If necessary, the Consumer Offset can be changed
    - For example, to reread messages
- **The Consumer Offset is stored in a special Kafka Topic**
- **A command-line Consumer tool exists to read messages from the cluster**
  - Useful for testing, debugging, etc.

Distributed Consumption

- **Different Consumers can read data from the same Topic**
  - By default, each Consumer will receive all the messages in the Topic
- **Multiple Consumers can be combined into a *Consumer Group***
  - Consumer Groups provide scaling capabilities
  - Each Consumer is assigned a subset of Partitions for consumption



The Result: A Linearly Scalable Firehose



Kafka Fundamentals

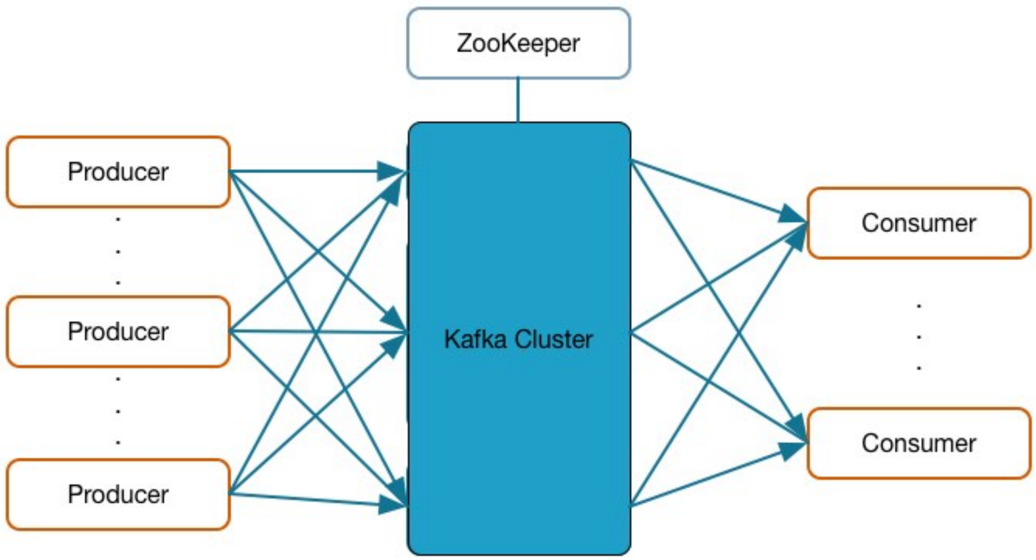
- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- **Kafka's Use of ZooKeeper**
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

What is ZooKeeper?

- **ZooKeeper is a centralized service that can be used by distributed applications**
  - Open source Apache project
  - Enables highly reliable distributed coordination
  - Maintains configuration information
  - Provides distributed synchronization
- **Used by many projects**
  - Including Kafka and Hadoop
- **Typically consists of three or five servers in an *ensemble***
  - This provides resiliency should a machine fail

How Kafka Uses ZooKeeper

- **Kafka Brokers use ZooKeeper for a number of important internal features**
  - Cluster management
  - Failure detection and recovery
  - Access Control List (ACL) storage



Quiz: Question

- **Provide the correct relationship — 1:1, 1:N, N:1, or N:N**
  - Broker to Partition — ? N:N
  - Key to Partition — ? N:1
  - Producer to Topic — ? N:N
  - Consumer Group to Topic — ? N:N
  - Consumer (in a Consumer Group) to Partition — ? q:N

1:N

Quiz: Answer

- **Provide the correct relationship — 1:1, 1:N, N:1, or N:N**
  - Broker to Partition — N:N
  - Key to Partition — N:1
  - Producer to Topic — N:N
  - Consumer Group to Topic — N:N
  - Consumer (in a Consumer Group) to Partition — 1:N

Kafka Fundamentals

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- **Comparisons with Traditional Message Queues**
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- *Chapter Review*

Kafka's Benefits over Traditional Message Queues

- **Durability and Availability**
  - Messages are replicated on multiple machines for reliability
  - Cluster can handle Broker failures
- **Excellent scalability**
  - Even a small cluster can process a large volume of messages
    - Tests have shown that three low-end machines can easily deal with two million writes per second
- **Very high throughput**
- **Supports both real-time and batch consumption**
- **Data retention**

Multi-Subscription and Scalability

- **Multi-subscription provides easy distribution of data**
  - Once the data is in Kafka, it can be read by multiple different Consumers
  - For instance, a Consumer which writes the data to the Hadoop Distributed File System (HDFS), another to do real-time analysis on the data, etc.
- **Multiple Brokers, multiple Topics, and Consumer Groups provide very high scalability**
  - Kafka was designed as a distributed system from the ground up
  - Enables parallelism

The Advantages of a Pull Architecture

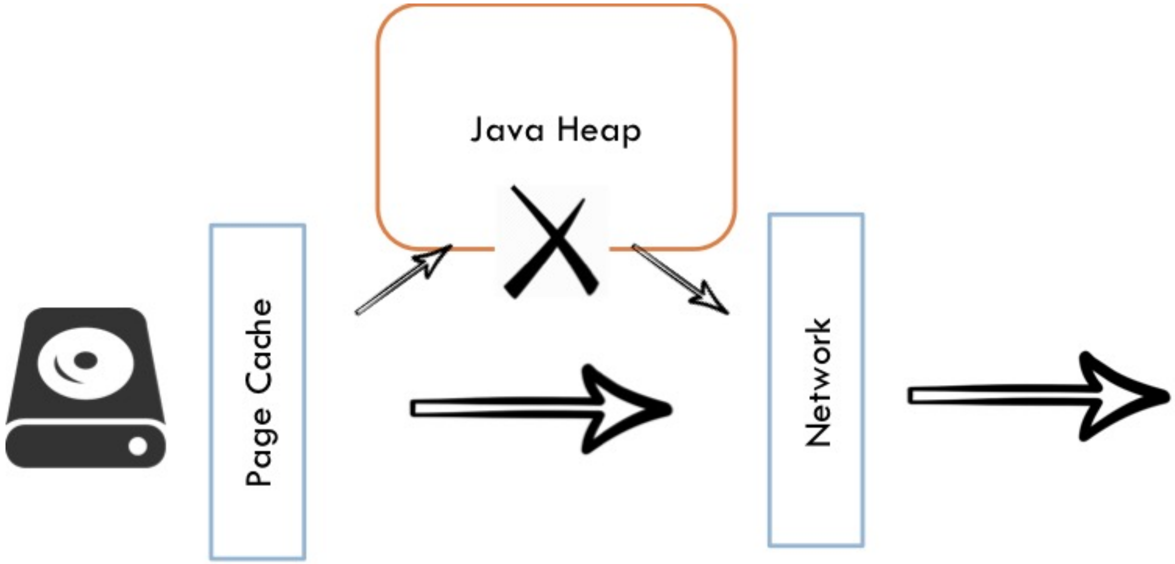
- **Kafka Consumers pull messages from the Brokers**
  - This is in contrast to some other systems, which use a *push* design
- **The advantages of pulling, rather than pushing, data, include:**
  - The ability to add more Consumers to the system without reconfiguring the cluster
  - The ability for a Consumer to go offline and return later, resuming from where it left off
  - No problems with the Consumer being overwhelmed by data
    - It can pull, and process, the data at whatever speed it needs to
    - A slow Consumer will not affect Producers

The Page Cache for High Performance

- **Unlike some systems, Kafka itself does not require a lot of RAM**
- **Logs are held on disk and read when required**
- **Kafka makes use of the operating system's page cache to hold recently-used data**
  - Typically, recently-Produced data is the data which Consumers are requesting
- **A Kafka Broker running on a system with a reasonable amount of RAM for the OS to use as cache will typically be able to swamp its network connection**
  - In other words the network, not Kafka itself, will be the limiting factor on the speed of the system

Speeding Up Data Transfer

- **Kafka uses zero-copy data transfer (Broker → Consumer)**



Kafka Fundamentals

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- **Hands-On Exercise: Using Kafka's Command-Line Tools**
- *Chapter Review*

Hands-On Exercise: Using Kafka's Command-Line Tools

- **In this Hands-On Exercise you will use Kafka's command-line tools to Produce and Consume data**
- **Please refer to the Hands-On Exercise Manual**

Kafka Fundamentals

- *An Overview of Kafka*
- *Kafka Producers*
- *Kafka Brokers*
- *Kafka Consumers*
- *Kafka's Use of ZooKeeper*
- *Comparisons with Traditional Message Queues*
- *Hands-On Exercise: Using Kafka's Command-Line Tools*
- **Chapter Review**

Chapter Review

- **A Kafka system is made up of Producers, Consumers, and Brokers**
  - ZooKeeper provides co-ordination services for the Brokers
- **Producers write messages to Topics**
  - Topics are broken down into partitions for scalability
- **Consumers read data from one or more Topics**

Kafka’s Architecture

Chapter 4

Course Contents

Kafka’s Architecture

- How Kafka’s log files are stored on the Kafka Brokers
- How Kafka uses replicas for reliability
- How Consumer Groups and Partitions provide scalability

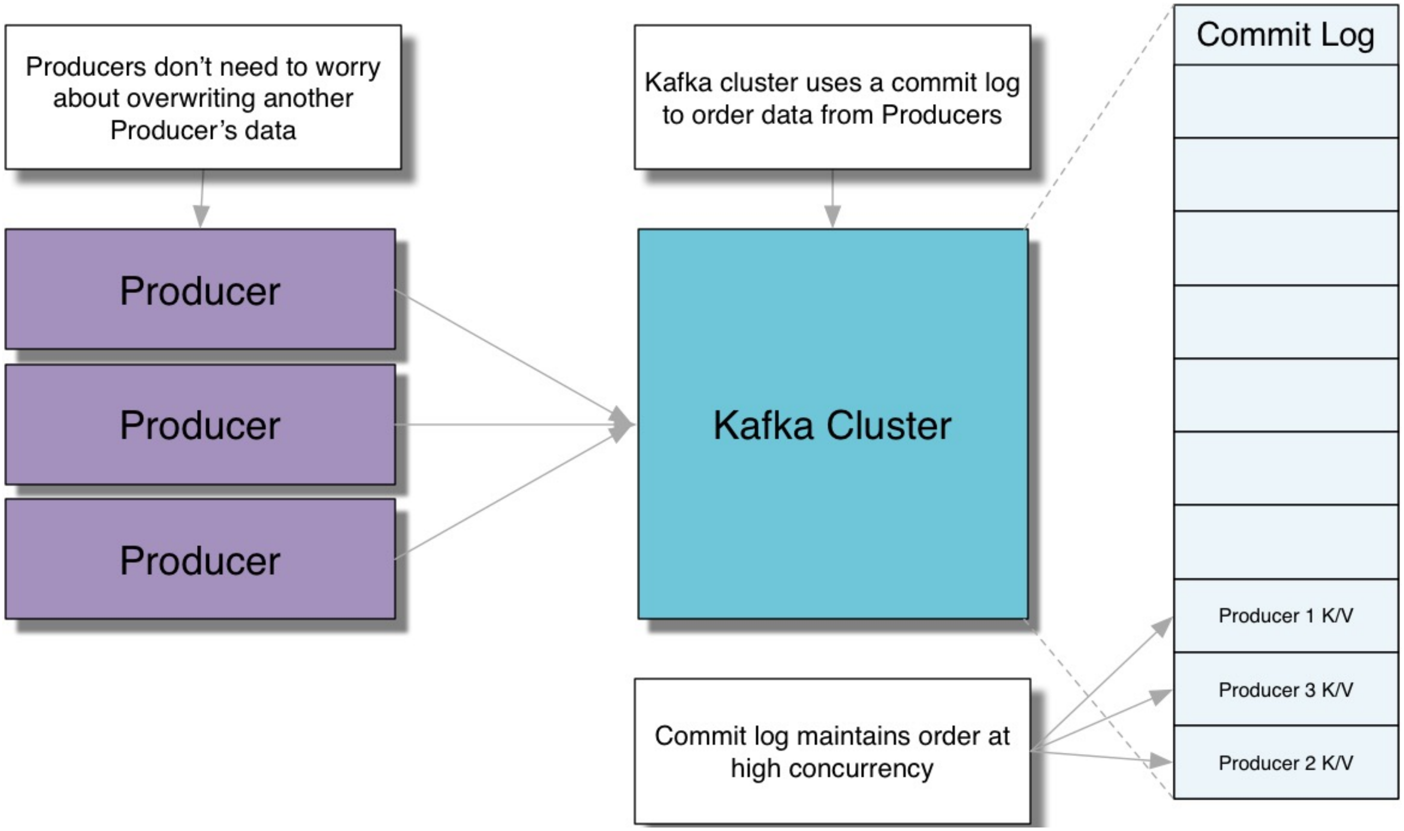
Kafka’s Architecture

- **Kafka’s Log Files**
- *Replicas for Reliability*
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Review*

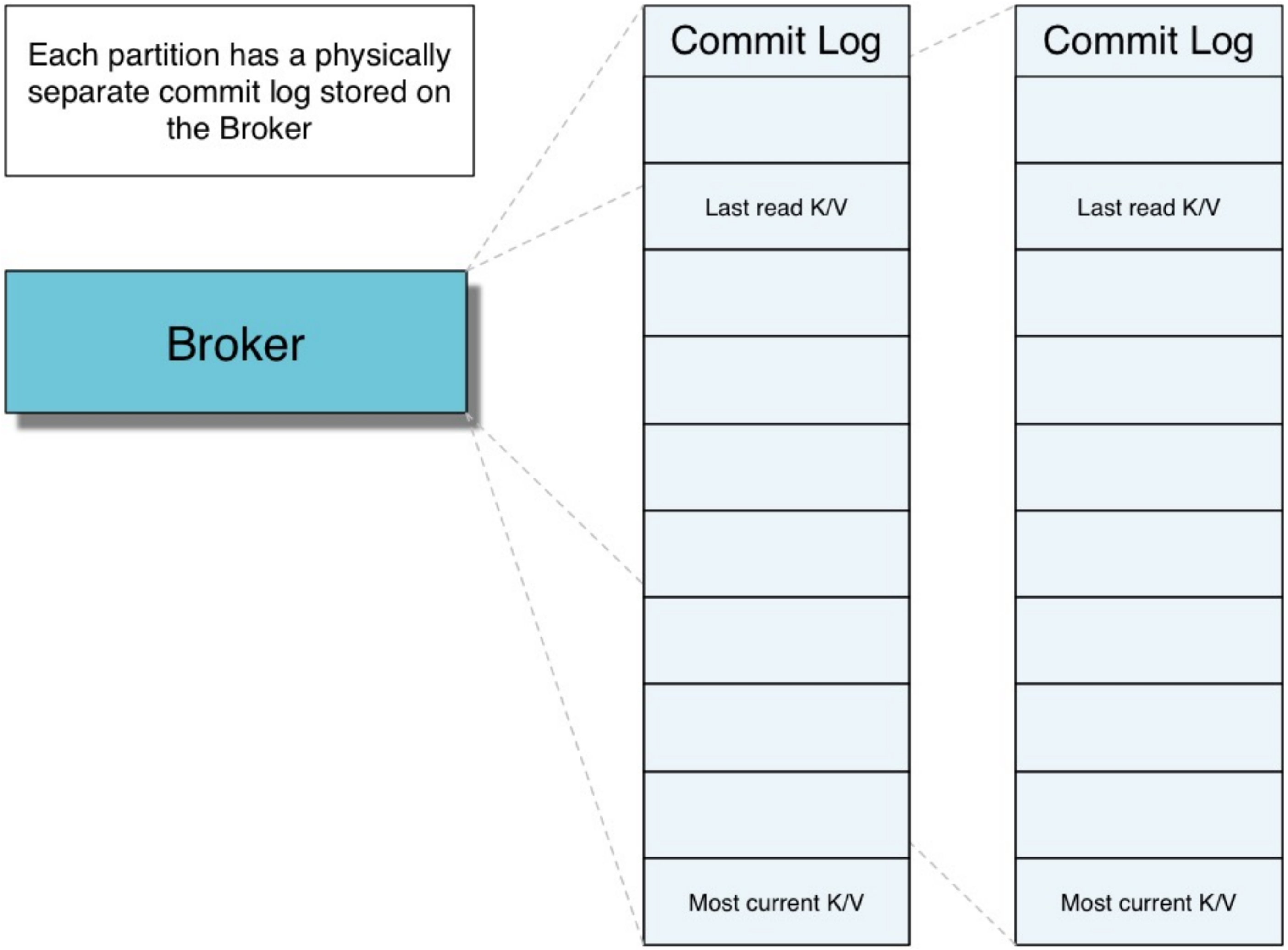
What is a Commit Log?

- **A Commit Log is a way to keep track of changes as they happen**
- **Commonly used by databases to keep track of all changes to tables**
- **Kafka uses commit logs to keep track of all messages in a particular Topic**
  - Consumers can retrieve previous data by backtracking through the commit log

The Commit Log for High Concurrency



Partitions Are Stored as Separate Logs



Kafka’s Architecture

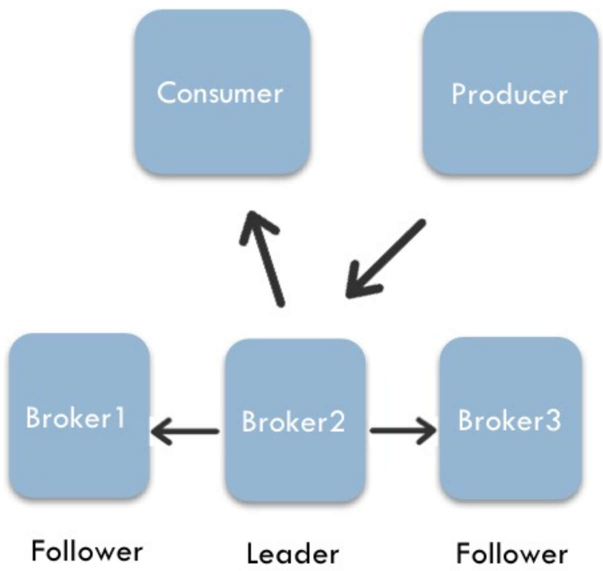
- *Kafka’s Log Files*
- **Replicas for Reliability**
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Review*

Problems With our Current Model

- **So far, we have said that each Broker manages one or more Partitions for a Topic**
- **This does not provide reliability**
  - A Broker failing would result in all of those Partitions being unavailable
- **Kafka takes care of this by replicating each partition**
  - The replication factor is configurable

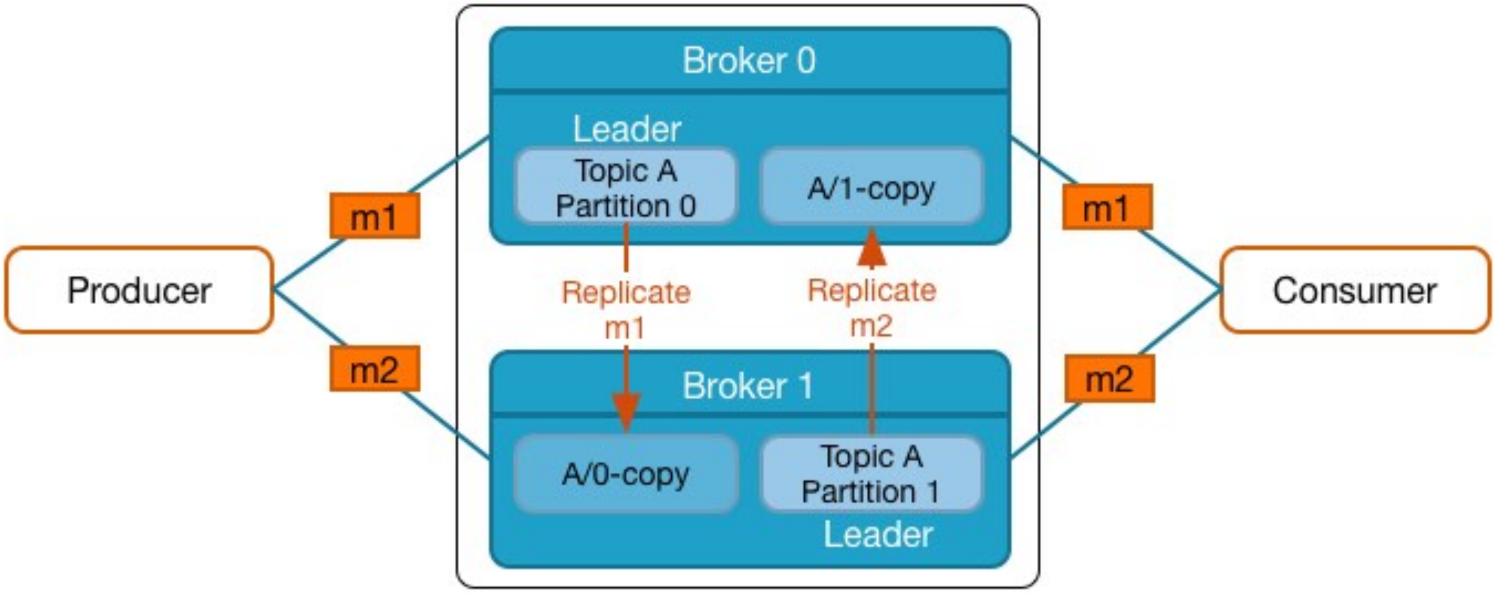
Replication of Partitions

- **Kafka maintains replicas of each partition on other Brokers in the cluster**
  - Number of replicas is configurable
- **One Broker is the leader for that Partition**
  - All writes and reads go to and from the leader
  - Other Brokers are followers



Important: Clients Do Not Access Followers

- **It is important to understand that Producers only write to the leader**
- **Likewise, Consumers *only* read from the leader**
  - They do not read from the replicas
  - Replicas only exist to provide reliability in case of Broker failure
- **In the diagram below, `m1` hashes to Partition 0 and `m2` hashes to Partition 1**



- **If a leader fails, the Kafka cluster will elect a new leader from among the followers**

In-Sync Replicas

- **You may see information about "In-Sync Replicas" (ISRs) from some Kafka command-line tools**
- **ISRs are replicas which are up-to-date with the leader**
  - If the leader fails, it is the list of ISRs which is used to elect a new leader
- **Although this is more of an administration Topic, it helps to be familiar with the term ISR**

Managing Broker Failures

- **One Broker in the entire cluster is designated as the Controller**
  - Detects Broker failure/restart via ZooKeeper
- **Controller action on Broker failure**
  - Selects a new leader and updates the ISR list
  - Persists the new leader and ISR list to ZooKeeper
  - Sends the new leader and ISR list changes to all Brokers
- **Controller action on Broker restart**
  - Sends leader and ISR list information to the restarted Broker
- **If the Controller fails, one of the other Brokers will become the new Controller**

Kafka's Architecture

- *Kafka's Log Files*
- *Replicas for Reliability*
- **Partitions and Consumer Groups for Scalability**
- *Hands-On Exercise: Consuming from Multiple Partitions*
- *Chapter Review*

Scaling using Partitions

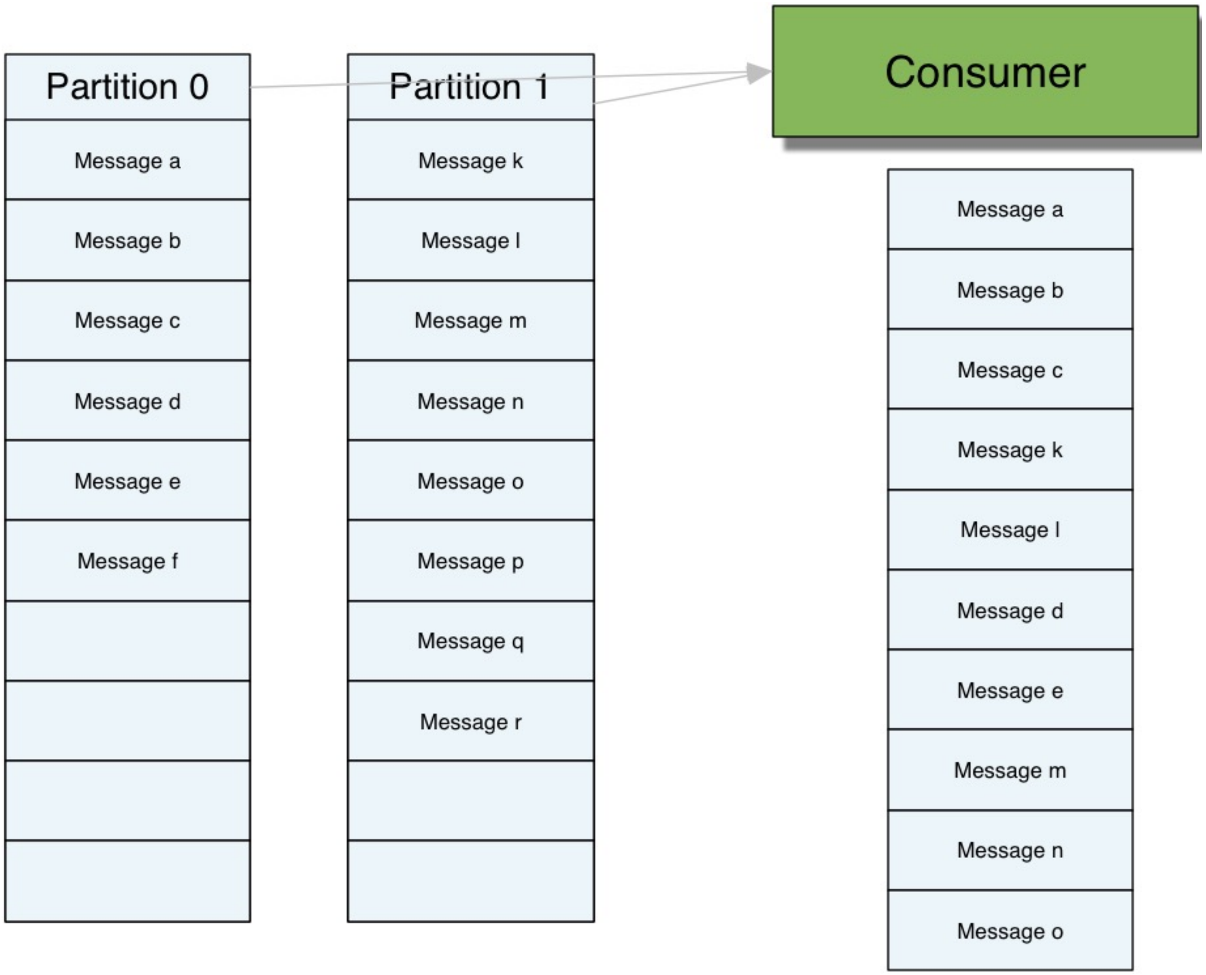
- **Recall: All Consumers read from the leader of a Partition**
  - No clients write to, or read from, followers
- **This can lead to congestion on a Broker if there are many Consumers**
- **Splitting a Topic into multiple Partitions can help to improve performance**
  - Leaders for different Partitions can be on different Brokers

Preserve Message Ordering

- **Messages with the same key, from the same Producer, are delivered to the Consumer in order**
  - Kafka hashes the key and uses the result to map the message to a specific Partition
  - Data within a Partition is stored in the order in which it is written
  - Therefore, data read from a Partition is read in order *for that partition*
- **If the key is null and the default Partitioner is used, the record is sent to a random Partition**

An Important Note About Ordering

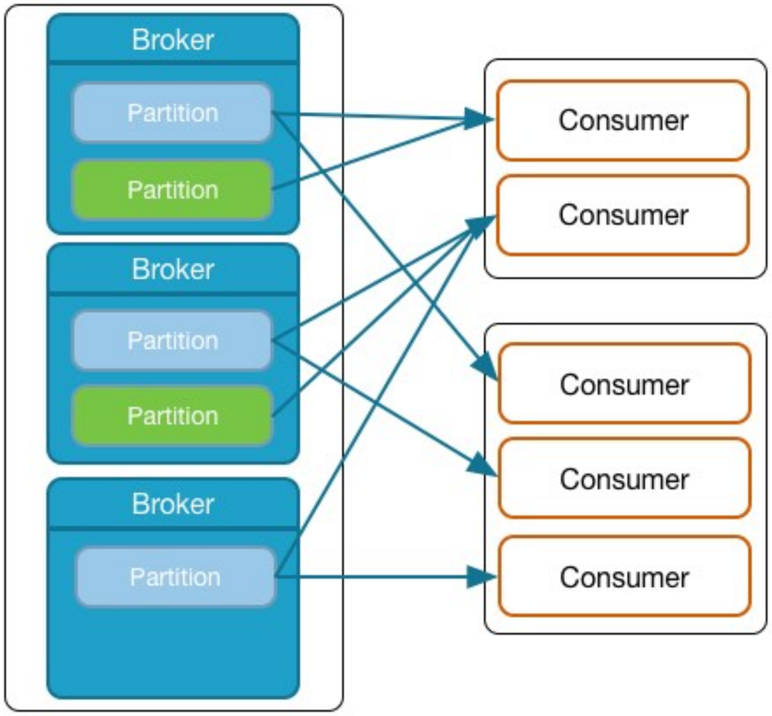
- **If there are multiple Partitions, you** will not get total ordering across all messages **when reading data**



- **Question: how can you preserve message order if the application requires it?**

Group Consumption Balances Load

- **Multiple Consumers in a *Consumer Group***
  - The `group.id` property is identical across all Consumers in the group
- **Consumers in the group are typically on separate machines**
- **Automatic failure detection and load rebalancing**

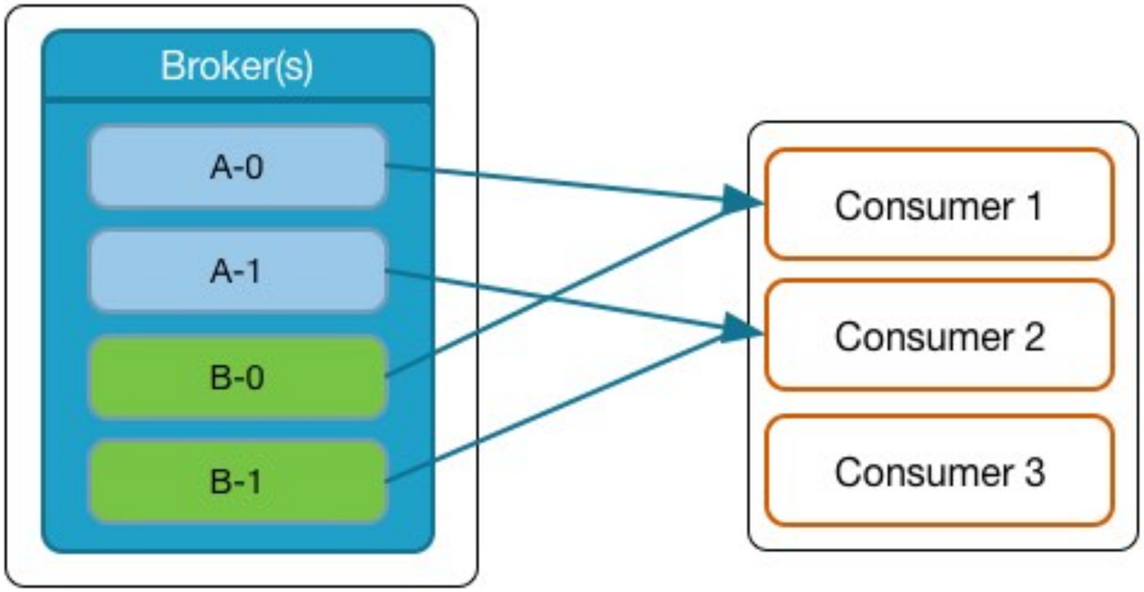


Partition Assignment within a Consumer Group

- **Partitions are 'assigned' to Consumers**
  - A single Partition is consumed by only one Consumer in any given Consumer Group
  - *i.e.*, you are guaranteed that messages with the same key will go to the same Consumer
    - Unless you change the number of partitions (see later)
  - `partition.assignment.strategy` in the Consumer configuration

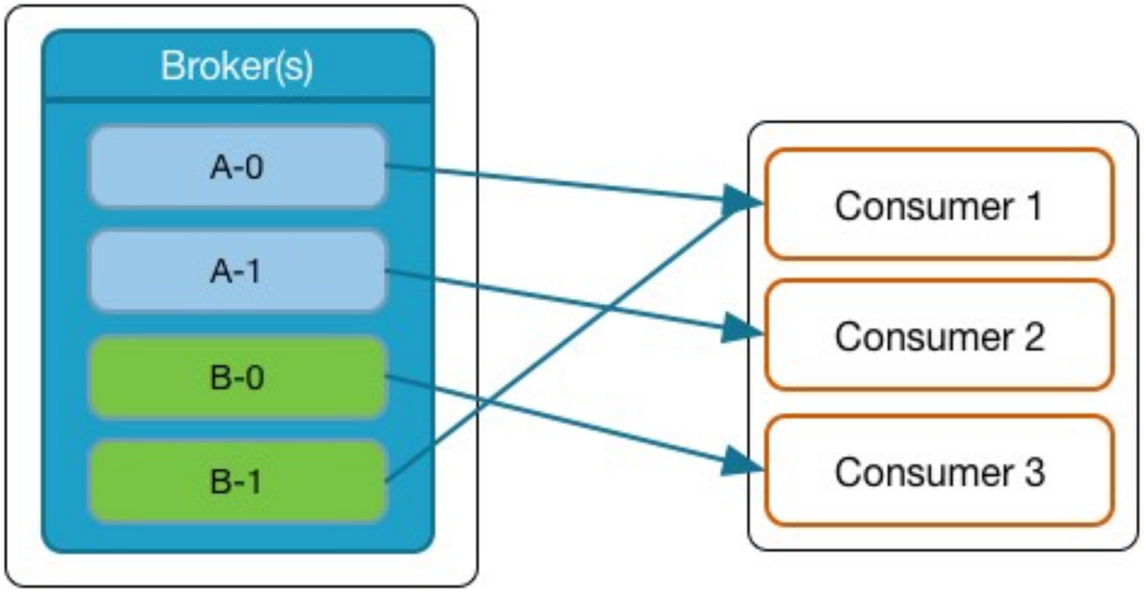
Partition Assignment Strategy: Range

- **Range (default)**
  - Topic/Partition: topics `A` and `B`, each with two Partitions `0` and `1`
  - Consumer Group: Consumers c1, c2, c3
  - c1: {A-0, B-0}   c2: {A-1, B-1}   c3: {}



Partition Assignment Strategy: RoundRobin

- **RoundRobin**
  - c1: {A-0, B-1}   c2: {A-1}   c3: {B-0}



- **Partition assignment is automatically recomputed on changes in Partitions/Consumers**

Partition Assignment Strategy: Sticky

- **Sticky assignment strategy was introduced in Kafka 0.11 (Confluent 3.3)**
- **Guarantees an assignment that is as balanced as possible**
- **Preserves existing Partition assignments to Consumers during reassignment to reduce overhead**
  - Kafka Consumers retain pre-fetched messages for Partitions assigned to them before a reassignment
  - Reduces the need to cleanup local Partition state between rebalances

Consumer Groups: Limitations

- **The number of useful Consumers in a Consumer Group is constrained by the number of Partitions on the Topic**
  - Example: If you have a Topic with three partitions, and ten Consumers in a Consumer Group reading that Topic, only three Consumers will receive data
    - One for each of the three Partitions

Consumer Groups: Caution When Changing Partitions

- **Recall: All messages with the same key will go to the same Consumer**
  - However, if you change the number of Partitions in the Topic, this may not be the case
    - Example: Using Kafka's default Partitioner, Messages with key *K1* were previously written to Partition 2 of a Topic
    - After repartitioning, new messages with key *K1* may now go to a different Partition
    - Therefore, the Consumer which was reading from Partition 2 may not get those new messages, as they may be read by a new Consumer

Kafka's Architecture

- *Kafka's Log Files*
- *Replicas for Reliability*
- *Partitions and Consumer Groups for Scalability*
- **Hands-On Exercise: Consuming from Multiple Partitions**
- *Chapter Review*

Hands-On Exercise: Consuming from Multiple Partitions

- **In this Hands-On Exercise, you will create a Topic with multiple Partitions, write data to the Topic, then read the data back to see how ordering of the data is affected**
- **Please refer to the Hands-On Exercise Manual**

Kafka's Architecture

- *Kafka's Log Files*
- *Replicas for Reliability*
- *Partitions and Consumer Groups for Scalability*
- *Hands-On Exercise: Consuming from Multiple Partitions*
- **Chapter Review**

Chapter Review

- **Kafka uses commit logs to store all its data**
  - These allow the data to be read back by any number of Consumers
- **Topics can be split into Partitions for scalability**
- **Partitions are replicated for reliability**
- **Consumers can be collected together in Consumer Groups**
  - Data from a specific Partition will go to a single Consumer in the Consumer Group
- **If there are more Consumers in a Consumer Group than there are Partitions in a Topic, some Consumers will receive no data**

Developing With Kafka

Chapter 5

Course Contents

Developing With Kafka

- **In this chapter you will learn:**
  - How to write a Producer using the Java API
  - How to use the REST proxy to access Kafka from other languages
  - How to write a basic Consumer using the New Consumer API

Developing With Kafka

- **Programmatically Accessing Kafka**
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

The Kafka API

- **Since Kafka 0.9, Kafka includes Java clients in the `org.apache.kafka.clients` package**
  - These are intended to supplant the older Scala clients
  - They are available in a JAR which has as few dependencies as possible, to reduce code size
- **There are client libraries for many other languages**
  - The quality and support for these varies
- **Confluent provides and supports client libraries for C/C++, Python, Go, and .NET**
- **Confluent also maintains a REST Proxy for Kafka**
  - This allows any language to access Kafka via REST
    - (REpresentational State Transfer; essentially, a way to access a system by making HTTP calls)

Our Class Environment

- **During the course this week, we anticipate that you will be writing code either in Java...**
  - In which case, you will use Kafka's Java API
- **...or Python**
  - In which case, you will use the REST Proxy
- **If you wish to use some other programming language to access the REST proxy, you can do so**
  - Be aware that your instructor may not be familiar with your language of choice, though

Developing With Kafka

The Producer API

- **To create a Producer, use the `KafkaProducer` class**
- **This is thread safe; sharing a single Producer instance across threads will typically be faster than having multiple instances**
- **Create a `Properties` object, and pass that to the Producer**
  - You will need to specify one or more Broker host/port pairs to establish the initial connection to the Kafka cluster
    - The property for this is `bootstrap.servers`
    - This is only used to establish the initial connection
    - The client will use all servers, even if they are not all listed here
    - Question: why not just specify a single server?

Important **`Properties`** Elements (1)

Name	Description
`bootstrap.servers`	List of Broker host/port pairs used to establish the initial connection to the cluster
`key.serializer`	Class used to serialize the key. Must implement the **`Serializer`** interface
`value.serializer`	Class used to serialize the value. Must implement the **`Serializer`** interface
`compression.type`	How data should be compressed. Values are **`none`**, **`snappy`**, **`gzip`**, **`lz4`**. Compression is performed on batches of records

Important `Properties` Elements (2)

Name	Description
`acks`	Number of acknowledgment the Producer requires the leader to have before considering the request complete. This controls the durability of records. **`acks=0`**: Producer will not wait for any acknowledgment from the server; **`acks=1`**: Producer will wait until the leader has written the record to its local log; **`acks=all`**: Producer will wait until all in-sync replicas have acknowledged receipt of the record

Creating the **`Properties`** and **`KafkaProducer`** Objects

```java
Properties props = new Properties;
props.put("bootstrap.servers", "broker1:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

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

- **Other serializers available: `ByteArraySerializer`, `IntegerSerializer`, `LongSerializer`**
- **`StringSerializer` encoding defaults to UTF8**
  - Can be customized by setting the property `serializer.encoding`

Helper Classes

- **Kafka includes helper classes `ProducerConfig`, `ConsumerConfig`, and `StreamsConfig`**
  - Provide predefined constants for commonly configured properties
- **Examples**

```
// With helper class
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
// Without helper class
props.put("bootstrap.servers", "broker1:9092");
```

```
// With helper class
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
// Without helper class
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

```
// With helper class
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
// Without helper class
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

Sending Messages to Kafka

```
String k = "mykey";
String v = "myvalue";
ProducerRecord<String, String> record = new ProducerRecord<String, String>("my_topic", k, v); ①
producer.send(record); ②
```

① **ProducerRecord can take an optional timestamp if you don't want to use current system time**

② **Alternatively:**

```
producer.send(new ProducerRecord<String, String>("my_topic", k, v));
```

`send()` Does Not Block

- **The `send()` call is asynchronous**
  - It does not block; it returns immediately and your code continues
- **It returns a `Future` which contains a `RecordMetadata` object**
  - The Partition the record was put into and its offset
- **To force `send()` to block, call `producer.send(record).get()`**

When Do Producers Actually Send?

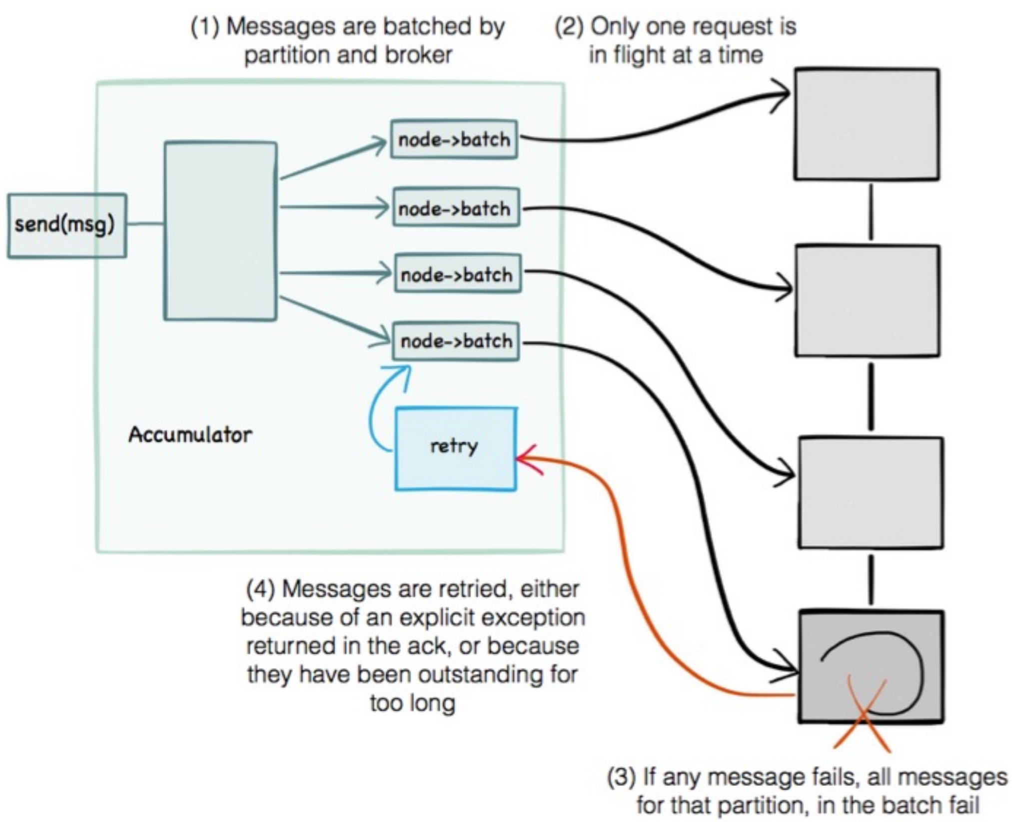
- **A Producer `send()` returns immediately after it has added the message to a local buffer of pending record sends**
  - This allows the Producer to send records in batches for better performance
- **Then the Producer flushes multiple messages to the Brokers based on batching configuration parameters**
  - You can also manually flush by calling the Producer method `flush()`
- **Do not confuse the Producer method `flush()` in a Producer context with the term `flush` in a Broker context**
  - `flush` in a Broker context refers to when messages get written from the page cache to disk

Producer Retries

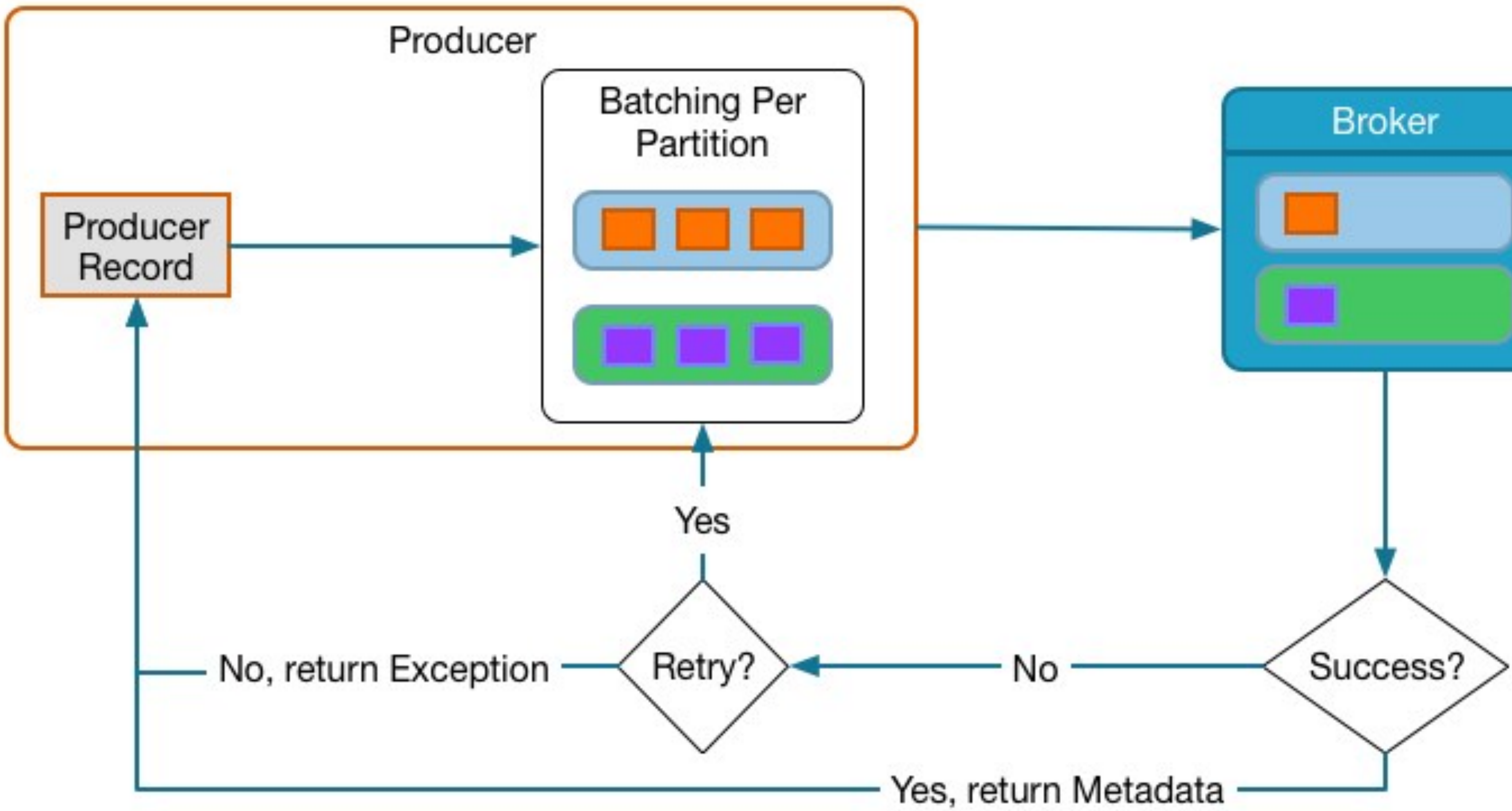
- **Developers can configure the `retries` configuration setting in the Producer code**
  - `retries`: how many times the Producer will attempt to retry sending records
    - Only relevant if `acks` is not `0`
  - Hides transient failure
  - Ensure lost messages are retried rather than just throwing an error
    - `retries`: number times to retry (Default: 0)
    - `retry.backoff.ms`: pause added between retries (Default: 100)
    - For example, `retry.backoff.ms=100` and `retries=600` will retry for for 60 seconds

Preserve Message Send Order

- **If `retries` > 0, message ordering could change**
- **To preserve message order, set `max.in.flight.requests.per.connection=1` (Default: 5)**
  - May impact throughput performance because of lack of request pipelining



Batching and Retries



Performance Tuning Batching

- **Producers can adjust batching configuration parameters**
  - `batch.size` message batch size in bytes (Default: 16KB)
  - `linger.ms` time to wait for messages to batch together (Default: 0, *i.e.*, send immediately)
    - High throughput: large `batch.size` and `linger.ms`, or flush manually
    - Low latency: small `batch.size` and `linger.ms`
  - `buffer.memory` (Default: 32MB)
    - The Producer's buffer for messages to be sent to the cluster
    - Increase if Producers are sending faster than Brokers are acknowledging, to prevent blocking

`send()` and Callbacks (1)

- **`send(record)` is equivalent to `send(record, null)`**
- **Instead, it is possible to supply a `Callback` as the second parameter**
  - This is invoked when the send has been acknowledged
  - It is an Interface with an `onCompletion` method:

```
onCompletion(RecordMetadata metadata, java.lang.Exception exception)
```

- **`Callback` parameters**
  - `metadata` will be `null` if an error occurred
  - `exception` will be `null` if no error occurred

send() and Callbacks (2)

- **Parameters correlated to a particular record can be passed into the Callback's constructor**
- **Example code, with lambda function and closure instead of requiring a separate class definition**

```
producer.send(record, (recordMetadata, e) -> {
  if (e != null) {
    e.printStackTrace();
  } else {
    System.out.println("Message String = " + record.value() + ", Offset = " + recordMetadata.offset());
  }
});
```

Closing the Producer

- **`close()`: blocks until all previously sent requests complete**

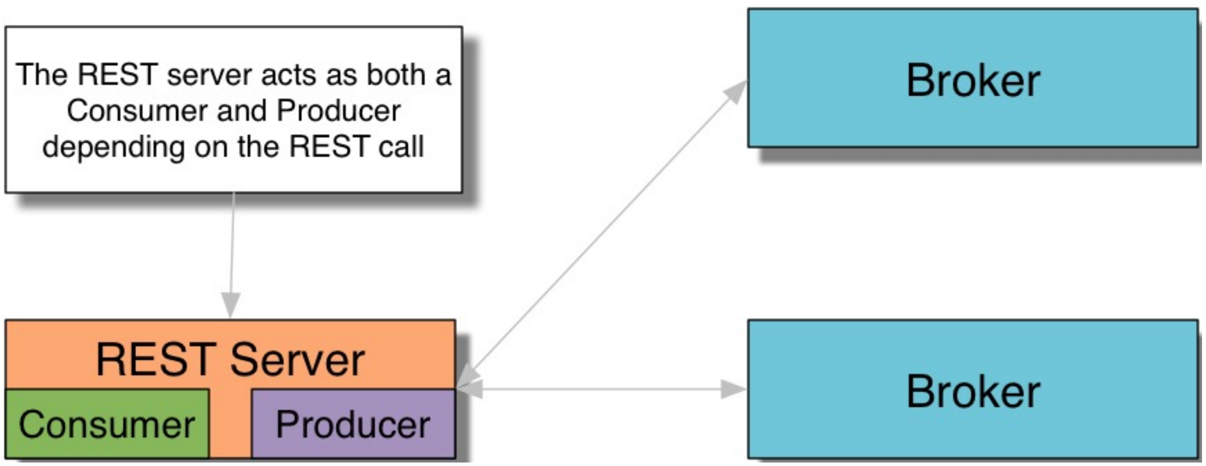
```
producer.close();
```

- **`close(timeout, timeUnit)`: waits up to timeout for the producer to complete the sending of all incomplete requests**
  - If the producer is unable to complete all requests before the timeout expires, this method will fail any unsent and unacknowledged records immediately

Developing With Kafka

About the REST Proxy

- **The REST Proxy allows you to use HTTP to perform actions on the Kafka cluster**
- **The REST calls are translated into native Kafka calls**
- **This allows virtually any language to access Kafka**
- **Uses POST to send data to Kafka**
  - Embedded formats: JSON, base64-encoded JSON, or Avro-encoded JSON
- **Uses GET to retrieve data from Kafka**



A Python Producer Using the REST Proxy

```python
#!/usr/bin/python

import requests
import base64
import json

url = "http://restproxy:8082/topics/my_topic"
headers = {
  "Content-Type" : "application/vnd.kafka.binary.v1+json"
    }
# Create one or more messages
payload = {"records":
  [{
    "key":base64.b64encode("firstkey"),
    "value":base64.b64encode("firstvalue")
  }]}
# Send the message
r = requests.post(url, data=json.dumps(payload), headers=headers)
if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
```

Developing With Kafka

Hands-On Exercise: Writing a Producer

- **In this Hands-On Exercise, you will write a Kafka Producer either in Java or Python**
- **Please refer to the Hands-On Exercise Manual**

Developing With Kafka

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- **Writing a Consumer in Java**
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

Consumers and Offsets

- **Each message in a Partition has an offset**
  - The numerical value indicating where the message is in the log
- **Kafka tracks the Consumer Offset for each partition of a Topic the Consumer (or Consumer Group) has subscribed to**
  - It tracks these values in a special Topic
- **Consumer offsets are committed automatically by default**
  - We will see later how to manually commit offsets if you need to do that
- **Tip: the Consumer Offset is the value of the next message the Consumer will read,** not **the last message that has been read**
  - For example, if the Consumer Offset is 9, this indicates that messages 0 to 8 have already been processed, and that message 9 will be the next one sent to the Consumer

Important Consumer Properties

- **Important Consumer properties include:**

Name	Description
`bootstrap.servers`	List of Broker host/port pairs used to establish the initial connection to the cluster
`key.deserializer`	Class used to deserialize the key. Must implement the **`Deserializer`** interface
`value.deserializer`	Class used to deserialize the value. Must implement the **`Deserializer`** interface
`group.id`	A unique string that identifies the Consumer Group this Consumer belongs to.
`enable.auto.commit`	When set to **`true`** (the default), the Consumer will trigger offset commits based on the value of **`auto.commit.interval.ms`** (default 5000ms)

Creating the **`Properties`** and **`KafkaConsumer`** Objects

```
Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "samplegroup");
props.put(ConsumerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringDeserializer.class);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my_topic", "my_other_topic"));  ① ②
```

① **The Consumer can subscribe to as many Topics as it wishes. Note that this call is not additive; calling `subscribe` again will remove the existing list of Topics, and will only subscribe to those specified in the new call**

② **You may also use regular expressions (*i.e.*, `Pattern`) for topic subscription**

Reading Messages from Kafka with **`poll()`**

```
while (true) { ①
 ConsumerRecords<String, String> records = consumer.poll(100); ②
 for (ConsumerRecord<String, String> record : records)
     System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
}
```

① **Loop forever**

② **Each call to `poll` returns a (possibly empty) list of messages.**

Controlling the Number of Messages Returned

- **By default, `poll()` fetches available messages from multiple partitions across multiple Brokers**
  - Up to the maximum data size *per partition*
    - `max.partition.fetch.bytes`: default value `1048576`
  - A Topic with many partitions could result in extremely large amounts of data being returned
- **The optional timeout parameter in the call to `poll()` is across all fetch requests**
  - This controls the maximum amount of time in milliseconds that the Consumer will block if no new records are available
  - If records are available, it will return immediately
- **In Kafka 0.10.0, `max.poll.records` was introduced**
  - Property limits the total number of records retrieved in a single call to `poll()`

Performance Tuning Consumer Fetch Requests

- **Consumption goal: high throughput or low latency?**
  - High throughput: send more data at a given time
  - Low latency: send immediately when data is available
- **`fetch.min.bytes`, `fetch.max.wait.ms`**
  - `fetch.min.bytes` (Default: 1)
    - Broker waits for messages to accumulate to this size batch before responding
  - `fetch.max.wait.ms` (Default: 500)
    - Broker will not wait longer than this duration before returning a batch
- **High throughput**
  - Large `fetch.min.bytes`, reasonable `fetch.wait.max.ms`
- **Low latency**
  - `fetch.min.bytes=1`

Message Size Limit

- **Try not to change the maximum message size unless it is unavoidable**
  - Kafka is not optimized for very large messages
- **If you must change the maximum size for a batch of messages that the Broker can receive from a Producer**
  - Broker: `message.max.bytes` (Default: 1MB)
  - Topic override: `max.message.bytes` (Default: 1MB)

Preventing Resource Leaks

- **It is good practice to wrap the code in a `try{ }` block, and close the `KafkaConsumer` object in a `finally{ }` block to avoid resource leaks**
- **It is important to note that `KafkaConsumer` is not thread-safe**

```
try {
  while (true) { ①
    ConsumerRecords<String, String> records = consumer.poll(100); ②
    for (ConsumerRecord<String, String> record : records)
      System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
  }
} finally {
  consumer.close();
}
```

Developing With Kafka

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- **Using the REST API to Write a Consumer**
- *Hands-On Exercise: Writing a Basic Consumer*
- *Chapter Review*

A Python Consumer Using the REST API (1)

```
#!/usr/bin/python

import requests
import base64
import json
import sys

# Base URL for interacting with REST server
baseurl = "http://restproxy:8082/consumers/group1" ①

# Create the Consumer instance
print "Creating consumer instance"
payload = {
  "format": "binary"
  }
headers = {
"Content-Type" : "application/vnd.kafka.v1+json"
  }
```

① **We are creating a Consumer instance in a Consumer Group called group1**

A Python Consumer Using the REST API (2)

```
r = requests.post(baseurl, data=json.dumps(payload), headers=headers)

if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
  sys.exit("Error thrown while creating consumer")

# Base URI is used to identify the consumer instance
base_uri = r.json()["base_uri"]
```

A Python Consumer Using the REST API (3)

```
# Get the message(s) from the Consumer
headers = {
  "Accept" : "application/vnd.kafka.binary.v1+json"
    }

# Request messages for the instance on the Topic
r = requests.get(base_uri + "/topics/my_topic", headers=headers, timeout=20)

if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
  sys.exit("Error thrown while getting message")
```

A Python Consumer Using the REST API (4)

```
# Output all messages
for message in r.json():
  if message["key"] is not None:
    print "Message Key:" + base64.b64decode(message["key"])
  print "Message Value:" + base64.b64decode(message["value"])

# When we're done, delete the Consumer
headers = {
  "Accept" : "application/vnd.kafka.v1+json"
    }

r = requests.delete(base_uri, headers=headers)

if r.status_code != 204:
  print "Status Code: " + str(r.status_code)
  print r.text
```

Developing With Kafka

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- **Hands-On Exercise: Writing a Basic Consumer**
- *Chapter Review*

Hands-On Exercise: Writing a Basic Consumer

- **In this Hands-On Exercise, you will write a basic Kafka Consumer**
- **Please refer to the Hands-On Exercise Manual**

Developing With Kafka

- *Programmatically Accessing Kafka*
- *Writing a Producer in Java*
- *Using the REST API to Write a Producer*
- *Hands-On Exercise: Writing a Producer*
- *Writing a Consumer in Java*
- *Using the REST API to Write a Consumer*
- *Hands-On Exercise: Writing a Basic Consumer*
- **Chapter Review**

Chapter Review

- **The Kafka API provides Java clients for Producers and Consumers**
- **Client libraries for other languages are available**
  - Confluent provides and supports client libraries for C/C++, Python, Go, and .NET
- **Confluent's REST Proxy allows other languages to access Kafka without the need for native client libraries**